1. Widget Development / Front End . . . .

Widget Development / Front End

Widget Developer's Guide

Welcome

Welcome to the ETNA Trader Widget Developer’s Guide. This guide introduces the key concepts, tools, and libraries you’ll encounter when building widgets for ETNA Trader Web Front End. The topics in this guide span project organization, coding, debugging, testing, optimizing, and publishing your widget.

In order to understand the concepts in this guide you need to be familiar with the following technologies:

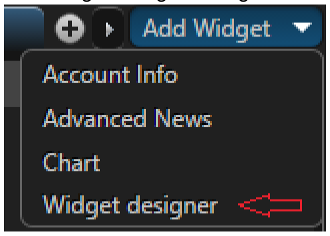
- HTML5
- JavaScript
- CSS

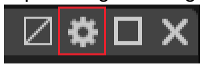
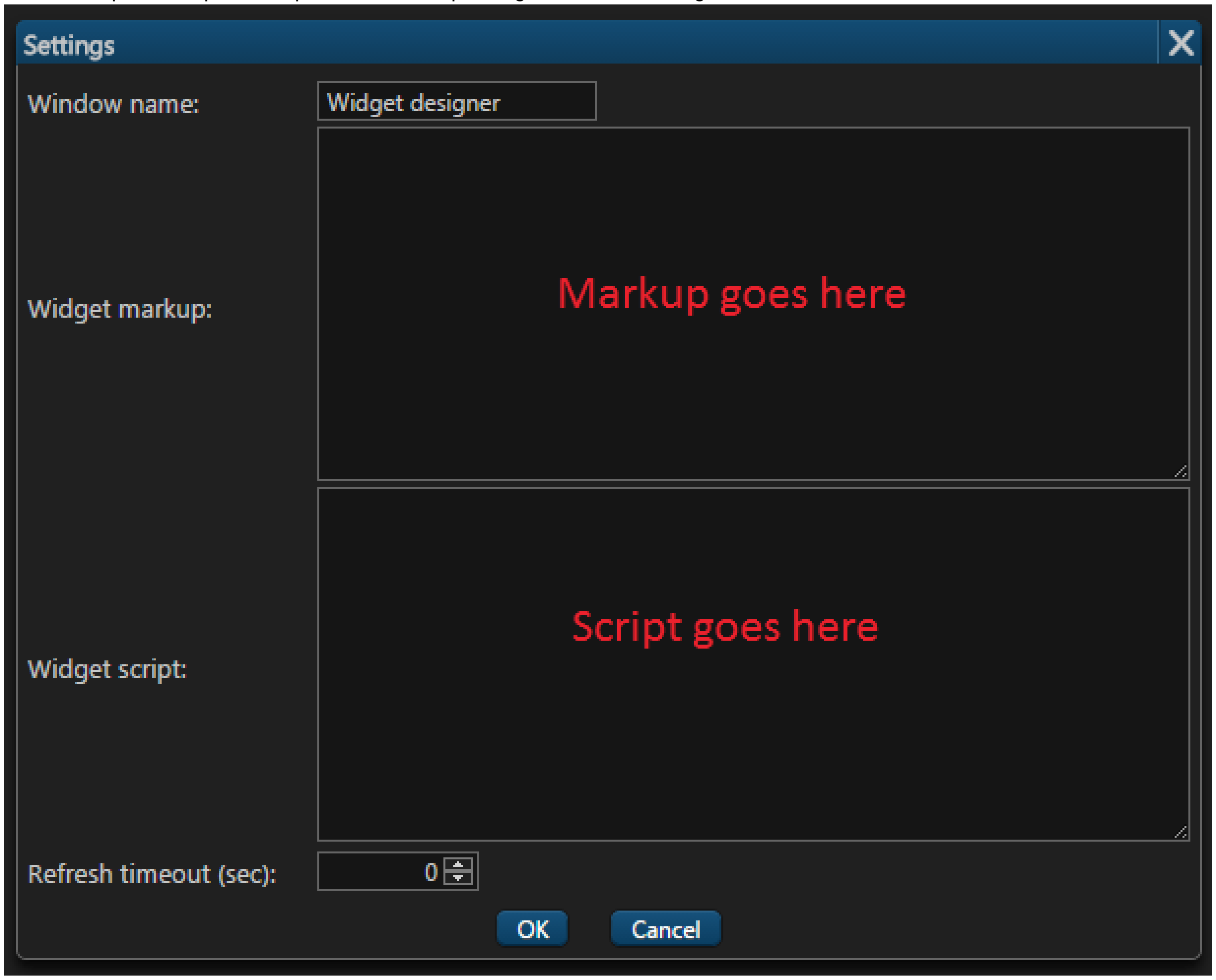
It may also help if you are familiar with the concepts of securities trading.

