Working with XML queries and the query manager

This page probably does not do much for you

The *query manager* has been with re-motion from its very beginning, and it works very well. However, the recent introduction of re-linq technology into re-motion (see re-linq made the query manager obsolete for almost all queries. For some rare cases, re-linq (and Linq in general) has its limits, and you might wish to get closer to the SQL-metal in your project. Use the query manager when

- re-linq generates slow queries
- you want to use database features not covered by re-linq (rare)
- your queries can't be expressed in (re-) Linq, because they are too complex

In other words, this paper on the query manager won't be interesting for most re-motion programmers. *This paper is a fringe-programme for experts*.

In the following sections we will re-examine the `GetLocations` and `FindPersonsByLocation` methods from the re-store chapter in the PhoneBook tutorial (FIXME). The author assumes that the reader is familiar with re-store already. Note that these simple methods are for illustration only. In practice it is unnecessary work to use the query manager for queries as simple as `GetLocations` or `FindPersonsByLocation`, because for those, re-linq will work just fine.

Queries.xml

Queries intended for the query manager are stored in an XML "configuration" file, typically named `queries.xml` – a regular staple when programming databases on Microsoft Windows. The `queries.xml` file gives SQL queries an ID for identifying them from program code, and specifies the query's parameter(s), if any. Of course you can name the query manager configuration file any way you want, but "queries.xml" is the canonical name. You find a complete `queries.xml` for the exercises here at FIXME.

GetLocations revisited

In the PhoneBook tutorial, the following static method for retrieving all Location objects from the database is introduced. The method is based on Linq, of course:

```
public static Location[] GetLocations ()
{
  var query = from l in QueryFactory.CreateLinqQuery<Location> ()
                  select l;
  return query.ToArray ();
}
```

The fun and easy way, the modern way! As old-schoolers at rubicon remember, the query man way of implementing this simple method is slightly more cumbersome. And less modern.
It requires some preparatory work. You declare the actual SQL statement in the `queries.xml` file and place it where both your library and your client application can find it at run-time. Here is the `queries.xml` listing for the query manager version of `GetLocations`, called `QueryManGetLocations`:

```
<?xml version="1.0" encoding="utf-8" ?>
<queries xmlns="http://www.re-motion.org/Data/DomainObjects/Queries/1.0">

  <query id="QueryManGetLocations" type="collection">
    <statement> SELECT * FROM LocationView </statement>
  </query>
</queries>
```

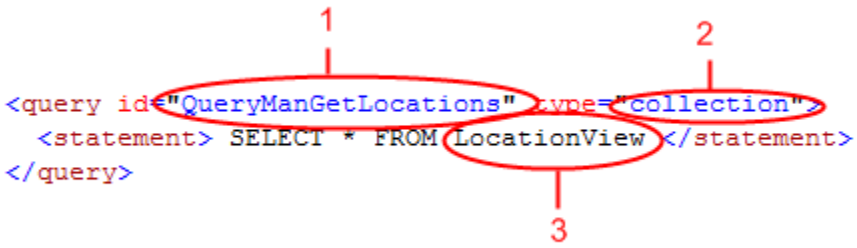
You can copy this basic `queries.xml` file from the location

```
PhoneBook.Domain\queries.xml in the PhoneBook sample (FIXME)
```

Of course there is an API for constructing query objects entirely with C# code, but this is usually more work.

Understanding query nodes

The most important item is the node `<query id="GetLocations" type="collection">`.
This snippet of uncommented code requires some discussion, so there you go with a diagram and a set of keys to it:



1. The id `QueryManGetLocations` simply is a handle for your code to identify the query. By convention (and good practice), this id should have the same name as the method that uses the query.
2. The type "`collection`" signals that the query returns more than one object, potentially none at all (the alternative is "`scalar`").
3. `LocationView` is what programmers can use for actually accessing object instances for reading. As the name suggests, LocationView is a... wait for it... database view. So if you write a query that only reads data, use "X-View" instead "X-", where "X-" is the domain object class. (Views abstract away the particular type of table inheritance used.)
   Declaring the query in `queries.xml` is not enough, of course. You must wrap it up in a method as well. For the given query you do that by

- creating a new query object from that <query> node in queries.xml
- passing that query object to the query manager
- asking the query manager to execute the query and give you the instantiated domain object(s)

Just as with Linq, you let re-store's query factory create the query for you and let that query retrieve the data. In contrast to the Linq-query, the use of the current transaction for retrieved data is not invisible. When using the query manager, you explicitly work with `ClientTransaction.Current`. You ask the current transaction itself to load the desired data.
Here is the listing for the `QueryManGetLocations` method for the Locations class, wrapping the query:

```
    // In class Location
    public static Location[] QueryManGetLocations ()
    {
      var query = QueryFactory
                  .CreateQueryFromConfiguration ("QueryManGetLocations");
      return ClientTransaction
             .Current
             .QueryManager
             .GetCollection<Location> (query).ToArray ();
    }
```

This won't compile, because the `Query` type is not known to `Location.cs` yet. Supplement

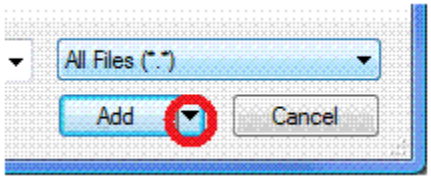
```
using Remotion.Data.DomainObjects.Queries;
```

to fix this.

