Evoke

cappture your imagination

Working with Generated Visual Studio projects

Document version 1.0

Table of Contents

Introduction

This document describes the structure of the Visual Studio projects created by Evoke's app generator. It also describes the various ways in which the developer can use Visual Studio to integrate custom (non-generated) code with the files created by the generator.

Additionally, sections at the end of the document describes the internal APIs which Evoke provides in order to expose Evoke implemented functionality that your custom code may interact with.

The structure of the Visual Studio solution created by the app generator is dependent on the run-time platforms selected at the time of code generation. However, each of the types of project that can be generated share a common set of principles in terms of how the developer may augment the generated code with custom content.

The overall Visual Studio solution structure

Evoke's app generator creates a multi-project Visual Studio solution that contains at least two projects. One project will always be dedicated to data repository access and another will implement a web site/service infrastructure. Any others will correspond to the types of run-time platforms selected within the app generator.

The overall solution structure will only be created the first time the app generator is targeted at a blank or non-existent folder. Subsequent code generations will only update sections of the relevant projects.

The data repository access project

The app generator will always create a Visual Studio project called "RepositoryAccess". This project contains all of the generated code relating to data access via one of the supported data repository types.

The repository access project is then used (referenced) by all of the other types of generated projects ("UI projects") in order to gain access to persisted data storage.

Folder structure overview

The repository access project breaks down the generated code into two main parts. The first part contains the classes directly referenced by all of the UI projects. These classes then, in turn, utilize the second part which consists of a corresponding set of data repository specific classes via a series of interfaces. There may be more than one instance of the second part depending on the number of different data repositories specified within the app generator.

This structure allows a clean partitioning of repository agnostic and repository specific code.

The ViewModels folder

This folder contains the repository agnostic series of classes that represent the Data Entities created within the app designer.

All of the classes contained within this folder are defined as *partial* classes. This is the mechanism by which custom code relating to each class can be integrated.

The RepositoryEntities folder

This folder holds a series of interface definitions, one for each data entity defined within the app designer. This is the mechanism used by Evoke to connect the repository agnostic (view model) classes with the platform specific repository access classes.

As in the ViewModels folder, all of the classes contained within this folder are defined as *partial* classes in order to allow custom code relating to each class to be integrated.

The RepositoryPlatforms folder

This folder contains a sub-folder for each data repository type selected within the code generator. Each of these sub-folders contains an identical structure - the only difference being the actual implementation code in terms of repository access.

Custom code files

Both the RepositoryEntities and the RepositoryPlatforms folders comprise code files containing partial classes. The partial class usage allows a series of "shadow" custom code files to contain any custom logic required for each data entity. The custom code files are located in the “Custom” folder.

When the app generator first creates the Visual Studio solution, it will create both the generator owned code files and the custom code files (which will be stub files containing no implementation code – that’s what you fill in as necessary).

Within the generator owned code files, calls into the custom code stub files will be inserted. It is therefore important that, even though you logically own the custom code files, you do not delete the entry points (functions) within these files otherwise your project will not compile. If you don’t need to use one of the entry points, just leave its implementation code as the generator first creates it.

The Web project

The app generator will always create a Visual Studio project called "xxx Web", where xxx is the name of your app. This project contains generated code relating to the implementation of a RESTful web service. This project, in turn, references the RepositoryAccess project in order to interact with back-end repositories.

If you select to generate code for a run-time platform which uses an HTML-based UI (e.g. Desktop browser app or web app) the generator will also incorporate UI-related code into the Web project.

The Web project is based on the standard Visual Studio ASP.NET MVC Web API template. It uses MVC for the URI routing of the RESTful service’s HTTP request handling and also for conditional inclusion of web assets (HTML, CSS and JavaScript files).

The Web project also makes use of ASP.NET’s bundling technology to optimize the transport of files to the client, along with web grease to automatically compress files. All of the above is automatically set up for you within Visual Studio by the generator.

As well as the RESTful web service and ASP.NET infrastructure, the Web project also contains all content relating to the app’s UI for HTML based platforms. This content is organized under 3 top-level folders, collectively termed the *main content folders*:

CSS	Holds all CSS code files used within the app
Scripts	Holds all JavaScript code files used within the app
Views	Holds all HTML files used within the app

These is also an *Images* folder which holds all of the images used within the app

These main content folders contain both generated and custom codes files. They all contain a sub-folder for each of the run-time platforms selected within the code generator (e.g. Desktop, iPad) and they all contain a sub-folder called *Shared* – this folder can be used to add custom content common to all run-time platforms.

The Script main content folder also contains a sub-folder called *Entities* – this holds the JavaScript classes that represent the data entities defined within the app designer.

Within all of the main content folders, there will be files with a name containing the string ".custom."; these are the files that you may manually modify within Visual Studio and they will be only touched by the generator once – i.e. on first creation. After that, the generator will not attempt to make any changes to existing custom files.

Under each of the run-time specific sub-folders there is a folder called *Pages*. This folder contains a collection of sub-folders, one per page series within the app design.

Adding custom HTML content

Because of the architecture of web applications, the way in which you add custom HTML content is slightly different to the way in which you add custom CSS and JavaScript.

If you need to add custom HTML to your app, you will need to set the content type of the relevant page segment within the page's design to "Custom". This will result in dedicated file being created within the Views main content folder. For example, let's say you have the following within your app design:

- a page series called "Customers"
- within this series a page called "Main"
- within this page a segment with custom content that has a widget Id of "Details"

Within the Views main content folder, under the Pages sub-folder you will have a folder called "Customers" – this obviously relates to the Customers page series. Within this folder there will be a file called "Customers.cshtml" – this will hold the generated HTML. There will also be a file called "Customers.custom.Details.cshtml" – this is where you need to put the custom HTML for the custom content page segment.

