AGILELIGHTFORMS DEVELOPER GUIDE

DOC ID: 12111-FMG-1.0

Table of Contents

AgileLightForms Developer Guide

1. Introduction

AgileLightForms (ALF) Software Development Kit (SDK) allows you to customize forms beyond the capabilities of Forms Designer.

There are two main ways for such customization:

- **Adding a Form Handler**: Form Handlers allow you to add code to handle events such as the user changing a value. This allows you to implement advanced validation, conditional formatting, and many other features
- **Editing the Form User Interface**: You can export form design to use XAML editors like Visual Studio or Expression Blend. You can thus change colors, control properties, or substitute controls, possibly using third-party controls.

1.1 Disclaimer of warranty

AgilePoint Inc. makes no representations or warranties, either express or implied, by or with respect to anything in this document, and shall not be liable for any implied warranties of merchantability or fitness for a particular purpose or for any indirect, special or consequential damages.

2. Adding a Form Handler

A Form Handler is a Silverlight class that handles some events that happen during form lifecycle.

2.1 Creating a Form Handler

To create a Form Handler, follow these steps:

1. Go to ALF physical directory (usually C:\ALF)
2. Enter the ClientBin folder
3. Copy the ApFormsClient.xap file to a working folder
4. Rename ApFormsClient.xap to ApFormsClient.zip
5. Open ApFormsClient.zip
6. Extract ApFormsClientSDK.dll to a working folder
7. Close ApFormsClient.zip
8. Delete ApFormsClient.zip
9. In Visual Studio, create a new Silverlight 4 Class Library project
10. In this project, add a reference to ApFormsClientSDK.dll
11. Create a class that implements Ascentn.Forms.Client.Sdk.IAgileLightHandler

This class will be your Form Handler.

2.2 Form Handler SDK

The next sections describe the methods you provide and ALF will call when certain events happen (Form Handler Events), the methods you can call for generic, non-form related features (Form Handler API), and the methods you can call to access form data and controls (Form Access)

2.2.1 Form Handler Events

The methods of your Form Handler will be called when some events happen:

- **Start:** Will be called when your handler is loaded, before the form is displayed

  You should store the two references you are passed:
  - **`api`**: can be used to access non-form related features, such as tracing. See Using Form Handler Api below
  - **`form`**: is the reference to your form, and can be used to access both the user interface and the data. See Accessing Form below

```
public void Start(IAgileLightApi api, IAgileLightForm form)
{
      // Remember parameters
      this.api = api;
      this.form = form;

      // TODO: Add your initialization code here
}
```

- **DataLoaded:** Will be called whenever part of the form is loaded. Forms are currently loaded tab by tab. So, in a three-tab form, this method will be called four times: one for the form, and one more for every tab. Take into account that the whole form data and controls will be available after the last call only.

```
public void DataLoaded()
{
      // TODO: Adapt the just-loaded user interface
}
```

Note: Due to this tab-by-tab loading feature, you should always check that the attributes and/or controls have already been loaded before accessing them. Their tab might not have been loaded yet.
This is the place to customize the form beyond the capabilities of form designer, as explained below.

- **IsDirtyChanged:** This method will be called whenever the Dirty flag changes. This flag indicates whether there is unsaved data or not. So, it will be called after the end-user changes the value of an attribute in the form (making the form dirty).

```
public void IsDirtyChanged()
{
      // TODO: Adapt to new dirty flag value
}
```

Take into account that if the user changes that attribute back to its original value, the dirty flag will be cleared and this event will be called again.

- **Submitting:** This method is called when the user clicks the Submit button and the form is successfully validated, but before the form is actually submitted. If this method returns true, submission is canceled.

```
public bool Submitting()
{
      bool result = false;

      // TODO: Perform advanced validation
      return result;
}
```

This is so the right place to perform advanced validation, as seen below.

- **Submitted:** This method is called after the form is submitted. Depending on how the form was opened, three possible actions can follow:
    - The window is closed
    - The window navigates to a different URL
    - The form is refreshed

```
public void Submitted()
{
      // TODO: Perform advanced validation
}
```

No matter which of the three actions is taken, the form (and the handler) will be disposed of. So, this is the place to perform any clean-up work that might be necessary.

