Guide for Building Extensible Controls

# Introduction

In the Rainier we are introducing the ability for developers to extend the existing application UI as well as define entirely new UI patterns to create compelling new user experiences. Using modern tools such as HTML5, CSS3 and jQuery developers are free to define customized visualizations of business data and drastically enhance the application’s interaction patterns.

## Server-side architecture

The Control Extensibility Framework leverages the existing and familiar X++ language for developing server-side data access and business logic. We do not place artificial restrictions on the code developers write when building Extensible Controls, we instead allow developers to declaratively define the modeling experience and the runtime behavior through a set of X++ class attributes and method attributes. The developer must define one X++ class to govern the control’s design-time behavior (the form modeling experience). The design-time X++ class is referred to as the [X++ Build Class](#_The_X++_Build). The developer must also define one X++ class to govern the runtime behavior of the control. The runtime class is referred to as the [X++ Runtime Class](#_The_X++_Runtime). Together, the X++ Build Class and the X++ Runtime Class are referred as the Logical Control (the server-side control).

## Client-side architecture

The client-side behavior for the control is defined using HTML and JavaScript. In the context of a Model-View-View Model architecture, the HTML for the control comprises the "View", the JavaScript comprises the "View Model", and the “Model” is represented by the client-side copy of the form data source that the client-side form runtime provides. Together, the HTML View and the JavaScript View Model are referred to as the Physical Control (the client-side control).

The Control Extensibility Framework provides an HTML-based binding syntax that enables binding HTML elements within the View to properties and commands in the JavaScript View Model. The binding syntax also provides and API for defining visualization behavior that reacts to expressions which are computed from properties or commands in View Model. The X++ Runtime Class uses attributes to determine which properties and commands are available in the JavaScript View Model. For each property (or command) exposed by the X++ Runtime Class, the JavaScript View Model is initialized with a client-side representation of the property. The X++ Runtime Class properties and the JavaScript View Model properties are bound and synchronized by the framework, allowing the developer bind the HTML View to the properties and commands defined in X++. If the developer wishes to add client-side only properties or commands (those with no backing X++ logic) to the JavaScript View Model, or wishes to implement visualization behavior that must be defined in JavaScript, they may extend the View Model beyond those properties and commands passed from the X++ Runtime Class on the server. Using client-side only properties and commands the developer may leverage the powerful JavaScript APIs provided by the framework to implement advanced control interaction patterns, leverage 3rd party JavaScript libraries and leverage the powerful jQuery library.

## Architecture Diagram

The following diagram visually describes the artifacts involved in the Control Extensibility Framework and their relation to each other.

3

4

1

2

5

6

7

## [The X++ Build Class](#_Creating_X++_Properties)

The developer writes the Build Class to define the design-time behavior for a control. This includes defining the design-time properties for the control which appear in the Visual Studio property sheet, determining the data accepted by those design-time properties (data source names, data field names, enums, menu item names, etc.), determining how to manage children controls and modeling components, default values for the properties, and more. The Build Class is consumed by the Form Designer to display the properties in the property sheet, among other design-time behaviors. The X++ Build Class is described in detail in the section titled [X++ Build Class](#_Creating_X++_Properties).

## [The X++ Runtime Class](#_X++_Runtime_Class)

The developer writes the Runtime Class to define the runtime behavior of a control. This includes defining the runtime properties and commands as well as implementing data access patterns and business logic. The framework serializes the properties and commands in the logical control with the appropriate method attributes, and makes them available to the physical control in the client. The X++ Runtime Class is described in detail in the section titled [X++ Runtime Class](#_X++_Runtime_Class_1).

## [AOT Resources](#_Visual_Studio_Setup)

AOT Resources are used to organize the CSS, JavaScript, HTML, Images, and media files for Extensible Controls. The resources are consumed by the Client/Browser to render the control. These resources have no dependency on any part of the X++ development of an extensible control, meaning you cannot create the HTML or directly call JavaScript using X++, nor vice-versa. The Control Extensibility Framework handles the binding between the logical and physical control. The decoupling of the logic and physical control is intentional, and is a design imperative of the framework. AOT Resource and other file management topics are discussed in the section titled [Visual Studio Setup](#_Visual_Studio_Setup).

## [The HTML View](#_HTM_file)

The HTML for the control contains declarative markup that is interpreted at runtime to display business data and handle user interaction patterns. Developers may design the layout and style of a control with standard HTML, then supply the framework’s attribute-based binding syntax to link UI elements to business data in the View Model. The binding syntax also provides the ability to declaratively define style and layout behaviors such as repeating elements and conditional visibility of elements. Designing the HTML View is described in more detail in the section titled [HTM Development](#_HTM_Development).

## [The JavaScript View Model](#_JS_Development)

The View Model for the control may be used to perform many tasks related to data access, visualization and integration with 3rd party libraries. Chief among these tasks is providing the View with the appropriate data and executing client-side business logic.   
  
When instantiating the physical control, the framework generates and passes a JSON property bag to the View Model constructor for the control. This property bag contains the set of serialized properties and commands that mirror the properties and commands declared in the logical (server-side) control by the X++ Runtime Class. The View Model constructor consumes the property bag and exposes the properties to the HTML view. Outside of exposing the server-side properties to the HTML view, the View Model provides a place for developers to use JavaScript for building a wide range of compelling visualizations, interactivity, and external integrations. Any properties or commands that are declared in the View Model, rather than passed into the constructor by the JSON property bag, are client-side only properties and have no corresponding server-side code. Building the JavaScript View Model is described in more detail in the section titled [JS Development](#_Bindings).

## The Interaction Service

The Interaction Service ensures that the state of the logical control and physical control are synchronized appropriately. In particular, it ensures that changes executed on the client (command executions, property value changes and data source updates) are propagated in the correct sequence to the server form, and vice versa. A detailed description of the Interaction Service and its behaviors is outside the topic of this document, but we will refer to the Interaction Service as it pertains to the Control Extensibility Framework.

## The Form Interaction Runtime

The Form Interaction Runtime represents the client-side instance of the Form and is responsible for maintaining the state of the Form in the client. This includes maintaining the state of all controls (including nested Forms) and all data sources currently in use by the Form(s). The runtime is also responsible for batching client-side events (such as navigation, data source updates, and command executions) into Interactions to be synchronized with the server-side instance of the Form through the Interaction Service. It is important to understand the timing of Interaction synchronizations as it pertains to controls as there will be occasions when the physical control and the logic control are out of sync.  
  
Usually, Interactions are batched in the client before being sent to the server. For instance, a user my enter data into two fields of a control. These two field changes result in changing the value of two properties on the control. The two property changes are batched into an Interaction, and the client continues listening for more property changes to add to the Interaction before sending the Interaction to the server. This batching mechanism prevents the client from becoming overly “chatty” and sending unnecessary network traffic after every small change in the client. The client only sends an Interaction to the server when the user takes action that requires server-side logic to run immediately before the user can take the next action. For instance, a save action will cause the client to “flush” the current Interaction to the server because server side logic needs to process the save before the user can continue to use the client safely. Some other examples of events that will cause an Interaction to be flushed are Form navigations, CRUD operations, [command](#_Creating_Control_Commands_1) executions or a value change of an [auto post-back enabled property](#_Auto-Postback).

# Building the physical Control

## Overview of the physical control

There are 3 main artifacts that comprise the physical control, collectively referred to as a resource bundle. An HTM file plays the part of head of the resource bundle. When a Form instantiates a control it first loads the HTM file, the HTM file is then responsible for indicating the other files in the resource bundle that need to be loaded (such as JavaScript or CSS files). This process is further described in [Resource Bundles](#_The_Resource_Bundle). A resource bundle consist of:

* An HTM file containing standard HTML elements, and optionally consumes Dynamics-specific HTML attributes and functions.
* A JS file containing standard JavaScript that serves as a View Model for the HTML (the View)
* A CSS file containing standard CSS styles and rules that apply to the elements in the .HTM file

### HTM file

The HTM file contains the "View" of the control. The HTML View is the primary artifact of the physical control and is responsible for the control's appearance in the client. Standard HTML is leveraged in the HTM file to produce the control's visualization. In conjunction with the standard HTML, the Control Extensibility Framework provides a set of simple-yet-powerful [HTML binding attributes](#_Data_Binding_Overview) to address various methods of interacting with business data, business logic, or the JavaScript View Model.

### JS File

This JavaScript file contains the View Model for the control. The JavaScript View Model encapsulates control-specific properties, commands and visualization behaviors. The View Model exposes the properties and commands defined in the [X++ Runtime Class.](#_X++_Runtime_Class) A developer may [extend the View Model](#_Extending_the_View) to include additional client-only properties, commands or other JavaScript functions.

### CSS File

Each control may leverage CSS to style the HTML elements contained within the control. CSS classes should be appropriately name-spaced to prevent clashes with the styling of other controls.

### Resource Bundles

A resource bundle describes a set of web resources (HTM, CSS and JS files) required by a control. The first occurrence of a control on the Form causes the resource manager to fetch the associated HTM file (the head of the resource bundle). The HTM file is parsed for references to additional resource bundle files, and a client-side resource manager loads the additional resource bundle files asynchronously. Subsequent occurrences of the control on a Form do not require fetching the control's resource bundle files again.

## Visual Studio Setup

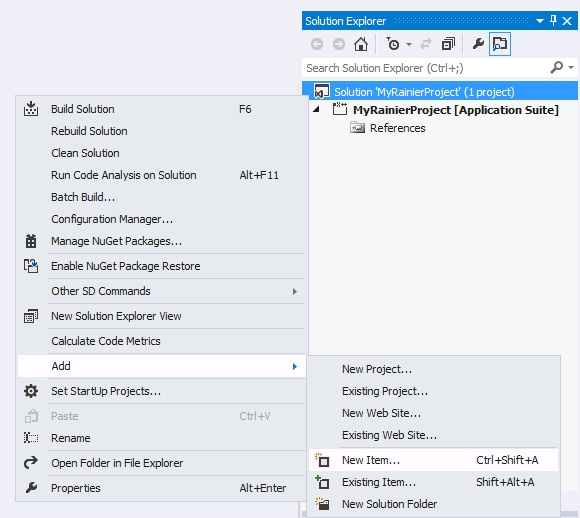
When developing controls you must use the Rainier development tools for Visual Studio. The control resource bundle files must be stored in Ax Resources to ensure they are saved and deployed correctly. After storing the resource bundle files in Resources, they are deployed and available to the web client upon building the containing Rainier Project. On a OneBox development environment, the files are deployed to C:\CustomerServiceUnit\DObind\Packages\Cloud\AosWebApplication\AosWebApplication.csx\roles\AosWeb\approot\Resources\[file extension specific folder]

The following steps will walk through creating the minimum artifacts needed to get a “Hello world” control working in the client. After getting this basic “Hello world” control working, we will go in depth on the various topics for control building.

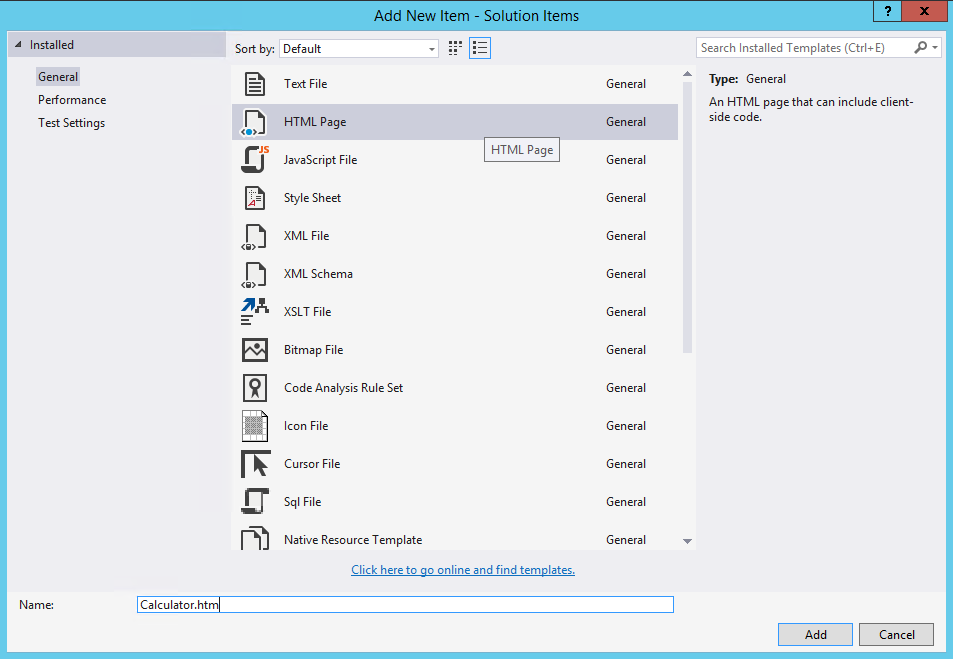
### Create basic resource files for control

We will begin by creating the **.HTM** file. The other Resource Bundle files (the **.JS** and **.CSS**) follow the same pattern.

