

NetQuarry

**The Enterprise Application Software Development Platform
for Microsoft.NET**

IssueTrak Tutorial – 3 – Extensions and Tasks

Table of Contents

Table of Contents.....	2
Purpose of this document	5
Generated Objects	5
Typed Mappers	6
Picklist Enumerations	8
Session Properties	9
Defining Session Properties	9
Declaring Preference Inheritance Hierarchy	10
Generating the Objects from Metadata.....	12
When to Generate Code?	12
Steps to Successful Code Generation	13
What Might Cause Code Generation to Fail?	14
Namespaces	15
The Common DLL	16
Startup.cs.....	16
Generated Sub Folder	16
Individual.cs	17
Session.cs	18
ExtensionBase.cs	20
Attaching the Application Extension	21
Mapper Extensions	22
Individual Extension	23
Creating the Extension	23
Register the Extension in the Studio	25
Attach the Extension to the Individual Mapper.....	26
Override and Handle Events	27
Extension in Action	31
Debugging the Extension	32
Adding Real Functionality	33
Issue Extension.....	37
Create a RowBeforeInsert Handler.....	38
Create a MapperBeforeLayout Handler	39
Create a RowBeforeUpdate Handler	40
Create a RowAfterInsert Handler	41
Create a RowAfterUpdate Handler	42
Create the Email Notification Handler.....	43
Issue Extension as a Typed Mapper Extension.....	46
Create a RowBeforeInsert Handler.....	48
Create a MapperBeforeLayout Handler	49
Create a RowBeforeUpdate Handler	50
Create a RowAfterInsert Handler	51
Create a RowAfterUpdate Handler	52
Create the Email Notification Handler.....	53
Page Extensions.....	55
Issue Console Page Extension.....	55

Create a ConsolePaneBeforeLayout Handler	56
Create a ConsolePaneBeforeRequery Handler	58
Create a ConsolePaneAfterRequery Handler	59
Issue Wizard Page Extension.....	60
Create a WizardBeforePageLoad Handler	61
Create a WizardNext Handler.....	62
Create a WizardPrevious Handler	64
Create a WizardCancel Handler.....	65
Manipulating Sets of Data	66
Making Bulk Updates to Issues	66
Creating the Action Menu	67
Handling the Action Menu Command event	70
Create the Issue Bulk Modify Wizard	71
Fix the Bulk Modify Wizard UI.....	74
Make Bulk Modify Wizard Co-Exist with Other Wizards	76
Performing the Bulk Modify	80
Scheduled Tasks	83
Create the Open Issues Due in 2 Day Task	84
Register the task in the Studio.....	86
Define the Task in the Studio	87
Create a Mapper for Identifying who to Notify	88
Create the Notification Template	89
Declare More Typed Mappers	90
Create the Notification Code in the Notification Object	91
Use the TaskRunner to test your code.	93
Preferences	94
Create the Preference Extension.....	95
Create Company Preferences	97
Create the Company Preferences Mapper.....	97
Create the Company Preferences Page.....	99
Associate the Preference Page to the Correct Preference hierarchy level.	100
Add the Link to Company Preferences.....	101
Tweaks to Preference Page	102
Create the Individual Preferences.....	104
Create the Individual "Late Bound" Picklist	104
Create the Individual Preferences Mapper	105
Create the Individual Preferences Page	108
Associate the Preference Page to the Correct Preference hierarchy level.	109
Create UI to Get To Individual Preferences Page	110
Set the Individual Name.....	112
Using Preferences.....	113
Preference Behavior.....	113
Using Preferences in Code.....	114
Using Preferences in Metadata	116
Asynchronous Requests	120
Immediate Save.....	121
Immediate Save on MiniDetail	121
Immediate Save on MiniList.....	122
Use Ajax to Lookup and Populate field on Client.....	123

Add JavaScript Function Handlers to Fields.....	123
Tell the Application Where the JavaScript File Lives	124
Create the Component Mapper Extension.....	125
Tweak the component mapper	126
Localizing Notification Message	127
Localizing Strings in the XTMIssue extension	128
This Completes the Tutorial.....	130

Purpose of this document

This document is a tutorial that explains how to build an application using the NetQuarry Enterprise Application Platform. The document is split into three parts.

Part 1 – Explains how the NetQuarry Platform works and how the platform should be installed and configured.

Part 2 – Takes you through a set of steps to create a functional bug tracking application. The steps will be dealt entirely with manipulating metadata to create the Issuetrak application.

Part 3 – Enhances the basic functionality of the IssueTrak application by adding complex business rules with C# extensions, creating scheduled tasks and debugging.

Generated Objects

The NetQuarry platform can take some of the metadata defined in the studio and generate strongly typed objects that can be referred to in code. From the mapper metadata, we generate TypedMapper objects. From standard picklist meta data we generate typed enumerations. From session properties, we generated typed session properties. We haven't mentioned Session Properties as yet as they are mainly for coding use, rather than metadata use, even though they are defined in metadata.

Typed Mappers

A typed mapper is a C# generic (template class) for a mapper. A typed mapper is generated (by the developer, never automatically in the build) for mapper objects that are NOT marked with the "SkipCodeGeneration" attribute. During the New Page wizard you might remember this option is displayed on the Choose the Mapper screen and is defaulted to checked.

Typically you always want a mapper to be code generated as you will then be able to take advantage of the generated class in the future. However, some mappers you may not want to generate. For example, mappers that are not based on views or tables in the database and are dynamically populated in code through a datatable.

To best show what the typed mapper gives is to show a simple code example with equivalent code for regular mapper and typed mapper.

```
//--- Creating a basic mapper
//--- Note: This code won't work it's just an example
Guid? individual_id = null;
IAppContext cxt = null;

//--- create a mapper
IMapper individualMapper = NetQuarry.Data.Mapper.CreateAndLoad("individual", cxt, 0);

//--- Get to the value of the individual_id field
string sIndividualID = EAPUtil.ToString(individualMapper.Fields["individual_id"].Value);
if (!sIndividualID.IsNullOrEmpty())
{
    //--- if there is a value then assign to the local variable
    individual_id = new Guid(sIndividualID);
}
else
{
    //--- or set a null value
    individual_id = (System.Guid?)null
}

//--- using the mapper as a Typed Mapper
Individual individualTypeMapper = Individual.Attach(individualMapper);
//--- retrieving the value of the individual_id field
individual_id = individualTypeMapper.individual_id;
```

You can see from the above example that accessing a field that contains a unique identifier value is much simpler in the typed mapper case than the basic mapper.

Here is how the typed mapper has created the accessor method for the individual field.

```
public System.Guid? individual_id
{
    get
    {
        object value = this.Fields.individual_id.Value;
        if (value is string)
        {
            string guidVal = value as string;
            return (string.IsNullOrEmpty(guidVal)) ? (System.Guid?)null : new Guid(guidVal);
        }
        else
            return (System.Guid?)value;
    }
    set { this.Fields.individual_id.Value = value; }
}
```

Picklist Enumerations

In a similar way, the platform converts standard picklists from the meta data into an enumeration. Here's an example of equivalent code with and without using picklist enums.

```
//--- this is OK comment but using hard coded value.  
//--- force the individual to change password on next login  
individualTypeMapper.attr_bits = 4;  
  
//--- little better in explicitly declaring a local variable to the right value  
const int forcePasswordChange = 4;  
individualTypeMapper.attr_bits = forcePasswordChange;  
  
//--- much better referring to generated attribute  
individualTypeMapper.attr_bits =  
(int)IssueTrak.Data.Picklists.user_attrs.force_password_change;
```

However there is even a better way. Note in the last way you had to cast the enumerated value to the typed int value of the underlying field.

We can override the typed mapper definition of the attr_bits property wrapper, typing the property to the picklist enum type.

```
/// <summary>Override the default bahavior of the attr_bits field</summary>  
public new IssueTrak.Data.Picklists.user_attrs attr_bits  
{  
    get { return ((IssueTrak.Data.Picklists.user_attrs)base.attr_bits); }  
    set { base.attr_bits = (int)value; }  
}
```

And with this we can directly assign the typed mapper's attr_bits to the enumerated picklist.

```
//--- perfect  
individualTypeMapper.attr_bits = IssueTrak.Data.Picklists.user_attrs.force_password_change;
```


Session Properties

Session properties are a way of associating inheritable/hierarchical property values to users. We typically refer to these property values as Preferences.

These preferences are classed as hierarchical because a user can inherit a base set of preference values from a company. For example, for people with "Users" role, you might want them to not see the Home dashboard by default but give them the ability to choose that they do want to see the dashboard when they login.

Later we'll show you how this could be achieved using preferences, through code.

Defining Session Properties

Creating a session property is very simple. In the Studio, click on the Session Properties link (near the bottom of the Studio Explorer tree).

Create the following Session Properties

Name	Settings	Notes
CompanyID	Name: CompanyID Type: String Category: Custom Attributes: DynamicOnly, GenEmbeddedFunc, SessionPersist	GenEmbeddedFun attribute will force the code generator to register the value of this preference as an embedded function. You've already come across these when setting the default value for a user_id field. In the New Page wizard the Mapper Options screen where you select fields representing who created a record. That field is set with a default value of !fnUserID(). The embedded function generated for one of these custom preferences has the name of the preference prefixed with !fn and suffixed with (). So here we are creating an embedded function !fnCompanyID(). Also note that these names ARE case sensitive.
IndividualID	Name: IndividualID Type: String Category: Custom Attributes: DynamicOnly, GenEmbeddedFunc, SessionPersist	Create an embedded function called !fnIndividualID().
UsersViewDashboard	Name: UsersViewDashboard Type: Boolean Category: Custom Attributes, DynamicOnly, SessionPersist DefaultValue: 0	SessionPersist means that we want the value of the preference to be persisted across sessions. We will use this preference later.

Declaring Preference Inheritance Hierarchy

In the studio, you also have to declare the preference hierarchy. This describes how the preference values are inherited from one level to the next. Typically this preference hierarchy is equivalent to an organization hierarchy.

Company->Division->Building->People

For the IssueTrak preference inheritance, we are only declaring preference hierarchy of

Company->Individual

In the studio, Go to Modules and create a new module "IssueTrak-preferences".

Click on the "Preference Levels" link. Just above "Session Properties"

Create a record with the following values

Name	Value	Notes
Module	IssueTrak-preferences	
Preference Level	company	The name of the hierarchical preference level
Session Instance Creator	CreateInstanceCompany	The name of a function that you will declare in your custom session class, located in the IssueTrak.Common.dll component.
Owner Key Name	company_id	The name of the primary key field in a 'preference mapper' for this preference level.
Owner Lookup Sql	SELECT company_id FROM company WITH(NOLOCK) WHERE {0}	A SQL statement that is used to lookup the key value of the object to which the preferences at that level are attached. The WHERE clause of the SQL uses the parameterized string syntax. The replaceable portion is replaced by a filter expression that the platform has created. It is expected that the filter expression will return a single row,
Attributes	NoParent	This is the top level preference in the hierarchy and has no parent preference level

and create an individual preference level

Name	Value	Notes
Module	IssueTrak-preferences	
Preference Level	individual	
Session Instance Creator	CreateInstanceIndividual	
Owner Key Name	individual_id	
Owner Lookup Sql	SELECT individual_id FROM individual WITH(NOLOCK) WHERE {0}	
Parent Lookup Sql	SELECT company_id FROM individual WITH(NOLOCK) WHERE individual_id={0}	Here you specify a SQL expression that lookups up the ID of the parent hierarchy from the current hierarchy. You must always have a way to get from a current hierarchy level to the parent hierarchy level,

For now, that is all we will look at for preferences. We will complete the implementation of preferences as we continue through this document.

Generating the Objects from Metadata

To generate the relevant objects is a simple task. If you have installed the NQ Links shortcuts on the task bar, you can execute a batch file to generate the code files. The shortcuts are found under "Code", "GenCode", "All". Or if you want to go there from file system, the batch files are located in

C:\NetQuarry\Customers\IssueTrak\Database\Metadata

Shortcut/Batch File	Generates	Notes
All/issues-gen-code.bat	all generated files	See below for all files generated
Mappers/issues-gen-type_mappers.bat	TypedMappers.cs	
Picklists/issues-gen-picklists.bat	PicklistEnums.cs	
Remote/issues-gen-remote.bat	RemoteMappers.cs	
Session/issues-gen-session.bat	Session.cs	

When to Generate Code?

Since the process of generating this code is a manual step by a developer, the question becomes when should you generate the code? Typically you only need to generate the code when you need to. When you are adding fields to tables and views and to the mappers and you want to refer to those field objects in code through the typed mapper interfaces.

Steps to Successful Code Generation

Having added fields to your schema and to your meta, you execute the batch file to generate your new objects. Then you compile your code against the new generated objects. All being well, the code compiles and tests. You then decide to check in your changes.

STOP!

Before you check in your regenerated code you must guarantee that you have successfully generated the code against all the latest metadata. The latest metadata means your latest and the latest that is currently checked in.

Here are the steps to successful code generation.

1. Get the latest schema scripts, metadata, and code
2. Save your metadata (check out your metadata first) and schema changes to disk.
3. Diff your metadata and schema changes – resolve differences.
4. Reload your database and metadata using the database update script. **CORRECT ANY ERRORS BEFORE CONTINUING**
5. Run the batch file to generate all the code objects. **CORRECT ANY ERRORS BEFORE CONTINUING**
6. Recompile your source code against the latest generated objects. **CORRECT ANY COMPILE ERRORS BEFORE CONTINUING**

Once you successfully execute all 6 steps, you can then safely check in all of the changes you've made.

What Might Cause Code Generation to Fail?

If your code generation process fails it's likely due to the following

Failed At	Reason
Step 4 Database Update	<p>Missing schema updates against updates to views</p> <p>Incorrect join syntax on views</p> <p>Missing a GO statement</p>
Step 5 Code Generation	<p>The mapper field metadata no longer matches the current schema due to changes in the underlying table or view. To solve, check the schema and correct.</p> <p>Last time since the code generation was run a mapper was added that cannot support code generation. To solve mark those mappers with the SkipCodeGeneration attribute.</p>
Step 6 Compilation	<p>A field is added to the mapper which has a name equivalent to a reserved word in C#. For example a field might be called "private". When the code generator runs it tries to create a field property called private that is public!</p> <pre>public System.Boolean private</pre> <p>Another situation is where a field on the mapper has exactly the same name as the mapper. The generated code then creates a class and a property accessor with the same name, and that is not allowed.</p> <p>To solve either of these issues you either change the schema name, or set a field property on the field to define a different key name specifically for code generation. Changing the schema name is the preferable solution.</p> <p>A data type of a field has been changed in the database. The code generation process analyzes the data types based on the underlying schema, rather than the data types defined by the metadata. – To solve this you MUST change the data type of the field in the underlying schema.</p> <p>Flavor modification (typically setting Include flavor) cause the field to NOT appear in the list of mapper fields when the code is generated. Code generation occurs with no flavor applied to the mapper, so fields that have explicit Include flavors will be excluded! To solve, you either have to work out a way to include the field for code generation, or change the code that references that field through typed mapper accessors, to use the general accessor methods through regular mapper object.</p>

Once you successfully solve all errors and complete all these steps you can check in the schema scripts, meta data and any code

Namespaces

The code generation associates the generated objects with specific namespaces. The TypedMappers and Picklists are associated with the IssueTrak.Data namespace. The Session object is associated IssueTrak.Common namespace.

Although there are separate namespaces, we host all three sets of objects under one component dll called the Common DLL. We do this as a way of eliminating complexities of referencing objects from one namespace to another and vice versa.

The Common DLL

All applications will have a component called a "Common" dll. This DLL typically performs the following tasks.

1. Handling Application extension events for modifying authentication and application load behavior
2. Container for holding references to generated objects from metadata
3. Container for storing implementation code of typed mappers
4. Managing Sessions and Session Properties

A Common dll is provided with this Tutorial that you can refer to, but will also utilize and build upon.

If you have not already done so, execute the batch file to generate all the code objects.

In Visual Studio, open the project file

C:\NetQuarry\Customers\IssueTrak\Source\Common\IssueTrak.Common.csproj

Startup.cs

Open the startup.cs file. The Startup class is an extension that responds to events fired by the NetQuarry application object. Currently this has two main functions.

To instantiate a session object for the application once the user is authenticated.

Once the application has completely loaded all of its objects (based on roles/policies), initialize the session object with data about the user (loading the user's preferences).

Generated Sub Folder

The folder that stores the generated code objects. You can inspect what is generated, but you must never manually edit these files and check them in. That is because they will be overwritten the next time the code is generated and checked in.

Individual.cs

The implementation of the individual typedmapper object from the typed mapper generic.

```
public class Individual : IssueTrak.Data.Generated.individual<Individual>
```

In this bare bones example, we have overridden two fields that have standard picklists associated with them to provide fully typed fields.

We will be adding more code and functionality to this class.

Session.cs

The implementation of the Session object. This object manages the loading of session properties for the user during initial login and for the loading of preferences to support hierarchical preferences.

A little earlier we declared in metadata that preferences were a hierarchy of two levels, Company and Individual. In that metadata we declared two functions to load preferences for that specific hierarchical level.

