NetQuarry, Inc.

Training

400 - Coding

Generated Code

The platform provides a mechanism to convert application meta data into application code. The generated code consists of Typed Mapper objects, Session properties and Picklists. The code generation allows you to refer to meta data in code as objects.

How to safely generate, generated code

The generated code files are located in the common project. This is the top level project that allows the Data object and extensions all refer to the generated code. The generated code is all created from meta data but it's not always necessary to regenerate the code whenever you make meta data changes. However there are times when regenerating the code is mandatory.

- Added/Modified Session property
- Added field in mapper you want to refer to in extension
- Changed the data type of a mapper field
- Added a picklist you want to refer to in code.

When you do need to regenerate the code (by running the gen-code.bat file in Database\Meta) you should follow this procedure.

- Make sure your meta data (for modules you haven't modified) is completely up to date.
- Check out the three generated files, PickListEnums.cs, Session.cs, TypedMappers.cs
- Run the batch file gen-code.bat file
- Verify the code generation ran without any errors. The most common code generation error is with trying to use the underlying view for a mapper and the view is invalid or missing.
- Fix any problems
- Rebuild the entire code base
- Verify the build succeeds. After regenerating typed mappers it's possible to introduce build problems. The most common reason for introducing build issues are…
    - Fields have been deleted from mappers
    - Fields have been flavored with an include flavor (so not included on mappers with 0 flavor)
    - Data types have been modified on fields

    - Fields have been added that use .Net reserved names as key names

- If there are any problems with the build, you must fix the meta data. If the problem is related to a field having a name same as a .Net reserved word, there is a field property on a field called “PropertyName” that you can set and that name will be used for the generated property.
- After you’ve fixed the meta data, you must re run the batch file to regenerate.
- And then rebuild to confirm your changes have fixed all the problems
- Check in any meta data changes (identify CodeGen fixes as such)
- Check in the generated files.
- Never manually edit the generated files.

Mapper Extensions

The NetQuarry tutorial goes into excellent detail on how to create extensions, but to summarize, there are two types of extensions. Those that are derived from a basic mapper object and those derived from a TypedMapper object.

Generic Extension

```
public class MyExtension : NetQuarry.Data.MapperExtensionKernel
```

TypedMapper Extension

```
public class MyTMExtension :
NetQuarry.Data.TypedMapperExtension<Comensura.Data.MyTM>
```

or, if you have a base class that has additional common handling for all typed mapper extensions, that is derived itself from `NetQuarry.Data.TypedMapperExtension`

```
public class MyTMExtension : Comensura.Extensions.TypedExtensionBase<Comensura.Data.
MyTM>
```

Use Generic or TypedMapper Extension?

If you need to create an extension, you are faced with the choice of creating a generic extension, or typed mapper extension. The biggest question you need to ask yourself is whether the extension needs to be attached to several mappers, either of the same module, or even disparate modules. This is the situation with the ExportToExcel extension provided by the core. It adds an Export To Excel menu item to any mapper it is attached to, and manages the event when the Export To Excel is clicked.

A secondary reason for creating just a generic mapper extension is if you don't have a typed mapper generated for the mapper your want business logic for and/or it's just not worth the effort.

When you know you want to add some more than trivial logic to your extension, then it's worth creating the typed mapper extension.

Mapper Extension Events

The following table describes the events that can be handled from a mapper extension.

Event	Description
AuditDelete	The mapper has performed a delete operation and any auditing should be performed in response to this event.
AuditInsert	The mapper has performed an insert operation and any auditing should be performed in response to this event.
AuditUpdate	The mapper has performed an update operation and any auditing should be performed in response to this event.
Custom	An event in the range of ExtensionEvents.CustomStart and ExtensionEvents.CustomEnd has been fired.
FieldBuildFilter	The field is attempting to build a filter from the criteria entered in the filter by form row of a list, or from the find criteria of a Fin/MultiAdd screen.
FieldButtonClick	The field's button has been clicked. A button click event is supported in both the detail and datasheet versions of the mapper.
MailBeforeSend	Generated by the template mailer component just before the mailer constructs the email to send.
MailAfterSend	Generated by the template mailer component just after the mailer has sent an email to the recipient.
MapperAfterLayout	The mapper has just layed out the mapper's UI.
MapperAfterLoad	The mapper's meta-data has been loaded, fields created, etc, but no field controls created. This is typically the only opportunity to alter field CellTypes programmatically.
MapperAfterRequery	The mapper has just requeried its operational data.
MapperBeforeLayout	The mapper is just about to begin laying out the mapper's UI.
MapperBeforeRequery	The mapper is just about to requery its operational data.
MapperBulkAfterDelete	A bulk delete operation, using this mapper, has been completed.
MapperBulkAfterInsert	A bulk insert operation, using this mapper, has been completed.

Event	Description
MapperBulkAfterUpdate	A bulk update operation, using this mapper, has been completed.
MapperBulkBeforeDelete	A bulk delete operation, using this mapper, is being started.
MapperBulkBeforeInsert	A bulk insert operation, using this mapper, is being started.
MapperBulkBeforeUpdate	A bulk update operation, using this mapper, is being started.
MapperCommand	A mapper command has been invoked.
MapperExecSQL	The mapper is about to execute a SQL statement. The SQL to be executed is contained in the ExecSQLArgs parameter. Any change made to the SQL in that parameter cause the altered SQL to be used instead of the mapper-generated SQL.
OnUnload	Notifies the extension that it should perform any cleanup necessary just prior to being unloaded. (inherited from MapperExtensionBase)
Other	Handles an un-recognized event.
RowAfterDelete	The mapper has just completed deleting an entire row into the database.
RowAfterInsert	The mapper has just completed inserting an entire row into the database.
RowAfterUpdate	The mapper has just completed updating an entire row into the database.
RowBeforeDelete	The mapper is just about to delete an entire row into the database.
RowBeforeInsert	The mapper is just about to insert an entire row into the database.
RowBeforeUpdate	The mapper is just about to update an entire row into the database.
RowCurrent	The mapper has just positioned to a different record.
RowExecSQL	The mapper is just about to execute a single SQL statement that is row-specific (e.g. executing an INSERT, UPDATE or DELETE statement).
RowSetDefaults	The mapper has had its values populated from defaults. You now have the opportunity to modify the default values on the mapper fields. This event exists to consolidate all the different mapper events where default values have been set.

By far the most commonly handled events are Before/After Insert/Update, MapperBeforeRequery, MapperBeforeLayout.

Event Arguments

All mapper events are fired with an EAPEventArgs parameter. This section describes what event args are available and whether the args are specific to certain events only.

Event Argument Details

Args Type	When Used
EAPEventArgs	Base class for all event args. Passed for all other events, not listed here.
BuildFilterArgs	Passed to event when handling the FieldBuildFilter event.
EAPCommandEventArgs	Passed to event when handling the MapperCommand event.
EAPCustomEventArgs	Passed to event when handling Custom event.
ExecSQLArgs	Passed to event when handling the MapperExecSQL and RowExecSQL events.
MailEventArgs	Passed to event when handling MailBeforeSend, MailAfterSend events.

EAPEventArgs

EAPEventArgs Property/Method	**Description**
BulkContext	A property of type NetQuarry.EventBulkContext. This property is set whenever a bulk operation is performed. Typically when performing multiple updates through editable list, or multiple deletes from list view, or multiple inserts, through mult-add process. You can tag your own properties onto the BulkContext object to provide a way of maintaining contextual information between insert/update/delete events. If this object is null, there is no bulk operation in progress
Cancel	A method that takes a string argument. The string is a message that is presented to the user after the event has been cancelled. Calling this function also sets the Result property to ExtResult.Cancel.
Error	A method that takes a string argument. When called it sets the Result property to ExtResult.Error. when the event is completed, the error message is presented to the user.
ErrorMessage	A property of type string that lets you get/set the error message. It does not set the Result to ExtResult.Error.
Event	Contains an enum value of type NetQuarry.ExtensionEvents that specifies which event is being fired.
Result	A property of type NetQuarry.ExtResult, defining what status to return to the platform. By default, the value is ExtResult.Continue.

ExtResult value	**Description**
Continue	The default result status. If you do nothing, execution will continue as normal processing additional extension code and platform code.
DataChanged	When handling the RowSetDefault event, you specify this result to let the platform know to re-apply the field values from default values.
ContinueIgnore	Continue as if the cause of the event had not occurred. This is generally returned in response to events which indicate an anomaly
ContinueNoMoreExt	Your extension may perform a task where you cannot allow any more extensions to fire and perform additional work. Your extension is the only one that should fire for a particular event. If this is the case you return this result to prevent any more extensions being called for this event. For this to work correctly, your extension must be a higher priority than the extensions you don't want the event to be handled
ContinueNoExec	Return this status when you don't want the platform to execute the default operation, but allow the event to continue.
CancelWhenDone	Return this status when you don't want the operation to complete, but you want all the extensions to fire
HandledByExt	Return this status when your extension has performed a function in place of the default action about to be executed by the platform. For example RowBeforeInsert. Returning this status, the platform will not execute any insert statements. The after events will continue to fire as normal as though the platform performed the operation. This is similar to ConinueNoExec.
Cancel	Cancel the operation immediately and do not fire the event to additional extensions.
Error	An error occurred and the operation should be aborted.

BuildFilterArgs

BuildFilterArgs Property/Method	**Description**
Description	A string property containing a description of the filter clause that has been applied. This overrides the default description from the platform and should match the context of the new filter being applied. This description will appear in the filter caption of the list view. Keep this description short.
Dirty	A boolean property determining whether the filter has been modified. This property is automatically set to true if you assign a new Filter. If you set this to true, without setting a Filter property, then filtering for the field will be ignored.
Filter	A string property where you set the filter to apply. Typically you will analyze the current filter criteria and construct a different filter criteria to apply instead of the default criteria generated by the platform.
FilterFlags	A read only property of type NetQuarry.FieldFilterFlags. This gives you contextual information about the filter parameters you need to analyze to construct a new filter string.

BuildFilterArgs Property/Method	Description
ParsedOperand	A ready only string property that contains the filter operand extracted from the raw criteria (e.g. ::startswith::) The operands you might expect to see are ::or::, ::and::, ::not::, ::between::, ::contains::, ::startswith::, ::doesnotcontain::, ::soundslike::, ::null::, ::notnull::, ::doesnotstart::, ::like::, ::notlike::
ParsedValue	The parsed value from the raw criteria entered into the filter by form row.
RawCriteria	The value as it was entered into the filter by form row (or the criteria field on a Find/Myulti-Add)

EAPCommandEventArgs

EAPCommandEventArgs Property/Method	Description
CommandName	A string property that contains the name of the command.
Params	A property of type System.Collections.Specialized.NameValeCollection. This contains the querystring parameters in a NameValueCollection. Currently this property is only populated whan handling Ajax button clicks.

EAPCustomEventArgs

EAPCustomEventArgs Property/Method	Description
Args	A read only property of type Object to which you can attach your own properties. Since you are likely to be generating custom events, you will know what is contained within the args parameter.