1. Open **Visual Studio** and create a new **Rainier Project** (or open an existing project). Ensure the project model is set to Application Suite.
2. Right click on the **Solution** (not the **Rainier Project**) and select **Add > New Item …**

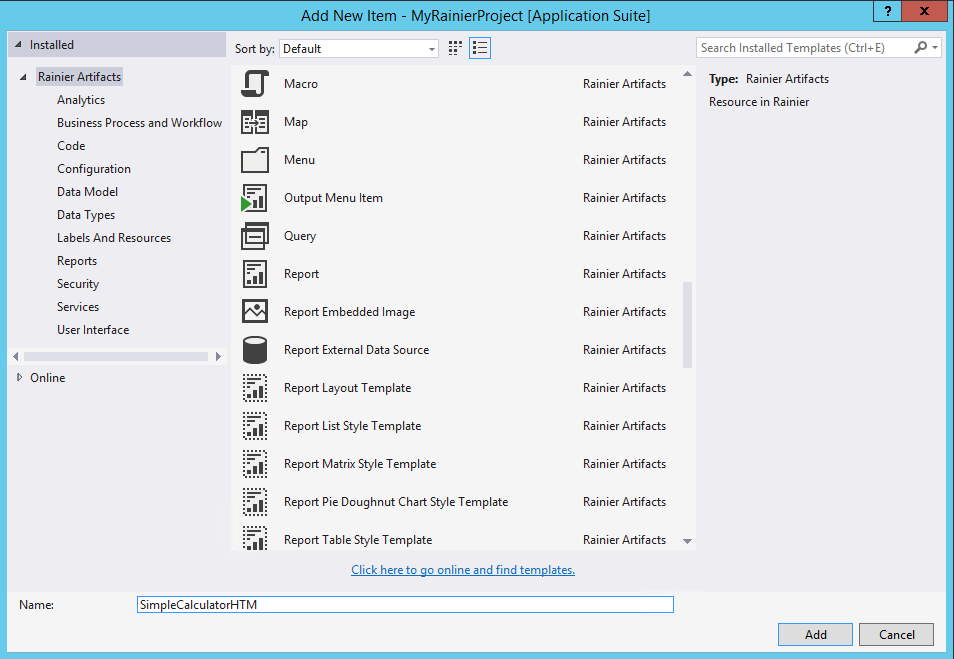


* 1. In the **Add New Item** window select **General** from the left-hand pane.
  2. Select **HTML Page** from the center pane.
  3. Supply a name for the **HTML Page**, be sure to provide the file extension "**.htm**" instead of “**.html”**
  4. Click **Add**



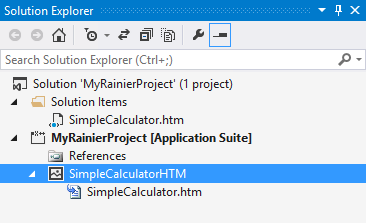
* 1. The newly created **.HTM** file appears in the Visual Studio HTML editor

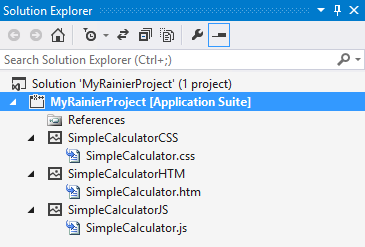
1. Click the **.HTM** file in the **Solution** and note in the **Properties** windows the **File path**. You will navigate to this **File path** when creating the **Resource** below
2. Right click on the **Rainier Project** and select **Add > New Item …**
   1. In the **Add New Item** window select **Rainier Artifacts** from the left-hand pane
   2. Select **Resource** from the center pane.
   3. Supply a name of the **Resource** that will contain the **.HTM** file. Use the following naming convention: [ControlName][FILE EXTENSTION]
   4. Click **Add**. If another **Resource** already exists with the same name you have provided, then provide a different unique name to the **Resource**.



* 1. In the **Select a Resource File** window, navigate to the **File path** noted above
  2. Select the **.HTM** file previously created
  3. Click **Open**

1. Locate the newly created **Resource** in the **Rainier Project**



1. Upon the creation of a **Resource,** a copy of the original file is created and placed in the model store under AxResources/ResourceContent/[sub folder based on file extension]. You may delete the original **.HTM** file located in the **Solution Items** folder as it is no longer necessary.
2. Repeat these steps to create the .JS and .CSS files for the Resource Bundle. Your solution should look similar to the following after adding the .JS and .CSS files.  
   
3. Open the HTM file you’ve created by double-clicking on it in the Solution Explorer.
4. Copy the following HTML into the HTM file you’ve created.

<script src="/resources/scripts/SimpleCalculator.js"></script>

<link rel="stylesheet" src="/resources/scripts/SimpleCalculator.css"></link>

<div id="SimpleCalculator">

<div>Hello World!</div>

</div>

1. Save the HTM file.
2. Open the JS file you’ve created.
3. Copy and paste the following JavaScript code into it

(function () {

'use strict';

$dyn.ui.defaults.SimpleCalculator = {

};

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

};

$dyn.controls.SimpleCalculator.prototype = $dyn.extendPrototype($dyn.controls.Input.prototype, {

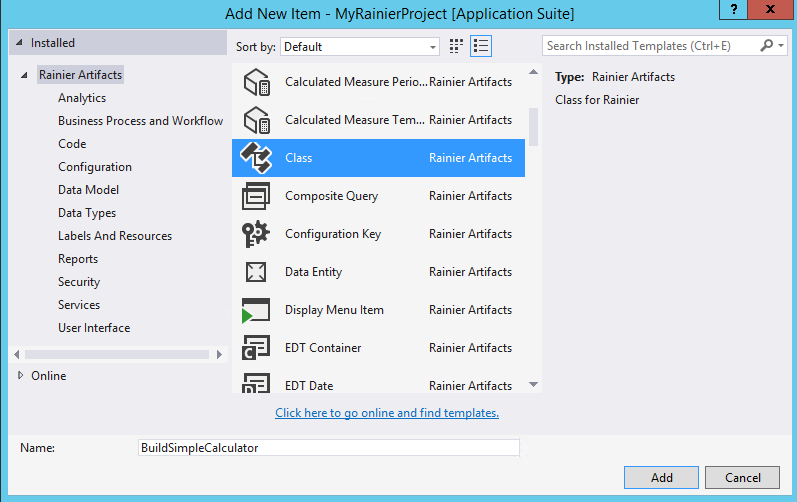
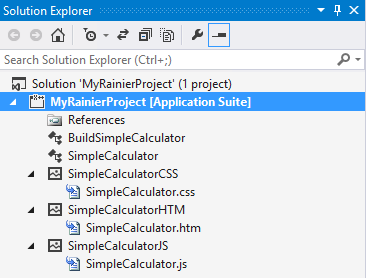
});

})();

1. Save the JS file
2. This is the minimum HTML and JavaScript needed for a control to load in the client. The details of these files are described further below in [HTM Development](#_.HTM_Development) and [JS development](#_Bindings) below.

### Create basic X++ Classes for the control

We will create only enough of the X++ Runtime Class and X++ Build Class to add the control to a form and have the control appear in the client.

1. To create the X++ Build Class, right-click on the **Rainier project** and select **Add > New Item**
   1. In the **Add New Item** window select **Rainier Artifacts** from the left-hand pane
   2. Select **Class** from the center pane.
   3. Supply a name of the **Class** that will be the X++ Build Class for the control. Use the following naming convention: Build[ControlName]
   4. Click **Add**  
      
2. To create the X++ Runtime Class, follow the same process as above, using the naming convention: [ControlName]
3. The Solution should now have 5 items, as shown below.  
   
4. Open the X++ Build Class by double-clicking on it in the **Solution Explorer**
5. Copy and paste the following code into the class

[FormDesignControlAttribute("Simple Calculator")]

class BuildSimpleCalculator extends FormBuildControl

{

}

1. Save the X++ Build Class
2. Open the X++ Runtime Class by double-clicking on it in the **Solution Explorer**
3. Copy and paste the following code into the class

[FormControlAttribute("SimpleCalculator","SimpleCalculator",classstr(BuildSimpleCalculator))]

class SimpleCalculator extends FormTemplateControl

{

public void new(FormBuildControl \_build, FormRun \_formRun)

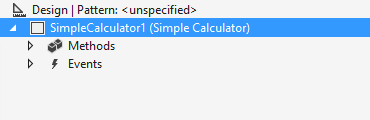
{

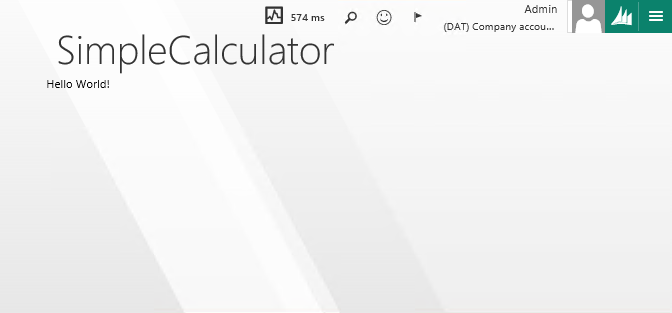
super(\_build, \_formRun);

this.setResourceBundleName("/resources/html/SimpleCalculator");

}

}

1. Save the X++ Runtime Class
2. Build the **Rainier Project**
3. Next, add a form to the project to host the control.
   1. Right-click the **Rainier Project** and select **Add > New Item**
   2. In the **Add New Item** window select **Rainier Artifacts** from the left-hand pane
   3. Select **Form** from the center pane.
   4. Supply a name to the form. There is no specific naming convention that needs to be following for forms that host controls
   5. Click **Add**
4. Now add the control that we’ve created to the form
   1. Double-click on the form in the **Solution Explorer** to open the form in the **Form Designer**
   2. Right-click on the **Design** node of the form, then select **New > Simple Calculator**(if **Simple Calculator** doesn’t not appear in the list, ensure that you have correctly copied and pasted the code for the X++ Build Class, and that you have built the **Rainier Project**.)
   3. You should see that the **Simple Calculator** has been added to the form  
      
   4. Next, run the form
      1. Right-click on the form in the **Solution Explorer** and click **Set as startup object**
      2. Press **Ctrl+F5**, or click **DEBUG > Start without debugging** to run the form
      3. The SimpleCalculator form shows the SimpleCalculator control containing “Hello world!” text



## HTML and JavaScript development

### Control Definition and TemplateID

Let’s inspect the HTML used in the example above. Below, we have highlighted the TemplateID for the control as well as the Control Definition

<script src="/resources/scripts/SimpleCalculator.js"></script>

<link rel="stylesheet" src="/resources/scripts/SimpleCalculator.css"></link>

<div id="SimpleCalculator">

<div>Hello World!</div>

</div>

For every control, inside of the HTM file, a *div* element must be provided, along with an *id* attribute whose value defines the physical control’s type. The physical control type is also referred to as the templateID throughout development. Each control type/templateID should be uniquely defined across all controls. The templateID is a universal reference to the control when it is [referenced by other control definitions](#_Control_Composition).

For most control building scenarios we encourage naming the physical HTM file the same as the templateID. There are cases where the names might not match, described in [Multi-Template Resource Bundles](#_Multi-Template_Resource_Bundles).

The HTML inside of the div with the templateID is referred to as the Control Definition. Within the Control Definition you can use any of the standard HTML elements, attributes and other HTML artifacts. We do not restrict the usage of standard HTML when defining Controls. We instead supply an HTML binding syntax on top of the standard HTML syntax. Using this binding syntax a developer may express binding behaviors that are interpreted by the framework at runtime.

### Data Binding Overview

The HTML Control Extensibility Framework provides attributes that can be used on any HTML element in the Control Definition to bind the HTML properties to business data and presentation logic. The basic syntax is extremely simple but also allows for very complex binding patterns.

#### Data Binding Syntax

<div data-dyn-bind="[binding handler]: [value to bind to]"></div>

A *binding handler* determines how a binding is interpreted. Below is a list of some supported binding handlers. The *value to bind to* may be static (0, 1, 2, true/false, 'string'), a [View Model property](#_Binding_to_View), or derived from a [JavaScript function or expression](#_Binding_to_a).

#### 

#### Binding to Static Data

Let's continue the Simple Calculator control example, this time leveraging binding handlers instead of hardcoding HTML.

1. Copy and paste the following HTML into SimpleCalculator.htm

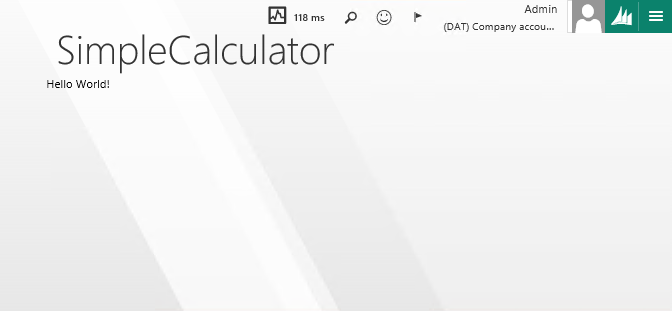
<script src="/resources/scripts/SimpleCalculator.js"></script>

<link rel="stylesheet" src="/resources/scripts/SimpleCalculator.css"></link>

<div id="SimpleCalculator">

<div data-dyn-bind="text: 'Hello world!'"></div>

</div>

1. Save changes, Build and run the form again. The form should render and look the same as before.  
     
   Note: if one occasion is appears as if your changes to HTML or JavaScript are not recognized in the browser, you may have to clear the browser cache. In IE and Chrome, this can be done through the browser developer tools.

The HTML above is functionally equivalent to the original HTML, but replaces the static HTML with data bindings. In this case, we use the text binding handler to bind to the textContext property of the div element. This textContext property is a standard HTML property of elements. You can find more info on the textContent property at [W3CSchools](http://www.w3schools.com/jsref/prop_node_textcontent.asp).

A short list of available binding handlers can be found in the appendix under [List of Binding Handlers](#_List_of_Binding_1).

In practical cases, there is little utility in creating bindings to static values as is done in this example. The benefits of using the Data Binding framework come into play when we bind HTML to View Model properties.

#### Binding to View Model Properties

The View Model for a control servers as the root data binding context for the [control definition HTML element](#_Control_Identifier).

