Dataprovider

Introduction

Welcome to the Wakanda Dataprovider API. The Dataprovider API is part of the Wakanda Application Framework (WAF), which includes all the APIs that are available on the client-side of Wakanda.

The Dataprovider works with data at a lower level than the Datasource manager (see Datasource). It is intended to simplify data access and is in charge of:

- sending the necessary REST requests to Wakanda Server according to the needs of the client-side application, in particular for the needs of both widgets and datasources.
- formatting the data received in a readable form.
- managing the entity cache intended to optimize the operation of Web applications. In other words, playing the role of proxy to datastore classes, entity collections, and entities from the server.

Suppose, for example, that you have a Page with a Grid widget displaying 20 entities. If you perform an "allEntities" type request, the Dataprovider sends the corresponding REST request to the server but only retrieves the first 40 entities (default setting) even if the request finds 200,000 entities. As the user scrolls the list of entities, the Dataprovider triggers the necessary requests. This functioning is completely automatic when the Dataprovider works with datasources and widgets.

The Dataprovider also provides an API that lets you work directly with entity collections and entities on the client-side, *almost* like you do on the server with . Using the Dataprovider API, you have direct access to the datastore data and can perform any type of processing while freeing yourself from the automatic functioning of the datasources.

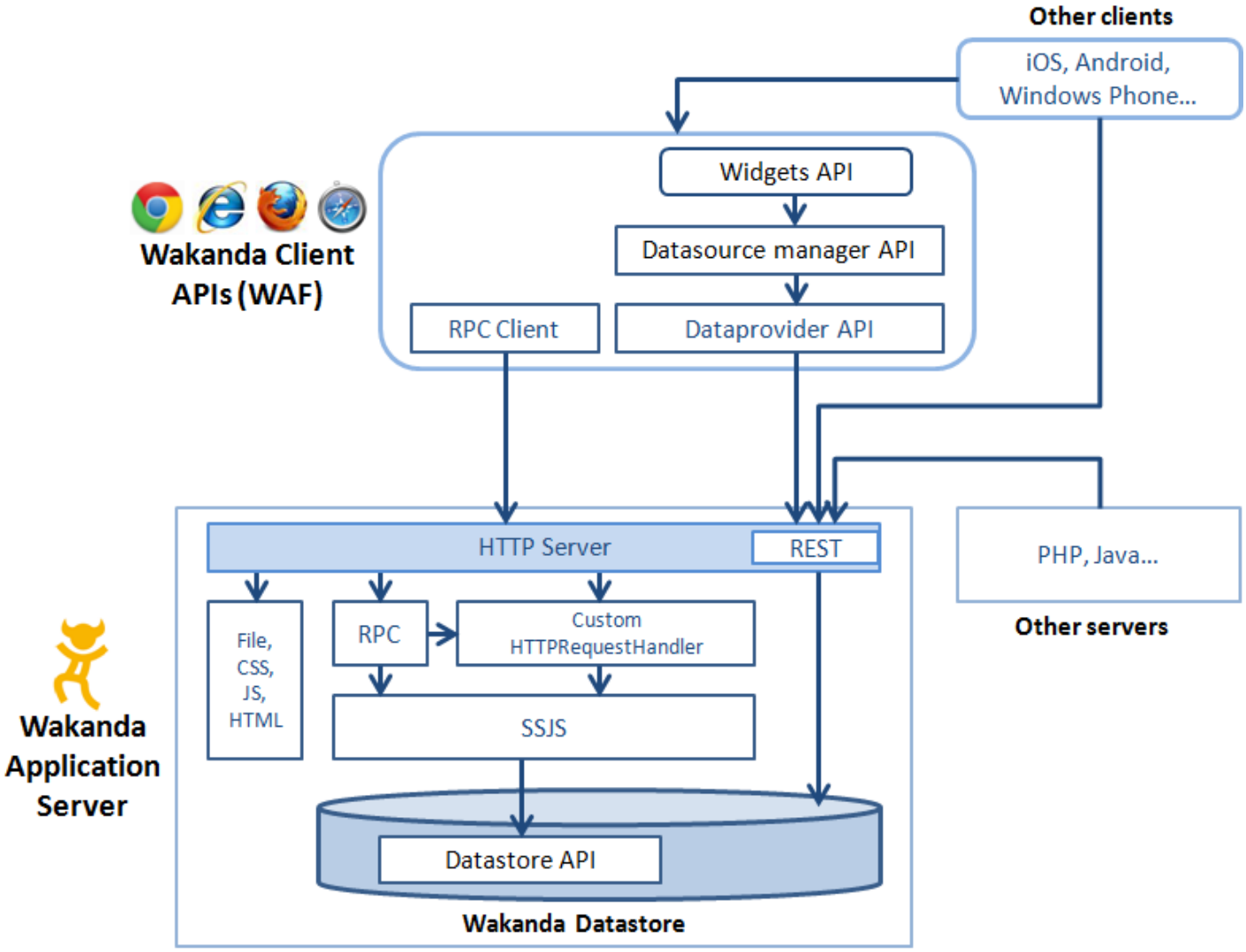
Note that there is one significant difference: you usually need to call methods asynchronously from the client in order to avoid freezing the browser. You can notice this difference in the syntax of the methods, which always work with blocks containing callback functions that are called when the server responds.

Integration into the Wakanda Architecture

The *Dataprovider API* is one of Wakanda's three **JavaScript data access APIs**. You can access it on the **client side** and it belongs to the overall Wakanda Application Framework (WAF) API. You use the methods of the Dataprovider API to manage the entity collections and the current datastore's entities directly. You generally execute them in asynchronous mode.

The two other JavaScript data access APIs are the *Datastore* and the *Datasource manager*. You access the Datastore API on the **server side**; it provides full access to the datastore models and data of the Wakanda applications on the server machine (see Datastore). You access the Datasource API on the **client side**; it provides high-level methods used to manage widgets and their associated datasources.

This diagram represents the architecture of the JavaScript data access APIs in Wakanda:



You choose which API to use, whether on the server or client, according to the business logic of your application and your model-related needs. More specifically:

- When you need to control how datasources operate, use the high-level methods of the *Datasource manager* (client API);
- When you need to perform advanced processing on client-side data, without necessarily displaying the data or enabling the automatic datasource mechanisms, use the methods of the *Dataprovider* (client API);
- When you need to directly process server-side datastore classes and data, use the *Datastore API* (server API).

Usually, your Wakanda applications use a combination of several different APIs.

Generally speaking, you must aim to minimize the number of requests to the server since these operations are likely to slow down the execution of your application. Therefore, it is recommended to run as much code as possible on the server using the Datastore and to only send requests to execute a function and retrieve the result. All operations related to the business logic must be run on the server, since the client APIs are responsible for data presentation and user interaction. For example, it is more efficient to sort a datastore class on the server and return the resulting entity collection than to sort the current entity collection of a datasource. You can use datastore class methods or remote procedure calls (RPCs) to execute code on the server.

Accessing Datastore Objects with the Dataprovider

In your JavaScript code, you will need to have access to datastore objects. In the Dataprovider API, you can designate the following objects directly using dynamic properties:

- the datastore itself,
- the datastore classes,
- the datastore class attributes.

Accessing a Datastore

You can access the current datastore object of the open project (i.e., the application that is running) using the ds property. This property is the "proxy" access to the datastore classes of the datastore through the Dataprovider.

For example, the following statement returns all the datastore classes specified in the datastore:

```
var allClasses = ds.getDataClasses();
```

Note: You can access the data classes of another datastore in the same solution using the WAF.DataStore.getCatalog( ) method.

Note that on the client side, the **ds** property allows specifying a call to the Dataprovider API ; to call the Datasource API, you should use the **source** property (as described in the Using Server Datasources section). It is important to know this difference because different objects (methods, properties, classes...) can have the same name in both APIs. For example, you can defined a "Company" datastore class and a "company" datasource (that can be or not based on the datastore class). In the code, both lines could be written:

```
var theDsClass = ds.Company; // refers to the datastore class
var theDatasource = sources.company; // refers to the datasource object
```

Accessing Datastore Classes

Each datastore class in the datastore is available directly on the client as a property of the Datastore object. For example:

```
var theTeachers = ds.Teacher; // returns the Teacher datastore class of the current datastore as an object
```

Once you have designated it like this, the datastore class becomes an object of the *DatastoreClass* class. These objects have specific properties and methods that are described in DataClass.

Accessing Datastore Class Attributes

On the client, the attributes of datastore classes can be accessed either directly as datastore class properties or through generic methods.

- Direct access: just like server-side in the Datastore API (see Accessing attributes of classes), on the client the attributes of datastore classes are DatastoreClassAttribute objects that you can access directly as properties of these classes. For example:

  ```
  var salary = ds.Employee.salary; // returns the salary attribute of the Employee class
  var compCity = ds.Company.city; // returns the city of the Company class
  ```

- The getAttributeByName( ) method returns the attribute of a datastore class as well. You can use this method for generic needs.
- The getAttributes( ) method returns the list of attributes for a datastore class.

Note that an object of type DatastoreClassAttribute is a description of the attribute object in the datastore class, and not the value of the attribute in each entity. To get the value of an attribute, you have to access an entity and then use the getValue( ) and setValue( ) methods on the attribute. The attributes themselves can be directly accessed as properties (see Using entity attributes).

For more information about the properties of type DatastoreClassAttribute, refer to the description of the getAttributeByName( ) method.

Executing Code on the Server

You will often need to execute code on the server. For example, when you process data or execute queries on the data in datastore classes and then send the result to the client or when you do a backup or import data. On the server, you have a complete Server-Side JavaScript (SSJS) API that enables you to access the data (Datastore API).

Wakanda offers many solutions, each with its own set of characteristics, advantages, and drawbacks, as shown in the table below. These solutions are as follows:

- Using JSON-RPC Services
- HTTP Server Request Handlers
- Using Datastore Class Methods or
- Using the client-side **callMethod( )**, available in both the Datasource (callMethod( )) and the Dataprovider (callMethod( )).

Comparing the possibilities for server-side code execution

This table shows the main similarities and differences between the various ways that you can execute server-side code in Wakanda.

Execution	Implementation	Synchronous/Asynchronous	Main Uses	Advantages/Limitations
JSON-RPC Services	Installation of function files on the server. Initialization of classes, called using client-side code	Optional: Synchronous or Asynchronous	Any type of use	Requires initializing WAF and instantiating the RPC class on the client.
HttpRequestHandler	Server-side installation of function files. Called using HTTP request	Set by context of the call on the client-side	Particularly suited for setting up "services" that can be accessed by any HTTP client.	Management of powerful "patterns." No library to initialize on the client-side. Possibility to modify function code without restarting the server.
Datastore class method	Direct call using JavaScript (client or server)	Synchronous or Asynchronous (depends on the syntax)	Any operation on the datastore class	Direct access to the data in the datastore class.
Datastore class event	Automatic on event call	Not applicable (no call on the client-side)	Applying or verifying datastore rules	Automatically executed. Direct access to the data in the datastore class.

Note: The explanation of how code can be executed synchronously or asynchronously is described below.

Synchronous or asynchronous execution?

The mode that a Web client uses to call a method on the server defines both the functioning of the Web application as well as the user experience. It also influences the programming necessary to coordinate access to values exchanged between server and client machines.

- When you call a server-side JavaScript method in **synchronous mode**, the code executed on the client is suspended where the request was sent while waiting for the server's response. This mode makes sure that the value returned by the function is available for the rest of the client-side script code. In this case, server-side processing must be quick so as to not slow down the HTML page. It is therefore a good idea to insert a synchronous call into a "try catch" JavaScript structure so that you can process any errors that may occur.
  Synchronous mode is reserved for particular cases and can only be used with Datastore class methods and RPC calls. In most of cases, asynchronous mode, which is more consistent with the "best practices" on the Web, is preferred.
- When you call a server-side JavaScript method in **asynchronous mode**, the code executed on the server and client is carried out independently. When execution requests are sent by the client, the script continues to process without waiting for the server's response. Asynchronous mode is highly recommended for Web interfaces since it does not block the client and thus preserves processing fluidity.
  In this mode, you manage server responses regardless of when they are received on the client. You retrieve this response in a callback function specified during the call to the main method. This function

is usually set using the onSuccess (called when successful) or onError (called when an error occurs) parameters. For more information, please refer to the Syntaxes for Callback Functions section.

For more information about asynchronous calls, please refer to the Principles of asynchronous execution section.

Principles of Asynchronous Execution

In the Wakanda Application Framework (WAF) API, calls to built-in methods usually involve sending requests to the server and waiting for a response. For example, when you query the entities in a datastore class using the query( ) method, a request is sent to the server and it responds by returning the resulting entity collection.

Therefore, client-side API methods must always be called **in asynchronous mode**. When requests are sent to the server, the script continues to run normally. When calling a method, you pass an *options* block containing a function that is called automatically when the server responds (also known as a **callback** function). The callback functions allow applications to run seamlessly regardless of the server's response time.

Note: In a hypothetical synchronous mode, requests would be sent to the server and, when a response is expected, the executing of the script would be suspended until the server responds. During this lapse of time, the entire browser would be blocked and no action could be performed. If the server response is delayed for whatever reason, the application becomes unusable. This operation does not conform to proper ergonomics in Web applications.

Executing an Asynchronous Call

To execute a request in **asynchronous** mode, you just have to pass at least one callback function using either an options object parameter or a direct function call (the presence of a callback function triggers the asynchronous mode). All the WAF API methods operate on this principle in asynchronous mode. The asynchronous syntax is in the following forms:

MethodName (*mandatory_parameters* , *options*)
or
MethodName (*mandatory_parameters* , *function1* [, *function2*] [, *options*])

Here is an example of an asynchronous call:

```
col.buildFromSelection(sel, { onSuccess: buildsel });  // asynchronous call
```

For more information about the asynchronous syntax, please refer to the Syntaxes for Callback Functions section.

Availability of Data in the Code

It is important to take into account the problems related to receiving data and the availability of data in asynchronous mode. For example, let's look at the following (incorrect) code:

Note: This code uses methods from the Dataprovider API, but the explained concepts are valid for the Datasource API as well.

```
// Example of incorrect code
var vcount;
var myset = ds.Person.query("ID > 100 and ID < 300", {
    onSuccess: function(event) // we pass a function that receives the server's response
    {
        vcount = event.entityCollection.length; // we retrieve the size of the entity collection
    }
    $("#display").html("selection : "+vcount);
        // we display the size of the entity collection in the container whose ID is "display"
});
```

This code does not produce the desired result because its execution is not linear and the expected values are not available at the correct time:
1. The client sends the request and moves on to the next statement.
2. The client displays the display container, which is empty because *vcount* is not (yet) available.
3. The server returns the entity collection and calls the callback function, but without processing anything.

In order for this code to work, you need to place the appropriate processing inside the callback functions, i.e., where the data is available. In this case, the correct code would be:

```
// Valid code
var vcount;
var myset = ds.Person.query("ID > 100 and ID < 300", {
    onSuccess: function(event) // we pass a function that receives the server response
    {
        vcount = event.entityCollection.length; // we retrieve the size of the entity collection
        $("#display").html("selection : "+vcount);
            // we display the size of the entity collection in the container whose ID is "display"
    }
});
```

Case of synchronous executions

As mentioned above, methods called in the Dataprovider API must be passed in asynchronous syntax. Synchronous calls are not suitable for Web applications.

However, you can pass methods directly without the options block in two cases:

- When you call a Datastore class method that does not send back a value (see Calling Datastore Class Methods);
- When you execute a method on the client that does not generate a server request. This is the case, for example, of the getValue( ) method when it is used with storage attributes. Note that you can in this case use a call with the options block without this having any influence on the application performance.

Syntaxes for Callback Functions

Most client-side methods in the Wakanda framework, including "public" datastore class methods, set callback functions to execute asynchronously. Wakanda executes these functions automatically based on events (server response or error). Each function receives a single parameter, which is the event. In these functions, there are different properties that give you access to the data that the server returns. For example, **event.result** contains the function result. The result can be placed in the **event.entityCollection** or **event.entity** property, when you execute a built-in method when the type of the object returned is known.

Note that if a callback function also executes a method asynchronously or attempts to access relation attributes (requiring the sending of additional requests), you will need to nest the different callback functions (see the example for the Dataprovider API's getValue( ) method).

These callback functions can be designated in one of two ways:

- in the options parameter of the method: the functions are properties of the options object.
- directly as a parameter of the method: the functions must be passed in a specific order.

Using the "options" Object

You pass the callback function(s) in the method's options parameter. In this case, functions are named and can be passed in any order in the options object (which can contain other properties as well). Two functions are available for all methods:

- **onSuccess**: called when everything is performed correctly and receives the execution result in the event parameter. If this function is called, the **onError** function is not called.
- **onError**: called when an error (exception) occurs and receives a description of the JavaScript error in the error parameter (for more information about this parameter, refer to the Error Management section). If this function is called, the **onSuccess** function is not called.
  Note: A third function, "atTheEnd", is available for the forEach( ) method .

