Custom Sub-Entities SDK Guide

Contents

Overview

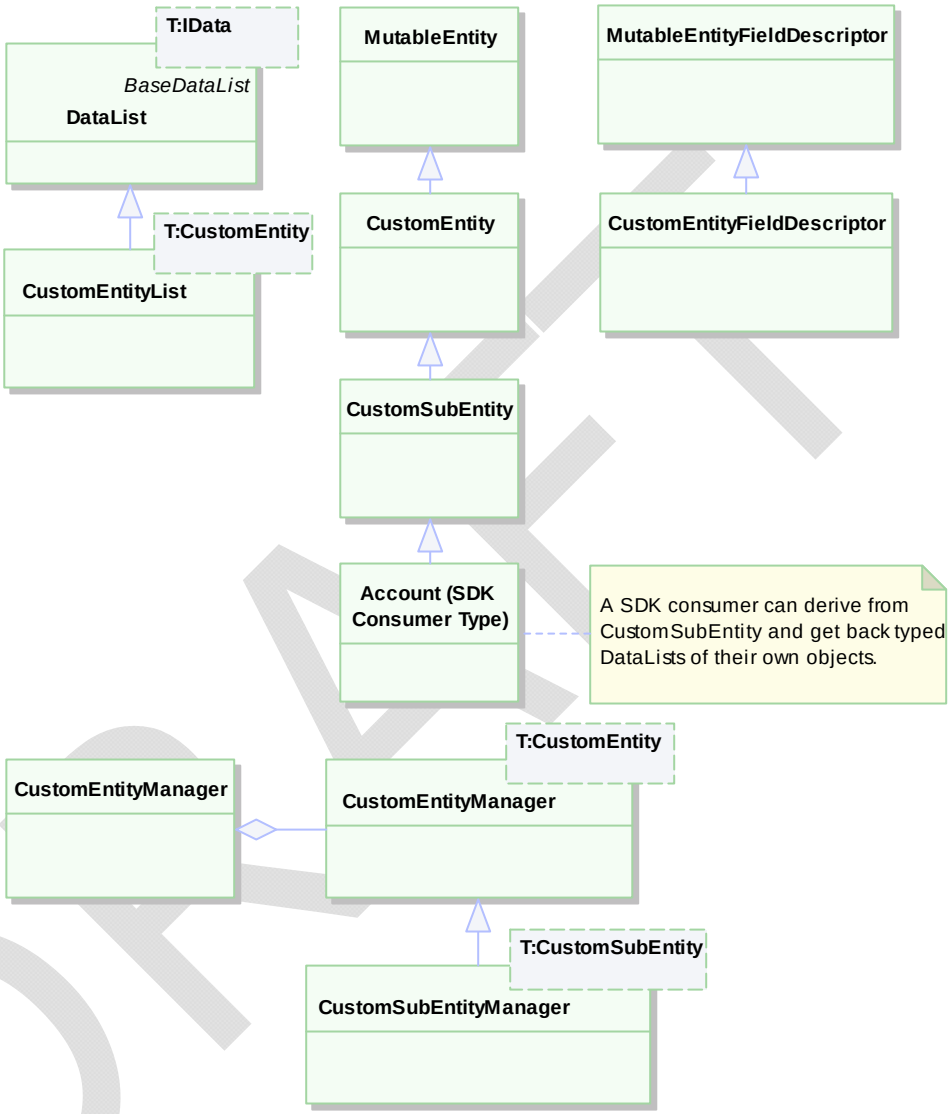
Custom sub-entities provide the SDK consumer the ability to add many to many tables to an ACT! database. Like Notes and Histories which are sub-entities in ACT! you can create your own custom sub-entities that hang off Contacts, Groups or Companies.

The entry point into the custom sub-entities feature is the CustomEntityManager which can be accessed as a property off the ActFramework object using the property CustomEntities. This manager provides the ability to create new custom sub-entities, update custom sub-entities and retrieve information about what custom sub-entities exist in the database.