ExecSQLArgs

ExecSQLArgs Property/Method	Description
Dirty	A Boolean property indicating whether the SQL in the arguments has been changed by the extension. This property is automatically set to true when the SQL property is updated.
SQL	A string property that contains the SQL that is about to be executed. You can replace the SQL with your own and that will be executed instead.
StatementType	A read only enum property of type NetQuarry.Data.ExecSQLArgs.ExecuteStatement. This indicates what type of SQL is about to be executed. The enum values are… Unknown, Select, Insert, Update, Delete, LogicalDelete
TableName	A read only string property indicating the name of the table on which the SQL will be executed. For a SELECT statement, this will be the view name.

MailEventArgs

MailEventArgs Property/Method	Description
MailerAttributes	A property of type NetQuarry.Services.MailerAttrs. These attrs only contain one enum value, Journal.
Message	A property of type System.Net.Mai.MailMessage. This property gives you access to the mail message that has been constructed by the template mailer and is about to be sent by the platform. You can interrogate the mail object to extract any pertinent information and journal it (e.g a fully rendered html email body)

Event Handling

This section summarizes some of the actions you may take and want to handle when performing certain actions.

<table>
<tr><th>Action</th><th>Event Fired</th></tr>
<tr><td>Open a mapper in list view</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>RowCurrent</td><td>Only if records exist and one for every row</td></tr>
</table>
</td></tr>
<tr><td>Open a mapper in a detail</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
</table>
</td></tr>
<tr><td>Open a new detail</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>RowSetDefaults</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
</table>
</td></tr>
<tr><td>Save a new record (Popup window back to list)</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>RowSetDefaults</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
<tr><td>RowBeforeInsert</td><td></td></tr>
<tr><td>RowExecSQL</td><td></td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>AuditInsert</td><td></td></tr>
<tr><td>RowAfterinsert</td><td></td></tr>
<tr><td>RowSetDefaults</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
<tr><td>MapperAfterLoad</td><td>Display list after popup disappears</td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Action</th><th>Event Fired</th></tr>
<tr><td>Save a new record (Detail window back to detail)</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>RowSetDefaults</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
<tr><td>RowBeforeInsert</td><td></td></tr>
<tr><td>RowExecSQL</td><td></td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>AuditInsert</td><td></td></tr>
<tr><td>RowAfterinsert</td><td></td></tr>
<tr><td>RowSetDefaults</td><td></td></tr>
<tr><td>RowCurrent</td><td>End of Save Process</td></tr>
<tr><td>MapperAfterLoad</td><td>Re-display detail after save on DIFFERENT mapper</td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
</table>
</td></tr>
<tr><td>Save an existing record (Detail to Detail)</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
<tr><td>RowBeforeUpdate</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>RowExecSQL</td><td></td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td></td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>AuditUpdate</td><td></td></tr>
<tr><td>RowAfterUpdate</td><td>End of save</td></tr>
<tr><td>FieldBuildFilter</td><td>Re-display detail after save on DIFFERENT mapper</td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
</table>
</td></tr>
</table>

Action: Update multiple rows through editable list

Event Fired

Event Order	Notes
MapperAfterLoad	
FieldBuildFilter	
MapperBeforeLayout	
MapperAfterLayout	
MapperAfterLoad	Cloned Mapper
MapperBulkBeforeUpdate	Cloned Mapper
FieldBuildFilter	Cloned Mapper, Repeated for each updated row
MapperBeforeRequery	Cloned Mapper, Repeated for each updated row
MapperExecSQL	Cloned Mapper, Repeated for each updated row
MapperAfterRequery	Cloned Mapper, Repeated for each updated row
RowCurrent	Cloned Mapper, Repeated for each updated row
RowBeforeUpdate	Cloned Mapper, Repeated for each updated row
FieldBuildFilter	Cloned Mapper, Repeated for each updated row
RowExecSQL	Cloned Mapper, Repeated for each updated row
MapperBeforeRequery	Cloned Mapper, Repeated for each updated row
FieldBuildFilter	Cloned Mapper, Repeated for each updated row
MapperExecSQL	Cloned Mapper, Repeated for each updated row
MapperAfterRequery	Cloned Mapper, Repeated for each updated row
AuditUpdate	Cloned Mapper, Repeated for each updated row
RowAfterUpdate	Cloned Mapper, Repeated for each updated row
MapperBulkAfterUpdate	Cloned Mapper
MapperVeforeRequery	
MapperExecSQL	
MapperAfterRequery	
MapperBeforeRequery	
MapperExecSQL	
MapperAfterRequery	
RowCurrent	

Action: Delete multiple rows on list view

Event Fired

Event Order	Notes
MapperAfterLoad	
FieldBuildFilter	
MapperBeforeLayout	
MapperAfterLayout	
MapperAfterLoad	Cloned Mapper
MapperBulkBeforeDelete	Cloned Mapper
FieldBuildFilter	Cloned Mapper, Repeated for each deleted row
MapperBeforeRequery	Cloned Mapper, Repeated for each deleted row
MapperExecSQL	Cloned Mapper, Repeated for each deleted row
MapperAfterRequery	Cloned Mapper, Repeated for each deleted row
RowCurrent	Cloned Mapper, Repeated for each deleted row
RowBeforeDelete	Cloned Mapper, Repeated for each deleted row
FieldBuildFilter	Cloned Mapper, Repeated for each deleted row
RowExecSQL	Cloned Mapper, Repeated for each deleted row
AuditDelete	Cloned Mapper, Repeated for each deleted row
RowAfterDelete	Cloned Mapper, Repeated for each deleted row
MapperBulkAfterDelete	Cloned Mapper
MapperBeforerequery	
MapperExecSQL	
MapperAfterRequery	
RowCurrent	

<table>
<tr><th>Action</th><th>Event Fired</th></tr>
<tr><td>Execute an Action Menu Item resulting in a command on a detail</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperBeforeRequery</td><td></td></tr>
<tr><td>MapperExecSQL</td><td></td></tr>
<tr><td>MapperAfterRequery</td><td></td></tr>
<tr><td>RowCurrent</td><td></td></tr>
<tr><td>MapperCommand</td><td></td></tr>
</table>
</td></tr>
<tr><td>Execute an Action Menu Item resulting in a command on a list view</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>FieldBuildFilter</td><td>One for every field with filter criteria</td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>MapperCommand</td><td></td></tr>
</table>
</td></tr>
<tr><td>Click on a button/link on a detail or list view</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>MapperAfterLoad</td><td></td></tr>
<tr><td>MapperBeforeLayout</td><td></td></tr>
<tr><td>MapperAfterLayout</td><td></td></tr>
<tr><td>MapperAfterLoad</td><td>Cloned Mapper</td></tr>
<tr><td>FieldBuildFilter</td><td>Cloned Mapper, One for every field with filter criteria</td></tr>
<tr><td>MapperBeforeRequery</td><td>Cloned Mapper</td></tr>
<tr><td>MapperExecSQL</td><td>Cloned Mapper</td></tr>
<tr><td>MapperAfterRequery</td><td>Cloned Mapper</td></tr>
<tr><td>RowCurrent</td><td>Cloned Mapper</td></tr>
<tr><td>FieldButtonClick</td><td>Cloned Mapper</td></tr>
</table>
</td></tr>
<tr><td></td><td></td></tr>
</table>

TypedMapper Objects

TypedMapper objects that have custom functionality added to them should all be placed in the Data project. This allows that functionality to be shared with other consumers. You can add TypedMapper objects anywhere in your code base and add different functions to each instance. That is not recommended.

We create a typed mapper object by deriving the class from the equivalent generated template typed mapper object. The template object gives you all the type safe declarations for fields on the mapper that are on the mapper when flavor 0 is applied to the mapper. All fields with an include flavor are not in a generated TypedMapper.

To declare a typed mapper, you derive your class from the generated class template

```
public class People : Comensura.Data.Generated.people<People>


public class lkpJobCategories : Comensura.Data.Generated.lkp_job_categories<lkpJobCategories>


public class CompaniesTemplates : Comensura.Data.Generated.companies_templates<CompaniesTemplates>
```

More generally

```
    public class TMName : Comensura.Data.Generated.tm_name<TMName>
```

You give your typed mapper class the name almost exactly derived from the generated class name, removing any underscores and proper casing the letters for legibility. Your template object is your class. Then just treat your object like any other class and add public and private methods as appropriate.

Code in Extension or TypedMapper?

A commonly asked question is where to put your business logic. The temptation is to add business logic to Mapper extensions. Well, the basic rule of thumb is that only decision/workflow/UI logic is put in the extension and data manipulation/business logic is performed in the TypedMapper object.

Having said that, there is nothing inherently wrong with putting data manipulation code in an extension as long as it's self contained.

If you add data manipulation code that is likely to be shared then you should probably think about putting that code as a method on the TypedMapper object. The alternative to sharing code via the TypedMapper is to share the code in the Common class, but as a static method.

So, back to the TypedMapper object. Remember a TypedMapper is just a mapper and whenever you call your TypedMapper functions, your logic is going to read and write data from the current row. Of course there are many things you can do in your functions but they all relate to some operation controlled or directed by the values in the current row.

Page Extensions

Page extensions are used to handle events from certain types of page. At the moment there are two types of page that support extensions and fire page events. Console pages and Wizard pages. To create a page extension, you create a module that derives from `NetQuarry.PageExtensionBase.` This extension handles the following events

Event	Description
ConsolePaneBeforeLayout	Called on each page element when a console page is loaded. At this point in the page lifecycle, the page element's mapper does not contain any data, so you can only interrogate the querystring parameter data in order to make decisions regarding the page/page element behavior. After this even has completed, you will not be able to modify the following page, or pane attributes… • Whether the pane can be expanded/collapsed • Pane header visibility • Pane element drag/drop (this is a global attribute setting on the page itself) • Pane visibility Currently you should not change the "FixedAt…" position attributes of the console pane. Also you should not access the LinkAdd, LinkList, LinkNew objects in this event as the objects have not yet been added to the console pane.
ConsolePaneBeforeRequery	Called on each page element after all the page elements have been initially loaded. At this point in the page lifecycle, the page element's mapper does not contain any data, so you can only interrogate the querystring parameter data in order to make decisions regarding the page/page element behavior. At this point, it's too late to change the behavior of the following page, or pane attributes… • Whether the pane can be expanded/collapsed • Pane header visibility • Pane element drag/drop (this is a global attribute setting on the page itself) • Pane visibility However, at this point you could modify any other panel attributes to modify panel behavior.