You can pass pass only the "onSuccess" function if you do not want to manage any errors or you can pass the same function to both the "onSuccess" and "onError" methods so that you can manage the events or errors in your code.

The call could be as follows:

```
methodName( [arguments ,] {
             onSuccess: function(event) {...},
             [onError: function(error) {...},]
             [otherOptions]
            }
        );
```

Direct Function Calls

For greater simplicity, you can pass callback functions directly as parameters to Wakanda methods. In this case, the position of the functions in the callback string designates which methods are called in the case of success or error.

The call is in the following form:

```
methodName( [arguments ,]
             function1(event) {...},
             [function2(error) {...},]
            [{otherOptions}]
        );
```

- function1 is called when everything is performed correctly and receives the execution result in the event parameter. If this function is called, function2 is not called.
- function2 is called in the event of an error and receives a description of the JavaScript error in the error parameter. If this function is called, function1 is not called.

If you pass only function1, this function will be called for all events (including errors), exceptions will be passed in the event parameter. It's up to you to handle events or errors properly in your code through this parameter. To know if an error occurred, you need to test if the event.error value is **not null**. For example, the structure of your code could be:

```
var myset = ds.Person.query("age =:1", 30,
    function(event){ // single callback function in direct call
        if( event.error != null) // if an error is returned
            var err = event.error[0].message; //handle the error
        else //otherwise a result is returned
            var vcount = event.entityCollection.length;
            ...
    }
);
```

Note: You can use a third function in the context of the forEach( ) method. In this case, it is mandatory to pass function2 even if you do not use it.

This syntax lets you write more compact code. For example, compare the following statements:

```
myCollection.getEntity(5, { onSuccess: myGotEntity, onError: myGotError } );
    // is exactly the same as:
myCollection.getEntity(5, myGotEntity, myGotError );
    // and you can also write:
myCollection.getEntity(5, myGotEntity ); //will be called for events and errors
```

You can use this syntax in all cases even when you need to pass additional parameters using the options block or want to use the userData parameter. Wakanda detects and processes the options object automatically as soon as it contains at least one of the following properties:

Properties of the *options* object

addToSet
atOnce
atTheEnd
autoExpand
callWithGet
catalog
delay
delayInfo
filterSet
first
forceReload
generateRESTRequestOnly
limit
method
onError
onSuccess
orderby
orderBy
pageSize

params
position
progressInfo
queryPath
queryPlan
queryString
skip
sync
top
userData

Note: In the case of query( ) type statements using placeholders such as :n, you must integrate the userData object in the options parameter if you use the direct syntax.

Calling Datastore Class Methods

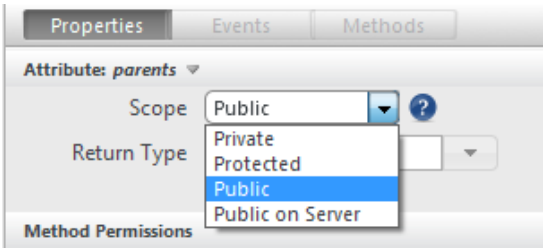
You can call any public datastore class method from the Dataprovider API. For more information about creating and specifying datastore class methods, refer to the Datastore Class Methods section in the *Wakanda Studio Reference Guide*.

Making a Datastore Class Method Available on the Client

Only **public** datastore class methods can be called from the client-side API. You define the scope of a datastore class method is an attribute on the method's Properties tab in the Datastore Model Designer (for more information, refer to Datastore Class Method Properties section in the *Wakanda Studio Reference Guide*). The following scopes are available:

- **Public**: the method can be called from anywhere, including a client-side API.
- **Private**: the method can only be used from the datastore class on the server.
- **Public on Server**: the method can be used only on the server (default).
- **Protected**: the method can be used from datastore classes as well as from derived datastore classes on the server.

By default, a datastore class method's scope is **Public on Server**. If you want a datastore class method to be available client-side, you must explicitly set the scope to **Public**:



Syntaxes for Calls

There are two ways to call datastore class methods:

- You can call the datastore class method directly on an entity, an entity collection or a datastore class, depending on the type of method.
- Using the callMethod( ) method on an entity, an entity collection or a datastore class, depending on the type of method. This method lets you write generic code since the name of the datastore class method is passed as parameter. For more information, refer to the description of the callMethod( ) method.

In both cases, calling a datastore class method triggers a request to the server. Like all methods of the WAF API, you should usually call them in asynchronous mode and handle the server answer through callback functions.

Use one of the following syntaxes:

ds.object.methodName ([options,] [params])
or

ds.object.methodName (function1 [,function2] [,params])

Basically, these syntaxes are almost similar to those of built-in WAF methods regarding callback functions (called through options object or by direct calls). For a description of these syntaxes, please refer to the Syntaxes for Callback Functions section.

params are parameters to pass to the datastore class method. You can pass one or more parameters separated by commas.

Note that unlike built-in WAF methods, you cannot pass a userData block to datastore class methods. The ability to pass parameters to the class method using the param parameter makes this principle inapplicable. You can, however, pass a userData object by including it in the options block. It is then passed as is to the callback function and you access it using the **event.userData** property.

Synchronous Calls

Wakanda lets you call public datastore class methods in **synchronous mode**, for example when the method does not return a value or when you want to debug. To do call a datastore class method in synchronous mode, omit the 'onSuccess' and 'onError' functions from the options block. Although allowed in this case, synchronous calls are usually not recommended in a Web application context.

Here is an example of a synchronous call:

```
var result = ds.Log.writeEntry(myEntry); // synchronous call of the datastore method
```

Handling Binary Return

Datastore class methods can return different types of values, that you receive client-side within the options parameter of the function call (asynchronous calls) or directly as the function result (synchronous calls). Usually, datastore class methods return standard type values such as entities, entity collections, objects, etc. These values are parsed by WAF and can be handled directly by JavaScript (see examples 1 and 2).

However, datastore class methods can also return raw binary values, such as files, text streams, pictures, etc., as described in the Returning Values section of the server-side Datastore class methods chapter. When binary values are returned, a specific processing must be set to handle the server response properly. To be able to handle returned binary data, you can use the following properties in the options parameter of the datastore class method call:

- **generateRESTRequestOnly**: *Boolean* (example: **generateRESTRequestOnly: true**)
  This option is mandatory when the server response is a binary value. When this option is passed with the **true** value, the server response will only contain the REST request necessary to access the binary data on the server. Once you get this URL, you can actually do whatever you want with the value: display it, create a download link, etc.
  By default, the option is **false**: the server response is parsed by the Dataprovider.

- **callWithGet**: *Boolean* (example: **callWithGet: true**)
  When this option is set to **true**, parameters passed to the datastore class method are sent as a '**params**' array in the request itself, rather than in the body part of the request. This option is useful when a GET HTTP request is sent to the server because, in this case, the request does not contain a body part. HTML objects such as iframes (that can be used to display binary data), execute GET requests.

For example, if you want to call a datastore class method named "getPict" which returns an image as binary data, you could write (synchronous call):

```
    //using the person datasource
var urlPict = sources.person.getPict({generateRESTRequestOnly:true, callWithGet:true}, "Smith",
"John");
    //same call using the Person datastore class
var urlPict2 = ds.Person.getPict({generateRESTRequestOnly:true, callWithGet:true}, "Smith", "John");
```

The parameters will be passed to the datastore class method properly and you will get the REST request in *urlPict*.

To see a comprehensive example of a binary return usage through an iFrame and a datasource, refer to example 3 of the Calling Datastore Class Methods Datasource section.

Example

Here is an example of calling the datastore class method client-side:

- On the server, you create the *MoreThanAverage* datastore class method associated with the "Employee" datastore class. This method is of the "Class" type ('this' in the method represents the whole datastore class, i.e., ds.Employee) and its scope is "Public". It returns an entity collection containing all employees whose salary is higher than the average salary:

```
Employee: // Datastore class method (executed on the server)
 {
     moreThanAverage:function()
     { // looks for all employees whose salary is higher than average
         var entSet = this.query("wages > :1", this.all().average("wages"));
         return entSet ; // returns resulting entity collection
     },
```

- On the client, you want to call this method asynchronously. To do this, you just execute the following code:

```
var myset = ds.Employee.moreThanAverage({onSuccess:myFunction, onError:failure,
autoExpand:"father" });
// myFunction and failure receive the resulting entity collection
```

- On the client, you can also use the callMethod( ) method:

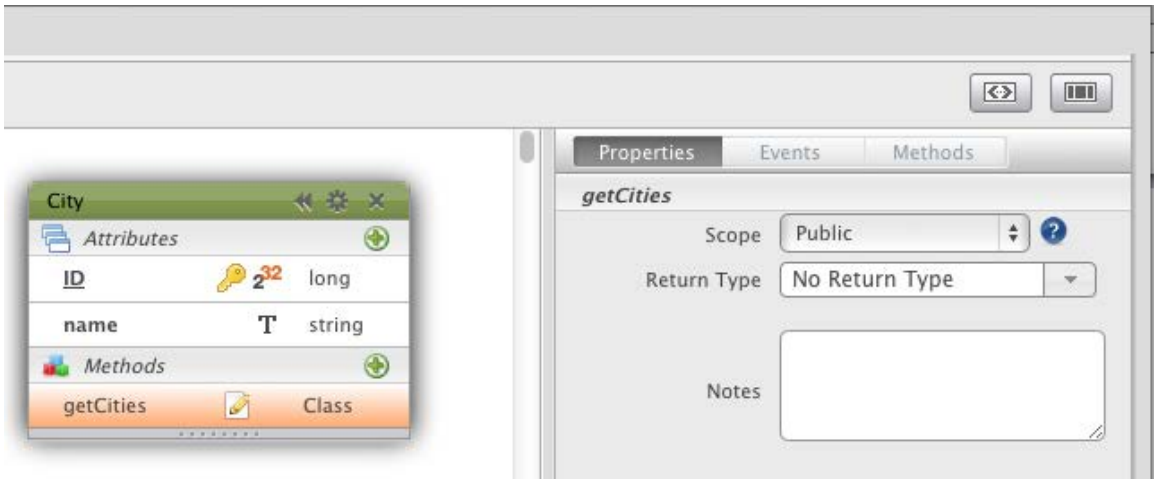
```
var myset = ds.Employee.callMethod({method:"moreThanAverage", pageSize: 60,
onSuccess:myFunction, onError:failure});
// Exactly the same as the previous call
```

- On the client, the *myFunction* method is called when the onSuccess event is generated and receives the result of the query:

```
function myFunction (event)
{
    var myset = event.result; // receives the resulting entity collection of the query
    var nbEmp = myset.length; // we retrieve the number of entities found
    $("#display").html(nbEmp); // display in a container
}
```

Example

In this example, we create a datastore class method in the City datastore class that returns an array of the city names.




```
City :
{
    methods :
    {
        getCities:function()
        {
            // Return array of city names.
            return ds.City.all().orderBy("name").toArray("name, ID");
        }
    }
}
```

We use the Dataprovider to call this method:

```
ds.City.getCities({
```

```
    onSuccess: function(event) {
        var myHTML = '';
        var myArray = event.result;
        myArray.forEach(function(elem) {
            myHTML += '<p>' + escapeHTML(elem.name) + '</p>';
        });
        $('#container4').html(myHTML);
    }
});
```

Error Management

On the client, any errors generated during execution can be retrieved in the callback function(s) specified during asynchronous calls (**onError** function or single callback function, see Syntaxes for Callback Functions section). All kinds of errors are returned, whatever their source - datastore validation (user-defined errors) or a Wakanda server internal process.

Note: You can also use automatic error management tools available in the GUI Designer. For more information, please refer to the Error Handling section in the "GUI Designer" chapter of the Wakanda Studio Reference Guide.

When an error occurs, it is returned by the server as an **error** property of the parameter (usually named event) of the callback function. The **event.error** property is an array that consists of a stack of error objects whose element 0 contains the highest-level error.

Each element in the **error** array has the following properties:

Property	Type	Description
error[n].errCode	number	error code
error[n].message	string	message of the error
error[n].componentSignature	string	signature of the component at the origin of the error "dbmg" = Wakanda database engine errors "dbev" = User-defined validation errors

Component signatures give additional information on the origin of the errors:

- *dbmg* = Wakanda database engine errors signature.
  - errCode 0 to 1499 = errors related to the database
  - errCode 1500 to 1799 = errors related to datastore classes
  - errCode 1800 and 2099 = errors related to REST requests on datastore classes
- *dbev* = User-defined validation errors signature.
  - customized errCode range

Example

In your model, if you state that the 'salesVolume' attribute cannot be equal to 0, then in the **onValidate** event for the datastore class, you can write:

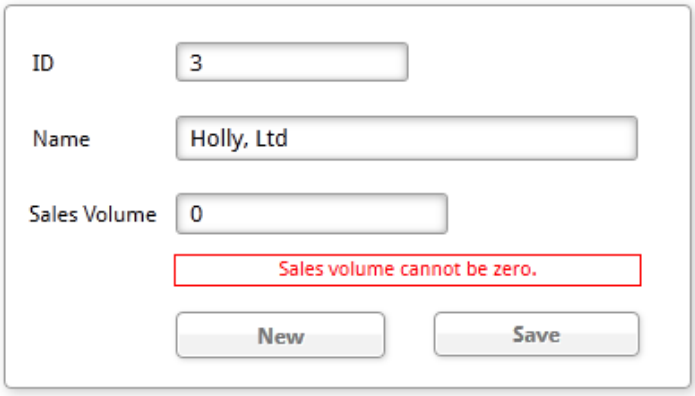
```
Company:
{
    events :
    {
        onValidate:function()
        {
            if(this.salesVolume == 0){
                return {error: 100, errorMessage: 'Sales volume cannot be zero.'}
            }
        }
    }, ...
}
```

Your Page contains Text Input widgets for attributes. A Display Error widget has been added to the page (selected in the picture below); its ID is used as the **Display Error** property of the salesVolume attribute.

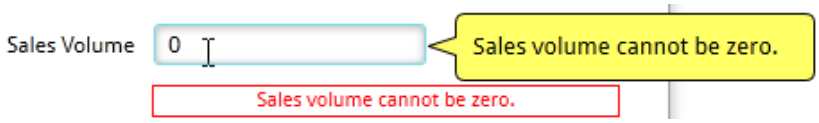
You associate the following code with the **Save** button:

```
saveButton.click = function saveButton_click (event)
{
     sources.company.save({
        onSuccess: function (event){
                //handle successful save
            sources.company.addEntity(sources.company.getCurrentElement());
        },
        onError: function(event){
                // an error occurred
                // display the top-level error message in the Display Error widget
            $$('salesVolume').setErrorMessage({message: event.error[0].message, tooltip: true});
        }
    });
};
```

When executing the application, if the user enters '0' for the "Sales Volume" attribute and clicks the **Save** button, the error is displayed in the area:



The error can also be displayed as a tip (provided that the *tooltip* option is set to **true** in the [#cmd id="700200"/] method call):



Behind the scenes, the returned **event.error** object is an array with two elements:

Elements	Contents
event.error[0]	{componentSignature: "dbev", errCode: 100, message: "Sales volume cannot be zero."}
event.error[1]	{componentSignature: "dbmg", errCode: 1517, message: "The entity #4 of the datastore class "Company" cannot be saved."}

Defining Queries (Client-side)

Building a Query

Querying data is the most common operation in a datastore. You will always need to search, filter, and order your data using different criteria.

Several client-side JavaScript methods are designed to execute query strings on your data:

- Datasource API: query( ) and filterQuery( )
- Dataprovider API : query( ) and find( ) for datastore classes, and query( ) for entity collections.

Note: Server-side JavaScript methods are also available for querying the server: find( ) and query( ) on a datastore class, and find( ) and query( ) on an entity collection. Defining the queryString parameter is a little bit different from the client-side. For more information, please refer to the section Defining Queries (Server-side).

The query parameter is named queryString. This parameter always uses the following syntax:

```
attribute comparator value {conjunction attribute comparator value...{ order by attribute }}
```

- *attribute*: The datastore class attribute on which you want to execute the query. For example, "employee.name". This parameter can also be any valid attribute path such as "father.father.name".
- *comparator*: The comparison made between *attribute* and *value*. The comparator is one of the symbols or keywords listed in the .
- *value*: The value to compare to the current value of the attribute of each entity from the entity collection. It can be any expression that evaluates to the same data type as the attribute. The value is evaluated once, at the beginning of the query. It is not evaluated for each entity.
  To query a string contained in a string (a “contains” query), use the wildcard symbol (*) in value to isolate the string to be searched for as shown in this example "*Smith*". Note that in this case, the search only partially benefits from the index.
  *Compatibility note: In Wakanda v1 and in Dev Branch versions prior to build 108437, the @ operator was used as the wildcard symbol instead of the *.*

  You can compare the NULL value in a query by using the "null" keyword.
- *conjunction*: The conjunction operator is used to join multiple conditions into the query (optional). You can use one of the following logical operators (pass the name or the symbol):

Conjunction name	Conjunction symbol	Comments
AND	&	&& can be used
OR	|	|| can be used
NOT	!	
EXCEPT	^	equivalent to AND NOT

- *order by attribute*: You can include an order by statement in the query; the resulting data will be sorted according to the statement. Pass '**desc**' to define a descending order and '**asc**' to define an ascending order. By default, the order is ascending.