CreateInstanceCompany and CreateInstanceIndividual. Add these functions to the session.cs file as shown below

```
/// <summary>
/// Creates and loads a session for a particular company.
/// </summary>
/// <param name="appCxt">The application context object.</param>
/// <param name="companyID">The company_id value.</param>
/// <returns>A session object with the specified company preferences loaded.</returns>
new public static Session CreateInstanceCompany(IAppContext appCxt, string companyID)
{
    ///--- create an new instance of the session object
    Session session = (Session)NetQuarry.Session.CreateInstance<IssueTrak.Common.Session>(appCxt);
    ///--- load the preferences for this company
    session.LoadPreferences(EAPUtil.ToString(companyID), PreferenceLevel.COMPANY);
    session.CompanyID = companyID;
    return (session);
}

/// <summary>
/// Creates and loads a session for a particular individual record.
/// Overloads the base user session created from the user_name
/// </summary>
/// <param name="appCxt">The application context object.</param>
/// <param name="individualID">The individualID value.</param>
/// <returns>A session object with the specified individual preferences loaded.</returns>
new public static Session CreateInstanceIndividual(IAppContext appCxt, string individualID)
{
    ///--- determine the company_id for this individual
    string companyID = EAPUtil.ToString(appCxt.DataDB.DBLookup("company_id", "individual",
    string.Format("individual_id={0}", EAPUtil.AnsiQuote(individualID))));
    ///--- create a populated session object for the individuals company from the company
    preferences
    Session session = CreateInstanceCompany(appCxt, companyID);
    ///--- load the individual's preferences on top of those
    session.LoadPreferences(individualID, PreferenceLevel.INDIVIDUAL);
    session.IndividualID = individualID;
    return (session);
}
```

Basically when we want to load the preferences for a company we ask the platform to load the preferences at the specified level (COMPANY) with the specified key.

When we want to load the preferences for a user/individual, we first determine the company associated with that user/individual. Load the preferences for that company and then load on top of that, the preferences for the user/individual.

ExtensionBase.cs

Declare some classes that derive from the NetQuarry Platform Extensions. These provide ways to add extra functionality to the extension objects. In this case, to provide a simple accessor property to the current IssueTrak.Common.Session object.

Make sure you can compile the IssueTrak.Common.dll project.

Attaching the Application Extension

The IssueTrak.Common.dll component implements an application extension. We have to register this extension into the Studio and then associate the extension with the application.

In the Studio, go to Modules and create a new module called IssueTrak-common,

Then go to the Extensions list under Components and create a new extension with the following values.

Name	Value	Notes
Module	IssueTrak-common	
Name	Startup	
Component Type	Extension	
Component Name	IssueTrak.Common.Startup	The name of the component. It is IMPERATIVE that you spell this correctly, exactly matching the name in the code AND remembering case sensitivity.
Assembly Name	IssueTrak.Common.dll	The dll that the component is located in
Assembly Path	C:\NetQuarry\Customers\IssueTrak\Source\Common\bin\debug	The path to the locally compiled dll for development
Assembly Path Prod	%NQROOT%\Apps\IssueTrak\bin	The path where the component would be installed in a production environment

Go to the list of Applications in the Studio.

Click on the Extensions subform under the IssueTrak application.

Add the IssueTrak.Common.Startup extension.

Mapper Extensions

About 50% of the coding you will perform will likely be in mapper extensions with the other 50% will be in the typed mapper objects. In this section we will create a new mapper extension and hook it into the application and debug some of the events that are fired.

We will create only a "generic" (where generic means not based on a typed mapper) extension to provide business rules for Individuals. With the generic extension we are able to attach the extension to different mappers.

Next, we will create an extension for managing Issues. Initially we will create the extension as a "generic" extension, but then we will also create another extension for Issues based on a typed mapper object with exactly the same functionality, so you can directly compare the difference in code between the two approaches.

We will then add preference support to the application

Individual Extension

Creating the Extension

Close Visual Studio to close the IssueTrak.Common project. Now navigate one folder up and open the solution file

C:\NetQuarry\Customers\IssueTrak\Source\IssueTrak.sln

There's just one project hosted within it, the IssueTrak.Common project. Right click on the Solution items and "Add -> New Project".

Set the following options

Item	Value	Notes
Project Type	Visual C#	
Template	Class Library	
Name	Individual	
Location	C:\NetQuarry\Customers\IssueTrak\Source\Extensions	

Click OK

Now change the identified value as follows

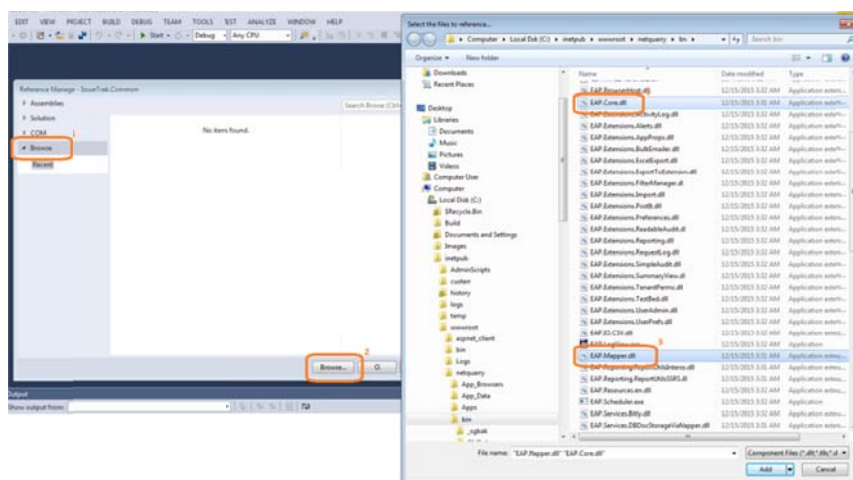
In the solution explorer

Individual -> Issuetrak.Extensions.Individual

Class1.cs -> XIndividual.cs

Add references to...

NetQuarry Core (EAP.Core.dll), NetQuarry Data Binding (EAP.Mapper.dll) browse to %NQBIN%



IssueTrak.Common.dll (browse to
C:\NetQuarry\Customers\IssueTrak\Source\Lib\IssueTrak.Common.dll)

Drag the project.build file from IssueTrak.Common to IssueTrak.Extensions.Individual

Open the project.build file in the IssueTrak.Extensions.Individual component.

Change the solution value to "IssueTrak.Extensions.Individual"

Drag the AssemblyInfo.cs file from IssueTrak.Common \Properties to
IssueTrak.Extensions.Individual \Properties, overwriting when prompted.

Open the AssemblyInfo.cs file in the IssueTrak.Extensions.Individual\Properties.

Change the Assembly Description to "IssueTrak Individual Extension"

Right click on the IssueTrak.Extensions.Individual project and choose Properties

Change the Assembly Name to IssueTrak.Extensions.Individual

Change the Default Namespace to IssueTrak.Extensions (save and close the Properties)

In the XIndividual.cs file, use this code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NetQuarry;
using NetQuarry.Data;
using IssueTrak.Common;
using IssueTrak.Data;
using System.Data;
using System.Web;

namespace IssueTrak.Extensions
{
    public class XIndividual : IssueTrak.Extensions.ExtensionBase
    {
    }
}
```

Having made these changes you should be able to successfully compile the
IssueTrak.Extensions.Individual object.

Register the Extension in the Studio

Go to the Extension list in the Studio

Create a new extension with the following values.

Name	Value	Notes
Module	individual	
Name	XIndividual	
Component Type	Extension	
AssemblyPathProd	%NQROOT%\Apps\IssueTrak\bin	

Move off the row to save and back to the XIndividual row.

Click on the button in the Assembly Name field to browse for the Individual Extension and the XIndividual component. This will fill in the remaining fields with the necessary information.

Attach the Extension to the Individual Mapper

Go to the Mappers list (Alt+M) and select the individual mapper. In the extensions list, add the extension you just registered.

Add the same extension to the individual_import mapper.

Log in to the application and navigate to the Individual list page. Then click on an area of whitespace and press F8 to display the page debug information. You should see the individual extension has been attached to the mapper.

Override and Handle Events

Go back to your solution in Visual Studio, and the XIndividual.cs file.

Inside the class XIndividual, simply handle an event that's available by typing "override <space>" and then a type ahead list of all the available events pops up.

Create event handlers for the following events.

MapperBeforeLayout, MapperBeforeRequery, MapperExecSQL, RowAfterInsert, RowAfterUpdate, RowBeforeInsert, RowBeforeUpdate, RowCurrent, RowSetDefaults

In each of these event handler stubs, you will automatically generate a call to the base method. Simply delete the calls to the base object, leaving the event handler stubs with an empty body.

MapperBeforeLayout

In the lifecycle of event handlers, this event is the last one that is fired where any changes you make to a field, or mapper are effective. After this event has been handled you can still make some UI changes, but you are limited in scope as to what changes are still effective. There are some field UI changes you can make later in the event life cycle, but this event supports modification of all field parameters.

Add the following code to make the email_address field hidden by default in the list presented to people with the 'Users' role.

```
if (sender.Application.UserContext.HasProfile("Users"))
{
    FieldFilterOptions ffo =
    (FieldFilterOptions)sender.Fields["email_address"].Properties.GetIntValue("FilterOptions");
    ffo |= FieldFilterOptions.DefaultHideInList;
    ffo &= ~FieldFilterOptions.DefaultShowInList;
    sender.Fields["email_address"].Properties.Add("FilterOptions", ffo);
}
```

MapperBeforeRequery

This event is fired just before construction of the mapper's SQL. Typically you can use this event to change the filtering of the mapper.

Add this code to make the individual list filter out users who are system admins when the logged in user does not have an allowed policy to see system admins.

```
if (!sender.Application.HasPolicy("CanViewSysAdmin", false))
{
    MapperFilter mf = new MapperFilter("CanViewSysAdmin", "user_id NOT IN (SELECT user_id FROM
user_roles WITH(NOLOCK) WHERE role_nm='System Admin')");
    sender.Filters.Add(mf);
}
```

To support this code, go into the Studio and navigate to the "Policies" list under Permissions and add the following Policy

Item	Value	Notes
Module	IssueTrak-roles	
Name	CanViewSysAdmin	
Description	Allow viewing sys admin individuals	

In the Permissions subform, uncheck the User role and Manager role. Then click the Save button.

MapperExecSQL

This event is fired just before the mapper's SQL is executed against the database. You can completely replace the executed SQL if you wish.

Add this code to the extract the executing SQL. This rebuilds the executing SQL and sets it back.

```
if (e.StatementType == ExecSQLArgs.ExecuteStatementType.Select)
{
    string theSQL = e.SQL;
    string selectSQL = EAPUtil.ToString(sender.Exec(MapperExecCmds.GenerateSelectSQL, 0));
    string whereSQL = e.RowRequeryFilter.IsNullOrEmpty() ? sender.Filters.ToString() :
e.RowRequeryFilter;
    string orderSQL = EAPUtil.ToString(sender.Exec(MapperExecCmds.OrderByClause, 0));
    string newSQL = string.Format("{0} FROM {1} WITH(NOLOCK) ", selectSQL, sender.View);

    if (!whereSQL.IsNullOrEmpty())
    {
        newSQL += "WHERE " + whereSQL;
    }
    if (!orderSQL.IsNullOrEmpty())
    {
        newSQL += "ORDER BY " + orderSQL;
    }
    e.SetSQL(newSQL, e.StatementType);
}
```

RowAfterInsert

Fired after a record has been successfully inserted into the database. This event would add business rules that should fire once a record has been successfully inserted.

Add this code to send an email to the user welcoming them to IssueTrak once a new user has been created.

```
string email_address = sender.Fields.GetStringValue("email_address", null,
FindBehaviour.ErrIfNotFound);
sender.Send(email_address, "email-welcome-to-issuetrak");
```

RowAfterUpdate

Fired after an existing record has been successfully updated in the database. This event would add business rules that should fire once a record has been successfully updated.

Add this code to send an email to a user with the same welcome template if the users user id has been changed.

```
if (sender.Fields["user_id"].Dirty)
{
    string email_address = sender.Fields.GetStringValue("email_address", null,
FindBehaviour.ErrIfNotFound);
    sender.Send(email_address, "email-welcome-to-issuetrak");
}
```

RowBeforeInsert

Fired before a new record is inserted into the database. Here you should validate the input parameters and pull in additional data that can be derived from the data about to be inserted.

Add this code to force the user to provide a gender for a new user. Ideally you would do this in the metadata by setting the field's "Required" attribute

```
if (sender.Fields.GetIntValue("gender_id", 0, FindBehaviour.ErrIfNotFound) == 0)
{
    e.Error("You must provide a gender for this individual");
}
```

RowBeforeUpdate

Fired before an existing record is updated in the database. Here you should validate the input parameters and pull in additional data that can be derived from the data about to be updated.

Add this code to set the address field from the constituent parts.

```
if (sender.Fields["address"].Dirty)
{
    string fullAddress = string.Format("{0}\r\n{1}, {2} {3}",
        sender.Fields.GetStringValue("address", null, FindBehaviour.ErrIfNotFound),
        sender.Fields.GetStringValue("city", null, FindBehaviour.ErrIfNotFound),
        sender.Fields.GetStringValue("state", null, FindBehaviour.ErrIfNotFound),
        sender.Fields.GetStringValue("postal_code", null, FindBehaviour.ErrIfNotFound));

    sender.Fields.SetValue("full_address", fullAddress);
}
```

RowCurrent

Fired when the mapper cursor moves to the next row. On a detail screen this fires once because the data for only one record is displayed. On a page rendering a list, it is fired for each row displayed in the list.

Add this code to upper case the last name of the user, if the user is a System Admin. Note that this is an EXPENSIVE operation to perform for each row. It's only showed as an example and NOT to be blindly followed as a recommended pattern.

```
if (sender.Database.DBExists("user_roles", string.Format("{0} AND role_nm='System Admin'",
sender.Fields["user_id"].BuildFilter()))
{
sender.Fields.SetValue("last_name",
    sender.Fields.GetStringValue("last_name", null, FindBehaviour.ErrIfNotFound).ToUpper(),
    SetValFlags.DoNotMakeDirty,
    FindBehaviour.OkIfNotFound);
}
```

RowSetDefaults

Fired when the mapper is displaying a new page. This gives you a chance to set or override default values. After this event is handled, default values are assigned to the values of the fields.

Add this code to set the default value for the password and company on a new individual.

```
sender.Fields["password"].DefaultValue = "password1";
sender.Fields["company_id"].DefaultValue = Session.CompanyID;
e.Result = ExtResults.DataChanged;
```

Extension in Action

Now compile your extension. If you have a problem compiling due to file being locked in use by another process, it's either the worker process (that you need to recycle) or due to the NQ Studio having a lock on the file after you have browsed and added components.

To solve that, use the NQ Links shortcut under tools "ResetAppPool.bat" to recycle the app pool, and if that doesn't work, close and re-open the NQ Studio.

Then log in to the application as an admin user and navigate to the individual list.

You can see some individuals have an upper case last name. And you can navigate to those users to confirm they are system admins.

You can create a new individual and confirm that the password and company field is defaulted.

Continue creating a new user. If you don't provide a Gender, you get an error. Provide a gender and save. You should then get an email notification delivered to your MailTrap account (or popup on your local SMTP4Dev app).

Now edit the user you just created. Change the user_id to force an email to be delivered, and set an address so the full_address field is populated. If you make other changes to an individual without touching user_id, or the address field, nothing happens.

Logon as a user with a Manager role to verify that they cannot see individuals who are administrators. There should be no individuals listed where their last name is capitalized.

Debugging the Extension

If you wish to debug the extension, you can do so by launching the debugging tools in Visual Studio by attaching to the w3wp.exe process. There is one w3wp.exe process per application pool, so if your development machine has multiple app pools, then you may have to simply guess the correct process to attach to.

Once you have attached to the correct worker process, you can set breakpoints and step through the code inspecting the underlying objects.

Adding Real Functionality

What we've got in the extension is a set of example code. One thing we should do is ensure that when we create a user, that user has a role created for them by default. We'll choose the Users role as the default.

We'll add some code in the RowBeforeInsert and RowAfterInsert extension handlers for the XIndividual extension.

First we'll create the function to ensure the user_role will always be created when necessary.

```
private void SetDefaultRole(IMapper sender)
{
    if (sender.Fields.GetStringValue("user_role", null,
    FindBehaviour.OkIfNotFound).IsNullOrEmpty())
    {

    }
}
```

We're using the helper method on the fields collection to return a value typed to a string. However, we have set the FindBehavior flag to OkIfNotFound. This means that if the field does not exist on the mapper, the platform won't throw an error and instead will return a default value, which we've set as null. If we get a null value back from the call, it actually means one of two things. The field exists and the value is not set, or there is no field. So we have to further differentiate between those two situations.

This is because this XIndividual mapper extension is attached to two different mappers. The individual mapper (where there is no user_role field) and the individual_import mapper (where there is a user_role field).

If the field exists, it must be the individual_import mapper so we have to set the field value to "User" BEFORE saving. The set up of the mapper will ensure the user_role record is created. We know this because we imported a set of individuals.

If the field does not exist, it must be the individual mapper and we have to manually create a user_role record for this user AFTER the user record has been created.

The code now becomes

```
private void SetDefaultRole(IMapper sender)
{
    if (sender.Fields.GetStringValue("user_role", null,
    FindBehaviour.OkIfNotFound).IsNullOrEmpty())
    {
        if (sender.Fields.ContainsKey("user_role"))
        {
            here //--- the field exists so we set the value. It must be empty otherwise we wouldn't get
            sender.Fields.SetValue("user_role", "Users");
        }
    }
}
```

```

else
{
    ///--- need a way to insert a new record into user_roles table
    ///--- ENTER YOUR FOUR METHODS HERE!
}
}
}

```

There are a number of ways to perform the insert of the user_role record. We'll cover four possible ways.

Adding a Record with SQL

```

string sInsert = string.Format(@"INSERT INTO user_roles
                                (user_role_id, user_id, role_nm)
                                VALUES (NEWID(), {0}, 'Users')",
                                EAPUtil.AnsiQuote(sender.Fields.GetStringValue("user_id", null,
FindBehaviour.ErrIfNotFound)));
sender.Database.Execute(sInsert, "IssueTrak.Extensions.XIndividual.SetDefaultRole");

```

Adding a Record using SQLInserter

The NetQuarry Platform comes with a set of classes that help you execute a SQL statement without having to explicitly write SQL. There is a SQLUpdater, SQLInserter and SQLDeleter. We are inserting a record so we'll use the SQLInserter for this task.