- **Asynchronous Submitted:** Sometimes, the actions that have to be taken when the form is submitted take too long to complete, because they have to wait for a Web service to answer, or the user to close a window. In this case, you can implement the `IAgileLightHandlerAsyncSubmitted` interface in your form handler. This interface contains a single method, the asynchronous version of `Submitted`.

```
public void Submitted(Action onSubmittedEnded)
{
      // TODO: Remember calling onSubmittedEnded();
}
```

If your handler contains the asynchronous version of submitted, you must call `onSubmittedEnded` when you want the form to be closed, redirected or refreshed, depending of form parameters.

Note: User interface should be disabled until you call `onSubmittedEnded`. You can do that either opening a child window or disabling the form. You can disable the form with `form.IsDisabled = true;`

- **ValueChanged:** This method is called whenever an attribute changes its value. This is usually caused by end-user actions on controls, but also by handler code changing attribute values. Most controls change the attribute value when they lose focus. Textboxes, on the other hand, change the attribute value whenever the text changes, raising the event once for every character the user types.

```
public void ValueChanged(IAgileLightAttribute attribute)
{
      // TODO: Perform conditional formatting
}
```

This is the right place to perform conditional formatting and online validation, as depicted below.

2.2.2 Form Handler API

These are the properties and method that can be accessed through the reference passed to `Start` event:

- **ApFormsServerUri:** Gets the URI of ALF Server

```
Uri ApFormsServerUri { get; }
```

- **MapApFormsServerUri:** Returns an URI that can be used to access a web resource using ALF server as a proxy, i.e., the result URI will navigate to ALF server which, in turn, will navigate to the provided URI and return the result.

```
Uri MapApFormsServerUri(string relativeUri, NetworkCredential credential);
string MapApFormsServerUri(string relativeUri);
```

By using this method, end-user computers require access to ALF site only.
If you use the overload that accepts credentials, ALF will use that credentials to access the target site. Keep in mind that this method requires the credentials to be sent to client computers (in handler) and then back to ALF. This could pose a security concern.
If, on the other hand, you use the overload without credentials, ALF will use the credentials of its Application Pool identity account. This is the preferred method, for security reasons.

- **GetApFormsServerImage:** Downloads an image given its URL. All considerations of previous method apply to this one too.

```
ImageSource GetApFormsServerImage(string relativeUri);
ImageSource GetApFormsServerImage(string relativeUri, NetworkCredential credential);
```

- **ShowError:** Displays an error window. Very useful when performing advanced validation. You can provide the error message you want to display or an exception.

```
void ShowError(Dispatcher dispatcher, Exception ex);
void ShowError(Dispatcher dispatcher, string text);
```

- **GetErrorMessage:** Gets the error message associated with an exception. GetErrorMessage gets rid of wrapper exceptions, such as `TargetInvocationException`, and always gets a meaningful message. In any case, take into account that a meaningful doesn't have to have a meaning for end users.

```
string GetErrorMessage(Exception ex);
```

- **WriteLine:** Writes a message to trace pane (see Monitoring the execution of your Handler below)

```
void WriteLine(object trace);
void WriteLine(string trace);
void WriteLine(string format, params object[] args);
```

- **MainForm:** Gets a reference to the main form. Take into account that a form can open child forms.

```
IAgileLightForm MainForm { get; }
```

- **SetControlAndDescendantsPropertyValue:** Sets the value of a property in a control and all of its descendants. Useful, for example, to set the Background of a user control made up from many other controls.

```
void SetControlAndDescendantsPropertyValue(
      UIElement control,
      string propertyName,
      object value);
```

2.2.3 Form Access

Form Data Access is divided in three sections: getting references to forms, accessing the user interface (controls) in that forms, and accessing data (attributes) in forms.

Accessing Forms

To access form data, you first have to get a reference to a form. You usually use the reference to the handler form passed to the `Start` event. You can also use the `MainForm` property of the API to get a reference to the main form, or use the `ChildForms` enumeration of a form to access its child forms.

Form User Interface

AgileLightForms are usually composed of tabs, which are composed of sections. Each section is a grid containing controls and labels. Each control is usually bound to an attribute. Each label-control pair is called a cell.

All this default behavior can be changed by either changing the form XAML or programmatically customizing the form.

To access a Tab from its container Form, use the `Tabs` enumeration or the `GetTab` method if you know its name. Use the property `ParentForm` to access the form that contains a Tab.
To access a Section from its container Tab, use the `Sections` enumeration or the `GetSection` method if you know its name. Use the property `ParentTab` to access the tab that contains a Section.