Implementation Note: In the current implementation of Wakanda, queries are NOT case sensitive.

Here are some examples of valid queries:

```
'employee.name = "smith" AND employee.firstname = "john"'
```

Note: Double quotes " " or quotes ' ' can be omitted for string values if there is no ambiguity.

```
'employee.city = Chicago && employee.salary < "10000" order by salary asc'
```

You can use parentheses in the query to give priority to the computation. For example, you can organize a query as follows:

```
'(employee.age >= "30" OR employee.age <= "65") AND (employee.salary <= "10000" OR employee.status = "Manager")'
```

When comparing dates, you should use date values in the following format: YYYY-MM-DDTHH:MM:SSZ (e.g., "2010-10-05T23:00:00Z" for October 5, 2010). The time is included in the date format and is based on GMT +1 where midnight is 23:00 (11pm).

```
'employee.dateHired > 2011-12-14T23:00:00Z' //To search for all employees hired after December 14, 2011'
```

Note: The wildcard (i.e., "") is not supported in date query string values.*

Query on 'Date only' attributes

If the "Date only" option is checked for a date attribute in the Model designer, you can simply compare dates

on the 'date' portion of standard JavaScript dates, without regard to the Timezone. For example:

```
'employee.dateHired > 2012-12-24' //To search for all employees hired after December 24, 2012, whatever the time zone
```

For more information on the "Date only" option, please refer to the Attributes section.

Using the '==' operator

Starting with Wakanda 4, we recommend that you use the standard JavaScript equality operator (==) in Wakanda queries, instead of the single (=) operator. The single (=) operator could be confused with the standard JavaScript assignment operator.

For example, the following query string:

```
'employee.name = "smith" AND employee.firstname = "john"' //until Wakanda 4
```

should now be written:

```
'employee.name == "smith" AND employee.firstname == "john"' //starting with Wakanda 4
```

Documentation and examples will be progressively updated to use this new syntax. Of course, the single (=) operator is still supported for compatibility.

Using :n placeholders for values (parameterized queries)

You can use special placeholders in your queries, so that you do not have to worry about formatting issues, in particular when the values to compare contain or may contain special characters such as slashes (/) or single/double quotes (", '). In addition, in some cases, using placeholders is mandatory (see below). When using a placeholder, you just pass values as parameters in the query and Wakanda will manage all value formatting issues.

When working client-side, queries are to be asynchronous. Thus, the actual values to compare should be passed inside the options parameter, in an array named "params". In this context, **:n** means: "use the n^{th} value of the **params** array as the value to compare".

For example :

```
ds.Employee.find("name = :1 and age > :2", // asynchronous call
    { onSuccess: function(event)
        {var theEmp=event.entity; // the result is an entity
            // it would have been in event.entityCollection for the query() method
            // here you can handle attribute values using getValue() and setValue()
        var theJob = theEmp.jobName.getValue(); // direct access for storage attributes
    }, params:["Jones",30]
});
```

- "Jones" is the first parameter: it will be used as the :1 value
- 30 is the second parameter: it will be used as the :2 value

It could be easily nested in a function as well:

```
function findEmployee(theName, theAge)
    {
        var Result = ds.Employee.find("name = :1 and age > :2",
        { onSuccess: function(event)
             {
                var theEmp= event.entity;
                return theEmp;
            }, params:[theName,theAge]
        }
    });
}
```

You can then invoke it:

```
var myEmp = findEmployee("Jones", 30);
```

When evaluating the query, Wakanda will use the current value of both theName and theAge parameters to compare to, respectively, "name" and "age" attributes. If an entity contains a name with special characters such as quotes, it will not be an issue and the query will be evaluated correctly.

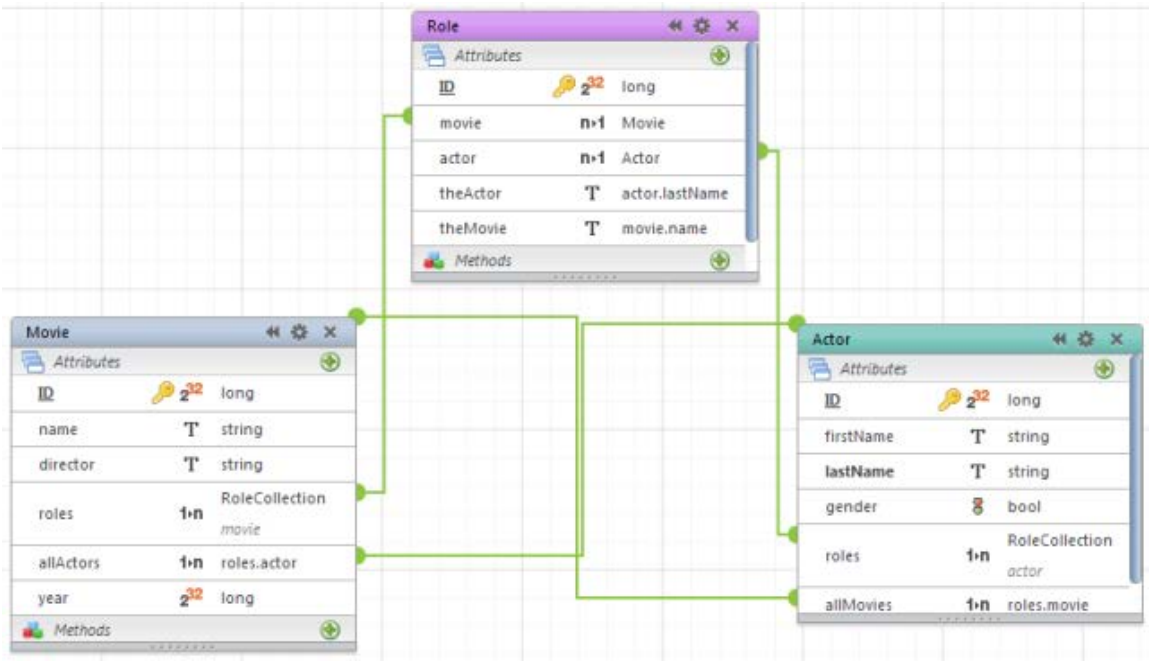
Mandatory Placeholders

Placeholders are mandatory when you use complex objects such as arrays in a query, as shown in the following example:

```
var coll = ds.Customer.query( "country in ['US','SP','GM']"); // NOT supported
var coll = ds.Customer.query( "country in :1", ['US','SP','GM']); //supported
```

Queries in N <-> N Relations

Wakanda offers a special syntax to facilitate queries in the context of N <-> N relations. In this context, you may need to search for different values with an AND operator but in the same attribute. For example, take a look at the following structure:



Imagine that you want to search all movies in which *both* actor A and actor B have a role. If you write a standard query using an AND operator, it will not work:

```
// invalid code
ds.Movie.query("'allActors.lastName' = :1 AND 'allActors.lastName' = :2", "Hanks", "Ryan");
```

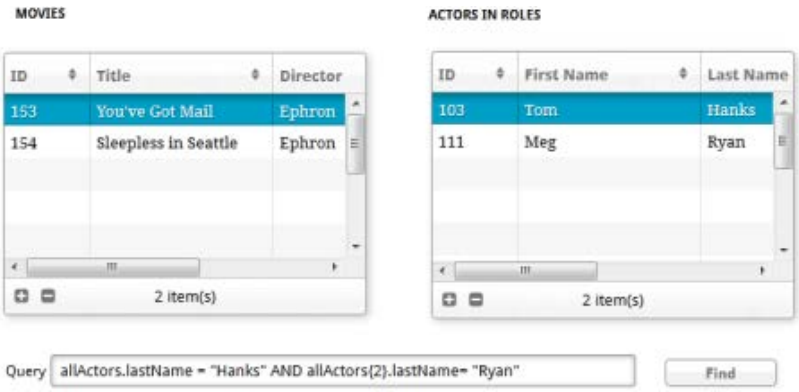
Basically, the issue is related to the internal logic of the query: you cannot search for an attribute whose value would be both "A" *and* "B".

To make it possible to perform such queries, Wakanda allows a special syntax: you just need to add a *class index* in all additional relation attributes used in the string:

"'relationAttribute.attribute' = :1 AND 'relationAttribute{n}.attribute [AND 'relationAttribute{m}.attribute...]'"

In our example, it would be:

```
// valid code
ds.Movie.query("'allActors.lastName' = :1 AND 'allActors{2}.lastName' = :2", "Hanks", "Ryan");
```



{n} tells WakandaDB to create another reference for the *allActors* relation attribute. It will then perform all the necessary bitmap operations internally. Note that *n* can be any number except 0: {1} or {2} or {1540}... Wakanda only needs a unique reference in the query for each class index.

For example, this code builds the query string based on an array of actor names:

```
function queryMoviesWithSeveralActors(actorsArray) {
    var coll, count, queryStr;

    if(actorsArray && actorsArray.length) {
        count = 0;
        queryStr = '';
        actorsArray.forEach(function(actor) {
            count += 1;
            // First actor
            if(count === 1) {
                queryStr = 'allActors.name = "' + actor + '"';
            } else {
                queryStr += ' AND allActors{' + count + '}.name = "' + actor + '"';
            }
        });

        coll = ds.Movie.query(queryStr);
    } else {
        coll = ds.Movie.createEntityCollection();
    }

    return coll;
}
```

Calling the function with:

```
    queryMoviesWithSeveralActors(['Hanks', 'DiCaprio', 'Garner']);
```

generates the following query string:

```
    'allActors.name = "Hanks" AND allActors{2}.name = "DiCaprio" AND allActors{3}.name = "Garner"'
```

... and finds any movie(s) containing all 3 of these actors (*Catch Me if You Can*).

Comparator List

The comparator is one of the symbols shown below. For some comparators, you can use a symbol, one of the alternate symbols, or keywords listed in the "Alt. keywords" column:

Comparison	Symbol	Alt. keywords	Comments
Like	==	eq, like	Gets matching data, supports the wildcard (*), neither case-sensitive nor diacritic. See examples.
Equal to	===	is, eqeq	Gets matching data, considers the wildcard (*) as a standard character, neither case-sensitive nor diacritic.
Is in array	in		Gets data equal to at least one of the values in an array
Not like	#	!=	
Not equal to	!==	nene, isnot, ##	
Greater than	>	gt	Strictly greater than
Greater than or equal to	>=	gteq, gte	
Less than	<	lt	Strictly less than

Less than or equal to	<=	lteq, lte	
Begins with	begin		"begin t" is equivalent to "like t*"
Contains keyword	%%		Keywords can be used in attributes of text or picture type
Matches	matches	=%, %*	Uses JavaScript Regex
Does not match	!=%	!%*	Uses JavaScript Regex

Note: To keep compatibility with Wakanda 3 applications, the (=) operator is still supported for 'like' queries.

Using JavaScript functions

Unlike server-side queries (see Defining Queries (Server-side)), for security reasons you are not allowed to call JavaScript functions directly in your client-side queries.

However, you can write datastore class methods allowing you to take advantage of JavaScript function-based queries from clients. This way, you can execute JavaScript functions and keep control of the code executed on the server.

For example, you want to know if a given string is present in the "comments" attribute of your Book datastore class collection. You want to use the indexOf( ) JavaScript function.

You can write the following datastore class method (applied to collection):

```
model.Book.methods.mySearch = function(stringToSearch)
{
   var myCol = this.query("$(stringToSearch.indexOf(this.comments) != -1)", // this refers to the
collection
         { allowJavascript: true });   // to allow javascript execution
   return myCol;
}
model.MyClass.methods.mySearch.scope = "public"; // do not forget to make the method public
```

Then, on the client side, you can call for example:

```
sources.book.mySearch("abracadabra");
```

Examples

Please refer to examples in the Defining Queries (Server-side) section.

DataClass

The methods of this class apply to the datastore classes of the current datastore.

Working with Datastore Classes on the Client

Accessing Datastore Class Methods

For objects of type DatastoreClass, you can access datastore class methods of type "class" (applied to the datastore class), specified in the Datastore Model Designer provided their scope is "Public". You cannot call datastore class methods of type "entity" or "entityCollection" on these objects.

all( )

void **all**([Object *options*])
ParameterType Description
options Object Block of options for asynchronous execution

Description

The **all( )** method is an alias to the allEntities( ) method. For more information, please refer to the allEntities( ) method description.

allEntities( )

void **allEntities**([Object *options*])
ParameterType Description
options ObjectBlock of options for asynchronous execution

Description

Note: You can also call this method's alias all( ).

The **allEntities( )** method returns an entity collection containing all the entities in the datastore class. This method is the same as performing a query( ) when the queryString parameter contains an empty string.

Since only the first 40 entities (by default) are transmitted when an entity collection is established, there is no negative performance impact to specifying **allEntities( )** on a datastore class with millions of entities.

As this method is called asynchronously, you will retrieve the resulting entity collection in the callback function specified in the options parameter through the **event.entityCollection** property. By default, the entities are returned in the order in which they were created.

Note: Keep in mind that if a restricting query has been defined on the server, this method will work only with available entities, that is, entities resulting from the restricting query. For more information, please refer to the Programming Restricting Queries section

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **allEntities( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **allEntities( )** method:

- **orderBy**: *a string containing one or several attribute name(s) delimited by commas. To define the sort order, include either 'asc' or 'desc' after the attribute's name*. For example, **orderBy: "name, firstname desc"**.
  Allows you to sort the resulting entity collection. The order in which the attributes are passed determines the sorting priority of the entities. By default, attributes are sorted in ascending order unless otherwise specified.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

This example replaces the datasource's current entity collection with all the entities sorted by name and firstName:

```
ds.Person.allEntities({onSuccess: function(event)
    {
        // use the new set to replace the entity collection of a datasource
```

```
        sources.person.setEntityCollection(event.entityCollection);
    },
    // ask for sorting the resulting collection (from v2 only)
    orderBy: "name, firstName"
});  //async all elements
```

callMethod( )

Mixed **callMethod**(Object *options* [, String *params*])
ParameterType Description
options ObjectBlock of options for asynchronous execution
params String Parameter(s) to pass to the datastore class method

Returns Mixed Value returned by the method in synchronous mode

Description

The **callMethod( )** method executes a datastore class method on the entity, entity collection, or datastore class to which it is applied.

When you call this method in asynchronous mode, the result, if any, is retrieved through the **event.result** property in the callback function defined in the options parameter.

Note: You can call a datastore class method directly as the property of an entity, entity collection, or datastore class (see Calling Datastore Class Methods). The main advantage of using the callMethod( ) method is that it is useful when writing generic code because the method name is passed as a string.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the *options* parameter, you pass an object containing the "onSuccess" and "onError" callback functions as well as additional properties (that depend on the method called). Each callback function receives a single parameter, which is the event.

You can also pass directly the onSuccess and onError functions as parameters to the **callMethod( )** method. In this case, they must actually be passed just before (and outside) the object parameter.

With the **callMethod( )** method, you must pass the name of the method as a member of the options object: {method:"MethodName"}. *MethodName* must be passed as a string. For example:

```
method:"moreThanAverage"
```

The following attributes are also available in the options block for the **callMethod( )** method:

- **pageSize**: *number* (e.g., **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. The default value is 40. If the entity collection contains 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a Grid or Matrix widget.
  You can vary the value sent to this parameter for optimization reasons, e.g., according to the height of the widget.

- **autoExpand**: *string containing one or more relation attributes* (e.g., **autoExpand: "worksFor, fatherOf"**)
  By default, the values of relation attributes are not expanded in the entity collections returned by the server for optimization reasons. They are only expanded on request, i.e., when a user or a function accesses them (the Dataprovider then sends the corresponding requests automatically). You may want to preload these values in order to display them or to access them in the case of nested asynchronous requests.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to later retrieve. You simply pass the data to this property and retrieve it from inside the callback function in the

event.userData object.
The **userData** property can contain any JavaScript valid value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters...) that you want to use again in the callback function.

- **params**: *arguments array* (e.g., **'params' : [100, 200,300]**)
  Array of parameters to pass to the datastore class method. These parameters are used in the order they are defined for the array.
  With this attribute, you pass all the parameters as strings in the params parameter. If you pass the parameters directly to the datastore class method, the params parameter is not needed.

params

You use the params parameter to pass one or more parameters to the datastore class method (except when you are using the params attribute of the options block). You must pass parameters as strings and delimit each one by commas.

