Act! v22 Architecture Reference

Act! Architecture Reference

Released 2019 for Act! Pro v22 and Act! Premium v22

Version: 11_2019

Published by

Swiftpage ACT! LLC

621 17th Street, Suite 500

Denver, CO 80293

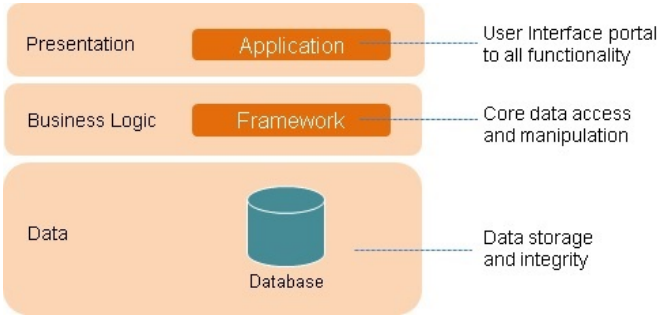
Contents

Introduction

Overview of the Act! Development Platform

The Act! platform consists of feature-rich components that are highly extensible. The platform is built on the .NET Framework and provides a base for rich customization, personalization, and integration. Where Act! seeks to empower users to customize the product to their business, the Act! SDK helps third parties extend that vision through independent development.

The Act! platform has three logical tiers, the Application, the Framework, and the Database, as shown in the following figure.



The Application encompasses all user interface aspects, including screens known as views, navigational components such as menus and toolbars, and design-time components.

The Framework is the engine, providing core functionality including data access, schema metadata and modifications, security, synchronization, database creation and maintenance, data exchange and interoperability, and other essential elements. The Framework includes access to first-class entities such as contacts, groups, companies, and opportunities. It also provides access to extended data including notes, histories, activities, and documents.

The Database is the storage container for primary data. It maintains data integrity and relationships.

About the Architecture Reference

This Architecture Reference provides an overview of entities, relationships, the Framework Object Model, and the Application Object Model. See "Sample Code" on page 12 for sample code provided for using Framework metadata.

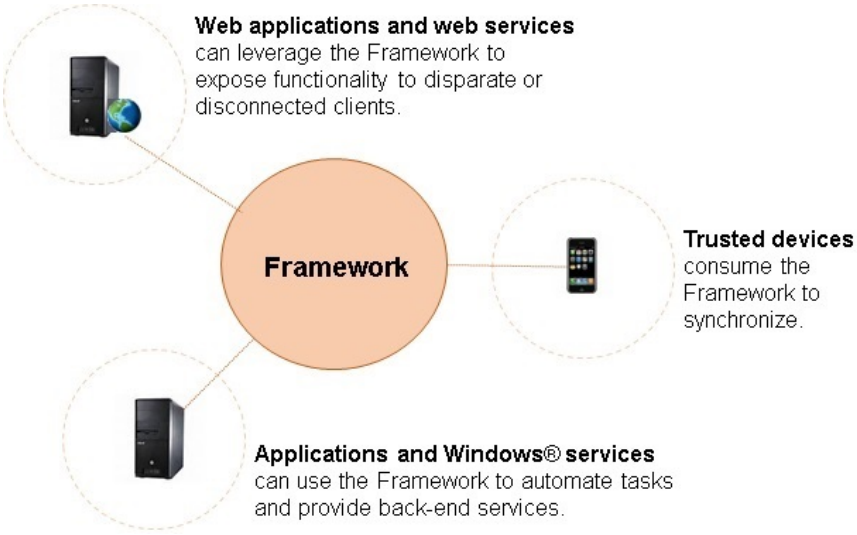
Extensibility Model

Both the Framework and Application tiers have special ways for third parties to access data and customize, integrate with, automate, and extend Act!. The unique needs and interactions of third parties will determine which integration path they should use.

This chapter explains when third parties will need to use the Framework and explains some ways to extend the application using plugins, custom controls, and custom tabs.