To access a Cell from its container Section, use the `Cells` enumeration, or the `GetCell` method. Use the property `ParentSection` to access the section that contains a Cell.
Once you have a reference to a cell, you can use it to access the label control, the value control, or the attribute itself.

All ALF controls implement interfaces that make it easier to access them:

- **IAgileLightAttachmentListControl:** Use this interface to change the configuration of your Attachment List controls. You can even access attachments, as described in a section below.
- **IAgileLightCheckControl:** Use this interface to change the configuration of your Check controls, used with Boolean attributes.
- **IAgileLightChildrenViewControl:** Use this interface to change the configuration of your Children View controls. You can even access displayed information, as described in a section below.
- **IAgileLightDateTimePickerControl:** Use this interface to change the configuration of your Date & Time controls. You can even access selected date and time.
- **IAgileLightLabelControl:** Use this interface to access cell labels. Every label can have an image, usually a requirement indicator.

- **IAgileLightLabeledTextBoxControl:** Use this interface to access your Labeled Text Box controls. These controls are used to display a textbox with a prefix and / or a suffix. ALF uses these controls to display currency attributes, prefixing the quantity with the currency sign.
- **IAgileLightLookupControl:** Use this interface to change the configuration of your Lookup controls. These controls can edit both Lookup and Party List attributes, as described in a section below.
- **IAgileLightOptionSelectorControl:** Use this interface to change the configuration of your Drop-down Selector controls. You can even access selected option (or just value).
- **IAgileLightRadioGroupControl:** Use this interface to change the configuration of your Radio Group controls, used with Boolean and Picklist attributes.

Working with Attachments

You can access the list of attachments in an Attachment List control using the property `Attachments` in `IAgileLightAttachmentListControl`.

This property returns an array of Attachments. You can access these attachments using the `IAgileLightAttachment` interface. This interface allows you to get the `Name`, `Title`, `Description`, `MimeType`, `Size`, and `Content` of every attachment.

You shouldn't change the value of any of these properties in returned attachments. To make any change, you must create a new array of attachments and assign it to the `Attachments` property of the control.

Only attachments in the server will have a value in the properties `MimeType` and `Size`, but their `Content` property will be empty.

To create attachment objects, use the method `CreateAttachment` in the `IAgileLightAttachmentListControl`.

To add, edit, or delete attachments, create a new array of attachments, each one created with the previous method.

- To add a new attachment, create a new Attachment object and set its `Name`, Title, `Description` and `Content` properties
- To delete an existing attachment, don't add it to the new array
- To keep an existing attachment, add it to the new array
- To update an existing attachment, change its properties and add it to the new array. Set the `Content` property only if you want to change it. Keeping it null will keep server content.

Once you've finished creating the new array, assign it to the `Attachments` property of the control.

Working with Child Entities

Unlike Attachment List control, that gives you full control to its contents, Children View control gives you read-only access only to the data it displays. This is due to the complexity that child forms can have. A form to create a child entity is not restricted to just that (creating the child entity). In fact, it can create and update many entities at the same time, even in different data sources. A child form can also open many more subforms, through Lookup and Children View controls.

Apart from members that let you access and / or change control configuration, there are three members that give you the power to access child entities themselves:

- **Refresh:** This parameterless method allows you to ask the control to refresh its contents. This could be useful to make it display changes done on the data source (CRM) by calls to Web Services from your handler.

- **Refreshed:** This event allows you to know when the control has been refreshed. This is useful to allow you to read its contents whenever they are retrieved from CRM.
  For functionalities like limiting the number of child entities that a parent entity can have, subscribe to this event and disable Adding new entities if the maximum number has been reached.
  It might be necessary to disable Searching and Paging to be able to know the exact number of child entities.

- **ChildEntities:** This read-only attribute gives you access to control contents. For every child entity, you get a class with two properties you can access using the `IAgileLightChildEntity` interface:
  - **Status:** Indicates the status of the child entity:
    - **Unchanged:** The child entity is exactly as it was retrieved from data source
    - **Updated:** The child entity has been updated using a child form. It will be updated in data source when the main form is submitted to server
    - **New:** The child entity has been created using a child form. It will be created in data source when the main form is submitted to server

      Note: ALF creates a fake entity Id for new child entities, because real Id won't be accessible until entity is created in CRM. If you plan to use this Id, read the section "Working with new entities" below.
    - **Deleted:** The child entity has been deleted. It will be removed from data source when the main form is submitted to server
  - **Attributes:** This dictionary contains the attributes of every child entity, indexed by internal attribute name. Only the primary key attribute and the attributes in the selected view will be available.
    If the view contains attributes in related entities, they won't be available in new entities until they are created. Also, the values of these attributes won't change in updated entities until they are updated in CRM.