A *data binding context* contains the properties/commands which may be used as [binding values in binding handlers](#_Data_Binding_Syntax). When a binding value is a property or command defined on the context, then framework evaluates the property/command to provide a value to the binding handler. The *data binding context* is available for use in binding parameters via the *$data* alias.  
  
Let’s update our JavaScript View Model and add a property to be available on the context for the control definition.

1. Open the JS for the control
2. Replace the Text property in the JavaScript View Model with the highlighted code below

(function () {

'use strict';

$dyn.ui.defaults.SimpleCalculator = {

};

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

self.Text = "Hello world!";

};

$dyn.controls.SimpleCalculator.prototype = $dyn.extendPrototype($dyn.controls.Input.prototype, {

});

})();

1. Update the HTML View to bind to the property added to the View Model

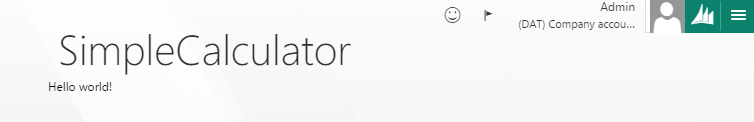
<script src="/resources/scripts/SimpleCalculator.js"></script>

<link rel="stylesheet" src="/resources/scripts/SimpleCalculator.css"></link>

<div id="SimpleCalculator">

<div data-dyn-bind="text: $data.Text"></div>

</div>

1. Save changes, Build, and run the form  
   

Data context is also *inherited* by nested elements from their parent element. The framework sets the data context of the control definition (the root element) to the View Model, and the nested elements in the control definition will inherit the property bag as their data context. Let’s build an example that shows how the data context (accessible via the $data alias) can be changed.

1. Copy and paste the highlighted code into the JavaScript View Model

(function () {

'use strict';

$dyn.ui.defaults.SimpleCalculator = {

};

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

self.MyObj =

{

Text: "Hello world!",

MyObj2: {

SubText: "Foo",

SubText2: "Bar",

},

};

};

$dyn.controls.SimpleCalculator.prototype = $dyn.extendPrototype($dyn.controls.Input.prototype, {

});

})();

1. Copy and paste the following into the HTML View

<script src="/resources/scripts/SimpleCalculator.js"></script>

<link rel="stylesheet" src="/resources/scripts/SimpleCalculator.css"></link>

<div id="SimpleCalculator">

<div data-dyn-bind="text: $data.MyObj.Text"></div>

<div data-dyn-context="$data.MyObj">

<div data-dyn-bind="text: $data.Text"></div>

<div data-dyn-context="$data.MyObj2">

<div data-dyn-bind="text: $data.SubText"></div>

</div>

</div>

</div>

Notice in the example above that the $data is set to the View Model on the first element in the control definition. The MyObj property is accessible from $data. In the next element, we use the data-dyn-context attribute to change the data context. From there one $data is set to the MyObj property, and the members of the MyObj property are accessible from $data. This example serves to highlight that the $data alias is not always set to the View Model of the control, and that it can change at various levels of the HTML. Sometimes the data binding context can be explicitly set by the developer as seen in this example, other times the data binding context can be implicitly changed by other binding handlers.

#### Implicitly changing the Data Context

Some binding handlers dynamically change the data context as a part of their binding behavior. For example, the *foreach* binding handler accepts an array of primitive values or objects. When the array contains values, in each loop iteration the current value is set as the data binding context, and is thus available through $data. When the array contains objects, in each loop iteration the current object is set as the data binding context and is available using $data. In both cases, the current index is available via $index.

Let’s see an example of the foreach binding handler in action.

1. Open the JS for the control
2. Replace the MyObj property in the JavaScript View Model with the highlighted code below

(function () {

'use strict';

$dyn.ui.defaults.SimpleCalculator = {

};

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

self.MyArr = [1, "Foo", "Bar"];

self.MyObjArr = [{Name: "John"}, {Name:"Jane"}];

};

$dyn.controls.SimpleCalculator.prototype = $dyn.extendPrototype($dyn.controls.Input.prototype, {

});

})();

1. Update the HTML View to bind to the property added to the View Model

<script src="/resources/scripts/SimpleCalculator.js"></script>

<link rel="stylesheet" src="/resources/scripts/SimpleCalculator.css"></link>

<div id="SimpleCalculator">

<div data-dyn-bind="foreach: $data.MyArr">

<div data-dyn-bind="text: $data"></div>

</div>

<div data-dyn-bind="foreach: $data.MyObjArr">

<div>

<span data-dyn-bind="text: $data.Name"></span>

</div>

</div>

</div>

1. Save changes, Build, and run the form  
   

Notice that the HTML elements nested beneath the element with the *foreach* binding handler have a dynamics data binding context (via the $data alias). This example serves to show that the $data alias is not always equivalent to the JavaScript View Model, but can be set to properties within the JavaScript View Model explicitly or implicitly. It is important to keep mental notes of when the data binding context changes when getting into advanced control building scenarios.

#### Binding to Expressions and Functions

The previous examples exhibit very simplistic data bindings. There are many cases where it is useful to write JavaScript functions, expressions, or both to provide the binding values to binding handlers.

##### Binding to a function

You may use JavaScript functions as binding values. Using this pattern you can conditionally supply values to a binding handler based on function evaluations. In the example below the element will show the current time as evaluated by the function on the View Model.

1. Add a DateTime function to the JavaScript View Model inside of the control constructor

self.DateTime = function () { return $dyn.util.today();};

1. Add an element to the control definition bound to this function

<div data-dyn-bind="text: $data.DateTime()"></div>

1. Save the files. Build the **Rainier Project**, then run the form.  
   

Note that in this example, we use the today function to get the datetime value. “$dyn.util” contains many useful date/time manipulation functions, and is integrated with the DateTime and User Local systems from the AOS. Use these functions to manipulate and format your date values in a localized manner.

##### Binding to an expression

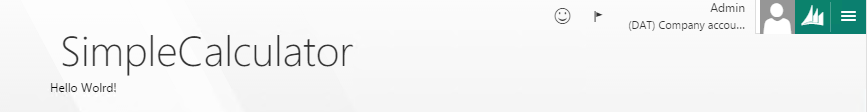
You may supply an expression to a binding handler. This will result in the expression being evaluated and producing a value for the binding handler to consume. In the example below the text will only show when the ShowText view model property is true valued.

1. Add a DateTime function to the JavaScript View Model inside of the control constructor

self.InTheWorld = true;

1. Add an element to the control definition bound to this function

<div data-dyn-bind="text: $data.InTheWorld? 'Hello Wolrd!': 'Where am I?'"></div>

1. Save the files. Build the **Rainier Project**, then run the form.  
   

### Using observables

An important concept to understand during the JavaScript development of a control is the Observable pattern. The Observable pattern allows the developer to declare a property as observable. When a property is observable, other properties or functions can react to value changes of the observable property. Another property or function that is reacting to an observable property is said to be an *observer* of the observable property. When the value of an observable property changes, it automatically notifies all of its observers. This feature comes hand when providing observable properties to binding handlers, because binding handlers are *observers* of observable properties. This means that the binding handlers will re-execute their binding behaviors when the supplied observable property changes. This is how the automatic updating of the UI works in most cases.

Properties defined in the JavaScript View Model must be explicitly defined as observables in order for an observer to be notified of a property change. Because there is a performance impact of using observable properties versus non-observable properties, you should only use observables properties when the property is expected to be dynamically updated at runtime *and* the property plays a part in the rendering of the UI (by being supplied to a binding handler).

There are 4 framework JavaScript functions that allow developers to take part in the Observable pattern:

* Observable - $dyn.observable([initialValue])
  + Creates an observable property, with an optional initial value.
  + Use when creating properties that are [supplied to binding handlers](#_Binding_to_View).
* Observer - $dyn.observe(ObservableorFunction, function(){…})
  + Creates an Observer, observes the supplied Observable for changes and executes its computation when the Observable changes value.
  + Does not return a value.
  + Use when writing code that should be executed when the value of an Observable changes.
  + ***It is a Best Practice to avoid using observers to manually update the value of other properties. Instead, those properties should be created as computed observable properties. This way they can take a passive dependency on other observables.***
* Computed observable - $dyn.computed(function(){…})
  + Creates an observable property, whose value is calculated based on other observables.
  + A computed observable’s property cannot be set directly.
  + Creates an observer that implicitly observes all observable properties that are accessed within the scope of the function. Re-executes its computation when the value of any such observable’s value changes.
  + Use when creating observable properties that rely on the value of other observable properties.
* Value - $dyn.value(observable)
  + Gets the value of an observable property

Below we update the previous JavaScript example to participate in the Observable pattern, enabling the UI to update when the properties' values change.

1. Remove any existing properties in the JavaScript View Model, and add the following observable and computed observable properties

self.Expression = $dyn.observable("1+1");

self.Solution = $dyn.computed(function () {

return $dyn.value(self.Expression);

});

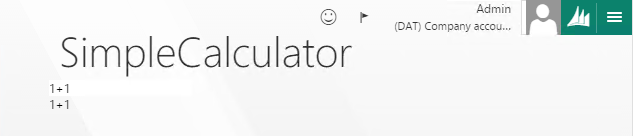
1. Remove any existing HTML inside the control definition (remember the control definition is the element with the templateID). Then add the following HTML to the control definition

<div>

<input data-dyn-bind="value: $data.Expression"></input>

</div>

<div data-dyn-bind="text: $data.Solution"></div>

1. Save the changes. Build the **Rainier Project**. Run the form  
   

Notice that the lower div element is showing the same data as the input element. This is due to the Solution computed observable property, which implicitly observes the Expression observable property and returns the current value of the Expression. Feel free to type into the input element, and notice that then you leave the focus of the input element, the Expression property will update, triggering an update to the Solution property. It’s important to understand the sequence of events in this process.

1. The user enters a value into the input element. As the user enters text into the input element, the input element updates its internal value property. This updating behavior is standard to the HTML input element and not controlled by the rainier control framework.
2. The user leaves the focus of the input element (by pressing TAB or by clicking outside of the element)
3. The *value* *binding handler* that is on the input element notices that the input element has lost focus. The binding handler reads the new value of the input element’s internal value property. The binding handler sets the Expression to the value of the input’s value property.
4. The Expression observable notices that its value has changed. It notifies its observers that its value has changed.
5. The value binding handler is notified that the Expression has changed. However, the new value of the Expression matches the current value, so the value binding handler ignores the change.
6. The Solution computed observable is notified that the Expression has changed. The Solution re-evaluates itself by re-executing the function supplied to its constructor.
7. The Solution observable notices that its value has changed. It notifies its observers that its value has changed.
8. The *text binding handler* that is on the lower div element is notified that the Solution has changed. The text binding handler re-evaluates itself and updates the text property on the div element.

From this sequence of events it’s important to note the following.

* The input element has its own internal value property. The input element doesn’t know anything about observables and has no knowledge of the control framework.
* The value binding handler uses a JavaScript event subscription to know when focus leaves the input element. The value binding handler knows to update the observable that it was supplied when the input value changes.
* The new value for the Expression was set by the value binding handler.
* All observables keep an internal list of all of their registered observers. In this case, Expression has two registered observers: the value binding handler and the Solution computed observable. Solution has only one registered observer, the text binding handler.
* If you look at the internal implementation of the value or text binding handlers, you will see that they each make a call such as “$dyn.observe(observableValue)”, where observableValue is either the Expression or Solution, depending on which was provided to the binding handler.
* The text binding handler sets the text on a div element using the textContent property of the element.

Study well this example of observables, computed observables and observers. A thorough understanding of these concepts is a necessary foundation for building extensible controls.

### Control Composition

An Extensible Control may compose itself of other controls in order to leverage the existing visualization logic contained in the Views and View Models other controls. Leveraging the functionality of existing controls, as opposed to reproducing their functionality, saves development time by reducing code duplication, improving code maintainability and assuring consistent UI behavior and patterns across the application. This process is commonly referred to as Control Composition. The *role attribute* allows a developer to instantiate other controls within a control definition. The role attribute accepts the control type/templateID which corresponds to the type of the desired control (String, Grid, Tab, etc.).

#### Role Attribute Syntax

<div data-dyn-role="[templateID]"></div>

By default, the framework only loads the resource bundle which contains the basic controls. In order to use controls that are not loaded in the basic control set, the role attribute must be supplied the resource bundle containing the control.

Role Attribute Syntax for external resource bundle

<div data-dyn-role="Control: {type: [templateID], resource: [path to resource bundle]}"></div>

After instantiating a control using the role attribute, you will need to initialize the control's View Model properties using the [bind attribute](#_Data_Binding_Syntax). Because each control is responsible for its own rendering behavior *you cannot use the HTML binding handlers to manipulate the rendering behavior of a control*. Instead of supplying one of the [HTML binding handlers](#_Data_Binding_Syntax) to the bind attribute, supply the bind attribute with the View Model property names you wish to bind to. The examples below exhibit the usage of the role and the bind attribute when instantiating controls within the HTML View. Notice that Text, Items, and SelectedItem are not binding handlers, but are View Model properties of the respective control.