Event	Description
ConsolPaneAfterRequery	Called on each page element after all the page elements have been initially loaded. At this point in the page lifecycle, the page element's mapper has been requeried. Now you can make decisions on page/page lement behavior based on the data in the mapper object (as well as querystring parameters). At this point, it's too late to change the behavior of the following page, or pane attributes… • Whether the pane can be expanded/collapsed • Pane header visibility • Pane element drag/drop (this is a global attribute setting on the page itself) • Pane visibility However, at this point you could modify any other panel attributes to modify panel behavior.
WizardPageLoad	Called when a wizard page is loaded.
WizardNext	Called when the user clicks on the next button. Gives you the chance to modify the workflow of the wizard pages by letting you specify what the next page to display should be.
WizardPrevious	Called when the user clicks on the previous button. Gives you the chance to modify the workflow of the wizard pages by letting you specify what the next page to display should be.
WizardCancel	Called when the user clicks on the Cancel button. Nothing is saved. You have the ability to change the CancelAction navigation parameters in order to navigate to a different page than those specified in meta data.
WizardFinish	Called when the user clicks on the Finish button. This event is fired before any mappers are saved and gives you one last chance to modify instance values in the wizard, or make the mapper dirty before the save. You also have the ability to change the FinishAction navigation parameters in order to navigate to a different page than those specified in meta data.
WizardDataExchange	Called when the user clicks on either the Next button, or the Finish button. The primary purpose of this event is to get access to the data from a page just after the data has been transferred from the page to the mapper and into the UserData collection. When this event is fired, the mapper and user data has all the data entered from that page.. When this event is fired, an event argument is provided to tell you if the event is fired prior to moving to the Next page, or prior to Finishing.

What's in the Console Event Arguments?

When a Console Pane event is called, you receive the console pane firing the event in the sender argument. Additional contextual information is passed via the ConsoleEventArgs object.

The following useful objects are available to you from the sender object.

IConsolePane Parameter	What do you get
sender	The console pane firing the event.
sender.Console.PageInfo	The page info of parent container of this console page element.
sender.ElementInfo	The page element info object of the console pane firing the event.
sender.Renderer	The user control object that is rendering into the console pane. In order to manipulate the properties of the object, you have to know the object type of the pane to render and cast the sender.Renderer to that object type.

The ConsoleEventArgs parameter currently only returns a result property on e.Result. There are no other useful properties at the moment.

What's in the Wizard Event EventArgs?

There are two sets of event args for wizard page events. For WizardDataExchange events, you get WizardDataExchangeArgs. For all other events, you receive the WiardPageEventArgs.

The following useful objects and functions are available to you from the WizardPageEventArgs parameter.

WizardPageEventArgs Parameter	What do you get
e.NextPage	The index of the next page expected. You can set this index to change the next page
e.Wizard.GetPageNumber(pageName)	The index of a page in the wizard given the pages name
e.WizardPage.Mapper	The mapper on the page that caused the navigation (either next or previous)
e.WizardPage.PageElementInfo	Information about the page of the wizard that cause the navigation (either next or previous)

WizardPageEventArgs Parameter	What do you get
e.WizardPage.UserData	Access to the in memory cache of wizard data

And for WizardDataExchange events.

WizardDataExchangeArgs Parameter	What do you get
e.ExchangeType	Indicates whether the data exchange is occurring as a result of the Next button being pressed (e.ExchangeType == WizDataExchangeType.NextPageToUserData) or the if the finish button was pressed (e.ExchangeType == WizDataExchangeType.FinalPageToUserData).
e.Wizard.GetPageNumber(pageName)	The index of a page in the wizard given the pages name
e.WizardPage.Mapper	The mapper on the page that caused the navigation (either next or previous)
e.WizardPage.PageElementInfo	Information about the page of the wizard that cause the navigation (either next or previous)
e.WizardPage.UserData	Access to the in memory cache of wizard data

Event Handling

It is important to understand when the page events fire and what data is available during the event

Console Pane Events

<table>
<tr><th>Action</th><th>Event Fired</th></tr>
<tr><td>Navigate to console</td><td>
<table>
<tr><th>Event Order</th><th>Notes</th></tr>
<tr><td>ConsolePaneBeforeLayout</td><td>Event fired sequentially on each console pane. Before further events are fired</td></tr>
<tr><td>ConsolePaneBeforeRequery</td><td>Fired on each console pane</td></tr>
<tr><td>ConsolePaneAfterRequery</td><td>Fired on each console pane</td></tr>
</table>
</td></tr>
</table>

Wizard Page Events

<table>
<tr><th>Action</th><th>Event Fired</th></tr>
<tr><td>Navigate to wizard</td><td>
<table>
<tr><th>Event Order</th><th>Event Arg Info</th></tr>
<tr><td>WizardPageLoad</td><td>e.NextPage == 1</td></tr>
</table>
</td></tr>
<tr><td>Next Page (Page 1 to Page 2)*</td><td>
<table>
<tr><th>Event Order</th><th>Event Arg Info</th></tr>
<tr><td>WizardPageLoad</td><td>e.NextPage == 1 (page load of the leaving page)</td></tr>
<tr><td>WizardDataExchange</td><td>e.ExchangeType == WizDataExchangeType.NextPageToUserData</td></tr>
<tr><td>WizardNext</td><td>e.NextPage == 2</td></tr>
<tr><td>WizardPageLoad</td><td>e.NextPage == 2 (page load of the next page)</td></tr>
</table>
</td></tr>
<tr><td>Previous Page (Page 2 to Page 1)</td><td>
<table>
<tr><th>Event Order</th><th>Event Arg Info</th></tr>
<tr><td>WizardPageLoad</td><td>e.NextPage == 2</td></tr>
<tr><td>WizardNext</td><td>e.NextPage == 2</td></tr>
<tr><td>WizardPageLoad</td><td>e.NextPage == 1</td></tr>
</table>
</td></tr>
<tr><td>Cancel (Any page)</td><td>
<table>
<tr><th>Event Order</th><th>Event Arg Info</th></tr>
<tr><td>WizardPageLoad</td><td>e.NextPage == Page num of page clicked</td></tr>
<tr><td>WizardCancel</td><td>e.NextPage == Page num of page clicked</td></tr>
</table>
</td></tr>
<tr><td>Finish*</td><td>
<table>
<tr><th>Event Order</th><th>Event Arg Info</th></tr>
<tr><td>WizardPageLoad</td><td>e.NextPage == (last page)</td></tr>
<tr><td>WizardFinish</td><td>e.NextPage == (last page)</td></tr>
<tr><td>WizardDataExchange</td><td>e.ExchangeType == WizDataExchangeType.FinalPageToUserData</td></tr>
</table>
</td></tr>
</table>

* Note the difference in event firing order between the Next Page handling and the Finish handling. In the Next Page events, the WizardDataExchange occurs before the WizardNext event. In the Finish events, the WizardDataExchange occurs after the WizardFinish event. This also implies that the values from the final page of the wizard are not transferred from the page to UserData nor Mapper in the WizardFinish event.

Related Mapper Context

Frequently you want to know about the mapper of the parent record. This is particularly the case when you want to manipulate the behavior of a mapper when used to display subform records.

A mapper will only have a valid RelatedMapperContext when the page is navigated to, with a parmop and a parrk query string parameter. (Or parmop, parval, parkey parameters if no parrk).

These query string parameters identify the parent information sufficiently to instantiate a mapper requeried onto the row of the parent. Strictly speaking you could instantiate the parent mapper object yourself after interrogating the query string parameters, but the RelatedMapperContext object hides all that complexity from you.

The RelatedMapperContext property on a Mapper is accessed from the ParentContext property

```
using NetQuarry.Data;
using Comensura.Data;
namespace Comensura.Extensions
{
  public class MyClass : NetQuarry.Data.TypedMapperExtension<Comensura.Data.MyTypedMapper>
  {
    public override void MapperBeforeRequery(Comensura.Data.MyTypedMapper sender, EAPEventArgs e)
    {
      RelatedMapperContext rc = sender.Mapper.ParentContext;
    }
  }
}
```

Property/Method	Description
GetParent	An overloaded method to instantiate a live parent mapper object.
GetParent()	Providing no parameters you will get a parent mapper loaded and requeried to the row specified by the value of the RowKey property
GetParent(string parentKey)	You can override the parent mapper to obtain an instance of a different mapper but that mapper will be filtered to the value of the RowKey property.
GetParent(string parentKey, string parentRowKey)	You can override both the parent mapper and parent row key to get an instance of a mapper.
Key	The name of the parent mapper.
Rowkey	The value of the rowkey (primary key or uniquekey) of parent mapper. The parent mapper knows what field on the parent mapper this value applies to. If all you want to know is the value of the parent's rowkey, then this is the only property you'll need to access.

When you use the GetParent method and you have an instance of a mapper object, you shouldn't explicitly close the object. The lifetime of the mapper attached to the RelatedMapperContext object is managed by the RelatedMapperContext object.

If you close the mapper and there are other extensions attached to the same mapper, all expecting to use the RelatedMapperContext, once the mapper is closed, their code will likely fail.

Task Handlers

Task handlers are components that are derived from the NetQuarry.ScheduledHandler. These components are used to schedule processes via the scheduler.

There is one event handler method for the scheduled handlers class. OnExec. When you create a new class to handle a scheduled event, implement something like the following example.

```
namespace Comensura.Tasks.Invoice
{
 /// <summary>
   /// Scheduled Task handler for invoicing.
 /// </summary>
 public class InvoiceHandler : NetQuarry.ScheduledHandler
 {
  private IDatabase _database;
  private IAppContext _app;

  /// <summary>
  /// Add a constructor if necessary
  /// </summary>
  public InvoiceHandler()
  {

  }

  /// <summary>
  /// Handles the timer event
  /// </summary>
  /// <param name="cmdID">The commmand from the ScheduledTasks table.</param>
  /// <param name="args"></param>
  ///
  protected override void OnExec(int cmdID, params object[] args)
  {
   _database = this.Application.DataDB;
   _app = this.Application;

   ProcessInvoiceQueue();
  }
  /// <summary>
  /// ProcessInvoiceQueue - Main Procedure for InvoiceHandler Trigger
  /// </summary>
  /// <param name="database"></param>
  public void ProcessInvoiceQueue()
  {
   string where = "some filter";
   // Create a mapper reader to iterate through some records.
   using (TypedMapperObjectName map =
       TypedMapperObjectName.OpenReader(_app, where, 0, MapperAttrs.NoRowRequery))
   {
    while (map.MoveNext() && !this.IsServiceStopped)
    {
     // Do some stuff.
    }
   }
  }
 }
}
```

What your handler does is not going to affect the basic ideas of the above code.

- Your constructor will perform some basic initialization if any
- You implement the OnExec event handler. If necessary, you read the cmdID to differentiate which task to run. If your handler has only one use you can ignore this parameter.
- You cache the app and database objects.
- Your main handler functionality is not in the OnExec method.
- More than likely your handler will perform an operation on many records and therefore your code will contain a loop. You must allow the loop to be interrupted if the scheduler has been requested to stop (via windows services management tool). If the task has been interrupted, the IsServiceStopped property would be set to true.

Setting up a Scheduled Task

To set up a scheduled task, first step is to create a task handler module as described above.

Then, you have to add your new component to the studio as a Handler component. As with adding any component to the system, you associate with a module, give it a name and specify the appropriate component information.

Once you have added the task as a handler component, you can add a new Scheduled Task to the studio.