```

SQLInserter si = new SQLInserter(sender.Database, "user_roles");
si.AddColumn("user_role_id", EAPUtil.NewGuid(), System.Data.OleDb.OleDbType.Guid);
si.AddColumn("user_id", sender.Fields.GetStringValue("user_id", null,
FindBehaviour.ErrIfNotFound), System.Data.OleDb.OleDbType.VarChar);
si.AddColumn("role_nm", "Users", System.Data.OleDb.OleDbType.VarChar);
si.Execute("IssueTrak.Extensions.XIndividual.SetDefaultRole");

```

Adding a Record using a Mapper

```

using (IMapper mapUR = NetQuarry.Data.Mapper.CreateAndLoad("user_roles", this.Application, 0))
{
    mapUR.MoveNew();
    mapUR.Fields.SetValue("user_id", sender.Fields["user_id"].Value);
    mapUR.Fields.SetValue("role_nm", "Users");
    mapUR.Save();
    mapUR.Close();
}

```

This example shows a pattern that you must follow religiously. When you create a mapper in your code, you must explicitly Close it. Yes even though the Mapper is opened in a using block which should normally dispose.

Adding a Record Using a Typed Mapper

Before we describe the method through the typed mapper, we have to create a typed mapper object for the user_roles mapper.

In Visual Studio, go back to the IssueTrak.Common project. Add a NEW ITEM of template type "Class" and call it Users.cs

Make sure the content of the Users.cs file looks as follows

```
using System;
using System.Collections.Generic;
using System.Text;
using NetQuarry.Data;
using NetQuarry;
using IssueTrak.Common;
using IssueTrak.Data;

namespace IssueTrak.Data
{
    public class Users : IssueTrak.Data.Generated.users<Users>
    {
    }

    public class UserRoles : IssueTrak.Data.Generated.user_roles<UserRoles>
    {
    }
}
```

Compile the Common Project

Now go back to the individual extension to enter the code for Typed Mapper record creation.

```
using (UserRoles tmUR = UserRoles.OpenNew(this.Application))
{
    tmUR.user_id = sender.Fields.GetStringValue("user_id", null, FindBehaviour.ErrIfNotFound);
    tmUR.role_nm = "Users";
    tmUR.Save();
    tmUR.Close();
}
```

The last step is to call the SetDefaultRole function.

The SetDefaultRole function performs a different task depending on which mapper is performing the insert. The different mapper implies which process is occurring. If the mapper key is individual_import, then the process is for an import. If the mapper key is individual, then the process is for a manual individual creation.

For an import, the user_role field exists on the mapper and is available. Therefore we can directly set the default role in the before insert. The platform will handle the save of the default role.

For a manual creation (through page, issue!detail), the user_role field doesn't exist, so we have to wait until the after insert handler before we can add the default role record.

After the existing code in the body of the RowBeforeInsert handler, add the following code.

```
if (sender.Key.EqualsCI("individual_import"))
{
    SetDefaultRole(sender);
}
```

In the RowAfterInsert handler (covering the issue!detail insert case), add the following code.

```
if (sender.Key.EqualsCI("individual"))
{
    SetDefaultRole(sender);
}
```

You can now compile the Individual Extension.

Log into the application and create a new Individual. If you entered all four sets of code in the SetDefaultRole function, you should have four user_role records for your user!

Issue Extension

Follow the steps described for the Individual Extension, to create the Issue extension

As a reminder, you first create the Extension as Issue in Extensions folder then change the project name to IssueTrak.Extensions.Issue. Add references to Platform and Common assemblies, Copy AssemblyInfo.cs and project.build from the individual extension and modify. Change the properties of the project. Change the name of the cs file in the extension and modify the content to create a public class derived from the issuetrak extension base.

In this extension we are going to create the following business rules.

If the milestone is not specified explicitly, then set the milestone to the next nearest milestone in the future based on the selected project.

If the issue is not assigned to a specific user when the issue is saved, then choose the assigned user from the user associated with the selected component.

To support these two requirements for creating a new issue, we have to TURN OFF the Required attributes on the milestone_id and assigned_user fields. This will allow a user to leave the fields blank and let the extension handle setting the appropriate default issue.

However, on an existing record, we want to force back the required attributes for these fields.

Set the assigned user to the creator of the issue when an issue is resolved (unless a user has explicitly selected a different user in the resolution wizard)

Set the assigned user to the default assignee based on the selected component if an issue is rejected (unless a user has explicitly selected a different user in the rejection wizard)

Send an email to the assignee when: The status of the issue changes or the assigned user has changed but only if the assigned user is not the same as the current user.

Create a RowBeforeInsert Handler

In your Issue extension, create an event override for RowBeforeInsert

We will add the following code to pull in default values into the issue if none are set for assigned user and milestone.

```
//--- set the default assignee
if (!sender.Fields["assigned_user"].Dirty)
{
    sender.Fields["assigned_user_id"].Value = sender.Database.DBLookup("user_id", "component",
sender.Fields["component_id"].BuildFilter());
}

//--- set the default milestone
if (!sender.Fields["milestone_id"].Dirty)
{
    string where = string.Format("{0} AND milestone_due_date > GETDATE() ORDER BY
milestone_due_date", sender.Fields["project_id"].BuildFilter());
    sender.Fields["milestone_id"].Value = sender.Database.DBLookup("milestone_id", "milestone",
where);
}
```

You can compile the project and test it.

Remember, however, that you must first register the component into the studio as an extension. (Associate the extension with the issue module) and then attach this extension to the Issue mapper. Follow the instructions per the Individual mapper extension.

Create an issue and don't set a specific assigned user, or milestone. After the save is complete, both will be set to the appropriate default value. Next create an issue but explicitly set an assigned user and milestone. After the save, they will be the same as you selected.

Create a MapperBeforeLayout Handler

Add a Reference in the Issue project to the System.Web assembly. Then add the statement

```
using System.Web;
```

to the top of the XIssue.cs file.

Now create an event override for MapperBeforeLayout.

We will add the following code to enforce requiredness of the assigned_user and milestone fields on existing records

```
//--- Check to see if there is a http context. Sometimes your code runs under context of
scheduler
//--- and in that case the HttpContext.Current object is null
if (HttpContext.Current != null)
{
    //--- if the "req" query string parameter is not to a "new" page then make the fields
    required.
    string req = HttpContext.Current.Request[NetQuarry.ReqParams.Request];
    if (req != "new")
    {
        sender.Fields["assigned_user"].Attributes |= FieldAttrs.Required;
        sender.Fields["milestone_id"].Attributes |= FieldAttrs.Required;
    }
}
```

To see the effects of this you can compile the extension and then go to an existing issue, and "edit" it. Both the Assigned User and Milestone fields are required. Start to create a new issue and both fields are not required.

Create a RowBeforeUpdate Handler

Create an event override for RowBeforeUpdate

We will add the following code to set the assigned user to the creator of the issue when an issue is resolved and set the assigned user to the default assignee based on the selected component if an issue is rejected. Also take into account that we should not default these values if the user has made an explicit assignment in the workflow wizard

```
//--- since status can only be changed by workflow wizards (at the moment)
//--- we can ask the mapper what page it's on to determine what to do.
if (!sender.Fields["assigned_user"].Dirty)
{
    switch (sender.MOP)
    {
        case "issue!wiz_resolve":
            sender.Fields["assigned_user_id"].Value = sender.Fields["created_by_id"].Value;
            break;
        case "issue!wiz_close":
            sender.Fields["assigned_user_id"].Value = null;
            break;
        case "issue!wiz_reject":
            sender.Fields["assigned_user_id"].Value = sender.Database.DBLookup("user_id",
"component", sender.Fields["component_id"].BuildFilter());
            break;
        case "issue!wiz_reopen":
            sender.Fields["assigned_user_id"].Value = sender.Database.DBLookup("user_id",
"component", sender.Fields["component_id"].BuildFilter());
            break;
    }
}
```


Create a RowAfterInsert Handler

Create an event override for RowAfterInsert.

We will add the following code to send an email notification when an issue is created. Because there is some commonality between this handler and RowAfterUpdate handler, we'll make a function call to some code that handles all the send scenarios.

```
SendEmailNotification(sender, true);
```

Create a RowAfterUpdate Handler

Create an event override for RowAfterUpdate.

Again the body of this function is a call to another function

```
SendEmailNotification(sender, false);
```

Create the Email Notification Handler

Now create the handler function to send an email notification to the assigned user.

```
private void SendEmailNotification(IMapper sender, bool isCreated)
{
    string notifyTemplate = null;
    //--- we said only to send an email notification if the assigned_user_id has changed
    //--- and the assigned_user_id is not the same as the currently logged in user.
    if ((sender.Fields["assigned_user_id"].Dirty || sender.Fields["status_id"].Dirty) &&
        !EAPUtil.ToString(sender.Fields["assigned_user_id"].Value).EqualsCI(this.Application.UserContext.ID))
    {
        switch (sender.MOP)
        {
            case "issue!wiz_resolve":
                notifyTemplate = "issue-resolved";
                break;
            case "issue!wiz_close":
                //--- there is no notification when an issue is closed
                break;
            case "issue!wiz_reject":
                notifyTemplate = "issue-rejected";
                break;
            case "issue!wiz_reopen":
                notifyTemplate = "issue-reopened";
                break;
            default:
                notifyTemplate = isCreated ? "issue-created" : "issue-assigned";
                break;
        }
        //--- check for some error conditions and throw a descriptive error if necessary
        if (!notifyTemplate.IsNullOrEmpty())
        {
            string to = sender.Fields.GetStringValue("email_address", "", FindBehaviour.ErrIfNotFound);
            if (!to.IsNullOrEmpty())
            {
                //--- send an email to the assigned user with the given template
                sender.Send(to, notifyTemplate);
            }
            else
            {
                throw new EAPException(string.Format("Could not determine an email address for the assigned user: {0} on issue: {1}. An email notification was not sent", sender.Fields["assigned_user"].Value, sender.Fields["issue_number"].Value));
            }
        }
        else
        {
            if (!sender.MOP.EqualsCI("issue!wiz_close"))
            {
                throw new EAPException(string.Format("Unable to determine notification template for issue {0}. An email notification was not sent", sender.Fields["issue_number"].Value));
            }
        }
    }
}
```

You can compile the Issue extension, but you cannot test it yet. We have to create the five email templates referred to in the code. To simplify this task, we will re-use the same physical html file that was created by the application wizard for the issue!main page.

In the NetQuarry Studio, create the required five templates (either from scratch or duplicating an existing template) with the following settings.

Template	Settings	Notes
issue-created	Module: issue Template Name: issue-created Type: File Category: Email FileName: IssueTrak\issue\issue-detail-layout.html Subject: Issue {{issue_number}} has been created and assigned to you.	
issue-assigned	Module: issue Template Name: issue-assigned Type: File Category: Email FileName: IssueTrak\issue\issue-detail-layout.html Subject: Issue {{issue_number}} has been assigned to you.	
issue-resolved	Module: issue Template Name: issue-resolved Type: File Category: Email FileName: IssueTrak\issue\issue-detail-layout.html Subject: Issue {{issue_number}} has been resolved and awaiting verification by you.	

Template	Settings	Notes
issue-rejected	Module: issue Template Name: issue-rejected Type: File Category: Email FileName: IssueTrak\issue\issue-detail-layout.html Subject: Issue {{issue_number}} has been rejected and assigned back to you.	
issue-reopened	Module: issue Template Name: issue-reopened Type: File Category: Email FileName: IssueTrak\issue\issue-detail-layout.html Subject: Issue {{issue_number}} has been re-opened and assigned to you.	

Now you can login to the application to test the issue workflow and email notifications.

Issue Extension as a Typed Mapper Extension

Now we will create the same extension logic using the typed mapper based extension

In Visual Studio, go back to the IssueTrak.Common project. Add a NEW ITEM of template type "Class" and call it Issue.cs

Make sure the content of the Issue.cs file looks as follows

```
using System;
using System.Collections.Generic;
using System.Text;
using NetQuarry.Data;
using NetQuarry;
using IssueTrak.Common;

namespace IssueTrak.Data
{
    public class Issue : IssueTrak.Data.Generated.issue<Issue>
    {
        /// <summary>Override the default behavior of the status_id field</summary>
        public new IssueTrak.Data.Picklists.issue_status status_id
        {
            get { return ((IssueTrak.Data.Picklists.issue_status)base.status_id); }
            set { base.status_id = (int)value; }
        }

        /// <summary>Override the default behavior of the severity_id field</summary>
        public new IssueTrak.Data.Picklists.issue_severity severity_id
        {
            get { return ((IssueTrak.Data.Picklists.issue_severity)base.severity_id); }
            set { base.severity_id = (int)value; }
        }

        /// <summary>Override the default behavior of the priority_id field</summary>
        public new IssueTrak.Data.Picklists.issue_priority priority_id
        {
            get { return ((IssueTrak.Data.Picklists.issue_priority)base.priority_id); }
            set { base.priority_id = (int)value; }
        }

        /// <summary>Override the default behavior of the category_id field</summary>
        public new IssueTrak.Data.Picklists.issue_category category_id
        {
            get { return ((IssueTrak.Data.Picklists.issue_category)base.category_id); }
            set { base.category_id = (int)value; }
        }
    }
}
```

Having created the typed mapper object which we shall shortly use to create the typed mapper extension, we also have a container for storing useful utility functions related to issues. As we re-enter the code for the typed mapper based extension, we will also move out some "useful" functions into the typed mapper object and call those functions from the typed mapper extension.

Compile the Common Project

Now go back to the Issue extension and create the Typed Mapper extension class in the XIssue.cs file as follows.

```
public class XTMIssue : IssueTrak.Extensions.TypedExtensionBase<IssueTrak.Data.Issue>
{
}
```

Now create the same event handlers as the previous issue extension.

Create a RowBeforeInsert Handler

In your Issue extension, create an event override for RowBeforeInsert

Then add this code

```
//--- set the default assignee
if (!sender.Fields.assigned_user.Dirty)
{
    sender.SetAssigneeFromComponent();
}
```

```
//--- set the default milestone
if (!sender.Fields.milestone_id.Dirty)
{
    sender.SetMilestoneFromProject();
}
```

In the Issue Typed Mapper class (in IssueTrak.Common) add the code for the two functions

```
public void SetAssigneeFromComponent()
{
    this.assigned_user_id = EAPUtil.ToString(this.Database.DBLookup("user_id", "component",
this.Fields.component_id.BuildFilter()));
}

public void SetMilestoneFromProject()
{
    string where = string.Format("{0} AND milestone_due_date > GETDATE() ORDER BY
milestone_due_date", this.Fields.project_id.BuildFilter());
    this.milestone_id = EAPUtil.ToInt(this.Database.DBLookup("milestone_id", "milestone", where));
}
```


Create a MapperBeforeLayout Handler

Now create an event override for MapperBeforeLayout.

Then add the following code.

```
//--- Check to see if there is a http context. Sometimes your code runs under context of scheduler
//--- and in that case the HttpContext.Current object is null
if (HttpContext.Current != null)
{
    //--- if the request is not to a "new" page then make the fields required.
    string req = HttpContext.Current.Request[NetQuarry.RequestParams.Request];
    if (req != "new")
    {
        sender.Fields.assigned_user.Attributes |= FieldAttrs.Required;
        sender.Fields.milestone_id.Attributes |= FieldAttrs.Required;
    }
}
```

Create a RowBeforeUpdate Handler

Create an event override for RowBeforeUpdate

Then add the following code.

```
//--- since status can only be changed by workflow wizards (at the moment)
//--- we can ask the mapper what page it's on to determine what to do.
if (!sender.Fields.assigned_user.Dirty)
{
    switch (sender.Mapper.MOP)
    {
        case "issue!wiz_resolve":
            sender.assigned_user_id = sender.created_by_id;
            break;
        case "issue!wiz_close":
            sender.assigned_user_id = null;
            break;
        case "issue!wiz_reject":
            sender.SetAssigneeFromComponent();
            break;
        case "issue!wiz_reopen":
            sender.SetAssigneeFromComponent();
            break;
    }
}
```

Create a RowAfterInsert Handler

Create an event override for RowAfterInsert.

We will add the following code to send an email notification when an issue is created. Because there is some commonality between this handler and RowAfterUpdate handler, we'll make a function call to some code that handles all the send scenarios.

```
sender.SendEmailNotification(true);
```

Create a RowAfterUpdate Handler

Create an event override for RowAfterUpdate.

Again the body of this function is a call to another function

```
sender.SendEmailNotification(false);
```

Create the Email Notification Handler

In the Issue Typed Mapper class (in IssueTrak.Common) add the code for the email send function

```
public void SendEmailNotification(bool isCreated)
{
    string notifyTemplate = null;
    //--- we said only to send an email notification if the assigned_user_id has changed
    //--- and the assigned_user_id is not the same as the currently logged in user.
    if ((this.Fields.assigned_user_id.Dirty || this.Fields.status_id.Dirty) &&
        !this.assigned_user_id.EqualsCI(this.Application.UserContext.ID))
    {
        switch (this.Mapper.MOP)
        {
            case "issue!wiz_resolve":
                notifyTemplate = "issue-resolved";
                break;
            case "issue!wiz_close":
                //--- there is no notification when an issue is closed
                break;
            case "issue!wiz_reject":
                notifyTemplate = "issue-rejected";
                break;
            case "issue!wiz_reopen":
                notifyTemplate = "issue-reopened";
                break;
            default:
                notifyTemplate = isCreated ? "issue-created" : "issue-assigned";
                break;
        }
        //--- check for some error conditions and throw a descriptive error if necessary
        if (!notifyTemplate.IsNullOrEmpty())
        {
            if (!this.email_address.IsNullOrEmpty())
            {
                //--- send an email to the assigned user with the given template
                this.Send(this.email_address, notifyTemplate);
            }
            else
            {
                throw new EAPException(string.Format("Could not determine an email address for the assigned user: {0} on issue: {1}. An email notification was not sent", this.assigned_user, this.issue_number));
            }
        }
        else
        {
            if (!this.Mapper.MOP.EqualsCI("issue!wiz_close"))
            {
                throw new EAPException(string.Format("Unable to determine notification template for issue {0}. An email notification was not sent", this.issue_number));
            }
        }
    }
}
```

Having made these changes, you will have to compile the IssueTrak.Common.dll and then the Issue Extension.

Register the Typed Mapper version of the extension into the Studio, then attach the Issue typed mapper extension to the issue mapper. On the original Issue mapper extension, you can set it's attribute to "Disabled". This allows you to keep the meta data around but not use it.

There's more functionality to add to this extension, but let's move away to look at page extensions for the time being.

Page Extensions

Page extensions can be attached to Console Template pages and Wizard Pages. They are typically used to manipulate the UI based on complex business rules.

As with the Mapper Extension example, we'll run through a few of the Page extension events to show you how you might want to use them.

Firstly we'll look at Page events for the Console Template pages.

Issue Console Page Extension

In the XIssue.cs file, declare the following page extension.

