Contents

Why **query asynchronously**? You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming.

Also see the "program asynchronously" topic for addiitonal information.

Async/await

With the new task-based asynchronous programming model, executing asynchronous queries has never been easier. An asynchronous query and result handling can now be written similarly to the familiar synchronous programming model.

Here's a simple example. We take a query, in this case for customers from the "UK", call *ExecuteQueryAsync* and **await** the return of the results. The *await* keyword tells the compiler to suspend further execution of this method and resume when the asynchronous method completes.

```
public async void SomeMethod() {
  var query = manager.Customers.Where(c=> c.Country == "UK");
  var customers = await manager.ExecuteQueryAsync(query);
  doSomething(customers);
}
```

```
Public Async Sub SomeMethod()
   Dim query = From c In manager.Customers Where c.Country = "UK"
   Dim customers = Await manager.ExecuteQueryAsync(query)
    doSomething(customers)
End Sub
```

It's that simple. We added the async (or Async in Visual Basic) modifier to indicate that the method contains asynchronous code, and the *await* (or *Await*) keyword to indicate that further processing in the method should be suspended until the asynchronous task completes. In the snippet above, the *doSomething* method will be called when the asynchronous query completes.

Task results

Like its synchronous counterpart, asynchronous query execution comes in both generic and non-generic flavors:

On the *EntityManager*:

- Task<IEnumerable<T>> ExecuteQueryAsync<T>(IEntityQuery<T> query)
- Task<IEnumerable> ExecuteQueryAsync(IEntityQuery query)

As query extensions:

- Task<IEnumerable<T>> ExecuteAsync<T>(this IEntityQuery<T> query)
- Task<IEnumerable> ExecuteAsync(this IEntityQuery query)

These async query methods return a Task<TResult>, where *TResult* will be the IEnumerable or IEnumerable<T> of returned objects. In the earlier example retrieving customers from the UK, the return results are an IEnumerable<Customer>.

The task represents the asynchronous operation, and will indicate the status of the operation, the results of a completed operation, and whether the operation was cancelled or failed.

Note that the task returned from a DevForce async method is "hot": it has already started and is scheduled for execution.

Async scalar queries

A scalar immediate execution query is a LINQ query which performs an aggregation (such as Count or Group) or returns only one element (such as First or Single). Because these methods force immediate execution of the query they can't be directly used with asynchronous queries, but using the **AsScalarAsync** method you can execute scalar immediate execution queries asynchronously. We cover these queries in detail in a separate topic.

Cancelling a query

There are a number of ways to cancel an asynchronous query.

1. In a *Querying* event handler
2. In a *Fetching* event handler
3. In an *EntityServerQueryInterceptor*
4. With a *CancellationToken*

The first three options all work the same whether the query is synchronous or asynchronous. The last option, the CancellationToken, is unique to asynchronous tasks. To cancel the task for an asynchronous query, provide a *CancellationToken* in the method call:

- Task<IEnumerable<T>> ExecuteQueryAsync<T>(IEntityQuery<T> query, CancellationToken cancellationToken)
- Task<IEnumerable> ExecuteQueryAsync(IEntityQuery query, CancellationToken cancellationToken)
- Task<IEnumerable<T>> ExecuteAsync<T>(this IEntityQuery<T> query, CancellationToken cancellationToken)
- Task<IEnumerable> ExecuteAsync(this IEntityQuery query, CancellationToken cancellationToken)

The *CancellationToken* is a cancellation request. DevForce will attempt to honor the request and cancel the async task, but the request may arrive too late in the query lifecycle.

You'll generally use a *CancellationToken* when you wish to cancel an async query which is taking too long, or you have multiple async tasks you wish to cancel at one time with the same *CancellationToken*.

Here's a simple example:

```
public async void TryQuery() {
  var manager = new DomainModelEntityManager(false);
  var cts = new CancellationTokenSource();
  cts.CancelAfter(2000);
 try {
    var customers = await manager.ExecuteQueryAsync(manager.Customers, cts.Token);
  } catch (OperationCanceledException oce) {
    MessageBox.Show("The query was cancelled after 2 seconds.");
  } catch (EntityServerConnectionException esce) {
    MessageBox.Show("The query failed.");
  }
}
```

The *Task* will be cancelled regardless of which of the query cancellation options you use to cancel an asynchronous query. If you await the *Task*, an *OperationCanceledException* will be thrown.

Error handling

An *awaited* task will throw an exception if it's either faulted or cancelled. This is why you should wrap any await calls in a try/catch.

Query execution exceptions are passed to the *EntityManager's EntityServerError* handler if one is defined. If you do mark the error as handled the exception will not be rethrown.

Try and the QueryResult

The async query methods returning an *IEnumerable* or *IEnumerable<T>* will all raise an exception if cancelled or an error occurs. But in some situations you might instead prefer to "try" to execute the query, and always return a "query result" which provides query execution status and results. For these situations, you can use the *TryExecuteQueryAsync* methods on the *EntityManager*:

- Task<QueryResult> TryExecuteQueryAsync(IEntityQuery query, CancellationToken cancellationToken)
- Task<QueryResult<T>> TryExecuteQueryAsync<T>(IEntityQuery<T> query, CancellationToken cancellationToken)

The *QueryResult*:

Member	Summary
Cancelled	Whether the query was cancelled by any means.
ChangedEntities	All entitites retrieved as part of the fetch.
Error	The exception if an unhandled error was raised.

Query	The query executed.
ResolvedFetchedStrategy	The actual *FetchStrategy* used.
Results	The results of the query.
UntypedQuery	The *IEntityQuery*.
WasFetched	Whether the data was fetched from the *EntityServer*.