Name	Description
Task Name	The readable name of the task.
Component	The name of the Handler that performs the functions of the task
Interval Mins	The interval in minutes when the Scheduler will tell the task to perform its function.
Command Id	A single Handler may be configured to perform many different functions. If that is the case, you can associate each piece of functionality in the handler with a specific Command ID, so that when you handle the request, you can process the correct function.
Enabled	Determines whether the task is enabled or not. When you check in your meta data, you must remember to disable your task from running. It should not be enabled by default as it will be running in production when it's not ready to run. You enable a task on a machine by setting the appropriate UPDATE statement in the cnet_data-migration.sql file. Look at the section ENABLE_SPECIFIC_SCHEDULED_TASKS
Restrict Times	You may want to have a task run at a regular interval but during a certain time of day. Check this box to specify a time restriction for interval based tasks.
Start Time	The time at which your task should start running at the required interval
End Time	The time at which your task should stop running at the required interval

Name	Description
Run Once Time	If you want the scheduler to run at a specific time every day, you would set the Run Once Time. The scheduler will fire your task once per day. This setting overrides the Interval property.
Attributes	Only one attribute is currently valid and that is Suynchronous. By default all tasks are asynchronous and they will run simultaneously on different threads (depending on the interval and time restriction properties). If you set a scheduled task to synchronous, it will always run before any asynchronous tasks have started. Only when synchronous tasks have completed, will the asynchronous tasks be allowed to run. Be careful with synchronous tasks and only to have them run at large intervals (like once per day). If the period is too short, they can run in preference to any/all asynchronous tasks and not allow asynchronous tasks to run.
Param Txt	Any text arguments you might want to pass to your task. They will be passed to your handler in the Arguments parameter.

Tasks have an optional property, specifying a machine name on which the task should run. If no machine name is provided, then the task will run on all machines where the scheduler is running.

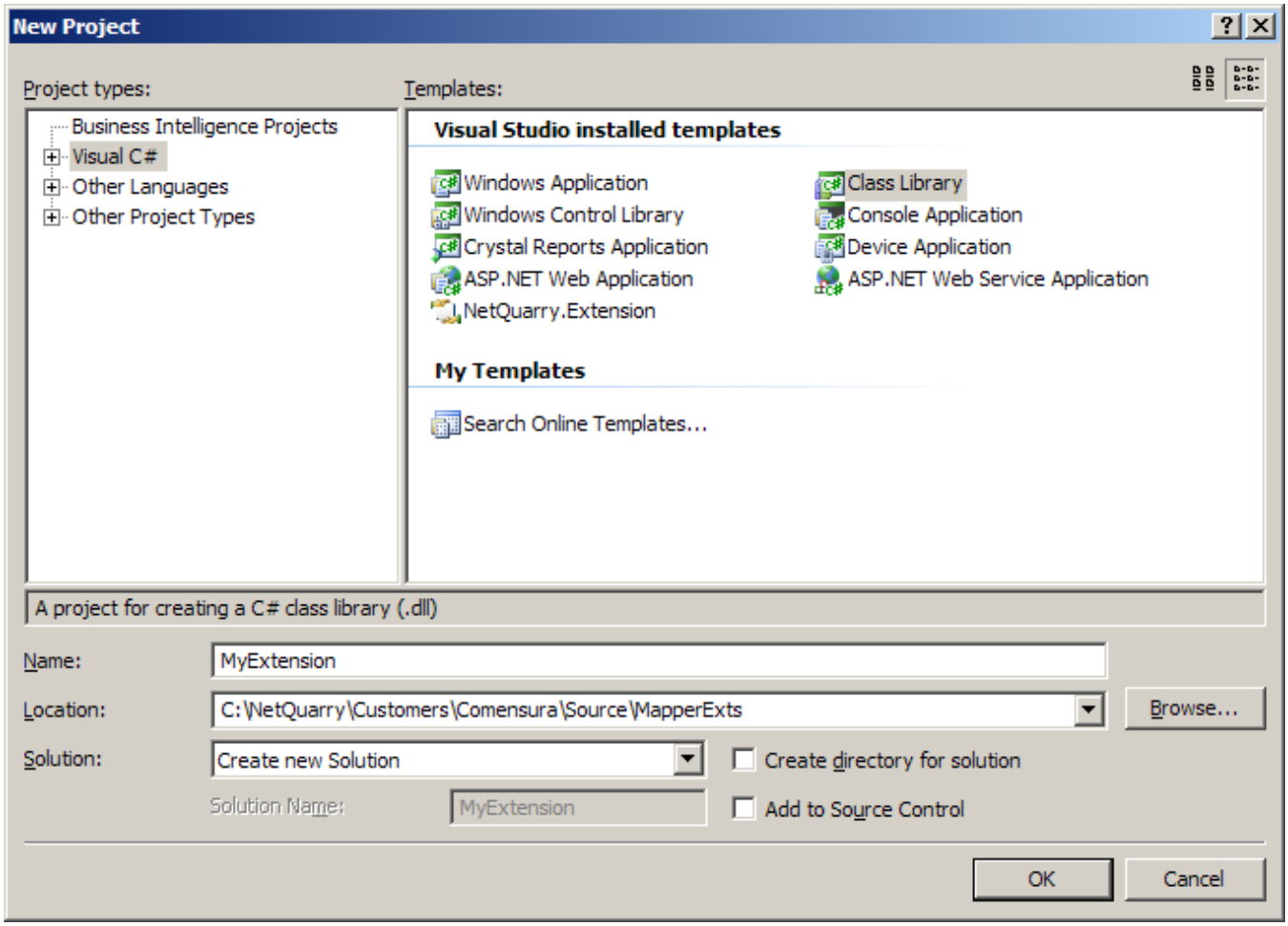
You can also add custom properties to your scheduler task, but it might be better to provide an operational data table to configure task behavior so end user administrators (and developers) can manipulate task behavior from your front end application and not have to log into an app server and use the studio.

Creating a new extension

We have a dev studio wizard installed that creates a new mapper extension. Here we will describe the manual process for creating a new mapper extension.

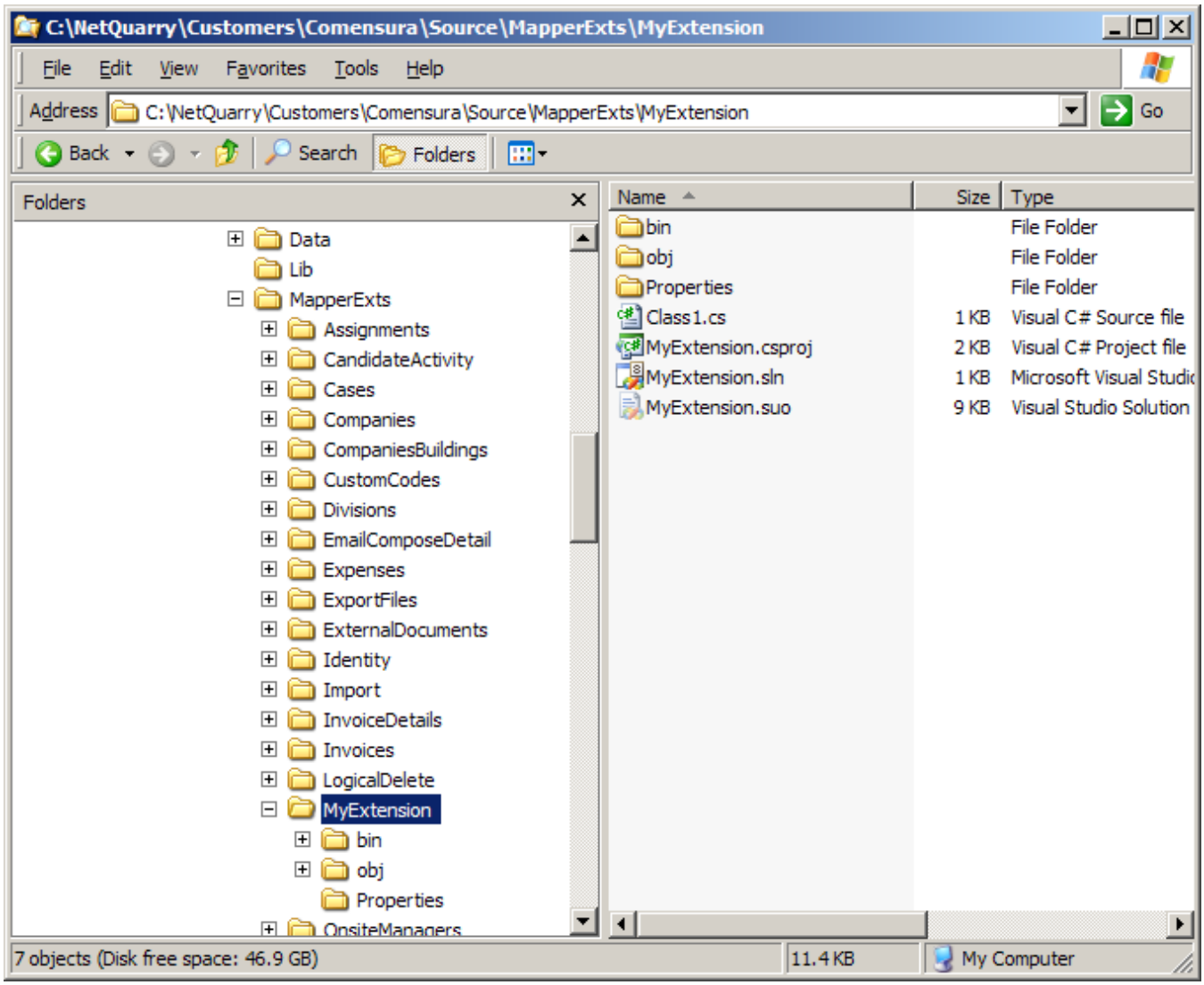
In dev studio, do File, New, Project

Enter the information appropriate to your extension



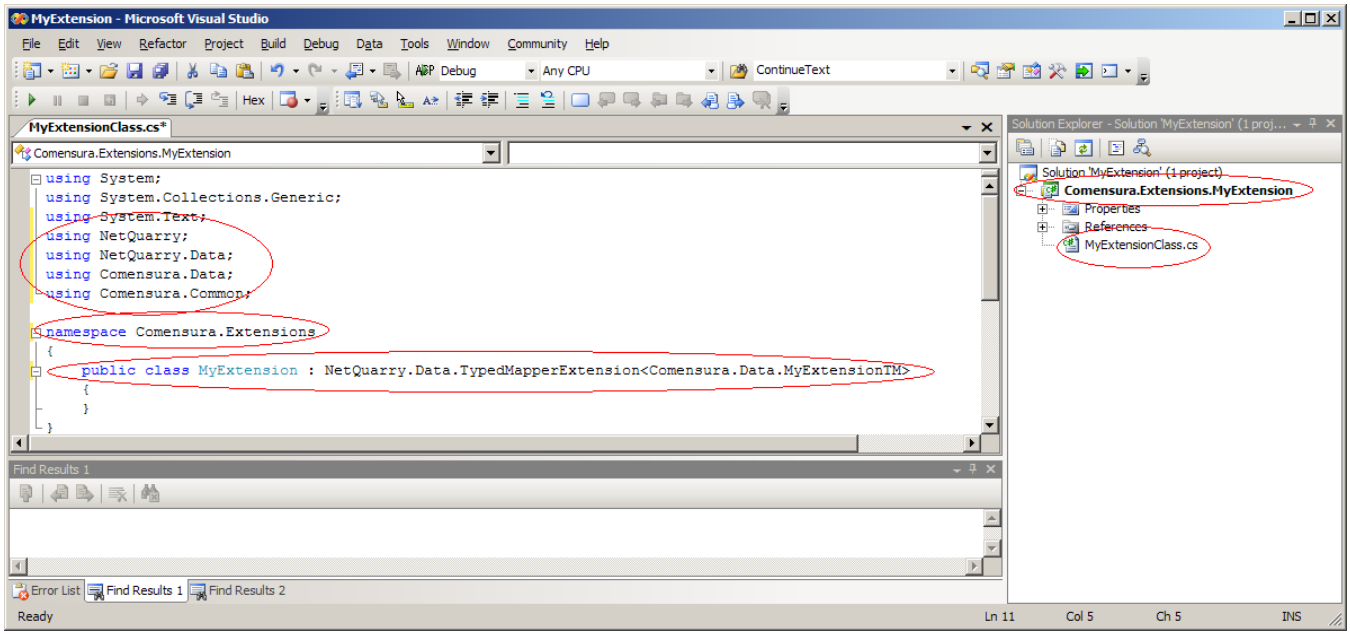
Give your project a name and make sure you've set the location to your MapperExts folder. Click OK.

