

TMS TableView for FireMonkey DEVELOPERS GUIDE

May 2012
Copyright © 2012 by tmssoftware.com bvba
Web: http://www.tmssoftware.com
Email: info@tmssoftware.com

Index

Availability 4
Description 4
List of available controls 5
TMS TableView control 7
- Architecture 7
- Styling 9
- Properties / Methods / Events 10
- Item storage and buffering 15
- Adding and removing items 16
- Sorting 17
- Categories 18
- Lookup 21
- Filtering / Searching 22
- Editing / Deleting 23
- DetailView 24
- Layout 26
- User interface interaction with the TableView 29
- MultiSelect 29
- Additional Item Elements 30
- Performance 31
- Binding Controls 32
- LiveBindings 34
- HTML support 36

TMS Popup control 38
TMS Mini HTML rendering engine 40
General FireMonkey component usage guidlines 43
- Visual part 43
- Non-visual part 43
- Naming convention 43
- Styling 43
- Components 47
- Cross-platform deployment of applications with TMS TableView for FireMonkey components 48

Samples 50
TMS TableView Overview Demo 50
TMS TableView LiveBindings Demo 1 & 2 52
FireMonkey Styles 58

Availability

TMS TableView for FireMonkey is a component that is suitable for cross platform development with the Embarcadero FireMonkey framework and is designed for use with Win32, Win64, OSX32 and iOS operating systems.

versions:

TMS TableView for FireMonkey is available for Delphi XE2 & C++Builder XE2 update 4 and newer releases.

Description

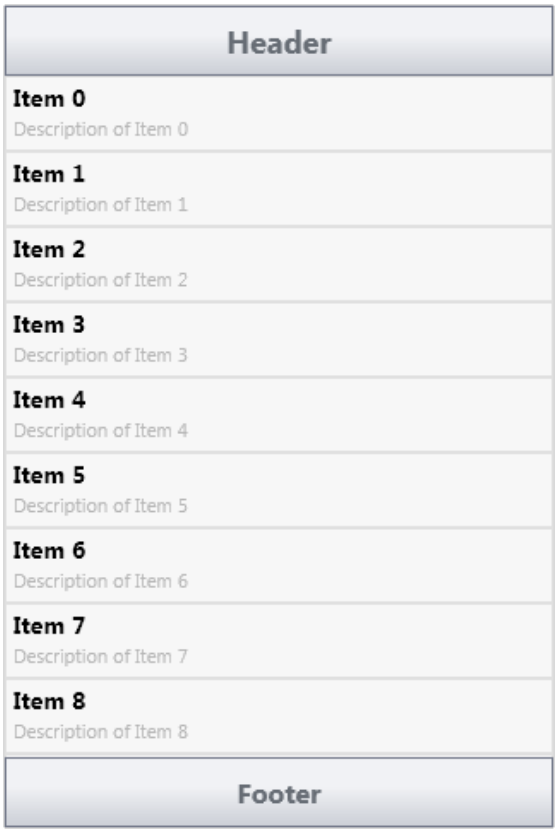
The TMS TableView control TTMSFMXTableView is a smooth scrolling list based on the concept of the iOS UITableView control and is used for displaying and editing hierarchical lists of information, allowing the user to scroll, select, view details, and delete items...

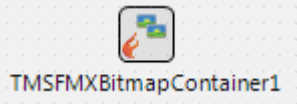
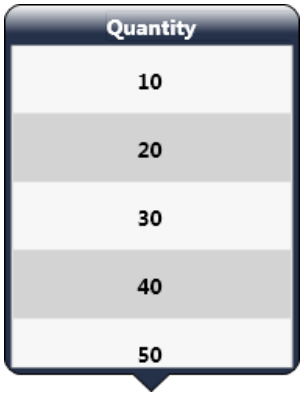
The TableView can be configured to show custom, alphabetic or alphanumeric sections with direct alphabetical lookup. It also supports filtering / searching with a separate designed searchedit control. The TableView implements a basic, default layout but is fully customizable via the FireMonkey styles architecture.

In this developers guide an overview of the TTMSFMXTableView control and controls used inside the TableView is given, features are highlighted and it is explained how to work with them and how they are designed.

IMPORTANT NOTICE:
If the FireMonkey framework is new to you, please see the chapter “General FireMonkey component usage guidelines” that offers an introduction that is recommended to read before you start working with the TMS TableView component. Another interesting source of information is http://docwiki.embarcadero.com/RADStudio/en/FireMonkey_Application_Platform

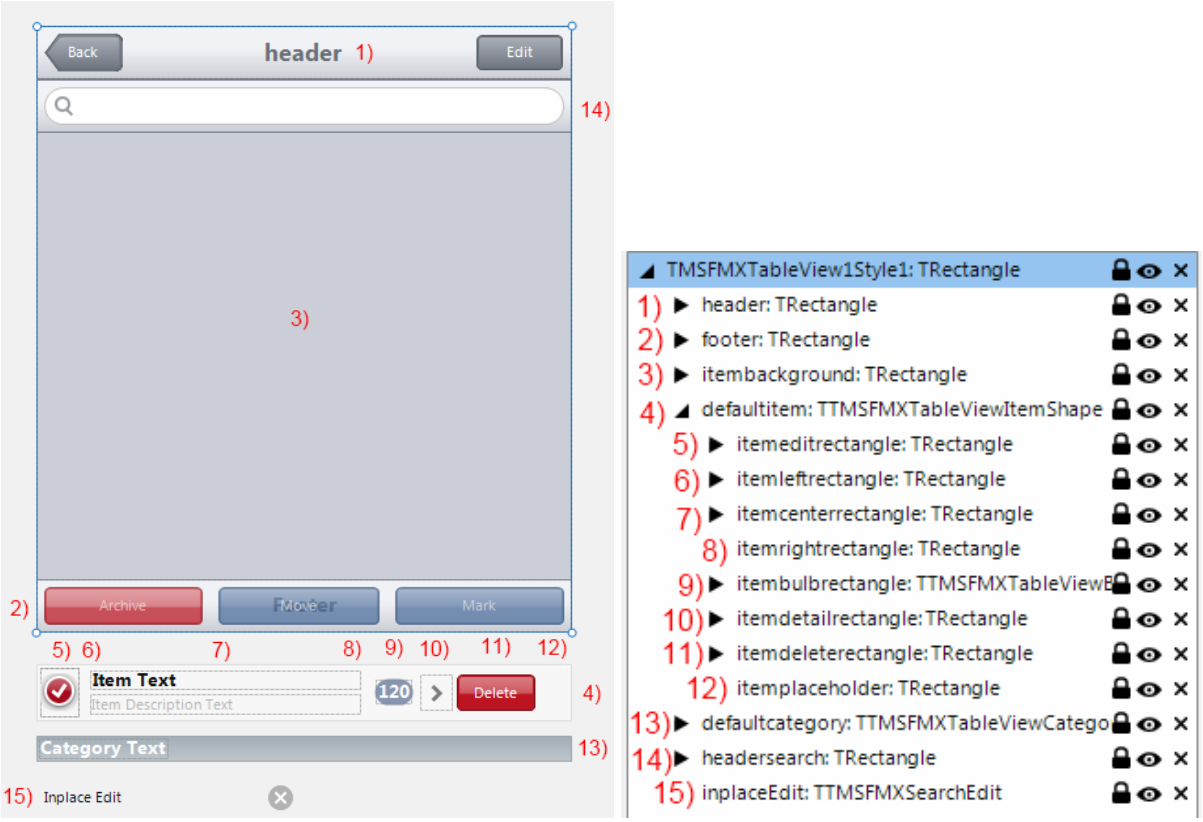
List of available controls

TTMSFMXTableView The main component.	
TTMSFMXHTMLText Text shape that supports HTML (see **MiniHTML chapter**)	
TTMSFMXBitmap Bitmap shape which can load and display an image directly or through a BitmapContainer.	

TTMSFMXBitmapContainer Container that holds multiple bitmaps	
TTMSFMXSearchEdit Inherits from TEdit and adds an optional clear and search button, and has a rounded appearance.	
TTMSFMXBarButton Inherits from TButton and adds a Layout and Kind property for different appearances used in the TableView.	
TTMSFMXPopup Component that allows displaying any type of control inside a customizable popup dialog.	

TMS TableView control

Architecture



1) **Header**: a rectangle shape with a caption and that can hold optionally a backbutton and an editbutton. The backbutton is used to return from the detail view and the edit button is used to put the tableview in edit mode.

2) **Footer**: a rectangle shape with a caption. When the TableView is in edit mode, the 3 buttons (Archive, Move and Mark) are displayed in this footer rectangle.

3) **BackGround**: the background is a placeholder for the items that also enables scrolling of items and keyboard handling.

4) **DefaultItem**: the basic default layout of the item. When adding items, the layout is copied (cloned) for each item.

5) **Edit Rectangle**: the rectangle that is displayed when the TableView is in edit mode.

6) **Left Rectangle**: a placeholder rectangle that can contain other elements, such as, but not limited to, a checkbox, radiobutton or image.

7) **Center Rectangle**: the rectangle that contains the Caption and Description of the item.

8) **Right Rectangle**: the right rectangle has the same purpose as the left rectangle, but is right aligned.

9) **Bulb Rectangle**: a custom designed shape that can display text in a rounded shaped appearance.

10) **Detail Rectangle**: a rectangle that contains a predefined detail image and is enabled when an item has a DetailView assigned.

11) **Delete Rectangle**: a rectangle that contains a delete button and is enabled when swiping on the item.

12) **PlaceHolder Rectangle**: element internally used to position the elements with extra spacing when the lookupbar / scrollbar are enabled.

13) **Default Category**: the basic layout of a category. When adding items, the category is automatically added according to the kind of item.

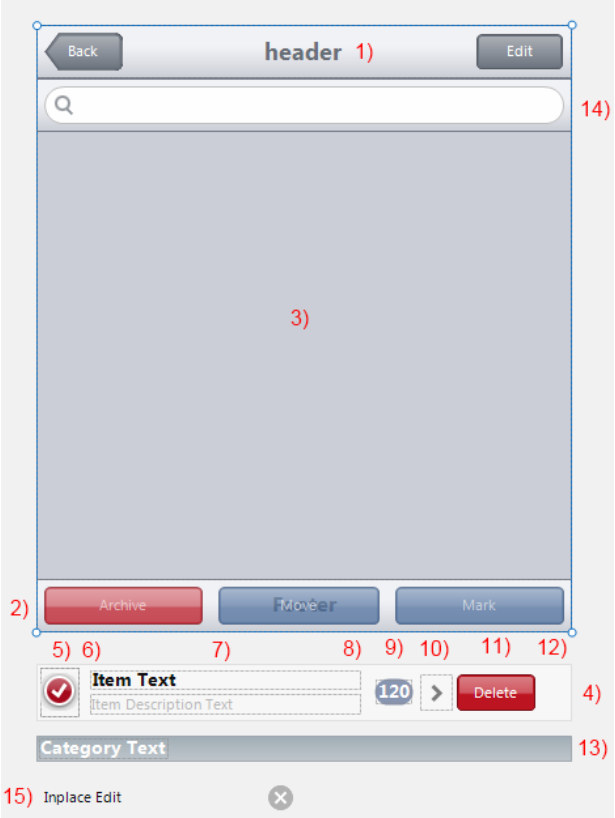
14) **Header Search**: a search edit box displayed when filtering or search is enabled.

15) **Inplace Edit**: search edit box that is transformed to work as an inplaced editor and is displayed when clicking on the caption or description when the appropriate properties are set to enable editing.

Styling

With the FireMonkey design philosophy in mind, we have made the TableView completely styleable. When editing the custom or default style when right-clicking on the component (See: General Firemonkey component usage guidelines) the basic TableView layout is defined with several styleable elements. Elements can be removed, added and modified and updates are reflected in the component when applying the edited style.

Programmatically, almost every element is accessible with a function. When you want to style an element programmatically, you can use the appropriate function. Below is an overview of each element that is styleable at designtime and at runtime:

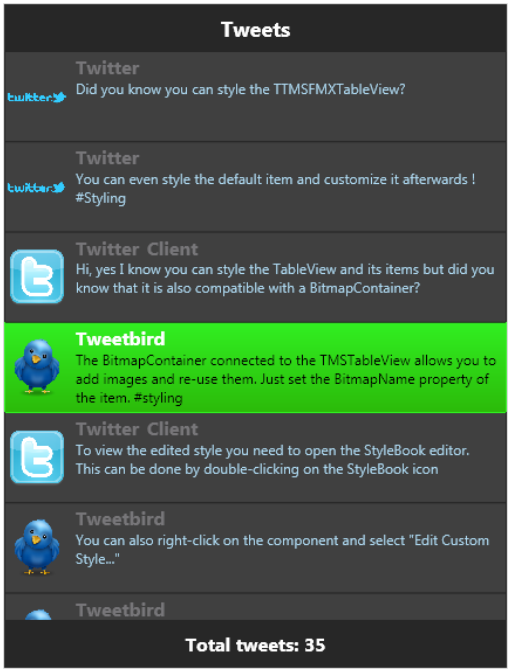


1) `TTMSFMXTableView.GetHeaderRectangle`
2) `TTMSFMXTableView.GetFooterRectangle`
3) `TTMSFMXTableView.GetListBackGround &`
   `TTMSFMXTableView.GetListContainer`
4) `TTMSFMXTableView.GetDefaultItem & TTMSFMXTableViewItem.Shape`
5) `TTMSFMXTableViewItem.ShapeEditRectangle`
6) `TTMSFMXTableViewItem.ShapeLeftRectangle`
7) `TTMSFMXTableViewItem.ShapeCaption &`
   `TTMSFMXTableViewItem.ShapeDescription &`
   `TTMSFMXTableViewItem.ShapeCenterRectangle`
8) `TTMSFMXTableViewItem.ShapeRightRectangle`
9) `TTMSFMXTableViewItem.ShapeBulbRectangle`
10) `TTMSFMXTableViewItem.ShapeDetailRectangle`
11) `TTMSFMXTableViewItem.ShapeDeleteRectangle`
12) `TTMSFMXTableViewItem.ShapePlaceHolder`

13) `TTMSFMXTableView.GetDefaultCategory & TTMSFMXTableViewCategory.Shape`
14) `TTMSFMXTableView.GetSearchEdit`
15) `TTMSFMXTableView.GetInplaceEdit`

These are the most important functions that return the element that is available in the default style or the style of the item when it is cloned. There are more functions available specific for the item and the tableview such as the deletebuttonshape, bulbshape, checked and uncheckedimage and default animations on the backbutton. A complete list can be found when typing the name of the component and search through all the "Get*" functions and for the Tableview Item the functions that have "Shape" in the name.

Below is a sample that shows the power of styling. This sample is also available as a separate demo.



Properties / Methods / Events

Below is a list of properties, methods and events in alphabetic order that expose the core functionality of the component and that need a short introduction before delving into the details of the component.

TTMSFMXTableView published properties

- **ArchiveText**: The text of the left button that is displayed in the footer when in edit mode.
- **AutoDeleteItem**: deletes the item automatically.
- **AutoFilter**: When filtering or searching is enabled, the filtering is applied automatically.
- **AutoLoadBuffer**: When buffering is used (BufferSize > 0) the list automatically loads the next buffer when scrolling in the list.
- **AutoLookup**: When using the lookupbar, the list automatically displays the correct category.
- **AutoToggleDetail**: When a DetailView is assigned and the item is clicked. The DetailView is automatically displayed with animation.
- **BackButton**: Displays a back button in the left corner of the header when the DetailView is displayed.