The App namespace

In order to add structure, clarity and logical partitioning to the various sections of JavaScript that comprise an Evoke web app, Evoke uses a pseudo namespace architecture to organize its code. At the root of this namespace is the global variable "App". With only a couple of exceptions, everything within Evoke (in terms of JavaScript) sits within this variable's scope.

Adding custom code to entities

The Scripts main content folder contains a sub-folder called "Entities" – this holds the generated

JavaScript code relating to the app's data entities defined within the designer. There will be one sub-folder per entity type and this will contain a custom file allowing you to add your own custom code to the entity class definition.

The code generator will create stub content within this custom file that you can either fill in or leave as-is. In all of the stub entry points, the ***entityInstance*** argument (if present) holds a reference to the related instance of the entity type. The following stub entries are created:

```
_populate: function (entityInstance) { }
```
This function is called after an instance of the entity type has been populated with data.

```
_serialize: function (entityInstance, serialization) { }
```
This function is called after an instance of the entity type has been serialized ready for transport down to the RESTful web service. The ***serialization*** argument holds a JSON object that contains the serialization data automatically created by Evoke. You may amend/augment the content of this JSON object as required.

```
_validate: {
    propertyname: function (entityInstance, candidateValue) { },
    _all: function (entityInstance) { },
}
```
This namespace allows you to add custom validation on a per-property basis. Add one function per property name as required, and set the function name as the name of the relevant property. The ***candidateValue*** argument holds the value that is about to be assigned to the property. If the candidate value is not valid, the custom function should return a string value holding a description of the detected error.

```
_validationControl: {
    propertyname: function (dataValidationControl, entityInstance) { },
}
```
This namespace allows you to adjust data validation settings prior to the launch of a per-property asynchronous remote validation task. The ***dataValidationControl*** argument holds a pre-configured dataValidationControl instance (see section below) the content of which you may amend as necessary.

If you wish to suppress the validation action, you should set the *suppressValidation* property of the dataValidationControl argument to *true.*

```
_validationCompleted: function (entityInstance, errorDescription, dataValidationControl) {
}
```
This function is called after an instance of the entity type has been remotely validated. The ***errorDescription*** argument holds the error description reported by the remote validation logic – or is null if no validation error occurred. The ***dataValidationControl*** argument holds the same dataValidationControl instance as supplied to the _validationControl function above.

```
calculatedpropertyname: {
    get: function (entityInstance) {
        // return the property's value
    },
    set: function (entityInstance, candidateValue) {
        // perform required action (if any) on property value set
    },
}
```
This stub entry is a sample boiler plate indicating how you should implement calculated properties. A property of a data entity can be flagged as calculated within the Data page of the app designer.

You may remove this example boiler plate from the custom code file if you wish.

Adding custom code to pages

Each of the run-time platform folders within the *Scripts* main content folder contains a sub-folder called “Pages” – this holds the generated JavaScript code relating to the app’s page series definitions created using the app designer. Within the *Pages* folder there will be one sub-folder per page series and this will contain a custom file allowing you to add your own custom code to the page series definition.

In terms of JavaScript code, each page series exists within the App.PageSeries object. All custom code for a particular page series (as a whole) exists within a dedicated object: App.PageSeries.*xxx*.custom, where *xxx* represents the page series name.

The custom code relating to individual pages within the series exist with the object: App.PageSeries.*xxx*.Pages.*ppp*.custom where *ppp* represents the page name.

The custom code entry points for each App.PageSeries.*xxx*.custom object (the page series as a whole) are as follows (in all of the above the ***self*** argument holds a reference to the App.PageSeries.*xxx* object):

`initialize: function (self) { }`
This function is called when the page series is first loaded into memory at the start of app execution. At the point when this function is called the DOM will be fully loaded and the internal JavaScript structure of the page will exist.

`activated: function (self) { }`
This function is called when the page series is made the currently displayed series (usually when the end-user selects a menu option).

`deactivated: function (self) { }`
This function is called when the page series loses its status as the currently displayed series (usually when the end-user selects a different menu option).

`generalLogic: container/namespace object { }`
This object acts as a container for you to place logic which applies to the page series as a whole.

`loadTopLevelData: function (self, position) { }`
This function is called when the page series is first initialized and allows any custom data state to be loaded as necessary. This function is, in fact, called twice – once just before the system performs data loading as defined by the data source definition(s) of the page and again once this system-initiated loading has completed. The **`position`** argument holds a string value set to either "before" or "after" indicating which of these 2 possible invocations is active.

`editStarted: function (self) { }`
This function is called when the page first enters the edit in progress state – i.e. when the first data edit is made in any of its constituent pages.

`editTriggered: function (self, pageId, triggerWidget, triggerInfo) { }`
This function is called whenever an editable widget in any of its constituent pages is used to make a data amendments. The **`pageId`** argument holds the programmatic ID of the source page and the **`triggerWidget`** argument holds a reference to the source widget. The **`triggerInfo`** argument holds information specific to the trigger event.

`navigateToChild: function (self, pageId) { }`
This function is called whenever a child navigation action is initiated within the series. The **`pageId`** argument holds the programmatic ID of the target page.

`navigateToParent: function (self, pageId) { }`
This function is called whenever a navigate to parent action is initiated within the series. The **`pageId`** argument holds the programmatic ID of the child page.