That creates an initial folder structure as follows

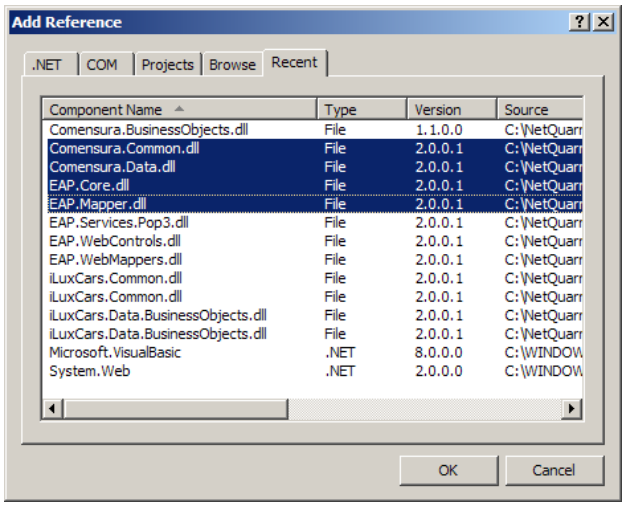


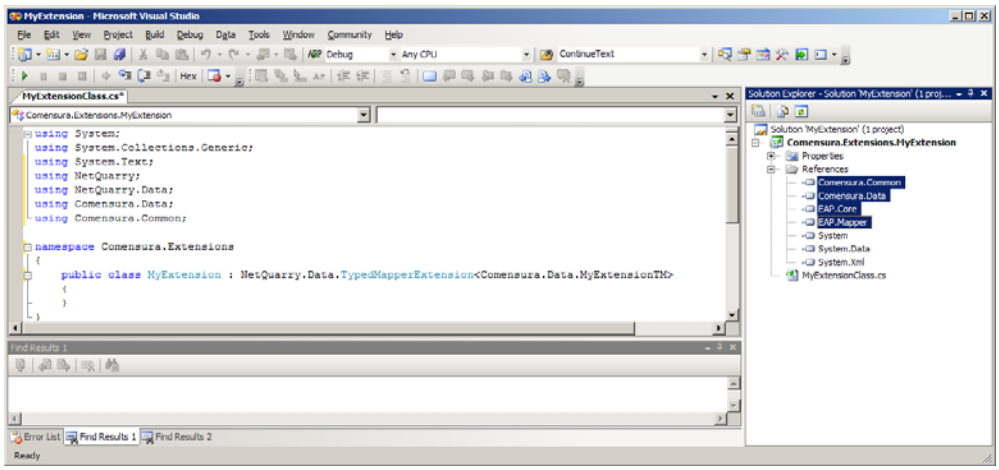
In your studio, modify the following settings.

- Add the using statements
- correct the namespace
- declare your class and derive from the appropriate base class.
- change the project name to the fully qualified namespace name
- change the name of the class file to be the same as the class name



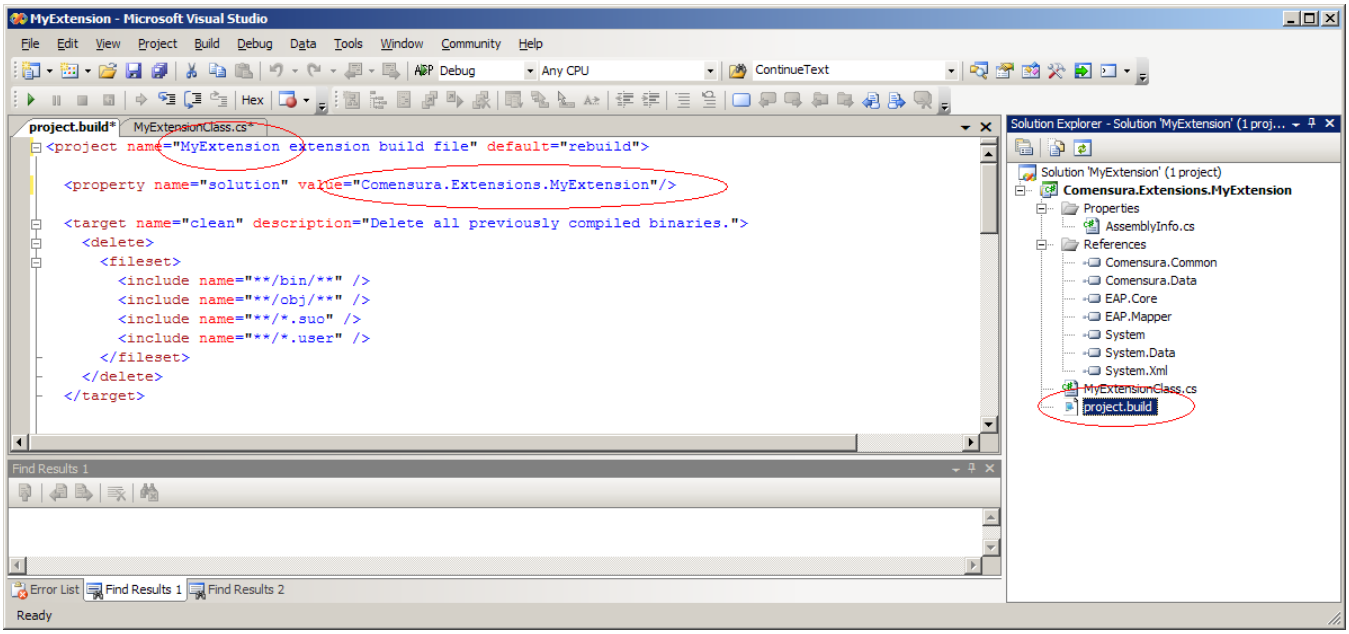
Add references to NetQuarry and Comensura classes



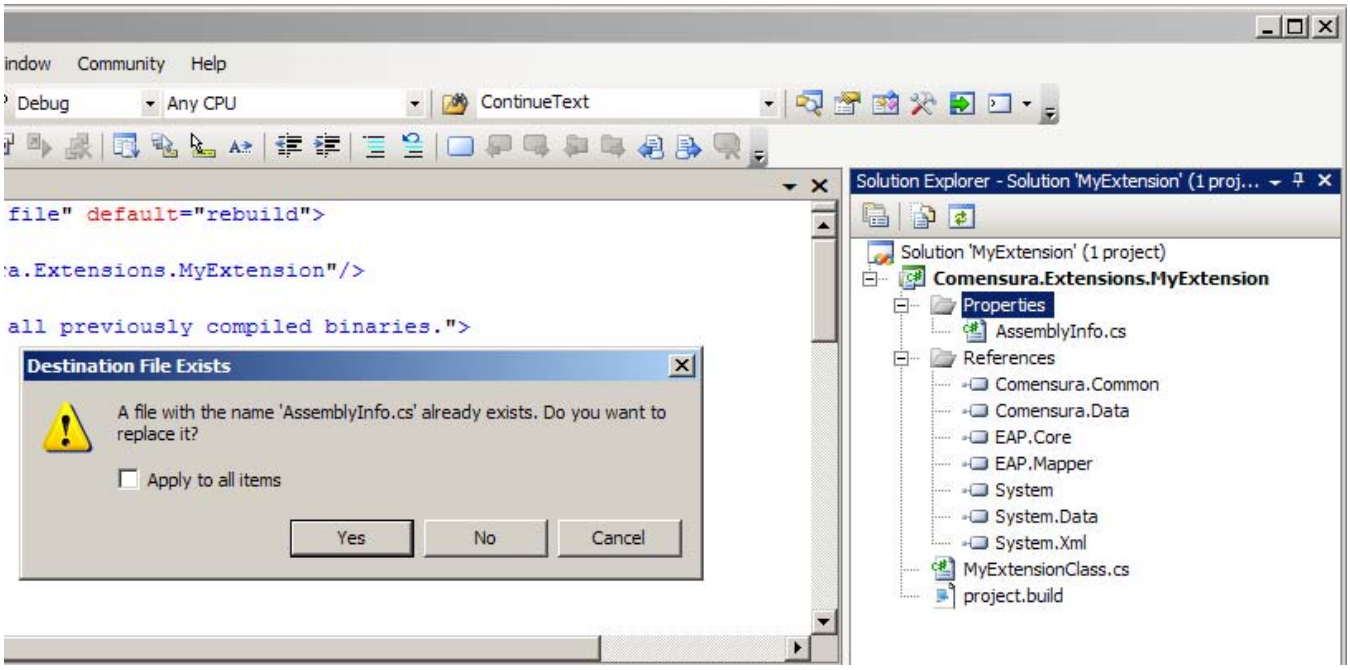


Add a project.build file. This ensures your component is included in the build process. The best thing to do here is copy a build file from an existing project. Even easier is to drag a project.build file from explorer into the project