- **BitmapContainer**: Support for BitmapContainer, which is able to contain a collection of reusable TBitmap items.
- **BufferSize**: Used to load an initial buffer of the total collection of items. When scrolling the buffer is repositioned to load the next BufferSize of items.
- **Categories**: Collection of custom categories, used when the CategoryType is set to ctCustom.
- **CategoryType**: The type of categories that are used. A choice can be made between alphabetic, alphanumeric or custom.
- **DefaultDetailView**: The Default detail view that is used by all items if the DetailView property of the item is nil.
- **EditButton**: Displays an edit button in the right corner of the header. This button can be used to set the TableView in edit mode. This button is also used in combination with filtering / searching. In this mode the button is switched to a Cancel button.
- **Filtering**: the type of filtering the list applies. In combination with the search edit box, the vfFilterStart and vfFilterRandom option creates a subset of items and vfSearch highlights the item that matches the search string.
- **FooterText**: The text in the footer.
- **HeaderText**: The text in the header.
- **ItemOptions**: Enables or disables cloning of certain elements that are present in the default item. The performance of the list can be increase when using in combination with the BufferSize property.
- **Items**: a collection of TableView items.
- **LayoutMode**: defines the layout of the item list. The lmNormal option displays the list as a normal TableView and the lmGroup displays the items in grouped mode. The grouped mode used the GroupIndex per item to create groups.
- **LookupBar**: Enables or disables the lookupbar.
- **MarkText**: The text of the right button that is displayed in the footer when in edit mode.
- **MoveText**: The text of the center button that is displayed in the footer when in edit mode.
- **MultiSelect**: Enables or disables multiselect on items.
- **ScrollIndicator**: Shows or hides the scrollindicator that visualizes the amount of items in the collection with an optional fading animation.
- **SelectedItemIndex**: The index of the item that has been selected.
- **ShowFilter**: Show the search edit box on top of the list which can be used to apply filtering or highlight an item in the list.
- **ShowFilterOnSwipe**: Enables or disables the ability of showing the search edit box

Category item published properties in Categories collection

- **Caption**: The Caption displayed in the category.
- **ID**: To link items to the specific category.
- **LookupText**: The text displayed in the lookupbar.

Item published properties in Items collection

- **Bitmap**: The bitmap displayed in the left rectangle image when enabled in the ItemOptions.
- **BitmapName**: The name of the bitmap displayed in the left rectangle image when enabled in the ItemOptions and when a BitmapContainer is assigned and contains an image with the correct name.
- **BulbText**: The text of the bulb rectangle, that is displayed when enabled in the ItemOptions.
- **CanDelete**: When CanDelete is false, the deletebutton will not be displayed and the item cannot be deleted.
- **CanEditCaption**: Displays an inplace editor when clicking on the caption.
- **CanEditDescription**: Displays an inplace editor when clicking on the description.
- **CanSelect**: Enables selection of an item.
- **Caption**: The caption of the item.

- **CategoryID**: The id of the category linked to the item when displaying custom categories.
- **DataBoolean**: Boolean property that is linked to a control placed in the right rectangle. This property also triggers the OnItemData event.
- **DataGroupIndex**: The index that is used in combination with a radiobutton, to create a RadioButtonGroup.
- **DataObject**: Has the same functionality as the DataBoolean property but of type TObject.
- **DataString:** Has the same functionality as the DataBoolean property but of type String.
- **DataValue**: Has the same functionality as the DataBoolean property but of type Single.
- **DeleteButton**: Displays the deletebutton.
- **Description**: The description of the item.
- **DetailView**: The view that is displayed when clicking on an item, to navigate to the detail.
- **GroupIndex**: The index that is used to group items in the lmGroup layoutmode.
- **LeftMargin**: The margin / indenting of an item from the left side.
- **RightMargin**: The margin / indenting of an item from the right side.
- **Selected**: Sets the item selected.
- **Tag**: Tag property that can be used to hold additional information.
- **Visible**: Shows or hides the item.

Public procedures, functions, properties

- ```
  property TopItem: Integer;
  ```
  Sets the first visible item in the list.

- ```
  function SelectedItem: TTMSFMXTableViewItem;
  ```
  Returns the selected item.

- ```
  function CountSelected: Integer;
  ```
  Returns the count of selected items.

- ```
  function ItemFromDifferentCategory(item1, item2:
  TTMSFMXTableViewItem): Boolean;
  ```
  Returns whether the first item is from a different category of the second item.

- ```
  function ItemAtXY(X, Y: Single): TTMSFMXTableViewItem;
  ```
  Returns the Item at a specific position in the TableView.

- ```
  function GetCharacterForItem(AItem: TTMSFMXTableViewItem): String;
  ```
  Returns the Character of the item that is used when categories are applied.

- ```
  function FindItemWithFilter(AFilter: String): TTMSFMXTableViewItem;
  ```
  Returns the item with a specific filter applied.

- ```
  function IsItemInItemList(AItem: TTMSFMXTableViewItem): Boolean;
  ```
  Returns if the item is loaded in the list (depending on the BufferSize and scrollposition)

- ```
  function IsItemVisibleInList(AItem: TTMSFMXTableViewItem): Boolean;
  ```
  Returns if the item is visible in the list.

- ```
  function isEditMode: Boolean;
  ```
  Returns if the edit mode is active or inactive.

- ```
  function isDetailMode: Boolean;
  ```
  Returns if a detail view is visible or not visible.

- ```
  function GetCurrentListView: TControl;
  ```

Returns the current view in edit mode, the main list control or the detail list control.

- `function FindFirstItemWithCategory(ACategory: String): TTMSFMXTableViewItem;`
  Returns the first item with a specific Category when using AlphaBetic / AlphaNumeric categories.

- `function FindFirstItemWithCategoryID(ACategoryID: Integer): TTMSFMXTableViewItem;`
  Returns the first item with a specific CategoryID when using custom categories.

- `procedure ToggleEditMode;`
  Toggles between edit mode and normal mode.

- `procedure ApplyFilter(AFilter: String);`
  Applies and automatically updates the list with a specific filter.

- `procedure UpdateTableView;`
  Forces a complete update of the list of items in the TableView.

- `procedure BeginUpdate;`
  Blocks an update of the TableView. This is necessary when adding a large amount of items. Used in combination with EndUpdate;

- `procedure EndUpdate;`
  Blocks an update of the TableView. This is necessary when adding a large amount of items. Used in combination with BeginUpdate;

- `procedure LoadNextBuffer(ABufferSize: Integer = -1);`
  Loads the next list with specific BufferSize. (ABufferSize = -1 loads the default BufferSize).

- `procedure LoadPreviousBuffer(ABufferSize: Integer = -1);`
  Loads the previous list with specific BufferSize. (ABufferSize = -1 loads the default BufferSize).

- `procedure EditMode;`
  Sets the list in edit mode.

- `procedure CancelEditMode;`
  Cancels the edit mode. Sets the list back to normal mode.

- `procedure ShowDetailView;`
  Shows the detail view of the item.

- `procedure HideDetailView;`
  Hides the detail view of the item.

- `procedure LoadBufferBetween(BufferStart, BufferStop: Integer);`
  Loads the list between a start and stop buffer.

- `procedure LookupCustomCategory(ACategoryID: Integer);`
  Automatically positions the category on top in the list with a specific CategoryID. This can be used in combination with custom categories.

- `procedure LookupCategory(ACategory: String);`

Automatically positions the category on top in the list with a specific Category. This can be used in combination with AlphaBetic / AlphaNumeric categories.

- `procedure LookupItem(AItemIndex: Integer; SelectItem: Boolean = False);`
  Automatically positions the item on top in the list with a specific index, and the possibility to select that item.

- `procedure DeSelectAllItems;`
  Deselects all items.

- `procedure SelectAllItems;`
  Selects all items that are loaded in the list.

- `procedure SelectItemsBetween(AStartIndex, AStopIndex: Integer; AddToSelection: Boolean = False);`
  Selects a subset of items between a StartIndex and StopIndex. AddToSelection parameter adds the subset to an internal list, when AddToSelection is false, the list is clear before adding the new subset.

- `procedure SelectItems(Selection: Array of Integer; AddToSelection: Boolean = False);`
  Selects a subset of items for which the item indexes are passed as an array of integers. This procedure works in the same way as the subset between a StartIndex and StopIndex.

Published events

- **OnAfterFilter**: Event called after Filtering is applied.
- **OnAfterSearch**: Event called after Seaching is applied.
- **OnAnimationCustomize**: The transition between the list and the detailview is animated. These 2 list animations can be customized through this event.
- **OnApplyStyleLookup**: When editing the default or custom Style through the StyleBook, this event is called after applying the style. When starting the application, modifications can be made in this event to the style that is either the default style or the edited style from the StyleBook.
- **OnArchiveClick**: Event called when clicking on the ArchiveButton, which is displayed when the TableView is set in EditMode.
- **OnBackButtonClick**: When the TableView is in edit mode, the BackButton is enabled. The OnBackButtonClick event is triggered when clicking on this button.
- **OnBeforeFilter**: Event called before Filtering is applied.
- **OnBeforeSearch**: Event called before searching is applied.
- **OnCategoryAnchorClick**: Event called when clicking on an anchor when using HTML in the Category.
- **OnCategoryClick**: When using categories and the lookupbar is enabled, the OnCategoryClick event is called when clicking on a custom category or an alphanumeric character.
- **OnCategoryCustomize**: When using categories, this event is called after creating a clone of the default category that is available in the default TableView style. Modifications on appearance and functionality can be made through this event.
- **OnHeaderAnchorClick**: Event called when clicking on an anchor when using HTML in the Header.
- **OnFooterAnchorClick**: Event called when clicking on an anchor when using HTML in the Footer.
- **OnItemAfterDetail**: Event called when the animation is completed that animates the detailview after clicking on the item.

- **OnItemAfterDraw**: Event called after the item is drawed. This event can be used for custom drawing on top of the item.
- **OnItemAfterReturnDetail**: Event called when the animation is completed that animates the main list view back to the original state after clicking on the backbutton.
- **OnItemBeforeDetail**: Event called before the animation is started when clicking on an item to animate the DetailView.
- **OnItemBeforeDraw**: Event called before the item is drawn. This event can be used to hide the background and perform custom drawing.
- **OnItemBeforeReturnDetail**: Event called before the animation is started when animating the main list back to the original state after clicking on the backbutton.
- **OnItemCaptionAnchorClick**: Event called when clicking on an anchor when using HTML in the Item Caption.
- **OnItemClick**: Event called when clicking on an item.
- **OnItemCompare**: Event called when sorting the items. Custom sorting can be applied to the items collection.
- **OnItemCustomize**: This event is called after creating a clone of the default item that is available in the default TableView style. Modifications on item appearance, interaction and functionality can be made through this event.
- **OnItemData**: Event called when modyfing one of the Data* properties of an item.
- **OnItemDescriptionAnchorClick**: Event called when clicking on an anchor when using HTML in the Item Description.
- **OnItemDelete**: Event called when deleting an item.
- **OnItemSelected**: Event called when selecting an item.
- **OnManualLoadNextBuffer**: When AutoLoadBuffer is false, this event is called when the list tries to load the next buffer.
- **OnManualLoadPreviousBuffer**: When AutoLoadBuffer is false, this event is called when the list tries to load the previous buffer.
- **OnManualLookup**: When AutoLookup is false, this is event is called when the lookupbar is used to lookup a category.
- **OnMarkClick**: Event called when clicking on the mark button which is displayed when the TableView is set in EditMode.
- **OnMoveClick**: Event called when clicking on the move button which is displayed when the TableView is set in EditMode.
- **OnScroll**: Event called when scrolling in the list.
- **OnScrollFinished**: Event called when the scrolling is finished.

Item storage and buffering

The TableView implements a specific way of loading / displaying items. The design choices are driven by making the TableView performant and resource friendly. Items are stored in a TCollection. When the component is first made visible on a form, from this collection a displaylist of item shapes that are cloned from the default item shape are automatically created. Only a small number of items are created initially. The other remaining items remain in the collection waiting to be shown in the control by means of the generation of display items (shapes). This loading process is necessary to make sure the TableView offers a good performance on various operating systems. In other words, there is no speed impact when adding 10 items or 10000 items to the Items collection. Displayed items are only on demand and incrementally created. This loading / displaying process is controlled by a single property: BufferSize.

The BufferSize property is set, by default, to 50. This means that only for the first 50 items display items are created. While scrolling in the list, the list automatically creates display items for the next 50 items. This buffering method increases performance if balanced correctly. Setting the BufferSize to 0 will create display items at once for all items. Loading 10000 items with BufferSize set to 0 will have an initial large performance hit as the creation of display items is a resource intensive process (FireMonkey framework limitation) Setting BufferSize to 50 is good balance.

If the BufferSize is set, the AutoLoadBuffer property will create the display items automatically when scrolling beyond the end of the list. When AutoLoadBuffer is set to False, the TableView will not create the display items automatically but instead the appropriate events will be triggered. Creating display items can then be done manually.

OnManualLoadNextBuffer: Event triggered if AutoLoadBuffer is false, BufferSize > 0 and the list is scrolled to the end.

OnManualLoadPreviousBuffer: Event triggered if AutoLoadBuffer is false, BufferSize > 0 and the list is scrolled to the beginning.

When implementing these events, loading the buffer can be done with:

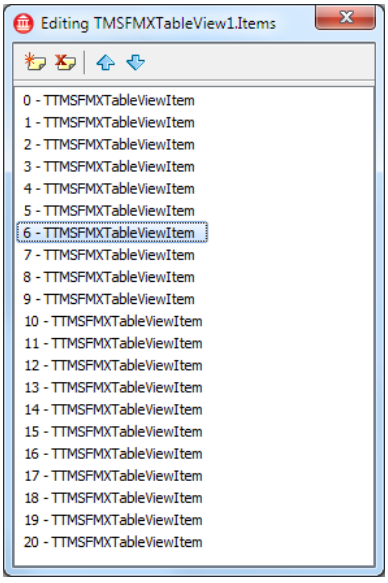
```
TMSFMXTableView1.LoadNextBuffer();
TMSFMXTableView1.LoadPreviousBuffer();
```

These two methods have a parameter with which you can decide how many display items are loaded. If no parameter is passed the loading will use the BufferSize property.

It's a matter of experimenting with the starting BufferSize, the buffersize that is loaded next and the quantity of the items in the collection to have the best performance.

Adding and removing items

After dropping a TableView component (TTMSFMXTableView) on the form you will notice that, by default, the TableView has some items already added. Adding items can be done at designtime and at runtime. Click on the component to view the Items property. This is a default collection editor and can be used to add or remove items.



Items can also be added programatically:

```
var
  it: TTMSFMXTableViewItem;
  i: Integer;
begin
```

```
  TMSFMXTableView1.BeginUpdate;
  for I := 0 to 100 do
  begin
    it := TMSFMXTableView1.Items.Add;
    it.Caption := 'Item ' + inttostr(I);
    it.Description := 'Hello World !';
  end;
  TMSFMXTableView1.EndUpdate;
```

Note that all calls to the TMSFMXTableView must include a BeginUpdate and EndUpdate statement. Whenever an item is added, the display list is rebuilt. With a BeginUpdate and EndUpdate statement the update process is blocked and then executed once.
To remove an item programmatically, just call the item's destructor.

Sorting

After adding items, sorting can be applied to the collection. This will be done alphabetically by default, but can be customized with the OnItemCompare event.

If no sorting is applied, the items are displayed in the order that they are added in the collection.

```
var
  it: TTMSFMXTableViewItem;
  i: Integer;
begin
  TMSFMXTableView1.BeginUpdate;

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Pear';
  it.Description := 'This is a Pear';

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Banana';
  it.Description := 'This is a Banana';

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Apple';
  it.Description := 'This is an Apple';

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Lemon';
  it.Description := 'This is a Lemon';

  TMSFMXTableView1.EndUpdate;
end;
```



When calling `TMSFMXTableView1.Items.Sort;` the Items are sorted alphabetically. Two optional parameters can be passed to sort case sensitive (default) or incase sensitive and ascending (default) or descending:

TableView.



Sorting is important for the next paragraph in this manual: Categories.

Categories

In the TableView there are 4 types of categories:

- AlphaBetic
- AlphaNumericFirst (Numbers are added to the lookupbar and are displayed first)
- AlphaNumericLast (Numbers are added to the lookupbar and are displayed last)
- Custom

These types can be set with the CategoryType property. With the AlphaBetic / AlphaNumeric categories, the TableView automatically searches and adds categories with the first letter of the item caption. The range that is used to display categories in AlphaBetic mode is A…Z and AlphaNumeric adds 0…9. In the sample below the items are sorted and the CategoryType property is set to ctAlphaBetic.

```
var
  it: TTMSFMXTableViewItem;
  i: Integer;
begin
  TMSFMXTableView1.BeginUpdate;
  TMSFMXTableView1.CategoryType := ctAlphaBetic;

  it := TMSFMXTableView1.Items.Add;
```

```
  it.Caption := 'Pear';
  it.Description := 'This is a Pear';

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Banana';
  it.Description := 'This is a Banana';

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Apple';
  it.Description := 'This is an Apple';

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Lemon';
  it.Description := 'This is a Lemon';

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Apple 2';
  it.Description := 'This is an Apple 2';

  TMSFMXTableView1.Items.Sort;

  TMSFMXTableView1.EndUpdate;
end;
```



When using categories, the LookupBar on the right side of the control is enabled and displayed by default. The LookupBar can be used to navigate directly through the categories. This is explained in the lookup paragraph.