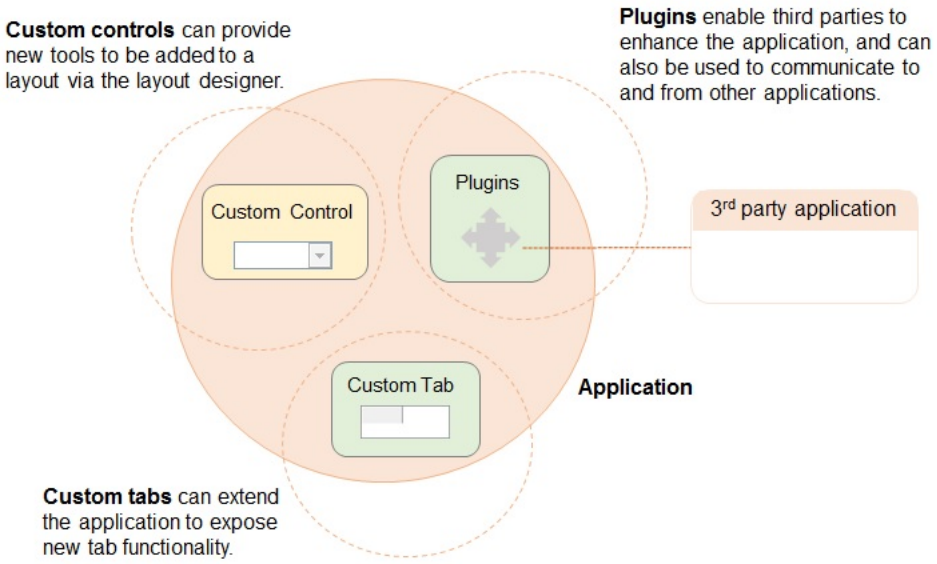
Consuming the Framework

The Framework can be consumed when third parties need to integrate with Act! and when no interaction with the Application or User Interface is needed. Applications and Windows® services can consume the Framework to access data, automate functionality, and provide back-end services. Web applications and services can consume the Framework to provide client applications or back-end solutions across network boundaries. Trusted devices can also synchronize using the Framework.



Extending the Application

The Application has several extensibility points, including plugins, custom controls, and custom tabs. Third parties can use these separately to provide new functionality or together to create more complex solutions.



Plugins

Plugins enable third parties to extend the application behaviorally and/or visually. Plugins can also serve as gateways to other applications or services that need live interaction with the Application. As in other applications, plugins in Act! are given context in the hosting application when they are loaded. Plugins can access all of the Application (and Framework). Typically, plugins will subscribe and react to events in the Application and Framework to perform some specialized functionality.

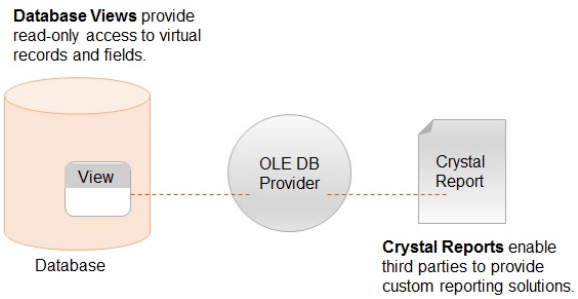
Custom Controls

Third parties can use custom controls to extend the Application's designable views. These include the Contact, Group, and Company detail views. Custom controls also can support rich design-time behavior and integrate with the Layout Designer.

Custom Tabs

Third parties can add custom tabs to provide new ways to view data, for example, in detail views of the application.

Act! includes an OLEDB Provider, which enables read-only access to database views. This is the lowest form of data access, since it circumvents the Framework. It can be used to generate custom reports with tools such as Crystal Reports®. Security is maintained using the OLEDB provider.