1. Introduction

Using a Widget Designer

All of these examples are based on *Widget designer* widget. To reproduce this particular example, you must first prepare the workspace:

1. Add *Widget designer* widget


2. Open widget settings

3. Insert example markup and script into the corresponding fields in the settings


API Access

ETNA Traders' back-end exposes various data retrieval methods that allows developers to create powerful widgets. Most methods are available via POST requests, but some of them support only GET. The common way to access those methods is to use the *Service* JS component. It allows developers to perform POST requests to previously registered endpoints. Here's how to use it:

```
window.Service.[ControllerName].[MethodName](requestParams)
```

That call makes use of the *jQuery.ajax* method and results in a *jqXHR* object which can be used to subscribe and to request done/fail/etc. You can find more info on jQuery.ajax

For methods that supports GET, it is recommended to use the *ajax* JS component. The usage scheme is the following:

```
window.ajax.get([RequestURL], [DoneCallback], [ErrorCallback], requestParams);
```

Example (get all existing positions):

```
requestParams = {
 Take: -1,
    Page: -1,
    SortField: null,
    SortDirection: null
};
window.Service.Position.Read(requestParams).done(function(positions) { /* handle
positions */ });
```

Common available methods are:

- ***[POST] Position.Read***

  Accepts: object with the following fields
    - Take - page size (default: -1)
    - Page - page number (default: -1)
    - SortField - name of sorting field (default: null)
    - SortDirection - 'asc' or 'desc' (default: null)

  Returns: object with the following fields

-
    - AccountId - ID of account
    - Positions - array of position items
    - total - total amount of positions

- ***[POST] Order.GetOrders***

  Accepts: object with the following fields
    - SecurityId - ID of target security (default: null)
    - Status - target order status (default: null)
    - Take - page size (default: -1)
    - Page - page number (default: -1)
    - SortField - name of sorting field (default: null)
    - SortDirection - 'asc' or 'desc' (default: null)

  Returns: object with the following fields

-
    - accountId - ID of account
    - orders - array of order items
    - total - total amount of orders

- ***[POST] MultipleWatchList.Get***

  Accepts: no parameters

  Returns: array of all existing watchlists

- ***[POST] Account.GetBalanceInfo***

  Accepts: no parameters

  Returns: string with serialized pairs "Attribute-value" of balance parameters

- ***[GET] Security.GetByMask***

  Accepts: object with the following fields
  - mask - search mask (default: null)
  - count - search count (default: 0)

  Returns: array of matching securities

Streaming

ETNA Trader offers a streaming engine that delivers various types of real-time data to the platform. The common way to subscribe to some data is the following:

```
window.streamingClient.subscribe([StreamingEntity], [SubscriptionKey],
[OnDataCallback]);
```

List of all supported entities might be found in the object *window.StreamerEntityType*. The most frequently used entities are symbol quotes (in quote and candle format), orders and positions. The only difference between subscriptions is the subscription key. Here are the list of some used keys:

Streaming entity	Subscription key format
Quote	symbol (e.g. AAPL)
Candle	symbol, exchange and currency, delimited with "|" symbol, and followed by colon and period (e.g. AAPL|NSDQ|USD:1m)
Order	account ID (e.g. 276)
Position	account ID (e.g. 276)
AccountBalance	account ID (e.g. 276)

MessageBus

MessageBus is the infrastructure that allows different parts of platform to communicate between each other (e.g. notify listeners on account change). Supported message types are available via the *window.MessageBusEvents* object. The subscription scheme is the following:

```
window.MessageBus.subscribe([Event], [Callback], [AdditionalParameters])
```

AdditionalParameters is a regular JS object. Right now, only *retrieveValue* key is supported, which allows developers to get the current value right after subscription without waiting for the specified event. To submit a message the following syntax is used:

```
window.MessageBus.submit([Event], [Parameters]);
```

Parameters specified on *submit* will be passed to the callback.

Example (account change):

Submit

```
window.MessageBus.submit(window.MessageBusEvents.ChangeAccount, accountId);
```

Subscribe

```
window.MessageBus.subscribe(window.MessageBusEvents.ChangeAccount, onAccountChanged, {
retrieveValue: true })
```

List of all currently availbale subscription keys might be found below:

```
window.MessageBusEvents = {
    LoadChart: 'LOAD_CHART',
    LoadLayout: 'LOAD_LAYOUT',
    LoadTheme: 'LOAD_THEME',
    ChangeSymbol: 'CHANGE_SYMBOL',
    ChangeAccount: 'CHANGE_ACCOUNT',
    ChangeAccentColors: 'CHANGE_ACCENT_COLORS',
    ChangeRoomSettings: 'CHANGE_ROOM_SETTINGS',
    EditableChanged: 'EDITABLE_CHANGED',
    RoomChanged: 'ROOM_CHANGED',
    AvatarChanged: 'AVATAR_CHANGED',
    AccountAliasChanged: 'ACCOUNT_ALIAS_CHANGED',
    OpenUserSettings: 'OPEN_USER_SETTINGS',
    SafeModeChanged: 'SAFE_MODE_CHANGED'
};
```

2. Market Data

Streaming

To demonstrate the streamers functionality, subscription to the AAPL candle will be used. This example consists of two steps: search of the AAPL symbol via an AJAX request and then a subscription to its' candle.