If the CategoryType is set to ctAlphaBetic / ctAlphaNumericFirst or ctAlphaNumericLast the categories are predefined. You can use custom categories by setting the CategoryType to ctCustom and adding your own categories. Custom categories works with linking the CategoryID property of the item to the ID of the category.

```
var
  it: TTMSFMXTableViewItem;
  cat: TTMSFMXTableViewCategory;
  i: Integer;
begin
  TMSFMXTableView1.BeginUpdate;
  TMSFMXTableView1.CategoryType := ctCustom;

  cat := TMSFMXTableView1.Categories.Add;
  cat.Id := 0; // all items with CategoryID 0 belong to this category
  cat.Caption := 'Category 1';
  cat.LookupText := 'Cat 1';

  cat := TMSFMXTableView1.Categories.Add;
  cat.Id := 1; // all items with CategoryID 0 belong to this category
  cat.Caption := 'Category 2';
  cat.LookupText := 'Cat 2';

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Pear';
  it.Description := 'This is a Pear';
  it.CategoryID := 0;

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Banana';
  it.Description := 'This is a Banana';
  it.CategoryID := 0;

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Apple';
  it.Description := 'This is an Apple';
  it.CategoryID := 1;

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Lemon';
  it.Description := 'This is a Lemon';
  it.CategoryID := 1;

  it := TMSFMXTableView1.Items.Add;
  it.Caption := 'Apple 2';
  it.Description := 'This is an Apple 2';
  it.CategoryID := 1;

  TMSFMXTableView1.Items.Sort;

  TMSFMXTableView1.EndUpdate;
end;
```



As result you can see the 2 categories in the list and the corresponding lookuptext for each category in the lookupbar. Sorting can also be applied on categories in the same way as sorting is applied to items. The categories are then sorted alphabetically and ascending based on the caption.

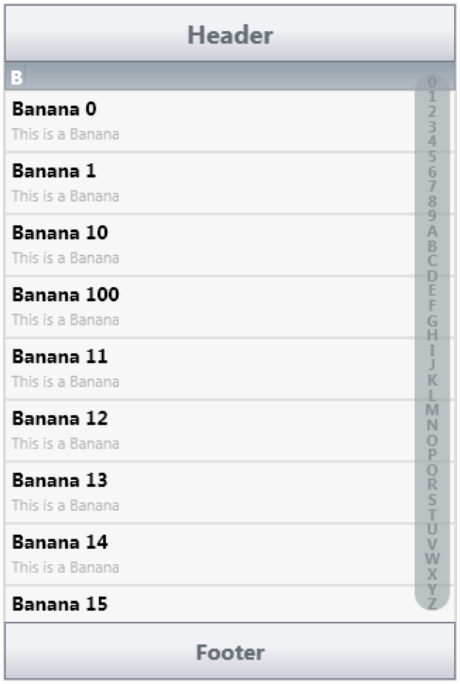
Lookup

After activating categories, the (optional) lookupbar is automatically displayed. Clicking and dragging along the lookupbar will automatically perform a lookup. This can be disabled with the AutoLookup property. If the AutoLookup property is false, the OnManualLookup event is triggered when a category is clicked. From there you can perform a lookup with one of the two methods below depending on the type of categories you have chosen:

```
  TMSFMXTableView1.LookupCategory(ACharacter);
  //Performs a lookup of an AlphaBetic / AlphaNumeric category with a
specific character.

  TMSFMXTableView1.LookupCustomCategory(ACharacterID);
  //Performs a lookup of a Custom category with a specific characterID.
```

When navigating through the categories the lookupbar will, when necessary, load the buffer that is needed to display the selected category. Depending on the BufferSize and the active buffered items the lookup will be either instant or could have a small loading time. Adapt the BufferSize to an optimal value depending on the device on which the application will run.



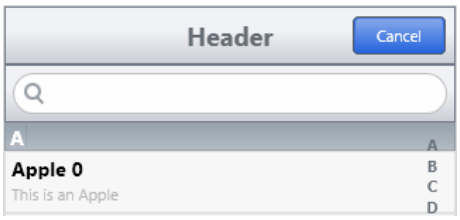
Filtering / Searching

The TableView supports Filtering or Searching. Filtering only shows the items that match the search text while searching highlights the first item found. Similar to the operation of the lookupbar, this way of navigating through items automatically takes care of loading the correct buffer that is needed to display the matched items.

Filtering / Searching can be enabled in two ways: either programmatically with the ShowFilter:Boolean property or by swiping the list down. The last method is optional and can be configured with the ShowFilterOnSwipe property. To programmatically show the filter entry edit control use:

```
TMSFMXTableView1.ShowFilter := True;
```

After setting ShowFilter to true or swiping the list down, the search edit box appears.



The search edit box can be hidden by clicking the cancel button in the TableView header.
By default, when filtering is performed, the TableView only shows the items that match the string in the edit box.
The filter method can be changed by setting the Filtering property and can be set to vfFilterStart, vfFilterRandom, vfSearch or vfNone.

- **vfFilterStart**: Shows the items that match the string from the beginning of the text in the edit box.

- **vfFilterRandom**: Shows the items that contains the filter in the text, not necessarily from the beginning of the text.
- **vfSearch**: Highlights the first item that matches the string in the edit box.
- **vfNone**: No filtering or searching is applied.

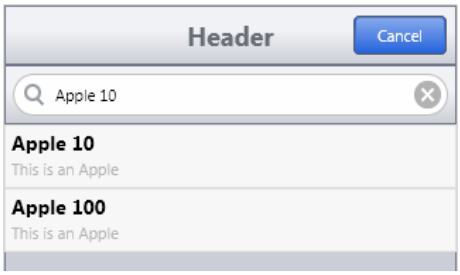
The filtering or searching is automatically performed everytime the text in the search edit box is changed. With the AutoFilter property set to false, this is not longer the case. When AutoFilter is false, the filtering is applied by pressing the Enter key.

Following events are triggered when performing filtering / searching:

OnBeforeFilter, OnBeforeSearch, OnAfterFilter and OnAfterSearch

Via these events, custom filtering or custom searching can be done. To perform a manual filtering you can use the ApplyFilter method:

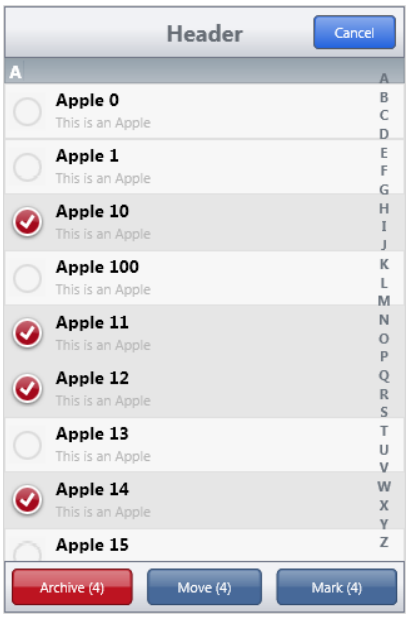
```
TMSFMXTableView1.ApplyFilter('Filter');
```



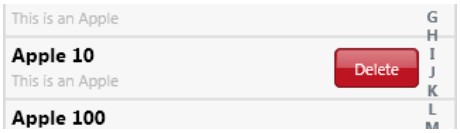
Editing / Deleting

Editing can be enabled with the EditButton: Boolean property. When clicking the edit button, the TableView is set in edit mode. This automatically animates checkboxes from the left and adds buttons in the footer of the TableView. Clicking on the items automatically checks the checkbox and updates the buttons. The text for these editing related buttons can be set with the MarkText, ArchiveText and MoveText properties.





Clicking on the Archive Button will delete the selected items from the list. Each button has a separate event that can be used to perform different operations. Deleting an item can also be done from a delete button associated with the item. The delete button is made visible by swiping from right to left on the item or programmatically:

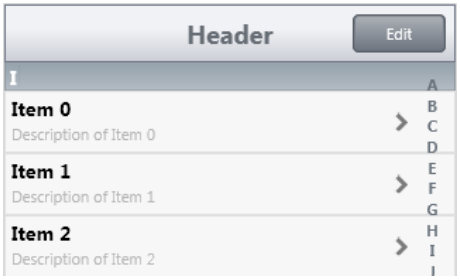


Clicking on the delete button will delete the item. The automatic delete functionality in the deletebutton and the archive button can be disabled with the AutoDeleteItem property.

Note: Editing and searching are mutually exclusive functions. When searching is enabled the edit functionality is disabled and vice versa.

DetailView

The TableView has support for showing a Detailview. The detail view can be any type of component that descends from TControl. Each item in the TableView can have its own DetailView or all items can share the same DetailView (DetailView property on item level versus DefaultDetailView on TableView level). If the DetailView is assigned, the item is marked with a default detail arrow image. If the DetailView property is nil, the item automatically looks for the DefaultDetailView control that can be assigned on the TableView level.

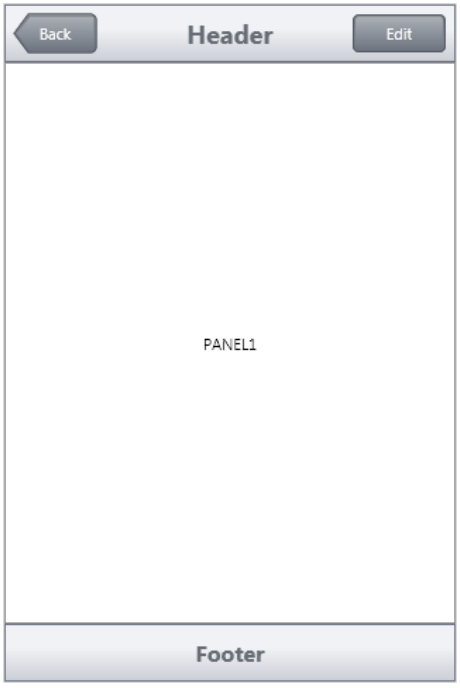


Clicking on the item that has a DetailView assigned automatically slides the main items list out of view and slides in the DetailView. An extra back button, controlled by the TableView BackButton property can be optionally shown and when clicked, it will slide out the DetailView and slide in the regular main items list.

In this sample, we have placed three panels on the form and assigned each panel to a different item:

```
TMSFMXTableView1.Items[0].DetailView := Panel1;
TMSFMXTableView1.Items[1].DetailView := Panel2;
TMSFMXTableView1.Items[2].DetailView := Panel3;
```

Clicking on the first item shows Panel1, the second item Panel2 and the third item Panel3.



The automatic toggle between the main list and the detail list can be disabled by setting AutoToggleDetail: Boolean to false. From the OnItemClick or OnItemSelected event handler you can use the ToggleDetailView() method to show the assigned DetailView of the item. With the OnBackButtonClick event you can programmatically return back to the main list.

```
TForm1.TMSFMXTableView1BackButtonClick(Sender: TObject;
  AItem: TTMSFMXTableViewItem);
```

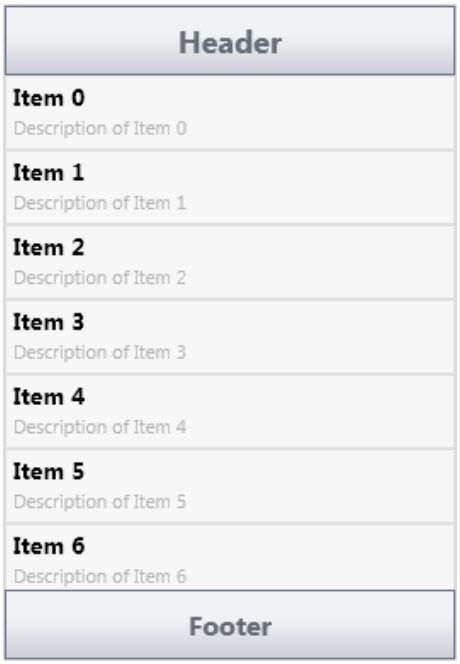
```
begin
  AItem.ToggleDetailView;
end;

TForm1.TMSFMXTableView1ItemClick(Sender: TObject;
  AItem: TTMSFMXTableViewItem);
begin
  AItem.ToggleDetailView;
end;
```

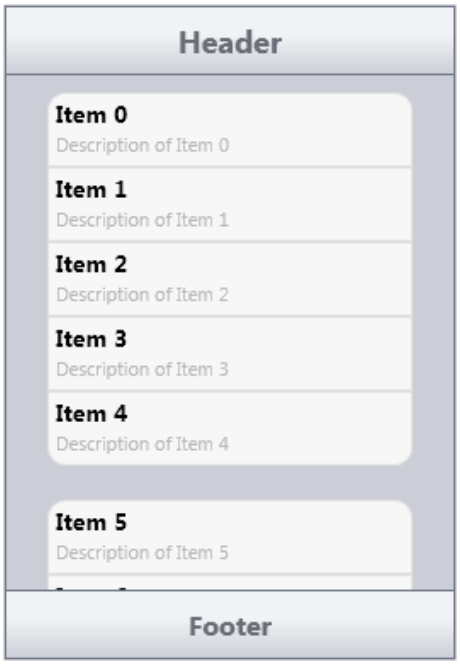
Layout

The TableView has 2 ways of displaying the list. The list can either be displayed in a normal view (default) or in a grouped view. This can be selected with the TTMSFMXTableView.LayoutMode property.

Normal view:



Grouped view:



Items are assigned to a group (Category) via the Item's CategoryID property. When using categories, multiple items can be assigned to a group (Category).

This sample code snippet initializes two groups with each two items in the TableView:

```
var
  itm: TTMSFMXTableViewItem;
begin
  TMSFMXTableView1.BeginUpdate;
  TMSFMXTableView1.Items.Clear;

  TMSFMXTableView1.Categories.Clear;
  // group 0
  TMSFMXTableView1.Categories.Add.Caption := 'Internet settings';
  // group 1
  TMSFMXTableView1.Categories.Add.Caption := 'Email settings';

  TMSFMXTableView1.CategoryType := ctCustom;

  itm := TMSFMXTableView1.Items.Add;
  itm.CategoryID := 0;
  itm.GroupIndex := 0;
  itm.Caption := 'Proxy';
  itm.Description := 'Proxy server name';

  itm := TMSFMXTableView1.Items.Add;
  itm.CategoryID := 0;
  itm.GroupIndex := 1;
  itm.Caption := 'IP address';
  itm.Description := '192.168.0.1';
```

```
  itm := TMSFMXTableView1.Items.Add;
  itm.CategoryID := 1;
  itm.Caption := 'POP server';
  itm.Description := 'pop.myserver.com';

  itm := TMSFMXTableView1.Items.Add;
  itm.CategoryID := 1;
  itm.Caption := 'SMTP server';
  itm.Description := 'smtp.myserver.com';


  TMSFMXTableView1.LayoutMode := lmGroup;
  TMSFMXTableView1.EndUpdate;

end;
```



As you can see with in the above result, the items from the first category are added separately. This can be controlled with the GroupIndex property. Items can be added to different categories with the CategoryID and within the category, multiple groups can be created with the GroupIndex property.

User interface interaction with the TableView

The list supports scrolling, navigation and selection. Interaction with the list can be done with the keyboard and the mouse.

With the keyboard you can navigate through the items, each time you press the up, down, left or right keys the list selects the previous or next item. The home key selects the first item while the end key selects the last item. When pressing the pagedown or pageup key the list jumps in steps of the BufferSize, if the BufferSize = 0 the step is set to 10.