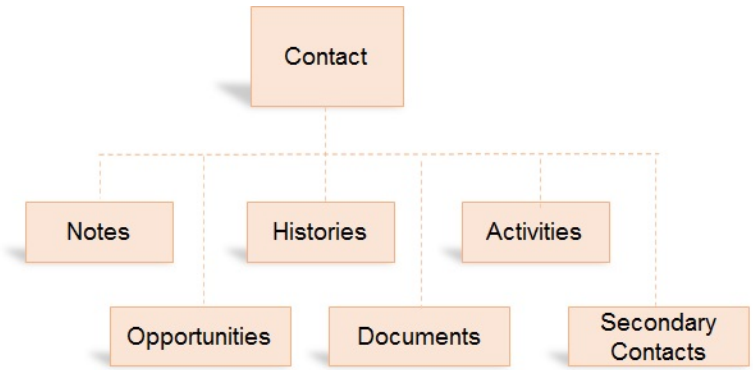
Entities and Relationships

Entities

Act! consists of primary data or entities and extended data or entities. Primary data includes contacts, groups, companies, and opportunities. Extended data includes notes, histories, activities, secondary contacts, and documents.

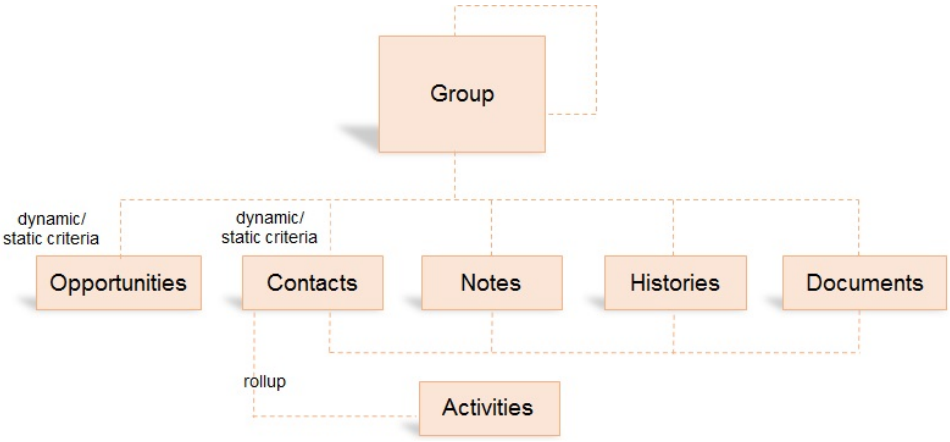
Contacts

Contacts are a first-class entity. Any kind of extended data can be associated with a contact, including notes, histories, activities, opportunities, documents, and secondary contacts.



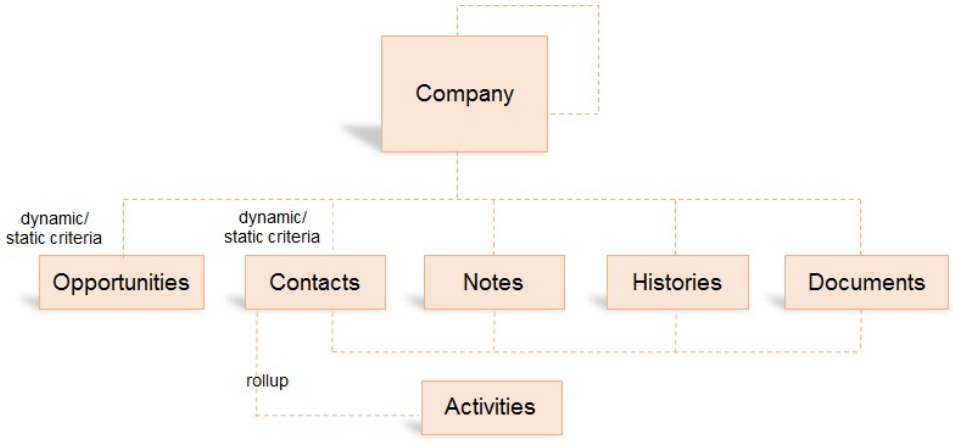
Groups

Groups are dynamic or static sets of contacts and opportunities. Groups can have their own notes, histories, opportunities, and documents. Activities can be rolled up, so that users can access any of the items associated with contacts in a group. Groups can also contain subgroups.