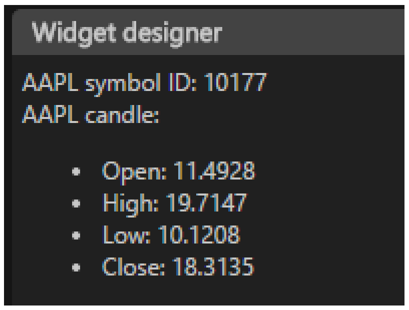
Markup

```
<label>AAPL symbol ID: <span id="aaplId"></span></label>
<br />
AAPL candle:
<ul>
    <li>Open:  <span id="aaplOpen"></span></li>
    <li>High:  <span id="aaplHigh"></span></li>
    <li>Low:   <span id="aaplLow"></span></li>
    <li>Close: <span id="aaplClose"></span></li>
</ul>
```

Script

```
(function() {
    var openSpan = document.getElementById('aaplOpen'),
        highSpan = document.getElementById('aaplHigh'),
        lowSpan = document.getElementById('aaplLow'),
        closeSpan = document.getElementById('aaplClose');

 function onAaplCandle(candle) {
     openSpan.innerHTML = candle.Open;
     highSpan.innerHTML = candle.High;
     lowSpan.innerHTML = candle.Low;
     closeSpan.innerHTML = candle.Close;
 }

 function subscribeToSecurity(security) {
  var candleSignature = String.format('{0}|{1}|{2}:1m', security.Symbol,
security.Exchange, security.Currency);

  window.streamingClient.subscribe(window.StreamerEntityType.Candle, candleSignature,
onAaplCandle);
 }

 function handleSecuritySearchRequest(data) {
  var securities = JSON.parse(data);

  document.getElementById('aaplId').innerHTML = securities[0].Id;

  subscribeToSecurity(securities[0]);
 }

 //form and perform request for securities that contain 'aapl'
 requestParams = {
  mask: 'aapl',
  count: 1
 };

 window.ajax.get('/Security/GetByMask', handleSecuritySearchRequest, null,
requestParams);
})();
```

If everything is alright, the following result should be shown (values may vary):


Widget designer

AAPL symbol ID: 10177
AAPL candle:

- Open: 11.4928
- High: 19.7147
- Low: 10.1208
- Close: 18.3135

For multiple securities subscription please use ';' as delimiter of signatures. You can find example below:

```
function subscribeToSecurities(securities) {

 var signatures = [];
 for (var i = 0; i < securities.length; i++) {
  signatures.push(String.format('{0}|{1}|{2}:1m', securities[i].Symbol,
securities[i].Exchange, securities[i].Currency));
 }

 window.streamingClient.subscribe(window.StreamerEntityType.Candle,
signatures.join(";"), onCandleHandler);
}
```

To get security by symbol use SecurityCache

```
window.SecurityCache.getSecurityBySymbol(['AAPL', 'GOOG']).done(function (items){
console.log(items); })
```

3. Account Data

Accessing Balance Information

Retrieving current account information when user switches an account. It also displays the JSON info with names of available parameters

Markup

```
<label>Current account: <span id="accountId"></span></label>
<p/>
<label>Total equity: <span id="accountEquity"></span></label>
<p/>
<label>Stock buying power: <span id="stockBP"></span></label>
<p/>
<label>Option buying power: <span id="optionBP"></span></label>
<p/>
<label>BalanceInfo JSON string: <span id="jsonString"></span></label>
```

Script

```
function onAccountChange(data) {
        document.getElementById('accountId').innerHTML = data.CurrentAccount;
        window.Service.Account.GetBalanceInfo().done(setAccountBuyingPower);
    }

    function setAccountBuyingPower(json) {
        var items = JSON.parse(json.Items),
            accountEquity = items.filter(function (item) { return item.Name ==
'equityTotal'; })[0],
            stockBuyingPower = items.filter(function (item) { return item.Name ==
'stockBuyingPower'; })[0],
            optionBuyingPower = items.filter(function (item) { return item.Name ==
'optionBuyingPower'; })[0];

        document.getElementById('accountEquity').innerHTML = accountEquity.Value;
        document.getElementById('stockBP').innerHTML = stockBuyingPower.Value;
        document.getElementById('optionBP').innerHTML = optionBuyingPower.Value;
        document.getElementById('jsonString').innerHTML = JSON.stringify(json);
    }
    MessageBus.subscribe(MessageBusEvents.ChangeAccount, onAccountChange, {
retrieveValue: true });
```

Result

Listening to Account Switch Events

The example below demonstrates how to catch the event when a user switches his trading account.