Note: ALF creates a fake entity Id for new child entities, because real Id won't be accessible until entity is created in CRM. If you plan to use this Id, read the section "Working with new entities" below.

Working with Lookups and Party Lists

Lookup control is used to edit both lookup and party list attributes. Property `Value` is used to get / set lookup values, whereas property `ListValue` is used to get / set party list values.

Some CRM party list attributes can have at most one lookup value (this is the case of the From attribute in some activities). In this case, you still must use the `ListValue` property, but the property `MultiSelect` will be set to false to avoid the control letting the user select more than one value.You can use the `CreateLookup` method of Lookup control to create lookups to assign them to `Value` or `ListValue`.

Note: ALF creates a fake entity Id for new child entities, because real Id won't be accessible until entity is created in CRM. If you plan to use this Id, read the section "Working with new entities" below.

Working with New Entities

There are three ways of creating new entities in ALF forms:

- New entities can be created using form schema. This is the usual way
- New parent entities can be created by configuring an appropriate child form in a Lookup control. Upon submit, ALF must create these new parent entities BEFORE the entity that references them, so the real Id can be used for the Lookup attribute
- New child entities can be created by configuring an appropriate child form in a Children View control. Upon submit, ALF must create these new child entities AFTER their parent entity, so the real Id can be used for the foreign key attribute

Also, there's no limit on how many child forms can be opened, or how many levels of subforms can be used. This creates a tree-like hierarchy of forms, each one possibly updating and creating many entities. Extreme care must be taken to ensure that every new entity gets created before its Id is used in any lookup or party list attribute of any other entity.

ALF takes care of creating the entities in the right order, and changing the fake Id to the real one at the right time.

As for your handler code, you are not allowed to copy the fake Id of any child entities created in a Children View control.

On the other hand, you can copy the fake Id of any parent entities created in a Lookup control and use it in any of the attributes of any of the entities of the form that contains the Lookup control, or any of its parent forms (the forms that were open when the form with the Lookup control was opened).

Any attempt to use a fake Id in a lookup or party list attribute other than those will fail because the fake Id won't be changed to the real one.

Form Data

Form Data can be accessed by using the property Data of a Form.

You can access entities by using the `Entities` enumeration or the `GetEntity` method of form data. You can access the attributes of an entity by using the `Attributes` enumeration or the `GetAttribute` method.

You can access the user interface cells (there could be more than one) associated with an attribute by using the `Cells` enumeration.

Lastly, you can access the attribute value through the property `Value` of the attribute.

For values in selector controls (either drop-down or radio groups), you can access the selected data option through the attribute `OptionValue`. A `DataOption` contains a `Label`, an `Image`, and a `Value`.

For lookup values, you can access the lookup information through the attribute `LookupValue`. A `Lookup` contains a `Label`, an `Image`, an `EntityType`, and an `EntityId`.

For any other attribute type, cast the `Value` to the appropriate type.

You can also access the Attribute a cell is bound to by using the property `Attribute` in the cell.

Then, you could use the attribute `Parent` in an attribute to access its container Entity.

2.3 Typical Handler Features

This is a short list of some of the most common uses for Form Handlers.

2.3.1 Programmatically Customizing a Form

This is usually performed in `DataLoaded` event. You usually look for an entity in form data, then look for an attribute in that entity, and then for controls on the form for that attribute… once you grab a reference to an attribute control, you can change the properties of that control, remove it, change it to a different control…

```
public void DataLoaded()
{
    // Find the textbox associated with the attribute "Title" in entity "Existing Case" and
    // change its font size

    // Use the name of the entity in form schema
    IAgileLightEntity existingCase = form.Data.GetEntity("Existing Case");

    // We have to check if the case has been loaded, because it might not have been
    // loaded yet
    if (existingCase != null)
    {
        // Use the internal name to search for an attribute
        IAgileLightAttribute caseTitle = existingCase.GetAttribute("title");

        // We have to check if the case title has been loaded, because it might not
        // have been loaded yet
        if (caseTitle != null)
        {
            // There might not be cells for this attribute, or be more than one, but
            // we assume there is only one
            IAgileLightCell titleCell = caseTitle.Cells.Single();

            // Text values are user controls that contain other controls. Using
            // SetControlAndDescendantsPropertyValue makes sure the property is set in
            // all component controls
            api.SetControlAndDescendantsPropertyValue(
                titleCell.ValueControl,
                "FontSize",
                20);
        }
    }
}
```