Example

This example calls the *teaching* datastore class method, applied to a "class", and passes it a subject name as parameter:

```
ds.Teachers.callMethod({method:"teaching", onSuccess:gotTeachers, onError:failure},"Math");
```

Example

Below we execute a method with the **callMethod( )** function:

```
//execute a method with the callMethod function
ds.Person.callMethod({method:"sendWelcomeEmail",
    onSuccess:function(event){
            //handle success
    }, onError:function(error){
            //handle error
}});
```

Then, we execute the same method directly as a standard method:

```
ds.Person.sendWelcomeEmail({onSuccess: function(event) {
        //handle success
    }, onError: function(error) {
        //handle error
    }}
);
```

clearCache( )

void **clearCache**( )

Description

The **clearCache( )** method clears the entity cache on the client for the specified datastore class. Once this method is executed, there are no more entities in the cache managed by the Dataprovider on the client machine.

This method is useful for ensuring that the Dataprovider reloads the entities from the server. You can use it when you work with an entity collection and want to get the latest version of one or more entities that it contains without having to execute the initial request again.

The cache is only used to optimize access to the data retrieved since the last query. Any new request retrieves the entities on the server and updates the cache with the result.

distinctValues( )

void **distinctValues**(DatastoreClassAttribute | String *attribute* [, Object *options*])
ParameterType Description

attribute	DatastoreClassAttribute, String	Attribute for which you want to get the list of distinct values
options	Object	Block of options for asynchronous execution

Description

The **distinctValues( )** method retrieves an array containing all the distinct values stored in attribute for the entity collection or datastore class. By default, this command takes into account all the entities in the datastore class or entity collection to calculate the distinct values; however, you have the option of filtering them through the attributes of the options block so as to limit the number of entities.

Since this method is called asynchronously, you must retrieve the resulting array in the callback function specified in the options parameter through the **event.distinctValues** property.

Note: You can also use the event.result property.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **distinctValues( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following attributes are also available in the options block for the **distinctValues( )** method:

- **skip**: *numeric value* (example: **skip: 20**)
  This method starts the array of distinct values at the *X* value defined by *skip*.

- **top**: *numeric value* (example: **top: 40**)
  This method returns a set of *X* elements in the array of distinct values, starting from the first one or from the one defined by *skip*.

  The above parameters are useful when you want to paginate the results.

- **progressBar**: *string indicating a progress bar ID* (example: **progressBar: "myProgressBarID"**)
  The ID referencing an existing Progress Bar widget that the server will use to indicate the method's state of progress.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

In the following example, we create an HTML select list and load into it the distinct values of the *country* attribute in the "Employee" datastore class:

```
button2.click = function (event)
    {
        ds.Employee.distinctValues("country", { onSuccess: function(event)
        {
            var myArray = event.distinctValues; // receives the array of distinct values
                    // you can also use event.result
            var html = "";
            html += "<select>"; // pop up menu type object
            for (var i = 0; i < myArray.length; i++)  // building of select list with array values
            {
                var val = myArray[i];
                html += '<option value="'+val+'">'+val+'</option>';
```

```
            }
            html += "</select>";
            $("#popup").html(html); // assignment to pop up
        } });
    };
```

find( )

void **find**(String *queryString* [, Object *options*])
Parameter Type Description
queryString String Search criteria
options ObjectBlock of options for asynchronous execution

Description

The **find( )** method applies the search criteria specified in the queryString to all of the entities of the DatastoreClass to which it is applied and returns the first entity found in an object of the Entity type. You can then, for example, read the values of this object, pass it as a parameter, etc.

Since this method must be called asynchronously, the entity is retrieved through the **event.entity** property in the callback function specified in the options parameter.

Pass a valid search string in queryString. For a detailed description of this parameter, refer to Defining Queries (Client-side). You can use parameterized queries using placeholders of the type :n; in this case, the values of the parameters must be passed in options through the **params** array (see below).

Executing the **find( )** method amounts to executing a query( ) followed by the retrieval of the first entity. However, there is a significant difference in how they work when the search is not successful that must be considered:

- In the case of **find( )**, the method simply returns Null because the search did not find anything,
- In the case of code related to query( ), an error is returned because we are trying to access an array element that does not exist. In this case, you should test to make sure that the entity collection is not empty before accessing the element [0].

Note: Keep in mind that if a restricting query has been defined on the server, this method will work only with available entities, that is, entities resulting from the restricting query. For more information, please refer to the Programming Restricting Queries section

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **find( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **find( )** method:

- **params**: *arguments array* (example: **'params' : [100, 200,300]**)
  Array of arguments to use in the queryString of a "parameterized query". These arguments are used in order for placeholders (see Using :n placeholders for values (parameterized queries)).

- **autoExpand**: *string containing one or more relation attributes* (example: **autoExpand: "worksFor, livesIn"**)
  By default, the values of relation attributes are not calculated in the entity returned by the server, for optimization reasons. Access to these values automatically triggers the corresponding requests. You may want to precalculate these values, for example to be able to display them. To do this, you can just pass the attribute(s) to calculate in the autoExpand parameter.

- **progressBar**: *string indicating a progress bar ID* (example: **progressBar: "myBar"**)
  ID referencing an existing widget of the Progress bar type that the server will use to indicate the

method's state of progress.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

In this example, we want to perform a query in a datastore class and display the information relating to the first entity found in a container:

```
ds.Person.find("lastname = :1 and ID > :2", {
    params: ['A'+WAF.wildchar, 300],      // search for a person whose name begins with A and whose
ID > 300
        // WAF.wildchar contains '*'
    onSuccess: function(event)     // callback for asynchronous execution
    {
        var myEntity = event.entity; // retrieve the entity directly
        var html = ""; // build the contents of the container
        html += "ID : "+myEntity.ID.getValue() + "<br/>";   // access the entity attributes
        html += "lastname : "+myEntity.lastname.getValue() + "<br/>";   // with the getValue( )
method
        html += "firstname : "+myEntity.firstname.getValue() + "<br/>";
        html += "wages : "+myEntity.wages.getValue() + "<br/>";

        $("#display").html(html); // display in the container whose ID is "display" (jQuery
notation)
    }
});
```

getAttributeByName( )

DatastoreClassAttribute **getAttributeByName**(String *attributeName*)

Parameter	Type	Description
attributeName	String	Name of attribute to retrieve

Returns	DatastoreClassAttribute	Attribute of the datastore class

Description

The **getAttributeByName( )** method returns an object containing the datastore attribute whose name is passed in attributeName. It is mainly useful when writing generic code.

Note: A datastore class attribute can also be accessed directly as a property of the datastore class. For more information, refer to the Accessing Datastore Class Attributes paragraph.

The DatastoreClassAttribute object returned contains numerous properties describing the attribute as it is specified in the datastore model. The number of properties varies according to the attribute type.

These properties can be useful for generic programming. The available properties are:

- *kind*: kind of attribute
  Possible values:
  - "storage": storage (or scalar) attribute
  - "calculated": calculated attribute
  - "relatedEntity": relation attribute
  - "alias": alias attribute

  Note that the *kind* property is not determinative client-side because calling and using these attributes are identical and do not depend on the property.
- *name*: name of the attribute
- *type*: type of the attribute (native type for storage attributes, datastore class name for relation attributes)
- *owner*: datastore class to which the attribute belongs
- *identifying*: true or false
- *indexed*: true or false

- *readOnly*: true or false (for example in the case of a "calculated" attribute with *get* but not *set*).
- *related* : true or false
- *relatedClass*: related datastore class (for relation attributes). By default, for optimization reasons, this attribute is not calculated. You must explicitly call the getRelatedClass( ) method to be able to retrieve this information.
- *relatedOne*: true or false
- *resolved*: true or false
- *simple*: true or false
- as well as any *maxValue*, *minValue*, *formats*, etc. properties specified in the Datastore Model Designer.

Example

In this example, we obtain the 'kind' property for an attribute:

```
var myAttribute = ds.Employee.getAttributeByName( "lastName");
var theKind = myAttribute.kind;
```

getAttributes( )

Object **getAttributes**( )

Returns ObjectList of datastore class attributes

Description

The **getAttributes( )** method returns an object containing the list of all the attributes in the datastore class. For each attribute, this method returns the name and value of its properties such as, for example, "autocomplete".

This method is useful for generic programming. It returns only the attributes in the datastore class (and not the associated methods and functions), which makes it easier to work with them.

Example

Here is a simple example that outputs a list of attribute names to a :

```
var html = "";
 var allAttributeNames = ds.Person.getAttributes();
 for (var i in allAttributeNames) {
     var attr = ds.Person.getAttribute(allAttributeNames[i]);
     html += attr.name + "<br/>";
 }
 $("#attributeNames").html(html);
```

getCacheSize( )

Number **getCacheSize**( )

Returns NumberNumber of entities to keep in cache

Description

The **getCacheSize( )** method returns the current size of the entity cache on the client for the specified datastore class.

The entity cache is the number of entities kept on the client and managed by the Dataprovider in order to optimize access to the data. Cache management is transparent for the client.

By default, the size is 300 entities. You can increase this value according to your application's requirements by using the setCacheSize( ) method.

Example

You want to find out the size of the entity cache for the Cities entity model:

```
var cacheCities = ds.Cities.getCacheSize();
```

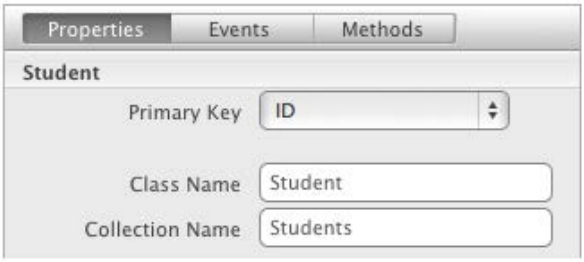
getCollectionName( )

String **getCollectionName**( )

Returns StringName of an entity collection of the datastore class

Description

The **getCollectionName( )** method returns the name of an entity collection in the datastore class used to designate collections of entities in the code more naturally. This name is specified in the Properties of each datastore class:



You can use the plural form of name of the datastore class, or use the suffix _collection, or any other name reflecting the singularity of the entities (for example, People / Person).

In your code, you can access this information as follows:

```
var entityCollName = ds.Student.getCollectionName(); //entityCollName is "Students"
```

getDataStore( )

Datastore **getDataStore**( )

Returns DatastoreDatastore of class

Description

The **getDataStore( )** method returns the datastore for the specified datastore class.

By default, the current datastore is placed in the **ds** object (see Accessing a Datastore). However, you can open and use different datastores within your application as described in Accessing datastore classes of the Datastore API (server API). In this case, you can use this method to find out the datastore for a class.

Example

We want to retrieve a list of all the datastore classes for the datastore of a given datastore class:

```
var listClasses = ds.Employee.getDataStore().getClasses();
// returns all the datastore classes of the datastore
```

getEntity( )

Entity **getEntity**(Number | String *keyValue* [, Object *options*])

Parameter	Type	Description
keyValue	Number, String	Value of primary key of entity to be retrieved
options	Object	Block of options for asynchronous execution

Returns Entity Entity returned in case of synchronous execution

Description

The **getEntity( )** method retrieves the entity whose primary key value is passed in keyValue in the specified datastore class.

Warning: This method expects a different parameter (position) when it is applied to an entity collection (see getEntity( )).

This method is called asynchronously so the entity is retrieved through the **event.entity** property in the callback function specified in the options parameter.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions section.

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **getEntity( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following attributes are also available in the options block for the **getEntity( )** method:

- **forceReload**: *boolean* (example: **forceReload: true**)
  When this option is set to **true**, the latest version of the entity is always reloaded from the server. In this way, you can get the internal stamp values if they were modified on the server in the meantime. If this option is set to **false** or omitted, the **getEntity( )** method may get the entity from the local cache.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

In this simplified example, we retrieve an entity by its primary key value and display it in a Container widget whose ID is "display":

```
button1.click = function (event)
    {
          ds.Employee.getEntity(100,{onSuccess:function(event) // get the employee whose ID
(primary key) is 100
          {   myEnt=event.entity; // get the entity
              $("#display").html("result = "+myEnt.lastName.getValue());
          }});
    };
```

getName( )

String **getName**( )

Returns StringName of the datastore class

Description

The **getName( )** method returns the name of the datastore class to which it is applied. This method's main purpose is for writing generic code. For example, when you want to pass the name of the datastore class as a parameter, you can use this method to retrieve its name.

Example

To retrieve the name of a datastore class from a collection:

```
var myset = ds.Employee.query("salary > 10000");
var myClass = myset.getDataClass().getName(); // returns the name of the datastore class
```

newCollection( )

EntityCollection **newCollection**([Object *colRef*] [,Object *options*])

Parameter	Type	Description
colRef	Object	Reference to an entity collection on the server
options	Object	Block of options for asynchronous execution

Returns EntityCollection New blank entity collection (synchronous call)

Description

The **newCollection( )** method creates a new blank object of type EntityCollection attached to the datastore class. When it is created, the entity collection does not contain any entities.

- If you call this method without any parameters, the collection is created locally and is returned directly by the method. No request is sent to the server. Server requests will be sent only if necessary, for example, if you sort the collection.
- If you pass the colRef parameter, the method will be executed asynchronously and you will get the new collection through the **event.entityCollection** object in the options parameter. To colRef, pass an entity collection server reference, which was returned by the getReference( ) method.

This method lets you build entity collections gradually by making subsequent calls to the add( ) method. You can also pass the collection as the new current collection of a server datasource using the setEntityCollection( ) method.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **newCollection( )** method. In this case, they must be passed just before (and outside) the options parameter.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

newEntity( )

Entity **newEntity**( )

Returns EntityNew entity created in memory

Description

The **newEntity( )** creates a new entity in the datastore class and returns an empty Entity object. By default, the *null* value is assigned to each attribute of the new entity.

The entity is not saved in the datastore until you call the save( ) method.

Note: Using this method is similar to use the new operator with the WAF.Entity function (for more information, see section Working with Entities on the Client).

Example

In an application that contains an Employee datastore class and a Company datastore class linked by a relation attribute, we want to create an Employee entity and assign a Company to it by clicking a button. In our interface, the Company entity is already selected:

```
button5.click = function (event)
{
    var comp = sources.company.getCurrentElement(); // retrieve the current entity of the
datasource
    var emp = ds.Employee.newEntity(); // create the entity
    emp.lastName.setValue("Miller"); // assigning storage attributes
    emp.firstName.setValue("Anne");
    emp.salary.setValue(45000);
    emp.employer.setValue(comp); // pass the relation entity as an attribute value
    emp.save({
        onSuccess:function(event)
        {
            $("#display").html("saved ok"); // display the result in a Container widget
        },
            onError:function(event)
        {
            $("#display").html("error on save");
        }
    });
};
```

query( )

EntityCollection **query**(String *queryString* [, Object *options*])

Parameter	Type	Description
queryString	String	Search criteria
options	Object	Block of options for asynchronous execution
Returns	EntityCollection	New entity collection made up of entities meeting search criteria specified in the queryString in the case of synchronous execution

Description

The **query( )** method searches for entities meeting the search criteria specified in queryString among all the entities in datastore class or entity collection, and returns a new object of type EntityCollection containing all the entities found.

Using several consecutive **query( )** methods on the entity collections lets you perform searches by successively reducing the scope of the search. If you keep the intermediary entity collections, you can also provide a rollback system without regenerating server requests

This method is called asynchronously, so you must retrieve the resulting entity collection for the search in the callback function set in the options parameter, usually using the **event.entityCollection** property. However, note that in the context of a **query( )** method executed on the client, Wakanda lets you use the entity collection returned by the method even if at first it is in a "non-finalized" state. The reference to this entity collection is valid so you can work with it and use it once the callback function has been successfully called.

Pass a valid search string in queryString. For a detailed description of this parameter, refer to Defining Queries (Client-side). You can use parameterized queries using placeholders of type **:n**. In this case, the parameter values must be passed in options through the **params** array (see below).

Note: Keep in mind that if a restricting query has been defined on the server, this method will work only with available entities, that is, entities resulting from the restricting query. For more information, please refer to the Programming Restricting Queries section

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions section.

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **query( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **query( )** method:

- **params**: *arguments array* (example: **'params': [100, 200,300]**)
  Array of arguments to use in the queryString of a "parameterized query". These arguments are used in order for placeholders (see Using :n placeholders for values (parameterized queries)).

- **pageSize**: *number* (example: **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. By default, the value is 40: if the search "finds" 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a list.
  You can have this parameter vary for optimization issues, according, for example, to the size of the widgets.

