Contents

This topic introduces **client development** as practiced by DevForce application developers. It highlights the central role of the ***EntityManager*** while also describing entity caching, querying with LINQ, validation, and other model layer activities.

Introduction

DevForce developers spend most of their time building out the **presentation layer** of the client application. The presentation layer is responsible for displaying information to an end-user and responding to end-user input and gestures.

Much of the information displayed and modified derives from persisted data, data housed in remote storage. These data are represented on the client as **entities**, objects with data properties, navigation properties that connect them to other entities, and business logic such as validation, authorization, and workflow rules. These entities and their relationships constitute the **application model**.

The presentation layer relies upon a **model layer** to retrieve, hold, validate, and save entities safely and securely. The model layer consists of model management components, the entity classes, and the application business logic to handle these core tasks.

The model layer is where DevForce contributes most. The presentation layer is largely out of scope, appearing in these discussions primarily as the consumer of the model layer through requests to DevForce components.

This topic identifies the basic elements of the model layer and describes how the presentation layer interacts with the model layer in broad terms. The details are spelled out among the many topics in the "DevForce Development" section.

Entity classes

In a DevForce .NET application, the entity classes on the client are the same as the entity classes on the Application Server. You can learn the details of defining and customizing entity class in the "Model" topic.

These classes are **physically** the same for all .NET clients except Silverlight and mobile clients. These environments each has its own version of .NET and can't make direct use of the full .NET entity classes defined on the server. In these projects you link to the full .NET entity class source code and recompile with reference to the platform-specific libraries. The resulting entities are **source-code identical**, as close to the original as technically possible. When you modify an entity class, your changes are immediately and faithfully reflected on both server and client the moment you recompile.

The EntityManager

The EntityManager is perhaps the most important component in DevForce. It is the client application's gateway to the server and to entity data. It is the primary interface between the presentation and model layers. Through the *EntityManager* API you can:

- query for entities
- find, add, and remove entities from the manager's cache
- save entity changes back to the server
- call methods on the server to perform services that you've defined
- log-in to the server and log-out
- save and restore cached entities in local storage
- operate offline for extended periods, using the entity cache as an in-memory database

All operations with the potential to involve the server can be expressed in synchronous or asynchronous manner, except in Silverlight where only asynchronous communications are allowed. Asynchronous programming can improve perceived responsiveness even for non-Silverlight applications.

A client application always creates at least one *EntityManager*. Some applications create multiple *EntityManagers* in order to isolate units-of-work from each other. For example, a call-center user could juggle several open trouble-ticket screens at the same time, each with its own basket of entities. Saving or discarding changes in one screen won't effect the state of entities displayed on any other screen.

The entity cache

Each *EntityManager* maintains its own **cache of entities**. Every queried entity is placed in cache. You create new entities and add them to cache. You can import entities from other caches. And when you tell the *EntityManager* to save changes, it look in its cache for the entities to save.

An entity is registered in cache by its EntityKey which uniquely identifies it by *type* and primary key value(s). A cache holds at most one instance of an entity with a given *EntityKey*; for example, there can be only one *Company* instance in a particular cache with ID = 42.

In a RIA or smart client application, the *EntityManager* tends to be long-lived; it may hang around for hours, accumulating thousands of entities in its cache from hundreds of queries. Application responsiveness improves over time because many queries can be fulfilled entirely from cached entities without the cost of a trip to the server. On the other hand, some of those queries (and their entities) become stale as other users add and modify the database. Memory consumption may become an issue. Fortunately, features abound for managing the cache and tuning queries to provide the appropriate balance of fast response, fresh data, and small footprint.

The cache can be inspected and changed through dedicated methods of the *EntityManager*. You can take a "snapshot" of the cache at a moment in time in an *EntityCacheState* object. You can snapshot the entire cache or a subset of the entities that interest you. You can save this snapshot to file, import it into another *EntityManager*, even send it to the server and receive one in return.