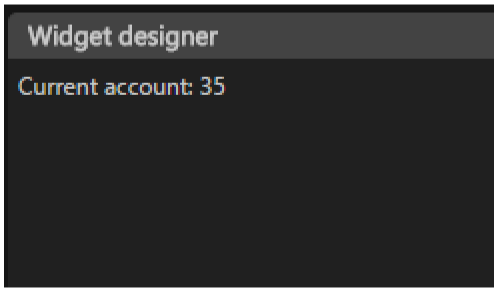
Markup

```
<label>Current account: <span id="accountId"></span></label>
```

Script

```
(function() {
    function onAccountChange(data) {
        document.getElementById('accountId').innerHTML = data.CurrentAccount;
    }

    MessageBus.subscribe(MessageBusEvents.ChangeAccount, onAccountChange, {
retrieveValue: true });
})();
```

If everything is alright, the following result should be shown (account ID may vary):



Current account: 35

Accessing Positions

Retrieves current account information when user switches an account and displays current balance, positions and watch lists.

Markup

```
<label>Current account: <span id="accountId"></span></label>
<p/>
<label>BalanceInfo JSON: <span id="jsonBalanceInfo"></span></label>
<p/>
<label>Positions JSON: <span id="jsonPositions"></span></label>
<p/>
<label>All WatchLists JSON: <span id="jsonWatchLists"></span></label>
<p/>
<label>First WatchList JSON: <span id="jsonFirstWatchList"></span></label>
```

Script

```
Service.register('Positions.Load', '/Position/Read');
    Service.register('WatchList.Get', '/MultipleWatchList/Get');
    Service.register('WatchList.GetSymbols', '/MultipleWatchList/GetSymbolsPaged');
    MessageBus.subscribe(MessageBusEvents.ChangeAccount, onAccountChange, {
retrieveValue: true });
    function onAccountChange(data) {
        document.getElementById('accountId').innerHTML = data.CurrentAccount;
        window.Service.Account.GetBalanceInfo().done(setAccountBuyingPower);
        window.Service.Positions.Load().done(setPositions);
        window.Service.WatchList.Get().done(setWatchLists);
    }
    function setPositions(json) {
        document.getElementById('jsonPositions').innerHTML = JSON.stringify(json);
    }
    function setWatchLists(json) {
        document.getElementById('jsonWatchLists').innerHTML = JSON.stringify(json);

window.Service.WatchList.GetSymbols({"listId":json[0].Id,"startIndex":0,"pageSize":30,
"sortField":"Symbol","sortDirection":"asc"}).done(setFirstWatchList);
    }
    function setFirstWatchList(json) {
        document.getElementById('jsonFirstWatchList').innerHTML =
JSON.stringify(json);
        window.Service.Security.ResolveSecurities(json.Items).done(resolveSecurities);

    }
    function resolveSecurities(json) {
        document.getElementById('jsonFirstWatchList').innerHTML =
JSON.stringify(json);
    }

    function setAccountBuyingPower(json) {
        var items = JSON.parse(json.Items),
            optionBuyingPower = items.filter(function (item) { return item.Name ==
'optionBuyingPower'; })[0];

        document.getElementById('jsonBalanceInfo').innerHTML = JSON.stringify(json);
    }
```

Result

Subscribing to Positions updates

Retrieves current account information when user switches an account and subscribes to positions changes via Streaming service

Markup

```
<label>Current account: <span id="accountId"></span></label>
</p>
<label>Last updated position: <span id="jsonLastPosition"></span></label>
</p>
<label>All Positions: <span id="jsonPositions"></span></label>
```

Script

```
Service.register('Position.Read', '/Position/Read');
MessageBus.subscribe(MessageBusEvents.ChangeAccount, onAccountChange, { retrieveValue:
true });

function onAccountChange(data) {
        document.getElementById('accountId').innerHTML = data.CurrentAccount;
        window.Service.Position.Read().done(setPositions);
        window.streamingClient.subscribe(window.StreamerEntityType.Position,
data.CurrentAccount, onPositionsChange);
}
function onPositionsChange(data) {
        document.getElementById('jsonLastPosition').innerHTML = JSON.stringify(data);
        window.Service.Position.Read().done(setPositions);
}
function setPositions(data) {
    document.getElementById('jsonPositions').innerHTML = JSON.stringify(data);
}
```

Result

3. Watch Lists

Accessing Watch Lists

Retrieves current account information when user switches an account and displays current balance, positions and watch lists.