- **autoExpand**: *string containing one or more relation attributes* (example: **autoExpand: "employer, livesIn"**)
  By default, the values of relation attributes are not calculated in the entity collections returned by the server, for optimization reasons. Only access to these values automatically triggers the corresponding requests. However, you may want to precalculate these values, for example to be able to display them or to cut down on requests to the server. The **autoExpand** option, once applied during the creation of an entity collection, remains valid during the entire life duration of this entity collection, i.e., all access to the data of this entity collection executes an **autoExpand** whenever it is needed. For example, if the entity collection is large, its contents are retrieved by pages of 40 entities each on the client: every time a new page of entities is accessed, an autoExpand is performed.

- **progressBar**:*string indicating a progress bar ID* (example: **progressBar: "myBar"**)
  ID referencing an existing widget of the Progress bar type that the server will use to indicate the method's state of progress. For more information, refer to the ProgressIndicator server side class description.

- **queryPlan**: *Boolean* (example: **queryPlan: True**)
  In the entity collection, returns or does not return the detailed description of the query just before it is executed, i.e. the planned query. The information recorded includes the type of query (indexed, sequential), the number of entities processed and the breakdown in the case of a complex query. This option is useful during the development phase of the application. It is usually used in conjunction with **queryPath**.

- **queryPath**: *Boolean* (example: **queryPath: False**)
  In the entity collection, returns or does not return the detailed description of the query as it is actually performed. This information is usually identical to that of the **queryPlan**, but may differ if the engine manages to optimize the query. This option is useful during the development phase of the application. For more information about these options, refer to queryPlan and queryPath.

- **orderBy**: *a string containing one or several attribute name(s) delimited by commas. To define the sort order, include either 'asc' or 'desc' after the attribute's name*. For example, **orderBy: "name, firstname desc"**.
  Allows you to sort the resulting entity collection. The order in which the attributes are passed determines the sorting priority of the entities. By default, attributes are sorted in ascending order unless otherwise specified.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

Here is the code for a button that executes a simple search in the "Person" datastore class and returns the size of the resulting entity collection:

```
button1.click = function (event)
{
    var myset = ds.Person.query("ID > :1 and ID < :2", {
        params: [100, 300], // searches for IDs between 100 and 300
        pageSize: 60, // we want to retrieve the entities by groups of 60
        onSuccess:function(event) // we pass a single function that receives the server response
        {
            var count = event.entityCollection.length;
                    // we could have also written:
                    // var count = myset.length
                    // because the myset reference is valid in this case
            $("#display").html("selection : "+count);
                    // display in the container that has "display" as its ID (jQuery notation)
        }
    });
};
```

Example

In this example for a button, the code calls the "gotEntity" callback function, that is specified at another location in the script, and passes it parameters through userData:

```
button1.click = function (event)
{
    var p = ds.getDataClass("People"); // we retrieve the datastore class
    var myset = p.query("id > 100 and id < 300 order by name", { // sorted search
        onSuccess:function(event) // callback function specified here
        {
            myset.getEntity(0, {onSuccess: gotEntity }, // callback function specified elsewhere in
the script
            {
                curelem: 0, // passing values in userData
                maxelem: myset.length,
                html: ""
            });
        }
    });
};
```

Example

This example illustrates the use of the **autoExpand** option: it lets you cut down on requests when accessing relation attributes.

```
button2.click = function (event)
{
    ds.Employee.query("lastName = 'Jones'", { autoExpand:"employer", onSuccess:function(event)
                // search for an employee and retrieve his/her employer
    {
        var myset = event.entityCollection; // we retrieve the resulting entity collection in myset
        myset.getEntity(0, { onSuccess:function(event) // we load the first entity of the entity
collection
                    // it is possible that this does not generate any server request because this
entity
                    // has already been sent back by the server but using this syntax guarantees
the result
            {
                var myEntity = event.entity;
                var html = ""; // initialize the display
                html += "Last Name: "+myEntity.lastName.getValue()+"<br/>"; // direct access to a
storage attribute
                html += "First Name: "+myEntity.firstName.getValue()+"<br/>";
                myEntity.employer.getValue( { onSuccess: function(event) // access to a relation
attribute
                {   // it is therefore necessary to nest an asynchronous call
                    // because of the autoExpand option, this access does not trigger a new request
                    var comp = event.entity;
                    if (comp != null) // if an entity has actually been found
                            // if an entity has not been found, this is not an error
                        html += "works for: "+comp.name.getValue()+"<br/>";
                    $("#display").html(html);
                } });

        } });
    }});
};
```

setCacheSize( )

void **setCacheSize**(Number *cacheSize*)

Parameter	Type	Description
cacheSize	Number	New size of entity cache

Description

The **setCacheSize( )** method sets a new size on the client for the entity cache of the datastore class.

The entity cache is the maximum number of entities kept on the client and managed by the Dataprovider to optimize access to the data Cache management is transparent for the client.

By default, the size is 300 entities. You can increase this value according to your application's needs.

Note: The default size of 300 is a minimum: it allows the application to provide a good level of performance. If you pass a value less than 300, Wakanda uses the default value of 300.

Example

You want to set the entity cache size to 500 for the Company datastore class:

```
ds.Company.setCacheSize(500);
```

Datastore

The methods in this theme return general information about the datastore model and let you change it. These methods belong to the WAF.DataStore class.

[className]

Description

Each of the datastore classes in your application is made available as a property of the **ds** object, which is a shortcut to the default application.

ds

Description

The **ds** property is a reference to the Wakanda application's current datastore (see Accessing a Datastore). This reference is used in your client-side code as a shortcut to **WAF.ds** to reference the current datastore and access its datastore classes.

Example

Each class in the current datastore is available directly on the client as a property of the **ds** object:

```
var theTeachers = ds.Teacher; // returns the Teacher datastore class of the current datastore as an object
```

getDataClass( )

DatastoreClass **getDataClass**(String *className*)

Parameter	Type	Description
className	String	Name of datastore class to retrieve

Returns	DatastoreClass	Datastore class of this name in the datastore model

Description

The getDataClass( ) method returns the DatastoreClass object whose name is the same as the string passed in the className parameter of the current datastore. This method is useful when writing generic code.

Example

There are two ways to retrieve a datastore class object using the Dataprovider:

```
var myName = "Employee";
var myClass = ds.getDataClass( myName );

// exactly the same as:
var myClass = ds.Employee;
```

getDataClasses( )

Object **getDataClasses**( )

Returns ObjectDatastore classes of current datastore

Description

The **getDataClasses( )** method returns all the datastore classes specified in the model of the current datastore along with their attributes and methods.

The information that this method returns is similar to the information that is available directly in the **ds** object except that it only contains datastore classes, and not datastore functions or the "_private" object. This method makes it easier to count, list, and work with datastore classes.

Example

To get the list of datastore classes:

```
var classes = ds.getDataClasses( );
```

WAF.DataStore.getCatalog( )

void **WAF.DataStore.getCatalog**(Object *options*)
ParameterType Description
options ObjectBlock of options for asynchronous execution

Description

The **WAF.DataStore.getCatalog( )** method gets the datastore model for an application other than the current one (the current datastore model is obtained directly through the **ds** object).

This method must be executed in asynchronous mode. You must pass several objects to specify the request in the options parameter:

- **app**: designates the name of the application whose datastore model you want to retrieve. In this parameter, you pass the ID (name or IP address) of the application (the project in the solution).
- **catalog**: designates the datastore classes to be retrieved. You can pass:
  - null (or *catalog* object omitted) = retrieve all the datastore classes
  - a string containing datastore classes in the form "class1", "class2", and so on
  - an array of datastore class names
- **onSuccess**: function to be called back once the server response is received and no error is generated
- **onError**: function to be called back when an error is generated

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You

can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Entity

The methods of this class apply to objects of the Entity type.

Working with Entities on the Client

As on the server, objects of type Entity contain properties, which are the attributes that make up the datastore class.

However, there is one significant difference with respect to the server (see Working with Entities) concerning the assignment and reading of the entity's attribute values. On the server, you access these values through object notation:

```
// Assigning and reading on server, not possible on client
myValue = ds.Employee.first().lastname; // Reading on server
 ds.Employee.first().lastname = "Jones"; // Assigning on server
```

For internal reasons, this principle does not work on the client. You must use the getValue( ) and setValue( ) methods:

```
myValue = ds.Employee.getEntity(0).lastname.getValue(); // Reading on client
 ds.Employee.getEntity(0).lastname.setValue( "Jones"); // Assigning on client
```

Also note that unlike datasources on the client (see Using Server Datasources), there is no notion of a current entity on the Dataprovider. You can work on the Dataprovider with several entities in the same entity collection simultaneously.

Creating a new entity

To create a new entity with the Dataprovider, you can use either the newEntity( ) method of the DataClass class or the **new** operator with the **WAF.Entity** function (constructor of the WAF.Entity class). To both, you pass the datastore class as a parameter:

```
// Create a new entity in the MyClass datastore class
var newEntity = ds.MyClass.newEntity();

// equivalent to:
var newEntity2 = new WAF.Entity ( ds.MyClass );
```

Either one of these statemenst create a new entity in the datastore class and by default assigns the *null* value to each attribute of the entity.

Note that the entity is not saved in the datastore until you call the save( ) method. A complete example for creating an entity is provided in the documentation of this method.

Access to entity methods

On objects of type Entity, you can access datastore class methods of type "entity", created in the Datastore Model Designer, provided that their scope is "Public". You cannot call Datastore class methods of type "entityCollection" or "class" on these objects.

The call can be synchronous or asynchronous (see Accessing Datastore Objects with the Dataprovider).

callMethod( )

Mixed **callMethod**(Object *options* [, String *params*])

Parameter	Type	Description
options	Object	Block of options for asynchronous execution
params	String	Parameter(s) to pass to the datastore class method

Returns Mixed Value returned by the method in synchronous mode

Description

The **callMethod( )** method executes a datastore class method on the entity, entity collection, or datastore class to which it is applied.

When you call this method in asynchronous mode, the result, if any, is retrieved through the **event.result** property in the callback function defined in the options parameter.

Note: You can call a datastore class method directly as the property of an entity, entity collection, or datastore class (see Calling Datastore Class Methods). The main advantage of using the callMethod( ) method is that it is useful when writing generic code because the method name is passed as a string.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the *options* parameter, you pass an object containing the "onSuccess" and "onError" callback functions as well as additional properties (that depend on the method called). Each callback function receives a single parameter, which is the event.

You can also pass directly the onSuccess and onError functions as parameters to the **callMethod( )** method. In this case, they must actually be passed just before (and outside) the object parameter.

With the **callMethod( )** method, you must pass the name of the method as a member of the options object: {method:"MethodName"}. *MethodName* must be passed as a string. For example:

```
method:"moreThanAverage"
```

The following attributes are also available in the options block for the **callMethod( )** method:

- **pageSize**: *number* (e.g., **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. The default value is 40. If the entity collection contains 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a Grid or Matrix widget.
  You can vary the value sent to this parameter for optimization reasons, e.g., according to the height of the widget.

- **autoExpand**: *string containing one or more relation attributes* (e.g., **autoExpand: "worksFor, fatherOf"**)
  By default, the values of relation attributes are not expanded in the entity collections returned by the server for optimization reasons. They are only expanded on request, i.e., when a user or a function accesses them (the Dataprovider then sends the corresponding requests automatically). You may want to preload these values in order to display them or to access them in the case of nested asynchronous requests.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to later retrieve. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any JavaScript valid value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters...) that you want to use again in the callback function.

- **params**: *arguments array* (e.g., **'params' : [100, 200,300]**)
  Array of parameters to pass to the datastore class method. These parameters are used in the order they are defined for the array.
  With this attribute, you pass all the parameters as strings in the params parameter. If you pass the parameters directly to the datastore class method, the params parameter is not needed.

params

You use the params parameter to pass one or more parameters to the datastore class method (except when you are using the params attribute of the options block). You must pass parameters as strings and delimit each one by commas.

getDataClass( )

DatastoreClass **getDataClass**( )

Returns DatastoreClassDatastore class to which the entity belongs

Description

The **getDataClass( )** method returns the DatastoreClass to which the entity belongs. This method is useful for writing generic code.

getKey( )

String | Number **getKey**( )

Returns Number, StringPrimary key value of entity

Description

The **getKey( )** method returns the value of the entity's primary key.

The value type depends on the type of the storage attribute designated as the datastore class's primary key. This value is unique among all the entities in the datastore class.

getStamp( )

Number **getStamp**( )

Returns NumberCurrent value of internal stamp of the entity

Description

The **getStamp( )** method returns the current value of the entity's internal *stamp* on the client.

The internal stamp of entities is used by the mechanism that manages simultaneous modifications of entities from different clients and/or contexts. For more information about this mechanism, refer to Locking Entities in the Datastore API manual (server-side).

isNew( )

Boolean **isNew**( )

Returns BooleanTrue if entity has just been created; otherwise, False

Description

The **isNew( )** method returns True when the entity to which it is applied has just been created on the client (and is not yet saved on the server). In all other cases, this method returns False.

You can use this method to call specific code when an entity has been created.

isTouched( )

Boolean **isTouched**( )

Returns BooleanTrue if entity has been modified; otherwise, False

Description

The **isTouched( )** method returns True or False according to whether or not the entity or attribute has been modified.

This method tests, for example, if an attribute was modified by a user so as to perform any necessary processing.

remove( )

void **remove**([Object *options*])
ParameterType Description
options ObjectBlock of options for asynchronous execution

Description

Note: You can also call this method using its alias drop( ).

The **remove( )** method deletes the entity from the datastore on the server. Executing this method triggers a call to the **onRemove** event on the server if one was specified for the datastore class or one of the entity's datastore class attributes. The deletion of an entity can be refused and this event can return an error (see Refusing an event).

Note that if the deleted entity is used in one or more existing entity collections, it is not removed from the displaying of the entity collections. You have to update the entity collections on the clients yourself.

If the entity has already been deleted, an error is returned on the stack in the parameter of the "onError" callback function (or in the *event* parameter of the callback function if you only use a single function). You can access the error stack through the *event.error* array, for example **event.error[0].message**, to get the message of highest-level error returned. For more information, please refer to the Error Management section. Note that when there is an error, the server returns the values of the entity in the callback function as they were saved in the datastore so that you can display them (see Locking Entities in the Datastore API manual).

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **remove( )** method. In this case, they must be passed just before (and outside) the options parameter.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

save( )

void **save**([Object *options*])
ParameterType Description
options ObjectBlock of options for asynchronous execution

Description

The **save( )** method saves the modifications made to a specific entity in the datastore. On the server, any code associated with the datastore class's **onValidate** and **onSave** events is executed. If you want to save the entity, you must call this method after creating or modifying each entity.

This method is called asynchronously so you have to retrieve the entity as it was saved through the **event.entity** property in the callback function specified in the options parameter. Note that in this case, you access the entity's attributes as they were saved on the server, i.e., after applying the business rules defined on the server. In this way, you can retrieve any calculated values. Any not null values that were modified are returned.

If the entity was modified by another user between the time it was loaded and the time it was saved, an error is returned onto the stack as a parameter of the "onError" callback function (or in the *event* parameter of the callback function if you only use a single function). You can access the error stack through the *event.error* array, for example **event.error[0].message** to get the message of highest-level error returned. For more information, please refer to the Error Management section. Note that when there is an error, the server returns the values of the entity in the callback function as they were saved in the datastore so that you can display them for example (see Locking Entities in the Datastore API manual).

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions section.

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **save( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **save( )** method:

- **overrideStamp**: *boolean* (example: **overrideStamp:true**)
  Note: This option is taken into account only if you have checked the "Allow Stamp Override" box in the Datastore Model Designer for the target datastore class: 

  If you pass *{overrideStamp:true}* in the options parameter of the **save( )** method, you bypass the internal entity locking mechanism running on the server (see the Locking Entities section). As a consequence, the entity or current entity will be saved regardless of the client-side and server-side stamp values -- that is, no error will be generated if these values do not match. Since some internal checks are then disabled, this option can speed up the saving process on the server.
  This feature must be used wisely because data are no longer protected against concurrent client modifications. It should be reserved for "admin" features or specific data on which conflicts cannot occur.
  By default, the value is **false**: stamp control is on.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

The following script for a Button creates a new entity, assigns default values to it, saves it, and displays the values returned by the server, which contain more particularly a calculated attribute, in a container:

```
createBlankButton.click = function (event)
    {
        var e = ds.People.newEntity(); // create an entity with attributes set to null
        e.lastName.setValue("Last name"); // enter a few default attributes
        e.firstName.setValue("First name");
        e.save({
            onSuccess:function(event) // if the save on the server is performed correctly
```

```
            {
                var html = "";
                html += " ID : "+event.entity.ID.getValue()+"<br/>"; // contains the ID of the
entity (assigned automatically)
                html += " fullName : "+event.entity.fullName.getValue()+"<br/>"; // contains a
calculated field
                $("#display").html(html); // display the values in the container whose ID is
'display'
            },
            onError: function(error) // save has failed
            {
                var mess = error.error[0].message ;
                $("#display").html("error :"+ mess +"<br/>");
            }
            });
    };
```

touch( )

void **touch**( )

Description

The **touch( )** method indicates that the entity or one of its attributes must be saved during the next save( ).