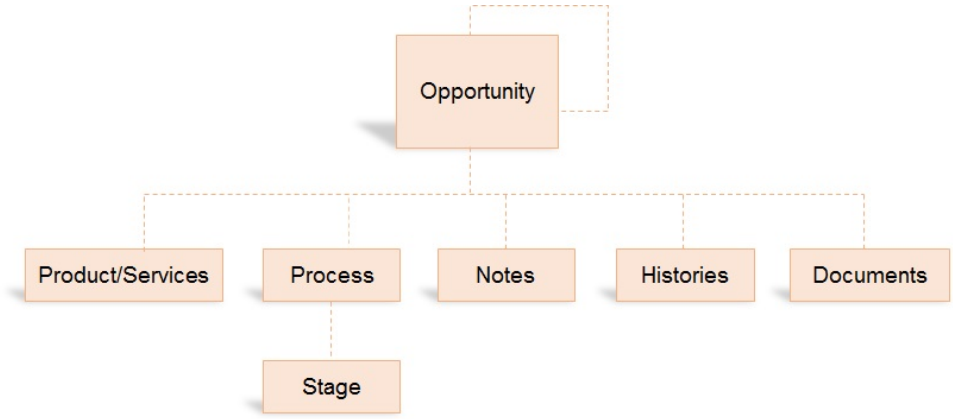
Companies

Companies are similar to groups. They can have their own notes, histories, opportunities, and documents, and they can roll up activities. Companies can also have sub-companies, known as divisions.



Opportunities

Opportunities are a first class entity and can be associated with any number (including zero) of groups, companies, and contacts. An opportunity must be associated with one process and one stage within the process. Opportunities can have many products/services.



The Framework Object Model

This chapter gives an overview of the Framework object model.

ActFramework Class

ActFramework, found in Act.Framework.dll, is the root Framework class. It is the entry point to all Act! core functionality.

Getting Started

To use ActFramework, you must be authenticated as an Act! user and log in as described in the following:

```
ActFramework framework = new ActFramework();
framework.LogOn("CHuffman", "password", "localhost", "MyDatabase");
```

Managers

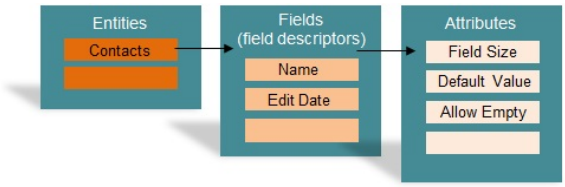
ActFramework exposes Managers via properties. Managers are gateways to feature- or entity-related functionality. For example, a ContactManager is accessible for the Contacts property, which is responsible for contact-related operations.

Metadata

Much of the structure that defines extended data classes, such as Note, History, and Activity, is well defined and unalterable. The data surrounding these is available via members of the data's respective classes. However, you can change the topology of primary entities, such as Contact, Group, Company, and Opportunities. You do this via Define Fields (using the application) or via the DatabaseManager (using the Framework). These entities are largely metadata driven. Determining their landscape and data is a process of discovery.

The Framework exposes such an entity's fields via field descriptors. Specifically, entities have specialized types deriving from DBFieldDescriptor. Most of the functionality is based on a .NET native type, the PropertyDescriptor. A PropertyDescriptor is a virtual representation of a property. PropertyDescriptor objects can depict the topology of some classes, which may or may not differ from their real properties. Typically, PropertyDescriptor objects are retrieved via reflection, to dynamically discover the properties of some class. This virtualization can also be used when databinding a list to a grid (by implementing ITypedList). PropertyDescriptor objects also contain an AttributeCollection, which represents any attributes or declarative metadata of the Property.

The Framework extends the capability of a PropertyDescriptor to represent the virtualization of a record's field; more specifically, to dynamically discover an entity’s fields. Likewise, the Framework leverages Attribute objects on the PropertyDescriptor (exposed via the AttributeCollection on the Attributes property) to provide attributes of that field (such as size, mask, or default value). These attributes typically depend on the ACTType of the field (e.g., mask is not applicable to picture fields).