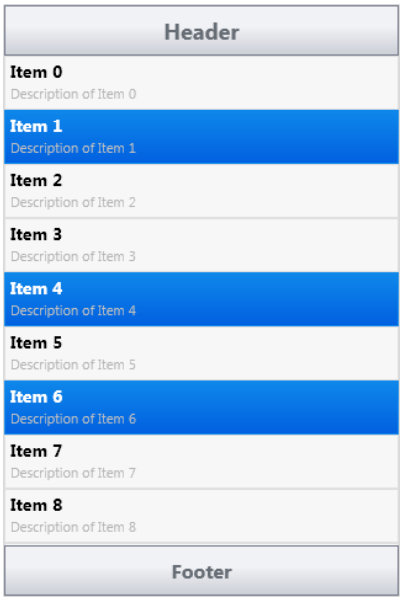
When navigating through the list with the keyboard, the list automatically loads the next buffer.

To navigate in the list with the mouse, you can click and hold your left mouse button on the list then drag up or down depending in the direction you want the list to scroll. When making a flick gesture the list will scroll a certain amount of items with an inertia animation. The faster you flick with the mouse the faster and the further the list will scroll.

MultiSelect

The TableView supports multiple item selection. Multiple item selection can be enabled by setting the MultiSelect property to true. Clicking on an item selects that item and when you click on another item, the previous item is deselected. To select multiple items, you need to hold the CTRL or SHIFT key on the keyboard. With the CTRL key you are able to select items of choice and with the SHIFT key the list selects all items between the first and the last item you have clicked.

MultiSelect can be done by keyboard alone, pressing the direction keys and holding the CTRL or SHIFT key at the same time. Below is a sample of multiple items selected in the list by holding the CTRL key and clicking several items.



Programmatically, in multiselect mode, the selected of an item can be get or set with the item's Selected: Boolean property. In addition, two helper methods exist: SelectAllItems, DeSelectAllItems to programmatically select or unselect all items at once.

This sample code snippet programmatically selects items 1,3,5:

```
TMSFMXTableView1.MultiSelect := true;
```

```
TMSFMXTableView1.DeSelectAllItems;
TMSFMXTableView1.Items[1].Selected := true;
TMSFMXTableView1.Items[3].Selected := true;
TMSFMXTableView1.Items[5].Selected := true;
```

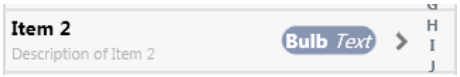
Additional Item Elements

Other than the delete button, the detail arrow image and the checked / unchecked image, the item has three additional elements that are optionally (See notes in Performance chapter) available:

- LeftRectangle
- RightRectangle
- BulbRectangle

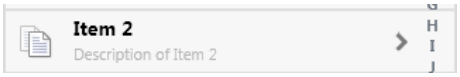
The BulbRectangle contains a TTMSFMXHTMLText shape that contains optional HTML formatted Text and can be set with the BulbText property.

```
TMSFMXTableView1.Items[0].BulbText :='<b>Bulb</b> <i>Text</i>';
```



The Left Rectangle contains a TTMSFMXBitmap component that can be linked to a BitmapContainer. By default the Left Rectangle is disabled. You can enable it by checking the correct option in the ItemOptions property.

Note that an extra area is offered where a bitmap can be shown. You can either show a BitmapContainer item and link it via the BitmapName, or directly load the bitmap in the item.




```
var
  bmp: TTMSFMXBitmapItem;
begin
  bmp := TMSFMXTableView1.Items.Add;
  bmp.Name := 'item_bitmap';
  bmp.Bitmap.LoadFromFile('mybitmap.png');

  TMSFMXTableView1.BitmapContainer := TMSFMXBitmapContainer1;
  TMSFMXTableView1.Items[0].BitmapName := 'item_bitmap';
  //or
  TMSFMXTableView1.Items[0].Bitmap.LoadFromFile('mybitmap.png');
end;
```

The right rectangle is an empty rectangle by default and can contain any visual element that is available in FireMonkey. The TableView supports a small set of controls, that are ready to use and do not need further binding with properties and events. This is explained in the next chapter.

Performance

A default TableView has some performance related options already preset to optimal values for the most typical use of the TableView. The BufferSize is set to 50, which will only load and display the first 50 items in the UI. Second performance related setting is the ItemOptions property. This contains a set of options that can be used to decrease the time the TableView needs to build up the display items list.

As explained in the Design chapter, the default item contains various elements that are cloned when adding items. These setting under ItemOptions affect what parts of an item are cloned or not. When an element is not checked in the ItemOptions, the element is not cloned when the display item list is built and that increases the performance.

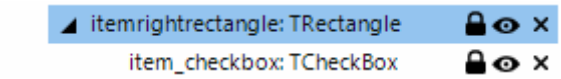
By default the ItemOptions have unchecked LeftRectangle, RightRectangle and BulbRectangle. This means that these elements will not be cloned and will not be available in the item. You can further improve performance by unchecking elements that are not necessary for your application.

Binding Controls

When the right rectangle is activated in the ItemOptions, an extra space will be available that can be filled with the control added in the StyleBook. You can add any type of control, but the list already implements a small set that does not need extra binding.

- CheckBox
- RadioButton
- TrackBar
- ArcDial

Open the StyleBook editor by clicking on the Edit Custom Style option in the popupmenu after right-clicking on the component. Select the default item -> right rectangle element and place a new TCheckBox component inside the rectangle.



Select the TCheckBox component and set the StyleName to 'item_checkbox'. When adding a RadioButton, TrackBar or ArcDial, the component base classname – 'T' must be prepended with 'item_'. The StyleName must be lowercase.

- CheckBox: 'item_checkbox'
- RadioButton: 'item_radiobutton'
- TrackBar: 'item_trackbar'
- ArcDial: 'item_arcdial'

Applying the style in the IDE style editor results in the Checkbox being visible at designtime in each item in the list. In the TableView, the CheckBox OnChange event has been assigned and the IsChecked property is linked to the Item from the Items collection.

To persist the value of the checkbox, the Item has a few Data* properties to store the value:

- DataBoolean
- DataGroupIndex
- DataString
- DataValue
- DataObject

For the CheckBox and RadioButton the DataBoolean property is used, for the TrackBar and ArcDial the DataValue property is used. When changing the DataBoolean property to true, the checkbox will be checked. When checking the checkbox, the DataBoolean property will be set.

This binding happens in the same way for the other controls. For the RadioButton the Item has an extra DataGroupIndex property that can be used to create a RadioGroup, i.e. all items with a RadioButton with the same DataGroupIndex value form a group.

Manual binding is also available through events for non-supported controls. Below is a sample binding a TEdit to the DataString property of the item.

First, add a TEdit control to the right rectangle element in the StyleBook. Set the TEdit control stylename to 'item_edit'. This will be important to access the edit with the correct name. Second, there are 3 important events that are needed for binding with TEdit.

- OnItemCustomize
- OnItemData
- OnChange

The OnItemCustomize event is used to connect the edit with the DataString property. The OnItemData is called when the Data properties are modified. The OnChange event is implemented and sets the DataString property with the value of the edit. The combination results in the code below:

```
TForm1.ItemEditChanged(Sender: TObject);
var
  i: Integer;
  it: TTMSFMXTableViewItem;
begin
  i := (Sender as TEdit).Tag;
  if (i >= 0) and (i <= TMSFMXTableView1.Items.Count - 1) then
  begin
    it := TMSFMXTableView1.Items[i];
    it.DataString := (Sender as TEdit).Text;
  end;
end;

TForm1.TMSFMXTableView1ItemCustomize(Sender: TObject;
  AItem: TTMSFMXTableViewItem; AItemShape: TTMSFMXTableViewItemShape;
  AItemControlShape: TControl);
var
  shpr: TRectangle;
  edt: TEdit;
begin
  shpr := AItem.ShapeRightRectangle;
  if Assigned(shpr) then
  begin
    edt := (shpr.FindStyleResource('item_edit') as TEdit);
    if Assigned(edt) then
    begin
      edt.OnChangeTracking := ItemEditChanged;
      edt.Tag := AItem.Index;
    end;
  end;
end;

TForm1.TMSFMXTableView1ItemData(Sender: TObject;
  AItem: TTMSFMXTableViewItem; AItemShape: TTMSFMXTableViewItemShape;
  AItemControlShape: TControl);
var
  shpr: TRectangle;
  edt: TEdit;
begin
  shpr := AItem.ShapeRightRectangle;
  if Assigned(shpr) then
  begin
    edt := (shpr.FindStyleResource('item_edit') as TEdit);
    if Assigned(edt) then
```

```
      edt.Text := AItem.DataString;
  end;
end;
```

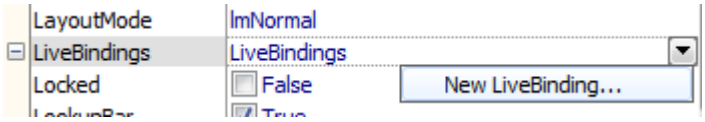
LiveBindings

LiveBindings is new way of data-binding controls, for more information and samples on LiveBindings, please read the documentation on http://docwiki.embarcadero.com/RADStudio/en/LiveBindings_in_RAD_Studio first.

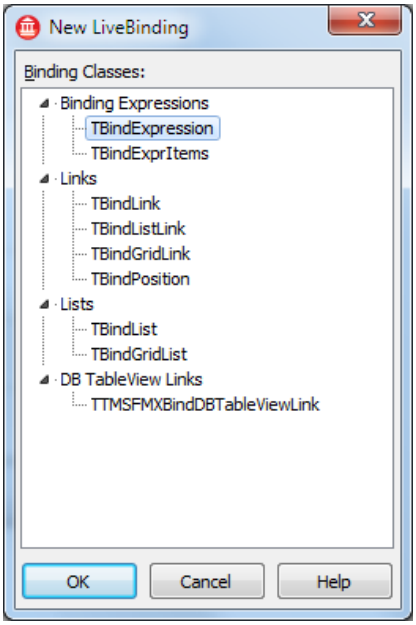
The TableView has a basic implementation that populates the list and takes care of binding the SelectedItemIndex. When dropping a TableView on the form you will notice a LiveBindings property.



To create a new LiveBinding, you can either click on the arrow and select “New LiveBinding…” or click directly on “New LiveBinding…” at the bottom of the object inspector.

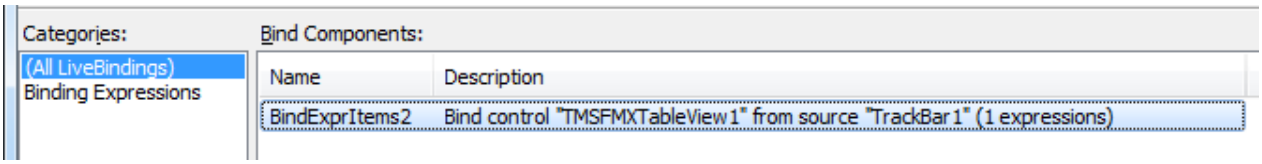


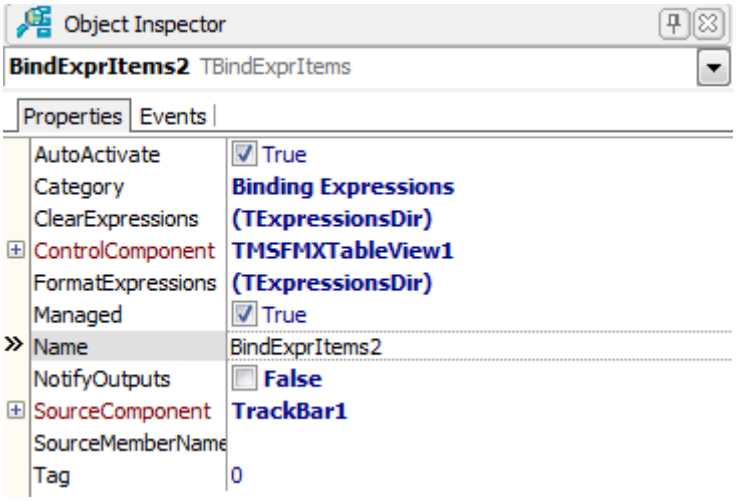
This action will automatically drop a BindingsList component on the form and will show the BindingsList editor window.



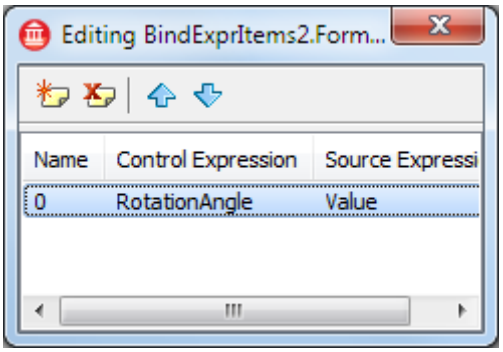
When reading the documentation in the “LiveBindings in RAD Studio” you will notice that LiveBindings is not limited to DataBase support. There is also support for binding various properties of the TableView to other controls. Below is a sample that binds the TrackBar position to the TableView rotationAngle.

Drop a new TableView and a TrackBar component on the form. Select the TrackBar component and add a new TBindExprItems expression. In the BindingsList component you see the TBindExprItems component listed.





Point the SourceComponent to the TrackBar and the ControlComponent to the TMSFMXTableView1. Start the FormatExpressions editor by double-clicking on the "(TExpressionsDir)" label. Click on the add button to add a new format expression and fill in the Binding properties. The ControlComponent is set to TMSFMXTableView1 and we want the RotationAngle to be modified. Fill in RotationAngle in the Control Expression field. For the Source Expression which is linked with the SourceComponent we fill in "Value".



Now there is one step left to implement before the application is ready. When dragging the slider of the TrackBar, the RotationAngle of the TMSFMXTableView component will be unaffected. This is because the TMSFMXTableView did not receive a notice from the BindingsList component, therefore we need to notify the bind component that the value of the TrackBar has changed.

This is done by implementing the OnChange event and notifying the BindingsList component:

```
var
  FNotifying: Integer;

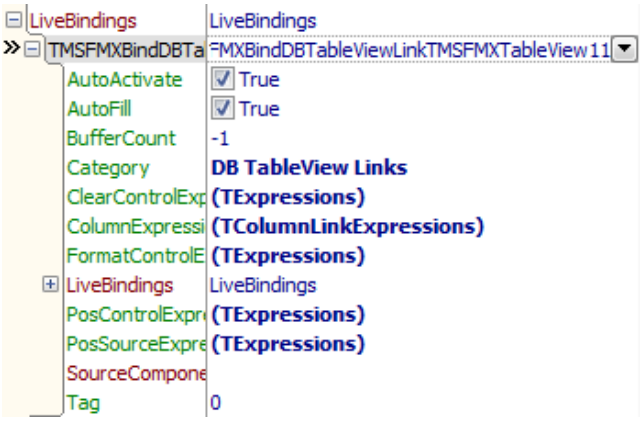
procedure TForm557.TrackBar1Change(Sender: TObject);
begin
  // Some controls send notifications when setting properties,
```

```
  // like TTrackBar
  if FNotifying = 0 then
  begin
    Inc(FNotifying);
    // Send notification to cause expression re-evaluation of dependent
expressions
    try
      BindingsList1.Notify(Sender, '');
    finally
      Dec(FNotifying);
    end;
  end;
end;
```

Now when dragging the slider of the TrackBar, the BindingsList is notified and the TMSFMXTableView rotates according to the Value of the TrackBar. Multiple bindings can be made and are triggered simultaneously due to the Notify procedure of the BindingsList.

In the BindingsList editor window you will notice that a new category is added for the TableView: “DB TableView Links”. The link “TTMSFMXBindDBTableViewLink” can be used to connect to a DataSource through a BindScope. This link is designed to work specifically with DataBase connections.

The TableView DB link automatically generates Expressions to link the SelectedItemIndex to the correct record in the Database. Internally the DataSet is automatically loaded and for each record in the DataSet an item is added.



In the object inspector, after expanding the LiveBindings property, you will notice a ColumnExpressions property. This is because the implementation inherits from the standard TBindGridLink and is necessary to link Database fields to multiple elements in the TableView.