The CustomEntityManager object also provides the ability to get typed CustomSubEntityManager objects back for a particular custom sub-entity. The custom sub-entities feature was designed to allow the SDK consumer to control what type of object is returned from the database. By deriving from CustomSubEntity you can have your own type returned from the CustomSubEntityManager for custom sub-entity records.

CustomEntityFieldDescriptors and CustomEntityLists can be retrieved from the CustomSubEntityManager. The CustomSubEntityManager is retrieved for a particular custom sub-entity so the fields and lists returned will always be for that custom sub-entity. For example, if you had an Accounts custom sub-entity, the CustomSubEntityManager would only retrieve fields for Accounts and would return lists of Accounts data.

Custom Sub-Entities Object Model



This class diagram shows the relationships between the custom sub-entity types and the base ACT! data types.

This shows that the CustomSubEntity class derives from MutableEntity just like Contacts, Groups and Companies. It also shows that CustomSubEntity is a specialized CustomEntity. In the future ACT! will be able to support custom top level entities or entities that don't require a parent entity.

Creating Custom Sub-Entities

To create a custom sub-entity you need to use the CustomEntityManager via the CustomEntities property on the ActFramework. The ActFramework object will need to be logged on to the database you want to create the custom sub-entity on. You will also need to lock the database to create a custom sub-entity because this changes the database schema.

The CreateCustomSubEntity method creates a new custom sub-entity in the database. To create a custom sub-entity you need to specify a unique name to use for the custom sub-entity. This name has to be unique across all the entities in the database. The recommendation is to do something that would make your entity name unique. You can put your company name as part of the name, your product name, or generate a GUID to use in the name. Whatever you choose you'll want to ensure the name is unique. The name is what you will use to retrieve your custom sub-entity from the database.

You also need to supply a unique display name for the custom sub-entity. This display name is what will be displayed to the user in help about and possibly in the future for define fields, lookups, etc…

The next parameter for creating a custom sub-entity is an enumeration ParentEntity. This is a bitmask of what entities you want your custom sub-entity to hang off of. ParentEntity currently includes Contacts, Groups and Companies. A custom sub-entity can belong to Contacts, Groups, Companies, Contacts & Groups, Groups & Companies or all three.

There is a boolean parameter where you specify if you want the custom sub-entity to be visible in the define fields UI. In the initial SDK release of custom sub-entities the define fields UI will not be present but in future versions of ACT! this could become available. This boolean provides a way to either have or not have your custom sub-entity appear in the UI.

The last parameter is an optional description.

CreateCustomSubEntity returns a CustomEntityDescriptor. The CustomEntityDescriptor object is what is used to represent a particular custom entity in the database. The descriptor is used to specify which custom entity you would like to perform an SDK operation on.

If the CustomEntityDescriptor returned is not a null object and you didn't receive any exceptions you can then create fields in that custom sub-entity.

Adding fields to a custom sub-entity

The API for creating fields in custom sub-entities is the same as creating custom fields in a regular ACT! entity like Contacts. You create a FieldDescriptor object and call to the FieldDescriptorManager to save it to the database. Instead of specifying Contacts, Groups or Companies when you save the field you will need to pass in the CustomEntityDescriptor you got from creating your custom sub-entity.

All of the same features and rules that apply to creating a custom field in Contacts, Groups and Companies apply to creating fields in custom sub-entities. Field level security is supplied as well as the normal define fields options in creating a field. If you've created a field on Contacts it should be the same when creating a field on a custom sub-entity.

Alias name

With the release of custom sub-entities there is a new name property for fields called alias that should be used. Alias is a way of specifying a name for your field that can be used to programmatically get the field descriptor or data from an object for your field. The alias works similar to the physical name of the field in the database. It cannot be changed after it is set and it will always reliably fetch the field since the name is not localized and is not shown to the end user. This solves the problem of creating custom fields in ACT! and then being able to programmatically get access to them. Previously you would have had to store off the physical name since the display name could be changed or localized.

The FieldDescriptor class from define fields now has an overload in the constructor to take the alias. This is the easiest way to set the alias. The alias can also be set on a field after its creation but it can only be set if the alias is null. Alias can can only be set once and behaves like a physical field name.

Alias is available for all define field entities so you can use it on custom Contact, Group and Company fields as well, it is not limited to custom sub-entities.