If you apply this method to an attribute, its effect is automatically extended to the entity (the isTouched( ) method returns True for the entity).

Since the save( ) method is optimized, an entity is only saved when the engine detects that at least one of its attributes was modified since the last save. You do not usually need to use the **touch( )** method since the Dataprovider dynamically detects modifications made to the entity, more specifically when you use the setValue( ) method. However, you may want to "force" the request to save an entity in specific cases. To do so, just execute this method on the entity.

After saving the entity, all the "touch" values are reset (the isTouched( ) method returns False for the entity).

Entity attribute

The methods of this theme apply directly to datastore class attributes of entities. These methods are available in three classes: WAF.EntityAttributeSimple, WAF.EntityAttributeRelated and WAF.EntityAttributeRelatedSet.

Using entity attributes

Accessing Datastore Class Attributes

Each entity contains a representation of the datastore class attributes specified in the datastore model as well as any values that are associated with them.

Entity attributes are accessible as properties of the entities. To get the value of an attribute, you must call the and methods on the attribute. For example:

```
myValue = ds.Employee.first().lastname.getValue(); // Getting the value of an attribute client-side
ds.Employee.first().lastname.setValue( "Jones"); // Assigning the value to an attribute client-side
```

Note: Do not confuse entity attributes with those of the datastore class (objects of type Attribute are specified in the datastore model). For more information, refer to Accessing Datastore Objects with the Dataprovider.

relEntityCollection

Description

The **relEntityCollection** property returns the collection of related entities from the related datastore class. This property is only available for "one" relation attributes of a N->1 relation.

The returned object is of the EntityCollection type. Such objects are handled through the EntityCollection class of the Dataprovider API.

Example

In a classic Employee -> Company model, you want to get the related entity collection of employees for the currently selected company. 'employees' is the N relation attribute in the Company datastore class.

```
var curCompany = sources.company.getCurrentElement(); // gets the current entity
var curCompEmployees = curCompany.employees.relEntityCollection;
    // returns the collection of employees of the current company
```

getValue( )

Mixed **getValue**([Object *options*])
ParameterType Description
options ObjectBlock of options for asynchronous execution

Returns Mixed Value of entity attribute

Description

The **getValue( )** method gets the value of the entity's attribute. You need to call this method in the Dataprovider API so as to be sure that the access to the values between the server and client is synchronized.

- When you access a storage attribute, i.e., a simple type, there is no request sent to the server for you to retrieve the attribute's value directly. The syntax to use is:

  ```
  value = myEntity.attributeName.getValue();
  ```

- When you access a relation attribute, i.e., a complex type, a request may be sent to the server for you to use the syntax for asynchronous execution and retrieve the attribute's value in the **event.entity** object of the callback function. The syntax to use is:

  ```
  myEntity.attributeName.getValue({function(event), userData, other_options});
  ```

  If the **autoexpand** option has been applied to one or more relation attributes during the initial query (using the query( ) method, for example), access to the data through the relation attribute does not necessarily trigger a request to the server, which optimizes application operation. However, it is necessary to use the asynchronous syntax in this case.

As part of writing generic code, you can find out the attribute type ("storage" or "relation") through its 'kind' property. You can get this property by using the getAttributeByName( ) method.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **getValue( )** method. In this case, they must be passed just before (and outside) the options parameter.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

This example illustrates how to nest asynchronous calls. This Button's script performs a query on the server and then accesses the values of the entity's attributes including its relation attributes. The results are then displayed in a container. Note that you can only obtain a valid result by accessing the entity from inside the last asynchronous query performed on the server:

```
button1.click = function (event)
{
    var myset = ds.People.query("id > 100 and id < 300", { // query in asynchronous mode
        autoExpand: "father,father.father,mother", // precalculate relation attributes
        function(event) // callback function
        {
            event.entityCollection.getEntity(5, { // access the 5th entity, in asynchronous mode
                function(event) // callback function
                {
                    var myEntity = event.entity; // retrieve the entity
                    var html = "";
                    html += "ID : "+myEntity.ID.getValue() + "<br/>"; // access to storage
attributes, the query is direct
                    html += "name : "+myEntity.name.getValue() + "<br/>";
                    html += "firstname : "+myEntity.firstname.getValue() + "<br/>";
                    html += "wages : "+myEntity.wages.getValue() + "<br/>";

                    myEntity.father.getValue({onSuccess: function(event) //access to relation
attribute
                    { // the query is therefore asynchronous
                        var father = event.entity;
                        html+= "father's name: "+father.name.getValue()+"<br/>";
                        html+= "father's firstname: "+father.firstname.getValue()+"<br/>";
                        $("#display").html(html); // display can only be done here
                    }});
                }
            });
        }
    });
};
```

isTouched( )

Boolean **isTouched**( )

Returns BooleanTrue if entity has been modified; otherwise, False

Description

The **isTouched( )** method returns True or False according to whether or not the entity or attribute has been modified.

This method tests, for example, if an attribute was modified by a user so as to perform any necessary processing.

setValue( )

void **setValue**(Mixed *value*)
ParameterType Description
value Mixed New value of entity attribute

Description

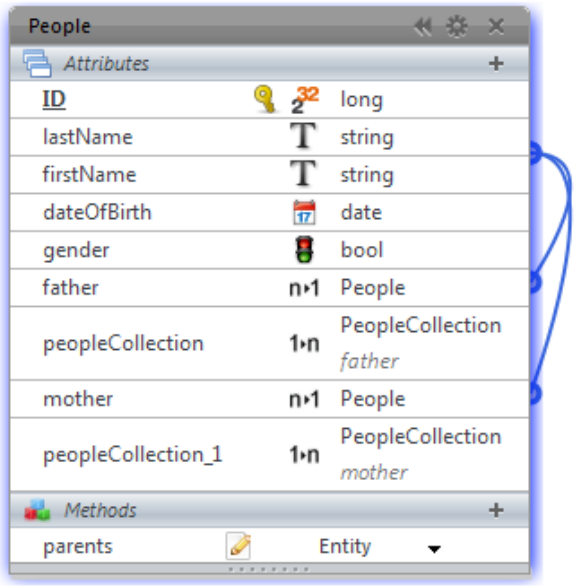
The **setValue( )** method modifies the entity's attribute value. Its syntax is:

```
myEntity.attributeName.setValue( myValue );
```

Calling this method is necessary in the Dataprovider API in order to ensure the proper synchronization between the server and client.

Example

Given the following datastore class that stores people and their relatives:



The following Button script creates an entity and associates it to its father:

```
    bindEntities.click = function (event)
    {
        ds.People.find('lastName = "Lawson"', { onSuccess:function(event) // look for the father
        {
            if (event.entity != null) // an entity is found
            {
                var theFather = event.entity; // store the father in a variable
                var e = ds.People.newEntity(); // create a new blank entity
                e.name.setValue("Lawson"); // assign the values values
                e.firstName.setValue("Peter");
                e.father.setValue(theFather); // assign the relation attribute and pass an entity
to it
                e.save(); // save in synchronous mode
            }
        } });
    };
```

touch( )

void **touch**( )

Description

The **touch( )** method indicates that the entity or one of its attributes must be saved during the next .

If you apply this method to an attribute, its effect is automatically extended to the entity (the method returns True for the entity).

Since the method is optimized, an entity is only saved when the engine detects that at least one of its attributes was modified since the last save. You do not usually need to use the **touch( )** method since the Dataprovider dynamically detects modifications made to the entity, more specifically when you use the method. However, you may want to "force" the request to save an entity in specific cases. To do so, just execute this method on the entity.

After saving the entity, all the "touch" values are reset (the method returns False for the entity).

EntityCollection

The methods of this theme apply to objects of the EntityCollection type.

Working with Entity Collections on the Client

Creating and working with entity collections on the client through the Dataprovider works in roughly the same way as it does on the server with the Datastore API (see Working with Entity Collections on the Server).

- You can get the number of entities in the entity collection by using the **length** property:

```
var nbEntities = myEntSet.length; // Returns the number of entities in the entity collection
```

- You can also create a new entity collection either by executing a query( ) or by calling a specific method, newCollection( ), on the client.

However, there is one significant difference as far as the selection of an entity from an entity collection is concerned: you cannot use standard array syntax (using brackets [ ]) on the client. You must use the getEntity( ) method:

```
var myEntity6 = myCollection[5]; // Server syntax, not possible on client
myCollection.getEntity(5, { // Correct syntax on client (asynchronous call)
    function(event) {
    {
        var myEntity6 = event.entity;
    }
});
```

See also the example of the getEntity( ) method.

Accessing Entity Collection Methods

On objects of type EntityCollection, you can access datastore class methods of type "entityCollection" provided their scope is "Public". You cannot call datastore class methods of type "entity" or "class" on these objects.

The call can be synchronous or asynchronous (see Calling Datastore Class Methods). For example:

```
var mySet = ds.Employee.query("salary > 20000");
var sumSalary = mySet.getSumSalary(); // synchronous call of a datastore class method of type
entityCollection
```

length

Description

The **length** property returns the current number of entities in the entity collection. For example:

```
var numEntities = myEntSet.length; //returns the number of entities in the "myEntSet" entity
collection
```

add( )

void **add**(Entity *entity*)

Parameter	Type	Description
entity	Entity	Entity to add

Description

The **add( )** method adds an object of type Entity passed as a parameter to the end of the entity collection.

This entity must have been already created or loaded and must belong to the same datastore class as the entity collection.

Example

In this example, we fill a datasource with entities selected from another datasource. In the GUI Designer, the Page appears as shown below:

Here is the script for the **Add >>** button:

```
button1.click = function (event)
    {
        var emp = sources.employee.getCurrentElement(); // retrieve the current element, which in
the case of a datasource is an entity
        var destSet = sources.mySet.getEntityCollection(); // retrieve the datasource's current
entity collection to the right
        if (emp != null) // if there is a selected entity
        {
            destSet.add(emp); // add the entity on the left to the entity collection
            source.mySet.setEntityCollection(destSet); // reapply the entity collection to the
datasource's collection
                // this then triggers events that update subscribers to the datasource
        }
    };
```

buildFromSelection( )

EntityCollection **buildFromSelection**(Selection *entitySelection* [, Object *options*])

Parameter	Type	Description
entitySelection	Selection	Subcollection of entities from the entity collection
options	Object	Block of options for asynchronous execution

Returns	EntityCollection	New entity collection

Description

The **buildFromSelection( )** method returns a new entity collection based on the entity Selection you passed in the entitySelection parameter.

An entity selection is a subset of the entity collection to which the method is applied. It references the current position of entities within the collection and is mainly useful in the context of a list-oriented widget. For more information, refer to the buildFromSelection( ) method description in the Datasource API.

This method is called asynchronously, so you have to retrieve the resulting entity collection for the search in the callback function set in the options parameter, usually using the **event.entityCollection** property. However, note that in the context of a **buildFromSelection( )** method executed on the client, Wakanda lets you use the entity collection returned by the method, even if at first it is in a "non-finalized" state. The reference to this entity collection is valid so you can work with it and use it once the callback function is called successfully.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **buildFromSelection( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **buildFromSelection( )** method:

- **pageSize**: *number* (example: **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. By default, the value is 40: if the entity collection contains 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a list.
  You can have this parameter vary for optimization issues, according, for example, to the size of the widgets.

- **autoExpand**: *string containing one or more relation attributes* (example: **autoExpand: "worksFor, livesIn"**)
  By default, the values of relation attributes are not calculated in the entity collections returned by the server, for optimization reasons. Access to these values automatically triggers the corresponding requests. You may want to precalculate these values, for example to be able to display them.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

On a Page for a hand-made items store, we want to display a list of items and a list of selected items. We add lists as Grids, based on two different datastore class datasources, "item" and "item2". The main list (on the left side) is in **Multiple** selection mode. At runtime, each time an item is selected or deselected in the main list by the user, the "Selected Items" list is updated:

Christelle's Store Items

ID	name	category	price
5	Pink	Lampshade	15
6	Pig and Sheep	Wood painting	26
7	Turtle	Wood painting	22
8	Blue pitcher	Glass painting	30
9	Dragonfly	Glass painting	35

9 items

Selected Items

ID	name	category
3	Giraffes	Wood painting
5	Pink	Lampshade
9	Dragonfly	Glass painting
8	Blue pitcher	Glass painting

4 items

To do this, we just had to write the following code in the **On Current Element Change** event of the main list datasource:

```
itemEvent.onCurrentElementChange = function itemEvent_onCurrentElementChange (event)
     {
         function buildsel(event) // callback function to update the collection of the right list
         {
             var collec = event.entityCollection; //gets the new entity collection from the event
             sources.item2.setEntityCollection(collec); //assigns the collection to the datasource
                //appropriate events are automatically generated
         }

         var sel = sources.item.getSelection(); // gets the selection of the main list
         var col = sources.item.getEntityCollection(); // gets the collection of the main list
         col.buildFromSelection(sel, { onSuccess: buildsel }); // returns a new collection based on
sel
     };
```

callMethod( )

Mixed **callMethod**(Object *options* [, String *params*])
ParameterType Description
options ObjectBlock of options for asynchronous execution
params String Parameter(s) to pass to the datastore class method

Returns Mixed Value returned by the method in synchronous mode

Description

The **callMethod( )** method executes a datastore class method on the entity, entity collection, or datastore class to which it is applied.

When you call this method in asynchronous mode, the result, if any, is retrieved through the **event.result** property in the callback function defined in the options parameter.

Note: You can call a datastore class method directly as the property of an entity, entity collection, or datastore class (see Calling Datastore Class Methods). The main advantage of using the callMethod( ) method is that it is useful when writing generic code because the method name is passed as a string.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the *options* parameter, you pass an object containing the "onSuccess" and "onError" callback functions as well as additional properties (that depend on the method called). Each callback function receives a single parameter, which is the event.

You can also pass directly the onSuccess and onError functions as parameters to the **callMethod( )** method. In this case, they must actually be passed just before (and outside) the object parameter.

With the **callMethod( )** method, you must pass the name of the method as a member of the options object: {method:"MethodName"}. *MethodName* must be passed as a string. For example:

```
method:"moreThanAverage"
```

The following attributes are also available in the options block for the **callMethod( )** method:

- **pageSize**: *number* (e.g., **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. The default value is 40. If the entity collection contains 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a Grid or Matrix widget.
  You can vary the value sent to this parameter for optimization reasons, e.g., according to the height of the widget.

- **autoExpand**: *string containing one or more relation attributes* (e.g., **autoExpand: "worksFor, fatherOf"**)
  By default, the values of relation attributes are not expanded in the entity collections returned by the server for optimization reasons. They are only expanded on request, i.e., when a user or a function accesses them (the Dataprovider then sends the corresponding requests automatically). You may want to preload these values in order to display them or to access them in the case of nested asynchronous requests.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to later retrieve. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any JavaScript valid value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters...) that you want to use again in the callback function.

- **params**: *arguments array* (e.g., **'params' : [100, 200,300]**)
  Array of parameters to pass to the datastore class method. These parameters are used in the order they are defined for the array.
  With this attribute, you pass all the parameters as strings in the params parameter. If you pass the parameters directly to the datastore class method, the params parameter is not needed.

params

You use the params parameter to pass one or more parameters to the datastore class method (except when you are using the params attribute of the options block). You must pass parameters as strings and delimit each one by commas.

distinctValues( )

void **distinctValues**(DatastoreClassAttribute | String *attribute* [, Object *options*])

Parameter	Type	Description
attribute	DatastoreClassAttribute, String	Attribute for which you want to get the list of distinct values
options	Object	Block of options for asynchronous execution

Description

The **distinctValues( )** method retrieves an array containing all the distinct values stored in attribute for the entity collection or datastore class. By default, this command takes into account all the entities in the datastore class or entity collection to calculate the distinct values; however, you have the option of filtering them through the attributes of the options block so as to limit the number of entities.

Since this method is called asynchronously, you must retrieve the resulting array in the callback function specified in the options parameter through the **event.distinctValues** property.

Note: You can also use the event.result property.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **distinctValues( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following attributes are also available in the options block for the **distinctValues( )** method:

- **skip**: *numeric value* (example: **skip: 20**)
  This method starts the array of distinct values at the *X* value defined by *skip*.
- **top**: *numeric value* (example: **top: 40**)
  This method returns a set of *X* elements in the array of distinct values, starting from the first one or from the one defined by *skip*.