<div data-dyn-role="Label"

data-dyn-bind="Text: 'Hello World!'" ></div>

<div data-dyn-role="ComboBox"

data-dyn-bind="Items: $data.Cities, SelectedItem: $data.Cities[0]" ></div>

#### List of pre-loaded controls

Each pre-loaded control can be added to an HTML Control using the syntax described above. Alphabetically, the available controlare:

AnchorButton

AppBar

AppBarSection

AppBarTab

Button

ButtonGroup

CheckBox

ComboBox

Date

DateTime

Dialog

DropDialogButton

DropList

FilterPane

Form

Group

HtmlLabel

Image

Input

Integer

Label

LightBox

List

ListBox

Lookup

LookupPopup

MenuButton

MenuItem

MessageBar

MessageBox

MultilineInput

Number

PanoramaItem

Panorama

Pivot

PivotItem

Popup

ProgressBar

RadioButton

Section

SectionPage

SectionPageHeader

SectionSummaryField

Tile

Time

UserButton

VerticalTabs

Note that some controls may only be instantiated as a child of another control. ActionPaneTab, PanoramaItem, PivotItem and SectionItem fall into this category. They may only be instantiated within their similarly-name parent control (e.g. ActionPane, Panorama, Pivot, Section, etc.).

There is an exhaustive example of using each of these controls in control composition available with your deployment. You can navigate to the following URL and inspect the source of the page to see examples of using control composition with each pre-loaded control.   
<https://usncax1aos.cloud.onebox.dynamics.com/ControlGallery.htm>

We will continue with the Simple Calculator and provide an example of using control composition.

#### Binding to other controls

Below is a continuation of the Simple Calculator example modified to use Framework Controls instead of standard HTML elements. By including Framework Controls in the Simple Calculator we can adhere to the styling guidelines set by UX and get some accessibility and localization features for free.

1. Ensure the View Model to contain an Expression property and a Solution property.

self.Expression = $dyn.observable("1+1");

self.Solution = $dyn.computed(function () {

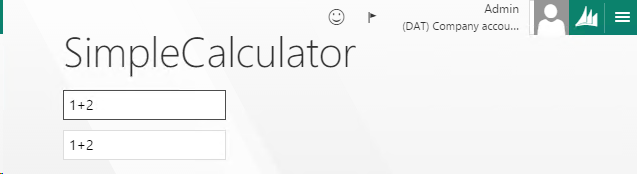
return $dyn.value(self.Expression);

});

1. Update the View to contain an Input control to allow user specified expressions, and a read-only Input control to display a calculated solution.

<div data-dyn-role="Input"   
 data-dyn-bind="Value: $data.Expression"></div>

<div data-dyn-role="Input"   
 data-dyn-bind="Value: $data.Solution, AllowEdit: false"></div>

1. Save changes. Build the **Rainier Project**. Then run the form.  
   

In this example, the Value and Text View Model properties for the Input and Label control are handling the updating of their respective control definitions.

This concludes the basic concepts of developing the HTML View for a control. Using the concepts discussed in data binding, observability, and control composition a new control author should be able to tackle all but the most complex control scenarios.

### Advanced concepts

The following topics and examples diverge from the primary end-to-end guide for building an extensible control. They are optional topics for further understanding of the control framework features. If you are reading through this guide for the first time, please skip directly to the section [Building the logical control](#_Building_the_logical).

#### Multi-Template Resource Bundles

Some controls readily fit into a componentized design, wherein the control contains reusable HTML elements. For example, imagine a control that features multiple near-identical UI elements, which differ only in the data they display or the logic they trigger. Rather than duplicate the HTML that composes each of the multiple UI elements inside of the control definition, a developer can choose to place the duplicated HTML into a separate reusable template. The parent control can then use just a single line of HTML to include each reusable UI element, supplying a different data context each time if necessary.

Template Attribute Syntax

<div data-dyn-template="templateId: [templateID], resource: [path to resource bundle]"></div>

##### BBall Starters example

See the example below, which reuses HTML markup representing a single basketball player to render a control that shows the 5 players on the starting lineup for a basketball team.

<div id="BBallStarters">

<div class="pointguard">

<div data-dyn-bind="text: $data.PointGuard.PlayerName" ></div>

<div data-dyn-bind="text: $data.PointGuard.Number" ></div>

<div data-dyn-bind="text: $data.PointGuard.Position" ></div>

<div data-dyn-bind="text: $data.PointGuard.Height" ></div>

</div>

<div class="shootingguard">

<div data-dyn-bind="text: $data.ShootingGuard.PlayerName" ></div>

<div data-dyn-bind="text: $data.ShootingGuard.Number" ></div>

<div data-dyn-bind="text: $data.ShootingGuard.Position" ></div>

<div data-dyn-bind="text: $data.ShootingGuard.Height" ></div>

</div>

<div class="smallforward">

<div data-dyn-bind="text: $data.SmallForward.PlayerName" ></div>

<div data-dyn-bind="text: $data.SmallForward.Number" ></div>

<div data-dyn-bind="text: $data.SmallForward.Position" ></div>

<div data-dyn-bind="text: $data.SmallForward.Height" ></div>

</div>

<div class="powerforward">

<div data-dyn-bind="text: $data.PowerForward.PlayerName" ></div>

<div data-dyn-bind="text: $data.PowerForward.Number" ></div>

<div data-dyn-bind="text: $data.PowerForward.Position" ></div>

<div data-dyn-bind="text: $data.PowerForward.Height" ></div>

</div>

<div class="center">

<div data-dyn-bind="text: $data.Center.PlayerName" ></div>

<div data-dyn-bind="text: $data.Center.Number" ></div>

<div data-dyn-bind="text: $data.Center.Position" ></div>

<div data-dyn-bind="text: $data.Center.Height" ></div>

</div>

</div>

We can move the repetitive bindings to each player's information into a reusable template, then use the Template attribute to load the reusable template for each player position, reducing the six lines of HTML required for each player to just one.

<div id="BBallStarters">

<div data-dyn-template="templateId: 'BBallPlayer'" data-dyn-context="$data.PointGuard"></div>

<div data-dyn-template="templateId: 'BBallPlayer'" data-dyn-context="$data.ShootingGuard"></div>

<div data-dyn-template="templateId: 'BBallPlayer'" data-dyn-context="$data.SmallForward"></div>

<div data-dyn-template="templateId: 'BBallPlayer'" data-dyn-context="$data.PowerForward"></div>

<div data-dyn-template="templateId: 'BBallPlayer'" data-dyn-context="$data.Center"></div>

</div>

<div id="BBallPlayer">

<div data-dyn-bind="text: $data.PlayerName" ></div>

<div data-dyn-bind="text: $data.Number" ></div>

<div data-dyn-bind="text: $data.Position" ></div>

<div data-dyn-bind="text: $data.Height" ></div>

</div>

Notice that we use the context attribute to ensure the correct data binding context is set on the BBallPlayer template. This example can be further optimized by using the [foreach binding handler.](#_List_of_Binding_2)

<div id="BBallStarters">

<div data-dyn-bind="foreach: $data.Players">

<div data-dyn-template="templateId: 'BBallPlayer'"></div>

</div>

</div>

<div id="BBallPlayer">

<div data-dyn-bind="text: $data.PlayerName" ></div>

<div data-dyn-bind="text: $data.Number" ></div>

<div data-dyn-bind="text: $data.Position" ></div>

<div data-dyn-bind="text: $data.Height" ></div>

</div>

Notice that the data binding context is not being explicitly set on the element with the Template attribute. This is because the foreach binding iterates through the array that it is supplied ($data.Players in this case), and implicitly sets the data binding context on the elements to the current object in the array. In this case, each iteration of the loop will result in a player being set as the data context.

The benefit of this approach is that the control can now be designed as agnostic to the number of players and the sport they are playing! It only assumes that each player has a name, number, position and height. Also, other controls can now take advantage of this "starting lineup" using the Template attribute. For example, let's say that we have a control that shows not only the starting lineup of a sports team, but also some basic information and statistics about the team as a whole.

<div id="TeamStats">

<div data-dyn-bind="text: "Team Name: " + $data.TeamName"></div>

<div data-dyn-bind="text: "Win-Loss: " + $data.SeasonRecord"></div>

<div data-dyn-bind="text: "Points/Game: " + $data.PPG"></div>

<div data-dyn-bind="text: "Coach: " + $data.CoachName"></div>

<div data-dyn-template="templateId:'BBallStarters', resource:'BBallStarters'"   
 data-dyn-context="$data.Starters"></div>

</div>

We simply pass the list of Starters to the BBallStarters template and let it handle all of the rendering from there.

# Building the logical control

## X++ Runtime Class

The X++ Runtime Class contains the X++ code that runs when a control has been instantiated on a Form. The X++ Runtime Class is opposed to the X++ Build Class which contains X++ code that is consumed only at design-time by the Form Designer. The main purpose of the X++ Runtime class is to provide a set of server-side properties and commands for the physical control to interact with, and provide an interface for the control to utilize other X++ artifacts (tables, classes, menu items, enums, etc.).

### Creating the Runtime Class

The Runtime classes for controls must extend the FormTemplateControl X++ class. The FormTemplateControl class contains several framework mechanisms that are used in managing control properties, commands, data bindings and more. The following outline summarizes the steps a developer must take to complete the X++ Runtime Class for a control.

1. Create an X++ Class that extends FormTemplateControl and supply the [FormControlAttribute](#_FormControlAttribute) to the class declaration
2. For each desired runtime property, create a backing [FormProperty](#_FormProperty) and associated X++ getter/setter method, then decorate the method with the [FormPropertyAttribute](#_FormPropertyAttribute_Usage)
3. Initialize each FormProperty in the [New method](#_Overriding_the_New()_1)
4. In the [ApplyBuild method](#_Overriding_the_ApplyBuild()) supply each property with a value or a data binding, based on values supplied to the [X++ Build Class](#_Creating_X++_Properties) through the Form Designer

We will describe each of these steps in detail as we progress through implementing the Simple Calculator example.

#### Class Declaration

The following is the declaration for the X++ Runtime Class of the Simple Calculator.

[FormControlAttribute("SimpleCalculator","SimpleCalculator",classstr(BuildSimpleCalculator))]

class SimpleCalculator extends FormTemplateControl

##### FormControlAttribute

The FormControlAttribute distinguishes X++ classes that represent controls from other X++ classes. Supplying an X++ class with the FormControlAttribute, and providing the required attribute arguments, will result in a new item appearing in the list of controls that can be added to a form in the Form Designer. The name that appears in the list for the new control is defined by the [FormDesignControlAttribute](#_Creating_the_Build) which is placed on the X++ Build Class associated with the control.

The View for the control, the [templateI](#_Overriding_the_New()_1)D, is supplied as the first argument to the FormControlAttribute. When the control is instantiated by the server form runtime, the framework reads this argument and sets the templateID property on the X++ class (this property is inherited from FormTemplateControl). Then the client control framework reads this property to locate the appropriate template for the physical control. The second argument is the name of resource bundle that contains the template. Currently this attribute is not being read. Instead, you should manually set the resource bundle name inside of the [**new** method](#_Overriding_the_New()_1).

#### Creating Properties

There are 4 primary steps necessary to add create a property that is shared between the logical and physical control.

1. Create a backing FormProperty
2. Initialize the FormProperty
3. Create an X++ getter/setter
4. Supply the FormPropertyAttribute to the getter/setter

##### Creating the FormProperty

The FormProperty is the object the Control Extensibility Framework uses to track and propagate property value changes between the physical and logical property. The FormProperty contains methods for setting and getting the current value of the property. Let’s add a FormProperty to the Simple Calculator.

1. Open the SimpleCalculator X++ class in the code editor.
2. Add a backing FormProperty field for the Expression property to the class, as highlighted below.

[FormControlAttribute("SimpleCalculator","SimpleCalculator",classstr(BuildSimpleCalculator))]

class SimpleCalculator extends FormTemplateControl

{

FormProperty expressionProperty;

public void new(FormBuildControl \_build, FormRun \_formRun)

{

super(\_build, \_formRun);

this.setResourceBundleName("/resources/html/SimpleCalculator");

}

}

##### Initializing the FormProperty

The **new** method for each X++ Runtime Class should be used to initialize FormProperties. Use the following pattern for initializing all FormProperties.

1. Add the highlighted FormProperty initialization code to the **new** method of the Simple Control X++ class

public void new(FormBuildControl \_build, FormRun \_formRun)

{

super(\_build, \_formRun);

this.setResourceBundleName("/resources/html/SimpleCalculator");

expressionProperty = this.addProperty(methodstr(SimpleCalculator, expression), Types::String);

}

The **addProperty** method is inherited from FormTemplateControl, and it determines the backing X++ getter/setter for a property, as well as the type. These argument should correspond with the method name and return/argument type of the getter/setter method created for the property.

##### Creating the X++ getter/setter

Next we add an X++ method to act as a getter/setter. The desired type of the property should be the same as the type the method returns. In this case, our Expression is string typed.

1. Add the following “expression” getter/setter to the class.

public str expression(str expression = expressionProperty.parmValue())

{

if(!prmIsDefault(expression))

{

expressionProperty.setValueOrBinding(expression);

}

return expressionProperty.parmValue();

}

This is the standard code snippet used for almost all FormProperty getter/setters. The return/argument type may vary, as well as the name of the method and FormProperty, but this snippet can be considered “boilerplate” code for creating FormProperty getter/setters.

###### X++ getter/setter return type

The X++ getter/setter of a FormProperty must accept/return one of the X++ types supported by the FormProperty serializer. The full list of supported types are below:

* Date
* Enum
* Guid
* Int64
* Integer
* Real
* String
* VarString
* Time
* UtcDateTime
* An X++ data contract

In order to use an EDT with a FormProperty, use the **base type** of the EDT, which should be one of the types listed above.

##### Supplying the FormPropertyAttribute

The FormPropertyAttribute is used to define and configure the physical counterparts of the logical property. The FormPropertyAttribute is supplied to the X++ getter/setter for the logical property.

