# MyIP Permissions Design

## GLOSSARY

“root table” – this is the “Filter” as defined on the role. Currently only tblInvention and tblAgreement, now extended to Products

## DESIGN

For root tables (see glossary) that need to be checked for Permissions and anything that hangs off them, there is now a consistent approach encapsulating the logic mainly in one place, the SQL database.

### SQL Database

spCreateUserViews creates 2 views on every relevant table (as explicitly defined by the developer) for a user. spInitializeUserViews calls spCreateUserViews for every user (to be run on deployment – already part of ApplyScripts.bat). spCreateUserViews also gets executed if you change a user’s role OR if you change a role’s permissions (via spUpdateRoleViews).

There is a read view – prefixed by the letter r i.e. admin.rtblInvention

And a write view i.e admin.tblInvention

The permissions logic is encapsulated in table valued functions tfnMyxxxxxxx e.g.tfnMyOpportunities. These provide the 2 key fields aAllowRead and aAllowUpdate. (NB These are the 2 fields that should be used NOT AllowRead and AllowUpdate which are the default fields on the base table and which also show up in the TVF because of the use of SELECT \* (you could potentially change SELECT \* to list the fields but then maintenance would be more difficult)

So the user views then use these aAllowRead and aAllowUpdate to calculate for a given table (which might have links to more than 1 root table) the overall AllowRead and AllowUpdate values (ie you might have aAllowRead on the Opportunity but not on the Agreement. So userxxx.tblInventionAgreement will “AND” permissions on both).

Sprocs like the enquiry sprocs can also use these Table valued functions – either as a join or in the main select. NB **REMEMBER to use aAllowRead and aAllowUpdate as the base table fields AllowRead and AllowUpdate will just return True.**

The write view has an update and delete trigger. This trigger checks the value of AllowUpdate and errors if that is false, thereby enforcing permissions checks at the database layer for updates.

The read view masks all data where possible except the Primary Key and any field reference by another table (eg Opportunity Number referenced by tblAlarms.CaseNumber).

### .NET App

Within the app the default data context has been changed to be a ReadContext. This uses the current User to map reads to the user view (though this needs to be set up within the PermissionedTableTypes list in ContextFactory.cs – only the user views you create in spCreateUserViews which are also listed here will actually be used. So you can add tables to spCreateUserViews and they will not be used until they are in this list. You cannot do it the other way round though, if it is in the list in ContextFactory, it MUST be a valid user view otherwise you will get an error.)

Using this Read context therefore will still return every record but those that you should not be able to see will not have any meaningful data – the reason that we still return them is for FK lookups. E,g if you are looking at a Patent Application that you have permission for, but you do not have permission to see the Parent – we must not return nothing otherwise that will risk updating the parent to nothing on an update. So we still return the parent’s referenced key field but its data fields are masked.

For a standard read of the main entity you can now ignore what permissions logic needs to be applied and just use the predicate “where AllowRead” as any required rules will be applied in the view. Until this predicate is used, you may see the “Out of Scope” records in your resultset but obviously with meaningless data.

For updates and deletes, no checks need to be done but you must use a WriteContext (ContextFactory.CreateWriteContext()). The db trigger will error if permission is denied, though obviously you would hope not to have got this far anyway.

Unfortunately Linq to SQL does not work well with insert triggers otherwise I would have used that too. Linq To SQL generates an insert statement that uses SCOPE\_IDENTITY() for tables with an IDENTITY column. Insert Triggers lose SCOPE\_IDENTITY() so we must use an alternative solution. So where permissions checks need to be done for an Insert (and that means checking the resultant record has AllowUpdate), you must wrap SubmitChanges in a TransactionScope and check the value of AllowUpdate afterwards (see InventioDAL.AddInvention for an example).

### To Extend this system

1. To add further tables (linked to the *root tables*) which are not currently included in the checks:
   1. Update spCreateUserViews.
      1. Add tables to the control list
      2. Change the @xxxlinks variables to show how the new tables are linked back to the root
   2. Add tables to control list in ContextFactory
2. To add new root tables
   1. Add a Table Valued function that encapsulates the permissions for that root table (see tfnMyOpportunities/tfnMyAgreements/tfnMyProducts)
   2. Update spCreateViews
      1. Use new variable similar to @allOpps and @allAgreements
      2. Add new tables to control list with a new flag
      3. Add new @xxxlinks variables to show how each table links back to the root
      4. Change the view creation statements to use the table valued functions in the same way that the existing ones do
   3. Add tables to control list in ContextFactory
   4. Change ManageUsers.aspx.cs EditUserRole to check if permissions for role have changed for this new type of root permission
   5. Change RolePermissionDAL.Save tocheck this new type of root permission

Within the app you may want to ensure that Inserts are wrapped in a transaction and a check for AllowUpdate is done before committing.

To use the AllowRead and/or AllowUpdate fields within the app you need to make sure they exist on the base table with a DEFAULT of 1. Within the model they should be defined as ReadOnly and AutoDBGenerated (you can just copy and paste from tblInvention)