```
public class PXIssue : NetQuarry.PageExtensionBase
{
}
```

We're going to create event handlers for ConsolePaneBeforeLayout, ConsolePaneBeforeRequery, ConsolePaneAfterRequery

Before we add code, add a reference to NetQuarry Web Host to the Issue Extension project.

Create a ConsolePaneBeforeLayout Handler

We're going to add code to hide the related issues tab if there is at least one related issue.

Also sometimes you may want to add functionality to force the navigation away from a page when you navigate to the page. We'll show this example when the audit log for an issue contains more than 10 rows, you automatically navigate to the full list of audit records for that issue.

This code is ok for an example, but you probably don't want to keep it!

```
//--- there is no live data at this time in the life cycle, so all
//--- data manipulation will rely on query param information
string pk = HttpContext.Current.Request[ReqParams.PrimaryKey];
if (!pk.IsNullOrEmpty())
{
    //--- get access to a mapper.
    //--- either get the mapper for the current console page (still no live data of course)
    //--- or the mapper for the main slot element (still no live data) if the current element has
    //--- no mapper (e.g. MiniNav)
    IMapper map = (EAPUtil.IsNullOrEmpty(sender.MapperObject) ? sender.Console.MainMapperObject :
sender.MapperObject) as IMapper;
    //--- but you still SHOULD check if the mapper is null or not.

    ////--- or if you just want access to an application object
    EAPControlBase rend = sender.Renderer as EAPControlBase;
    IAppContext cxt = rend.ApplicationContext;
    if (map != null)
    {
        //--- hide the related issues if there is at least one related issue
        if (sender.ElementInfo.Name == "related_to")
        {
            if (map.Database.DBExists("issue_rel", string.Format("issue_id={0}", pk.AnsiQuote())))
            {
                sender.Pane.Visible = false;
            }
        }
        //--- navigate to audit list of more than 10 audit records
        //--- navigate away ASAP which means when the main slot is being loaded. It's the first
        //--- slot loaded
        if (sender.IsMainPane)
        {
            string filter = string.Format("rel_id={0}", pk.AnsiQuote());
            if (10 < map.Database.DBCount("xot_audit_readable", filter))
            {
                //--- we need to filter the audit list to the list for this issue, not just any old
                //--- audit records
                string fltParam = SavedFilter.RegisterReqFilter(cxt, filter);
                fltParam = string.Format("flt={0}", EAPEncode.ForUrl(fltParam));
                cxt.Navigate("audit!list", null, fltParam);
            }
        }
    }
}
```


Compile this project. Then register this extension into the application. You register a page extension into the NetQuarry Studio just the same way as a mapper extension. The fact that we added the "P" prefix to the extension name helps to differentiate page extensions to mapper extensions since we don't explicitly differentiate the extensions by type in the studio.

When the extension is registered, go to the list of Pages and the issue!main page. Click on the Extensions subform and add the PXIssue extension to the page.

Login to the application and navigate to the Issues list. Drill down to an issue. If there are related issues, then the "Related Issues" pane is hidden. If not just relate some issues to an issue. On refresh, the list will disappear.

Also if you don't have an issue with more than 10 audit records, you can find an issue with no related issues, and then relate 10 issues to it (which generates 10 audit entries). Once the changes are committed, you will be navigated to the audit list, rather than staying on the issue page.

Create a ConsolePaneBeforeRequery Handler

In this handler, we're going to manipulate the navigation links of the workflow actions. Not only are we going to manage the visibility of the links based on the issue status, but we're also going to change the navigation parameters to allow system admins to manipulate issue workflow, without being required to add a document.

```
const string actResolve = "resolve";
const string actClose = "close";
const string actReject = "reject";
const string actReopen = "reopen";

if (sender.ElementInfo.ComponentInfo.Name.EqualsCI("mininav"))
{
    if (sender.ElementInfo.Name.EqualsCI("workflow_actions"))
    {
        MiniNavCtrl mnc = sender.Renderer as MiniNavCtrl;
        if (mnc != null)
        {
            //--- the main object mapper is populated and on the current row
            IMapper map = sender.Console.MainMapperObject as IMapper;
            //--- we can attach this mapper to a typed mapper if it's more convenient!
            Issue iss = Issue.Attach(map);

            //--- no doc requirement for workflow for admins
            if (map.Application.UserContext.HasProfile("System Admin"))
            {
                mnc.Navigator.SetTargetProperty(actResolve, "QueryParams", "dispgs=document");
                mnc.Navigator.SetTargetProperty(actClose, "QueryParams", "dispgs=document");
            }
            //--- set visibility per status.
            switch (iss.status_id)
            {
                case IssueTrak.Data.Picklists.issue_status.open:
                    mnc.Navigator.SetTargetVisibility(actResolve, true);
                    mnc.Navigator.SetTargetVisibility(actClose, false);
                    mnc.Navigator.SetTargetVisibility(actReject, false);
                    mnc.Navigator.SetTargetVisibility(actReopen, false);
                    break;

                case IssueTrak.Data.Picklists.issue_status.resolved:
                    mnc.Navigator.SetTargetVisibility(actResolve, false);
                    mnc.Navigator.SetTargetVisibility(actClose, true);
                    mnc.Navigator.SetTargetVisibility(actReject, true);
                    mnc.Navigator.SetTargetVisibility(actReopen, false);
                    break;

                case IssueTrak.Data.Picklists.issue_status.closed:
                    mnc.Navigator.SetTargetVisibility(actResolve, false);
                    mnc.Navigator.SetTargetVisibility(actClose, false);
                    mnc.Navigator.SetTargetVisibility(actReject, false);
                    mnc.Navigator.SetTargetVisibility(actReopen, true);
                    break;
            }
        }
    }
}
```

Create a ConsolePaneAfterRequery Handler

In this handler, we're going to manipulate the navigation links of the page element panes. These are the links that suggest "Click Here to View All", "New", or "Add".

We're going to add code that will hide the ability to add documents, notes, relate issues and edit an issue, when the issue has been closed.

```
if (sender.IsMainPane)
{
    ///--- requery of main pane, mapper only has data in after requery
    IMapper map = sender.Console.MainMapperObject as IMapper;
    Issue iss = Issue.Attach(map);
    bool canEdit = (iss.status_id != IssueTrak.Data.Picklists.issue_status.closed);
    sender.LinkList.Enabled = canEdit;
}
else if (sender.ElementInfo.ComponentInfo.Name.EqualsCI("minilist"))
{
    ///--- the main object mapper is populated and on the current row
    IMapper map = sender.Console.MainMapperObject as IMapper;
    if (map != null)
    {
        Issue iss = Issue.Attach(map);
        bool canEdit = (iss.status_id != IssueTrak.Data.Picklists.issue_status.closed);
        switch (sender.ElementInfo.Name)
        {
            case "notes":
            case "documents":
                sender.LinkNew.Visible = canEdit;
                break;
            case "related_to":
                sender.LinkAdd.Visible = canEdit;
                break;
        }
    }
}
```

Issue Wizard Page Extension

We are going to re-use the same page extension class for wizard events that we used for the console page events. However there is nothing stopping you from creating extensions explicitly for Console Pages and Wizards.

In the NetQuarry studio, go to the list of pages. For each of the issue wizards, add the extension `IssueTrak.Extensions.PXIssue`.

Create a WizardBeforePageLoad Handler

In your Issue page extension, create an event override for WizardBeforePageLoad

We will add code to manipulate the descriptive text that can be applied to a wizard. Now you can simply add a static description to a wizard just by setting the Description property on the page element.

```
IWizPage wp = e.WizardPage;
IAppContext cxt = wp.AppContext;
TextItem ti = wp.PageElementInfo.TextItems["Description"];
//--- if there is no description text item, you have to create a new text item
if (ti == null) ti = new TextItem("Description", "description", 0, null, null, null);

switch (wp.PageElementInfo.Name)
{
case "issue":
    ti.Text = "Your opportunity to change information about the issue.";
    break;
case "note":
    switch (wp.PageInfo.Name)
    {
        case "wiz_close":
            ti.Text = "Give a polite congratulatory message for fixing the issue.";
            break;
        case "wiz_reject":
            ti.Text = "Give a good reason for rejecting the issue.";
            break;
        case "wiz_reopen":
            ti.Text = "Justify why you think the issue should be reopened.";
            break;
        case "wiz_resolve":
            ti.Text = "Explain why you think the issue is fixed!";
            break;
    }
    break;
case "document":
    ti.Text = "Upload additional supporting information.";
    break;
}
wp.PageElementInfo.TextItems.Add("Description", ti);
```

Login to the application and go to the Issues list. Click on one of the action buttons to launch an issue workflow wizard. You will now see on each page a green banner displaying the descriptive text based on the context of the selected wizard button.

Create a WizardNext Handler

We will add code to skip the notes page in either the Resolve, or Close issue wizards. The logic to skip the page will be when a note text has been added. We'll show two ways to do this. Either by detecting that note text has been entered, or that we've visited the third page of the wizard at least once. You can't pass the page to enter a note until you provide a note, so having visited page 3 implies you entered data into page 2.

In addition we will manipulate the note text to append a timestamp to the beginning of a note.

Finally we will add code to prevent continuing through the wizard if the resolve wizard note text contains "could not reproduce" and the close wizard note text does not contain "thank".

```
IWizardTemplate w = e.Wizard;
IWizPage wp = e.WizardPage;
IAppContext cxt = wp.AppContext;
IMapper map = wp.Mapper as IMapper;
string noteText = string.Empty;

//--- determine what the next page should be. Never code a page number test directly
//--- e.g. if (e.NextPage == 2)
if (e.NextPage == w.GetPageNumber("note"))
{
    //--- if we already provided a note, then skip the note page
    switch (wp.PageInfo.Name)
    {
        case "wiz_resolve":
            //--- interrogate the wizard page's UserData for a note_text value in the note instance.
            noteText = wp.PageData.GetString("note", "note_text", null);
            if (!noteText.IsNullOrEmpty())
            {
                e.NextPage = e.Wizard.GetPageNumber("document");
            }
            break;
        case "wiz_close":
            //--- determine if the document page has ever been visited.
            if (w.PageVisits.Contains(w.GetPageNumber("document")))
            {
                e.NextPage = e.Wizard.GetPageNumber("document");
            }
            break;
    }
}
else if (e.NextPage == w.GetPageNumber("document"))
{
    switch (wp.PageInfo.Name)
    {
        case "wiz_resolve":
            //--- interrogate the wizard page's UserData for a note_text value in the note instance.
            noteText = wp.PageData.GetString("note", "note_text", null);
            if (noteText.Contains("could not reproduce",
StringComparison.InvariantCultureIgnoreCase))
```

```

    {
        e.Error("You can't say 'could not reproduce' when resolving an issue.");
        return;
    }
    break;
case "wiz_close":
    noteText = wp.PageData.GetString("note", "note_text", null);
    if (!noteText.Contains("thank", StringComparison.InvariantCultureIgnoreCase))
    {
        e.Error("You must say thank you when closing an issue.");
        return;
    }
    break;
}

//--- we navigate to the document page from the note page.
//--- so we have note text that we can modify
noteText = wp.PageData.GetString("note", "note_text", null);
noteText = string.Format("Added on: {0}\r\n{1}", wp.PageData.GetString("note", "date_created",
null), noteText);
wp.PageData.SetValue("note", "note_text", noteText);
}

```

Compile the extension and log in to the application.

Navigate to the Issues list and find an issue that is Open. Click on the "Resolve" action button. On page 2 add a note, and click next to the document page. Don't upload a document yet. Click on the "Back" button. When the "Note" page appears, you can see the note text has been modified with the date stamp. Click previous again to the first page "Issue". Then click the "Next" button. The wizard navigation will skip page 2 to the last documentation page.

Create a WizardPrevious Handler

Create an event override for WizardPrevious

Ordinarily we don't have to add any code to manipulate "Previous" navigation. The platform keeps track of which pages are visited and simply navigates back to the previous page in the stack of visited pages.

However, in the first pass through, we visit page 1, 2 then 3. Clicking "Previous" from page 3, the stack logic says go back to page 2. Since page 2 has data on it, we do not want to visit page 2 "Note" any longer.

Add the following code to manipulate the "Previous" navigation if note text is provided.

```
IWizardTemplate w = e.Wizard;
IWizPage wp = e.WizardPage;
IApplicationContext cxt = wp.AppContext;
IMapper map = wp.Mapper as IMapper;

//--- determine what the next page should be.
if (e.NextPage == w.GetPageNumber("note"))
{
    //--- if we already provided a note, then skip the note page
    switch (wp.PageInfo.Name)
    {
        case "wiz_resolve":
            //--- interrogate the wizard page's UserData for a note_text value in the note instance.
            string noteText = wp.PageData.GetString("note", "note_text", null);
            if (!noteText.IsNullOrEmpty())
            {
                e.NextPage = e.Wizard.GetPageNumber("issue");
            }
            break;
        case "wiz_close":
            //--- determine if the document page has ever been visited.
            if (w.PageVisits.Contains(w.GetPageNumber("document")))
            {
                e.NextPage = e.Wizard.GetPageNumber("issue");
            }
            break;
    }
}
```

Compile and login to the application. Go to issues list and find another open issue to resolve. This time when you get to the document upload page and click previous, the "note" page will be skipped

Create a WizardCancel Handler

Create an event override for WizardCancel.

You normally handle the cancel event when you want to manipulate the navigation behavior when the cancel button is clicked. The default handling is set in the Wizard Page metadata properties for CancelAction, etc. In the metadata we set up the issue wizards to "Return" back to the original caller. We'll override that behavior based on which is the current page.

```
IWizardTemplate w = e.Wizard;
IWizPage wp = e.WizardPage;
IAppContext cxt = wp.AppContext;
IMapper map = wp.Mapper as IMapper;

switch (w.CurrentPage)
{
case 1:
    //-- get the issue_id from the user data, or off the query string
    string issue_id = wp.PageData.GetString("resolve_issue", "issue_id", null);
    if (issue_id.IsNullOrEmpty())
    {
        issue_id = HttpContext.Current.Request[ReqParams.PrimaryKey];
        if (issue_id.IsNullOrEmpty())
        {
            issue_id = HttpContext.Current.Request[ReqParams.ParentRowKey];
        }
    }
    cxt.Navigate("issue!main", issue_id);
    break;
case 2:
    cxt.Navigate("issue!list");
    break;
case 3:
    string assignedUser = wp.PageData.GetString("close_issue", "assigned_user_id", null);
    string individualID = EAPUtil.ToString(map.Database.DBLookup("user_guid", "users",
string.Format("user_id={0}", assignedUser.AnsiQuote())));
    cxt.Navigate("individual!main", individualID);
    break;
}
```

Compile and log in to the application. Navigate to the Issue List and run through the issue wizards. On each page of the wizard, click the Cancel button to see the effects of the different logic.

Manipulating Sets of Data

A common requirement in an application is to perform a bulk operation on a set of records. We've already seen implementations of this idea when we added the Template Mailer and Export to Excel extensions to mappers. These added "Actions" to the menu bar (either via the Actions menu on the page, or as a separate toolbar button on the page).

The basic idea is to add the action button to a page that, when clicked, fires a command via the MapperCommand event back to an extension. In the extension you either directly perform actions on the selected records, or you cache the affected records and perform the action elsewhere.

Making Bulk Updates to Issues

We are going to implement a process that allows a user to select one or more issues from a list and make bulk changes to those issues based on a set of user defined choices. Once the bulk update process has finished, then we will navigate to the list of touched issues.

Creating the Action Menu

There are two ways we can do this. Create a menu navigator in metadata, or programmatically add the action menu in code. The template mailer and excel export extensions add the action items via code.

For now, we'll add the action menu via metadata.

Open the NetQuarry Studio and navigate to "Menus" under "Navigator".

Add a "Menu" with the following settings.

Item	Value	Notes
Module	issue	
Name	issue-mnu	
Type	Menu	

On the Targets subform of the "issue-mnu" Menu, create the target

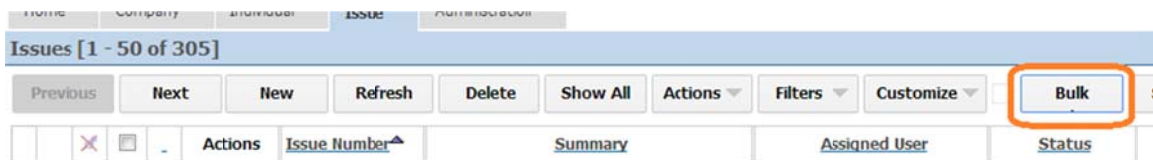
Item	Value	Notes
Module	issue	
Name	bulk-update	
TargetType	Command	
Target	bulk-update	
Sort Order	10	

Set the following properties

Property	Value	Notes
CommandLocation	Toolbar	
CommandAttributes	RequireSelection	Forces a user to choose at least one record from the list before the action is performed. If you want to allow the action to be performed against the entire set of records matching the filter, then don't set the RequireSelection attribute.
Caption	Bulk Update Issues	
Confirmation	Are you sure you want to perform a mass update of the selected issues?	
ToolTip	Make bulk changes to a set of issue you have selected.	

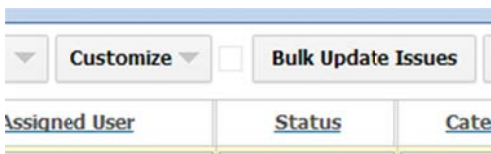
Now go to the Page list and select the page "issuelist". In the property sheet of the page, set the PageCommands property to "issue-mnu".

Login to the application and you can see the action button is added.



Notice that the button is not wide enough to display all the text. We have to go into the custom.css file and style the button to be a little larger. Open the file %NQROOT%\Apps\IssueTrak\Styles\custom.css and add the following style.

Script	Notes
<pre>#toolbar_bulk-update_b { width: 130px; }</pre>	Basically you can inspect the button element to get the id of the button. That name will be used in your style sheet to set a specific width.



IMPORTANT

The style sheet you modified is part of the platform installation. Therefore, if you reinstall a new platform version, the changes you made to the css file will be lost forever.

To avoid this loss, you should copy your modified css file to the folder
C:\NetQuarry\Customers\IssueTrak\Config\web\Styles

After the install has been completed, the Install.bat file contains a step to copy the backed up custom.css style sheet to the %NQROOT%\Apps\IssueTrak\Styles folder.

Handling the Action Menu Command event

Now go back to your Issue extension in visual studio. In the XTMIssue mapper extension component (the typed mapper version of the issue extension), add an override for "MapperCommand" as follows...