`navigateToSibling: function (self, pageId) { }`
This function is called whenever a sibling navigation action is initiated within the series. The **`pageId`** argument holds the programmatic ID of the target page.

`saveChanges: function (self, position, status, data, response) { }`
This function is called whenever a save of the outstanding data amendments within the page series is initiated. This function is called twice for each save action – once just before the system initiates the save action and once again when the save action has completed.

The **`position`** argument holds a string value set to either "before" or "after" indicating which of these 2 possible invocations is active.

The **`status`** argument holds a string value set to the return status of the save action – this can be either "success" or "fail".

The **`data`** argument holds either the JSON data returned by the RESTful service or the error message on failure.

The **`response`** argument holds the ajax response object associated with the save action.

`undoChanges: function (self, position) { }`
This function is called whenever a cancel of the un-saved data amendments within the page series is initiated. This function is called twice for each undo action – once just before the system initiates the undo action and once again when the undo action has completed.

The **`position`** argument holds a string value set to either "before" or "after" indicating which of these 2 possible invocations is active.

The custom code entry points for each App.PageSeries.*xxx*.Pages.*ppp*.custom object (each individual page) are as follows (in all of the above the ***self*** argument holds a reference to the App.PageSeries.*xxx*.Pages.*ppp*.custom object).

```
generalLogic: {
}
```
This custom code entry point provides a container for all of your general purpose code that is associated with the particular page. Typically you will add function declarations within the generalLogic object.

```
widgetEvents: {
}
```
This custom code entry point provides a container for all of your widget event handling code for the widgets contained on the particular page. Please refer to the Widgets API section below for more details on how to create widget event handlers.

Common JSON objects used within generated code

The following classes (JSON object structures) are used within the Web project.

dataRetrievalControl

This class is used to control the behaviour of nested/related entity data retrieval. Instances of this class can be created by using the App.Entity.dataRetrievalControl factory function:

```
App.Entity.dataRetrievalControl = function (propertyName, calculatedProperties,
partialProperties, entityProperties, retainRetrievalCache, useAppLevelCache,
pageLevelCacheName, suppressEpOptimization)
```

This factory function can be passed initial property values for the returned instance as arguments on its call signature. Any property values that are not relevant should be passed as null values. Alternatively you may pass in a single argument containing a JavaScript object with property names (and associated values) matching the call signature argument names as required.

The dataRetrievalControl class is designed to be used in a recursive manner in order to allow nested entity retrieval to be performed to an arbitrary depth for any number of properties in the parent entity type.

Perhaps the best way to explain how to use this class is to give a couple of examples. Let's start with an easy one first. The code below constructs a dataRetrievalControl instance that is then passed into the read of an Organization entity.

```
var retrievalControl = App.Entity.dataRetrievalControl({ entityProperties: "Contacts" });

App.Entity.Organization.read(key, retrievalControl)

   .done(function(data) {

       // actions on successful read

}

   .fail(function(error) {

       // actions on unsuccessful read

}
```

The above code will result in the *Contacts* property being populated with data as part of the read action. Put another way, it will result in the *Contacts* entity property data being "eager-loaded".

The next example shows how the dataRetrievalControl class can be used recursively in order to retrieve related entity data to an arbitrary depth. In plain words, in one round-trip the code below:

- reads an Organization
- and retrieves its associated sales orders
- and for each sales order, retrieves the associated order lines
- and for each order line retrieves the associated product and the associated delivery details

```
var orderLinesRetrieval = App.Entity.dataRetrievalControl({ propertyName: "OrderLines",
entityProperties: ["Product", "Deliveries"] });

var salesOrdersRetrieval = App.Entity.dataRetrievalControl({ propertyName: "SalesOrders",
entityProperties: orderLinesRetrieval });

var retrievalControl = App.Entity.dataRetrievalControl({ entityProperties:
salesOrdersRetrieval });

App.Entity.Organization.read(key, retrievalControl)

   .done(function () {

      … etc
```

The above code is reasonably complicated, so let's step through it bit-by-bit. The first 3 lines of code create dataRetrievalControl instances. To better understand the code, let's start with the 3rd line first – this line of code creates a dataRetrievalControl instance that indicates that the (top-level) related entity data to be loaded is defined by the variable salesOrdersRetrieval – note that no propertyName is supplied here – that' s because it's the top-level dataRetrievalControl instance.

On the 2nd line of code, the salesOrdersRetrieval variable gets assigned a dataRetrievalControl instance that relates to the *SalesOrders* property. So, the result of the 2nd line of code is that the entity property *SalesOrders* (of the read Organization) is populated **and** that the entity properties defined within variable orderLinesRetrieval are to be used to control the entity property retrieval of each sales order entity instance that exists within the populated *SalesOrders* property. That is, we are now reaching down into a 3rd level of related entity retrieval.

The 1st line of code then defines that for each *SalesOrder* entity its *OrderLines* entity property is to be populated and that for each order line within that property, the *Product* and *Deliveries* entity properties are to be populated. Note, that in the 1st line of code we provide the "Product" and "Deliveries" property names as literal strings because we don't need to reach down into those related objects to force a further level of nested retrieval.

Wow – that's quite a lot of work being triggered by just 3 or 4 lines of code! Read through the above a few times to make sure you follow the way in which the dataRetrievalControl class is being used in order to define the retrieval of what is often called a "graph" of related entities.

