**Read access to Universal Housing for the CRM system**

**Introduction**

Web services are being created by Civica to allow data to be created and updated in UH, but the CRM system will spend more time reading data than updating it. Civica could be asked to develop a full range of read services, but it is felt that a direct data access approach is probably better.

This approach has the following advantages:

* Development will be quicker and cheaper
* It’s much easier to amend the views than the web services, as no liaison with a third party is required
* This increased flexibility removes the requirement for extensive upfront business analysis and hence allows a more ‘agile’ approach to development
* Less expertise is required from OHG’s point of view
* Data access will be quicker as there are none of the additional overheads associated with services

There are of course disadvantages:

* A competent OHG employee will be required to create and maintain the views
* It would be sensible to share this knowledge with the CRM supplier so that maintenance is not reliant on an individual; this will be costly.
* Civica may make updates to the system that ‘break’ the views or – worse – cause them to behave in an unintended manner.

**Approach to read data access**

The simplest way of retrieving data is by directly accessing the database via views or stored procedures. Views will generally be preferred unless there are performance advantages to using stored procedures.

An analysis of the indexes on the tables needs to be undertaken to ensure that they are optimised for the intended data access methodology, since performance is critical. If indexes are required, it is perhaps best to seek Civica’s approval before applying them, though in the past they have not objected to this.

It is worth noting that it is fairly common for long-running batch processes to occur throughout the day which lock the tables. Therefore, use of NOLOCK / READ UNCOMMITTED is recommended for simple querying (UH itself uses this). For situations involving data updates, most if not all UH tables have a timestamp column which can be tested to see if the data has been changed since being read. This can help solve data integrity issues that would otherwise arise from using this access methodology, but data integrity will need to be considered carefully in the design of the data access strategy.

**Lookup objects**

Civica’s services generally return DTOs (Data Transfer Objects) which are essentially structured exactly in the same way as the source tables. Since the DTOs include codes and not their equivalent descriptions, it will be necessary to provide lookups for display and entry purposes.

*server-side services*

To implement the services, it would be possible to either provide one service (GetAllLookups) that provides all lookups, several grouped functionally (GetPropertyLookups, GetTenancyLookups etc.) or by some other division, or provide them singly (GetPropertyTypeLookups, GetEthnicityLookups etc.). The advantage of a single call is that it reduces latency, but this is probably too unwieldy. The service-per-lookup scenario would require too many calls, so functional grouping is probably the best compromise. This would also enable a client to call several services at once asynchronously. There are, however, a lot of lookups in the property table, for example, with most probably being unused by the client. It may therefore be good to be able to configure which lookups are returned by the services.

In terms of the structure of the service, it could be along the lines of the following XML fragment:

<Lookups>

<Lookup>

<Name>HousingOfficer</Name>

<Dictionary>

<Item Enabled=”True”>

<Key>001</Key>

<Value>Jeremy Riches</Value>

</Item>

<Item Enabled=”True”>

<Key>002</Key>

<Value>Stephen Shortall</Value>

</Item>

…

</Dictionary>

</Lookup>

<Lookup>

<Name>AreaOffice</Name>

<Dictionary>

<Item Enabled=”True”>

<Key>001</Key>

<Value>Coleshill</Value>

</Item>

<Item Enabled=”True”>

<Key>002</Key>

<Value>Dudley</Value>

</Item>

<Item Enabled=”False”>

<Key>003</Key>

<Value>Leeds</Value>

</Item>

…

</Dictionary>

</Lookup>

…

</Lookups>

*client-side implementation*

In terms of the client side, the lookups may be implemented as a number of classes deriving from a ‘UHLookup’ object. I have included sample C# code to illustrate one possible implementation of this as an appendix. This is extensible so that, in future, other information could be added to the lookups, such as display order (if UH were ever to have this feature). Each client using these lookups would then call the services upon start-up and populate these static classes with data. Both the initial start-up code and the classes should be generated using a code generator to avoid error.

It would be in OHG’s and Civica’s interests for Civica to provide the C# client code for this, as they can not only sell this as part of their package but they can ensure that a consistent approach is used at all sites thus making the consuming application more reliable. For any sites not using .NET, this still provides an architectural framework that could be copied.

**Implementation of lookups on the client**

namespace UHLookups

{

public class LookupDetail

{

public string Description { get; set; }

public bool Enabled { get; set; }

public LookupDetail() { }

public LookupDetail(string ItemDescription, bool ItemEnabled)

{

Description = ItemDescription;

Enabled = ItemEnabled;

}

public override string ToString()

{

return Description;

}

}

public abstract class UHLookup

{

private static Dictionary<string, LookupDetail>

dict = new Dictionary<string, LookupDetail>();

public static Dictionary<string, LookupDetail> Dict

{

get

{

return dict;

}

}

static UHLookup() { }

public static LookupDetail Lookup(string LookupRef)

{

if (dict.ContainsKey(LookupRef))

{

return dict[LookupRef];

}

else

{

return null;

}

}

public static int Add(string Key, LookupDetail Value)

{

dict.Add(Key, Value);

return 0;

}

}

public class AreaOffices : UHLookup { }

public class HousingOfficers : UHLookup { }

//Plus other lookup types as required

}