```
public override void MapperCommand(Issue sender, EAPCommandEventArgs e)
{
    switch (e.CommandName)
    {
        case "bulk-update":
            PerformBulkUpdate(sender);
            break;
        //--- DO NOT HAVE A DEFAULT CASE
        //--- ONLY HANDLE COMMANDS YOU KNOW ABOUT
    }
}

private void PerformBulkUpdate(Issue sender)
{
    //--- save of a 'bucket' of keys matching those selected and get its ID
    string filterName = "Issue Bulk Update";
    FilterAttributes fa = FilterAttributes.Static | FilterAttributes.Temp |
    FilterAttributes.KeyGuid | FilterAttributes.Hidden;
    //--- save a filter of the selected records
    SavedFilter sf = sender.Mapper.Exec(MapperExecCmds.FilterSave, fa, filterName) as SavedFilter;
    //--- then register this filter
    string fltQP = SavedFilter.RegisterReqFilter(this.Application, sf.Filter, "Bulk Modified
    Issues", "Issues affected by your bulk changes", "issue!list");

    //--- navigate to an issue modification wizard page
    //--- navigating as new so we don't need a PK to go to a specific record
    //--- and pass along the ID of the saved filter. These are the issues we are going to
    process.
    sender.Application.Navigate("issue!wiz_bulk_modify", null, "flt_id=" +
    EAPEncode.ForUrl(fltQP), "new");
}
```

Compile the Extension.

Create the Issue Bulk Modify Wizard

We now have to create a new wizard page to provide a set of options to modify an issue.

In the NetQuarry Studio go to the Pages list and add a new page with the following settings.

Item	Value	Notes
Module	issue	
Name	wiz_bulk_modify	
Moniker	make bulk issue changes	
Template	WizTemplate.aspx	
Mapper	<nothing>	

With that new Page selected, click on the Property sheet and set the following properties.

Property	Value	Notes
WizardAttributes	NoStepCaption	A one page wizard shouldn't need a step caption
CancelAction	Return to Caller	
FinishAction	Return to Caller	
Caption	Bulk Modify Issues	
Finish_Caption	Commit	

Go to the Extensions subform of the page and add the IssueTrak.Extensions.PXIssue.

This wizard will have a page to make changes to issues and also require a note to be specified to justify making a change. Previously, we have seen a wizard example where these separate ideas would be shown on separate pages. This time we'll create this wizard where both of the elements are on a single page.

Now add the Page Elements with the following settings

Item	Value	Notes
Slot Name	WizardPage	
Name	bulk_modify	
Component	WizardGrouper	This is the component type that allows multiple instances to be rendered on the same page. Think of this as a Wizard page in the form of a Console Page.
Order	10	

Create the next Page Element for modifying issue settings

Item	Value	Notes
Slot Name	WizardGrouper	
Name	issue_data	
Component	WizardPhantomDetail	
Order	20	

With that new Page Element selected, click on the Property sheet and set the following properties.

Property	Value	Notes
InstanceName	issue	
Mapper	issue	
FieldList	assigned_user;priority_id;severity_id;status_id	
Grouper	bulk_modify	
Caption	Modify Issue Data	

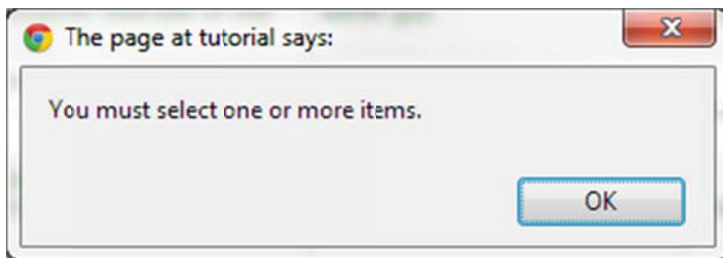
Create the next Page Element for adding a note about the change you make

Item	Value	Notes
Slot Name	WizardGrouper	
Name	required_note	
Component	WizardPhantomDetail	
Order	30	

With that new Page Element selected, click on the Property sheet and set the following properties.

Property	Value	Notes
InstanceName	note	
Mapper	note	
FieldList	note_text	
Grouper	bulk_modify	
Caption	Justify your modification	

Having created the page to modify issues, you can login to the application, go to the issues list. Select a couple of issues and click on the "Bulk Update Issues" button. If you have not selected an issue, you'll get the error message...



When you navigate to the bulk modify page you should notice a couple of things.

A screenshot of a web application interface for "Bulk Modify Issues". The page has a blue header with the title "Bulk Modify Issues". Below the header is a section titled "Modify Issue Data" with a light blue background. This section contains three fields: "Assigned User:" with a text input and a user icon, "Priority: *" with a dropdown menu, and "Severity: *" with a dropdown menu. The "Priority" and "Severity" fields are highlighted with an orange rectangle. Below this section is another section titled "Justify your modification" with a light blue background. This section contains a "Note Text: *" label and a large text area for input.

The Priority and Severity fields are required. Since this is an optional procedure, we have to make these two fields optional. Also, the Status field is missing even though we explicitly selected the field in the FieldList property. The Status field is missing because the field has a Hide Flavor of "new". This is a new record so the field is hidden.

We'll fix this through custom flavoring.

Fix the Bulk Modify Wizard UI

Go to the Studio and the Mappers list. Select the "issue" mapper and click on the Custom Flavor Names tab. Add a custom flavor with the following values.

Item	Value	Notes
Flavor	Custom1	
Name	bulk_modify	

Custom Flavor Names - (1 Rows)

Flavor	Custom Name
Custom1	bulk_modify

Click on the Field Subform. We are going to duplicate a set of fields that are essentially required, that we need to make NOT required. This set of fields includes the three fields on the screen as well as the set of fields that are required that are not visible.

Individually select each of the following fields, Right Click and Duplicate. (Do not Press F5 to refresh the list)

status_id, priority_id, severity_id, summary, description, category_id, project_id and component_id

On each of the fields that you've just duplicated, set the Exclude flavor of c1 – bulk_modify



Now press F5 to refresh the list. On the fields that you duplicated that DON'T have an exclude flavor of c1 – bulk_modify, add the INCLUDE flavor of c1 – bulk_modify. When you have finished your flavor settings must look like this.

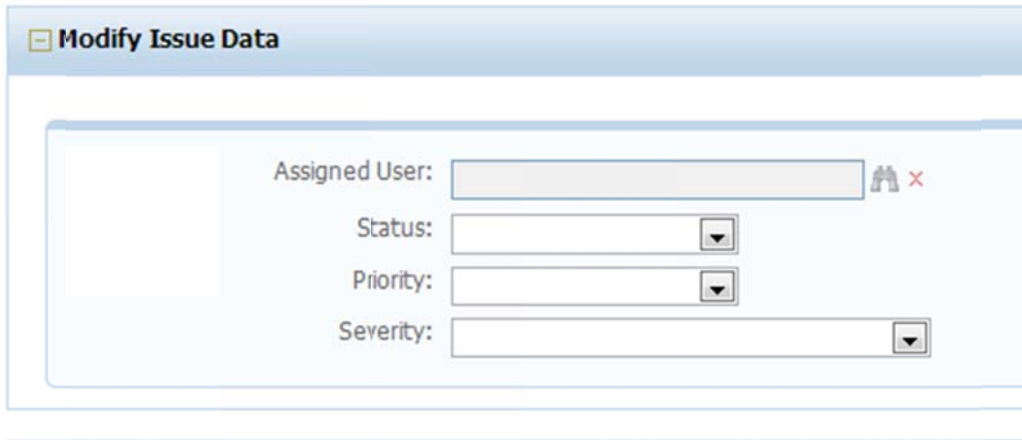
KeyName	Column Order	Picklist	Column Width	Cell Type	.NET Type	Data Type	Include Flavor	Exclude Flavor	Version
issue_number	20		20	TextBox	System.Int32	nt			
summary	30		28	TextBox	System.String	nvarchar		1	
description	40		40	TextBox	System.String	nvarchar		1	
assigned_user	50		20	TextBox	System.String	nvarchar	16384		
assigned_user_id	50		0	ComboBox	System.String	nvarchar			
assigned_user	50		20	Find	System.String	nvarchar		16384	
status_id	60	issue_status	12	ComboBox	System.Int32	nt	524288	1	
status_id	60	issue_status	12	ComboBox	System.Int32	nt		524288	
status_id	60	issue_status	12	ComboBox	System.Int32	nt	1	524288	
category_id	70	issue_category	12	ComboBox	System.Int32	nt		1	
project_id	80	projects	15	ComboBox	System.Int32	nt		1	
component_id	90	components	15	ComboBox	System.Int32	nt		1	
priority_id	100	issue_priority	12	ComboBox	System.Int32	nt		1	
priority_id	100	issue_priority	12	ComboBox	System.Int32	nt	1		
severity_id	110	issue_severity	20	ComboBox	System.Int32	nt		1	
severity_id	110	issue_severity	20	ComboBox	System.Int32	nt	1		
milestone_id	120	milestones	20	ComboBox	System.Int32	nt			

For each of the fields with the INCLUDE flavor of c1 – bulk_modify make the following changes.

Item	Value	Notes
Attributes	Remove Required Remove Audit Remove Locked	
HideFlavor	Remove New Flavor	
DefaultValue	Remove	

Go to the Pages list and select the issue!wiz_bulk_modify page. Select the issue_data Page Element. Set the Flavor property to c1 – bulk_modify.

Login to the application again and perform the bulk modify action.



The screenshot shows a web application window titled "Modify Issue Data". Inside the window, there is a light blue panel containing four form fields:

- Assigned User:** A text input field with a user icon and a red 'x' button to its right.
- Status:** A dropdown menu.
- Priority:** A dropdown menu.
- Severity:** A dropdown menu.

You can see the all the fields are visible and they are no longer required

Make Bulk Modify Wizard Co-Exist with Other Wizards

We now have the necessary UI created for the bulk update functionality. But before we can make use of this, we have to make quite a few changes to the extension code we've already created on both the issue typed mapper extension and the issue page extension. This is to ensure that we are not going to execute code in the wrong context.

In the RowBeforeInsert Handler of the XTMIssue extension, add the highlighted code as follows

```
if (!EAPUtil.BitsSet(sender.Mapper.Flavor, Flavors.Custom1))
{
    //--- set the default assignee
    if (!sender.Fields.assigned_user.Dirty)
    {
        sender.SetAssigneeFromComponent();
    }

    //--- set the default milestone
    if (!sender.Fields.milestone_id.Dirty)
    {
        sender.SetMilestoneFromProject();
    }
}
```

This code tests the flavor of the mapper to determine the context of execution. We've decided that Custom1 represents an issue mapper in the context of a bulk modify. In this context we don't want to be setting any default values.

Go to the Issue.cs file that declares the Issue typed mapper object (in the IssueTrak.Common project). Find the SendEmailNotification function.

We have to add some code to make sure the email_address field is in sync with the assigned_user_id field. In a normal situation of saving an issue, after the saving of the issue record, the record is re-queried and, because of the join in the underlying view of the issue mapper, the correct email_address is pulled into the view from the new user_id.

However in the code we're going to add, there will be no post save requery, so the email address will need to be manually updated to ensure notifications are sent to the correct assignee.

Add the highlighted code to the existing method

```
string notifyTemplate = null;
//--- we said only to send an email notification if the assigned_user_id has changed
//--- and the assigned_user_id is not the same as the currently logged in user.
if ((this.Fields.assigned_user_id.Dirty || this.Fields.status_id.Dirty) &&
    !this.assigned_user_id.EqualsCI(this.Application.UserContext.ID))
{
    switch (this.Mapper.MOP)
```

```

{
    case "issue!wiz_resolve":
        notifyTemplate = "issue-resolved";
        break;
    case "issue!wiz_close":
        ///--- there is no notification when an issue is closed
        break;
    case "issue!wiz_reject":
        notifyTemplate = "issue-rejected";
        break;
    case "issue!wiz_reopen":
        notifyTemplate = "issue-reopened";
        break;
    default:
        notifyTemplate = isCreated ? "issue-created" : "issue-assigned";
        break;
}

if (EAPUtil.BitsSet(this.Mapper.Flavor, Flavors.Custom1) &&
this.Fields.assigned_user_id.Dirty)
{
    ///--- make sure we pull in the correct email address for this user if the user_id has
    changed
    this.Fields.email_address.SetValue(this.Database.DBLookup("email_address", "users",
string.Format("user_id={0}", this.assigned_user_id.AnsiQuote()), SetValFlags.OnlyIfDiff,
null));
}

///--- check for some error conditions and throw a descriptive error if necessary
if (!notifyTemplate.IsNullOrEmpty())
{
    if (!this.email_address.IsNullOrEmpty())
    {
        ///--- send an email to the assigned user with the given template
        this.Send(this.email_address, notifyTemplate);
    }
    else
    {
        throw new EAPEException(string.Format("Could not determine an email address for the
assigned user: {0} on issue: {1}. An email notification was not sent", this.assigned_user,
this.issue_number));
    }
}
else
{
    if (!this.Mapper.MOP.EqualsCI("issue!wiz_close"))
    {
        throw new EAPEException(string.Format("Unable to determine notification template for issue
{0}. An email notification was not sent", this.issue_number));
    }
}
}
}

```

While we are in the IssueTrak.Common project, we are also going to add a new typed mapper definition for the note mapper.

In the IssueTrak.Common project, create a new C# class file called "note.cs".

And add the following code to the note.cs file.

```
using System;
using System.Collections.Generic;
using System.Text;
using NetQuarry;
using NetQuarry.Data;
using IssueTrak.Common;

namespace IssueTrak.Data
{
    {
    public class Note : IssueTrak.Data.Generated.note<Note>
    {
    }
    }
}
```

Compile the IssueTrak.Common project.

Now we have to make some changes to the existing code in the issue page extension, PXIssue. Again, to ensure the right code executes under the right context.

In WizardBeforePageLoad, add the following highlighted code.

```
IWizPage wp = e.WizardPage;
IAppContext cxt = wp.AppContext;

switch (wp.PageInfo.MOP)
{
    case "issue!wiz_close":
    case "issue!wiz_reject":
    case "issue!wiz_reopen":
    case "issue!wiz_resolve":
        TextItem ti = wp.PageElementInfo.TextItems["Description"];
        //--- if there is no description text item, you have to create a new text item
        if (ti == null) ti = new TextItem("Description", "description", 0, null, null, null);

        switch (wp.PageElementInfo.Name)
        {
            case "issue":
                ti.Text = "Your opportunity to change information about the issue.";
                break;
            case "note":
                switch (wp.PageInfo.Name)
                {
                    case "wiz_close":
                        ti.Text = "Give a polite congratulatory message for fixing the issue.";
                        break;
                    case "wiz_reject":
                        ti.Text = "Give a good reason for rejecting the issue.";
                        break;
                    case "wiz_reopen":
                        ti.Text = "Justify why you think the issue should be reopened.";
                        break;
                    case "wiz_resolve":
                        ti.Text = "Explain why you think the issue is fixed!";
                        break;
                }
            }
        }
    }
}
```

```

    }
    break;
case "document":
    ti.Text = "Upload additional supporting information.";
    break;
}
wp.PageElementInfo.TextItems.Add("Description", ti);
break;
}
}

```

We do not need to make similar changes to the WizardNext, or WizardPrevious handler. This bulk update wizard only has one page, so there will be no Next and Previous events fired.

Make the following changes to the WizardCancel event handler.