2.3.2 Adding Advanced Validation Code

Advanced validation code is usually added to the Submitting event. In this event you can search for the data you need and validate it. If data is not valid, you can display an error message to the user (using the `ShowError` method) and return true to cancel form submission.

```
public bool Submitting()
{
    // Check that Freight Amount isn't higher than Quote Amount when creating a new Quote
    bool cancel = false;

    // Use the name of the entity in form schema
    IAgileLightEntity newQuote = form.Data.GetEntity("New Quote");

    // We have to check if the case has been loaded, because it might not have been
    // loaded yet
    if (newQuote != null)
    {
        // Use the internal name to search for attributes
        IAgileLightAttribute quoteAmount = newQuote.GetAttribute("quoteAmount");

        // We have to check if the case title has been loaded, because it might not
        // have been loaded yet
        if (quoteAmount != null)
        {
            IAgileLightAttribute freightAmount = newQuote.GetAttribute("freightAmount");
            if (freightAmount != null)
            {
                if ((decimal)freightAmount.Value > (decimal)quoteAmount.Value)
                {
                    api.ShowError(null, "Freight Amount can't be higher than Quote Amount");
                    cancel = true;
                }
            }
        }
    }
    return cancel;
}
```

2.3.3 Adding Conditional Formatting Code

Conditional formatting code (like displaying negative numbers in red) is usually added to ValueChanged event. Here you can access the attributes you want, check their values, and then access the controls you need and change their formatting.

```
public void ValueChanged(IAgileLightAttribute attribute)
{
    // When Quote Amount changes, use red to indicate negative values
    if (attribute.AttributeName == "quoteAmount" &&
        attribute.Parent.EntityKey == "newQuote")
    {
        // Changes color in all controls bound to this attribute (there might be
        // more than one)
        foreach (IAgileLightCell cell in attribute.Cells)
        {
            // Uses api to change property no matter the type of controls
            api.SetControlAndDescendantsPropertyValue(
                cell.ValueControl,
                "Foreground",
                new SolidColorBrush(((decimal)attribute.Value < 0) ?
                    Colors.Red : Colors.Black));
        }
    }
}
```

2.4 Uploading a Form Handler

To upload your handler, follow these steps:

1. Compile your handler project
2. Open your form in design-mode
3. Click the Upload Handler… button
4. Navigate to the output folder for your handler project (usually bin\Debug or bin\Release under your project folder)
5. Select all the assemblies required by your handler (only assembly files are supported). See handler limitations below

You're now ready to test your Handler.

2.5 Monitoring the execution of your Handler

If you press `Ctrl+F12`, a trace panel will be open on the lower side of your form.
The trace panel will display a trace whenever your form handler is called.
If your handler generates an exception and doesn't handle it, the unhandled exception will also be traced.
You can add your own traces using the `WriteLine` methods in `IAgileLightApi`.
This is an easy and non-intrusive way to get execution data from production environments.

2.6 Debugging a Form Handler in Visual Studio

Class Library Projects can't be started. To debug your Form Handler, follow these steps:

1. Add an Existing Web Site to the solution that contains your Form Handler
2. Select the ALF Site on your development server (usually http://localhost:9999)
3. Set the Web Site project as your Startup Project
4. Open the Web Site project Property Pages
5. Select Start Options on the left
6. In Start URL, enter the URL of the form whose handler you want to debug. You can get the URL by following these steps:
    a. You should have a running instance of your process, waiting for a manual task to be completed
    b. Remember you can use Process Manager to change Process Flow so that you can advance / rollback to the manual task you want to debug
    c. Open the form whose handler you want to debug
        - For CRM activities, open the activity in CRM Workplace
        - For External tasks, open the task from SharePoint portal
    d. Press Ctrl+N so that Internet Explorer opens a New Window in which the URL is visible
    e. Copy the URL and paste it in the Start URL field of ALF Web Site Start Options
7. Under Debuggers, select Silverlight as your only debugger

You are now ready to debug your Form Handler.
Hint: For AgileLightForms embedded in CRM Activities, submitting the form won't complete the CRM Activity (this has to be done by clicking on Save as Completed button). This allows you to submit your form as many times as you want during your debugging session.
For External AgileLightForms, submitting the form will complete the task, and the URL you're using to debug will no longer be valid. To avoid this, you can temporarily remove the Complete Task Submit Action. Remember adding this Submit Action again or your process will never advance when the end user submits the form.