[FormPropertyAttribute([FormPropertyKind], [target name], [read only],[auto post back])]

###### FormPropertyKind

The FormPropertyKind identifies a physical property as either a value property or a reference property. The enumerated options are:

* FormPropertyKind::Value – For value properties
  + **Value** properties work as you might expect, the value of the property is shared between the logical and physical properties. The values are synchronized by the framework according to the rules defined by the Interaction Service.
* FormPropertyKind::BindableValue – For properties data bound to a data source
  + **BindableValue** or **Reference** properties should not be read from X++. This is because the X++ value of the FormProperty will be an instance of a FormBinding, which contains only a reference to a particular data source/data field pair. The value contained in the data field is resolved in the client and provided to the physical property, so reading this value is only possible from the physical property. Also, writing a value to the physical property will write the value to the referenced data field in the client’s copy of the data source, and will not result in an update to the FormProperty accessible in X++. More information on using **BindableValue** or **Reference** properties is described in the Simple Calculator example.

###### Target Name

The TargetName determines the desired name of the physical property, as it will appear in the JavaScript View Model. If, for example, the TargetName is set to "Foo" then the property will be accessible in the View Model as “self.Foo”.

###### Read Only (optional)

This flag determines if the physical property will be editable or not from the JavaScript View Model. The default is *false*, meaning the physical property is readable and writeable. The ReadOnly value **does not** apply to access level of the logical property accessible from X++, it only prevent changing the value via the physical property in the JavaScript View Model. To design a property as read-only from X++, define the getter/setter method signature such that it does not accept any arguments.

###### Auto Post-back (optional)

When true, this determines that a value change to the physical property should be propagated to the logical property at the next available moment, as determined by the framework. When false (default), the physical property value change event is added to the Interaction Queue and waits in the queue until the framework flushes the queue to the server. The Interaction Queue stores all state changes in the client across all controls and data sources, and flushes the queue to the server when an Interaction is queued that has the auto post-back (also named “execute immediate”) flag set to true. An example of such an interaction with the auto post-back flag set to true is the SetFocus command. After a SetFocus command is queued, the Interaction Queue will flush all interactions currently in the queue to the server, and the interactions are processed sequentially on the server.   
  
The default behavior (auto post-back = false) should be used for almost all control scenarios. Choosing to queue property changes whenever possible is one of the most important decisions a developer can make to maintain the performance and fluidity of the client.

Let’s add the FormPropertyAttribute to the expression method in the Simple Calculator.

1. Add the FormPropertyAttribute to the Expression property as highlighted below

[FormPropertyAttribute(FormPropertyKind::Value, "Expression")]

public str expression(str expression = expressionProperty.parmValue())

{

if(!prmIsDefault(expression))

{

expressionProperty.setValueOrBinding(expression);

}

return expressionProperty.parmValue();

}

Now that we have added the Expression property to the Runtime class, there is no longer a need to explicitly define the Expression property in the JavaScript View Model. The framework will now handle the job of creating a physical version of the Expression property and making it available in the View model automatically. It is important to note that all physical properties generated from the Runtime class using this method are created as **observable** properties in the View Model.

1. Remove the definition of the Expression property from the View Model. Below, we have commented out the code.

(function () {

'use strict';

$dyn.ui.defaults.SimpleCalculator = {

};

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

//self.Expression = $dyn.observable("1+1");

self.Solution = $dyn.computed(function () {

return $dyn.value(self.Expression);

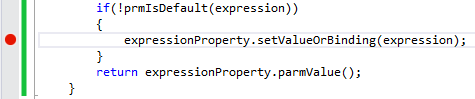
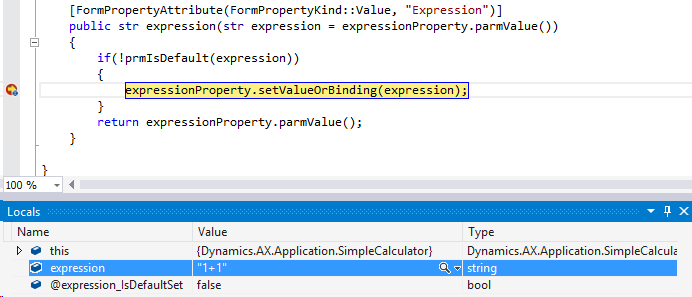
});

};

$dyn.controls.SimpleCalculator.prototype = $dyn.extendPrototype($dyn.controls.Input.prototype, {

});

})();

1. Set a break point inside of the property getter/setter for the logical Expression property  
   
2. Save the changes. Build the **Rainier Project**. Run the form with debugging on by pressing **F5** or selecting **DEBUG > Start Debugging**
3. When the form appears, type “1+1” into the input box. Then press TAB.   
     
   Pressing tab causes the focus to shift from the Input inside of the Simple Calculator, to the form. The focus change does two things.  
   1. It updates the Expression property. This is because the Input control is observing the focus of the text box. When the text box loses focus, the Input control reads the text in the text box and updates the Expression property accordingly. The property value change event for the Expression property is then queued in the Interaction Queue, and waits.
   2. It causes the focus to shift from the Input control inside of the Simple Calculator control, to the Form. This focus change event is queued in the Interaction Queue. Since the focus change event is set to “execute immediate” the framework flushes the Interaction Queue to the server, and our property change event is processed in our X++ class.
4. The X++ breakpoint is hit, and you can inspect the value of the expression using the Locals view  
   

### Properties versus Commands

We have seen how a basic property can be shared between the physical control and logical control. However, **for the vast majority of control scenarios writable properties are not suitable**, because a property value needs to undergo validation before it is acceptable as a valid value. For these scenarios we use commands to validate a value before updating the property. The browser client can produce invalid values for properties for many reasons, including malicious one. For this reason property validation logic resides on the server in the X++ class for the control.

### Creating Commands

Commands are used to expose X++ methods to the JavaScript View Model. These X++ methods are not property getters/setters and don’t use FormProperties to propagate state changes between the physical and logical control.

#### Supplying the FormCommandAttribute

For an X++ method to be accessible from the View Model via a command, the X+ method must be supplied the FormCommandAttribute. The attribute method signature is as follows.

[FormCommandAttribute([target name], [execute immediate], [target type name],[is internal])]

##### Target name

This argument is used similarly to the target name argument for the FormPropertyAttribute. It determines the name of the physical command, as it will be accessible from the JavaScript View Model.

##### Execute Immediate (optional)

This argument operates similarly to the Auto post-back argument of the FormPropertyAttribute. When this argument is set to its default behavior of true, calling the command from the JavaScript View Model will result in the Interaction Queue being flushed to the server, and all Interactions in the queue will be processed sequentially, until finally the command is executed and its backing X++ method is called.

The remaining two arguments are optional, and only to be used in very specific advanced scenarios, that are outside the scope of this guide.

##### Supplying the FormCommandAttribute

Let’s add a command to the Simple Calculator control for example.

1. Open the SimpleCalculator X++ class
2. Add the following methods to the SimpleCalculator class

public void setExpression(str exp = "")

{

*// Sanitize the expression*

exp = this.sanitize(exp);

*// Set the intnernal expression property*

this.expression(exp);

}

public str sanitize(str exp)

{

*// Remove any non-operators or non-numbers from the expression*

exp = System.Text.RegularExpressions.Regex::Replace(exp, @'[^-()\d/\*+.]','');

*// Replace any parentheses not followed by a number from the expression*

exp = System.Text.RegularExpressions.Regex::Replace(exp, @'[(](?=[^\d])','');

*// Remove any operators that are followed by an operator from the expression*

exp = System.Text.RegularExpressions.Regex::Replace(exp, @'[-/\*+.](?=[-/\*+.])','');

*// Remove operators from the end of the expression*

exp = System.Text.RegularExpressions.Regex::Replace(exp, @'[-(/\*+.=]+$','');

*// Remove any remaining empty parentheses from the expression*

exp = System.Text.RegularExpressions.Regex::Replace(exp, @'[(](?=[)])','');

*// Replace parenthetical multiplication with explicit multiplication in the expression*

exp = System.Text.RegularExpressions.Regex::Replace(exp, @'[)](?=[(])',')\*');

return exp;

}

1. Add the FormCommandAttribute to the setExpression method as highlighted below

[FormCommandAttribute("SetExpression")]

public void setExpression(str exp = "")

1. Make the physical Expression property read-only buy setting the Read only flag to true on the FormPropertyAttribute, as highlighted below.  
     
   Now the expression can only be set from within X++.

[FormPropertyAttribute(FormPropertyKind::Value, "Expression", true)]

public str expression(str expression = expressionProperty.parmValue())

1. Next we need to call the SetExpression command from the View Model when the user enters the Expression. Add the highlighted code to the JavaScript View Model constructor inside SimpleCalculator.js.  
     
   The $dyn.callFunction method is used to execute commands. The arguments to $dyn.callFunction are: the command to execute; the object containing the command, the parameterized arguments to the command, and an (optional) callback function. The parameterized arguments should be named according to their X++ method argument names.

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

self.Solution = $dyn.computed(function () {

return $dyn.value(self.Expression);

});

$dyn.observe(self.Expression, function (expression) {

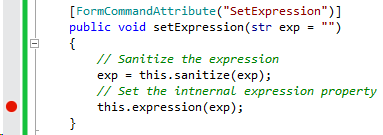
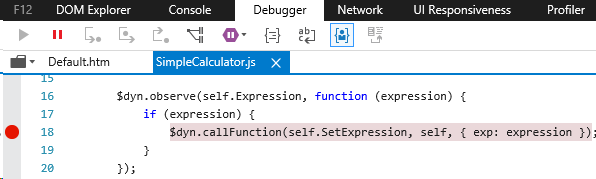
if (expression) {

$dyn.callFunction(self.SetExpression, self, { exp: expression });

}

});

};

1. Set a breakpoint in the setExpression X++ method.   
   
2. Save the changes. Build the **Rainier Project**. Then press **F5** or click **DEBUG > Start Debugging** to run the form.
3. In the browser, set a breakpoint in the View Model where the SetExpression command is called.  
   
4. Type “1+1” into the Input text box for the Simple Calculator, then press Tab.
5. Since the View Model is observing the Expression property, and the expression is not empty, the SetExpression command is being fired with the value of the Expression passed in the parameterized object argument. Click **Continue** in the browser’s debugger.
6. Next, the X++ breakpoint for the setExpressoin method is hit. Check the Locals to see that the “exp” argument is “1+1”. Click **Continue** in the Visual Studio debugger.

This completes the basics of using properties and commands with the X++ Runtime class. For the sake of completeness, let’s continue the Simple Calculator example and add the calculator’s solving functionality. We will introduce two new concepts while adding this functionality. We will show how to use buttons with commands, and how to property use labels with controls.

#### Using buttons with commands

1. Add an Evaluate command to the runtime class  
     
   The Product Configuration Math Evaluator is used here only as an example. It provides the ability to parse and evaluate a mathematical expression from an input string. Note that Evaluate returns a real, and not a string.

[FormCommandAttribute("Evaluate")]

public real evaluateExpression()

{

return Microsoft.Dynamics.Ax.Frameworks.Controls.ProductConfiguration.MathEvaluator::Evaluate(expressionProperty.parmValue());

}

1. Open the SimpleCalculator.js in Visual Studio
2. Update the JavaScript with the following code.  
     
   The JavaScript above makes the following changes

(function () {

'use strict';

Globalize.addCultureInfo("en", {

messages: {

SimpleCalculatorExpression: 'Enter an expression',

SimpleCalculatorSolution: "=",

SimpleCalculatorSolveButton: "Solve"

},

});

$dyn.ui.defaults.SimpleCalculator = {

};

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

self.Solution = $dyn.observable("");

$dyn.observe(self.Expression, function (expression) {

if (expression) {

var ret = $dyn.callFunction(self.SetExpression, self,

{

exp: expression

});

}

});

self.Solve = function () {

$dyn.callFunction(self.Evaluate, self, {}, function (solution) { self.Solution(solution);

});

}

};

$dyn.controls.SimpleCalculator.prototype = $dyn.extendPrototype($dyn.controls.Input.prototype,

{});

})();

* + Adds several localizable messages to Globalize, for the English language.
  + Creates a local observable names Solution. Note that Solution is not a property as it has no logical component.
  + Adds a Solve function to the View Model, which calls the Evaluate command defined in X++. When the Evaluate command returns from the server, the local Solution observable is updated.

1. Open the SimpleCalculator.htm in Visual Studio
2. Update the SimpleCalculator.htm with the following HTML.   
     
   The HTML code above does the following things

<script src="/resources/scripts/SimpleCalculator.js"></script>

<link rel="stylesheet" src="/resources/scripts/SimpleCalculator.css"></link>

<div id="SimpleCalculator">

<div data-dyn-role="Input"

data-dyn-bind="

Value: $data.Expression,

Label: $label('SimpleCalculatorExpression')"></div>

<div data-dyn-role="Input"

data-dyn-bind="

Value: $data.Solution,

AllowEdit: false,

Label: $label('SimpleCalculatorSolution')"></div>

<div data-dyn-role="Toolbar">

<div data-dyn-role="ToolbarGroup">

<button data-dyn-role="Button"

data-dyn-bind="

Label: $label('SimpleCalculatorSolveButton'),

Click: $data.Solve,

ImageType: 'Symbol',

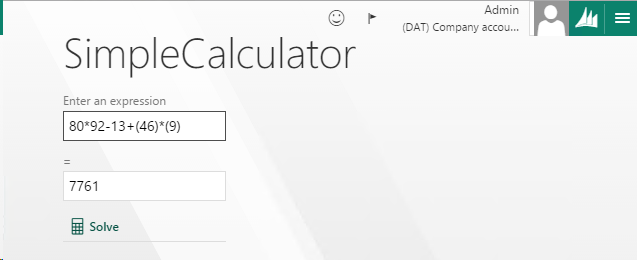
ImageName: 'Calculate'"></button>

</div>

</div>

</div>

* + Binds the labels of both Input controls. Each label references one of the localizable messages defined in the JavaScript file
  + Adds a toolbar to the Simple Calculator, adds a toolbar group to the toolbar, and adds a button to the toolbar group.
  + Binds the label on the button to the localizable message we defined in the JavaScript.
  + Binds the click of the button to the Evaluate command we’ve created.
  + Binds the button’s image to the calculator symbol, via the ImageType and ImageName values. ImageType of ‘Symbol’ indicates that the ImageName will be one of the predefined dynamics icons. ImageType could also be ‘URL’, in which case the ImageName must contain the URL to the image file.