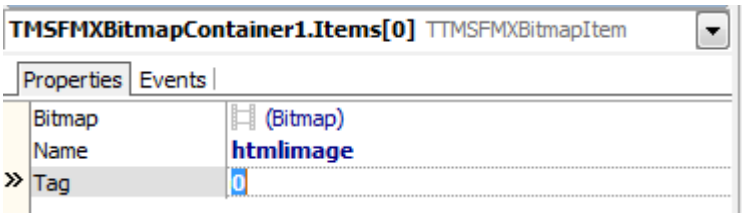
More on the TTMSFMXBindDBTableViewLink is explained in the Samples chapter at the TMS TableView LiveBindings Demo.

HTML support

The TableView supports HTML in various elements such as the headertext, footertext, bulbtext, the caption and the description. The HTML engine in FireMonkey is a port of the VCL HTML engine, thus supporting most functionality that was available in the VCL HTML Engine.

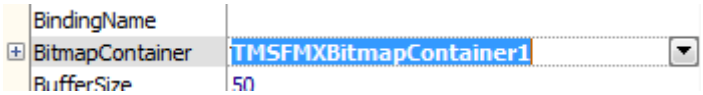
As ImageLists do no exist in the FireMonkey framework, the HTML engine supports displaying images from a BitmapContainer and this is demonstrated in the sample below.

Drop a TableView and a BitmapContainer on the form. In the BitmapContainer we have loaded an image and named it htmlimage:



The name is necessary because the HTML Engine will search for an image with a name that is set in the <img> tag.

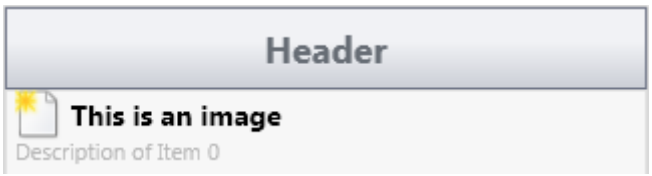
Now assign the BitmapContainer to the TableView component:



In this sample we will display an image in HTML in the caption of an item.
This can be done by using the <img> tag.

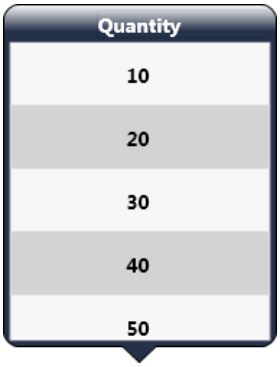
```
  TMSFMXTableView1.Items[0].Caption := '<img src="htmlimage"> This is an
image';
```

The item will display the caption with an image that comes from the BitmapContainer. In this way, one item can be used multiple times.



For more information about the HTML engine and supported tags please read the chapter HTML rendering engine.

TMS Popup control



The TTMSFMXPopup is a component that has the capability to display a FireMonkey control inside a fully customizable transparent popup window. It is specifically designed match the layout of a native iOS popup window. This component can be easily configured to display itself positioned at a specific control on the form or a given absolute position. The popup control can have optionally a footer and a header and in the footer and header buttons can be added as well as HTML formatted text.

Header and footer are configurable via following properties:

- **FooterButtons**: collection of TTMSFMXPopupButton instances displayed from left in the footer
- **FooterHeight**: sets the height in pixels of the footer
- **FooterText**: string holding HTML formatted text for the footer
- **HeaderButtons**: collection of TTMSFMXPopupButton instances displayed from left in the header
- **HeaderHeight**: sets the height in pixels of the header
- **HeaderText**: string holding HTML formatted text for the header

Following properties exist for the TTMSFMXPopupButton:

- **BitmapName**: name of image in the connected BitmapContainer to use for the button
- **Kind**: sets the type of the button, following button types are predefined: bkArchive, bkBack, bkCancel, bkDelete, bkDone, bkEdit, bkMark, bkMove, bkNormal
- **Layout**: property to set layout of the button to normal or pointer
- **ShowText**: when true, text is shown next to the image in the button
- **Text**: button caption text
- **Width**: width of the button

Properties on TTMSFMXPopup that control appearance and position of the popup:

- **ArrowHeight**: The height of the arrow.
- **ArrowPosition**: The position of the arrow.
- **ArrowWidth**: The width of the arrow.
- **DetailControl**: The control that will be displayed and automatically positioned inside the popup.
- **HeaderText**: The text of the header that supports HTML.
- **Placement**: Property to determine the position of the popup relative or absolute to a given placement control.

- **PlacementTarget**: The control used as a reference to position the popup after it is displayed.
- **PlacementRectangle**: A custom rectangle used to display the popup, in combination with the PlacementTarget and Placement properties.

After setting properties where the popup must be shown you can use the following methods to popup or close the dialog:

```
TMSFMXPopup1.Popup;
TMSFMXPopup1.Close;
```

Example:

This code snippet assigns a TTMSFMXTableView control to be displayed on the popup and configures the TTMSFMXPopup to open at the bottom of a button on the form with a button in the header to close the popup again:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // hide the popup footer
  TMSFMXPopup1.ShowFooter := false;
  // tableview should be displayed in the popup only
  TMSFMXTableView1.Visible := false;
  // assign the tableview as detail control for the popup
  TMSFMXPopup1.DetailControl := TMSFMXTableView1;
  // set the control as reference for position of the popup
  TMSFMXPopup1.PlacementTarget := Button1;
  // show the popup at the bottom of the button centered
  TMSFMXPopup1.Placement := TPlacement.plBottomCenter;
  // add a header button that closes the popup
  TMSFMXPopup1.HeaderButtons.Clear;
  with TMSFMXPopup1.HeaderButtons.Add do
  begin
    Kind := TTMSFMXBarButtonKind.bkBack;
    Width := 50;
  end;
end;