2.7 Form Handler limitations

Form Handlers are quite useful, yet they have some limitations you should be aware of:

2.7.1 File Types

You can upload several files as components of your Form Handler, but only Assemblies (.dll files) are currently supported. So, you can reference other assemblies (like third party controls), but you cannot use external resources (.jpg files, etc.). You can overcome this limitation by embedding those resources inside of your assembly, or deploying them to a web site and accessing them from there.

2.7.2 Handler Size

Due to some limitations in current version of Silverlight, all handler assemblies are loaded before form startup. So, upload only the assemblies that are really required for your handler. If you upload unnecessary assemblies, your forms will take longer to load.
Hint: Embed as few large resources (large images, sounds, videos, etc.) as possible in your handler assembly. You can deploy them to a Web Site and access them from code when they are really needed. This will save both time and memory on your end-user's computers.

3. Editing the Form User Interface

Form definition is stored as an attachment called AgileLightForm.zip in form entity in CRM. AgileLightForm.zip is a compressed archive that contains a file called AgileLightForm.alf. Finally, this is an XML file that contains, among other things, the XAML definition of the user interface.
Form designer allows you to export form definition to a Visual Studio project that can be opened with Expression Blend, Visual Studio, or other tools.

3.1 Exporting Form Definition

To export the definition of a form, open form designer and click on Export Form... button. Select the location for form definition in your hard disk. Form definition is a Visual Studio solution, made up of many fields.
For security reasons, Internet browsers can only write to the file selected by user in Save File dialog. To avoid asking the user for the location of every individual file, the whole solution is saved in a compressed archive, usually called FormDefinition.zip.
To access form definition, extract archive contents in an empty folder and open the solution with Expression Blend, Visual Studio, or any other application capable of editing Xaml files.

3.2 Editing your Form Definition

This Export / Edit / Import mechanism was designed to target some features, listed below. Nonetheless, both Visual Studio and Expression Blend (as well as other XAML editors) can change the form far beyond the capabilities of ALF. As a rule of thumb, only the features listed below are supported, although other changes might work too.

3.2.1 ALF XAML Structure

First, we'll take a look at XAML in AgileLightForms.

Form

Form is defined in Main.xaml. It is a control of type ap:ApTab, that inherits from standard Tab control. ApTab contains standard TabItem controls, but they must comply with the following guidelines.
Every Tab should have a name. Any name can server as a Tab name, although ALF uses GUIDs to make sure they are unique.
All ALF Tabs follow this pattern:

```
<basics:TabItem
    Name="7b5b80f6-b3e6-48cc-8e90-940816d7995d"
    ap:ApForm.TabName="7b5b80f6-b3e6-48cc-8e90-940816d7995d"
    Header="{Binding Converter={StaticResource LabelConverter},
        ConverterParameter=7b5b80f6-b3e6-48cc-8e90-940816d7995d}">
        <ap:DeferredContentControl ContentName="7b5b80f6-b3e6-48cc-8e90-940816d7995d" />
</basics:TabItem>
```

- **ap:ApForm.TabName:** This is the tab name. This is the name you must provide in handlers if you use the GetTab method.
- **Name:** Although not mandatory, it is recommended to set the standard attribute Name to the same value.
- **Header:** To leverage the ALF multilanguage support, use a construction similar to the one in the sample. ALF uses the Tab name as an index in the Label dictionary.
- **ap:DeferredContentControl:** This is the control that implements the deferred loading mechanism. When this control is made visible, it displays the Loading message and then loads its contents (a control with the provided name). This speeds up form loading, deferring the load of tabs not yet visible.

  When a form has a handler, all controls (even hidden tabs) are loaded when the form loads. This is done to simplify handler code, allowing it to assume that all entities and controls are loaded after the DataLoaded events have been fired.