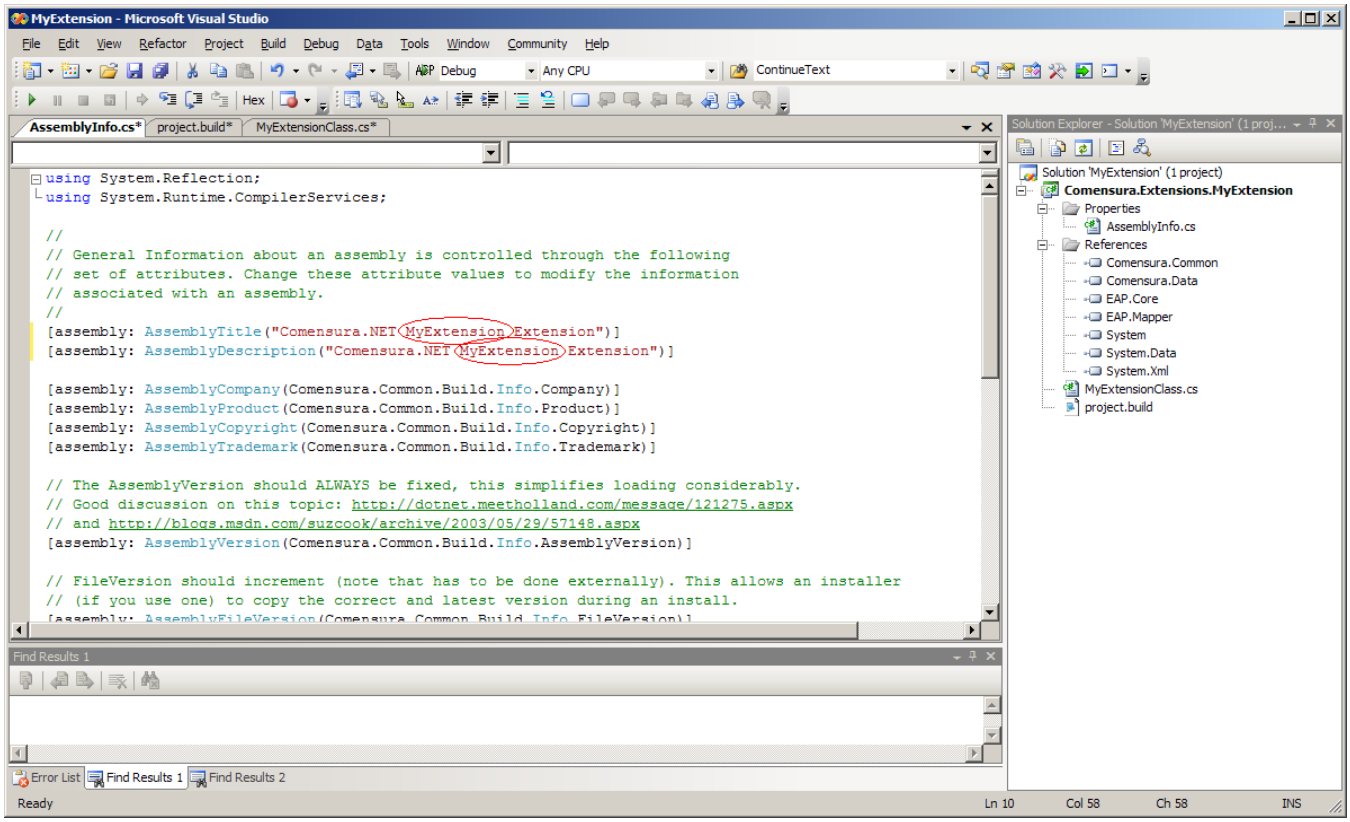
Make the changes highlighted



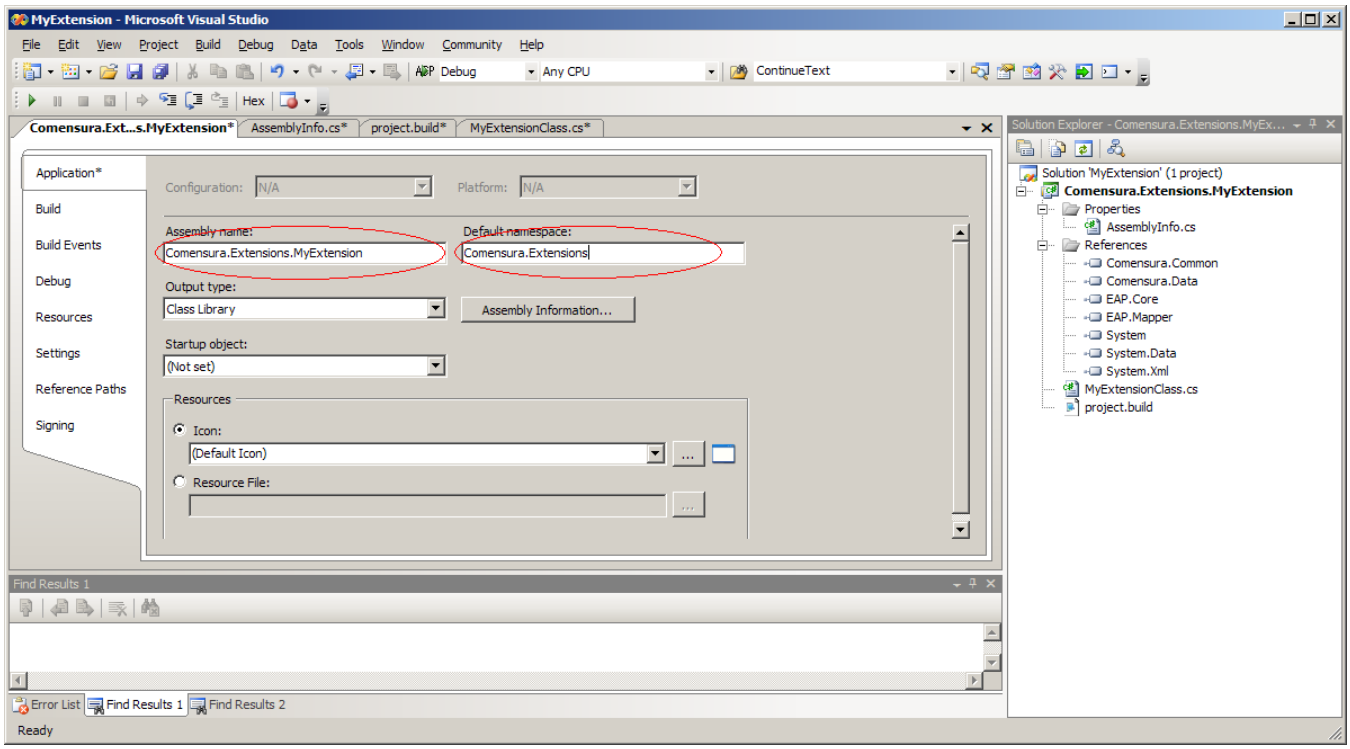
Now update the AssemblyInfo.cs file. Again, copy or drag from another project and update



Update the highlighted parts as appropriate



Then right click on the project file and choose properties, and change the entries as marked



Now build the project just to make sure (even though you have nothing in it yet)

Now you have to add your extension to the meta data so it can be referenced by a mapper as an extension.

In NQ studio, go to Components, Extensions and add the following data (appropriate to your extension)

Component Column	**Value**
Module	MyModule
Name	MyExtension
Component Type	Extension
Component Name	Comensura.Extensions.MyExtension
Assembly Name	Comensura.Extensions.MyExtension.dll
Assembly Path	C:\NetQuarry\Customers\Comensura\Source\MapperExts\MyExtension\bin\debug
Assembly Path Prod	%NQROOT%/Apps/cnet/bin
Attributes	0

Now you can refer to your extension in the list of extensions.

- Run the debug build batch file to make sure your component really is going to be included in the daily build.
- Check in
    - Comensura.Extensions.MyExtension.csproj
    - project.build
    - your class files
    - Properties\AssemblyInfo.cls
- DO NOT check in solution file Comensura.Extensions.MyExtension.sln
- Make sure you send us an email so we can add your component to the installer.

Module Naming Rules

1. All NetQuarry.Mapper extensions should be named the same as the module/mapper that they primarily extend. They should be stored in a folder named $/Source/MapperExts/ where the name of the folder is the same as the mapper extended. For example, for the people Mapper, the extension is stored in the "$/Source/MapperExts/People" folder.
2. All code should have a namespace starting with Comensura. For example, the common project is Comensura.Common, the Data project (where the derived TypedMapper objects live, the namespace is Comensura.Data.
3. Assemblies should be named the same as the primary namespace. For example, Comensura.Common.dll, Comensura.Data.dll.
4. Only the project file should be checked in, NOT the solution file. The project file name should be the same name as the assembly (with the proper extension). For example, for the People extension, the naming works like this:

- **Project File:** Comensura.Extensions.People.csproj
- **Assembly File:** Comensura.Extensions.People.dll
- **Extension Namespace:** Comensura.Extensions
- **Extension Class:** Comensura.Extensions.People
- **Extension Folder:** Source\MapperExts
- **TypedMapper:** Comensura.Data.People
- **PageExtension Class:** Comensura.PageExts.People
- **PageExtension Namespace:** Comensura.PageExts
- **PageExtension Folder:** Source\PageExts
- **Task Namespace:** Comensura.Tasks
- **Task Class:** Comensura.Tasks.PeopleTrigger
- **Task Folder:** Source\Tasks

Saved Filters

Saved filters provide a mechanism for filters on a mapper to be persisted. The platform itself uses Saved Filters to remember the pinned items on datasheet view, for remembering filter criteria for reports and bulk mailer.

The best place to start this is with describing the properties and methods of a SavedFilter object.

Constructor	Description
SavedFilter	Full constructor for the SavedFilter class. The constructor takes an application object and a database object in the constructor. The application object is required to allow the new SavedFilter to be added to the application's Filters collection and the database object is required for the filter to be persisted to the database.

Properties

<table>
<tr><th>Property Name</th><th>Description</th></tr>
<tr><td>Attributes</td><td>The attributes for the filter. The attributes are from the FilterAttributes enumeration. The attributes alter the behavior of the filter.
Some of these attributes are not typically set by an end user and are specified by inference when the SavedFilter is created.

<table>
<tr><th>Member Name</th><th>Description</th></tr>
<tr><td>Static</td><td>This view consists of a set of keys (vs. a query filter).</td></tr>
<tr><td>StaticFilter</td><td>This view consists of a SQL IN clause filter with a set of keys. This is for future use to support static filtering against databases that don’t have NetQuarry schema tables to store filter criteria. The</td></tr>
<tr><td>Temp</td><td>This view is temporary and can be deleted any time after its end_dt. For this to have any effect, the SavedFilter must also have its EndDate property defined.</td></tr>
<tr><td>Hidden</td><td>This view is not to directly selectable in a UI. This attribute is not yet supported.</td></tr>
<tr><td>Sorted</td><td>This view contains a sort clause. This attribute is not used</td></tr>
<tr><td>KeyString</td><td>The key for this item is a string (vs. numeric). If you create a SavedFilter manually, you have to specify one of the key type attributes. If you create a saved filter via the Mapper Exec, the key type is determined automatically from the type of the primary key field on the mapper.</td></tr>
<tr><td>KeyNumeric</td><td>The key for this item is numeric (vs. string). If you create a SavedFilter manually, you have to specify one of the key type attributes. If you create a saved filter via the Mapper Exec, the key type is determined automatically from the type of the primary key field on the mapper.</td></tr>
<tr><td>KeyGuid</td><td>The key for this item is a GUID (vs. string). If you create a SavedFilter manually, you have to specify one of the key type attributes. If you create a saved filter via the Mapper Exec, the key type is determined automatically from the type of the primary key field on the mapper.</td></tr>
<tr><td>DeleteAfterUse</td><td>The consumer of the filter may delete the filter when finished with it. In order to differentiate whether it’s safe to delete a SavedFilter, when creating the filter, you can set this attribute to indicate it should be deleted when it’s no longer used.
Setting and consumption of this attribute is completely under the control of the end user and not the platform.</td></tr>
</table>
</td></tr>
<tr><td>CreateDate</td><td>The date/time at which the filter was created. This property is initialized in the constructor of the SavedFilter object.</td></tr>
<tr><td>CreatorID</td><td>The creator of the filter.