  The above parameters are useful when you want to paginate the results.
- **progressBar**: *string indicating a progress bar ID* (example: **progressBar: "myProgressBarID"**)
  The ID referencing an existing Progress Bar widget that the server will use to indicate the method's state of progress.
- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

findKey( )

void **findKey**(Number | String *key* [, Object *options*])

Parameter	Type	Description
key	Number, String	Primary key value

options	Object	Block of options for asynchronous execution

Description

The **findKey( )** method returns the position in the entity collection of the entity whose primary key is passed in key.

Since this method must be called in asynchronous mode, the position is retrieved through the **event.result** property in the callback function specified in the options parameter.

If there is no matching entity in the entity collection, this method returns -1.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **findKey( )** method. In this case, they must be passed just before (and outside) the options parameter.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

forEach( )

void **forEach**([Object *options*])

ParameterType	Description
options	ObjectBlock of options for asynchronous execution

Description

Note: You can also call this method using its alias each( ).

The **forEach( )** method executes a function on each entity in the entity collection in ascending order.

If the function modifies the entity, you must call the **save( )** method for each entity in order to save it.

In the functions called back by the server (set through the option parameter), you can retrieve the entity collection and the entity being processed through the following properties:

- **event.entityCollection**: current entity collection
- **event.entity**: entity being processed. *Warning: this property is empty in the function called after processing the last entity ("atTheEnd" function or its equivalent).*
- **event.position**: position of the entity being processed in the entity collection.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the *options* parameter, you pass an object containing the 'onSuccess', 'onError' and 'atTheEnd' callback functions as well as additional properties (that depend on the method called). Each callback function receives a single parameter that is the event.

The '**atTheEnd**' function is called after processing the last entity of the entity collection. You must pass the final code of the iteration (for example, the display of the result) in this function. Note that in this event, there is no longer an entity (**event.entity** is empty).

You can also pass directly the onSuccess, onError and atTheEnd functions as parameters of the **forEach( )** method. In this case, they must actually be passed just before (and outside) the object parameter. It is mandatory to pass onError, even if you do not use it.

The following parameters are also available in the options block for the **forEach( )** method:

- **autoExpand**: *string containing one or more relation attributes* (example: **autoExpand: "worksFor, fatherOf"**)
  By default, the values of relation attributes are not calculated in the entity collections returned by the server, for optimization reasons. They are only calculated on request, when the user or a function accesses them (the Dataprovider then sends the corresponding requests automatically). You may want to precalculate these values, for example to be able to display them or to access them in the case of nested asynchronous requests.

- **skip**: *numeric value* (example: **skip: 20**)
  Function not applied to the first *X* entities of the entity collection.

- **top**: *numeric value* (example: **top: 40**)
  Function only applied to the first *X* entities of the entity collection, starting from the first entity or from the one resulting from the value of the *skip* attribute.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

We create an utility function named "gotTeachers" which displays a list of teachers in the container with id "display". This function itself can be called back by other functions which pass an entity collection as a parameter, for example functions querying the teachers.

```
function gotTeachers(event)
{
    var myset = event.result; // as it is a non-specialized function,
                                        // the result is returned in event.result
    source.Teachers.setEntityCollection(myset);  // The entity collection is applied to the
Teachers datasource (optional)
                        // This point is discussed in the Datasource API documentation

    var html = ""; // initialization
    myset.forEach({  // on each of the entity collection
        onSuccess: function(event)
        {
            var entity = event.entity; // get the entity from event.entity
            html += event.position + " : " + entity.fullName.getValue()+"<br/>";
                 // event.position contains the position of the entity in the entity collection
                 // you get the attribute value with entity.attribute.getValue()
        },
        onError: function(event)
        {
            $("#display").html("An error has been returned");
        },
        atTheEnd: function(event)
        {
            $("#display").html(html); // display of the final result
        },
    });
}
```

getDataClass( )

DatastoreClass **getDataClass**( )

Returns DatastoreClass Datastore class to which entity collection belongs

Description

The **getDataClass( )** method returns the DatastoreClass to which the entity collection belongs. This method is mainly used for writing generic code.

getEntity( )

Entity **getEntity**(Number *position* [, Object *options*])

Parameter	Type	Description
position	Number	Position in entity collection of entity to return
options	Object	Block of options for asynchronous execution

Returns Entity Entity returned in case of synchronous execution

Description

The **getEntity( )** method retrieves the entity whose position in the entity collection is passed to the position parameter.

Warning: This method expects a different parameter (the primary key) when it is applied to a datastore class (see getEntity( )).

Since this method must be called in asynchronous mode, the entity is retrieved through the **event.entity** property in the callback function specified in the options parameter.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **getEntity( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following attributes are also available in the options block for the **getEntity( )** method:

- **forceReload**: *boolean* (example: **forceReload: true**)
  When this option is set to **true**, the latest version of the entity is always reloaded from the server. In this way, you can get the internal stamp values if they were modified on the server in the meantime. If this option is set to **false** or omitted, the **getEntity( )** method may get the entity from the local cache.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

In this example, we want to perform a search in a datastore class and display in a container the information related to the entity found in the 5th position of the entity collection:

```
var myset = ds.Person.query("lastname = :1 and ID > :2", {
    params: ['A@', 100] // look for people whose last name begins with A and whose ID >100
    'onSuccess':function(event) // first callback because query is asynchronous
    {
        event.entityCollection.getEntity(5, { // we request that the 5th entity of the entity
```

```
collection be returned
            'onSuccess':function(event) // second callback because the getEntity method is
asynchronous
            {
                var userData = event.userData; // contains { x: 1, y: 2 }
                var myCount = userData.x++ // we can modify the value of the object
                var myEntity = event.entity; //we retrieve the entity
                var html = ""; // build the contents of the container
                html += "ID : "+myEntity.ID.getValue() + "<br/>"; // access the entity's attributes
                html += "Last Name: "+myEntity.lastName.getValue() + "<br/>"; // with the getValue(
) method
                html += "First Name: "+myEntity.firstName.getValue() + "<br/>";
                html += "wages: "+myEntity.wages.getValue() + "<br/>";

                $("#display").html(html); // display in container with the "display" ID (jQuery
notation)
            },
            'userData': { // example of data placed in userData
                x: 1,
                y: 2
            }
        }
        );
    }
});
```

getReference( )

Object **getReference**( )

Returns ObjectReference to the entity collection on the server

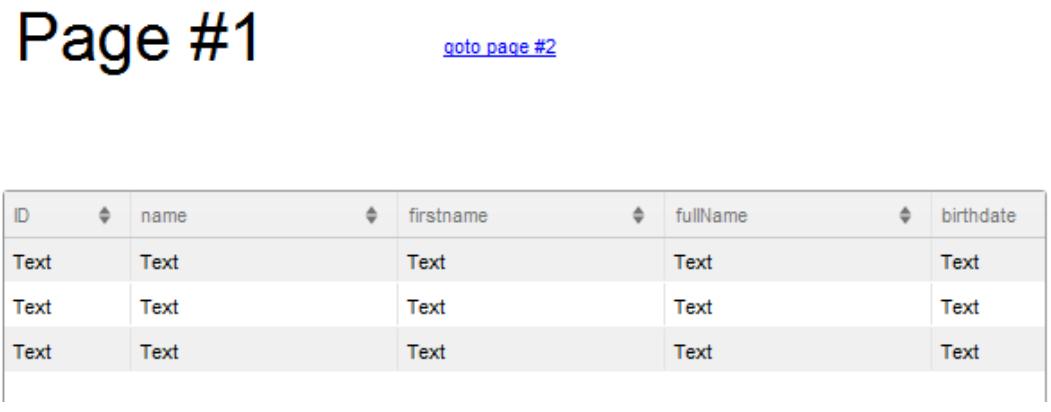
Description

The **getReference( )** method returns the internal reference of the entity collection on the server.

The returned object is mainly useful to allow client-side multi-page browsing without having to regenerate the entity collection: you can store the reference locally (for example in the local *sessionStorage*) and reuse it in other pages by using the newCollection( ) method.

Example

In our example, users can switch from one page to another by clicking a link. Each page contains a widget displaying data from the same server datasource, named "person", and a link to go to another page. We want them to keep their current entity selection during navigation.



- On each page, the navigation link (a Text widget in our example) has the following code in the **On Click** event:

```
    richText2.click = function richText2_click (event)
    {
        var ref = sources.person.getEntityCollection().getReference();  // gets the reference
of the current entity collection
        // the reference is an object; therefore, we must convert it as a string to be able
to store it in the client sessionStorage
        var savedRef = JSON.stringify(ref);
        sessionStorage.myCollectionSavedRef = savedRef; // locally stores the collection
reference as a string
    };
```

In the **On Load** event of each page, we load the saved entity reference (if any):

```
    documentEvent.onLoad = function documentEvent_onLoad (event)
    {
        var savedRef = sessionStorage.myCollectionSavedRef; // get the saved collection
reference
        if (savedRef == null) // if there is no stored collection
            sources.person.all(); // display all entities
        else
        {
            var ref = JSON.parse(savedRef); //create a valid reference object from the stored
string
            ds.Person.newCollection(ref, function(event) // asynchronous call for the
reference
            {
                var newcol = event.entityCollection; // get the collection saved on the server
                sources.person.setEntityCollection(newcol); // assign the collection to the
datasource
            });
        }
    };
```

As the collection reference contains the original query, the entity collection will be available even if the server has been restarted.

orderBy( )

void **orderBy**(String | DatastoreClassAttribute *attributeList* [, String *sortOrder*] [, Object *options*])

Parameter	Type	Description
attributeList	String, DatastoreClassAttribute	Attribute(s) to sort and (if string) sort direction(s)
sortOrder	String	Order by direction(s) (if first param is Attribute type), asc = ascending sort (default), desc = descending sort
options	Object	Block of options for asynchronous execution

Description

The **orderBy( )** method sorts the entities in the entity collection or datastore class and returns a new entity collection containing the sorted data.

The values are sorted according to the attribute(s) specified in the attributeList parameter. You can pass from one or more attributes, either in the form of attribute references delimited by commas, or a single string containing attribute names and sort orders delimited by commas. The order in which the attributes are passed determines the sorting priority of the entities.

By default, attributes are sorted in ascending order. You can set the sort order of an attribute by using the sortOrder parameter after the attribute. Pass the string "asc" to sort in ascending order or "desc" for a sort in descending order (include this parameter in the attributeList if you used a string for it).

Sorting is performed by the server.

This method is called asynchronously so you must use the options parameter in order to specify the functions to call when the server returns the resulting entity collection. You can get the sorted entity collection through the **event.entityCollection** (or **event.result**) object in the callback function.

Note: You can use the "order by" keyword directly in the search statement in order to return an entity collection that has already been sorted. For more information, refer to Building a query.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **orderBy( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **orderBy( )** method:

- **pageSize**: *number* (example: **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. By default, the value is 40: if the entity collection contains 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a list.
  You can have this parameter vary for optimization issues, according, for example, to the size of the widgets.

- **autoExpand**: *string containing one or more relation attributes* (example: **autoExpand: "worksFor, livesIn"**)
  By default, the values of relation attributes are not calculated in the entity collections returned by the server, for optimization reasons. Access to these values automatically triggers the corresponding requests. You may want to precalculate these values, for example to be able to display them.

- **progressBar**: *string indicating a progress bar ID* (example: **progressBar: "myBar"**)
  ID referencing an existing widget of the Progress bar type that the server will use to indicate the method's state of progress. For more information, refer to the server side class description.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

The following example performs a sort if the query was successful:

```
var myCollection = ds.Person.query("wages > 50000");
myCollection.orderBy("wages desc", {
    onSuccess: function(event) {  // handle anything special here
            var myCollection = event.entityCollection;
    },
    onError: function(event) {// handle sort errors here
    }
});
```

query( )

EntityCollection **query**(String *queryString* [, Object *options*])

Parameter	Type	Description
queryString	String	Search criteria
options	Object	Block of options for asynchronous execution

Returns	EntityCollection	New entity collection made up of entities meeting search criteria specified in the queryString in the case of synchronous execution

Description

The **query( )** method searches for entities meeting the search criteria specified in queryString among all the entities in datastore class or entity collection, and returns a new object of type EntityCollection containing all the entities found.

Using several consecutive **query( )** methods on the entity collections lets you perform searches by successively reducing the scope of the search. If you keep the intermediary entity collections, you can also provide a rollback system without regenerating server requests

This method is called asynchronously, so you must retrieve the resulting entity collection for the search in the callback function set in the options parameter, usually using the **event.entityCollection** property. However, note that in the context of a **query( )** method executed on the client, Wakanda lets you use the entity collection returned by the method even if at first it is in a "non-finalized" state. The reference to this entity collection is valid so you can work with it and use it once the callback function has been successfully called.

Pass a valid search string in queryString. For a detailed description of this parameter, refer to Defining

Queries (Client-side). You can use parameterized queries using placeholders of type **:n**. In this case, the parameter values must be passed in options through the **params** array (see below).

Note: Keep in mind that if a restricting query has been defined on the server, this method will work only with available entities, that is, entities resulting from the restricting query. For more information, please refer to the Programming Restricting Queries section

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **query( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **query( )** method:

- **params**: *arguments array* (example: **'params': [100, 200,300]**)
  Array of arguments to use in the queryString of a "parameterized query". These arguments are used in order for placeholders (see).

- **pageSize**: *number* (example: **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. By default, the value is 40: if the search "finds" 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a list.
  You can have this parameter vary for optimization issues, according, for example, to the size of the widgets.

- **autoExpand**: *string containing one or more relation attributes* (example: **autoExpand: "employer, livesIn"**)
  By default, the values of relation attributes are not calculated in the entity collections returned by the server, for optimization reasons. Only access to these values automatically triggers the corresponding requests. However, you may want to precalculate these values, for example to be able to display them or to cut down on requests to the server. The **autoExpand** option, once applied during the creation of an entity collection, remains valid during the entire life duration of this entity collection, i.e., all access to the data of this entity collection executes an **autoExpand** whenever it is needed. For example, if the entity collection is large, its contents are retrieved by pages of 40 entities each on the client: every time a new page of entities is accessed, an autoExpand is performed.

- **progressBar**:*string indicating a progress bar ID* (example: **progressBar: "myBar"**)
  ID referencing an existing widget of the Progress bar type that the server will use to indicate the method's state of progress. For more information, refer to the server side class description.

- **queryPlan**: *Boolean* (example: **queryPlan: True**)
  In the entity collection, returns or does not return the detailed description of the query just before it is executed, i.e. the planned query. The information recorded includes the type of query (indexed, sequential), the number of entities processed and the breakdown in the case of a complex query. This option is useful during the development phase of the application. It is usually used in conjunction with **queryPath**.

- **queryPath**: *Boolean* (example: **queryPath: False**)
  In the entity collection, returns or does not return the detailed description of the query as it is actually performed. This information is usually identical to that of the **queryPlan**, but may differ if the engine manages to optimize the query. This option is useful during the development phase of the application. For more information about these options, refer to .

- **orderBy**: *a string containing one or several attribute name(s) delimited by commas. To define the sort order, include either 'asc' or 'desc' after the attribute's name*. For example, **orderBy: "name, firstname desc"**.
  Allows you to sort the resulting entity collection. The order in which the attributes are passed determines the sorting priority of the entities. By default, attributes are sorted in ascending order unless

otherwise specified.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

refresh( )

void **refresh**([Object *options*])
ParameterType Description
options ObjectBlock of options for asynchronous execution

Description

The **refresh( )** method refreshes the entity collection on the client. Basically, the method flushes the entity cache of the collection on the client but keeps the collection reference. Any subsequent access to the entities of the collection (for example a call to getEntity( )) will trigger a query to the server.

This method allows you to make sure that entities in the collection on the client have the same values as those stored in the datastore on the server. It is useful for example when entity values are modified by some code on the server and you want them to be available on the client side.

Since this method is called asynchronously, you can retrieve the resulting entity collection in the callback function set in the options parameter using the **event.entityCollection** property.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **refresh( )** method. In this case, they must be passed just before (and outside) the options parameter.

Example

In this example, you want to make sure that an operation applied to the first entity of a collection uses the most recent values:

```
var myColl = sources.company.getEntityCollection();
myColl.refresh({ //access to the collection values on the server
    onSuccess: function(event){
        event.entityCollection.getEntity(0,{ // access to the first entity
            onSuccess:function(event){
                var e = event.entity;
                var ev = e.rate.getValue();
                e.name.setValue(ev*1.05);
                e.save();
            }
        });
    }
});
```

removeAllEntities( )

void **removeAllEntities**([Object *options*])
ParameterType Description
options ObjectBlock of options for asynchronous execution

Description

The **removeAllEntities( )** method removes from the server all the entities referenced in the entity collection. Once the method has been executed, the entity collection on client contains 0 entity.

Since this method is called asynchronously, you can get the resulting empty entity collection in the callback function set in the options parameter using the **event.entityCollection** property.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **removeAllEntities( )** method. In this case, they must be passed just before (and outside) the options parameter.