As you can see in the listing, we pass the string `QueryManGetLocations` as an ID and as a parameter to the `Query` constructor, so that it can locate the node with the query in `queries.xml` (Point 1. in the annotated illustration above). What should also be appearant from this listing is that the query manager is part of the current client transaction. We ask you to defer deeper understanding to later and treat this as a recipe for the time being.

Please note that this code does NOT establish a client transaction. Just like its Linq-programmed relative, `QueryManGetLocations` does assume that there is a valid client transaction (`ClientTransaction.Current...`), but setting up this client transaction is the responsibility of the user of this function.
In order to make this code work for your application, add the static `QueryManGetLocations` method above to your `Location` class and add a `queries.xml` file to your PhoneBook.Domain project. Then type (or copy) the code from the sample `queries.xml` listed above into that new queries.xml file in your project.
However, you must make it known to your `PhoneBook.Sample` sub-project, too. Here is a brief how-to:

- Right-click your `PhoneBook.Sample` project and go along *Add->Existing item...*
- The file picker dialog opens, but it doesn't show you any XML files, because the filter is set to **`.cs`, etc. Use the drop down list to select `.xml`**.
- Select the `queries.xml` file from the `PhoneBook.Domain` peer directory, but **DON'T click Add**.
- Instead, click on the small arrow on the *Add*-button:



- Select *Add as link*. `queries.xml` shows up in the `PhoneBook.Sample` project.
- Right-click this new link that looks like the real queries.xml thing and select "properties".
- The ***Copy to Output Directory*** property contains a drop list. Select *copy always* or *copy if newer*. You are done.

At this point, you are ready to try out your query. Do the obvious thing and code your `Program`'s `Main()` like this:

```
static void Main(string[] args)
{
  // Sigmund Freud and the royals from the PhoneBook:
  // EnterFreud();
  // EnterHabsburgs();

  using (ClientTransaction
         .CreateRootTransaction ()
         .EnterDiscardingScope ())
  {
    foreach (var loc in Location.QueryManGetLocations ())
    {
      Console.WriteLine ("{0} {1} {2}",
                         loc.Street,
                         loc.Number,
                         loc.City);
    }
  }

}
```

As you can see, we must set up the client transaction for `Location.QueryManGetLocations()`, since `Location.QueryManGetLocations ()` does not bring its own.

`FindPersonsByLocation` revisited – query methods with parameters

`Location.QueryManGetLocations ()` does not require a parameter, but what about the query-wrapper to get to all the people living at a given location? This is how you invoke the query-wrapper:

```
myLocation.QueryManFindPersonsByLocation(locationID)
```

The query behind the `GetPersons`-method is easy to do in the database, just use

```
SELECT * FROM Person WHERE LocationID = <some Location's GUID>;
```

If your database still contains the emperor and the empress and their palace location, you can easily try this out yourself. Run

```
SELECT * FROM Location WHERE Street = 'Schönbrunner Schloßstraße';
```

(Or use `LIKE '%brunner%'` if umlauts scare you.)

This will give you a row for the "Schönbrunner Schloßstraße" location. Its first column is an ID with the GUID. Copy that GUID, and use it in the SELECT for finding the royal couple:

```
SELECT * FROM Person WHERE LocationID = '<copied guid here>';
```

This should give you two rows for the emperor and the empress. (Note that `dbschema.exe` has rewritten the `Person` property `Location` as `LocationID` in the database. This is the default behavior of `dbschema.exe`: if a property is a reference to another domain object, the name is modified in this fashion.)

The Location's GUID is what must be passed to query in a wrapper like

```
myLocation.GetPersons()
Consequently, the XML-declaration for the query uses the @Location
parameter:
  <query id="QueryManFindPersonsByLocation" type="collection">
    <statement> SELECT * FROM PersonView WHERE LocationID = @location
    </statement>
  </query>
The wrapper uses the "GetPersons" query identifier to locate the node in
queries.xml. It uses the @Location place holder for putting the parameter,
the Location's ID there:
    public static Person[] QueryManFindPersonsByLocation (Location
location)
    {
      var query = QueryFactory
                  .CreateQueryFromConfiguration
                          ("QueryManFindPersonsByLocation");
      query.Parameters.Add ("@location", location.ID.Value);
      return ClientTransaction
             .Current.QueryManager
             .GetCollection<Person> (query).ToArray ();
    }
```

The most interesting part in this listing is probably the call to

```
query.Parameters.Add(),
```

because this is the spot where the `Location` object's ID is actually passed to the query.

After adding both the `QueryManFindPersonsByLocation` XML snippet to `queries.xml` and the `QueryManFindPersonsByLocation` C# snippet to `Location`, you are ready to try out this improvement.
The following updated version of the `Main()` method lists all locations, persons and phone-numbers:

```
static void Main(string[] args)
{
  EnterHabsburgs();
  foreach(Location loc in Location.QueryManGetLocations())
  {
    Console.WriteLine (loc.Street);
    foreach(Person p in loc.QueryManFindPersonsByLocation())
    {
      Console.WriteLine ("   {0} {1}", p.FirstName, p.Surname);
      foreach (PhoneNumber phone in p.PhoneNumbers)
      {
        Console.WriteLine ("    +{0} ({1}) {2}/{3}",
                           phone.CountryCode,
                           phone.AreaCode,
                           phone.Number,
                           phone.Extension);
      }
    }
  }

  Console.ReadLine();
```

Sample Code

The code presented here is part of the PhoneBook sample. The corresponding methods are parked in partial extensions to the `Location`, `Person` and `Program` class, respectively:

- QueryManLocation.cs
- QueryManPerson.cs
- QueryManProgram.cs