procedure TForm1.TMSFMXPopup1HeaderButtonClick(Sender: TObject;
  AButton: TTMSFMXPopupButton; AButtonShape: TTMSFMXBarButton);
begin
  if AButton.Index = 0 then
    TMSFMXPopup1.Close;
end;
```

TMS Mini HTML rendering engine

Another core technology used among many components is a small fast & lightweight HTML rendering engine. This engine implements a subset of the HTML standard to display formatted text. It supports following tags :

B : Bold tag
<B> : start bold text
</B> : end bold text

Example : This is a <B>test</B>

U : Underline tag
<U> : start underlined text
</U> : end underlined text

Example : This is a <U>test</U>

I : Italic tag
<I> : start italic text
</I> : end italic text

Example : This is a <I>test</I>

S : Strikeout tag
<S> : start strike-through text
</S> : end strike-through text

Example : This is a <S>test</S>

A : anchor tag
<A href="value"> : text after tag is an anchor. The 'value' after the href identifier is the anchor. This can be an URL (with ftp,http,mailto,file identifier) or any text.
If the value is an URL, the shellexecute function is called, otherwise, the anchor value can be found in the OnAnchorClick event </A> : end of anchor

Examples : This is a <A href= "mailto:myemail@mail.com ">test</A>
This is a <A href="http://www.tmssoftware.com">test</A>
This is a <A href="somevalue">test</A>

FONT : font specifier tag
<FONT face='facevalue' size='sizevalue' color='colorvalue' bgcolor='colorvalue'> : specifies font of text after tag.
with

- face : name of the font
- size : HTML style size if smaller than 5, otherwise pointsize of the font
- color : font color with either hexidecimal color specification or Borland style color name, ie clRed,clYellow,clWhite ... etc
- bgcolor : background color with either hexidecimal color specification or Borland style color name </FONT> : ends font setting

Examples : This is a <FONT face="Arial" size="12" color="clred">test</FONT>
This is a <FONT face="Arial" size="12" color="#FF0000">test</FONT>

P : paragraph
<P align="alignvalue" [bgcolor="colorvalue"] [bgcolorto="colorvalue"]> : starts a new paragraph, with left, right or center alignment. The paragraph background color is set by the optional bgcolor parameter. If bgcolor and bgcolorto are specified,
a gradient is displayed ranging from begin to end color.
</P> : end of paragraph

Example : <P align="right">This is a test</P>
Example : <P align="center">This is a test</P>
Example : <P align="left" bgcolor="#ff0000">This has a red background</P>
Example : <P align="right" bgcolor="clYellow">This has a yellow background</P>
Example : <P align="right" bgcolor="clYellow" bgcolorto="clred">This has a gradient background</P>*

HR : horizontal line
<HR> : inserts linebreak with horizontal line

BR : linebreak

 : inserts a linebreak

BODY : body color / background specifier
<BODY bgcolor="colorvalue" [bgcolorto="colorvalue"] [dir="v|h"] background="imagefile specifier"> : sets the background color of the HTML text or the background bitmap file

Example : <BODY bgcolor="clYellow"> : sets background color to yellow
<BODY background="file://c:\test.bmp"> : sets tiled background to file test.bmp
<BODY bgcolor="clYellow" bgcolorto="clWhite" dir="v"> : sets a vertical gradient from yellow to white

IND : indent tag
This is not part of the standard HTML tags but can be used to easily create multicolumn text
<IND x="indent"> : indents with "indent" pixels

Example :
This will be <IND x="75">indented 75 pixels.

IMG : image tag
<IMG src="specifier:name" [align="specifier"] [width="width"] [height="height"] [alt="specifier:name"] > : inserts an image at the location

specifier can be:
file : name is the full filename specifier
no specifier : name of image in a BitmapContainer

Optionally, an alignment tag can be included. If no alignment is included, the text alignment with respect to the image is bottom. Other possibilities are: align="top" and align="middle"

The width & height to render the image can be specified as well. If the image is embedded in anchor tags, a different image can be displayed when the mouse is in the image area through the Alt attribute.

Examples :
This is an image <IMG src="file://c:\my documents\test.bmp">
This is an image <IMG src="name">

SUB : subscript tag
<SUB> : start subscript text
</SUB> : end subscript text

Example : This is ⁹/₁₆ looks like 9/16

SUP : superscript tag
<SUP> : start superscript text
</SUP> : end superscript text

UL : list tag
<UL> : start unordered list tag
</UL> : end unordered list

Example : <UL>
<LI>List item 1
<LI>List item 2
<UL>
<LI> Sub list item A
<LI> Sub list item B
</UL>
<LI>List item 3
</UL>

LI : list item
<LI [type="specifier"] [color="color"] [name="imagename"]>: new list item specifier can be "square", "circle" or "image" bullet. Color sets the color of the square or circle bullet. Imagename sets the PictureContainer image name for image to use as bullet

SHAD : text with shadow
<SHAD> : start text with shadow
</SHAD> : end text with shadow

Z : hidden text
<Z> : start hidden text
</Z> : end hidden text

Special characters
Following standard HTML special characters are supported :
< : less than : <
> : greater than : >
& : &
" : "
  : non breaking space
™ : trademark symbol
€ : euro symbol
§ : section symbol
© : copyright symbol
¶ : paragraph symbol

General FireMonkey component usage guidlines

With the new FireMonkey framework, the methodology to create and use components has dramatically changed. A component now exists of 2 parts.

Visual part

The visual part is stored in a .style file, which is compiled to a .res file through an .rc file. The .rc file is included in the package and must be recompiled whenever a change is made to the .style file. For each component in this set you will find a .style file. In this file, the default layout of the component is stored.

You will notice different elements, basic elements such as an arc, ellipse, rectangle …
The elements combine and define the layout of a control. The basic elements are called shapes, and are already available by default. The TMS TableView for FireMonkey takes this new way of styling one step further: custom shapes. In several components you will find custom shapes registered and useable in a new application, and used in the component by default.

Each shape or element can have a StyleName, which is used in the non-visual part of the control for interaction. This name is key in the relationship or “style-contract” between style resource and component code.

Non-visual part

The non-visual part of the component interacts with the shapes defined in the .style file. This is a normal .pas unit file as was used for VCL component, yet little to no painting is done in code. As explained above, the visual part is already defined by the style.

The component defined in this unit needs to inherit from the TStyledControl class, which can be styled at designtime. This is the base class for all styleable controls, just like the TCustomControl class was the base class for most controls in the VCL framework.

Naming convention

It is always good practice to handle a consistent naming convention, therefore all .rc, .pas files and .style files should start with the FireMonkey unit scope name “FMX.”, such as the units: FMX.Types, FMX.Dialogs, FMX.Objects …

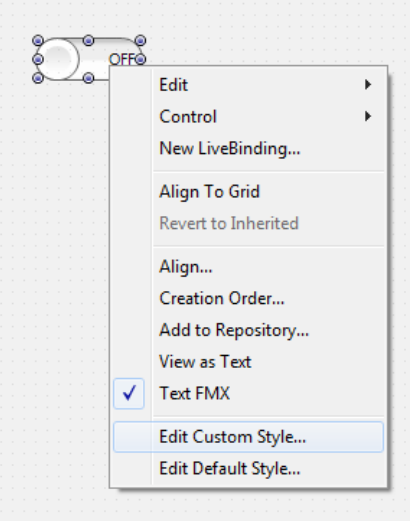
Inside the style file each element can have a StyleName, which can be used in the non-visual part to address the resource. Make sure each element has a unique StyleName to avoid mistakes when interacting with the component. All combinations of elements must be encapsulated within a rectangle element that is invisible by default (through the Fill.Kind and Stroke.Kind = bkNone), and has the StyleName of the component.

If you have a component named TFMXMyFirstControl, the the StyleName of the rectangle encapsulating all other elements must be set to FMXMyFirstControlStyle. The “T” is removed and “Style” is added.

Styling

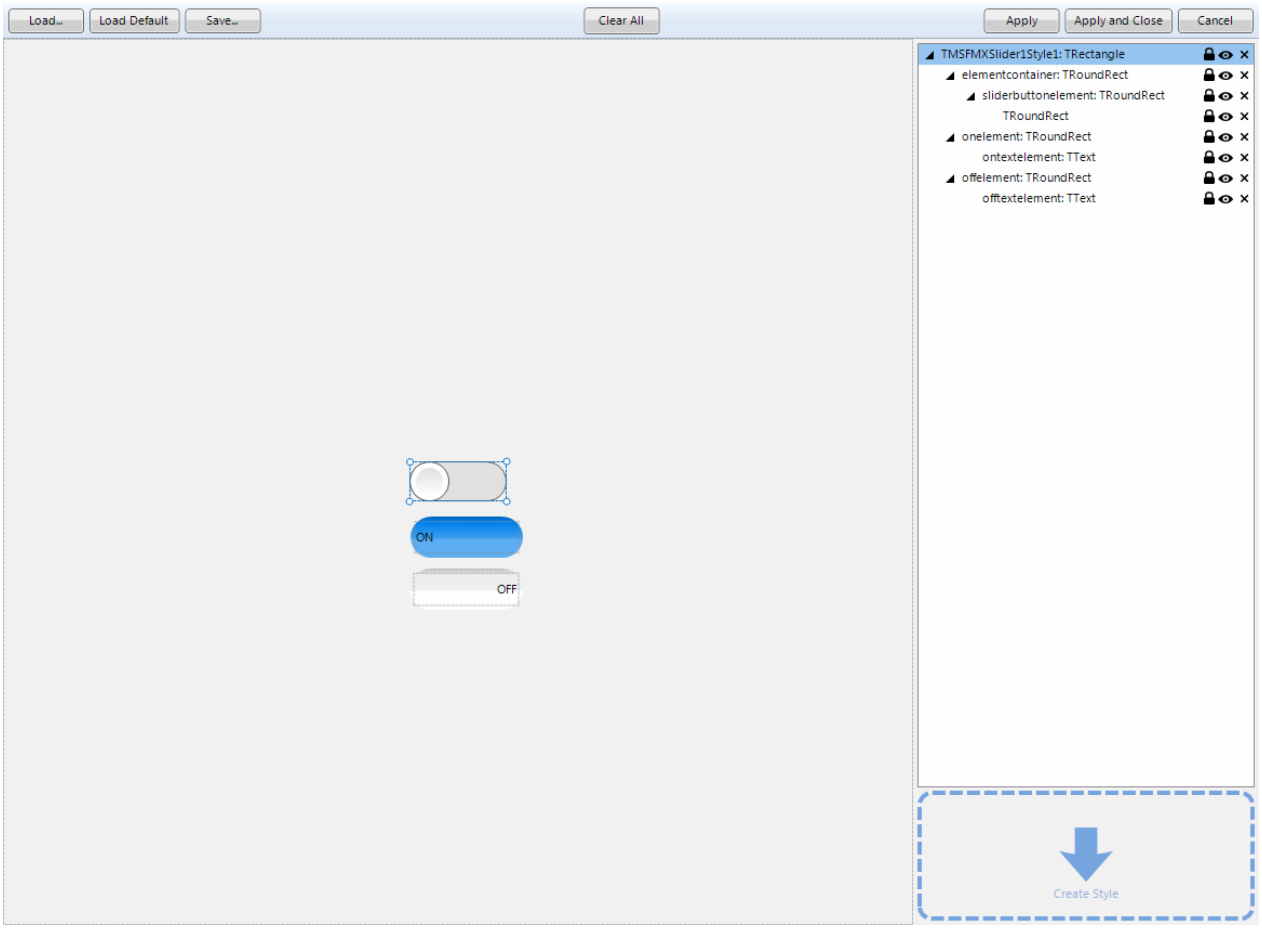
Each component inherits from TTMSFMXBaseControl which implements a basic Fill and Stroke, and handles the style resource files that define the default layout of the component. To change the visuals of the component you no longer have corresponding properties in the object inspector. Right-clicking on the component provides two extra menu items that can be used to edit the style of the component.

Clicking either of these items will automatically drop a StyleBook component on the form when there is not yet one available. A StyleBook holds custom and default styles. When the default style is changed, dropping a new component of the same class will automatically get this changed style as defined in the default style.

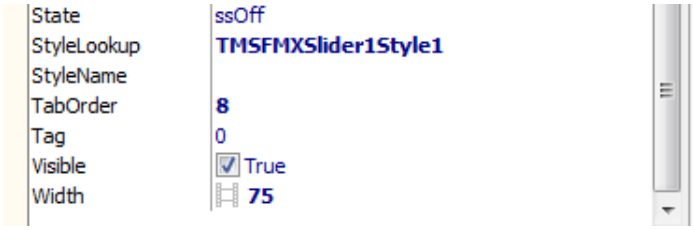


- Edit Custom Style: Clicking on this item starts the IDE style editor and copies the default style of the component. The name of the style is set to the component name on the form and appended with 'Style1'. After changing properties through the editor, the style is then applied to the component. You will notice that the StyleLookUp property is set to the name of the custom style in the stylebook.

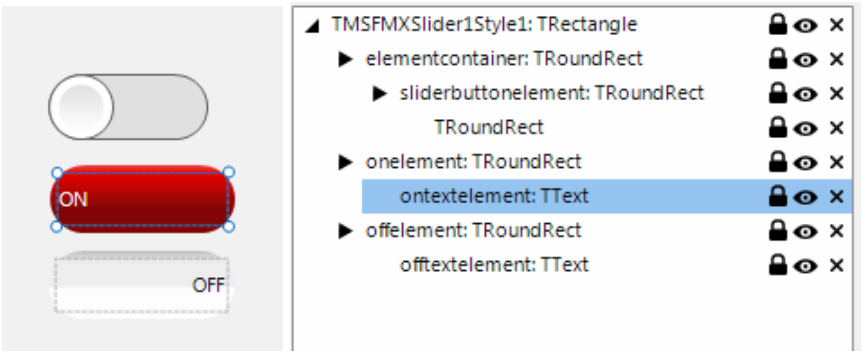
- Edit Default Style: Clicking on this item starts the IDE style editor and uses the default style of the component. As with the Edit Custom Style option, the name of that style is set. The difference between these 2 options is that the default style has a generic name and is applied to all new instances of the component that are dropped on the form. The StyleLoopup property is not set.



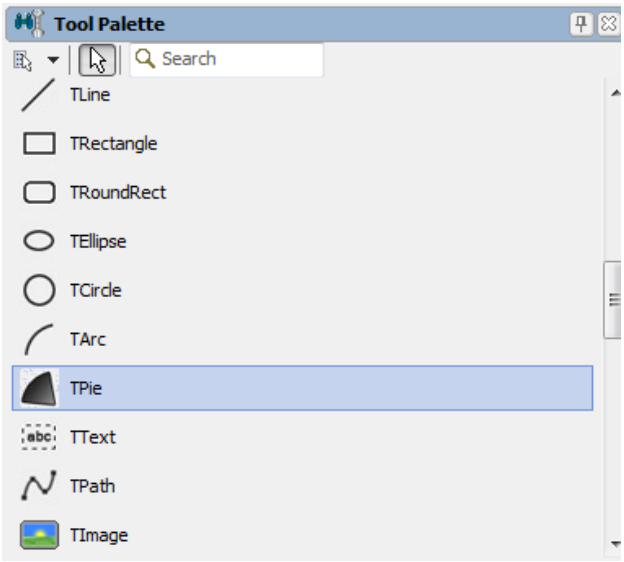
The IDE style editor can be started with these 2 options, or by double-clicking on the StyleBook editor icon on the form. In this example we have a TTMSFMXSlider component that will be altered with a custom style. Notice the TMSFMXSlider1Style1 name that is used for this style. When applying this style, you will also notice the StyleLookup property is set to TMSFMXSlider1Style1.



Each component exists of different styleable elements. Simple click on an element in the editor to change the appearance.



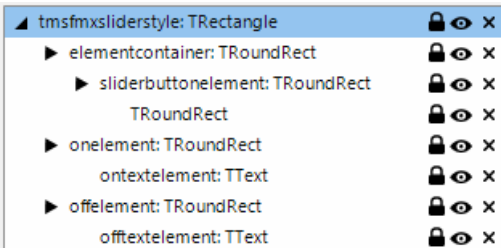
You can also add new elements from the Tool palette.



After applying the Style, the component will have the new custom style.



Dropping a new TTMSFMXSlider component on the form will not adopt this custom style and will have the default style applied. Editing the default style is done in the same way, yet the name of the style differs and each new instance of the TTMSFMXSlider adopts the edited default style.



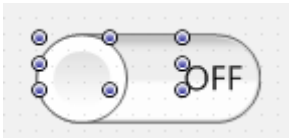
General component properties that do not directly define a visual appearance of the component are still displayed in the Object Inspector. Note though that some properties will affect what is available in the style editor! For example, if a component provides a collection of visible items displayed in the control and it is desirable that the visual appearance of each item can be customized, style elements (shapes) will be dynamically added or removed and be available in the IDE style editor.
In other cases, it is desirable that the appearance for a given type of items in a control is identical. This can be represented as a single style element in the style editor. The component will then internally copy the settings of the style element and apply it to each item displayed in the control.

Components

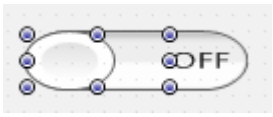
Most of the components in the FireMonkey framework can be scaled and rotated without loss of functionality and quality. As our base control implementation inherits from a base class which supports these features, all of the controls inside the TMS TableView set support scaling and rotation.

Scaling: With the Scale property you can specify how large the component must be. The default value of the X and Y property of the Scale is 1. This means that the default component layout is set at one, if you have a component which has 100 pixels width and height dimensions, setting the scale X and Y properties to 1.5 will automatically increase the width and height to 150 pixels. Below are some examples at designtime, which shows the capability of this property.

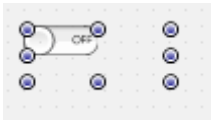
Scale 1.5



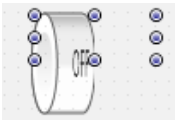
Scale X 1.5 Y 1



Scale 0.5

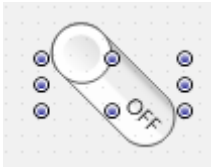


Scale X 0.5 Y 2



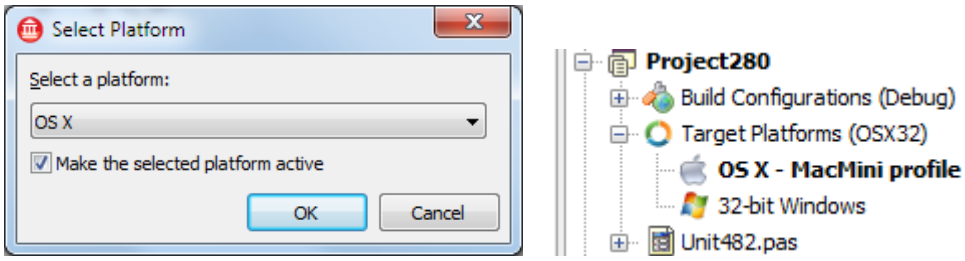
Rotation: The rotation property rotates the component around the center by default, which can be changed with the rotationcenter property. Rotating the component does not limit interaction capabilities and functionality.

45°



Cross-platform deployment of applications with TMS TableView for FireMonkey components

The FireMonkey framework supports deployment to Windows, Mac OS-X and iOS. Likewise, the TMS TableView for FireMonkey components can be used in applications targeted for these 3 platforms. If you have created a FireMonkey HD project, you can specify in the Project Manager under Target platforms Win32, Win64 or Mac OS-X platforms. Depending on the chosen platform, the compiler will generate a binary for debug or deployment for Windows 32bit, Windows 64bit or Mac OS-X. The app will be automatically compiled with the installed TMS Instrumentation for FireMonkey components.



For iOS, the FireMonkey HD project cannot be used. For this platform, create a new FireMonkey HD iOS project. The TMS TableView for FireMonkey components supports both FireMonkey HD iOS and FireMonkey HD projects and installs the components on the tool palette for both targets. As such, when choosing a FireMonkey iOS project, the TMS TableView for FireMonkey components will also appear in the IDE component palette. Technically, this is done by two different packages that are installed in the IDE: TMSFMXTableViewPkgDXE2 (for FireMonkey HD) and TMSFMITableViewPkgDXE2 (for iOS)

To actually run the FireMonkey iOS project in the iPhone or iPad simulator or real device, the project must be compiled for iOS which is done from the xCode environment from Mac OS-X. Note that it is xCode with the FPC compiler that will compile an iOS project. Such project must be created first from the FireMonkey iOS project created in the IDE. This is done with the tool dpr2xcode. You can find this tool in the RAD Studio bin folder after installation (Normally this is under C:\Program Files (x86)\Embarcadero\RAD Studio\9.0\bin\). Running dpr2xcode for your FireMonkey HD iOS app will generate an xCode folder with an xCode project and all files belonging to your project ready for compilation with xCode/FPC on the Mac OS-X. Note that this doesn't include the TMS TableView for FireMoney component sources yet that will be needed when you have used these components in your project.

Before opening this project file in xCode for compilation, you need to copy the correct source files from the TMS TableView for FireMonkey installation folder to this xCode folder of your project. Locate this folder and simply copy following files:

- FMX_*.pas files
- FMX_*.res files
- tmsdefsios.inc file

Note that for this compilation via xCode/FPC, full source code for the TMS TableView for FireMonkey components is needed. This also means that it is a limitation of the trial versions of our FireMonkey components (that are only supplied in binary form), that these cannot be used to actually deploy to iPhone or iPad. For this, the registered version of the components is needed.

The last step to take is to rename the tmsdefsios.inc file to tmsdefs.inc file in the xCode project folder in order to compile correctly with xCode/FPC. After opening the project in xCode, click the run button, which will automatically compile and run the project in the simulator or real device.

More information about deploying on different platforms can be found on:

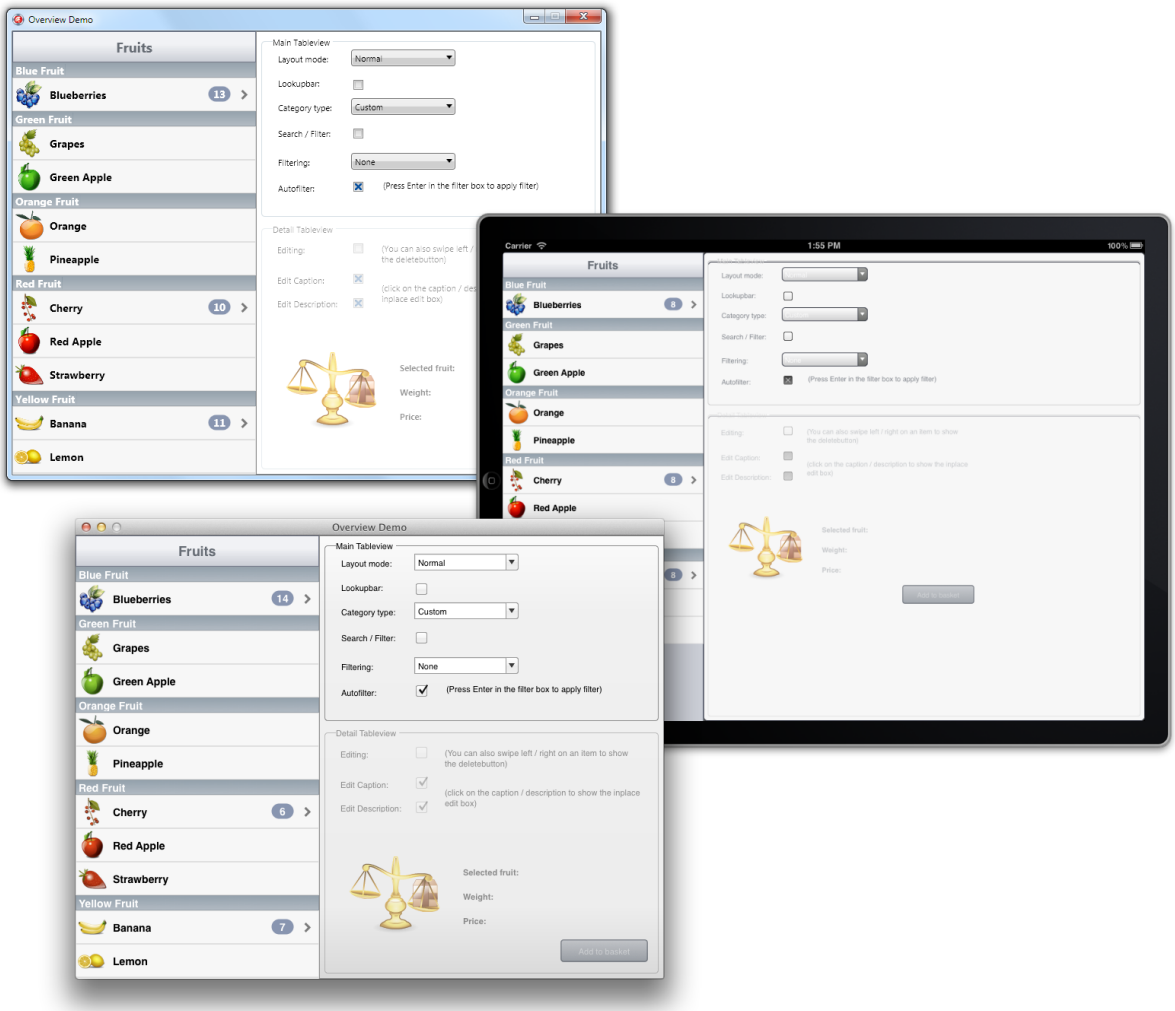
http://docwiki.embarcadero.com/RADStudio/en/Connecting_Your_PC_to_a_Mac

More information about deploying in xCode can be found on:

http://docwiki.embarcadero.com/RADStudio/en/FireMonkey_Development_Setup_for_iOS

Samples

TMS TableView Overview Demo



The demo comes in 2 versions:

- HD version: runs on Win32, Win64 and MAC OS-X.
- iOS version: runs on Win32 and iOS.

If you have followed the steps to distribute on Mac OS-X or iOS, described in the "General FireMonkey component usage guidelines" chapter, correctly, you should now be able to deploy the demo on the various platforms.

Running the HD version is as simple as selecting the correct platform.
To compile and run the iOS version on xcode, there are some important additional steps that need to be taken before the demo will run.

Please note that de demo is distributed in the trail version as well, but you will not be able to run the demo in the iOS simulator due to the lack of source code that is needed to compile the sample.

In the iOS version you will notice an xcode folder, with an xcode project file. Before opening this project file in xcode you need to copy the correct source files from the installation directory to the directory of this demo, where the UDemo.pas file is located:

- All the FMX_*.pas files
- All the FMX_*.res files
- The tmsdefsios.inc file

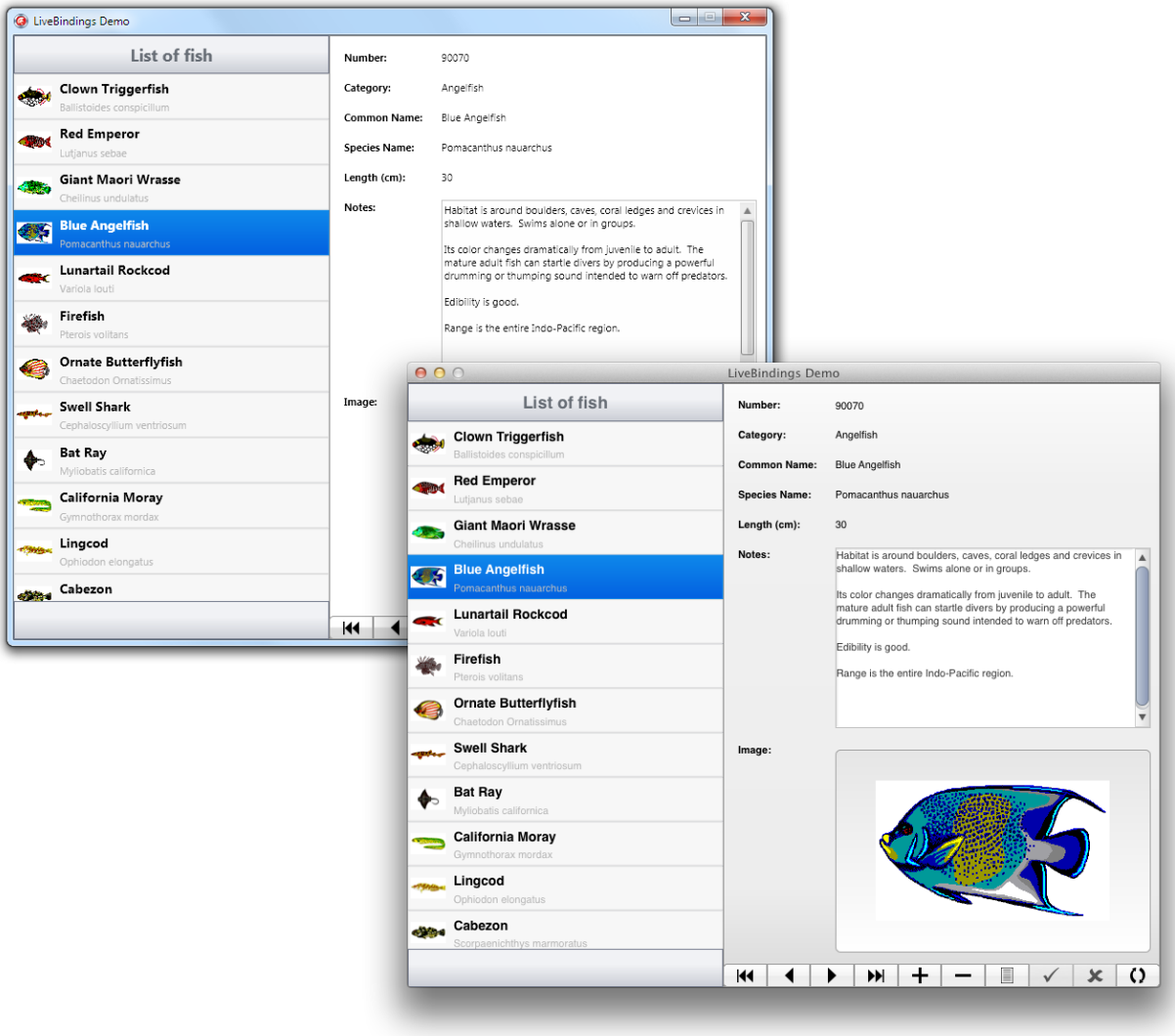
The last step to take is to rename the tmsdefsios.inc file to tmsdefs.inc file in order to compile correctly. After opening the project in xcode, click the run button, which will automatically compile and run the project in the simulator. Of course you can select to run it on a real device as well.

This demo also uses the TTMSFMXPopup when clicking on the Quantity button:

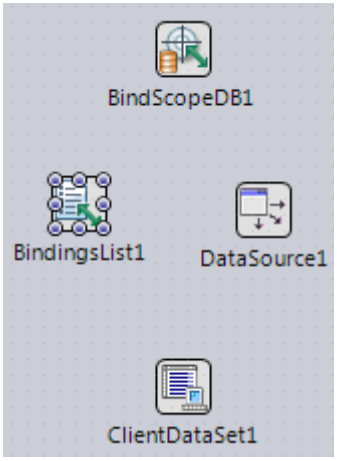


By default, the TTMSFMXPopup has a shadow that is displayed on Windows and Mac, and is disabled on iOS due to performance issues.

TMS TableView LiveBindings Demo 1 & 2



This demo is a LiveBindings sample, specifically designed in combination with the TMSTableView. The demo loads a ClientDataSet with a biolife.xml sample data file. The ClientDataSet is then connected, in combination with a DataSource, to a BindScope, which is needed to bind data to the ListBindings component.



The demo makes use of a TTMSFMXBindDBTableViewLink that is created and registered specifically for the TableView component. When opening the editor of the BindingsList you will see various bindings to different elements that are placed on the right side of the TableView. All these elements have a binding to a specific field and will update when navigating through the list or with the BindNavigator component.

Name	Description
DBLinkImageControl1Graphic1	Link control "ImageControl1" to source "BindScopeDB1, Graphic"
DBLinkLabel2SpeciesNo1	Link control "Label8" to source "BindScopeDB1, Species No"
DBLinkLabel12Category1	Link control "Label12" to source "BindScopeDB1, Category"
DBLinkLabel10Common_Name1	Link control "Label10" to source "BindScopeDB1, Common_Name"
DBLinkLabel6SpeciesName1	Link control "Label6" to source "BindScopeDB1, Species Name"
DBLinkLabel4Lengthcm1	Link control "Label4" to source "BindScopeDB1, Length (cm)"
DBLinkMemo1Notes1	Link control "Memo1" to source "BindScopeDB1, Notes"
TMSFMXBindDBTableViewLink1	Link grid control "TMSFMXTableView1" to source "BindScopeDB1"

Selecting the TMSFMXBindDBTableViewLink1 component will display its properties in the object inspector.

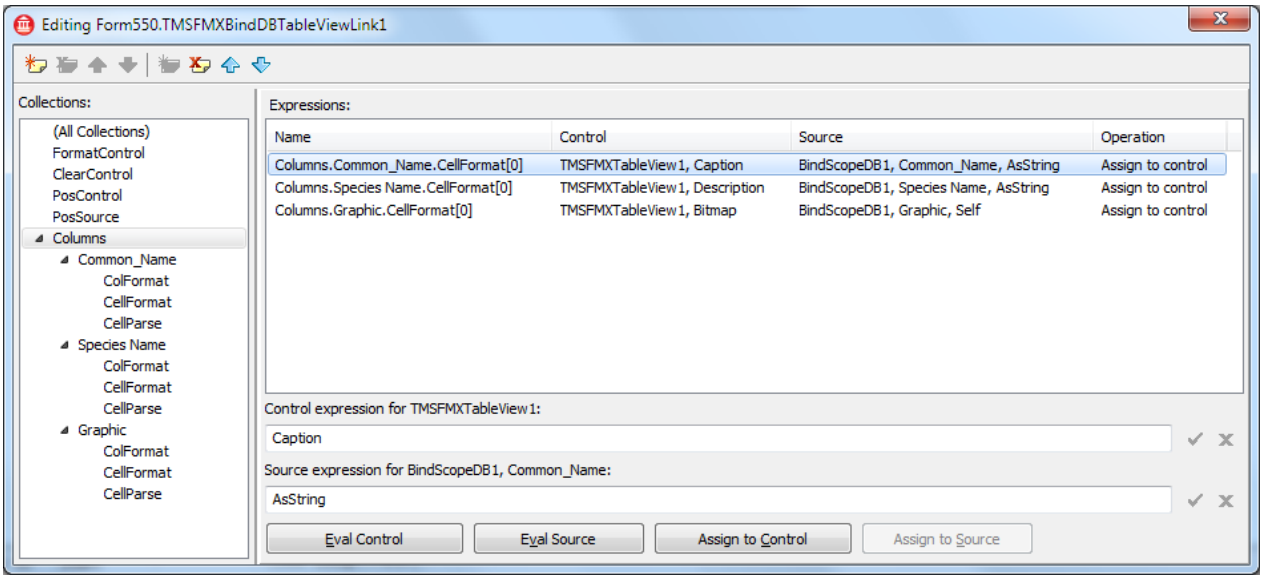
TMSFMXBindDBTableViewLink1 TTMSFMXBindDBTableViewLink

Properties | Events

AutoActivate	True
AutoFill	True
BufferCount	-1
Category	**DB TableView Links**
ClearControlExpress	**(TExpressions)**
ColumnExpressions	**(TColumnLinkExpressions)**
ControlComponent	**TMSFMXTableView1**
FormatControlExpre	**(TExpressions)**
Name	TMSFMXBindDBTableViewLink1
PosControlExpressic	**(TExpressions)**
PosSourceExpressio	**(TExpressions)**
SourceComponent	**BindScopeDB1**
Tag	0

As with the TrackBar sample in the LiveBindings chapter this link has a SourceComponent and a ControlComponent. The difference between this link and the link in the TrackBar sample is, that the TMSFMXBindDBTableViewLink component is able to bind multiple fields to multiple elements in the TableView.

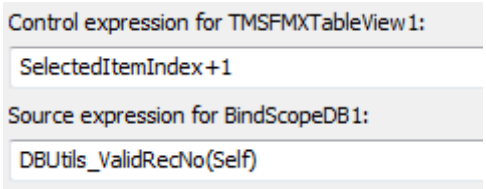
In the sample you will notice that there is binding to the Caption, Description and the Bitmap in the TableView. To bind data to these elements, an expression must be added per element to the ColumnExpressions collection. Double-clicking on the ColumnExpressions property opens the expressions editor.



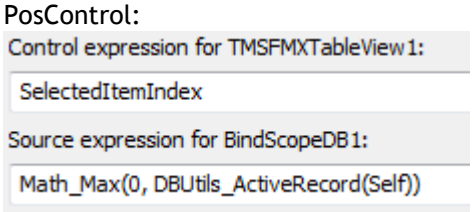
For the caption and the Description, the binding has been added to a string field in the DataBase. To return the value for the current record, the AsString function must be used in the Source expression. To bind the graphic, the source expression is Self. To know exactly which expression you must use, you can click on Eval Source which will tell you the type of the data.

When starting the application, the list will load the data and display the caption, description and bitmap. When clicking on the list, the navigation in the dataset is automatically handled. This is due to the initialization of the TTMSFMXBindDBTableViewLink that has already taken care of this functionality and has stored the correct values in the PosSource and PosControl expressions.

PosSource:



PosControl:



By default the tableview has properties for the Caption, Description and Bitmap. This can be expanded to link to multiple fields in the database. After installing the package you will also notice a TTMSFMXTableViewEx. This implements an extended item class that introduces a new property: Extra.
This property is then linked to an extra element that manually needs to be added through the stylebook editor with the correct name. This is a sample, so this can easily be expanded with multiple properties for multiple elements.

Below is the code of the extended list with the extended item property.