Example

You want to delete all entities of the datasource collection in a single call. You can write:

```
var myColl = sources.company.getEntityCollection(); //get the datasource current collection
myColl.removeAllEntities({ //delete the entity collection
    onSuccess: function(event){
        ... //put here the code to execute when the collection is emptied
    }
});
```

removeEntity( )

void **removeEntity**(Number *position* [, Object *options*])

Parameter	Type	Description
position	Number	Position in entity collection of entity to remove
options	Object	Block of options for asynchronous execution

Description

The **removeEntity( )** method removes from the collection the entity whose position in is passed to the position parameter, and deletes it on the server. If you do not want to delete the removed entity on the server, use the removeEntityReference( ) method instead.

If the method is executed successfully, a new entity collection is returned on the client. Since this method must be called in asynchronous mode, the collection is retrieved through the **event.entityCollection** property in the callback function specified in the options parameter.

An error is returned if the position value is outside of the collection length or if the entity cannot be deleted on the server, for example if the onRemove event rejected the action.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **removeEntity( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **removeEntity( )** method:

- **pageSize**: *number* (example: **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. By default, the value is 40: if the entity collection contains 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a list.
  You can have this parameter vary for optimization issues, according, for example, to the size of the widgets.

- **autoExpand**: *string containing one or more relation attributes* (example: **autoExpand: "worksFor, livesIn"**)
  By default, the values of relation attributes are not calculated in the entity collections returned by the server, for optimization reasons. Access to these values automatically triggers the corresponding requests. You may want to precalculate these values, for example to be able to display them.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

You want to delete the first entity of the datasource collection. You can write:

```
button4.click = function button4_click (event)
{
    var myColl = sources.company.getEntityCollection(); //get the datasource current collection
    myColl.removeEntity(0,{ //delete the entity
        onSuccess: function(event){
            sources.company.setEntityCollection(event.entityCollection)}
    }); // replace the collection
};
```

The first entity is removed from the datasource and deleted from the server.

removeEntityReference( )

void **removeEntityReference**(Number *position* [, Object *options*])

Parameter	Type	Description
position	Number	Position in entity collection of entity to remove
options	Object	Block of options for asynchronous execution

Description

The **removeEntityReference( )** method removes from the collection the entity whose position in is passed to the position parameter. The designated entity is not deleted from the server. If you want to delete the removed entity also on the server, use the removeEntity( ) method instead.

If the method is executed successfully, a new entity collection is returned on the client. Since this method must be called in asynchronous mode, the collection is retrieved through the **event.entityCollection** property in the callback function specified in the options parameter.

An error is returned if the position value is outside of the collection length.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **removeEntityReference( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **removeEntityReference( )** method:

- **pageSize**: *number* (example: **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. By default, the value is 40: if the entity collection contains 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically when the client accesses the following pages of entities, for instance by scrolling a list.
  You can have this parameter vary for optimization issues, according, for example, to the size of the widgets.

- **autoExpand**: *string containing one or more relation attributes* (example: **autoExpand: "worksFor, livesIn"**)
  By default, the values of relation attributes are not calculated in the entity collections returned by the server, for optimization reasons. Access to these values automatically triggers the corresponding requests. You may want to precalculate these values, for example to be able to display them.

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

Example

You want to remove the last entity of the datasource collection. You can write:

```
button4.click = function button4_click (event)
{
    var myColl = sources.company.getEntityCollection(); //get the datasource current collection
    myColl.removeEntityReference(myColl.length-1,{ //remove the last entity
        onSuccess: function(event){
            sources.company.setEntityCollection(event.entityCollection)
            }
    }); // replace the collection
};
```

Since the entity is not deleted from the server, it will be back in the datasource if the page is refreshed.

toArray( )

void **toArray**(String *attributeList* [, Object *options*])
Parameter Type Description
attributeList String List of attributes to return as an array or "" to return all the attributes
options ObjectBlock of options for asynchronous execution

Description

The **toArray( )** method creates and returns a JavaScript array in which each element is an object containing a set of properties and values corresponding to the attribute names and values in the datastore class. If the datastore class contains relation attributes, the values of these attributes are themselves objects containing sets of properties and values for the related entities.

This method must be applied to a valid entity collection. In a single request, it generates a complete array of values including multiple relation levels.

Pass a string containing a list of attributes delimited by commas in the attributeList parameter. You can pass either:

- a string containing the names or paths of attributes belonging to the datastore class, for example

("*lastName, salary, company,*" and so on). You can pass first-level attributes or relation attributes, for example ("*lastName,firstName, father.firstName, mother.firstName, father.lastName*"). In the resulting array, attributes corresponding to related data are themselves objects containing attributes and values.
    - In the case of a relation attribute for a N->1 relationship, the attribute contains a sub-object, itself consisting of the requested attribute/value pairs.
    - In the case of a relation attribute for a 1->N relationship, the attribute contains a sub-array listing the related entities. In this case, you can limit the number of sub-elements to be fetched by the main element by passing "*RelatedAttribute*: **X**" (where X represents the number of sub-elements to return) in the attributeList parameter. For example, in the case of a Company/Employee relationship, you can pass "Employee:10" so as to retrieve only the first 10 employees of the company.
- an empty string (*""*) or no parameter (): all the datasource's datastore class attributes are returned. If the datastore class contains relation attributes, you automatically retrieve the internal ID of each related entity (primary key + internal *stamp*, see below).

When this function is called from a client, the method automatically performs two operations:

- It adds the internal ID of each entity for each array element. This ID is made with the entity's primary key and internal *stamp*. The array receives this ID as an object named __KEY containing an *ID:value, __STAMP:value* pair.
- It performs all the necessary **autoExpand** operations according to the contents of the attributeList parameter in order to retrieve related data.

Since this method must be called asynchronously, the resulting array is retrieved through the **event.result** property in the callback function defined in the options parameter.

options

For detailed information about this parameter, please refer to the Syntaxes for Callback Functions *section.*

In the options parameter, you pass an object containing the "onSuccess" and (optionally) "onError" callback functions along with any additional properties, depending on the method. Each callback function receives a single parameter, which is the event.

You can also pass the onSuccess and onError functions directly as parameters to the **toArray( )** method. In this case, they must be passed just before (and outside) the options parameter.

The following parameters are also available in the options block for the **toArray( )** method:

- **skip**: *numeric value* (example: **skip: 20**)
  Lets you not return the first *X* entities of the entity collection.

- **top**: *numeric value* (example: **top: 40**)
  Lets you return only the first *X* entities of the entity collection, starting from the first entity or from the one resulting from the value of the *skip* attribute.

  skip and **top** parameters are useful when you want to paginate results.

- **orderBy**: *string* (example: **"lastName, courses.matter"**)
  Indicates the attribute(s) on which to sort the resulting array. If you retrieve the value of relation attributes of the 1->N type, you get sub-arrays that you can sort by passing an attribute path to orderBy. You can use the **asc** or **desc** keyword to specify an ascending or descending order sort (by default, the sort is ascending).

- **userData**: *any valid JavaScript value* (examples: **userData: {myTest: "Data to pass"}, userData: 2012**)
  In the options parameter, you can pass a **userData** property containing data you'd like to retrieve later. You simply pass the data to this property and retrieve it from inside the callback function in the **event.userData** object.
  The **userData** property can contain any valid JavaScript value (scalar value, object, array, etc.). You can use the **userData** member to pass any static or dynamic data (for example, references to widgets, counters, etc.) that you want to use again in the callback function.

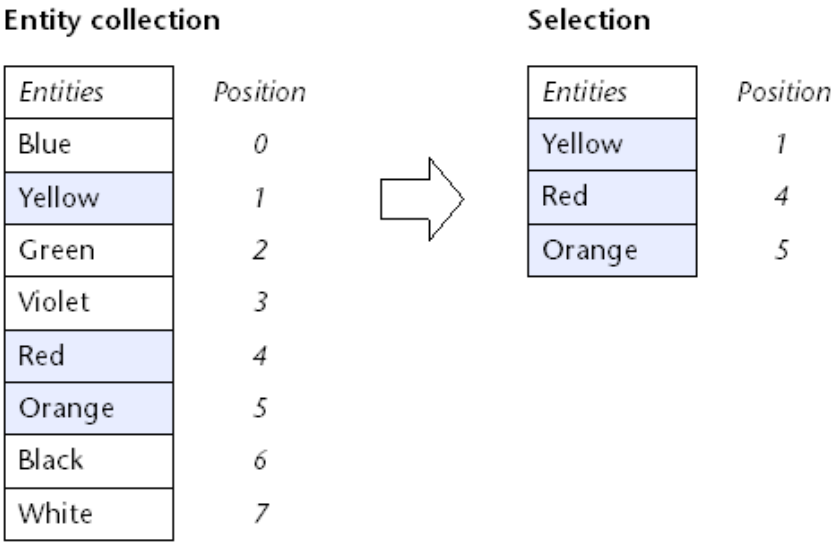
Example

We want to load a container with all cities in the datastore. This will need only two calls to the server.

```
ds.City.all({orderBy:"name", onSuccess:function(event)
{
    event.entityCollection.toArray("name, ID", {
        onSuccess: function(ev)
        {
            var arr = ev.result;
            var myHTML = '';
            arr.forEach(function(elem) {
                myHTML += '<p>' + escapeHTML(elem.name) + '</p>';
            });
            ('#container3').html(myHTML);
        }
    });
}});
```

Selection

A Selection (also named an *entity selection*) is a subset of an entity collection. It references one or several entities in the entity collection by their original position (and NOT their IDs):



There can be only one selection attached to a collection at a time.

This object is usually generated by a list-oriented widget (grid or matrix) when the user performs a continuous or discontinuous selection of entities by using **Shift-click** or **Ctrl/Command-click** in the widget. Once generated, it is automatically maintained by Wakanda: if the entity collection is reordered, the initial referenced entities in the selection are still referenced but with their new position.

You can get a new Selection object from the getSelection( ) method (Datasource class).

countSelected( )

Number **countSelected**( )

Returns NumberNumber of entities in the selection

Description

The **countSelected( )** method returns the number of entities in the Selection object. In other words, this method returns the number of selected entities in the entity collection.

getSelectedRows( )

Array **getSelectedRows**( )

Returns ArrayPositions of selected entities

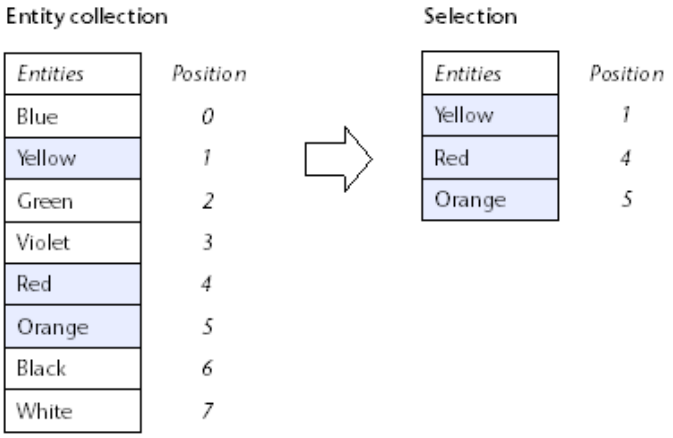
Description

The **getSelectedRows( )** method returns an array of the selected entity positions in the parent entity collection.

If Selection is in single selection mode, the array contains only one element.

Example

Based on the following situation:




```
var sel = sources.colors.getSelection(); // get the current selection
var selArray = sel.getSelectedRows ();  // selArray = [ 1 , 4 , 5 ]
```

isMultipleMode( )

Boolean **isMultipleMode**( )

Returns BooleanTrue if the selection is in multiple mode, False otherwise

Description

The **isMultipleMode( )** method returns True if the Selection works in multiple selection mode, otherwise False. By default, all widgets and, by consequent, all selections work in single selection mode. The selection mode for a widget can be defined in the GUI Designer. At the moment, the only widget that can have "Multiple" as its selection mode is the Grid widget.

isSelected( )

Boolean **isSelected**(Number *pos*)
ParameterType Description
pos Number Entity position in the parent entity collection

Returns Booleantrue if the entity is selected in the collection, false otherwise

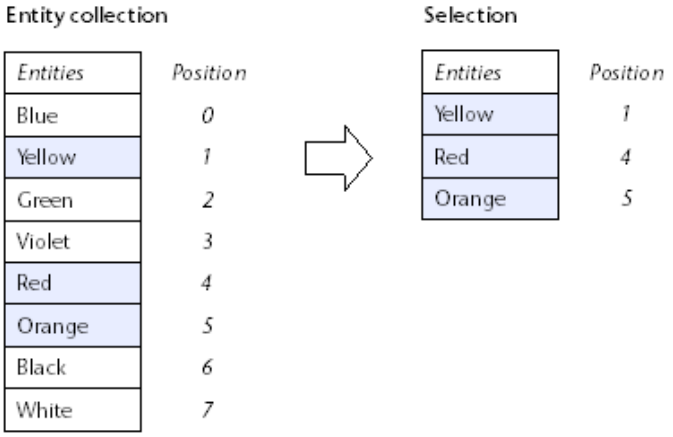
Description

The **isSelected( )** method returns **true** if the entity whose position you passed in pos is currently selected in the entity collection. In other words, the method returns **true** if the designated entity belongs to the current Selection attached to the entity collection.

The value you pass in pos is a position in the parent entity collection (starting from 0).

Example

Based on the following context:




```
var sel = sources.colors.getSelection(); // get the current selection
var isSel = sel.isSelected(4) // returns true - "Red" is selected
var isSel2 = sel.isSelected(0) // returns false - "Blue" is not selected
```

isSingleMode( )

Boolean **isSingleMode**( )

Returns BooleanTrue if the selection is in single selection mode, False otherwise

Description

The **isSingleMode( )** method returns True if the Selection works in single selection mode, otherwise False. By default, all widgets (except the Grid that works in multiple and single selection mode) work in single selection mode. The selection mode for a widget can be defined in the GUI Designer.

select( )

void **select**(Number *pos* [, Boolean *addToSel*])

ParameterType		Description
pos	Number	Position of entity to select
addToSel	Boolean	True = add to selection, False = replace selection

Description

The **select( )** method adds to Selection the entity whose position was passed in pos. In other words, the method selects the designated entity in the parent entity collection.

When Selection is in multiple selection mode, by default the designated entity replaces the current Selection. After the method is called, only the entity at the pos position is selected in the parent entity collection. If you want the entity to be added to the existing Selection, pass True to the addToSel parameter. In this case, if the entity was already included in the Selection, the method does nothing (the entity is not removed from the Selection).

Example

In this example, we select the last entity in the datasource's current selection:

```
var theSel = sources.item.getSelection(); // get the current selection
var theLast = sources.item.length; // size of the entity collection
theSel.select(theLast-1); // select the last entity (-1 because pos starts at 0)
```

selectRange( )

void **selectRange**(Number *startPos* , Number *endPos* [, Boolean *addToSel*])

Parameter	Type	Description
startPos	Number	Position of the first entity to select
endPos	Number	Position of the last entity to select
addToSel	Boolean	true = add to selection, false = replace selection

Description

The **selectRange( )** method adds to Selection the entities whose range is defined by startPos and endPos. These values refer to relative positions in the parent entity collection.

This method must be used with a Selection in multiple selection mode. By default, the designated entity range replaces the current Selection. If you want the entities to be added to the existing Selection, pass True to the addToSel parameter. In this case, if an entity was already included in Selection, the method does nothing (the entity is not removed from the Selection).

Example

We want to add the 10 first entities to the datasource selection:

```
var theSel = sources.item.getSelection(); // gets the current selection
theSel.selectRange(0 , 9 , true); // selects the 10 first entities
```

setSelectedRows( )

void **setSelectedRows**(Array *rowsToSelect*)

Parameter	Type	Description
rowsToSelect	Array	Array of positions of entities to select

Description

The **setSelectedRows( )** method allows you to set the selected entities in the parent entity collection thus modifying Selection.

Pass in rowsToSelect an array containing numbers representing the positions of entities to select in the parent entity collection.

If the Selection is in single selection mode, only the first entity position in rowsToSelect is selected.

Example

Based on the following situation:

Entity collection

Entities	*Position*
Blue	*0*
Yellow	*1*
Green	*2*
Violet	*3*
Red	*4*
Orange	*5*
Black	*6*
White	*7*

Selection

Entities	*Position*
Yellow	*1*
Red	*4*
Orange	*5*

```
var col = sources.color.getSelection(); //get the associated selection
col.setSelectedRows([0,2,6]); //set the selection
```

After the code is executed, the situation is:

Entity collection

Entities	*Position*
Blue	*0*
Yellow	*1*
Green	*2*
Violet	*3*
Red	*4*
Orange	*5*
Black	*6*
White	*7*

Selection

Entities	*Position*
Blue	*0*
Green	*2*
Black	*6*