Retrieving Custom Sub-Entities

After a custom sub-entity has been created you can use methods in the CustomEntityManager to get access to it. CustomEntityManager has methods to retrieve a custom sub-entity by name, retrieve a list of all the custom sub-entities, or retrieve a list by the type of the parent entity.

Retrieving a custom sub-entity by name

The CustomEntityManager provides a method called GetCustomEntityDescriptor which retrieves a CustomEntityDescriptor for the custom sub-entity which matches the name passed in. If the name is not a valid custom sub-entity then a null will be returned from the call. You can use this to check to see if your custom sub-entity has already been created in the database.

Retrieving a list of custom sub-entities

The GetCustomSubEntities method on CustomEntityManager returns an array of CustomEntityDescriptors. This is a list of all the custom sub-entities in the logged in database.

There is also an overload for GetCustomSubEntities that allows you to specify a record type. The record type allows you to retrieve the custom sub-entities that hang off of a specific record type, Contact, Groups or Companies.

Retrieving a CustomSubEntityManager

CustomEntityManager provides a method called GetSubEntityManager<T>. This method is a generic method. It lets the consumer determine the type that will be used when you retrieve data from the CustomSubEntityManager.

The generic parameter T has to be a class that derives from CustomSubEntity. You can use CustomSubEntity as the type if you don't want to get custom typed objects back or you can specify your own type that derives from CustomSubEntity.

Deriving from CustomSubEntity

To derive from CustomSubEntity just declare a new class that represents your custom sub-entity, for example Account and implement the required constructor.

The only requirement of the type by the SDK is that it has a public constructor that takes one argument of the type CustomSubEntityInitializationState. The CustomSubEntity class has a protected constructor that takes CustomSubEntityInitializationState, you just need to have your class call the base class constructor passing in the CustomSubEntityInitializationState object.

C# Example:

```
public sealed class Account : CustomSubEntity
{
    public Account(CustomSubEntityInitializationState state)
        : base(state)
    {
    }
}
// Using the type in a call to get the CustomSubEntityManager.
CustomSubEntityManager<Account> manager =
actFramework.CustomEntities.GetSubEntityManager<Account>(descr
iptor);
```

You also need to pass a CustomEntityDescriptor instance into GetSubEntityManager to specify which custom sub-entity to retrieve the CustomSubEntityManager for. GetSubEntityManager also provides an overload that takes the custom sub-entity name instead of a descriptor. This is provided to save you a call if the only thing you want to do is get the CustomSubEntityManager you don't have to call to get the descriptor and call back to get the CustomSubEntity Manager. It uses the same logic as GetCustomEntityDescriptor to find the custom sub-entity.

The CustomSubEntityManager provides a typed manager for a particular custom sub-entity. It is the functional equivalent of ContactManager in the SDK. ContactManager is the public entry point to Contacts, the typed instance of CustomSubEntityManager is the public entry point for a particular custom sub-entity.

CustomSubEntityManager provides methods to get CustomEntityFieldDescriptors, create custom sub-entity rows, get lists of custom sub-entity data, and more.

A typical usage pattern of CustomSubEntityManager is to hold on to a reference to the manager to use it for the life of your program so you don't have to call back to the CustomEntityManager to keep getting it. CustomEntityManager does cache the sub-entity managers so it doesn't keep creating instances if you call to get the same custom sub-entity's manager.

Editing A Custom Sub-Entity

CustomEntityManager provides methods to update some of the properties of a custom sub-entity.

UpdateCustomEntityDescription allows you to change the description of a custom sub-entity after it has been created.

UpdateCustomEntityDisplayName allows you to change the display name of a custom sub-entity. The display name needs to be unique across the database.

The update methods return an updated CustomEntityDescriptor that has the changes made.

You can also edit a custom sub-entity definition by creating fields, editing fields and deleting fields using the define fields SDK. The way to accomplish those tasks is identical to the way you would do it for Contacts, Groups or Companies. The only difference is you would pass in a reference to your CustomEntityDescriptor for the entity to perform the action on in the define fields API.