```
unit FMX.TMSTableViewEx;

interface

uses
  Classes, FMX.TMSTableView, FMX.Objects;


type
  TTMSFMXTableViewItemEx = class(TTMSFMXTableViewItem)
  private
    FExtra: string;
    procedure SetExtra(const Value: string);
  protected
    procedure UpdateExtra;
    procedure ApplyEx; override;
  public
    function ShapeExtra: TText;
  published
    property Extra: string read FExtra write SetExtra;
  end;

  TTMSFMXTableViewItemsEx = class(TTMSFMXTableViewItems)
  public
    function CreateItemClass: TCollectionItemClass; override;
  end;

  TTMSFMXTableViewEx = class(TTMSFMXTableView)
  protected
    function GetClassStyleName: String; override;
  public
    function CreateItems: TTMSFMXTableViewItems; override;
  end;


implementation

{ TTMSFMXTableViewItemsEx }

function TTMSFMXTableViewItemsEx.CreateItemClass: TCollectionItemClass;
begin
  Result := TTMSFMXTableViewItemEx;
end;

{ TTMSFMXTableViewEx }

function TTMSFMXTableViewEx.CreateItems: TTMSFMXTableViewItems;
```

```
begin
  Result := TTMSFMXTableViewItemsEx.Create(Self);
end;

function TTMSFMXTableViewEx.GetClassStyleName: String;
begin
  Result := ClassParent.ClassName + 'style';
  Delete(Result, 1, 1);
end;

{ TTMSFMXTableViewItemEx }

procedure TTMSFMXTableViewItemEx.ApplyEx;
begin
  inherited;
  UpdateExtra;
end;

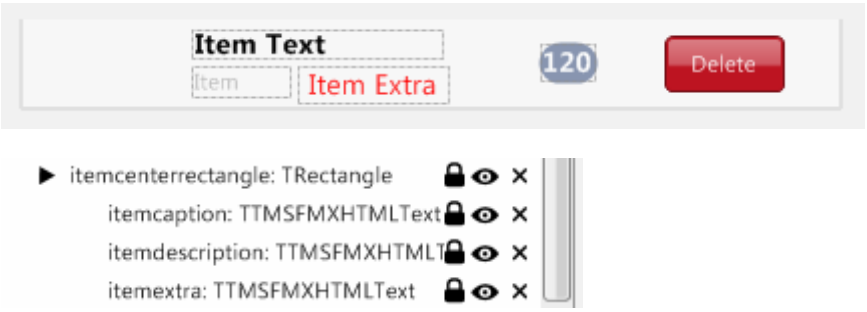
procedure TTMSFMXTableViewItemEx.SetExtra(const Value: string);
begin
  if FExtra <> Value then
  begin
    FExtra := Value;
    if TableView.GetUpdateCount = 0 then
      UpdateExtra;
  end;
end;

function TTMSFMXTableViewItemEx.ShapeExtra: TText;
var
  shp: TTMSFMXTableViewItemShape;
begin
  Result := nil;
  shp := Shape;
  if Assigned(shp) then
    Result := shp.FindStyleResource('itemextra') as TText;
end;

procedure TTMSFMXTableViewItemEx.UpdateExtra;
var
  shp: TText;
begin
  shp := ShapeExtra;
  if Assigned(shp) then
    shp.Text := Extra;
end;