```

IWizardTemplate w = e.Wizard;
IWizPage wp = e.WizardPage;
IAppContext cxt = wp.AppContext;
IMapper map = wp.Mapper as IMapper;

switch (wp.PageInfo.MOP)
{
case "issue!wiz_close":
case "issue!wiz_reject":
case "issue!wiz_reopen":
case "issue!wiz_resolve":
    switch (w.CurrentPage)
    {
        case 1:
            /*--- get the issue_id from the user data, or off the query string
            string issue_id = wp.PageData.GetString("resolve_issue", "issue_id", null);
            if (issue_id.IsNullOrEmpty())
            {
                issue_id = HttpContext.Current.Request[ReqParams.PrimaryKey];
                if (issue_id.IsNullOrEmpty())
                {
                    issue_id = HttpContext.Current.Request[ReqParams.ParentRowKey];
                }
            }
            cxt.Navigate("issue!main", issue_id);
            break;
        case 2:
            cxt.Navigate("issue!list");
            break;
        case 3:
            string assignedUser = wp.PageData.GetString("resolve_issue", "assigned_user_id",
null);
            string individualID = EAPUtil.ToString(map.Database.DBLookup("user_guid", "users",
string.Format("user_id={0}", assignedUser.AnsiQuote())));
            cxt.Navigate("individual!main", individualID);
            break;
    }
    break;
}
}

```

Performing the Bulk Modify

We now override a new Wizard Event, WizardDataExchange. This event handler will be used to override the default finish handling. We are using this event handler rather than WizardFinish due to the lifecycle of the data during the Finish handling of a wizard.

When the WizardFinish handler has fired, none of the data on the page has been transferred to the underlying mapper object for that last page, nor has that data been posted into the UserData collection of the wizard page.

Only in the WizardDataExchange event, and specifically when that event is fired under the WizardDataExchange.AfterMapperToUserDataFinal context is the necessary data supplied to the mapper and user data collection.

On a standard wizard where each page represents a single instance of data, this event is fired only once during the finish handling. However, we have constructed a wizard with two panes. Therefore the WizardDataExchange event will be fired for each Wizard pane of the wizard page.

Because of this sequential firing of events, we have to ignore the first event fired by the issue mapper because we also need to know the note text to associate with an updated issue. When we then get the second event fired for the note mapper, we have all the data we need to continue with the save. We can get directly to the note mapper data (therefore the note text) and extract the issue mapper from the wizard page.

Once the bulk modify process is complete, we navigate to a list of issues that were modified.

```
if (e.ExchangeType == WizDataExchangeType.AfterMapperToUserDataFinal)
{
    IWizardTemplate w = e.Wizard;
    IWizPage wp = e.WizardPage;
    IAppContext cxt = wp.AppContext;
    IMapper map = wp.Mapper as IMapper;
    Fields dirtyFields = null; //-- the set of fields that are dirty on issue mapper.

    switch (map.Key)
    {
        case "issue":
            //-- this is the issue mapper save. We're going to not save anything and handle
            everything when the note is saved.
            dirtyFields = map.Fields.Subset(FieldSubsetType.Dirty, true);
            if (dirtyFields.Count == 0)
            {
                //-- if there are no changes to make, then throw an error
                e.Cancel("You must change at least one of the issue parameters");
            }
            else
            {
                //-- if there are changes, tell the platform we've handled everything
                //-- it won't try to save the issue record wizard as a new issue.
                //-- but the finish processing will continue as though the save did occur
                e.Result = ExtResults.HandledInFull;
            }
    }
}
```



```

        break;
    case "note":
        ///--- this is the note saving time. We have to get to the issue mapper as well
        EAP.WebHost.WizGrouper wg = w.CurrentWizPage as EAP.WebHost.WizGrouper;
        WizContainerPanels wcp = wg.Panels;
        IWizContainerPanel iwcp = wcp["issue_data"];
        IWizPage wpIssue = iwcp.Renderer;
        IMapper mapIssue = wpIssue.Mapper as IMapper;

        ///--- identify the fields that are dirty on the issue mapper
        dirtyFields = mapIssue.Fields.Subset(FieldSubsetType.Dirty, true);
        string copyFields = null;
        ///--- create a semi-colon delimited list of modified issue fields.
        foreach(IField f in dirtyFields)
        {
            copyFields += copyFields.IsNullOrEmpty() ? f.Key : string.Format(";{0}", f.Key);
        }

        ///--- get the selected record filter. Get both the registered filter id and the actual
        filter expression
        string fltQP = HttpContext.Current.Request["flt_id"];
        string filterExpr = SavedFilter.ExtractFilterFromReq(cxt, HttpContext.Current.Request,
"flt_id");

        ///--- use the filter expression to filter for the right issues
        ///--- as we iterate through a list of records and save, we don't do a row query.
        ///--- it disrupts the positioning of the current record in the mapper
        using (Issue iss = Issue.OpenReader(cxt, filterExpr, 0, MapperAttrs.NoRowQuery))
        {
            ///--- instantiate the note mapper up front, outside the inner looping of issue updates
            ///--- always be mindful of performance
            using (Note note = Note.OpenNew(cxt))
            {
                while (iss.MoveNext())
                {
                    ///--- set the selected values from the wizard issue mapper onto the selected
issues
                    iss.Mapper.Exec(MapperExecCmds.CopyValuesFrom, 0, mapIssue, copyFields);
                    iss.Save();

                    ///--- create the note to go along with the bulk change
                    note.MoveNew();
                    note.related_id = iss.issue_id;
                    note.note_text = EAPUtil.ToString(map.Fields["note_text"].Value);
                    note.Save();
                }
                note.Close(); ///--- always close your mappers
            }
            iss.Close(); ///--- always close your mappers
        }
        ///--- navigate to the finished bulk update issues. The platform recognizes the
        ///--- registered filter on the "flt" paramter and automatically applies the required
filter.
        cxt.Navigate("issue!list", null, "flt=" + EAPEncode.ForUrl(fltQP), "nav");

        break;
    }
}

```

Compile the Issue extension.

Now login to the application. Navigate to the issues list and try to bulk change one or more issues. If you change the assigned user, that assigned user will receive a notification change through your email handling application.

Scheduled Tasks

A scheduled task is a mechanism that allows background tasks to be executed without any user intervention. Typical uses for scheduled tasks are

- Sending reminder notifications to users
- Performing data maintenance tasks
- Executing long running processes behind the scenes that are initiated by user actions on the front end (e.g. queuing a job)

In this tutorial we will create a scheduled task that runs nightly that will send out reminder notifications to users if they have any Open issues that are due in the next milestone when the milestone is due in two days.

Create the Open Issues Due in 2 Day Task

Go back to your IssueTrak code solution right click on the Solution items and "Add -> New Project".

Set the following options

Item	Value	
Project Type	Visual C#	
Template	Class Library	
Name	Notifications	
Location	C:\NetQuarry\Customers\IssueTrak\Source\Tasks	

In a process similar to when we created an new extension

Change the name of the project to IssueTrak.Tasks.Notifications

Change the name of Class1.cs to Notifications.cs

Add references to...

IssueTrak.Common.dll, NetQuarry Core EAP.Core.dll, NetQuarry Data Binding. Use the "Recent" tab of the Add Reference dialog for speed.

Drag the project.build file from IssueTrak.Common to IssueTrak.Tasks.Notifications

Open the project.build file in the IssueTrak.Tasks.Notifications component.

Change the solution value to "IssueTrak.Tasks.Notifications"

Drag the AssemblyInfo.cs file from IssueTrak.Common \Properties to IssueTrak.Tasks.Notifications\Properties, overwriting when prompted.

Open the AssemblyInfo.cs file in the IssueTrak.Tasks.Notifications\Properties.

Change the Assembly Title to "IssueTrak.Tasks.Notifications"

Change the Assembly Description to "IssueTrak Notifications Task"

Right click on the IssueTrak.Tasks.Notifications project and choose Properties

Change the Assembly Name to IssueTrak.Tasks.Notifications

Change the Default Namespace to IssueTrak.Tasks (save and close the Properties)

In the Notifications.cs file, use this code

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IssueTrak.Common;
using IssueTrak.Data;
using NetQuarry;
using NetQuarry.Data;

namespace IssueTrak.Tasks
{
    public class Notifications : NetQuarry.ScheduledHandler
    {
    }
}

```

Inside the Notifications class, override the OnExec Handler

```

protected override void OnExec(int cmdID, params object[] args)
{
}

```

Having made these changes you should be able to successfully compile the IssueTrak.Tasks.Notifications object.

Register the task in the Studio

You now have to register this task in the studio. Instead of going to the Extension list under Components to register this task, you instead register the component as a "Handler". Click on the "Handlers" list under Components and add the following data.

Name	Value	Notes
Module	issue	
Name	NotificationHandler	
Component Type	Handler	
Component Name	IssueTrak.Tasks.Notifications	
Assembly Name	IssueTrak.Tasks.Notifications.dll	
Assembly Path	C:\NetQuarry\Customers\IssueTrak\Source\Tasks\Notifications\bin\debug	
Assembly Path Prod	%NQROOT%\Apps\IssueTrak\bin	

Define the Task in the Studio

Now go to the Scheduled Tasks list (ALT+K) and define a new scheduled task as follows.

Name	Value	Notes
Module	issue	
Task Name	OpenIssueMilestoneIn2Days	
Component	NotificationHandler	This is the component you just registered.
Interval Mins	1	The frequency of the task. If this task was set to run between certain times, or had no time restriction, the task would fire every 1 minute. However this task will run once a day (see Run Once Time, below).
CommandID	1	Which command ID that represents this task. This ID will be passed to the OnExec override in your task so you know which of many tasks should be executed.
Run Once Time	4:00AM	This task will run once a day at 4:00 AM local time.

Create a Mapper for Identifying who to Notify

Go to the Mappers list and create a new mapper with the following settings

Item	Value	Notes
Module	individual	
Name	individual_milestone_notify	
View	individual_milestone_notify_view	
Data Source	IssueTrak	

Move off the row to save, then select the individual_milestone_notify mapper. Right click on it and choose "Create Fields...". Select all the fields and click OK.

This mapper is based on a view that returns the distinct set of users who have an issue in a milestone with a due date. We will use this mapper to obtain a set of users to notify by filtering on the milestone due date and the issue status (open).

The notification will also contain a list of issues that are assigned to the user in that milestone. We will add this support in the same mapper, by adding a MapperSummary type field to the mapper.

Go to the Fields subform of the individual_milestone_notify mapper and add a new field with the following values.

Field	Settings	Notes
related_issues	Key Name: related_issues Column Order: 90 CellType: MapperSummary .NET Type: System.String Data Type: varchar Attributes: ExcludeFromSelect Table: + Mapper: issue (make sure set this before you set the FieldList property. The fields populate into the FieldList property widget once the mapper is specified) FieldList: issue_number;summary;priority_id;project_id;component_id HTMLSummaryFlags: Horizontal Caption: Related Issues	Ordinarily on a MapperSummary field we would set the Filter property in the metadata but in this example we will be programmatically modifying the properties of the MapperSummary field in order to select the appropriate data.

Perform a Typed Mapper code generation.

Create the Notification Template

Go to the templates list and add a template with the following settings.

Item	Value	Notes
Module	individual	
Name	notify-milestone-2day-open	
Type	File	

With this template selected, click on the Property Sheet and set the following properties

Property	Value	Notes
FileName	IssueTrak\individual\notify-milestone-2day-open.html	This file already exists in this template location. Feel free to edit the contents once we get this functionality working.
Subject	You have some issues due to be completed on {{milestone_due_date}}	

Declare More Typed Mappers

In the IssueTrak.Common project. Add a new class file called project.cs and add the following code to the class file.

```
using System;
using System.Collections.Generic;
using System.Text;
using NetQuarry;
using NetQuarry.Data;
using IssueTrak.Common;

namespace IssueTrak.Data
{
    public class Project : IssueTrak.Data.Generated.project<Project>
    {
    }

    public class Milestone : IssueTrak.Data.Generated.milestone<Milestone>
    {
    }

    public class Component : IssueTrak.Data.Generated.component<Component>
    {
    }
}
```

In the existing class file, Individual.cs in the IssueTrak.Common project, declare the Individual milestone notify typed mapper as follows

```
public class IndividualMilestoneNotify :
IssueTrak.Data.Generated.individual_milestone_notify<IndividualMilestoneNotify>
{
}
```

Compile the Common Project.

Create the Notification Code in the Notification Handler

Go back to your Notification object in the IssueTrak solution and add the following code to your task handler class.

```
private const int cnOpenIssueMilestoneIn2Days = 1;
protected override void OnExec(int cmdID, params object[] args)
{
    switch (cmdID)
    {
        case cnOpenIssueMilestoneIn2Days:
            OpenIssueMilestoneInDays(2);
            break;
    }
}

private void OpenIssueMilestoneInDays(int offset)
{
}
```

Note that we have a switch statement to identify the right task to be executed and we call a separate function to perform the work. We may want to reuse the function for a different number of days to be offset.

In the OpenIssueMilestoneInDays function, add the following code.

```
//--- identify any milestones due in the next offset days
string milestoneFilter = string.Format("DATEDIFF(DD, {0}, milestone_due_date) = {1}",
DateTime.Today.ToString("yyyy-MM-dd").AnsiQuote(), offset);
if (this.Application.DataDB.DBExists("milestone", milestoneFilter))
{
    //--- create the notify mapper now, but don't query it until we know we have notifications to send
    using (IMapper notify = NetQuarry.Data.Mapper.CreateAndLoad("individual_milestone_notify",
this.Application, 0, 0, null, false, false))
    {
        //--- attach the generic mapper to a typed mapper.
        IndividualMilestoneNotify imf = IndividualMilestoneNotify.Attach(notify);

        //--- we can now identify to show open issues for the current milestone at the moment
        string notifyFilter = string.Format("status_id={0} AND milestone_due_date={1}",
(int)IssueTrak.Data.Picklists.issue_status.open,
DateTime.Today.AddDays(offset).ToString("yyyy-MM-dd").AnsiQuote());
        imf.Mapper.Filters.Add("OpenMilestoneIssues", notifyFilter);

        //--- set the filter criteria for the related issues
        string issueFilter = string.Format("assigned_user_id=[user_id] AND status_id={0} AND
milestone_id IN (SELECT milestone_id FROM milestone WITH(NOLOCK) WHERE
milestone_due_date={1})", (int)IssueTrak.Data.Picklists.issue_status.open,
DateTime.Today.AddDays(offset).ToString("yyyy-MM-dd").AnsiQuote());
        imf.Fields.related_issues.Properties.Add("Filter", issueFilter);

        //--- requery the notify mapper
        imf.Requery();
    }
}
```

```

    //--- and iterate through to notify
    //--- NEVER Loop on a potentially long process without checking whether to yield to a stop
service request
    //--- note this method has a little different looping pattern in that when we requery, we
are already on the first row.
    if (imf.HasRecords)
    {
        do
        {
            imf.Send(imf.email_address, "notify-milestone-2day-open");
        } while (imf.MoveNext() && !this.IsServiceStopped);
    }
    notify.Close();
}
}
else
{
    DevLog.LogMessage("IssueTrak.Tasks.Notifications", "OpenIssueMilestoneIn2Days", "There are no
milestones due in two days. Reminder notifications will not be sent.", LogMessageLevel.Info);
}

```

Compile the Notification Task project.

Use the TaskRunner to test your code.

In production, the tasks you create will run under the context of the NetQuarry scheduler process, EAP.Scheduler.exe. In development, you can also use this process to run your tasks. However, it's quite difficult to debug your code by attaching to the EAP.Scheduler.exe process.

The minimum launch frequency for a scheduled process is 1 minute. Debugging the EAP.Scheduler.exe, you have to wait 1 minute before your task will launch. If your task has a start time set to 4:00AM, you will have to wait until 4:00AM to literally debug your code! Also if you have multiple scheduled tasks configured, they will all start running unless you manually disable the tasks you don't want to debug.

To streamline the debugging of tasks, you can launch any task, on demand by running the EAP.Tools.TaskRunner.exe process. It is a windows based program that lets you choose to run any task from any configured application, immediately, on demand.

You can launch the Task Runner from the "NQ Task Runner" shortcut on the NQ Links menu on the task bar.

When the tool launches, it inspects the EAP.Scheduler.exe.config file for any configured applications. For each application that is configured, the tool attempts to establish a connection to each application to ascertain whether the machine can run tasks for that application. At the end of these checks, the available applications to test can be selected from the drop down and the errors that occurred trying to connect to other applications are listed in the status window.

Select the IssueTrak application and the task list will be populated with the configured task for that application. Some tasks may be disabled. These are disabled to the EAP.Scheduler.exe process, but are available to be launched by the EAP.TaskRunner.exe tool.

Select the task you want to run ("OpenIssueMilestonIn2Days") and click on the Execute Selected Task button.

If you get any errors with the TaskRunner they are presented in the Task Result panel below the list of tasks. It is recommended however that you inspect the devlog to see more details about the error. The devlog for the EAP.Tools.TaskRunner.exe is EAP.TaskRunner.exe.nql

You may find that when you execute the task, it finishes quickly and no emails are sent. This is not unusual because the task is a time based notification and you must be within 2 days of a milestone to even have the task execute and pick up any data.

If you wanted to force the code to run, you could modify the milestone due date for some milestones to be within the two day window.

Preferences

We finally get to add the preference support to the application. We've covered a few preliminary steps for preference support.

- Adding the Session Properties.
- Generating Session classes
- Configuring the preference hierarchy
- Coding the preference hierarchy

The next steps for preference support are

- Create the preference extension
- Create mappers to manage preference persistence at each hierarchy level
- Attach the preference extension to the mapper
- Create pages to display preferences for each hierarchy level
- Associate the preference pages with the appropriate hierarchy level

Create the Preference Extension

Go back to your solution in Visual Studio and add a new C# project as follows...

Item	Value	
Project Type	Visual C#	
Template	Class Library	
Name	Preferences	
Location	C:\NetQuarry\Customers\IssueTrak\Source\Extensions\Preferences	

In a process similar to when we created an new extension

Change the name of the project to IssueTrak.Extensions.Preferences

Change the name of Class1.cs to Preferences.cs

Add references to...

IssueTrak.Common.dll, NetQuarry Core EAP.Core.dll, NetQuarry Data Binding. Use the "Recent" tab of the Add Reference dialog for speed.

Also add a reference to EAP.Extensions.Preferences (under .Net). This is required for setting the base class of you preference extension to the Platform Preference Handler.

Drag the project.build file from IssueTrak.Common to IssueTrak.Extensions.Preferences

Open the project.build file in the IssueTrak.Extensions.Preferences component.

Change the solution value to "IssueTrak.Extensions.Preferences"

Drag the AssemblyInfo.cs file from IssueTrak.Common \Properties to IssueTrak.Extensions.Preferences \Properties, overwriting when prompted.