Record Identifier

Another feature added for custom sub-entities is the ability to set which fields of a row uniquely identify the row. Since custom sub-entities are dynamic and can have any number and type of fields record identifier provides you a way to indicate to the SDK what fields you are using to uniquely identify a row in a custom entity.

The SDK will use these fields when creating a history for a field change notification. In the define fields SDK you can set a field to create a history every time the field is modified. With custom sub-entities the history will be cut on the parent records that one that custom sub-entity record, i.e. the contacts, groups and companies.

Record Identifier is made up of up to 3 fields usually, any more than that makes it not very useful. The CustomSubEntityManager has a method called SetRecordIdentifier that takes an array of CustomEntityFieldDescriptors. The order of the fields in the array is preserved in the database and the fields show up in that order in the history that is cut for a field change notification.

The record identifier could be used in the future by the ACT! UI to know which fields to show when you need to pick a custom sub-entity. It also could be used to determine what fields should be shown for the ACT! default mutable entities like contacts, groups and companies in a record picker.

It is highly recommended you set the record identifier if you are going to have fields that have field change histories being cut. Even if you aren't it is a good way to dynamically determine what field to show to the user by default for choosing a custom sub-entity since in the future a UI could be provided to the user to change which fields they would like to see.

Deleting Custom Sub-Entities

CustomEntityManager provides a method called DeleteCustomEntity that will remove a custom sub-entity and all of its data from a database. As with all the methods that update the database schema, the database will need to be locked to call this method.

After this method returns the custom sub-entity is gone from the database and references to its CustomEntityDescriptor or CustomSubEntityManager should be set to null since they are now invalid and can be destroyed.

Adding Custom Sub-Entity Rows

Creating a new custom sub-entity object

Custom sub-entities derive from MutableEntity just like Contacts, Groups and Companies. Working with custom sub-entity data is just like working with data in those other mutable entities.

The CustomSubEntityManager provides methods to create new row objects and delete row objects.

The CreateCustomEntity method creates a new custom sub-entity object that represents a row in the database for that custom sub-entity. The return type for the method is a generic type T. This will be set to whatever the type you retrieved the CustomSubEntityManager for.

This allows you to work with custom sub-entities in a strongly typed fashion and provides type safety for your operations. This is especially useful if your application is using more than one custom sub-entity.

After you call CreateCustomEntity you can use the returned reference to set field data and add parent entities to the record.

To persist the object to the database you call the Update method.

To set field data you can either get field descriptors and set the values on the field descriptors to your CustomSubEntity object or use the Fields property indexer of the CustomSubEntity object.

Retrieving field descriptors

The CustomSubEntityManager provides methods to get all the field descriptors for the custom sub-entity, to get field descriptors by name (alias, real or display), and to get the fields by type.

This is the same pattern for getting field descriptors used in the other mutable entities, Contacts, Groups and Companies.

To get all the fields for a custom sub-entity use the GetCustomEntityFieldDescriptors method. This returns an array of CustomEntityFieldDescriptors. This method will only return the fields for the custom sub-entity the CustomSubEntityManager was retrieved for.

The GetCustomEntityFieldDescriptor method allows you to get a field descriptor by name. There is a new enumeration you can use when getting the field descriptor that lets you use alias to get the field. FieldNameType is the enumeration and there is an overload for GetCustomEntityFieldDescriptor that takes it. The enumeration can be used to indicate whether the field name is the real name, display name or the alias for the field.

Setting values for fields

Once you have retrieved a field descriptor you can use the SetValue method to set the data for that field. You pass in the CustomSubEntity object you want to set the value on and the value you want to set.

Another way to set the value of a field is to use the Fields property on the CustomSubEntity object. The fields property returns a fields collection that has an indexer where you use the field name to indicate which field you would like to set the value of.

C# example:

```
account.Fields["ACCOUNT_TYPE",
Act.Framework.MutableEntities.FieldNameType.Alias] =
"Checking";
```

This example uses the alias to se t the value of a field to a string.

The other way to add a custom sub-entity record is to use .net data binding. Data binding will be talked about it a separate section on data binding.