Tabs

As we've just seen, every TabItem control connects to its content control through the ap:ApForm.TabName control.
You'll find the Tab Contents control in a xaml file with the same name of the tab in the exported project.
A Tab Content control is a Grid and looks like this:

```
<Grid ap:ApForm.TabName="a22cc930-e91f-47ea-8990-67c87da38cd6">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="1*"/>
    </Grid.RowDefinitions>
    <StackPanel
        Orientation="Vertical"
        Name="2854b939-12d7-4d0b-9d6d-750c082480e3"
        ap:ApForm.SectionName="2854b939-12d7-4d0b-9d6d-750c082480e3"
        Grid.Row="0">
    </StackPanel>
    <controls:DockPanel
        LastChildFill="true"
        Name="7b5291b9-a596-4fe2-84ef-847e0ad36f0e"
        Grid.Row="1"
        ap:ApForm.SectionName="7b5291b9-a596-4fe2-84ef-847e0ad36f0e">
</Grid>
```

- **ap:ApForm.TabName:** This is the name of the Tab. Must be the same as the one in the containing deferred content control
- **Name:** Although not required by ALF, it's a good practice naming the XAML control as the ALF section.
- **Grid.RowDefinitions:** Fixed height sections are Height="auto", while variable height sections are Height="1*". You can change this to make one section stretch more than others (with Height="2*", for example)
- **StackPanel:** Fixed height sections are StackPanels with vertical orientation
- **DockPanel:** Variable height sections are DockPanels with LastChildFill set
- **Grid.Row:** Indicates the position of a section in the tab
- **ap:ApForm.SectionName:** This is the name of every section, either fixed height or variable height

Sections

As we've seen, sections are either StackPanels (fixed height) or DockPanels (variable height) in a row of a tab, with the attribute ApForm.SectionName.
As tab names, section names can be any string. ALF uses GUIDs to ensure they are unique. Section name is what you must pass to GetSection attribute in your handler to search for a specific section.
Similar to Tab contents, section contents are in a grid too. Sections do also have a TexbBlock for the label, and a rectangle for the bar below the label.
This is the structure of a fixed height section:

```
<StackPanel
    Orientation="Vertical"
    Name="2854b939-12d7-4d0b-9d6d-750c082480e3"
    ap:ApForm.SectionName="2854b939-12d7-4d0b-9d6d-750c082480e3"
    Grid.Row="1">
    <TextBlock
        Name="2854b939-12d7-4d0b-9d6d-750c082480e3#label"
        Text="{Binding
            Converter={StaticResource LabelConverter},
            ConverterParameter=2854b939-12d7-4d0b-9d6d-750c082480e3}"/>
    <Rectangle
        Name="2854b939-12d7-4d0b-9d6d-750c082480e3#bar"
        Fill="Black"
        Height="1"/>
    <Grid/>
</StackPanel>
```

And this one is the one for a variable height section:

```
<controls:DockPanel
    LastChildFill="true"
    Grid.Row="0"
    Name="7b5291b9-a596-4fe2-84ef-847e0ad36f0e"
    ap:ApForm.SectionName="7b5291b9-a596-4fe2-84ef-847e0ad36f0e">
    <TextBlock
        controls:DockPanel.Dock="Top"
        Text="{Binding
            Converter={StaticResource LabelConverter},
            ConverterParameter=7b5291b9-a596-4fe2-84ef-847e0ad36f0e}"/>
    <Rectangle
        controls:DockPanel.Dock="Top"
        Fill="Black"
        Height="1" />
    <Grid/>
</controls:DockPanel>
```

As you can see, the names of the label and the bar are based on the section name. Also, you can see

that the multilingual label text is retrieved using the same mechanism we so for tab headers.
In any case, section content is a Grid similar to this one:

```
<Grid Background="Transparent">
    <Grid.RowDefinitions>
        <RowDefinition Height="35"/>
        <RowDefinition Height="1*" MinHeight="70"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="1*" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="2*" />
    </Grid.ColumnDefinitions>
    ...
</Grid>
```

Section contents grid is divided in rows. Rows containing a variable height control have a Height=”1*”, whereas rows containing fixed height controls only have a Height=”35”. Rows with controls with minimum height other than 1 have a MinHeight=”70”, depending on how many rows that control must occupy.
Columns, on the other hand, define the section layout. Usually, label column with is “Auto”, whereas control column width is usually “1*”. Previous sample shows a section with a “Two Columns (1:2)” layout.