A PropertyDescriptor defines a property or field in many ways. It defines the type of the property via PropertyType and whether it is read-only via IsReadOnly. It assigns the field name via DisplayName. DBFieldDescriptor extends this to provide ACTType, which represents the Act! type (such as uppercase or phone number). Values can be retrieved from and updated to entities using the GetValue and SetValue methods, respectively.

Unlike typical retrieval of PropertyDescriptor objects via reflection, the Framework enables retrieving DBFieldDescriptor objects via an entity's Manager. For consistency and databinding, the ActFramework entity Managers expose all fields as if they were metadata-driven. Thus, all entity managers implement an interface ISupportMetaData. This interface allows retrieval of metadata for an entity and filtering of the type of metadata by Type or ACTType.

See "Using Framework Metadata" on page 12 for a sample of how field descriptors can be used.

You are not limited to field descriptors to get or set data in entities. You can also fetch and update field data using the FieldCollection indexer on mutable entities. However, using field descriptors is the preferred method of retrieving and setting data. Field descriptors cache needed metadata, so reusing a field descriptor to access or change data on multiple entities (e.g., looping over entities and getting a field value) performs far better than using the field collection on the entity, which has to look up the metadata each time.

Entity Lists

All entities can be retrieved via lists (collections). Like metadata, lists are retrieved through their respective entity Manager. Each entity Manager may have specialized parameters, such as filter criteria, that are used to retrieve a list. However, all must at least support retrieving a list passing SortCriteria. SortCriteria consists of the PropertyDescriptor (see "Metadata" in this chapter), specifying the field, and the ListSortDirection, specifying the direction on which to sort (ascending or descending). Some lists, such as mutable entity lists, support sorting on multiple fields.

Entity lists can be bound to any .NET-aware grid controls, which will use the lists ITypedList implementation to get the PropertyDescriptor objects (see "Metadata" in this chapter). The PropertyDescriptor objects are used to name the columns and get data for each row. Grid controls also will use the entity list's IBindingList implementation to sort, search, and react to list notifications, such as when a new item is added.

See "Getting a Contact List" on page 13 for a sample.

All entity lists, except for ActivityList, fetch data on demand and cache the data as a way to scale to large quantities. As a side effect, data may become stale. To refresh data, invoke the Refresh method. Also, be mindful that iterating over or accessing items in the list may cause data to be fetched from the database.

Working with Data

You can create and delete entities from their Managers. You can create and delete primary entities, including contacts, groups, opportunities, and companies, via their lists, using IBindingList.AddNew(), and IList.Remove() methods. You can retrieve extended data for a primary entity via the Manager of the extended data. For example, the NoteManager has GetNotes overloads to pass in a contact or a company.

Within a list, you can find items using the Find() method, which takes a field descriptor (see "Metadata" in this chapter) and a value. You can use a lookup to get a list of primary entities that matches a particular criteria. You perform lookups via the LookupManager. Lookups are essentially list criteria; each criteria is made up of a logical operator (AND/OR), a field, an operator valid for that field, and a value.

The Application Object Model

This chapter provides an overview of the Application object model.

The ActApplication Class

ActApplication, found in Act.UI.dll, is the parent application class and is the entry point to all User Interface functionality. The Application typically is not created and accessible directly, unlike the Framework, but third parties can acquire its context via extensibility points, such as plugins, custom controls, and custom tabs.

UI Managers

ActApplication exposes UI Managers via properties. Like ActFramework's Managers, the ActApplication's Managers are gateways to feature- or entity-related functionality.

Plugins

Plugins, as previously mentioned, are given context by the Application (and Framework via the Application). The Application serves as a loader and host for plugins, which can then react to events, customize or extend the application, and communicate with other applications.

To become an Act! plugin, a type must:

- Implement the IPlugin interface (defined in Act.UI.dll).
- Reside in an assembly in the Plugins directory under the Act! application directory.