The dataRetrievalControl object supports the following properties:

propertyName	The name of the property whose data retrieval is being defined. This should be left blank if the dataRetrievalControl instance is the one being directly passed into the read/select action, i.e. it is the top-level dataRetrievalControl instance.
calculatedProperties	A string value or an array of string values holding the names of the calculated property values that are to be generated as part of the read/select action – i.e. the UI is going to assume that these properties will contain the correct values after the read/select returns.
partialProperties	A string value or an array of string values holding the names of the properties that are to retrieved for the entity property supplied in the propertyName. This property can be used to optimize data retrieval for entities that contain very large number of proprties. If you supply this property value, you are indicating that rather than return (serialize) full entity instances for the entity property specified by propertyName, only a subset of property values are to be returned.
entityProperties	This may contain any one of the following: • a string value (property name) • an array of string values (multiple property names) • a dataRetrievalControl instance • an array of dataRetrievalControl instances The use of this property is explained in the 2 code examples above.
retainRetrievalCache	internal use only.
useAppLevelCache	internal use only.
pageLevelCacheName	internal use only.
suppressEpOptimization	internal use only.

dataSelectionControl

This class is used to control the behaviour of data selection actions triggered within the UI. Instances of this class can be created by using the App.Entity.dataSelectionControl factory function:

```
App.Entity.dataSelectionControl = function (selectionID, filterValues, sortProperties,
GUID, pageSize, pageIndex)
```

This factory function can be passed initial property values for the returned instance as arguments on its call signature. Any property values that are not relevant should be passed as null values. Alternatively you may pass in a single argument containing a JavaScript object with property names (and associated values) matching the call signature argument names as required.

The dataSelectionControl class allows the UI to execute selections (via the static *select* method of each entity) that have been defined as part of an entity definition. All selection activity within Evoke is performed via such (database-agnostic) selection definitions. Also note that the code generator will create a "Select" namespace as part of each entity's API. This Select namespace will contain an entry for each selection definition with each one having a calling signature based on the individual selection definition. You may, thus, find that using this Select namespace is a more intuitive way of initiating selections as opposed to using the select method on the entity class directly.

The dataSelectionControl object supports the following properties:

selectionID	The name of the selection definition to be used.
filterValues	A single or array of values to be used as part of the selection action. The value(s) here should be supplied in the same sequence as that of the filter value names defined within the selection definition being referenced.
sortProperties	A single or array of string values. The value(s) here should be the list of property names to be used to sort the selected items; highest sort order property first.
GUID	A GUID to identify this specific selection action or a GUID of a previously executed selection. This should only be supplied if you require paginated results or are retrieving a second or subsequent page of results of a previously executed selection.
pageSize	The page size to be used (i.e. the number of items per page). This should only be supplied if you require paginated results
pageIndex	The index of the page to be retrieved. This should only be supplied if you require paginated results or are retrieving a second or subsequent page of results of a previously executed selection.

dataUpdateControl

This class is used to control the behaviour of data update actions triggered within the UI. It supports the following properties:

ignoreErrors	Boolean value indicating that where a list of entity instances is being updated (as opposed to just a single instance), whether any errors encountered should halt (and abort) the update of other instances. The default behaviour is for this property to have a value of false, i.e. as soon as any error occurs, all previous updates will be cancelled and the update action as a whole will be aborted.
noSerializationProperties	A single or array of string values. The name(s) of the entity properties that should **not** be serialized and thus will not be part of the update payload.

dataValidationControl

This class is used to control the behaviour of remote property validation actions triggered within the UI. It supports the following properties:

validationPropertyName	The name of the property to be validated.
validationValue	The value to be validated.
propertyNames	A single or array of string values. The name(s) of properties whose values are required in order to perform the remote validation.
propertyValues	A single or array of string values. The value(s) corresponding to the supplied list of property names.

The anatomy of a JavaScript data entity

Each of the data entities that you define within your app will be manifested in the Web project as a JavaScript "class" (in fact 2 classes are present for each entity – more on this later). All of the JavaScript code relating to these classes are held within the Scripts/Entities main content folder, and they are all contained with the App.Entity namespace.

All entities share the same basic architecture, the main difference being their list of properties. This section describes the properties and methods that are common across all data entities.

One aspect to note about properties is that they are exposed as JavaScript functions and should be used as follows (here we assume that the entity instance is held in variable myEntity; has a property called "Name" and that a variable called "myVar" needs to receive and supply the Name property value).

To retrieve the current value of the Name property: `myVar = myEntity.Name();`

To assign a value to the Name property: `myEntity.Name(myVar);`

Each data entity has 2 separate classes defined – one that represents a single instance (the "singular" class) and another that represents a list containing multiple singular instances (the "collective" class).

The singular data entity class

Below are the standard properties and methods for the singular data entity class.

Some of the methods of this class trigger background actions against the underlying data repository. In these situations, a jQuery promise object is returned as the direct return value of the function. The .done and .fail methods of this promise object can then be used to attach functions to the successful or unsuccessful outcome of the repository action. Where relevant the .done method will be supplied the return data value; the .fail method will be supplied the error message text.

Property: IsModified

Indicates whether this entity instance has been modified since it was first read or last updated, whichever is the most recent.

Property: Key

The primary key of the entity. Note, when you create a new instance of an entity (via its static *create* method), this property will be blank; the assumption made by the UI is that the repository side of things will set the value of this property when the entity is saved via the *update* method. When the update method returns, the UI will have access to the newly assigned Key value.

Property: GUID

A system-generated unique integer value for the in-memory entity instance. Note, this is a UI only property used for unique identification purposes only and does not relate to any data value held within the underlying data repository.