Markup

```
<label>Current account: <span id="accountId"></span></label>
<p/>
<label>All WatchLists JSON: <span id="jsonWatchLists"></span></label>
<p/>
<label>First WatchList JSON: <span id="jsonFirstWatchList"></span></label>
```

Script

```
Service.register('WatchList.Get', '/MultipleWatchList/Get');
    Service.register('WatchList.GetSymbols', '/MultipleWatchList/GetSymbolsPaged');
    MessageBus.subscribe(MessageBusEvents.ChangeAccount, onAccountChange, {
retrieveValue: true });
    function onAccountChange(data) {
        document.getElementById('accountId').innerHTML = data.CurrentAccount;
        window.Service.WatchList.Get().done(setWatchLists);
    }
    function setWatchLists(json) {
        document.getElementById('jsonWatchLists').innerHTML = JSON.stringify(json);

window.Service.WatchList.GetSymbols({"listId":json[0].Id,"startIndex":0,"pageSize":30,
"sortField":"Symbol","sortDirection":"asc"}).done(setFirstWatchList);
    }
    function setFirstWatchList(json) {
        document.getElementById('jsonFirstWatchList').innerHTML =
JSON.stringify(json);
        window.Service.Security.ResolveSecurities(json.Items).done(resolveSecurities);

    }
    function resolveSecurities(json) {
        document.getElementById('jsonFirstWatchList').innerHTML =
JSON.stringify(json);
    }
```

Result

4. User Information

Identifying ETNA Trader Instance

Information about current company can be accessed as properties of window.Company object.

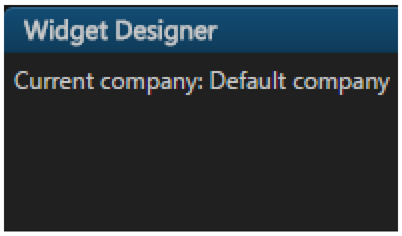
Markup

```
<label>Current company: <span class="companyName"></span></label>
```

Script

```
(function() {
   $('.companyName').text(window.Company.name);
})();
```

If everything is alright, the following result should be shown:



Current company: Default company

6. Trading

"Trade Ticket" with pre-populated data

Stock Message format ("StockTradeOrder")

Format	Description

```
{
 Legs :[{
 SecurityId: int,
 SecurityName: string,
 Quantity: int
 }];
 Price: float,
 StopPrice: float,
 OrderType: string,
 TimeInForce: string,
 AllOrNone: bool,
 Exchange: string,
 UseActualPrice: bool,
 params: {}
};
```
	Legs - single element array.
	SecurityId- security id.
	SecurityName - security symbol.
	Quantity - number of shares.
	Price - limit price.
	StopPrice - stop price.
	
	OrderType - type of order. Enumerable
	OrderType = {
	Market: 'Market',
	Limit: 'Limit',
	Stop: 'Stop',
	StopLimit: 'StopLimit',
	TrailingStop: 'TrailingStop',
	TrailingStopLimit: 'TrailingStopLimit',
	OTO: "OneTriggerOther",
	OCO: "OneCancelOther",
	All: 'All'
	};
	
	TimeInForce - time in force. Enumerable
	OrderExpiration = {
	Day: 'Day',
	GoodTillCancel: 'GoodTillCancel'
	};
	
	AllOrNone - all or none.
	Exchange - exchange name.
	UseActualPrice - set to false if price is specified explicitly.
	params - put additional params to the message. These params will not be checked

example:

```
var message = new StockMessage();

        message.Legs = [{
            SecurityId: 6,
            SecurityName: 'MSFT',
            Quantity: DefaultSettingsManager.stocks()
        }];
        message.OrderType = OrderType.Market;
        message.TimeInForce = DefaultSettingsManager.durationType();
        message.AllOrNone = DefaultSettingsManager.allOrNone();
        message.Exchange = DefaultSettingsManager.exchangeType();

        MessageProvider.putMessage(MessageTypes.StockTradeOrder, message);
```

Option Message format ("OptionTradeOrder")

Format	Description

```
{
 stock:{
 Id: int,
 Symbol: string
 },
 options:[{
 strike: float,
 type: string,
 action: string,
 expiration: string,
 name: string,
 quantity: int
 }],
 params: {}
};
```
	stock - underlying security.
	Id - security id.
	Symbol - security symbol.
	options - order legs.
	strike - strike price.
	
	type - option type. Enumerable
	OptionTypes =
	{
	Call: "Call",
	Put: "Put"
	};
	
	action - buy/sell parameter. Enumerable
	ActionType = {
	Buy: 'Buy',
	Sell: 'Sell',
	SellShort: 'SellShort',
	BuyToCover: 'BuyToCover',
	None: 'None'
	};
	
	expiration - expiration date.
	name - option symbol.
	quantity - number of shares.
	