Open the AssemblyInfo.cs file in the IssueTrak.Extensions.Preferences \Properties.

Change the Assembly Title to "IssueTrak.Extensions.Preferences"

Change the Assembly Description to "IssueTrak Preferences Extension"

Right click on the IssueTrak.Extensions.Preferences project and choose Properties

Change the Assembly Name to IssueTrak.Extensions.Preferences

Change the Default Namespace to IssueTrak.Extensions (save and close the Properties)

In the Preferences.cs file, use this code

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IssueTrak.Common;
using IssueTrak.Data;
using NetQuarry;
using NetQuarry.Data;

namespace IssueTrak.Extensions
{
    public class XPreferences : NetQuarry.Preferences.Extension<IssueTrak.Common.Session>
    {
        //--- if you override any events you must call the equivalent base override event handler
        //--- as all the events are being handled by the platform extension
    }
}

```

Compile and register this extension in the studio.

Name	Value	Notes
Module	IssueTrak-preferences	
Name	XPreferences	
Component Type	Extension	
Component Name	IssueTrak.Extensions.XPreferences	
Assembly Name	IssueTrak.Extensions.Preferences.dll	
Assembly Path	C:\NetQuarry\Customers\IssueTrak\Source\Extensions\Preferences\bin\debug	
Assembly Path Prod	%NQROOT%\Apps\IssueTrak\bin	

Create Company Preferences

Create the Company Preferences Mapper

Go to the mappers list and create the following mapper

Item	Value	Notes
Module	IssueTrak-preferences	
Name	prefs_company	
View	xot_preferences	We have to give this mapper a view name, but the data will not be extracted directly from this table. The preference extension extracts the relevant preferences from the xot_preferences table and transforms the data to display and manages the persistence.
Data Source	IssueTrak	
Attributes	SkipCodeGeneration	This mapper will not be a typed mapper.

Move off the row to save, then select the prefs_company mapper. This time we will NOT perform a Create Fields. All the fields you create in this mapper are manually added.

Now create a standard picklist with the following options

Item	Value	Notes
Module	IssueTrak_preferences	
Name	yes-no-nobblank	
Type	StandardPicklist	
Attributes	Cache, LimitToList, StoreAltInt, NoNullEntry	There will be no blank entry option for comboboxes using this picklist. Only Yes, or No option will be provided.

And then in the Items subform add these items

Name	Notes
yes, AlternateKeyInt = 1	
no, AlternateKeyInt = 0	

The preference extension you created that derives from the platform extension that actually does all the work, constructs a DataTable of preference values for the selected preference level and owner ID. It determines what fields to put in the data table based on the fields defined in the mapper.

Field	Settings	Notes
company_id	Key Name: company_id Column Order: 10 Column Width: 0 CellType: TextBox .NET Type: System.Guid Data Type: uniqueidentifier Attributes: PK	
UsersViewDashboard	Key Name: UsersViewDashboard Column Order: 20 Column Width: 10 Picklist: yes-no-noblock CellType: ComboBox .NET Type: System.Boolean Data Type: bit Table: + Caption: View Dashboard On Startup?	

Go to the Extensions subform and attach the following extensions to the prefs_company mapper.

Extension	Notes
IssueTrak.Extensions.XPreferences	
ReadableAudit.Audit	Mark the UsersViewDashboard field with the "Audit" attribute

Create the Company Preferences Page

We will create the company preferences page on the company admin page as an in place navigate link. Go to the Pages List and select the company admin page.

Add a Page Elements with the following settings

Item	Value	Notes
Slot Name	ConsolePage	
Name	preferences	
Component	ConsoleDetail	
Order	60	

With that new Page Element selected, click on the Property sheet and set the following properties.

Property	Value	Notes
PaneAttributes	PaneNavigation, EditModeOn	EditModeOn forces the pane to be editable at all times and eliminates the need to click on the "Edit" button of a toolbar to make changes
ParentViewKey	company_id	
ViewKey	company_id	
Column	1	
Mapper	prefs_company	
Caption	Preferences	

Associate the Preference Page to the Correct Preference hierarchy level.

Go to the Preference Levels list and the Pages subform.

Create a new record with the value company!main page.

Preference Levels - (2 Rows)			
	Module	Preference Level	Parent Preference Lev
▶	IssueTrak-preferences	company	
	IssueTrak-preferences	individual	company
*			

Pages |

page_list - (1 Rows)

	Preference Page
▶	company!main ▼
*	

Add the Link to Company Preferences

The final step is to create a navigation element to get to the company preferences page.

Continuing in the Studio, go to the list of Navbars and select the navbar we created for the company admin page, "company_links"

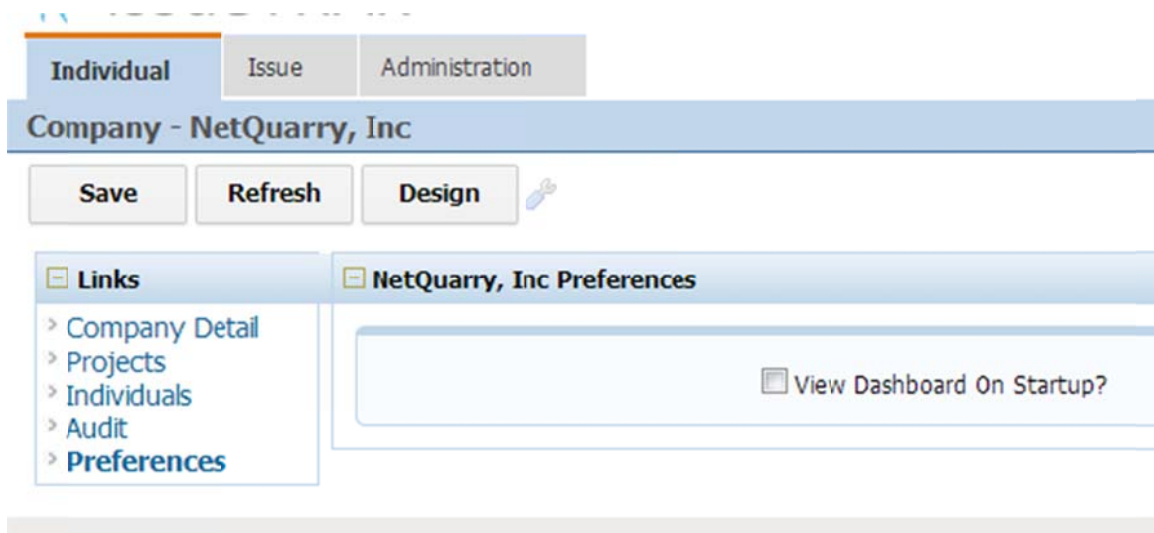
Add a new Target with the following settings.

Item	Value	Notes
Module	company	
Name	preferences	
Target Type	Pane	
Target	preferences	
Sort Order	50	

With that new Target selected, click on the Property sheet and set the following properties.

Property	Value	Notes
ParentViewKey	company_id	
ViewKey	company_id	
Caption	Preferences for [\$owner_name]	The \$ modify in the field reference syntax means to resolve the DISPLAY value of the field rather than the raw value.

Now login to the application. Navigate to the Admin page, Companies List and drill down to a company. Click on the link to Preferences and the following page will appear



Notice that the page is always editable. Any changes to the preferences of the company will be audited to the company record.

Tweaks to Preference Page

Ideally we would have an indication of which preferences we are changing. Let's add that now.

Add the Company Name

In the Studio, go to the `prefs_company` mapper. Add a field with the following settings.

Field	Settings	Notes
owner_name	Key Name: owner_name Column Order: 15 Picklist: companies Column Width: 30 CellType: Label .NET Type: System.Guid Data Type: uniqueidentifier Attributes: ExcludeFromSelect, Locked Style: font-weight: bold; border-bottom: ridge 1px silver; color: #666666; padding-bottom: 10px; padding-top: 20px Caption: Company	

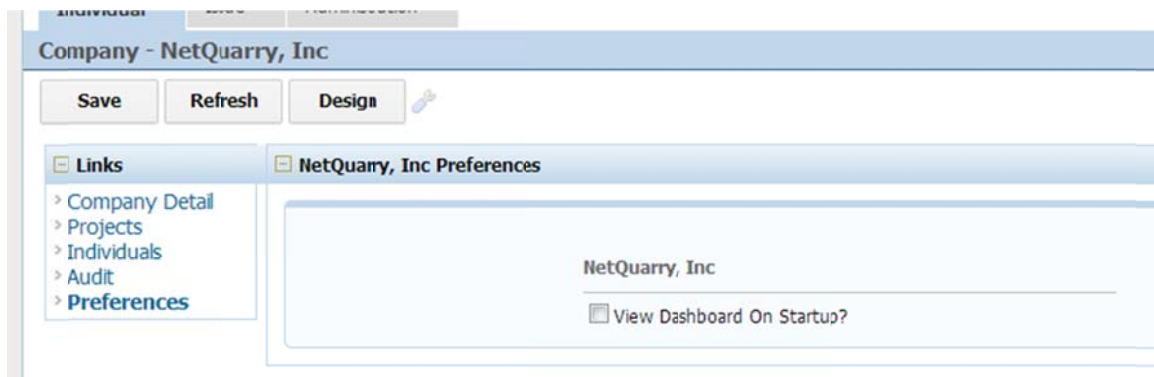
Set the Company Name

In the Preferences extension in Visual Studio, override the RowCurrent Event handler with the following code.

```
base.RowCurrent(sender, e);
//--- identify the primary key field for this preference mapper
IField f = sender.Fields.Find("", FieldFindType.PK);
if (f != null)
{
    //--- and assign the value to the "owner_name" field to be resolved by a picklist
    sender.Fields.SetValue ("owner_name", f.Value, 0, FindBehaviour.OkIfNotFound);
}
```

This code determines the primary key field of the preference mapper and sets its value to the value of the "owner_name" field on the mapper. If your preference mappers have an owner_name field with a picklist to resolve the raw value, this will work all the time. If the field doesn't exist to set, the FindBehaviorAttribute specifies it's ok if the field is not found and an error will not be thrown.

Compile the extension and log back into the application and navigate to the company preferences page.



Create the Individual Preferences

We will now add preferences to the individual. The individual page is not using in place navigation so the steps will be slightly different. We'll be creating a dedicate preference page. Another difference between the two sets of steps is that we require you to create a SQL picklist to resolve the individual_id to a name. We will create the picklist as a "late bound" picklist purely to resolve an individual_id to a name, but won't be usable as a picklist source for selecting in a combobox.

Create the Individual "Late Bound" Picklist

Create a new SQL Picklist with the following settings.

Item	Value		Notes
Module	individual		
Name	individual_lb		
Data Source	IssueTrak		
Source	SELECT individual_id, full_name, null, CASE ISNULL(is_deleted,0) WHEN 0 THEN 1 ELSE 0 END from individual with(nolock) order by 2		
Type	SQL		
Attributes	Cache, LimitToList		

Move off the row to save and then select the individual_lb picklist and click on the property sheet and add the following property.

Property	Value	Notes
LateBoundSQL	SELECT individual_id, full_name, null, CASE ISNULL(is_deleted,0) WHEN 0 THEN 1 ELSE 0 END from individual with(nolock) WHERE individual_id={{KEY}} order by 2	Setting the LateBoundSQL property is what makes a SQL picklist "Late Bound". At run time the value needing to be resolved replaces the {{KEY}} value in the WHERE clause of the late bound sql. The returned data is cached in memory against the key value so a subsequent resolution is just a cache hit, rather than a database hit.

Create the Individual Preferences Mapper

Go to the mappers list and create the following mapper

Item	Value	Notes
Module	IssueTrak-preferences	
Name	prefs_individual	
View	xot_preferences	
Data Source	IssueTrak	
Attributes	SkipCodeGeneration	This mapper will not be a typed mapper.

Move off the row to save, then select the prefs_individual mapper. Click on the Fields subform and add the following fields

Field	Settings	Notes
individual_id	Key Name: individual_id Column Order: 10 Column Width: 0 CellType: TextBox .NET Type: System.Guid Data Type: uniqueidentifier Attributes: PK	
owner_name	Key Name: owner_name Column Order: 20 Picklist: individual_lb Column Width: 30 CellType: Label .NET Type: System.Guid Data Type: uniqueidentifier Attributes: ExcludeFromSelect, Locked Style: font-weight: bold; border-bottom: ridge 1px silver; color: #666666; padding-bottom: 10px; padding-top: 20px Caption: Individual	

Field	Settings	Notes
UsersViewDashboard	Key Name: UsersViewDashboard Column Order: 30 Column Width: 10 CellType: ComboBox Picklist: yes-no-noblink .NET Type: System.Boolean Data Type: bit Table: + Caption: View Dashboard On Startup?	

Go to the Extensions subform and attach the following extensions to the prefs_individual mapper.

Extension	Notes
IssueTrak.Extensions.XPreferences	
ReadableAudit.Audit	Mark the UsersViewDashboard field with the "Audit" attribute

Create the Individual Preferences Page

Go to the Pages List and add a new page with the following settings.

Item	Value	Notes
Module	IssueTrak-Preferences	
Name	prefs_individual	
Moniker	Individual Preferences	
Template	TabbedSubformTemplate.aspx	
Mapper	prefs_individual	

With that new Page selected, click on the Property sheet and set the following properties.

Property	Value	Notes
Caption	Preferences for [\$owner_name]	

Now add a Page Elements with the following settings

Item	Value	Notes
Slot Name	Main	
Component	phantomdetail	

Associate the Preference Page to the Correct Preference hierarchy level.

Go to the Preference Levels list and the Pages subform.

Create a new record with the value `issuetrak-preferences!prefs_individual` page.

Create UI to Get To Individual Preferences Page

Create a navigation element to get to the individual preferences page.

Continuing in the Studio, we have to create a navigator to provide a nav target and then apply that navigator on a MiniNav component on the individual!main page.

Create the Navigator

Go to the Navigators list and create a new navigator with the following settings

Item	Value	Notes
Module	individual	
Name	individual_links	
Type	Navbar	
Attributes	AddParentInfo	This forces parent information on to all of the navbar's targets. This ensures that when you click on the workflow link it takes you to the wizard filtered to the correct issue.

Select the "individual_links" navbar and then in the Targets subform below, add a new Target with the following settings.

Item	Value	Notes
Module	individual	
Name	preferences	
Target Type	Page	
Target	issuetrak-preferences!prefs_individual	
Sort Order	10	

With that new Target selected, click on the Property sheet and set the following properties.

Property	Value	Notes
ParentViewKey	individual_id	
ViewKey	individual_id	
Caption	Preferences	

Add a MiniNav to the Individual Page

Click on the Pages link (or type Alt+P). Make sure the individual!main page is selected, then click on the Page Elements subform.

Create a new Page Element with the following settings

Item	Value	Notes
Slot Name	ConsolePage	
Name	individual_links	
Component	MiniNav	
Order	20	

With that new Page Element selected, click on the Property sheet and set the following properties.

Property	Value	Notes
Column	1	
Navigator	individual_links	
Caption	Links	

Set the Individual Name

There is no code change required for the preference extension to handle this new preference level. The code we added originally is generic.

Navigate to the preferences page of an individual to confirm everything is hooked together.

The screenshot shows a web application interface. At the top, there are three tabs: 'Individual' (selected), 'Issue', and 'Administration'. Below the tabs is a breadcrumb trail: 'Individual List :: Individual :: Preferences for admin guy'. Under the breadcrumb trail are four buttons: 'New', 'Save', 'Refresh', and 'Design' (with a small icon). The main content area shows the name 'admin guy' and a checkbox labeled 'View Dashboard On Startup?' which is currently unchecked.

Using Preferences

Preference Behavior

When you view preferences, it recursively loads the set of preferences for the current level from the highest level of inheritance, down to the required level, as defined by the preference hierarchy in the metadata.

Go to the company preferences page. Check the box for "View Dashboard On Startup" to make it true. Log out, and log in again, navigate to the preferences page for the same company. It will still be checked.

Now navigate to an individual's preferences. Notice that when we first tested the Individual Preferences, the "View Dashboard On Startup" was unchecked. Now, when you visit the preferences for an individual, it shows the preference as checked. This is because the individual has inherited its preference from the parent company.

On the individual preference, you can uncheck the "View Dashboard On Startup" setting and save. That individual now has a specific preference set for "View Dashboard On Startup" and overrides the parent company preference. From this point onwards, any changes to the parent company preference are ignored as the individual has a specific overriding preference.

Using Preferences in Code

Any time you have access to the local session object you can simply access the value of that preference for the current logged in user. Additionally, you can instantiate a session instance for any object that supports preferences, via the "CreateInstance..." methods defined in the IssueTrak.Common.Session class.

We'll now implement some code that refers to the "View Dashboard On Startup" preference. If that preference is set to true, then the user will initially logon to the Report Dashboard page. If the preference is set to false, the user will initially logon to the next available page.

Go to the Startup.cs file in the IssueTrak.Common. In the Startup application extension class, create a new function called SetPermissionsBasedOnPrefs, as follows...