Method (instance): cancel

Signature: `function ()`

Cancels any data modifications made since the entity was first read or since it was last updated, whichever is the most recent:

Method (instance): clear

Signature: `function (clearSimpleProperties, clearEntityProperties)`

Sets all properties (as indicated by the supplied arguments) to a blank/empty value.

Arguments:

`clearSimpleProperties`	Boolean value indicating whether simple (non-entity) properties should be cleared.
`clearEntityProperties`	Boolean value indicating whether entity properties should be set to empty singular/collective instances.

Method (instance): clearError

Signature: `function (propertyName)`

Clears an error currently attached to the supplied property name.

Arguments:

`propertyName`	String value holding the name of the property to have its error cleared.

Method (static): create

Signature: `function ()`

Creates a new singular instance of the entity.

Return value:

jQuery promise, the completed (.done) value being the new singular entity instance.

Method (instance): delete

Signature: `function ()`

Deletes the entity instance from the underlying repository.

Return value:

jQuery promise.

Method (static): delete

Signature: `function (key)`

Deletes the specified item from the underlying data repository.

Arguments:

`key`	The primary key of the item to be deleted.

Method (static): logic

Signature: `function (logicID, logicData)`

Executes a section of business logic contained outside of the UI environment.

Arguments:

`logicID`	The logical ID of the section of business logic to be executed.
`logicData`	A JavaScript array holding data values from the UI required to execute the associated logic.

Return value:

jQuery promise , the completed (.done) value being the value returned from the executed logic.

Method (static): read

Signature: `function (key, dataRetrievalControl)`

Reads a single entity instance from the underlying data repository.

Arguments:

`key`	The primary key of the entity to be read.
`dataRetrievalControl`	A dataRetrievalControl instance. See previous section for details.

Return value:

jQuery promise, the completed (.done) value being the singular entity instance read from the data repository.

Method (static): select

Signature: `function (dataSelectionControl, dataRetrievalControl)`

Selects multiple entity instances from the underlying data repository.

Arguments:

`dataSelectionControl`	A dataSelectionControl instance. See previous section for details.
`dataRetrievalControl`	A dataRetrievalControl instance. See previous section for details.

Return value:

jQuery promise, the completed (.done) value being the collective entity instance holding the singular entity instances selected from the data repository.

Method (instance): setError

Signature: `function (propertyName, errorDescription)`

Assigns an error message to the specified property. This will, implicitly, flag the property value as being invalid.

Arguments:

`propertyName`	The name of the property in error.
`errorDescription`	A description of the error.

Method (instance): update

Signature: `function (dataUpdateControl)`

Triggers a save of the entity's current property values to the underlying data repository.

Arguments:

`dataUpdateControl`	A dataRetrievalControl instance. See previous section for details.

Method (instance): validate

Signature: `function (propertyName)`

Triggers an explicit validation of the specified property.

Arguments:

`propertyName`	The name of the property to be validated. If this argument is omitted or is null/blank, all properties will be validated.

The collective data entity class

Below are the standard properties and methods for the collective data entity class (all methods are instance methods).

Some of the methods of this class trigger background actions against the underlying data repository. In these situations, a jQuery promise object is returned as the direct return value of the function. The .done and .fail methods of this promise object can then be used to attach functions to the successful or unsuccessful outcome of the repository action. Where relevant the .done method will be supplied the return data value; the .fail method will be supplied the error message text.

Property: IsModified

Indicates whether this entity instance has been modified since it was first read or last updated.

Property: items

Returns the underlying JavaScript array containing the constituent singular data entity instances.

Method: add

Signature: `function (newInstance, insertBefore)`

Adds an existing entity instance to the collection.

Arguments:

`newInstance`	The entity instance to be added.
`insertBefore`	(optional) The position at which to insert the added instance. If left blank, the instance will be added at the end of the collection.

Method: addNew

Signature: `function (insertBefore)`

Creates a new singular entity instances and adds it to the collection.

Arguments:

`insertBefore`	(optional) The position at which to insert the new instance. If left blank, the new instance will be added at the end of the collection.

Return value:

jQuery promise, the completed (.done) value being the newly created singular entity instance.

Method: cancel

Signature: `function ()`

Cancels any data modifications for all of the constituent singular instances made since they were first read or since last updated, whichever is the most recent:

Method: clear

Signature: `function (flagAsDeleted)`

Removes all instances currently held within the collection.

Arguments:

`flagAsDeleted`	Boolean value. If set to *true* all removed items will be deleted from the underlying data repository when the collective instance is saved (*update* method called).

Method: count

Signature: `function ()`

The number of singular instances currently held within the collection.

Return value:

Integer value (the count total).

Method: delete

Signature: `function (index, indexIsGUID)`

Deletes the specified singular entity instance held within the collection from the underlying data repository and removes it from the collection.

Arguments:

`index`	The position within the collection of the entity instance to delete.
`indexIsGUID`	(optional) If set to *true* indicates that the supplied *index* argument is the GUID property value of the entity to delete.

Method: find

Signature: `function (findKey, findKeyIsGUID, returnIndex)`

Locates a specific entity instance within the collection based on either primary key or GUID value.

Arguments:

`findKey`	The Key property value to locate.
`findKeyIsGUID`	(optional) If set to *true* indicates that the supplied *index* argument is the GUID property value of the entity to find.

`returnIndex`	(optional) If set to *true* indicates that the return value of the function should be the index of the located instance as opposed to the actual instance.

Return value:

The located entity instance or (if the *returnIndex* argument is set to *true*) the integer index value of the located instance. Null is returned if no match was found.