	params - put additional params to the message. These params will not be checked

example:

```
var message = {
  "stock": {
    "Id": 55,
    "Symbol": "AAL",
  },

  "options": [
  {
    "strike": 25,
    "type": "Call",
    "action": "Buy",
    "expiration": new Date("2015-08-21T00:00:00.000Z"),
    "name": ".AAL 150821C00025000",
    "quantity": 1
  },
  {
    "strike": 26,
    "type": "Call",
    "action": "Buy",
    "expiration": new Date("2015-08-21T00:00:00.000Z"),
    "name": ".AAL 150821C00026000",
    "quantity": 1
  }
  ],
  "params": {
    "OrderType": "Market",
    "TimeInForce": "Day",
    "AllOrNone": false,
    "Exchange": "Auto"
  }
};
MessageProvider.putMessage(MessageTypes.OptionTradeOrder, a);
```

7. Layout and Settings

Widgets' grouping

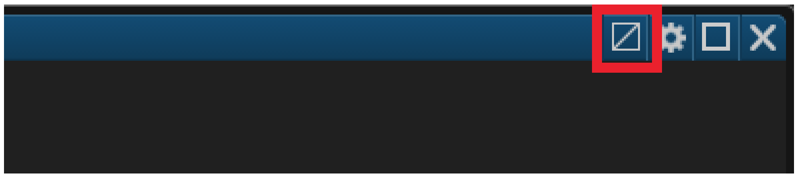
The platform allows to combine widgets into groups so they can share some common resources, e.g. selected security. To allow grouping for the particular widget, widget viewmodel should be inherited from *GroupableWidgetModel* in a following way:

```
window.ExampleWidgetViewModel = function (parameters) {
 window.GroupableWidgetModel.call(this, [WidgetContainerDOMObject], [WidgetId],
[WidgetSettings]);
 //some code here
};
```

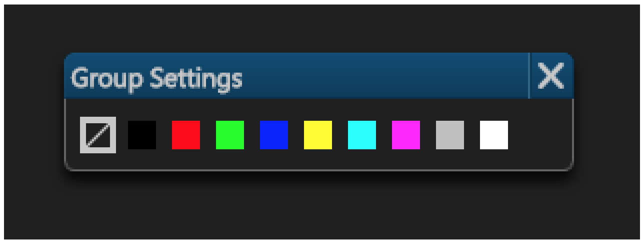
By calling *GroupableWidgetModel* via *call* method and passing *this* as an execution context, current context becomes extended with functions and properties necessary for grouping. To initialize grouping, *loadSettings* method should be called upon widget initialization:

```
function initialize() {
 //some initialization routines here
}

this.loadSettings(parameters.settings, function (loadedSettings) {
    //handle settings

    initialize();
});
```

This call will set default or saved group for current widget and thus widget becomes fully functional from grouping point of view. If everything is alright, new widget should have group icon in top right corner:



This icon means that current widget does not belong to any group. To assign (or join) group, you need to click this icon and select color from dialog:



Once group is assigned, widget might be subscribed to the group symbol change event:

```
function onSymbolChangeCallback(security) {
 //handle new security here
};

this.subscribeOnSymbolChange(onSymbolChangeCallback);
```

Current security is available via *this.symbol* at any time.

Widget settings

The widgets' infrastructure allows developers to add various settings for widgets. Generally, "Widget name" setting is added by default. To implement own settings, you need to override *getSettings* function. This function should define a list of necessary settings in a following format:

```
var settings = {
 [SettingName1]: { [SettingParameters] },
 ...,
 [SettingNameN]: { [SettingParameters] }
};
```

Settings parameters is a limited set of key-value pairs that allow to confiure setting behaviour. The list of common parameters might be found in a table below:

Parameter name	Parameter value	Required
value	current setting value	true
type	string, indicating settings type	true
name	settings name reflected on UI	true
captionWidth	width of setting caption	false
validator	validation function	false

Also settings parameters might contain setting-specific parameters (e.g. maxLength for text settings). Existing setting types are shown below:

Type	Description
select	selectbox with specified options
font-style	font-style selectbox
font-size	font-size selectbox
font-family	font-family selectbox
input	regular text input
caption	text input with optional tooltip and non-empty value
number	spinner with specified parameters
checkbox	refular checkbox
inline-color	inline positioned color palette
color	color dialog
multiline	regular text-area

Along with the setting list, you should specify *onApply* callback for settings and return settings dialog configuration object. Generally, it is recommended to use the following *onApply* callback structure:

```
var callback = function (data) {
 if (data.Name) {
     thisRef.getJqContainer().text(data.Name);
    }

 //custom settings logic goes here

    thisRef.settingsProvider.saveSettings(JSON.stringify(data));
};
```