```
private void SetPermissionsBasedOnPrefs(IAppContext sender)
{
    PageInfo pi = null;

    //--- get the session from the application
    IssueTrak.Common.Session itSess = (IssueTrak.Common.Session)sender.Session;

    //--- get the page info object of the dashboard page
    pi = sender.PageInfos["issuetrak-home!dashboard"];
    if (pi != null)
    {
        //--- make sure the page exists then test the user's preference
        if (!itSess.UsersViewDashboard)
        {
            //--- remove the page from the collection of pages
            sender.PageInfos.Remove(["issuetrak-home!dashboard"]);
        }
    }
}
```

Then in the same class add this override to the WakeUp event handler

```
public override void WakeUp(IAppContext sender, EAPEventArgs e)
{
    if (!IsPasswordUser(sender, sender.UserContext))
    {
        SetPermissionsBasedOnPrefs(sender);
    }
}
```

Then in the existing AfterLoad event handler, make the following highlighted changes

```

public override void AfterLoad(IAppContext sender, EAPEventArgs e)
{
    //--- do NOT continue with this if this is reset password
    if (IsPasswordUser(sender, sender.UserContext))
        return;

    else
    {
        IssueTrak.Common.Session session = sender.Session as IssueTrak.Common.Session;

        session.Init(); //--- initialize a session

        SetPermissionsBasedOnPrefs(sender);
    }
}

```

Compile the IssueTrak.Common project.

Before you test the code, you need to modify the page permissions for the issuetrak-home!dashboard page. Go to the Pages list and select the issuetrak-home!dashboard page. In the permissions subform tab, make sure all profiles can "read" the dashboard page.

Now login to the application. Depending on the preference settings at the company and individual level, you may, or may not be able to see the dashboard page. Make changes to preferences and logout and login as that user to see the effects of changing the preferences.

Using Preferences in Metadata

Another way to consume preferences, is to declare a preference as requiring an embedded function. With an embedded function wrapper to a preference, you can use the embedded function as a default value in meta data, so new records are created with values based on the set of user's preferences.

Create New Session Properties

In the NetQuarry Studio, go to the Session Properties and create two new properties with the following settings

Field	Settings	Notes
DefaultIssueProject	Name: DefaultIssueProject Type: String Attributes: DynamicOnly, GenEmbeddedFunc, SessionPersist	The preference is essentially going to store an integer value but a null integer value is cast to the value 0. we want null to be null, not 0
DefaultIssueComponent	Name: DefaultIssueComponent Type: String AttributesDynamicOnly, GenEmbeddedFunc, SessionPersist	

Generate the Session objects and compile the IssueTrak.Common project.

In the generated session class, in addition to declaring the new session properties, the code to register the values of these session properties as embedded functions has been added.

```
_appCxt.RegisterEmbeddedFunction("DefaultIssueComponent",  
EAPUtil.ToString(_defaultComponent));  
_appCxt.RegisterEmbeddedFunction("DefaultIssueProject", EAPUtil.ToString(_defaultProject));
```

Add the new Preferences to the prefs_individual mapper by adding the following fields to the prefs_individual mapper.

Field	Settings	Notes
lblIssuePrefs	<p>Key Name: lblIssuePrefs</p> <p>Column Order: 30</p> <p>Column Width: 30</p> <p>CellType: Label</p> <p>.NET Type: System.String</p> <p>Data Type: varchar</p> <p>Attributes: ExcludeFromSelect, Locked</p> <p>CellTypeAttributes: StaticFromCaption</p> <p>Style: font-weight: bold; border-bottom: ridge 1px silver; color: #666666; padding-bottom: 10px; padding-top: 20px</p> <p>Caption: Default Issue Preferences</p>	
DefaultIssueProject	<p>Key Name: DefaultIssueProject</p> <p>Column Order: 40</p> <p>Picklist: projects</p> <p>Column Width: 20</p> <p>CellType: ComboBox</p> <p>.NET Type: System.Int32</p> <p>Data Type: int</p> <p>Attributes: Audit</p> <p>Caption: Default Project</p>	

Field	Settings	Notes
DefaultIssueComponent	Key Name: DefaultIssueComponent Column Order: 50 Picklist: components Column Width: 20 CellType: ComboBox .NET Type: System.Int32 Data Type: int Attributes: Audit Discrim: [DefaultIssueProject] Caption: Default Component	

Now go to the "issue" mapper and make the following field changes...

Field	Settings	Notes
project_id	DefaultValue: !fnDefaultIssueProject()	
component_id	DefaultValue: !fnDefaultIssueComponent()	

Now login to the application. Go to an individual's preference page and set up a Project and Component default preference for Issues.

Individual List :: Individual :: Preferences for admin guy

New Save Refresh Design

admin guy


☒ View Dashboard On Startup?


Default Issue Preferences

Default Project: IssueTrak

Default Component: Company

Log out and log into the application as that user and create a new issue. The new issue will have default values for project and component

Issues  Issue Detail [new]

Save Refresh Design Actions 

Summary: *

Description: *

Assigned User:

Category: *

Project: * IssueTrak

Component: * Company

Priority: *

Asynchronous Requests

The platform supports a number of asynchronous request mechanisms. Perhaps the easiest is to implement an immediate save mechanism on one or more fields.

Immediate Save

Immediate Save on MiniDetail

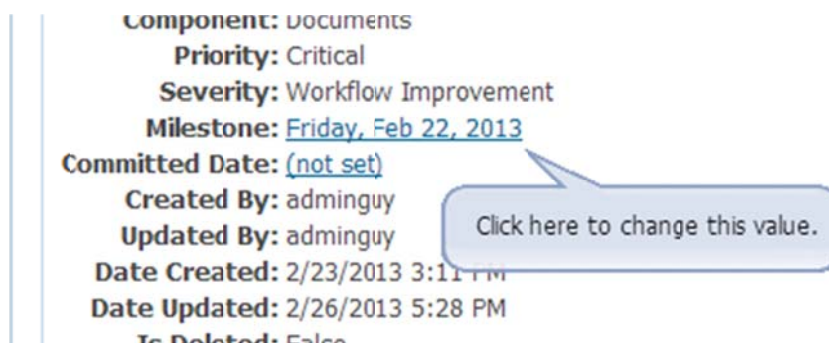
The MiniDetail console pane is based on a static HTML template. However, we can make a picklist, date field and check box editable (and immediately saving) via a special field token syntax in an HTML template.

Go to the issue template %NQROOT%\Templates\IssueTrak\issue\ issue-detail-layout.html and make the following highlighted modification

```
<div class="mini-detail">
<table>
<tr><td class="mini-cap">Issue Number:</td><td>{{issue_number}}</td></tr>
<tr><td class="mini-cap">Summary:</td><td>{{summary}}</td></tr>
<tr><td class="mini-cap">Description:</td><td>{{description}}</td></tr>
<tr><td class="mini-cap">Assigned User:</td><td>{{assigned_user_id}}</td></tr>
<tr><td class="mini-cap">Status:</td><td>{{status_id}}</td></tr>
<tr><td class="mini-cap">Category:</td><td>{{category_id}}</td></tr>
<tr><td class="mini-cap">Project:</td><td>{{project_id}}</td></tr>
<tr><td class="mini-cap">Component:</td><td>{{component_id}}</td></tr>
<tr><td class="mini-cap">Priority:</td><td>{{priority_id}}</td></tr>
<tr><td class="mini-cap">Severity:</td><td>{{severity_id}}</td></tr>
<tr><td class="mini-cap">Milestone:</td><td>{{+milestone_id}}</td></tr>
<tr><td class="mini-cap">Committed Date:</td><td>{{+committed_date}}</td></tr>
<tr><td class="mini-cap">Created By:</td><td>{{created_by_id}}</td></tr>
<tr><td class="mini-cap">Updated By:</td><td>{{updated_by_id}}</td></tr>
<tr><td class="mini-cap">Date Created:</td><td>{{date_created}}</td></tr>
<tr><td class="mini-cap">Date Updated:</td><td>{{date_updated}}</td></tr>
<tr><td class="mini-cap">Is Deleted:</td><td>{{is_deleted}}</td></tr>
<tr><td class="mini-cap">Revision:</td><td>{{revision}}</td></tr>
</table>

</div>
```

Save the change. Log out of the application and log in and drill down to an issue. You will see the milestone field has been decorated with a link.



Click on the link. Each click will cycle to the next milestone option in the picklist. During each change, an ajax request is made by the platform to save the change on that field. This save is executed through the mapper, so the relevant mapper extension events will be fired. For the date field, the link pops up the date picker where you can choose (or clear) a date.

You can verify the save occurs by refreshing the page and see the audit history modifications.

Severity: Workflow Improvement
Milestone: [Friday, Mar 22, 2013](#)
Committed Date: [2/27/2013](#)
Created By: adminguy
Updated By: adminguy
Date Created: 2/23/2013 3:11 PM
Date Updated: 2/26/2013 5:28 PM
Is Deleted: False
Revision: 5

Last 10 Notes (Click to View All) (3) » New

Related Issues (Click to View All) (0) » Add

Last 10 Audit Events (Click to View All) (10)

Changed Items	Change Description
Committed Date	Committed Date: 2/27/2013
Milestone	Milestone: Friday, Mar 08, 2013 Friday, Mar 22, 2013

Immediate Save on MiniList

This feature is also supported on MiniList components. However, to configure this requires a little more work. The field has to have the EditInList attribute and the MiniList pane must have the AllowEditInList property set to true.

We currently have no MiniLists that could provide a way to view this functionality, so we will add a MiniList pane on the individual!main page that will show all the OPEN issues assigned to the user.

On the mini list, we will show the summary, milestone and priority fields.

From the MiniList, we should provide a link from the MiniList caption that takes you to the full list of issues, filtered by the assigned user and open status.

Use Ajax to Lookup and Populate field on Client

A slightly more complicated implementation is to fire an Ajax event through to a mapper and have it determine a set of return values (via a JSON object) and apply those values to client side fields.

We're going to provide an example for this on the issue pages. We are going to add code to fire an Ajax request to a mapper when the Component field is changed. The Ajax request will look up which user is associated with that component and apply that user information into the user fields on the same page.

If you remember, we did perform this type of lookup in the RowBeforeInsert of the Issue mapper extension.

Add JavaScript Function Handlers to Fields

Go to the NetQuarry Studio and the "issue" mapper.

Set the following field properties on the component_id field.

Field	Settings	Notes
component_id	OnChange: OnComponentChange(this);	

Tell the Application Where the JavaScript File Lives

Go to the Application list and select the IssueTrak application. In the property sheet, set the JavaScript property to: apps/issuetrak/script/issuetrak.js

Open the .js file, %NQROOT%\Apps\IssueTrak\Script\IssueTrak.js and add the following code.

```
//--- handles the change of component_id field
function OnComponentChange(component)
{
  if (component.value)
  {
    var aui = GetSibling(component, 'assigned_user_id');
    var au = GetSibling(component, 'assigned_user');
    //--- only continue with ajax request if assigned user field is visible (the id field is
hidden of course).
    if (aui && au && au.type != "hidden")
    {
      //--- for convenience pass the id's of the fields we are going to set.
      params = "aui=" + aui.id + "&au=" + au.id + "&nodata=1";
      //--- fire the request against the component mapper
      //--- you can only fire events onto existing records. never new records (like the issue
mapper)
      RowAjaxRequest('component', component.value, 'get_default_assignee', params, function() {
    });
    }
  }
}

//--- this function is called by the Ajax response handler
function UpdateDefaultAssignee(rsp)
{
  //--- evaluate the json response
  var defAssignee = eval(rsp);

  //--- and set the values into the client side fields
  $('# + defAssignee.aui).val(defAssignee.assigned_user_id)
  $('# + defAssignee.au).val(defAssignee.assigned_user)
}
```

IMPORTANT

The JavaScript file you modified is part of the platform installation. Therefore, if you reinstall a new platform version, the changes you made to the JavaScript file will be lost forever.

To avoid this loss, you should copy your modified JavaScript file to the folder C:\NetQuarry\Customers\IssueTrak\Config\web\Script

After the install has been completed, the Install.bat file contains a step to copy the backed up IssueTrak.js file to the %NQROOT%\Apps\IssueTrak\Script folder.

Create the Component Mapper Extension

In your IssueTrak solution in visual studio, create a TypedMapper extension based on the Component TypedMapper (we have already declared the component TypedMapper in the Project.cs file of IssueTrak.Common project). Follow the previous methodology regarding naming conventions, references and copying and moving files, etc.

In your XComponent.cs file, use the following code...

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NetQuarry;
using NetQuarry.Data;
using IssueTrak.Common;
using IssueTrak.Data;

namespace IssueTrak.Extensions
{
    public class XComponent : IssueTrak.Extensions.TypedExtensionBase<Component>
    {
        public override void RowAjaxRequest(Component sender, string requestName,
        AjaxRequestEventArgs e)
        {
            switch (requestName)
            {
                case "get_default_assignee":
                    //--- when we get this event, we are on the current row already so it's an easy get
                    for default assignee

                    JsonSerializer json = new JsonSerializer("DefaultAssignee");
                    json.Add("aui", e.Params["aui"]);
                    json.Add("au", e.Params["au"]);
                    json.Add("assigned_user_id", sender.user_id);
                    json.Add("assigned_user", sender.Fields.user_id.DisplayTextGet(0));

                    e.JsonResponse = "UpdateDefaultAssignee(" + json.ToString() + ")";

                    break;
            }
        }
    }
}
```

We are handling the RowAjaxRequest mapper event that is fired through the platform by the JavaScript

We're taking advantage of the fact that the mapper has been re-queried to the correct record. We have extracted the parameters we sent on the original request and packaged up those values, together with the necessary user data in a JSON object.

Compile the Component Extension.

Tweak the component mapper

Register the Extension in the NetQuarry Studio and attach the extension to the "component" mapper.

Also on the component mapper, go to the Fields subform. Make the following field changes

Field	Settings	Notes
user_id	PickList: user_list Cell Type: Combobox	Setting the picklist ensures the user_id is resolved to a name and the RowAjaxRequest code (sender.Fields.user_id.DisplayTextGet(0)) correctly resolves the user id to a name.

Now login to the application and create a new issue. When you select a component from the dropdown the assigned user information will be populated.

The image displays two side-by-side screenshots of the NetQuarry application's 'Issue' form. Both screenshots show the 'Summary' field with the text 'async request' and the 'Description' field with the text 'when you select a component, the assigned user will be populated'. The 'Assigned User' field is a dropdown menu. In the left screenshot, the 'Component' dropdown is set to 'Issue', and the 'Assigned User' field is empty. In the right screenshot, the 'Component' dropdown is set to 'Issue', and the 'Assigned User' field is populated with 'Bill Gates'. The 'Category' dropdown is set to 'Feature', the 'Project' dropdown is set to 'IssueTrak', and the 'Priority' dropdown is set to 'Issue'. The 'Severity' dropdown is set to 'Audit', the 'Milestone' dropdown is set to 'Email', and the 'Committed Date' field is set to 'Issue'.

Localizing Notification Message

In this tutorial we have implemented quite a bit of functionality that presents information to the user in the way of alerts and status messages. These messages were simply hard coded into the source.

Clearly there are a couple of major faults with this approach.

1. There's no way to change the text unless you recompile the code and reinstall
2. The text is not localizable into other languages

The way round these limitations is to move the text strings into the metadata and refer to those strings in the code.

To access the text in metadata, you refer to the `TextItem` in the `TextItems` collection on an object. The hardest part of using `TextItems` is to decide which object you will associate the `TextItem` to.

For example.

If you have created an extension that programmatically adds a menu command to the UI, you don't want to have to add a custom text string to each mapper you attach the extension to. Instead you will associate the text directly with the extension. The common object is the extension.

If you have written a function to perform a common task, and that function is in a `TypedMapper` (`SendEmailNotification` in `Issue Typed mapper`), you would want to associate the text with the mapper, rather than an extension, or a page that uses that mapper. The common object is the mapper object.

So, having said the hardest part is choosing where to associate the text, it turns out to be not a very hard thing to do.

We'll go back through the issue extensions and implement some localizable strings.

Localizing Strings in the XTMIssue extension

In the bulk update code that responds to the Bulk Modify Issues command, we register a filter that will be used keep track of which issues to process and which issues to display at the end of the bulk update process. On that final page, the Platform uses the text associated with the filter to display contextual information about the list being displayed.

In the PerformBulkUpdate function, modify the code as highlighted.

```
private void PerformBulkUpdate(Issue sender)
{
    ///--- declare some string consts. either locally, or at class root if used across the class
    const string csFilterNameText = "IDS_BULK_MODIFY_NAME";
    const string csFilterNameDescription = "IDS_BULK_MODIFY_DESC";

    ///--- save of a 'bucket' of keys matching those selected and get its ID
    string filterName = "Issue Bulk Update";
    FilterAttributes fa = FilterAttributes.Static | FilterAttributes.Temp |
    FilterAttributes.KeyGuid | FilterAttributes.Hidden;
    ///--- save a filter of the selected records
    SavedFilter sf = sender.Mapper.Exec(MapperExecCmds.FilterSave, fa, filterName) as SavedFilter;

    ///--- get the localized the filter strings
    ///--- the default values are valid strings, but just different enough from the real string so
    ///--- that you recognize
    ///--- if the text item load isn't working for some reason
    ///--- we are pulling these from this, the Extension object
    string fltName = this.TextItems.GetText(csFilterNameText, "bulk modified issues");
    string fltDesc = this.TextItems.GetText(csFilterNameDescription, "issues affected by your bulk
changes");

    ///--- then register this filter
    string fltQP = SavedFilter.RegisterReqFilter(this.Application, sf.Filter, fltName, fltDesc,
"issue!list");

    ///--- navigate to an issue modification wizard page
    ///--- navigating as new so we don't need a PK to go to a specific record
    ///--- and pass along the ID of the saved filter. These are the issues we are going to
    process.
    sender.Application.Navigate("issue!wiz_bulk_modify", null, "flt_id=" +
EAPEncode.ForUrl(fltQP), "new");
}
```

Compile the extension.

Now go to the NetQuarry Studio and the list of extensions (under components). Select the Issue Typed mapper extension. Then click on the User Defined Text subform tab.

Add the following text items

Text Item	Value	Notes
IDS_BULK_MODIFY_NAME	Item Name: IDS_BULK_MODIFY_NAME Text: Bulk Modified Issues	
IDS_BULK_MODIFY_DESC	Item Name: IDS_BULK_MODIFY_DESC Text: Issues affected by your bulk changes	

Log in to the application and bulk modify some issues. The text items displayed should be those from the metadata and not the code.

When the items are not loading correctly...



And correctly...



This Completes the Tutorial

This completes the NetQuarry tutorial.

See <http://www.netquarry.com> for more information. For the latest API documentation, visit <http://help.netquarry.com> .