The flexibility and malleability of the entity cache are critical to applications that can survive dropped connections and run offline for prolonged periods.

Query

There are numerous ways to query entities from the client, all covered in detail in the "Query" topic. The choices include:

- LINQ query
- Query by EntityKey
- Property navigation among entities, e.g., *anOrder.OrderDetails*.
- Entity SQL
- Stored procedure query
- Entity refresh with a call to RefetchEntities

Developers tend to write LINQ queries on the client most of the time. They write queries on the client because that's where the majority of requirements are focused; it's the client UI that changes most frequently throughout the life of an application.

Changes to the UI often involve new queries to retrieve different entities in unexpected ways. One of the guiding principles of DevForce development is that you can create any query you need on the client without touching the server. Testing and redeploying the client executable is faster and less expensive than updating the server installation, especially in web farm and cloud deployments. Such speed and simplicity is critical to meeting deadlines under pressure, with minimal system disruption and without compromising quality and security.

On a DevForce client you can write any LINQ query that is supported by the Entity Framework including sub-queries, aggregations, includes, orderings, groupings, paging, variations on *First()*, and anonymous projections. DevForce knows how to deconstruct, serialize, transmit, and reconstruct that query on the server before forwarding it to the Entity Framework.

You *can* write **server-side named query methods** in DevForce, but you are *not obliged* to write them. Whether you use *named queries* or DevForce-generated queries, you remain in complete control of every query sent to the server thanks to server-side *query interceptors and query filtering*.

The same LINQ queries you send to the server are also applied to the entity cache. The query results retrieved from ("fetched" from) the server are merged with the results from querying the cache. DevForce takes care to preserve any pending unsaved changes. Thus a query for "companies in the western region" could return the existing company, "Alpha", and the newly created "Beta". The soon-to-be-deleted "Gamma" will be excluded as will "Delta" which the user is in the process of relocating to the eastern region.

The *EntityManager* maintains another, different kind of cache called the QueryCache. The *QueryCache* remembers every LINQ query that the server was able to answer. The *EntityManager* compares new query requests with cached queries; if it finds a match, it will fulfill the query from entities in the entity cache rather than waste a trip to the server. The second query for "western region companies" wouldn't hit the server; it would be answered from the cache alone.

You can manipulate or clear the *QueryCache* directly to re-make the *EntityManager*'s recollection of prior queries.

Most developers prefer to determine the query strategies for fetching and merging entities on a per-query basis. For example, if you think it's time to refresh local company data, you can add the *QueryStratagy.DataSourceThenCache* option to that "western region companies" query. The *EntityManager* sends the query request to the server this time and combines the newly retrieved entities with qualifying entities in cache; the query results account for pending changes to "Beta", "Gamma," and "Delta" as they did before.

Running an asynchronous, "data source only" query on a timer can ensure that the entity cache is refreshed on a regular basis.

Creating and modifying entities

You determine how to create a new entity, whether by default constructor, a custom constructor, or a factory method. You must add it to the *EntityManager* before saving it to the database; you typically add it immediately.

A new entity must have a unique EntityKey before it can be added to cache and saved. You are not limited to identity or *Guid* keys. DevForce supports many ways to initialize the key in keeping with the many kinds of keys encountered in real-world applications.

Most applications change new and existing entities by setting their properties. DevForce entities are equipped for RIA and smart client UIs with support for the primary Windows Forms, WPF, and Silverlight data binding interfaces: *INotifyPropertyChanged*, *IEditableObject*, *INotifyDataErrorInfo*, *IDataErrorInfo*, and *INotifyCollectionChanged* - all described in the "Display" topic.

You delete an existing entity in two steps. First you mark it for deletion (by calling *Delete*) whereupon it "disappears" from navigation property collections and all future queries. It remains in cache, waiting to be saved. The second step occurs when you save; only then is it physically removed from the database ... and from the cache.