Settings dialog configuration object should have the following structure:

```
{ settings: [SettingsList], behavior: { onApply: [onApplyCallback1, ...,
onApplyCallbackN] }}
```

Note that *onApply* field is actually an array of callbacks. Summing up, an example of *getSettings* function may look like the following code snippet:

```
thisRef.getSettings = function () {
 var setts = {
     'Name': { value: thisRef.getJqContainer().text(), type: 'caption', name: 'Window
Name:' },
        'Background': { value: thisRef.background.peek(), type: 'color', name:
'Background:' }
    };

    var callback = function (data) {
     if (data.Name) {
         thisRef.getJqContainer().text(data.Name);
        }

     applySettings(data);

        thisRef.settingsProvider.saveSettings(JSON.stringify(data));
    };

 return { settings: setts, behavior: { onApply: [callback] }};
};
```

Modal Dialogs

Modal Dialog is a customizable popup dialog, that can display custom markup and launch custom script, which can interact with ETNA.Trader API and subscriptions just as Widget Designer can. All that you need is execute command window.modalWidget.open() and send additional settings as parameters of this command. Available commands are:

- markup: String, that contains all the markup, which you want to display on dialog layout.
- script: String or function declaration, that contains all the scripts, which will be executed right after dialog appears.
- dialog: Object, that contains all the settings to configure dialog. Overrides JQuery UI Dialog settings.
- behavior: Object, contains two arrays: onOpen and onClose:
  - onOpen: Array, contains function handlers, which will be executed after dialog opening and executing main script.
  - onClose: Array, contains function handlers, which will be executed after dialog close .

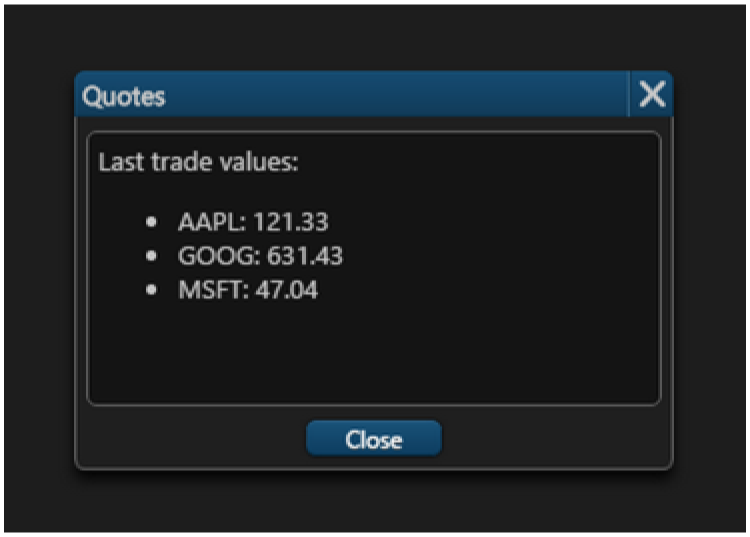
Note: All handlers have the same context, so you can transfer variables, functions etc. through "this" context. For Example:

```
this.variable = 1;
```

To launch the test script just execute command:

```
window.modalWidget.open({
 markup: '<label>Last trade values:</label><br /><ul><li>AAPL:  <span
id="aaplLast"></span></li><li>GOOG:  <span id="googLast"></span></li><li>MSFT:   <span
id="msftLast"></span></li></ul>',
 script: function() {
    var aaplLast = document.getElementById('aaplLast'),
    googLast = document.getElementById('googLast'),
    msftLast = document.getElementById('msftLast');
    var that = this;
    that.onQuoteHandler = function(quote) {
     switch(SecurityCache.getSecurity(quote.Key)().Symbol) {
      case 'AAPL':
       aaplLast.innerHTML = quote.Last;
       break;
      case 'GOOG':
       googLast.innerHTML = quote.Last;
       break;
      case 'MSFT':
       msftLast.innerHTML = quote.Last;
       break;
     }
    }
    window.SecurityCache.getSecurityBySymbol(['AAPL', 'GOOG', 'MSFT']).done(function
(items){
     that.signatures = [];
     for (var i = 0; i < items.length; i++) {
      that.signatures.push(String.format('{0}', items[i].Id));
     }
     window.streamingClient.subscribe(window.StreamerEntityType.Quote,
that.signatures.join(";"), that.onQuoteHandler);
    })
   },
 dialog: {
  width: 300,
  height: 200,
  title: 'Quotes'
 },
 behavior: {
  onClose: [function(){
   window.streamingClient.unsubscribe(window.StreamerEntityType.Quote,
this.signatures.join(";"), this.onQuoteHandler);
  }]
 }
});
```

The result will immediately appear on the screen:



Last trade values:

- AAPL: 121.33
- GOOG: 631.43
- MSFT: 47.04

Query String Parameters

There are several parameters, which can be passed via the query string to modify the display behavior of ETNA Trader. They can be used when only specific components need to be displayed in an IFRAME.