end.
```

When dropping this TTMSFMXTableViewEx on the form, you will notice that it has the same appearance as the normal TTMSFMXTableview. This is because the extra element needs to be added to the list. Edit the custom style and add a new TTMSFMXHTMLText to the itemcenterrectangle so it looks like the item appearance below:

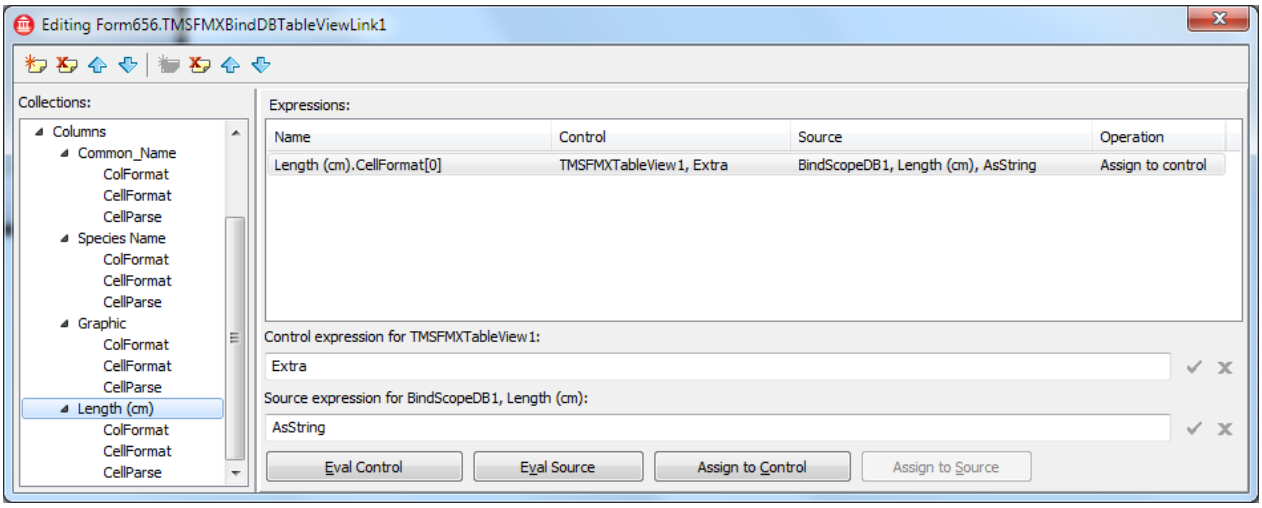


Set the stylename of that item to “itemextra” since that is the name that is used in the TTMSFMXTableViewEx component to access this element.

In the second demo the list is loaded with data from the biolife.xml and displaying an additional field in the item:



If you open the BindingsList editor and open the BindDBTableViewLink expression editor you will notice an extra column is now added and linked to the Extra property that is available in this extended TTMSFMXTableViewEx component.



This way of introducing properties can be done for other properties and can be combined with various elements to allow displaying all fields in the tableview component.

FireMonkey Styles

The StyleBook is not only the placeholder of customizations of the style of a component in the application. It can also hold the entire set of styles for the full FireMonkey standard controls set. This means that you can load in the StyleBook the entire set of styles for all controls to give your application a completely different look.

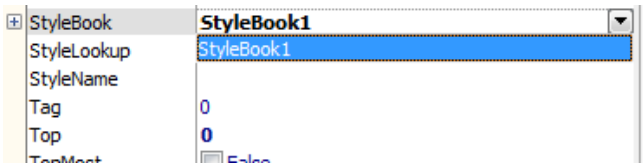
The standard FireMonkey styles are found in:

C:\Program Files (x86)\Embarcadero\RAD Studio\9.0\Redist\styles\Fmx

In the distribution you will see various .Style files in the FireMonkey Styles folder. These style files can be used to either style an individual component or a complete form. Unfortunately, at least till the XE2 Update 3 release, there is an issue that only one Style file can be used at a time. Loading a different style will overwrite the existing style information in the StyleBook. This implicates that it is currently not possible to provide a separate style file for a 3rd party component set that can be loaded without loosing style information for standard FireMonkey controls or other 3rd party components.

Therefore, we have merged our component specific style files with the default FireMonkey controls style files and have created a style file that can be used for all standard FireMonkey components and the TMS TableView.

These files can be found in the distribution and are stored in separate folders.
To use these files you must drop a StyleBook on the form, load a style file of choice and set the StyleBook property of the form:

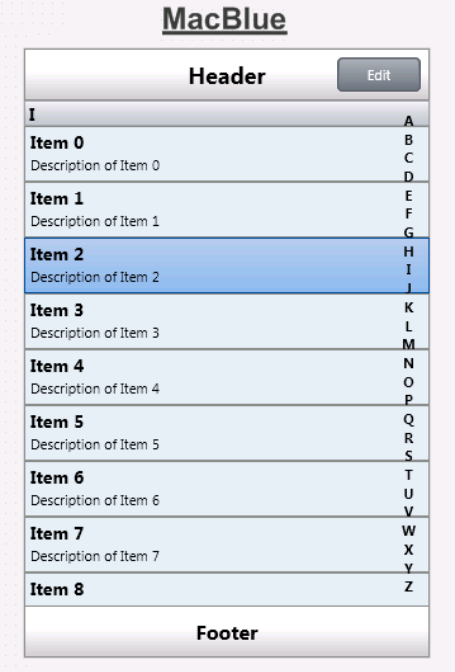


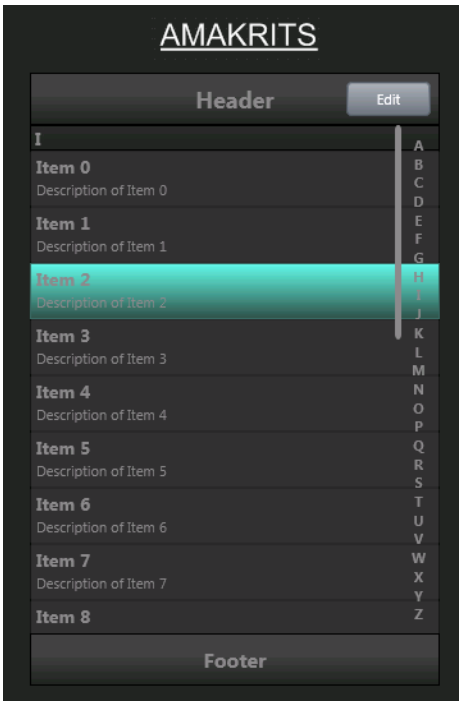
The appearance of the form will automatically change after applying the style.

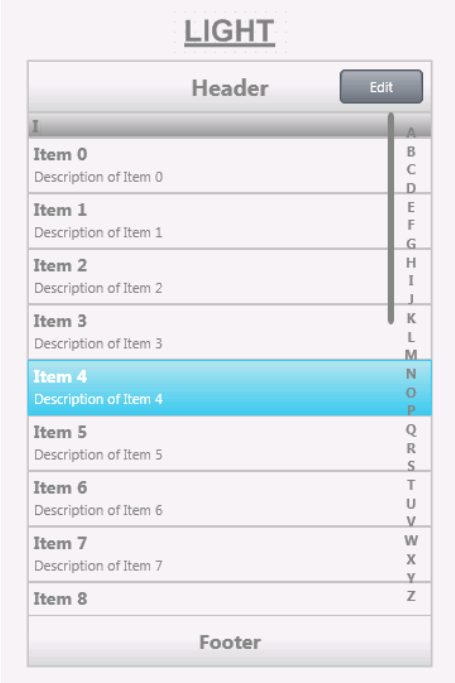
As a part of the distribution, we have for now added component style equivalents for the first styles:

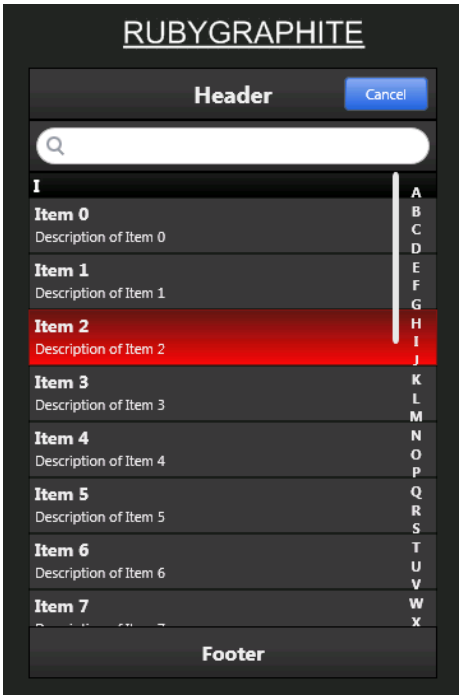
- Air
- Amakrits
- AquaGraphite
- Blend
- GoldenGraphite
- Light
- RubyGraphite
- Windows7
- MacBlue
- iOS
- Dark

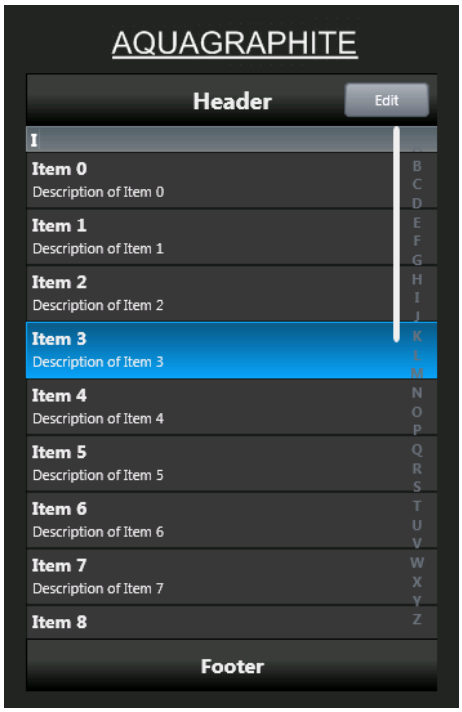
We're working on style files for our components to make these consistent with other stock FireMonkey style files. After applying styles you will notice that the appearance on the TMS TableView has changed. Below is a sample of our controls after applying different styles.

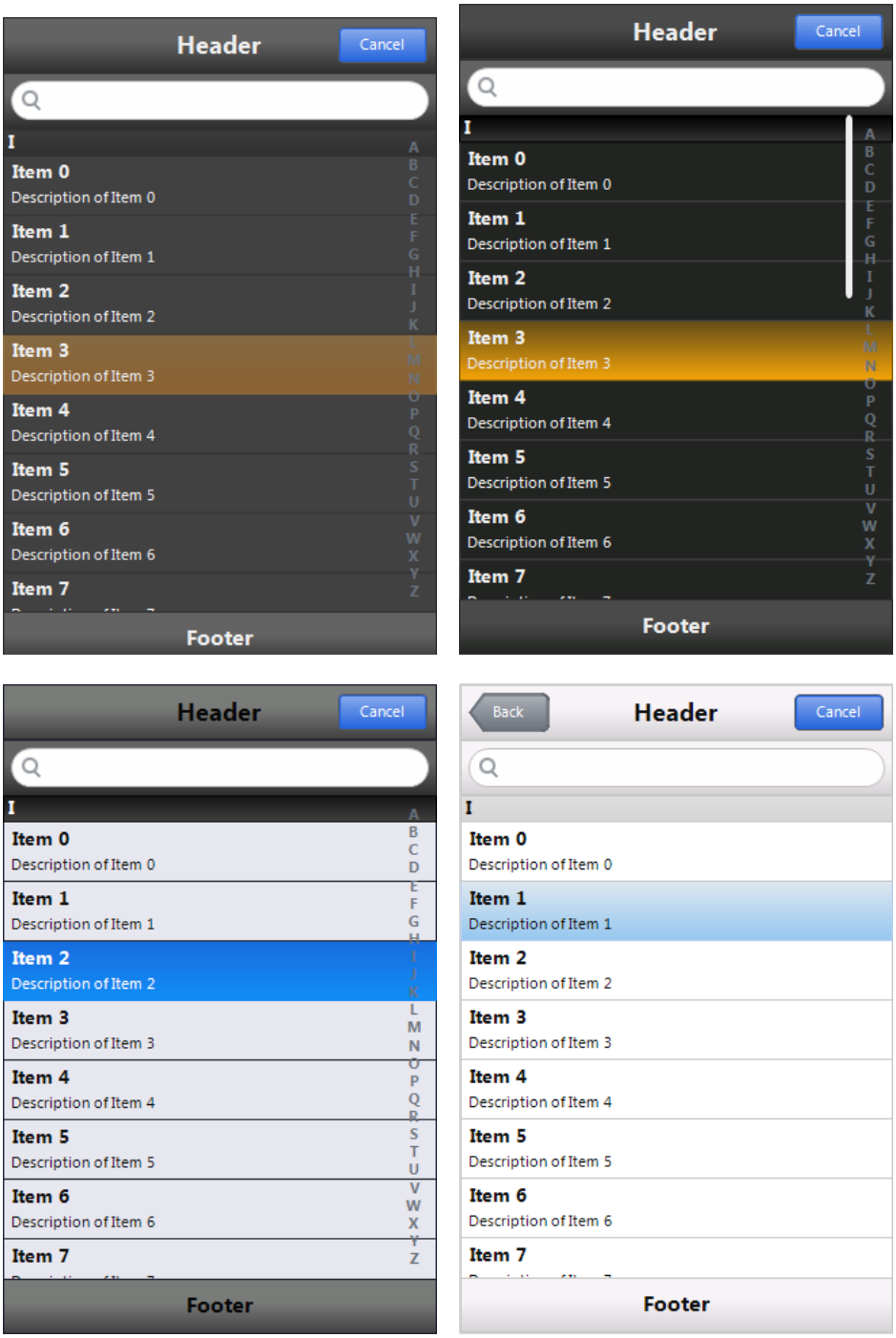


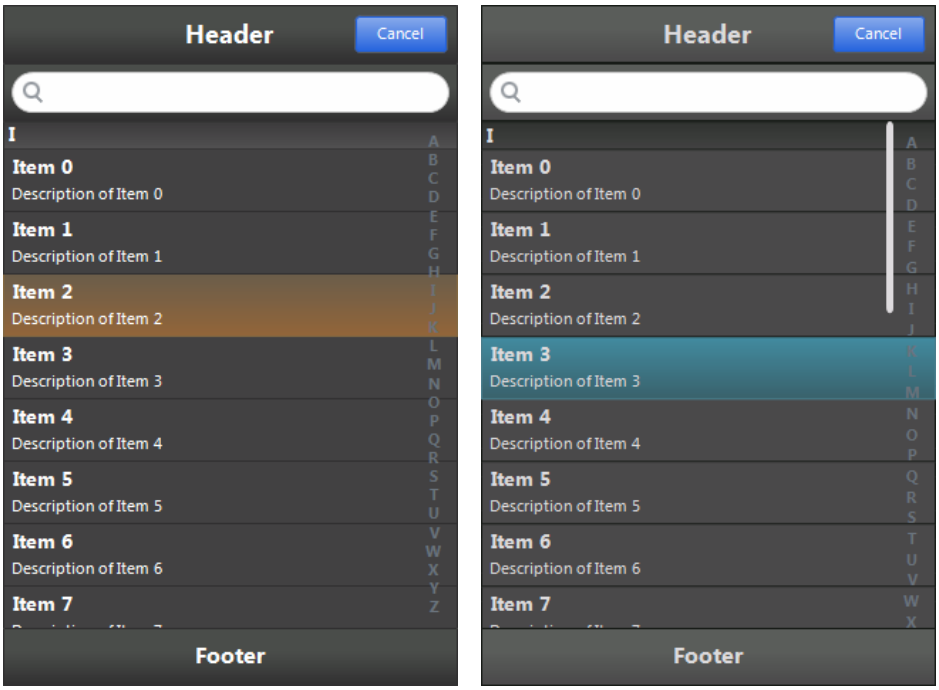

AMAKRITS
Header
Edit
I
Item 0
Description of Item 0
Item 1
Description of Item 1
Item 2
Description of Item 2
Item 3
Description of Item 3
Item 4
Description of Item 4
Item 5
Description of Item 5
Item 6
Description of Item 6
Item 7
Description of Item 7
Item 8
Footer
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z


LIGHT
Header
Edit
I
Item 0
Description of Item 0
Item 1
Description of Item 1
Item 2
Description of Item 2
Item 3
Description of Item 3
Item 4
Description of Item 4
Item 5
Description of Item 5
Item 6
Description of Item 6
Item 7
Description of Item 7
Item 8
Footer
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z


RUBYGRAPHITE
Header
Cancel
I
Item 0
Description of Item 0
Item 1
Description of Item 1
Item 2
Description of Item 2
Item 3
Description of Item 3
Item 4
Description of Item 4
Item 5
Description of Item 5
Item 6
Description of Item 6
Item 7
Footer
A B C D E F G H I J K L M N O P Q R S T U V W X


AQUAGRAPHITE
Header
Edit
I
Item 0
Description of Item 0
Item 1
Description of Item 1
Item 2
Description of Item 2
Item 3
Description of Item 3
Item 4
Description of Item 4
Item 5
Description of Item 5
Item 6
Description of Item 6
Item 7
Description of Item 7
Item 8
Footer
B C D E F G H I J K L M N O P Q R S T U V W X Y Z


Header
Cancel
I
Item 0
Description of Item 0
Item 1
Description of Item 1
Item 2
Description of Item 2
Item 3
Description of Item 3
Item 4
Description of Item 4
Item 5
Description of Item 5
Item 6
Description of Item 6
Item 7
Footer
Back


Header
Cancel
I
Item 0
Description of Item 0
Item 1
Description of Item 1
Item 2
Description of Item 2
Item 3
Description of Item 3
Item 4
Description of Item 4
Item 5
Description of Item 5
Item 6
Description of Item 6
Item 7
Footer
Header
Cancel
I
Item 0
Description of Item 0
Item 1
Description of Item 1
Item 2
Description of Item 2
Item 3
Description of Item 3
Item 4
Description of Item 4
Item 5
Description of Item 5
Item 6
Description of Item 6
Item 7
Footer