1. Save the changes. Build the **Rainier Project**. Run the form with **Ctrl+F5** or click **DEBUG > Run without debugging**
2. The updated calculator is shown below  
   

## X++ **Build Class**

### Creating the Build Class

The Build Class houses design-time logic that defines the design-time experience for a control. This includes defining designer property appearance/behavior in the Visual Studio property sheet. The designer properties do not necessarily need to map one-to-one to your control’s runtime properties. You may decide what designer properties your control needs and they may map to a subset of your runtime properties, to all of the runtime properties, or not map to a runtime property at all.

### Creating Designer Properties

As with the runtime properties, each designer property is backed by an X++ method with a specialized attribute. In the case of designer properties, each method accepts as an argument and returns the type of the value to appear in the property sheet. These values are most often strings, but can be any of the primitive types, including enums.

The behavior of the designer property as it appears in the property sheet is based on the the FormDesignPropertyAttribute. A FormDesignPropertyAttribute is required for each property you wish to appear in the property sheet at design-time. The first argument to the attribute is the name that will appear beside the property in the property sheet. The second is the section of the property sheet that the property should appear under. For designer properties that specify bindings to either data sources or data fields, one or two additional attributes are needed.

#### Creating simple designer properties

1. Open the BuildSimpleCalculator X++ class
2. Add the highlighted code to the class

[FormDesignControlAttribute("Simple Calculator")]

class BuildSimpleCalculator extends FormBuildControl

{

str initialExpression;

public str initialExpression(str expression = initialExpression)

{

if(!prmIsDefault(expression))

{

initialExpression = expression;

}

return initialExpression;

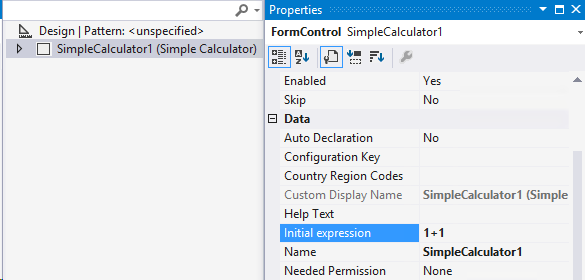
}

}

1. Add the FormDesignPropertyAttribute to the method

[FormDesignPropertyAttribute("Initial expression", "Data")]

public str initialExpression(str expression = initialExpression)

1. Save the changes. Build the **Rainier Project**.
2. Open the Simple Calculator form.
3. Right-click on the SimpleCalculator on the form, and select Properties to open the Visual Studio property sheet
4. In the property sheet find the Initial Expression designer property we have just added. Note that it was placed under the “Data” section of the properties. If instead of Data we supplied a section name that did not already exist, a section with that name would have been created and the designer property would have been added there.
5. Enter “1+1” into the Initial expression designer property  
   
6. Save the changes.

Next, let’s wire up the Initial Expression property to the runtime class.

1. Open the SimpleCalulator X++ class
2. Add the following method beneath the **new** method in the SimpleCalculator class  
     
   This overrides the **applyBuild** method inherited from the FormControl base class. In the lifecycle of a control, **applyBuild** is called during form initialization (during **init**, before **run**). This is the opportunity for a control to initialize itself based on information known at design time. In our case we simply initialize the expression runtime property based on the initialExpression designer property.

public void applyBuild()

{

BuildSimpleCalculator build;

super();

build = this.build();

if(build)

{

this.expression(build.initialExpression());

}

}

1. Set a breakpoint on the line that sets the expression.
2. Save the changes. Build the **Rainier Project**. Run the form with **F5** or by clicking **DEBUG > Start debugging**
3. Note that the breakpoint is hit during initialization of the form. When the form appears, “1+1” has been prepopulated in the Expression Input field.

### Binding to data sources

In order to bind a property to a data field from a data source, the data source and field must be specified. In most cases the data source and data field are specified in the property sheet for the control. After the data source(s) and data field(s) are specified, the control needs to bind a runtime property to the specified data field on the specified data source. The following steps show how to add these properties to the property sheet to allow for data source/field selection, and how to bind a runtime property to a data field.

#### Creating designer properties

The FormDesignPropertyDataSourceAttribute creates designer properties that lists the available data sources on the form.

1. Add a designer property to the BuildSimpleCalculator class, and supply the FormDesignPropertyDataSourceAttribute, as done in the highlighted code.  
     
   Be sure to note the FormDesignPropertyDataSourceAttribute that was added to the historyDataSource method.

[FormDesignControlAttribute("Simple Calculator")]

class BuildSimpleCalculator extends FormBuildControl

{

str initialExpression;

str historyDataSource;

[FormDesignPropertyAttribute("Initial expression", "Data")]

public str initialExpression(str expression = initialExpression)

{

if(!prmIsDefault(expression))

{

initialExpression = expression;

}

return initialExpression;

}

[FormDesignPropertyAttribute("History data source", "History"),

FormDesignPropertyDataSourceAttribute]

public str historyDataSource(str dataSource = historyDataSource)

{

if(!prmisDefault(dataSource))

{

historyDataSource = dataSource;

}

return historyDataSource;

}

}

1. Add a designer property for the data fields as well.  
     
   Note that the FormDesignPropertyDataFieldAttribute needs the data source method to be supplied, so that the property sheet knows which data source’s fields to show.

[FormDesignControlAttribute("Simple Calculator")]

class BuildSimpleCalculator extends FormBuildControl

{

str initialExpression;

str historyDataSource;

str expressionDataField;

str solutionDataField;

[FormDesignPropertyAttribute("Initial expression", "Data")]

public str initialExpression(str expression = initialExpression)

{

if(!prmIsDefault(expression))

{

initialExpression = expression;

}

return initialExpression;

}

[FormDesignPropertyAttribute("History data source", "Data"),

FormDesignPropertyDataSourceAttribute]

public str historyDataSource(str dataSource = historyDataSource)

{

if(!prmisDefault(dataSource))

{

historyDataSource = dataSource;

}

return historyDataSource;

}

[FormDesignPropertyAttribute("Expression data field", "History"),

FormDesignPropertyDataFieldAttribute(methodstr(BuildSimpleCalculator, historyDataSource))]

public str expressionDataField(str dataField = expressionDataField)

{

if(!prmisDefault(dataField))

{

expressionDataField = dataField;

}

return expressionDataField;

}

[FormDesignPropertyAttribute("Solution data field", "History"),

FormDesignPropertyDataFieldAttribute(methodstr(BuildSimpleCalculator, historyDataSource))]

public str solutionDataField(str dataField = solutionDataField)

{

if(!prmisDefault(dataField))

{

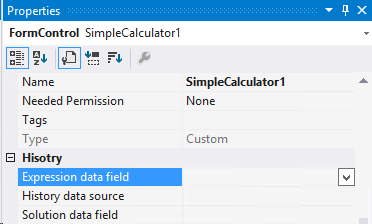
solutionDataField = dataField;

}

return solutionDataField;

}

}

1. Save the changes. Build the **Rainier Project**.
2. Open the **SimpleCalculator** form.
   * You should get a prompt informing you that the SimpleCalculator control definition has changed. This is due to the fact that we added designer properties to the control that now need to be added and stored to the form metamodel. Click **OK** on the prompt.
3. Right-click on the **SimpleCalculator** control on the form, and select properties. View the newly added properties under the **History** section.  
   

Now that we have added designer properties that allow selecting data source and data fields, we need to add a data source to the form. This data will host a temp table containing the most recently calculated expressions and their solutions.

1. Right-click on the Rainier Project and select **Add > New Item…**
2. Select Table from the list, and name the table **CalculatorHistory**.
3. Click **Add.**
4. Add a string field to the table named **Expression**
5. Add a real field to the table named **Solution**
6. Set the Table Type property on the table to **InMemory**
7. Save the changes. Build the **Rainier Project**.
8. Open the **SimpleCalculator** form and add the newly created table as a data source
9. Open the property sheet for the **SimpleCalculator** control and set the following properties
   * History data source = **CalculatorHistory**
   * Expression data field = **Expression**
   * Solution data field = **Solution**

Now that we have captured the data source and data fields for the SimpleCalculator, we need to establish runtime **reference** properties to bind to these data fields.

#### Creating runtime properties

1. Open the **SimpleCalculator** runtime class
2. Add a FormProperty field named **Solution**
3. Initialize the **Solution** form property as shown below

FormProperty expressionProperty;

FormProperty solutionProperty;

public void new(FormBuildControl \_build, FormRun \_formRun)

{

super(\_build, \_formRun);

this.setResourceBundleName("/resources/html/SimpleCalculator");

expressionProperty = this.addProperty(methodstr(SimpleCalculator, expression), Types::String);

solutionProperty = this.addProperty(methodstr(SimpleCalculator, solution), Types::Real);

}

1. Add a getter/setter method for the Solution property, and update the getter/setter for the Expression property as shown below.   
     
   Note that the **FormPropertyKind** is now set to **BindableValue** for both properties. This indicates that the properties will not be **reference** properties. Also note that the properties now accept **anyType** as the argument type. This is due to the fact that reference properties are initialized with instances of **FormDataFieldBinding** or **FormDataMethodBinding**, instead of with values like str or real.

[FormPropertyAttribute(FormPropertyKind::BindableValue, "Expression", true)]

public str expression(anytype expression = expressionProperty.parmValue())

{

if(!prmIsDefault(expression))

{

expressionProperty.setValueOrBinding(expression);

}

return expressionProperty.parmValue();

}

[FormPropertyAttribute(FormPropertyKind::BindableValue, "Solution", true)]

public real solution(anytype solution = solutionProperty.parmValue())

{

if(!prmIsDefault(solution))

{

solutionProperty.setValueOrBinding(solution);

}

return solutionProperty.parmValue();

}

The next step is to initialize the **Solution** and **Expression** properties with bindings based on the data source and data fields specified in by the designer properties in the X++ Build Class.

1. Update the **applyBuild** method with the highlighted code.   
     
   The code above sets the **Expression** and **Solution** properties to instances of **FormBindingDataField**. **FormBindingDataField** is simple object that contains a reference to a specified data field on a specified data source. It needs the formRun instance in order to access the data source specified and bind to it. You can look up the class definition of FormBindingUtil if you are curious about its inner workings.

public void applyBuild()

{

BuildSimpleCalculator build;

super();

build = this.build();

if(build)

{

this.expression(FormBindingUtil::initBinding(

build.historyDataSource(),

build.expressionDataField(), this.formRun()));

this.solution(FormBindingUtil::initBinding(

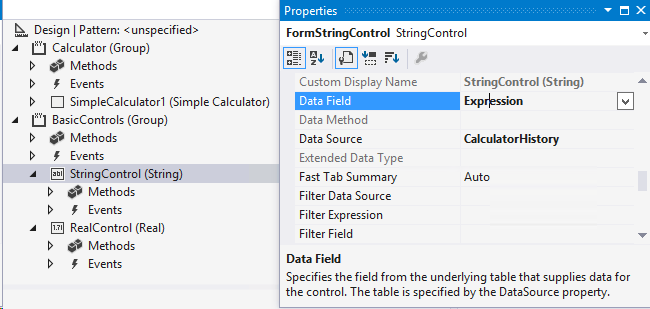
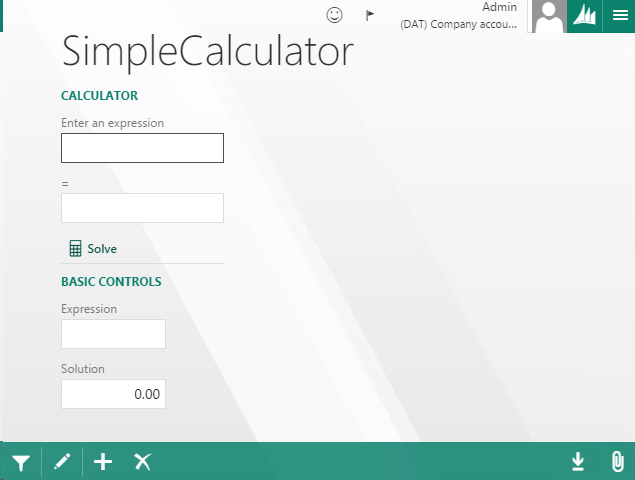
build.historyDataSource(),

build.solutionDataField(), this.formRun()));

}

}

We have finished binding the SimpleCalculator to a couple of data fields. Next we will add a few simple controls to the form to verify that the SimpleCalculator can indeed read from, and write to the data source.

1. Add a string and real control to the form and bind them to the appropriate data source fieldsWe have also added a few group controls to organize the flow of the form.  
   