Method: indexOf

Signature: **`function (findValue, matchWithProperty)`**

Locates a specific entity instance within the collection based on a supplied property value. This function will return the first instance (starting at index 0) within the collection found to match the supplied criteria

Arguments:

`findValue`	The value to search for.
`matchWithProperty`	The name of the property to inspect during the search process.

Return value:

The integer index value of the located instance. Null is returned if no match was found.

Method: move

Signature: **`function (sourceIndex, targetIndex, moveCount)`**

Moves one or more instances to a different position within the collection.

Arguments:

`sourceIndex`	The index position of the first instance to move.
`targetIndex`	The new position for the first instance being moved.
`moveCount`	The number of instances (default is 1) to be moved, starting with the *sourceIndex* position.

Method: remove

Signature: **`function (index, indexIsGUID, flagAsDeleted)`**

Removes (and optionally deletes from the underlying data repository) an instance from the collection.

Arguments:

`index`	The position within the collection of the entity instance to delete.
`indexIsGUID`	(optional) If set to *true* indicates that the supplied *index* argument is the GUID property value of the entity to delete.

`flagAsDeleted`	(optional) Boolean value. If set to *true* the remove item will be deleted from the underlying data repository when the collective instance is saved (*update* method called).

Method: setCurrent

Signature: `function (index, indexIsGUID)`

Sets the entity instance which the UI regards as the one currently selected by the user.

Arguments:

`index`	The position within the collection of the entity instance to set as current.
`indexIsGUID`	(optional) If set to *true* indicates that the supplied *index* argument is the GUID property value of the entity to set as current.

Method: update

Signature: `function ()`

Saves all modified entity instances within the collection to the underlying data repository.

Return value:

jQuery promise.

The anatomy of a JavaScript classification

Evoke's app designer supports the concept of a classification type. This provides a general purpose mechanism to cater for the very common requirement within business apps to hold reasonably static lists of status codes, categories and other classification type information.

Each classification type will (logically) contain one or more classifications items.

When your app loads, the full list of all classification items for each classification type is loaded into memory, more specifically, each classification type appears as a property of the App.GlobalData object and each of these contains the list of associated classification items.

The Menu System API

The Evoke app designer allows a logical menu structure to be defined for an app. The app generator then takes this logical design and generates the requisite code to implement it within the Visual Studio solution. Supporting this generated code are the Evoke run-time libraries, and one of the features that the run-time libraries provide is a way for you to interact with the menu system.

All JavaScript relating to the menu system is located within the App.MenuSystem object. The following methods are provided for you to use – they are all directly accessible via the App.MenuSystem object:

Method: getState

Signature: `function ()`

Returns a string value indicating the current display state of the menu system.

Return value:

String value indicating the menu system's current display state. Return value will be one of the following:

hidden	Menu system is not visible and occupies no UI space.
pinned	Menu system is visible.
floated	Menu system is visible but is floated over the top of other page content and thus takes up no UI space.

Method: hide

Signature: `function ()`

Hides the menu system.

Method: nodeEditInProgress

Signature: `function (nodeId, editInProgress)`

Allow the edit in progress indicator for a menu node to be hidden or shown.

Arguments:

`nodeId`	The programmatic ID of the target node.
`editInProgress`	Boolean value indicating whether the edit in progress indicator for the target menu node should be displayed – *true* indicating that it should be displayed.

Method: nodeHide

Signature: `function (nodeId)`

Allows an individual node to be hidden.

Arguments:

`nodeId`	The programmatic ID of the target node.

Method: nodeSelect

Signature: `function (nodeId)`

Allows programmatic selection of a specified menu node.

Arguments:

`nodeId`	The programmatic ID of the target node.

Method: nodeShow

Signature: `function (nodeId, excludeId)`

Allows one or more nodes to be made visible.

Arguments:

`nodeId`	The programmatic ID of the target node, or "*" if all nodes are to be displayed.
`excludeId`	An optional ID of a node to be excluded from the show action.

Method: nodesWithEditInProgress

Signature: `function ()`

Allows the list of nodes that are currently displaying the edit in progress indicator to be obtained.

Return value:

A JavaScript array of string values, each value being the programmatic ID of a menu node that is currently displaying the edit in progress indicator.

Method: show

Signature: `function (xxx)`

Make the menu system visible.

The Page System API

The Evoke app designer allows you to create a range of pages to contain the bulk your app's UI. The app generator then takes these page designs and generates the requisite code to implement them within the Visual Studio solution. Supporting this generated code are the Evoke run-time libraries, and one of the features that the run-time libraries provide is a way for you to interact with the page environment.

All JavaScript code relating to the page environment is located within the App.PageSystem object. The following methods are provided for you to use – they are all directly accessible via the App.PageSystem object:

Method: blockPageUI

Signature: `function (pageSeriesId)`

Blocks all user-initiated UI interactions within the specified page series.

Arguments:

`pageSeriesId`	The programmatic ID of the target page series.

Method: navigateToChild

Signature: `function (pageId, callbackFunction)`

Initiates a child (drill-down) page navigation action within the currently displayed page series.

Arguments:

`pageId`	The programmatic ID of the target page.
`callbackFunction`	Optional function to be called when the navigation action has been completed.

Method: navigateToModal

Signature: **`function (modalId, callbackFunction)`**

Initiates a modal (pop-up) page navigation action within the currently displayed page series.

Arguments:

`modalId`	The programmatic ID of the target modal page.
`callbackFunction`	Optional function to be called when the navigation action has been completed.

Method: navigateToParent

Signature: **`function (returnValue, callbackFunction)`**

Initiates a parent (back) page navigation action within the currently displayed page series.

