When DevForce executes a standard query, a sequence of steps is followed that involves code executing both on the client where the query was submitted as well as on the EntityServer where, in most cases, a backend datastore resides. This sequence, the **query lifecycle**, can be intercepted by the developer on both the client and the server, via a variety of mechanisms.

A workflow

The table below shows the lifecycle of a query assuming a *QueryStrategy* of *Normal*.

Table 1. Entity Query and Navigation Workflow When QueryStrategy = Normal

Component	Action
Client Tier – Application Code	1) The client application requests a particular set of entities (the "desired entities") either by entity query or by entity navigation
Client Tier – EntityManager	2) The *EntityManager* raises a *Querying* event. Listeners can see the query and, optionally, cancel the query.
	3) The *EntityManager* checks the *QueryCache* (and sometimes the *EntityCache* depending on the kind of query) to see if it can satisfy the query with the entities in the client-side cache. If so, it runs the query against the local cache only and jumps to step 14 below.
	4) The *EntityManager* raises a *Fetching* event. Listeners can see the query and, optionally, cancel the query.
	5) The *EntityManager* sends the query along with authentication information to the *EntityServer* on the middle tier. It may modify the request before sending it to the EntityServer if it can determine that some of desired entities are already in the client side cache.
Middle Tier – EntityServer	6) The *EntityServer* authenticates the client (the currently logged in "user") and creates a new instance of either a custom or a default **EntityServerQueryInterceptor**. The *EntityServerQueryInterceptor* is executed and it can either modify the query and/or run any developer-specified security checks in the *AuthorizeQuery* handler. If security checks fail, it raises a security exception and sends this back to the client tier.
	7) Having passed security checks, the *EntityServerQueryInterceptor* calls its base *ExecuteQuery* method where the EntityServer converts the, possibly modified, query into a format applicable to the datasource being queried. This will often be an Entity Framework LINQ to Entities query.
Data source – Data Source	8) The data source performs the query or queries and returns one or more result sets back to the *EntityServer*.
Middle Tier – EntityServer	9) The Entity Framework converts the result sets returned from the data source into ADO.NET entities and delivers them to the *EntityServerQueryInterceptor* (from above) and performs any post query authorization via the *AuthorizeQueryResult* method.
	10) The *EntityServer* repackages the result set into a format that can be transmitted efficiently. It then ships the entity data back to the client side *EntityManager*.
	11) After transmission, the *EntityServer* allows the server's local copy of the entities to go out of scope and the garbage collector reclaims them. This enables the *EntityServer* to stay stateless.
Client Tier –EntityManager	12) Compares fetched entities to entities already in the cache. Adds new entities to the cache. Replaces matching cached entities that are unmodified (in essence refreshing them). Preserves cached entities with pending modifications because the query strategy is normal.
	13) Reapplies the original query to the cache to locate all desired entities.
	14) Raises the *Queried* event. Listeners can examine the list of entities actually retrieved from the data source.
	15) Returns the desired entities to the application.
Client Tier – Application Code	16) The entities are available for processing.