Specifying parent entity row(s)

The only required data for saving a custom sub-entity is a parent entity record to associate the record to. If your custom sub-entity hangs off of Contacts you will need to set at least one contact as the owner of a custom sub-entity row before it will save to the database.

The CustomSubEntity object provides methods to set Contacts, Groups and Companies as parent records for the custom sub-entity row. SetContacts, SetGroups and SetCompanies methods allows you to specify

a ContactList, GroupList and/or CompanyList as the parent records for the custom sub-entity.

There are also methods to clear the parent records for each parent entity type. ClearContacts, ClearGroups and ClearCompanies methods delete all the appropriate parent entities for the custom sub-entity row.

Make sure you have set the Contacts, Groups and/or Companies on the CustomSubEntity object before calling the Update method to save your custom sub-entity row. If not you'll get an exception telling you that you have to have at least one parent row for a custom sub-entity record.

Setting row level security

Custom sub-entities work just like notes or histories, they do not have ACLs (access control lists). They can only be set public or private.

You have to have access to at least one parent entity to be able to see a custom sub-entity row as well. More about how the security model works for retrieval can be found in the retrieving custom sub-entity rows section of this document.

To set a custom sub-entity's row level security use the AccessType property. It can be set to either public or private using the AccessType enumeration.

Deleting Custom Sub-Entity Rows

CustomSubEntityManager provides a method to delete custom sub-entities. The method is called DeleteCustomEntity. The method takes a generic T parameter which will be set to the type of the CustomSubEntityManager.

Custom sub-entities can also be deleted via data binding using the .net data binding interfaces.

Retrieving Custom Sub-Entity Rows

CustomSubEntityManager provides many methods for retrieving custom sub-entity rows. All of the methods return the data as a CustomEntityList. CustomEntityList is a generic class where the type is the same type as the CustomSubEntityManager's template type.

For example, if the CustomSubEntityManager type is CustomSubEntity, then the get methods will return a CustomEntityList<CustomSubEntity> (a CustomEntityList of the type CustomSubEntity).

If you had specified your own type when getting the CustomSubEntityManager then the CustomEntityList will be of that type and return rows as that object type.

GetCustomEntities is the method that will return all the rows of a particular custom sub-entity. The rows returned will be secured by the cascading security model. You will only get rows back where the logged in user has access to at least one of the parent records and rows that are public or rows that are private to that user.

Just like Contacts, Groups and Companies, custom sub-entity lists have a tagged collection and you can tag records for retrieval. CustomSubEntityManager has a method called GetTaggedCustomEntities that returns a CustomEntityList of the records that are in the tagged collection of the passed in list.

GetCustomEntitiesByID can be used to retrieve custom sub-entities by their primary key. This will only return rows that the logged in user has access to.

CustomSubEntityManager provides the GetCustomSubEntities method to retrieve all the custom sub-entity rows for a particular parent record. There are overloads that take Contact, Group and Company.

CustomEntityList

CustomEntityList derives from DataList just like other list classes in the ACT! framework like ContactList. The only difference is that CustomEntityList derives from the DataList<T> template class so it can provide a strongly typed list.

Like other DataList classes, CustomEntityList provides the same functionality just for custom entities. If you have used ContactList, GroupList or CompanyList you should find the CustomEntityList has the same interface.

CustomEntityList has a field descriptor collection for choosing which fields are displayed when the list is data bound to a component that supports ITypedList. The CustomSubEntity property is called FieldDescriptors. Add CustomEntityFieldDescriptor objects to the FieldDescriptors property to have those fields show in a data bound list control.

Once you have retrieved a CustomEntityList you can use the indexer to retrieve a particular CustomSubEntity row.

The type returned from the indexer is the generic type of the CustomEntityList. For example, if you have a CustomEntityList<Account> then the indexer will return an Account object.

Just like other DataList classes, CustomEntityList supports IList, IBindingList and ITypedList interfaces so it can be used with data binding.

Editing A Custom Sub-Entity Row

Similar to adding a new custom sub-entity row you can get a CustomSubEntity from a CustomEntityList and change the column values for the row.