The IPlugin interface is a simple interface. It provides an entry point for the application to hand itself to the plugin and a way to notify the plugin when the application is unloaded. After the plugin is loaded, and the ActApplication is provided via the OnLoad method, the plugin is responsible for reacting to events appropriately. For example, the plugin would have to react in a manner appropriate to the context of the ActFramework's AfterLogon and BeforeLogoff events. The plugin would also have to react in a manner appropriate to the context of the BeforeDatabaseLock and AfterDatabaseLock events.

Views

Views are the main User Interface panes in the application. Views, other than calendaring, typically provide detail or lists. Views can be enumerated via the ViewManager. The ActApplication notifies listeners, via the CurrentViewChanged event, when a different view is shown. You can retrieve the current view via the CurrentView property of the ActApplication. Views can be changed and shown via entity UI managers. For example, UIContactManager.ShowDetailView() changes the view.

Application State

The Application contains information about the state of current entities and lists (such as the current contact and current contact list). You can access this information via the ApplicationState property in ActApplication. Events on the application also exist in the form XXXChanging and XXXChanged, which notify consumers of entity and entity list changes.

Menus and Toolbars

Act.UI.Core.dll contains all types and functionality related to menus and toolbars. The Explorer property on the ActApplication object is the main entry point for accessing, adding, and removing toolbars. Plugins can use this to add a new menu item and to perform an operation when the item is selected.

Custom Controls

Custom controls can provide new ways to visualize data and interact with the application in designable views. You can make custom controls available as part of a toolbox tool in the Layout Designer and add them to layouts to enhance the contacts, groups, and companies detail views.

To become a custom control, a type must:

- Implement IComponent (for example, by deriving from Component or Control).
- Be marked with the CustomControlAttribute attribute (defined in Act.Shared.ComponenModel.dll).
- Reside in an assembly in the Tools directory under the Act! application directory.

This enables end users to access the control by right-clicking on the Layout Designer toolbox and selecting the Customize menu. If selected, the custom control is available in the “Custom” category of the toolbox, for use in the Layout Designer.

Once the custom control is placed on a layout, and the layout is saved, that custom control must be installed on the client machine for the end user to use that layout.

Using .NET Design-Time Attributes and Types

The Layout Designer supports standard .NET design-time related attributes and types; most of this exceeds the scope of this document. However, minimal design-time related functionality is documented here to explain basic design-time interaction with the Layout Designer:

- **CategoryAttribute**- Mark a property with this attribute to control the category that will display in the Properties window in the Layout Designer.
- **DescriptionAttribute** - Mark a property with this attribute to control the description of the property that will display in the Properties window in the Layout Designer.

Using Design-Time Instruments

The Layout Designer supports design-time instruments such as TypeConverter and UITypeEditor. You can use these objects to control the behavior of the properties in the Properties window in the Layout Designer.

Using .NET Serialization Techniques

The Layout Designer also leverages standard .NET techniques for serialization. Custom controls can leverage these to control which properties get serialized in layouts and how this is done:

- **DefaultValueAttribute** - Mark a property with this attribute to skip serialization of a property whose value has not changed from the default.
- **DesignerSerializationVisibility** - Use this attribute to skip serialization of a property or to serialize the contents of a property (such as a collection).
- **ShouldSerializeXXX method** - Use this method, where XXX is the name of a property, to enable a control to programmatically manage whether or not a property should be serialized.

Using Act! Design-Time Attributes, Interfaces, and Types

You can use Act!-related design-time attributes, interfaces, and types in the creation of custom controls:

- **LayoutToolboxFriendlyNameAttribute** - Mark a type with this attribute to enable controls to have a friendly name in the designer (other than its type). Provide a public static string LayoutFriendlyName property to return the name.
- **TabableAttribute** - Mark a type with this attribute to enable controls to participate in Tab and Enter Stop functionality in the Layout Designer.
- **ToolboxBitmapAttribute** - Mark a type with this attribute to enable controls to specify a custom icon that will display in the toolbox next to the control names.
- **EmptyTypeConverter** - Mark a type with a TypeConverterAttribute to enable controls to completely hide their properties in the Properties window in the Layout Designer.
- **LayoutControlDesigner** - Mark a type with a DesignerAttribute to enable the right-click ”Edit Properties” menu in the Layout Designer.
- **ICustomClipboardSupport** - Implement this interface for controls that interact with Act!’s clipboard functionality (cut/copy/paste/delete/undo).
- **SpellCheckableSupportAttribute and ISupportSpellCheck** - Use these controls to support spell checking.
- **LayoutSingletonComponentAttribute** - Mark a type with this attribute to specify that a control can exist only once on a layout.