Arguments:

`returnValue`	Optional return value to be made available to the parent page.
`callbackFunction`	Optional function to be called when the navigation action has been completed.

Method: navigateToSibling

Signature: **`function (pageId, callbackFunction)`**

Initiates a sibling (associated) page navigation action within the currently displayed page series.

Arguments:

`pageId`	The programmatic ID of the target page.
`callbackFunction`	Optional function to be called when the navigation action has been completed.

Method: unblockPageUI

Signature: **`function (pageSeriesId)`**

Unblocks a previously UI-blocked page series.

Arguments:

`pageSeriesId`	The programmatic ID of the target page series.

The Widgets API

Evoke has the concept of UI component's called "widgets" – sometimes called "controls" in other toolsets. The widgets included within Evoke provide a sound basis for creating the UI of your app; however, you may incorporate controls/components from other vendors within your Visual Studio projects if required.

The key thing about the widgets supplied with Evoke is that an implementation of these widgets is supplied (by Evoke run-time libraries) for all of the target platforms supported by Evoke.

Widgets are added to a page within the app designer using the designer's *data template* feature. Please refer to the *App Designer Overview* document for more details on this feature.

The API of a widget is composed of properties, methods and events. The properties and methods can be accessed via the in-memory object representing the widget. This object is contained within the App.PageSeries.*xxx*.Pages.*ppp*.widgets object, where *xxx* represents the page series name and *ppp* represents the name of the page containing the widget.

You can create event handlers for the events raised by a widget within the custom code file associated with the page. The app generator creates a section of stub code specifically for this purpose. Using this stub code, all event handlers are contained within the object: App.PageSeries.*xxx*.Pages.*ppp*.custom.widgetEvents

You create one function per widget as required. All of your event handlers need to have the same call signature, as follows:

```
widgetId: function (eventInfo, page, widget)
```

widgetId is the programmatic name of the widget. This is entered within the app designer.

The ***eventInfo*** argument contains both standard pieces of data about the event that has just fired as well as data specific to the particular type of event. It will always have a property called "Id" which holds a string identifying the type of event that has fired. Thus, typically, all event handlers start with a switch statement based on this Id value:

```
switch (eventInfo.Id) {
    case "someEventType":
         event specific code…
         break;
}
```

The ***eventInfo*** argument will contain any event-specific data within a property called "data". It will also contain a property called "event" which contains the base JavaScript event object that Evoke has captured internally.

The ***page*** argument contains a reference to the host page of the widget. The ***widget*** argument contains a reference to the actual widget firing the event.

All widgets have a property called "widgetType" which allows you to identify the type of widget from an arbitrary reference.

This section describes the API for each of Evoke's built-in widgets.

Action Button

The action button widget is used to present a clickable icon that displays a drop-down list of actions when clicked/tapped.

Events

showActions

Fired when the menu associated with the action button is displayed.

Event-specific data:

Name	Description
position	Contains string value “before” or “after” indicating the point at which this event has fired. You can associate code with “before” in order to control the visibility/enabled state of options within the action menu.

action

Fired when an option from the menu associated with the action button is selected.

Event-specific data:

Name	Description
actionId	The Id of the selected action.

Methods

`actionDisable: function (actionId)`

Disables the specified action within the action button’s menu.

Arguments:

Name	Description
`actionId`	The Id of the action to be disabled. The action will still appear in the menu but will be lighter gray in color and will not e selectable.

`actionEnable: function (actionId)`

Enables the specified action within the action button’s menu.

Arguments:

Name	Description
`actionId`	The Id of the action to be enabled.

`actionHide: function (actionId)`

Hides the specified action within the action button’s menu.

Arguments:

Name	Description
`actionId`	The Id of the action to be hidden.

```
actionShow: function (actionId)
```

Shows a previously hidden action within the action button’s menu.

Arguments:

Name	Description
`actionId`	The Id of the action to be shown.

CheckBox

A standard checkbox UI element.

Events

change

Fired when the ticked state of the checkbox changes.

Event-specific data:

Name	Description
value	Boolean value. *true* if the checkbox is currently ticked.

ClickableRegion

A DOM element (typically a div) that can be clicked/tapped by the user.

Events

click

Fired when the user clicks or taps on the widget.

No event-specific data.

DataGrid

An editable grid composed of rows and columns.

Events

beforeRowSelect

Fired when a row is selected but before any activity associated with the row select action is performed.

Event-specific data:

Name	Description
selectedEntity	The data entity associated with the selected row.
selectedRowIndex	The (zero-indexed) ordinal position of the selected row.
selectedColumnIndex	The (zero-indexed) ordinal position of the selected column.
clickPositionX	The precise pixel x-coordinate of the click or tap action
$rowContainer	A jQuery object representing the DOM element containing the selected row

afterRowSelect

Fired when a row is selected after any activity associated with the row select action has been performed.

Event-specific data:

Name	Description
selectedEntity	The data entity associated with the selected row.
selectedRowIndex	The (zero-indexed) ordinal position of the selected row.
selectedColumnIndex	The (zero-indexed) ordinal position of the selected column.
clickPositionX	The precise pixel x-coordinate of the click or tap action
$rowContainer	A jQuery object representing the DOM element containing the selected row

clickSelected

Fired when the already selected row is clicked by the user.

Event-specific data:

Name	Description
selectedEntity	The data entity associated with the selected row.
selectedRowIndex	The (zero-indexed) ordinal position of the selected row.
selectedColumnIndex	The (zero-indexed) ordinal position of the selected column.
clickPositionX	The precise pixel x-coordinate of the click or tap action
$rowContainer	A jQuery object representing the DOM element containing the selected row

tickSelect

Fired when the state of a tick-select box associated with a row changes.