Just like adding rows you can change the field values either by using a CustomEntityFieldDescriptor's SetValue method or by using the indexer on the Fields property of the CustomSubEntity object.

There are also methods on CustomSubEntity for setting the access level of the row and changing the parent records.

None of the changes to the row are saved to the database until you call the Update method.

Data Binding With Custom Sub-Entities

Custom sub-entities use the ACT! DataList class as its base data binding mechanism. The CustomEntityList class supports .net data binding for lists of data as well as individual columns.

With the addition of alias to column names custom sub-entities can support standard .net windows forms data binding use the BindingSource class.

BindingSource is in the System.Windows.Forms namespace and was added in the 2.0 release of .net. BindingSource provides a lot of the data binding functionality that previously would have had to been coded everywhere data binding was used.

From the .net SDK docs:

BindingSource simplifies binding controls on a form to data by providing currency management, change notification, and other services between Windows Forms controls and data sources. This is accomplished by attaching the BindingSource component to your data source using the DataSource property.

Binding custom sub-entities to a list control

After retrieving a CustomEntityList from the CustomSubEntityManager you can bind the list directly to a list control. Set the DataSource property of the control to the CustomEntityList. If you have added fields to the FieldDescriptors property of the list you should see those columns in the control.

DataGridView is a .net 2.0 windows forms control that can be used to data bind to an ACT! DataList.

Another way you can bind the CustomEntityList to the list control is with the BindingSource class. Create a BindingSource object and set the DataSource property of the BindingSource object to the CustomEntityList. Then set the list control's DataSource property to the BindingSource object. The reason for using BindingSource is if the CustomEntityList will also be bound to individual controls for row editing and you want the grid and the controls to be in sync.

Binding custom sub-entities to individual controls

To bind a CustomEntityList row to individual controls use the BindingSource class. Create a new BindingSource and set the DataSource property of the BindingSource to the CustomEntityList.

Next bind the individual control to the field using a new instance of Binding. When creating the Binding class instance use the BindingSource as the source for the binding and use a fields alias as the name of the property to bind to.

C# Example:

```
// bind a check box to the private field.

privateCheckBox.DataBindings.Add("Checked", bindingSource,
"PRIVATE");

// bind a text box to the account name field

policyNameTextBox.DataBindings.Add("Text", this.bindingSource,
"ACCOUNT_NAME", false, DataSourceUpdateMode.Never);
```

This binds the boolean property Checked for a checkbox to the CustomEntityList that is the data source for the bindingSource object and the field with the alias PRIVATE is used.

When binding fields to individual controls, if you want to do validation it's best to use the overload of Add that takes a DataSourceUpdateMode enumeration at the end.

With this overload you can specify DataSourceUpdateMode.Never so those controls can be validated before sending the changes back to the data source.

Once you have validated the value in a control you can manually call and have it post the data back to the data source.

C# Example:

```
// Write the value from the control to it's databound property
descriptor.

control.DataBindings[0].WriteValue();
```

This would usually be done in the form's validated event handler.

Check the .net 2.0 SDK for the Binding class for more information on data binding.

United States
8800 North Gainey Center Drive,
Suite 200
Scottsdale, Arizona 85258
1-866-795-3711
5 users or more: 888-855-5222
Canada: 866-665-2688
www.act.com

France
10 rue Fructidor
75834 PARIS Cedex 17
France
+33 1 41 66 25 25
www.sagecrm.fr

Germany
Berner Strasse 23
60437 Frankfurt
Germany
+49-69-50007.6260
www.sage.de/act

Australia/New Zealand
Level 1, 114 William Street
Melbourne, VIC
Australia 3000
1 300 SAGE CRM
www.sagecrm.com.au

United Kingdom
North Park
Newcastle Upon Tyne
NE13 9AA
0845 245 0276
www.sage.co.uk/act

Spain
Labastida, 10-12
28034 Madrid
España
+34 91 334 92 92
www.sagecrm.es

Belgium
Quai Mativa 23
4020 Liège
+32 4 343 77 46
www.sagebobsoftware.be

Asia
51 Bras Basah Road
#06-04 Plaza By The Park
Singapore 189554
+65 6336 6118
www.act.com