2. Save the changes. Build the **Rainier Project**. Run the form using **Ctrl+F5** or click **DEBUG > Run without debugging**The form launches in **View** mode, but the SimpleCalculator still allows editing, while the basic controls do not. This is due to the fact that our SimpleCalculator is not yet wired to respet View modes.  
   Also note that the Initial Expression is no longer showing in the calculator. This is because we are no longer using that designer property. Setting default values for data bound properties does not happen at the control level, but at the form, data source or table level.
3. Type “1+1” into the expression field. Then press Tab
4. Note that the Expression field under Basic Controls updates to show the new value. This verifies that our Expression property is bound to the data source correctly.
5. Click **Solve** on the calculator.   
   Note that the calculator shows the solution, but the basic control does not. This is because the Solution Input control that the calculator is showing is not bound to the Solution reference property we just added. We have code in the View Model that is replacing the Solution reference property generated from our Runtime class.
6. Open the SimpleCalculator.js
7. Remove the following highlighted code. We have commented it out below.  
     
   The Solution property will now be accessible in the same manner as the Expression property from within the View Model. The read-only Input control in the HTML will now be bound to the Solution reference property, and our Solve function in the View Model will update the Solution based on the Evaluate command return value from the server.

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

//self.Solution = $dyn.observable("");

$dyn.observe(self.Expression, function (expression) {

if (expression) {

var ret = $dyn.callFunction(self.SetExpression, self, { exp: expression });

}

});

self.Solve = function () {

$dyn.callFunction(self.Evaluate, self, {}, function (solution) {

self.Solution(solution);

});

}

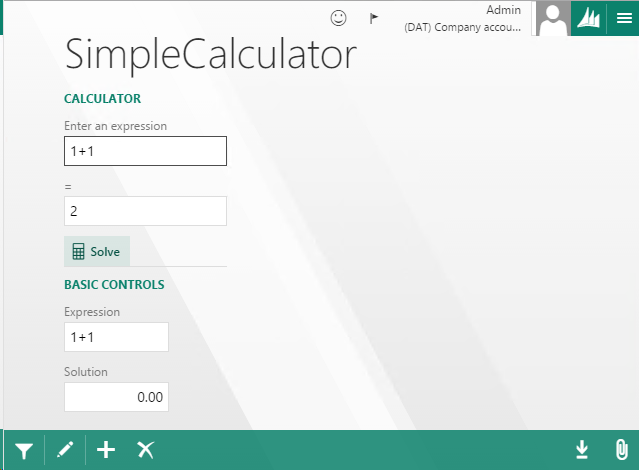
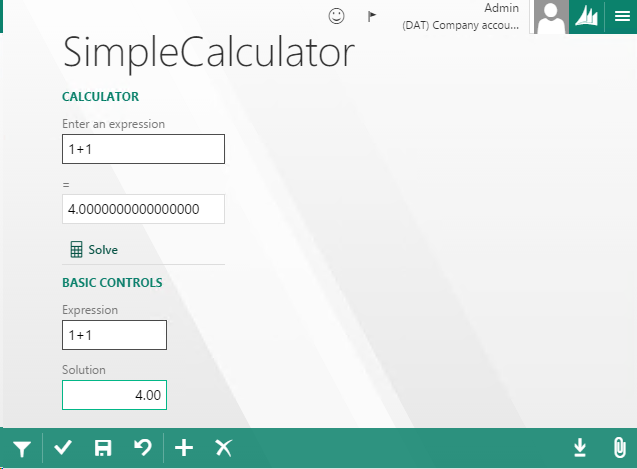
};

1. Save the changes. Build the **Rainier Project**. Run the form without debugging.  
   You may notice that the Solution field in the calculator now shows “0.0000….”. If it seems as though the changes you made to the HTML or JavaScript were not recognized, you can try clearing the browsers cache and reloading the form.

We’ve now covered the basics of data binding. However we will go into more depth describing how the framework synchronizes values across data bound properties.

#### Understanding how data binding

Following the simple calculator control example, let’s try to use the **Solve** capability of the control.

1. Enter “1+1” into the Expression of the simple calculator. Then press Tab
2. Click the **Solve** button.   
     
   Note that the Solution field in the calculator updates to “2”, but the Solution field under basic controls does not.
3. Click the **Edit** button on the form
4. Type “4” into the Solution field under Basic Controls. Then press Tab.  
     
   Note that the Solution field in the calculator updates to show “4”.

The behavior we see here is that the Solution property in the calculator doesn’t update the data field when the Solution value changes, however when the value for Solution control under the Basic Controls changes it correctly updates the data field, and causes the calculator’s Solution property to update as well.   
  
This behavior is due to the fact that all reference properties maintain a one-way binding to the data source in the client. The Solution property in the SimpleCalculator, and an internal property of the Solution control under Basic Controls, are both observing the same data field for changes, but they cannot directly set the value on the data field from the client. The Solution control handles writing to the data field by sending the new value to the logical Solution control, where the new value undergoes validation before being set on the data field on the server. When the server finishes processing this interaction, the client receives the new value for the data field. The client then updates its copy of the data field, and all observers of the data field are notified of the change. The Solution property in SimpleCalculator and the Solution control both update their values as a result.

The reason that the Expression property seems to work correctly is because we are not setting the Expression property directly in the View Model, but instead we are setting it via its X++ getter/setter from the SetExpression command.

The way to get the Solution property in the Simple Calculator to behave correctly is to set the Solution property in the same manner that the Expression is set: using the X++ getter/setter.

1. Remove the highlighted code from the View Model.   
     
   This removes the update to the Solution property in the View Model.

$dyn.controls.SimpleCalculator = function (props, element) {

var self = this;

$dyn.controls.Input.apply(this, arguments);

$dyn.ui.applyDefaults(this, props, $dyn.ui.defaults.SimpleCalculator);

$dyn.observe(self.Expression, function (expression) {

if (expression) {

var ret = $dyn.callFunction(self.SetExpression, self, { exp: expression });

}

});

self.Solve = function () {

$dyn.callFunction(self.Evaluate, self, {}, function (solution) {

self.Solution(solution);

});

}

};

1. Open the SimpleCalculator runtime class.
2. Update the **Evaluate** command’s X++ method as shown below.

[FormCommandAttribute("Evaluate")]

public void evaluateExpression()

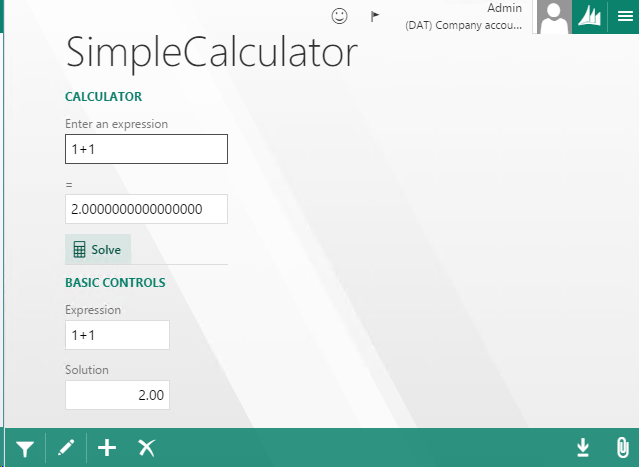
{

real sol;

sol = Microsoft.Dynamics.Ax.Frameworks.Controls.ProductConfiguration.MathEvaluator::Evaluate(expressionProperty.parmValue());

this.solution(sol);

}

1. Save the changes. Build the **Rainier Project**. Run the form without debugging.
2. Try to solve the expression “1+1”. The Solution property in the calculator should update as expected.  
   

### Advanced designer-time topics

You may create designer properties for more than data sources and data fields data fields.

#### Designer Property for Subclasses

The **FormDesignPropertyExtendsClassAttribute** displays a drop down that lists the X++ classes that extend the base class name supplied.

[FormDesignPropertyAttribute("Data", "Document Type"),

FormDesignPropertyExtendsClassAttribute(classStr("AifDocument"))]

public str documentType(str \_type = "")

#### Designer Property for Interface Implementers

The **FormDesignPropertyImplementsAttribute** displays a drop down that lists the X++ classes that implement the specified interface.

[FormDesignPropertyAttribute("Data", "Expense Type"),

FormDesignPropertyImplementsAttribute(classStr(("IPostalAddress"))]

public str addressType(str \_type = "")

#### Designer Property for AOT Artifacts

The **FormDesignPropertyReferenceAttribute** displays a deop down that lists the X++ AOT artifacts of the specified type.

[FormDesignPropertyAttribute("Data", "Temp Table"), FormDesignPropertyReferenceAttribute(FormDesignPropertyReferenceType::Table)]

public str tempTable(str \_tableName = "")

The enumerated AOT artifact options are:

* Table
* View
* Map
* EDT
* BaseEnum
* Query
* Classes
* Form
* MenuItemDisplay
* MenuItemOutput
* MenuItemAction

# Appendix

## Resource file loading

Use the following code snippet to load all of the JavaScript or CSS files required by a resource bundle. Place the code right before the *closing body tag* for the HTM file, the order of placement does not matter as all files are downloaded before code is executed. You may place as many file references as your resource bundle requires. Use a separate line for each reference.

### JavaScript

<div data-dyn-script="/resources/scripts/[JS filename].js"></div>

### CSS

<link data-dyn-css="/resources/styles/[CSS filename].css" rel="stylesheet" type="text/css" />

### HTM (Template Attribute Syntax)

The Template attribute is used when a control is componentized, and the [primary control definition](#_Control_Identifier) relies on other HTML templates to compose itself. If the control definition attempting to load a template happens to contain the template (see [Example 13](#_Multi-Template_Resource_Bundles)), then the resource bundle name parameter is optional.

<div data-dyn-template="templateID: [id of HTML element to load],

resourceBundleName: [HTM file containing the HTML element]"></div>

When you  [add a resource bundle file as an Ax Resource](#_Create_Resource_Bundle) the file will be deployed to the correct subfolder inside /resources depending on the file type. The deploy process for resources is triggered upon building the Rainier Project (if testing on a local machine), or uploading and deploying an AxPackage to a cloud CSU.

## List of Binding Handlers

This list is not exhaustive of all binding handlers. It details the most frequently used binding handlers. For an exhaustive list inspect the html-binding-handlers.js file (using the debug tools of your favorite browser).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Binding Handler | Description | Parameters | Example Syntax | HTML Output |
| visible | Shows or hides an HTML element by setting the CSS *display* style to *none* | *false* (or a false-like value: null, 0, undefined),  *true* (or a true-like value: non-null object, 1) | <div data-dyn-bind=  "visible: false"></div> |  |
| if | Conditionally removes all child HTML element from the DOM and abandons their bindings | *false*, *true* (see above) | <div data-dyn-bind=  "if: false"><span>Im Hidden!</span> </div> | <div data-dyn-bind=  "if: false"></div> |
| text | Allows binding the text content of an HTML element.  Use text binding handler if for displaying UI text, not binding to user data. Use an Input control for displaying data. | *Number* or *String*. *Objects* are rendered using toString(). Use a string when using numbers via globalize.format. | <div data-dyn-bind=  "text: 'Hello World!' "></div> | <div data-dyn-bind=  "text: 'Hello World!' ">Hello World!</div> |
| css | Adds or removes CSS classes of an HTML element | CSS class *name* and *condition in which to apply css* | <div data-dyn-bind=  "css: {titleFieldClass: true}"></div> | <div data-dyn-bind=  "css: 'titleFieldClass'"  class='titleFieldClass'></div> |
| style | Adds or removes CSS Styles of an HTML element | CSS Style *name* and *value* | <div data-dyn-bind=  "style: {color: 'green'} ">  Hey!</div> | <div data-dyn-bind=  "style: {color: 'green'} ">  Hey!</div> |
| extendedStyle | Adds or removes CSS classes of an HTML element | CSS class name as a string or [obervable](#_The_Observable_Pattern) | <div data-dyn-bind=  "extendedStyle: $data.ClassName"></div> | <div data-dyn-bind=  "extendedStyle: $data.ClassName" css='MyClass'></div> |
| attr | Allows binding any attribute on an HTML element | Attribute *name* and *value*. For a list of attributes see: (<http://www.w3schools.com/tags/ref_standardattributes.asp>) | <div data-dyn-bind=  "attr:  {title: 'Header', id: 'section1'}">  </div> | <div data-dyn-bind=  "attr:  {title: 'Header', id: 'section1'}"  id='section1' title='Header'>  </div> |
| foreach | Used with (observable) arrays. For each item in an array, re-renders the innerHTML of the element with *foreach* binding.  *foreach* is a [context-changing](#_Data_Context) binding handler. When inside the loop the context ($data) is the current value/object in the array. You can also access the current index using $index | *Array* of objects or primitives. | <!--People: John, Lisa and Jacob-->  <div data-dyn-bind=  "foreach: $data.People">  <span data-dyn-bind=  "text: $data.Name" ></span>  </div> | <div data-dyn-bind=  "foreach: $data.People">  <span data-dyn-bind=  "text: $data.Name">  John  </span>  <span data-dyn-bind=  "text: $data.Name">  Lisa  </span>  <span data-dyn-bind=  "text: $data.Name" >  Jacob  </span>  </div> |
| click | Allows binding to the onclicked event of an HTML element | The JavaScript function to evaluate when the HTML element is clicked. The function may be defined inline or in the JavaScript file | <div data-dyn-bind=  "click: function(){return alert()}">  </div> | <div data-dyn-bind=  "click: function(){return alert()}"  onclick='function(){return alert()}'>  </div> |
| selected | Allows binding to the aria-selected attribute | The attribute value | <div data-dyn-bind=  "selected: 'true'">  </div> | <div data-dyn-bind=  "selected: 'true'"  aria-selected='true'>  </div> |
| value | Allows binding to the Value property of an HTML element. Works well with <input> | The *value* to bind to | <input data-dyn-bind=  "value: 'enter search terms...'">  </input> | <input data-dyn-bind=  "value: 'enter search terms...'"  Value="enter search terms..." >  </input> |
| prop | Allows binding to any standard HTML element property. Using the innerHTML element property is the correct way to programmatically HTML into a control. The text binding handler will not work for this purpose. | The value to bind to the property | <div data-dyn-bind="prop: {innerHTML: '<span>Hi</span>'}">  </div> | <div data-dyn-bind="prop: {innerHTML: '<span>Hi</span>'}">  <span>Hi</span>  </div> |

## Task Recorder support

### Recording and Playback support

Task Recorder is able to record and playback controls through the properties and commands exposed by the X++ Runtime Class. Each FormPropertyAttribute or FormCommandAttribute on the class exposes an API that Task Recorder is capable of recording and automating for playback. In order to opt into this support, the property/command should call the Task Recorder logging APIs for user actions. When calling these APIs, the property or command name is supplied so that Task Recorder knows which property/command to execute during playback.