Event-specific data:

Name	Description
selectedEntity	The data entity associated with the selected row.
tickSelected	Boolean value. t*rue* if the tick-select box is currently ticked.
selectedRowIndex	The (zero-indexed) ordinal position of the selected row.
selectedColumnIndex	The (zero-indexed) ordinal position of the selected column.
clickPositionX	The precise pixel x-coordinate of the click or tap action
$rowContainer	A jQuery object representing the DOM element containing the selected row

Methods

addRow

Adds a new row at the foot of the grid. A new instance of the entity type associated with the grid will be created.

Arguments:

Name	Description
setAsSelected	Boolean value. *true* indicates that the newly added row will be set as the currently selected row.

columnVisibility

Alters the visibility of the specified column.

Arguments:

Name	Description
columnIndex	The ordinal position of target column.
isVisible	Boolean value. *true* indicates that the column is to be visible.

getCurrentRowIndex

Returns the ordinal position of the currently selected row.

getCurrentRow

Returns a jQuery object holding the container of the currently selected row.

getCurrentEntity

Returns the data entity instance associated with the currently selected row.

getRowContainer

Returns a jQuery object holding the container of the specified row.

Arguments:

Name	Description
index	The ordinal position of the required row

gridEntities

Returns the collective data entity instance associated with the grid.

setSelectedRow

Sets the specified row as the one currently selected.

Arguments:

Name	Description
targetIndex	The ordinal position of the row to set as currently selected.
suppressScrollTo	Boolean value. *true* indicates that the automatic scroll to the specified row is to be suppressed.

deselectCurrentRow

Deselects the currently selected row.

tickSelectVisibility

Activates/deactivates the tick-select feature of the grid.

Arguments:

Name	Description
isVisible	Boolean value. *true* indicates that the tick-select boxes of the grid are to be displayed; otherwise they will be hidden.
keepSelections	Boolean value. *true* indicates that the default action of resetting all tick-select boxes to an un-ticked state when tick-select is hidden is to be suppressed.

DatePicker

An input widget focused on date entry and display.

ImageButton

A clickable image.

Events

click

Fired when the user clicks or taps on the widget.

Event-specific data:

Name	Description
value	The data associated with the image

LookupSelectSingle

A widget that when clicked/tapped presents a list of user-selectable values, from which one may be selected.

Events

show

Fired when the list of options is displayed.

No event-specific data.

change

Fired when the currently selected option changes. The may be due to programmatic actions or due to end-user UI interaction.

Event-specific data:

Name	Description
selectedEntity	The newly selected data entity.
currentDisplayValue	The value currently being displayed by the widget. Only supplied if this value is available.
currentEntity	The data entity associated with the currently selected value. Only supplied if this value is available.

select

Fired when the list of options is displayed.

Event-specific data:

Name	Description
selectedEntity	The newly selected data entity.
position	Contains string value “before” or “after” indicating the point at which this event has fired; “before” indicates that the activity associated with the select action has not yet been performed.

Methods

setSelectedOption

Allows the option currently selected within the widget to be set.

Arguments:

Name	Description
optionIndex	The ordinal position of the target option.

SwitchPanel

A widget hosting a one-at-a-time display of 2 or more child page segments.

Methods

PanelHide

Hides the specified panel.

Arguments:

Name	Description
panelId	The ordinal position of the panel to be hidden.

PanelShow

Displays the specified panel.

Arguments:

Name	Description
panelId	The ordinal position of the panel to be shown.

TabPanel

A widget hosting a tabbed display of 2 or more child page segments.

Events

preselect

Fired just before a tab (child page segment) is about to be displayed.

Event-specific data:

Name	Description
tabId	The Id of the tab about to be made active.

select

Fired after a tab (child page segment) has been displayed.

Event-specific data:

Name	Description
tabId	The Id of the tab made active.

Methods

setSelectedTab

Sets the currently active tab.

Arguments:

Name	Description
targetTabId	The Id of the target tab.

TextArea

A multi-line text box input.

Events

keyup

Fires on keyboard key up.

Event-specific data:

Name	Description
keyCode	The internal key value associated with the key press.
event	The internal JavaScript event object associated with the event.
value	The current (new) value of the text area widget.

TextBox

A single-line text box input.

Events

clear

Fires when the clear all input button associated with the widget is clicked/tapped.

No event-specific data.

change

Fires when the value displayed by the widget changes.

Event-specific data:

Name	Description
value	The current (new) value of the widget.

focusin

Fires when the widget receives input focus.

Event-specific data:

Name	Description
value	The current value of the widget.

focusout

Fires when the widget loses input focus.

Event-specific data:

Name	Description
value	The current value of the widget.

keyup

Fires on keyboard key up.

Event-specific data:

Name	Description
keyCode	The internal key value associated with the key press.
event	The internal JavaScript event object associated with the event.
value	The current (new) value of the text area widget.

Methods

disable

Sets the widget into a non-editable state.

enable

Resets the widget back into an editable state.

focus

Moves input focus into the widget.

Arguments:

Name	Description
selectAllContent	Boolean value. *true* indicates that all content within the widget is to be selected.
moveToEnd	Boolean value. *true* indicates that the input caret is to be moved to the end of the widget content.

getValue

Returns the current value of the widget.

setValue

Sets the current value of the widget.

Arguments:

Name	Description
Value	The new value for the widget.