If the property has not been set by the time the SavedFilter object is saved it is defaulted to the logged in user.</td></tr>
<tr><td>Criteria</td><td>The criteria use to create the filter. This XML stores criteria information that can be used to maintain the filter.</td></tr>
<tr><td>Description</td><td>The localized description of the filter.</td></tr>
</table>

Property Name	Description
EndDate	The UTC date/time at which the filter expires. Setting the EndDate can perform two functions. If the EndDate is set, when the EndDate is passed, the filter is no longer loaded in the application's Filters collection. You would also need to set the EndDate property if you want the filter to be a temporary filter (set the Temp attribute). This means the filter specification will also be deleted from the database when the EndDate is passed. If you set the EndDate property, you probably also want the SavedFilter to be a temporary filter.
Filter	The SQL filter clause for the filter. Note that for true static filters the filter clause will be a subquery against the xot_saved_filter_keys table. The filter criteria will change to the subquery on xot_saved_filter_keys when a dynamic filter is converted to a static filter. When you manually create a SavedFilter, you would need to set this property. Using the Mapper Exec, the Filter property is set from the union of all the filters on the mapper creating the SavedFilter.
From	The SQL FROM clause (typically a table or view name) for which the filter is intended. When you manually create a SavedFilter, you would need to set this property. Using the Mapper Exec, the From property is set from mapper's View property.
ID	The unique (GUID) ID for the filter. The ID serves as the key in the SavedFilters collection. This property is defaulted in the constructor of the SavedFilter. It is this unique ID that can be passed in the `&fltid=` querystring parameter to have a mapper apply a saved filter associated with that key.
KeyColumn	The column in the underlying table containing the single primary key uniquely identifying a row in the table. This key is used in filter clause building. If the wrong key column is specified, the filter results will be unexpected, or at worse, return no results. When you manually create a SavedFilter, you would need to set this property. Using the Mapper Exec, the KeyColumn property is set from mapper's PK property. If the there are multiple primary keys on the mapper, the first primary key is used. In that case it's best to specify the UniqueKey property on one of the fields on the mapper. In that case the UniqueKey field will be used as the key column.

Property Name	Description
Keys	The list of keys for a static filter. If you manually create a SavedFilter that is static, you set this property as a NetQuarry.StringSet (which is derived from the .Net StringCollection). You can set the property directly, or add/remove individual keys using the Add or Remove methods of the StringSet object. After adding/removing static keys, you must save the saved filter for the changes to be persisted. The Keys property is lazy loaded. On the first get, we then load the keys from the database.
KeysLoaded	Gets/Sets a boolean value that indicates if the Keys for this filter have been loaded. This property is set automatically when the LoadKeys method is called on the SavedFilter. This occurs when the Keys property is accessed the first time.
LocaleKey	The locale of the base filter. Note that if localized text is available for the filter it will be used in place of the original, non-localized text from this locale. If the property has not been set by the time the SavedFilter object is saved it is defaulted to the current locale of the session.
ModuleKey	The module to which this filter belongs. When you manually create a SavedFilter, you would need to set this property. Using the Mapper Exec, the KeyColumn property is set from mapper's Module property.
Moniker	The pseudo-unique moniker for the filter. Ideally, this should be unique within the SavedFilters collection, but is not guaranteed to be so. When you manually create a SavedFilter, you don't need to set this property. This property is not set when using the Mapper Exec to create a SavedFilter.
MOP	The module to which this filter belongs. When you manually create a SavedFilter, you don't need to set this property. Using the Mapper Exec, the MOP property is set from a combination of the mapper's module and page property, rather than its MOP property. If you don't set a MOP property on your SavedFilter, it will not be available to new session instances through the Application's Filters collection. When that collection is loaded, the MOP of the filter is checked against the list of MOP's available to the user logging in. If there is no match, the filter is not loaded. Also if you use the Mapper Exec to create your SavedFilter, remember that the MOP property is defaulted from the mapper's properties. For a manually created mapper, those MOP related properties are not set.

Property Name	Description
Name	The localized, natural language name of the filter. The name of the filter is optional. The filters are loaded and keyed from the ID property, rather than the name. If you provide a name it means you can search for the filter by name on the application's filters collection.
OwnerID	The owner (if any) of the filter. If no owner is specified then the filter is globally available.
Sort	The sort clause for the filter, if any. This property is optional.
StartDate	The UTC date/time from which the filter is valid. This is not yet implemented.

Methods

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td>Close</td><td>Close the filter.

This method is not implemented.</td></tr>
<tr><td>Convert</td><td>Overloaded. Convert the Saved Filter from one form to another.
The Convert method takes a parameter of type SaveFilterConvertOptions. The overloaded method takes an integer parameter that limits the number of keys created (TopN) when converting a filter to a static filter.

<table>
<tr><th>SaveFilterConvertOptions</th><th>Description</th></tr>
<tr><td>ToStatic</td><td>Convert the filter to a static filter.</td></tr>
<tr><td>ToDynamic</td><td>Convert the filter to a dynamic filter. (not implemented)</td></tr>
<tr><td>NoSave</td><td>Do not save changes after the conversion. Note, however, that if the filter is being converted to a static filter the filter keys will have been saved.</td></tr>
</table>

The only supported conversion method at the moment is to convert a dynamic filter to a static filter.</td></tr>
<tr><td>Delete</td><td>Delete the filter.

This method deletes the filter from the database. If the filter is a static filter, the related keys are also deleted.</td></tr>
<tr><td>GenerateSubquery</td><td>Overloaded. Generate a subquery for this filter appropriate for use in a SQL IN clause.

The subquery performs a DISTINCT select on the requested columnname. The overloaded method provides an option for setting an alias for the column name. The DBMSType property is required to identify whether the generated clause contains a DBMS specific reserved word. If it does, the column name will be escaped by [and].</td></tr>
<tr><td>Loaded</td><td>Tells the SavedFilter that it was loaded from the database, rather than being constructed programmatically. This is used by the platform. It sets up internal values so that changes to the filter can be detected.</td></tr>
<tr><td>Open</td><td>A static method that allows a filter to be loaded from the database.

The filter is loaded by the ID. The created filter is not added to the application’s filters collection.</td></tr>
<tr><td>Save</td><td>Save the filter to the database.

The database to which the filter is saved is defined by the database object provided during initial construction of the SavedFilter</td></tr>
</table>

Create a SavedFilter

There are two ways to create a saved filter. You can directly create a SavedFilter object and populate it yourself. Take the more convenient approach to create a SavedFilter via a Mapper Exec.

Create New

This example is based on the code that handles the Mapper Exec

```
//--- Create filter object, passing app and database (database must have xot tables)
SavedFilter  flt = new SavedFilter(_appContext, _dbData);.
//--- Identify the key field for your SavedFilter.
IField    fldPK = Fields.Find(null, FieldFindType.PK);

flt.Name = "FilterName";
//--- Get the collection of filters from your mapper and have it generate an appropriate description
MapperFilters flts = mapper.Filters;
string fltDesc = EAPUtil.StripHTML(flts.GetShortDesc(FilterDescOptions.NoPrefix | FilterDescOptions.PlainText));
flt.Description = fltDesc;
flt.ModuleKey = mapper.ModuleKey;
flt.OwnerID = _appContext.UserContext.ID;

//--- MOP is necessary if you want to load filter into application in a different session, or in scheduled task
//--- the logged in user must have permission to see the MOP for the filter to be loaded.
flt.MOP = mapper.MOP;  //--- Care should be taken.  Not all mappers have a MOP specified.
flt.From = mapper.View;
flt.Attributes = filterAttrs;

if (fldPK != null)
{
  flt.KeyColumn = fldPK.Key;
}
else //--- Throw decent error when no PK found.
{
  throw new ApplicationException("Unable to create filter -- no primary key found");
}

//--- This section is to identify whether you have some keys selected (meaning user has checked rows in the datasheet)
ArrayList  lstKeys = mapper.SelectedKeys();

//--- if some keys found, it's going to be a static filter
if (lstKeys != null && lstKeys.Count > 0)
{
  flt.Attributes |= FilterAttributes.Static;
  flt.Attributes &= ~FilterAttributes.StaticFilter;

  for (int ii=0; ii<lstKeys.Count; ii++)
  {
    //--- add the keys
    flt.Keys.Add(lstKeys[ii]);
  }
}
else
{
  //--- Include ALL filters, not just user filters.  This converts the collection of filters to a string.
  flt.Filter = flts.GetFilter(GetFilterFlags.IncludeAllTypes);
}

//--- Set the key type based on the type of the primary key field
if (EAPUtil.TypeIsInteger(fldPK.OleDbType))
{
  flt.Attributes |= FilterAttributes.KeyNumeric;
}
else if (fldPK.OleDbType == OleDbType.Guid)
{
  flt.Attributes |= FilterAttributes.KeyGuid;
}
else if (EAPUtil.TypeIsChar(fldPK.OleDbType))
{
  flt.Attributes |= FilterAttributes.KeyString;
}

//--- Save the filter to the DB.
flt.Save(_appContext);

//--- and optionally add the filter to the application's Filters collection
_appContext.Filters.Add(flt);
```

User Mapper Exec

This example uses the Mapper Exec to create the SavedFilter for you. It's much more convenient.

```
//--- so construct an appropriate name
string fltName = string.Format("user:{0}:{1}", this.Application.UserContext.ID, DateTime.UtcNow);
SavedFilter sf = mapper.Exec(MapperExecCmds.FilterSave, 0, fltName) as SavedFilter;
//--- right now we have a new SavedFilter object (constructed similarly to the example above)
//--- The SavedFilter is attached to the application and it is saved to the database
//--- You can now override/modify the SavedFilter properties to customize the behavior
sf.OwnerID = "a_user_id_other_than_current_user";
//--- If necessary, convert the dynamic filter to a static.  It's up to you
//--- if the filter is already static (because keys were selected) nothing happens
sf.Convert(Application, people.Database, SaveFilterConvertOptions.ToStatic);
//--- save these changes to the database.
sf.Save(this.Application);
```

Get an Existing SavedFilter

There are two ways to get an existing saved filter. You can open the filter from the database, or find a filter in the application's Filters collection.

Open It

Basically use the static Open method on the SavedFilter object. To open a SavedFilter in this way, you have to know the filter's ID.

```
//--- you have to obtain a filter id.  Here we get the value from the querystring
//--- &fltid= is a platform supported querystring parameter.  Whenever it is discovered, the platform
//--- will add the SavedFilter to the mapper on the page.
string   fltId = HttpContext.Current.Request["fltid"];

if (!string.IsNullOrEmpty(fltId))
{
  SavedFilter flt = SavedFilter.Open(appCxt, fltId);
}
```

Find it

You can also get an existing SavedFilter by searching for it on the application's Filters collection.

```
//--- you have to obtain a filter id.  Here we get the value from the querystring
//--- &fltid= is a platform supported querystring parameter.  Whenever it is discovered, the platform
//--- will add the SavedFilter to the mapper on the page.
string   fltId = HttpContext.Current.Request["fltid"];

if (!string.IsNullOrEmpty(sFltId))
{
  SavedFilter flt = app.Filters.Find(fltId, FilterFindType.ByID);
}
```

You can also search for a filter by its moniker. Of course if you do search by moniker you have to ensure that each moniker you provide to a filter is sufficiently unique.