The details of entity changing operations are covered more thoroughly in the "Create, modify, delete" topic

Validation

Setting a generated data property triggers validation of the input value if the entity is attached to an *EntityManager*. Each *EntityManager* holds a reference to a DevForce *VerifierEngine*. That engine discovers validation rules, some inscribed in the entity classes and others added dynamically at runtime. The engine locates and applies the rules for the property being changed.

You can validate the entire entity instance at any time using the *VerifierEngine*. You are likely to validate entities on the client before asking the *EntityManager* to save them.

Whether you do or you don't, the *EntityServer* will use these same validation rules and a similar validation engine to re-validate entities before saving them to the database.

Validation is covered extensively in its own topic.

Saving changes

Adding, modifying, and deleting only affects entities in cache. They are purely local phenomena and have no effect on the database. Such pending changes are invisible to other application users.

The application calls one of the *EntityManager.SaveChanges* methods to make the changes permanent. While you can save an individual entity or an arbitrary list of entities, we highly recommend the default option which saves all entities with pending changes.

The *EntityManager* raises the *Saving* event before approaching the server with a save request. This is your opportunity to inspect the entities-to-be-saved, validate them, augment them, or perhaps cancel the save altogether.

Once past that hurdle, the *EntityManager* sends this **change-set** to the *EntityServer* in a save request. The *EntityServer* authorizes the save and validates the proposed changes. If all is well, it asks the Entity Framework to save them as a single transaction.

POCO entities follow a similar save path albeit without the automatic Entity Framework integration.

The *EntityServer* notifies the *EntityManager* when it has completed save processing. The *EntityManager* reports errors if the save failed and returns the saved entities if it succeeded.

When the save succeeds, the *EntityManager* merges the saved entities back into cache and marks them unmodified; the merge is necessary as saved entities may have been modified by database triggers. The *EntityManager* also removes the deleted entities from cache. Then it raises the *EntityManager Saved* event so your application can take appropriate action.

These particulars are discussed in much greater detail in the "Save" topic.

Security

Application security is a deep and difficult subject with few absolutes. Applications differ in their security requirements. Technologies differ in the threats they face and the security features they offer. Proper treatment of the subject is beyond the scope of this documentation. What we have to say is covered in the "Security" topic; a brief summary follows.

Any application that exchanges sensitive information over a public network demands special attention. That application should ensure the confidentiality of every such communication at the transport layer with SSL. That's a "must" for every application, not just DevForce applications.

The DevForce EntityServer should be programmed and configured to regard every client request as a potential threat. Users should be authenticated. DevForce supports ASP Authentication out-of-the-box and can accommodate a wide variety of custom authentication schemes. The *EntityServer* provides specific security extension points where you can inspect, alter, and reject every kind of client request including query, save, and custom service method calls. All server errors pass through the *EntityServerErrorInterceptor*, a class that you can customize to filter and modify exception messages with potentially sensitive information.

DevForce helps the developer write client applications that participate in server-side security measures and avoid simple mistakes.

An *EntityManager* must be logged in before it will make a server request. The *EntityManager* can log-in implicitly (e.g., with ASP.NET windows authentication) but only if the server is configured to support it. Otherwise the client application must call the *Login* method to establish an authentication context.

After successful authentication, the *EntityServer* returns to the client a security token containing information about the user. The user information is accessible from the *Principal* property. Client code can adjust the UI and authorize the user activities based on the roles and claims in that *Principal*.

You do not store database connection strings or other data source connection information with a distributed client application (e.g., a Silverlight client). The *EntityManager* does not connect to any data source directly and does not need connection information. No method of the *EntityManager* will accept a connection string. When the *EntityManager* queries or saves data, it refers to the pertinent data source by a symbolic data source key name. The data source key name is an arbitrary string whose value is understood on the *EntityServer* and translated there into an actual connection. The name has no intrinsic meaning or value.