Cells

In ALF, a cell is a label – control pair. While label displays the name (and requirement indicator) of an attribute, the control displays (or edits) its value.
When the label is visible, it occupies the grid cell immediately before the grid cell occupied by the attribute value control.
When the label is not visible, the value control expands to occupy its space. The value control can also expand several columns.
What glues both controls together to create a form is the attribute ap:ApForm.CellName. As Tabs and Sections, Cells have a name that can be any string, but ALF uses a GUID to ensure they’re unique. This is the name you have to pass to the method GetCell.
Cell name, like tab and section ones, is also used like an index to the multilingual label text.

Data Binding

Attribute value controls have a property bound to the attribute value. This binding is done through a clause similar to this one:

```
Text="{Binding
    Self,
    Mode=TwoWay,
    Converter={StaticResource AttributeConverter},
    ConverterParameter=Existing Account.name}"
```

The ConverterParameter is what actually binds a control property (Text in this case) to an attribute Value (in this case, the value of attribute “name” in entity “Existing Account”.
The name of the entity is the name given in Schema Designer, whereas the name of the attribute is the CRM internal attribute name.
Apart from control property to entity attribute binding, there are two other attributes used by forms designer to relate controls to entities and attributes:

- **ap:ApForm.EntityKey:** Indicates the key of the entity (in Schema designer) to which the control is bound.
- **Ap:ApForm.AttributeName:** Indicates the internal name of the attribute to which the control is bound.

All data-bound controls must implement a Boolean property called IsReadOnly. If the control you want to use does not implement this property, you must wrap it in another control that implements it. You can do it by either inheritance (creating a control that inherits from it and adds the IsReadOnly property) or containment (creating a UserControl with an IsReadOnly property than contains the original control).

3.2.2 Editing control properties

You can freely change the properties of any of the controls in your ALF form. You should keep form structure so the form keeps being editable by forms designer.

3.2.3 Adding unbound controls

You can add controls to your form. ALF designer will just ignore them. Take into account that any new assembly you reference from your form project will have to be loaded whenever the form is displayed. Reference necessary assemblies only. You should keep form structure so the form keeps being editable by forms designer.

3.2.4 Substituting bound controls

You can change the default ALF control for an attribute value to a different control (even a third-party control). Just move the following XAML attributes from the standard control to the new one, and remove the standard one:

- Name
- ap:ApForm.CellName
- ap:ApForm.EntityKey
- ap:ApForm.AttributeName
- Grid.Row
- Grid.Column
- Grid.RowSpan
- Grid.ColumnSpan

Remember that bound control must have an IsReadOnly property. If it doesn't, you must either inherit from it or create a UserControl to contain it.

3.2.5 Unsupported changes

These are some examples of changes that won't work.

Form Structure

In general, making any change that does not respect form structure will make the form unusable in design-time and / or run-time.

Code

This Export / Edit / Import mechanism is designed to edit form design (XAML). Don't add any code to your form, because it will be ignored.
If you want to add code to your form, create a form handler instead.

Transformations

Form designer assumes that cells, sections, tab headers, and tab contents are upright rectangles. If you use Scale, Rotate or Skew transformations, form designer might not be able to select and / or highlight your cells, sections and / or tabs.

3.2.6 Hints

Always create a backup of your form before editing it (by either saving the AgileLightForm attachment or exporting the forms with the AgileLightForms Mover application). Exporting forms is a very powerful tool, but might render your form unusable.
Reference only necessary assemblies. All referenced assemblies will be saved with your form, increasing both its size and its load time.
Stick as much as possible to ALF XAML structure. If, for example, you want to add a picture control to a section, do it inside a cell of the section contents grid. The more you adhere to ALF structure, the better that form designer will work after importing your changes.

3.3 Importing Form Definition

You must follow these steps to import form definition back into ALF. First, you must compile your form. You might wonder why should you compile your form if code is not being taken into account. In fact, compilation is necessary to bring referenced assemblies to the output folder of the solution.
Once you have compiled your solution, compress all your files back in a zip archive. Don't worry for it having many irrelevant files. ALF will only take the ones needed to import form design. Make sure the solution file is in the root of the archive, i.e., don't compress the folder containing form solution; compress the files and subfolders in the folder containing form solution.
The last step is opening your form in ALF designer, clicking Import Form… button, and browsing to you form definition. If everything works fine, you should see your edited form in form designer. Take into account that some features of form designer might stop working, depending on the changes made to your form.