Registered Filters

Registered Filters are essentially an implementation of a Saved Filter. You use a Registered Filter when you want to perform some filtering of the mapper on the target mop that cannot be achieved just by setting the pk querystring parameter. For example you want to navigate to the list view with the results filtered in a specific way.

In the past we allowed the filter clauses to be specified directly on the query string. For example

…&flt=order_id%3d'123ABC'%20AND%20people_id%3d'456XYZ'…

(%3d is the URL escaped value of =)

We even allowed this type of navigation parameter to be specified in meta data and to take field references

…&flt=order_id%3d['order_id]%20AND%20people_id%3['people_id]…

Because we allowed such parameters to be read from the query string, it was possible for a person to construct a URL in order to execute a SQL statement. Basically a SQL Injection attack. Therefore we removed the ability to enter filter parameters directly onto the query string. However, the functionality provided by the filter parameters was still necessary, so we devised a workaround where you can programmatically register a filter, and refer to a key to that filter on the querystring.

The platform will only accept a key to a Registered Filter when passed with the &flt= parameter. If a Registered Filter key is not specified, an exception is thrown.

Using Registered Filters

Using Registered Filters is very simple. You create the filter criteria, then register it using a static method on the SavedFilter object. You are given back a key, that you set as the &flt parameter value. Here's an example of using &flt, before this change.

```
using NetQuarry.Data;
public override void FieldButtonClick(IField sender, EAPEventArgs e)
{
  switch (sender.Key)
  {
    case "company":
// need to Encode the filter criteria so the qs handling not confused with "=" in filter clause
      string navFlt = string.Format("flt={0}", EAPEncode.ForUrl(sender.BuildFilter()));
      // If you do this now, an exception will be thrown
      this.Application.Navigate("companies_vendors!detail", null, navFlt, "nav");
      break;
  }
}
```

Here's the same example with the filter being registered

```
using NetQuarry.Data;
public override void FieldButtonClick(IField sender, EAPEventArgs e)
{
  switch (sender.Key)
  {
    case "company":
      // register the filter and get the key
      string filter = SavedFilter.RegisterReqFilter(this.Application, sender.BuildFilter();
      // pass the key as the value of the &flt parameter
      string navFlt = string.Format("flt={0}", EAPEncode.ForUrl(filter));
      this.Application.Navigate("companies_vendors!detail", null, navFlt, "nav");
      break;
  }
}
```

As you can see there is very little difference in the code to perform the same task. All we've done is converted the filter we want into a key and pass that on the query string instead of the filter clause.

There is a potential loss of functionality, however, with not being allowed to pass filter criteria directly. There is no way to specify filter criteria in meta data. It is not possible to register a filter in meta data for it to be used at run time. This means that meta data for navigation that relies on the &flt parameter must be converted to navigation via code.

If you are unsure where in meta data you might have used the &flt parameter, you can use this SQL (connected to the meta database) to determine what needs replacing.

```
SELECT * FROM xmt_properties WHERE prop_value like '%flt=%'
```

Consuming Registered Filters

For the most part, you will only ever want to register a filter and the platform will consume the registered filter and perform the appropriate action. However, you may need to extract and use a registered filter in your own code.

To do this you use a static method on the SavedFilter object to decode the filter key and return the actual filter clause.

This was how it may have been done in the past.

```
using NetQuarry.Data;
using System.Web;

public override void MapperAfterLoad(IMapper sender, EAPEventArgs e)
{
  string sFlt = HttpContext.Current.Request["flt"];
       //string                    sFlt = System.Web.HttpContext.Current.Request["flt"];
                  //--- BugzID: 5036 - Convert use of raw flt parameters. Now decode
  string sFlt = SavedFilter.ExtractFilterFromReq(sender.Application,
System.Web.HttpContext.Current.Request);
  sCompanyId = sender.Database.DBLookup("company_id", "companies", sFlt) as string;
}
```

And now with the registered filter

```
using NetQuarry.Data;
using System.Web;

public override void MapperAfterLoad(IMapper sender, EAPEventArgs e)
{
  // If you don't have a proper key on the &flt parameter, this will throw an exception
  string sFlt = SavedFilter.ExtractFilterFromReq(this.Application, HttpContext.Current.Request);
  sCompanyId = sender.Database.DBLookup("company_id", "companies", sFlt) as string;
}
```

Register Filter Methods

<table>
<tr><td>RegisterReqFilter</td><td>Register a filter clause for use in a URL's flt parameter. Registration will cause a NetQuarry.SavedFilter to be created and written to the database. The method returns a filter spec that can be included in a URL without risk of a SQL injection attack</td></tr>
<tr><td></td><td>Full parameter list
<table>
<tr><td>NetQuarry.IAppContext cxt</td><td>The application context</td></tr>
<tr><td>string filter</td><td>The filter clause to register</td></tr>
<tr><td>FilterAttributes attrs</td><td>Attributes to apply to the registered filter. The Attributes are for Saved Filters and indicate that Registered filters are just a type of saved filter. There are three attributes commonly used with Registered Filters.
Temp (all registered filters will be specified as Temp)
NoModule (the filter is not specific to a particular module)
Registered (the filter is a Registered Filter)</td></tr>
<tr><td>string name</td><td>The name of the filter. If the filter is intended to be applied to a list view navigation, then this name will appear in the caption “(Filtered On …) If no name is provided, a default name of “Parent” will appear in the caption.</td></tr>
<tr><td>string description</td><td>The description of the filter. . If the filter is intended to be applied to a list view navigation, then this string will be used as the tooltip text when hovering over the “(Filtered On …)” caption.</td></tr>
<tr><td>string mop</td><td>The mop of the navigation target. If a mop is provided then more often than not, it should be the same mop you are navigating to. This means that the filter can only be used by that one mop. You can also not provide any mop and the registered filter can be associated with any mop. When no mop is specified, the NoModule FilterAttribute will automatically be added.</td></tr>
<tr><td>DateTime ? expiration</td><td>The date at which the filter is expired. Since the registered filters are all temporary, they need to be cleaned out of the database at some point. If no expiration date is specified, the default is to have the filter expire in 1 day.</td></tr>
<tr><td>string owner</td><td>The owner of the filter . Setting the owner restricts the filter to be used only by the owner user. Use the value from cxt.UserContext.ID</td></tr>
</table></td></tr>
</table>

<table>
<tr><td>RegisterReqFilter</td><td>Register a filter clause for use in a URL's flt parameter. Registration will cause a NetQuarry.SavedFilter to be created and written to the database. The method returns a filter spec that can be included in a URL without risk of a SQL injection attack</td></tr>
<tr><td>RegisterReqFilter (2)</td><td>The most basic overloaded method. Filters created using this overload will always be marked as Temp with a default expiration (1 day) and will be restricted to the current user.
<table>
<tr><td>NetQuarry.IAppContext cxt</td><td>The application context</td></tr>
<tr><td>string filter</td><td>The filter clause to register</td></tr>
</table></td></tr>
<tr><td>RegisterReqFilter(3)</td><td>Use when you want to create a filter quickly but want an expiration date other than one day. Filters created using this overload will always be marked as Temp and will be restricted to the current user.
<table>
<tr><td>NetQuarry.IAppContext cxt</td><td>The application context</td></tr>
<tr><td>string filter</td><td>The filter clause to register</td></tr>
<tr><td>DateTime ? expiration</td><td>The date at which the filter is expired. Since the registered filters are all temporary, they need to be cleaned out of the database at some point. If no expiration date is specified, the default is to have the filter expire in 1 day.</td></tr>
</table></td></tr>
<tr><td>RegisterReqFilter(4)</td><td>Use when you want to create a filter with a default expiration (1 day) and restricted to current user, but you want to provide a filter name and description to appear on the target list view.
<table>
<tr><td>NetQuarry.IAppContext cxt</td><td>The application context</td></tr>
<tr><td>string filter</td><td>The filter clause to register</td></tr>
<tr><td>string name</td><td>The name of the filter. If the filter is intended to be applied to a list view navigation, then this name will appear in the caption “(Filtered On …) If no name is provided, a default name of “Parent” will appear in the caption.</td></tr>
<tr><td>string description</td><td>The description of the filter. . If the filter is intended to be applied to a list view navigation, then this string will be used as the tooltip text when hovering over the “(Filtered On …)” caption.</td></tr>
<tr><td>string mop</td><td>The mop of the navigation target. If a mop is provided then more often than not, it should be the same mop you are navigating to. This means that the filter can only be used by that one mop. You can also not provide any mop and the registered filter can be associated with any mop. When no mop is specified, the NoModule FilterAttribute will automatically be added.</td></tr>
</table></td></tr>
<tr><td>RegisterReqFilterForURL</td><td>This is essentially an overload of RegisterReqFilter(4) where the return value is pre-encoded for safe use in a URL.</td></tr>
</table>

ExtractFilterFromReq	Extract any general filter clause from the request. Currently this is from the flt query parameter which exposes a SQL injection attack risk. By accessing this parameter through this method the platform will immediately detect the risk and provide a single point at which to remove the risk through an improved general filtering mechanism. The method returns a filter clause back to the caller.

<table>
<tr><td>ExtractFilterFromReq</td><td>Extract any general filter clause from the request. Currently this is from the flt query parameter which exposes a SQL injection attack risk. By accessing this parameter through this method the platform will immediately detect the risk and provide a single point at which to remove the risk through an improved general filtering mechanism. The method returns a filter clause back to the caller.</td></tr>
<tr><td></td><td>Full parameter list<table>
<tr><td>NetQuarry.IAppContext cxt</td><td>The application context</td></tr>
<tr><td>HttpRequest req</td><td>The HTTP Request object.</td></tr>
<tr><td>string name</td><td>The name of the request parameter. Normally the registered filter key is pass on the &flt querystring parameter and you would typically use one of the impler overloads (2). However, if the register filter key is on a different querystring parameter, you need to use this method.</td></tr>
<tr><td>out SavedFilter filter</td><td>An out parameter. The SavedFilter, if any. This may include a filter description.</td></tr>
</table></td></tr>
<tr><td>ExtractFilterFromReq(2)</td><td>The most basic overloaded method. When all you need to retrieve is the filter clause, use this method. It assumes that the registered filter key is on the &flt parameter of the querystring.<table>
<tr><td>NetQuarry.IAppContext cxt</td><td>The application context</td></tr>
<tr><td>HttpRequest req</td><td>The HTTP Request object.</td></tr>
</table></td></tr>
<tr><td>ExtractFilterFromReq(3)</td><td>Use this overload to retrieve the associated Savedfilter object, along with the filter clause. It assumes that the registered filter key is on the &flt parameter of the querystring.<table>
<tr><td>NetQuarry.IAppContext cxt</td><td>The application context</td></tr>
<tr><td>HttpRequest req</td><td>The HTTP Request object.</td></tr>
<tr><td>out SavedFilter filter</td><td>An out parameter. The SavedFilter, if any. This may include a filter description.</td></tr>
</table></td></tr>
</table>