Binding Custom Controls to a Field

You can bind custom controls to a field, similar to binding a Field or Memo control in the Layout Designer. You must do the following to implement a bound custom control:

- Implement IXXXListBoundControl, where XXX is Contact, Group, Opportunity, or Company. This forces the control to provide a list component property, which is how it will get the context of the current entity. Once the list is set, you can attach the control to the PostionChanged and ItemChanged events on the CurrencyManager (standard .NET databinding) to be notified when the current entity changes. See SDK samples.
- Implement IXXXFieldBoundControl, where XXX is Contact, Group, Opportunity, or Company. This forces the control to provide a field descriptor property ("Metadata" on page 6), so the control can get and set values in the entity when it is updated. See SDK samples.
- Implement IUpdateableComponent. This tells the control when to update changes on the underlying entity. See SDK samples.

Custom Tab

You can enrich the Application by creating a custom tab, which may contain its own controls and interface. You must use a plugin to create a custom tab. The plugin adds a new tab to a layout after the layout loads. Typically, a plugin will attach to the LayoutChanged event on the ApplicationState object and add a tab using the UILayoutDesignerManager.AddTabToCurrentLayout() method. This method takes a .NET TabPage as a parameter. The plugin must create the tab and add controls to its collection. See SDK samples.

Sample Code

This chapter contains two samples of code: the first shows the use of Framework metadata; the second shows getting a contact list.

Using Framework Metadata

The following is an example of using Framework metadata. In this example: print the field display name of any contact string fields that can be edited, do not allow empty values, and do not have any default value (thus, string fields are 'blank' when we create a new contact):

```
// filter to just return string fields on a Contact
ContactFieldDescriptor[] fields =
framework.Contacts.GetContactFieldDescriptors(new Type[]{typeof(string)});

// we're going to look for editable fields that don't allow empty values
// and don't have default values, so we'll need these attribute types
Type allowsEmptyType = typeof(AllowEmptyFieldAttribute);
Type defaultValueType = typeof(DefaultFieldValueAttribute);

// initialize our attributes
AllowEmptyFieldAttribute allowsEmptyFieldAttr = null;
DefaultFieldValueAttribute defaultFieldAttr = null;

AttributeCollection attributes = null;
ContactFieldDescriptor contactField = null;
for (int i=0;i<fields.Length; i++)
{
    contactField = fields[i];

    // make sure we can modify this field
    if ( !contactField.IsReadOnly )
    {
    attributes = contactField.Attributes;

    // check if we don't all empty values
    allowsEmptyFieldAttr = attributes[allowsEmptyType]as
    AllowEmptyFieldAttribute;
    if ( allowsEmptyFieldAttr != null &&
    !allowsEmptyFieldAttr.AllowEmpty )
    {
    // now check to see we don't have a default value
    defaultFieldAttr = attributes[defaultValueType] as
    DefaultFieldValueAttribute;
    if ( defaultFieldAttr == null || defaultFieldAttr.DefaultValue ==
    null )
    {
    // we found one
    Console.WriteLine(contactField.DisplayName);
```

```
}
}
}
}
```

Getting a Contact List

The following sample shows how to get a contact list.

```
// get the company field descriptor
DBFieldDescriptor companyField =
framework.Contacts.GetFieldDescriptor("TBL_CONTACT.COMPANYNAME",
true);

// get contacts I have access to, sorted by company
ContactList contacts = framework.Contacts.GetContacts(
new SortCriteria[]{new SortCriteria(companyField,
ListSortDirection.Ascending)});
```

Index

M

O

P

S

T

U

V