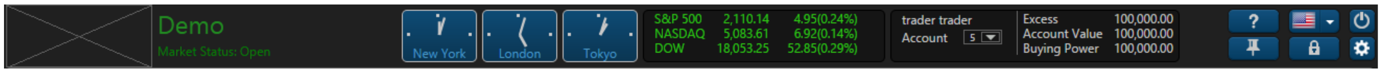
For example: http://demo.etnatrader.com/?safemode=1&showheader=0

showheader

Possible values: 0 or 1.

Default value: 1.

If parameter is set to 0, hides header panel.

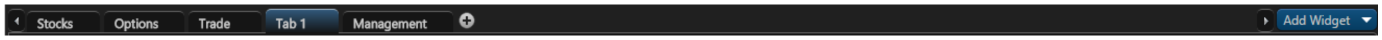


showtabs

Possible values: 0 or 1.

Default value: 1.

If parameter is set to 0, hides tabs panel.



showwidgetframe

Possible values: 0 or 1.

Default value: 1.

If parameter is set to 0, hides widgets' headers and backgrounds.

safemode

Possible values: 0 or 1.

Default value: 0.

Value 1 turns on safe mode, which prevents script execution in Widget designer.

This parameter has higher priority then checkbox in Widget Designer's options dialog.

tab

Possible value: name of the tab.

If there are several tabs with the same name in user's layout, first of them will be selected.

If there are no tabs with the passed name, last opened tab will be selected.

8. Special Cases

Twitter Widget

We have this script from Twitter (let me use formatted version):

```
<a class="twitter-timeline" href="https://twitter.com/RaiderTrader"
data-widget-id="532990979330936832">Tweets by @RaiderTrader</a>
```

```
!function(d, s, id) {
    var js,
        fjs = d.getElementsByTagName(s)[0],
        p = /^http:/.test(d.location) ? 'http' : 'https';

    if (!d.getElementById(id)) {
        js = d.createElement(s);
        js.id = id;
        js.src = p + "://platform.twitter.com/widgets.js";
        fjs.parentNode.insertBefore(js, fjs);
    }
}(document, "script", "twitter-wjs");
```

To apply fix we need to understand what operations given script performs. This code snippet is wrapped with self-executing function, which means that it will executed immediately once it is added to the page. The logics is pretty simple:

1. find all existing scripts on the page and take the first of them;
2. check if Twitter script already exists on the page:

a. if script exists – do nothing;

b. if no script exists – create script tag, set its' source to specific Twitter address and add this tag before script, that was found in step 1.

From how it works the first time, looks like it searches for links with class attribute value equals to "twitter-timeline" and does some magic to them. So it fails on further subsequent calls because Twitter script does not rendered and, plainly said, nothing happens at all. Obviously, to make it work we need to add Twitter script every time. So here comes the fixed version:

```
!function(d, s, id) {
   var e = d.getElementById(id);
   if (e) {
        e.parentNode.removeChild(e);
   }


   var js,
       fjs = d.getElementsByTagName(s)[0],
       p = /^http:/.test(d.location) ? 'http' : 'https';


   js = d.createElement(s);
   js.id = id;
   js.src = p + "://platform.twitter.com/widgets.js";
   fjs.parentNode.insertBefore(js, fjs);
}(document, "script", "twitter-wjs");
```

It is easy to notice the difference between initial and modified version: script existence is not checked anymore as this script is always removed from the document if it exists. In that case script will always be added to the document and will change links to a Twitter timeline.

Please note: it is important to understand that this fix cannot be treated as a general way to deal with such problems. This solution works only in that particular case.

Now to the subject.

Referencing Widget Designer DOM-context

It is possible to reference current DOM-context for the particular widget designer via some kind of hack:

Markup

```
<div id="iframe-placeholder"></div>
```

Script

```
$(function() {
    var placeholder = document.getElementById('iframe-placeholder'),
        root = placeholder.parentNode,
        testSessionId = window.Cookies.get('CurrentSession');

    root.removeChild(placeholder);
    var iframe = document.createElement('iframe');
    iframe.style.width = '100%';
    iframe.style.height = '100%';
    iframe.src = 'http://finance.yahoo.com/q?s=' + testSessionId;
    root.appendChild(iframe);
});
```

The idea is to use temporary placeholder to find root element for current widget. After that, placeholder element will be removed to ensure document consistency.