#### Recording commands

[FormCommandAttribute("MyCommand")]

public void myCommand(str myarg = "")

{

container cmdargs = [myarg]

using (var scope = SysTaskRecorder::addCommandUserAction(

"MyCommand", this ,cmdargs))

{

*// Command logic goes here*

}

}

The highlighted code shows the appropriate way to log a command user action to Task Recorder. Only commands that are the result of direct user action should be logged to Task Recorder. Commands/methods that occur as an indirect result of user action should not be logged.

In the code above, the **addCommandUserAction** method allows specifying the command that should be recorded and automated during playback. Other arguments include the control on which the command exists, and (optionally) any arguments to the command (passed as a container array). The reason for a syntax that allows specifying a command name that may differ from the command in which the logging occurs is to allow the control author the freedom to decide when and where to log the command user action to Task Recorder.  
  
The command logic is nested within the scope of the **addCommandUserAction** so that if other user actions are logged to Task Recorder within the scope of the command, Task Recorder is able to identify the cause and effect relationship between the command and the other user actions in the code path.

#### Recording properties

[FormPropertyAttribute(FormPropertyKind::Value, "MyProperty")]

public str myProp(str value = "")

{

using (var scope = SysTaskRecorder::addPropertyUserAction(

"MyProperty", this,value))

{

*// Property logic goes here*

}

}

The highlighted code shows the appropriate way to log a property change action to Task Recorder using **addPropertyUserAction**. The syntax follow the syntax for **addCommandUserAction** closely.

Remember that in most cases properties should be read-only from the View Model, and commands should be used the value of properties since property values often need validation to execute before the new value is accepted. For this reason, recording property user actions should be a rare concurrence and only used for properties that manage the state of the control and not with properties accepting user input. When recording user input, log a user action for the command which validates the value of the property, instead of logging the property user action directly. This ensures that Task Recorder will playback the command and run the validation, instead of applying the property value directly.

#### Example: Simple Calculator Recordability

1. Open the **SimpleCalculator** X++ Runtime Class in Visual Studio
2. In the **SetExpression** method, wrap all of the existing code in an **addCommandUserAction** call, as highlighted below

[FormCommandAttribute("SetExpression")]

public void setExpression(str exp = "")

{

container cmdargs = [exp];

using (var scope = SysTaskRecorder::addCommandUserAction(

"SetExpression", this ,cmdargs))

{

*// Sanitize the expression*

exp = this.sanitize(exp);

*// Set the intnernal expression property*

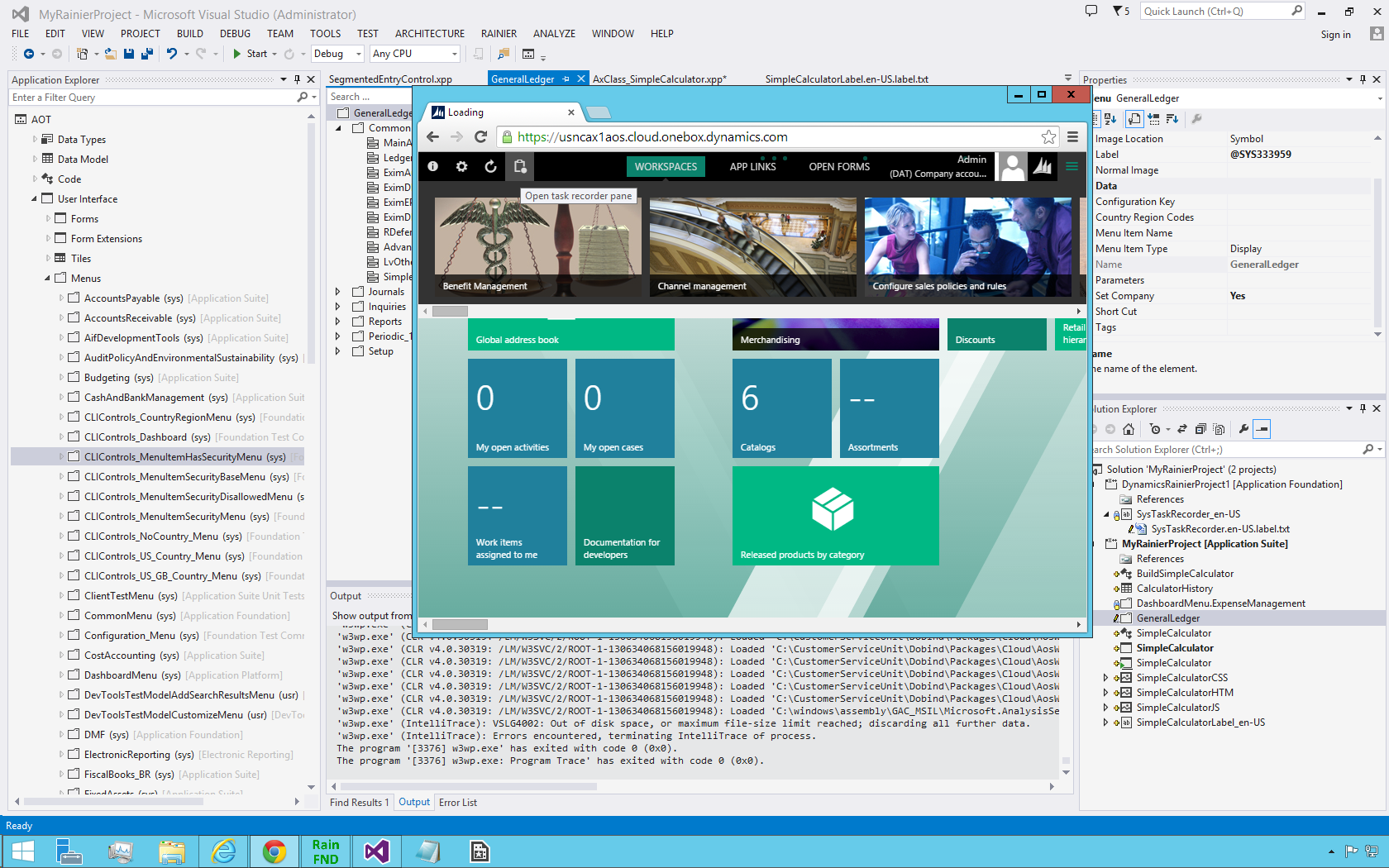
this.expression(exp);

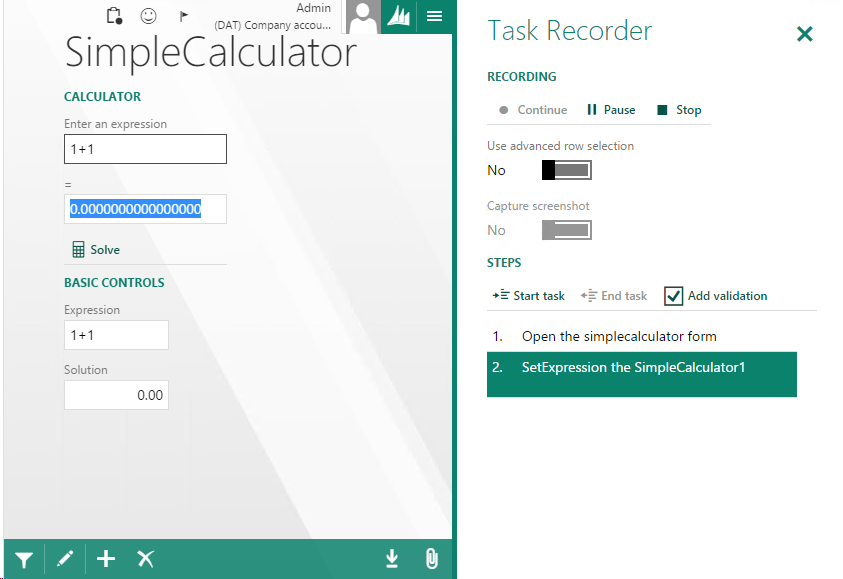
}

}

1. Save the changes.

In order to test the SimpleCalculator’s recordability you will need to add a menu item for the form so that the form can be opened from within the product. You will then need to add this menu item to one of the menus available from AppLinks. Once you have added a menu item for the form to AppLinks, try to record the form with Task Recorder

1. Build the **Rainier Project**. Run the form without debugging.
2. Open Task Recorder from the navigation pane  
   
3. Click **Create**
4. Provide a name for the recording and click **Start**
5. Navigate to the **SimpleCalculator** form using the menu and menu item just added
6. On the **SimpleCalculator** form, enter “1+1” into the expression for the calculator. Then press Tab



Note that a user action was recorded due to setting the expression in the calculator. Because we haven’t added an AX label for out control’s command, a default label is used which follows the format [command name] the [control name].

#### Creating labels

[TBD]

#### Right-click actions support

Task Recorder is able to record Task Recorder-specific user actions on controls. Currently the supported actions are **copy, paste** and **validate.**

Adding support for these actions is simple, but requires considering if the actions are appropriate for the control and what is the correct way to integrate them. In this guide, we will simply show how to enable these actions on the SimpleCalculator, and provide enough information to indicate how these actions may be used for other controls.

Each of the right-click actions are implemented using standard control Commands that follow a prescribed pattern for integrating with Task Recorder. Let’s begin with creating the commands for the **SimpleCalculator** example.

1. Add the following commands to the **SimpleCalculator** runtime class.

[FormCommandAttribute(identifierstr(copy\_Value))]

internal str copy\_Value()

{

return SysTaskRecorder::copyControlProperty("Expression", this);

}

[FormCommandAttribute(identifierstr(paste\_Value))]

internal void paste\_Value(str id, str value)

{

SysTaskRecorder::pasteControlProperty("Expression", this, id);

this.expression(this.sanitize(value));

}

[FormCommandAttribute(identifierstr(validate\_Value))]

internal void validate\_Value(str id, str value)

{

SysTaskRecorder::validateControlProperty(

"Solution", this, id == "" ? value : id, id != "");

}

The copy\_Value command logs a copy user action to Task Recorder by telling Task Recorder what property was copied. The value to be copied is determined in the View Model, as described further below. In the SimpleCalculator example only the Expression property is copyable, so the command logs a copy user action for the Expression property.

The paste\_Value command logs a paste user action to Task Recorder. It tells Task Recorder which property was pasted into, and the value that was pasted into the property. The command accepts two values, an ID which may reference another control, and a value to paste. In cases where the user is pasting a value by reference, meaning the value to be pasted comes from the copied property of another control, then the ID of the copied control is passed in. In cases where the user is not pasting by reference, then only the value is passed in. In the SimpleCalculator only the Expression property is pasteable, so the command “pastes” value by using the sanitize and expression methods. This ensures that the value undergoes validation before being set on the Expression property. We don’t call the setExpression method because that method will log a command user action to Task Recorder as a part of setting the expression. We only wish to log the paste user action, not the command user action.

The validate\_Value command logs a validation user action to Task Recorder. It tells Task Recorder the property that is being validated, and against what value to validate the property. The value can be static or a reference to the value of another copied control, so the ID of the copied control may be passed in as with the paste\_Value command. The command tells Task Recorder that the value to validate against is either the current value of the property as it is passed to the command, or the value is sourced from another copied control if the ID is set, so the ID of the copied control is passed Task Recorder to resolve the reference to the copied control’s property.

1. Add the following recordability function to the prototype for the SimpleCalculator JavaScript View Model.

$dyn.controls.SimpleCalculator.prototype = $dyn.extendPrototype($dyn.controls.Input.prototype,

{

recordability: function () {

var self = this;

return {

copy: !self.copy\_Value ? undefined : function (callback) {

var copiedValue = $dyn.value(self.Expression);

$dyn.callFunction(self.copy\_Value, self, [], function (copiedId) {

if (callback) {

callback({ Value: copiedValue, Id: copiedId });

}

});

},

paste: !self.paste\_Value ? undefined : function (id, value) {

$dyn.callFunction(self.paste\_Value, self, [id, value]);

},

validate: !self.validate\_Value ? undefined : function (id, value) {

if (id === null) {

value = $dyn.value(self.Solution);

}

$dyn.callFunction(self.validate\_Value, self, [id, value]);

}

};

}

});

The recordability function exposes APIs that Task Recorder will call when a user right-clicks on a control and selects one of the options under “Task Recorder” in the menu.

The copy function exposed by the control will call the underlying copy\_Value command implemented by the logical control when the user clicks the Copy option in the right-click menu. It also stores the current value of the property from the View Model in order to relay that value to Task Recorder when a paste or validate right-click action is used.

The paste function simply calls the paste\_Value command and passed in the relevant information.

The validate function first determines whether the user has chosen to validate against the current value, or against the value of a copied control, by checking to see if the ID of the copied control has been set. If the ID is null, then the current value is passed to the command.