Dataprovider

Introduction

Welcome to the Wakanda Dataprovider API. The Dataprovider API is part of the Wakanda Ajax 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**). Intended to simplify data access, it is in charge of:

- sending the necessary REST requests to the Wakanda server according to the needs of the client-side application, in particular for widget and datasource needs.
- 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 the datastore classes, entity collections and entities from the server side.

Suppose for example that you have an interface containing a datagrid type 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 of them. As the user scrolls the list of entities, the Dataprovider triggers the necessary requests as the need arises. 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.

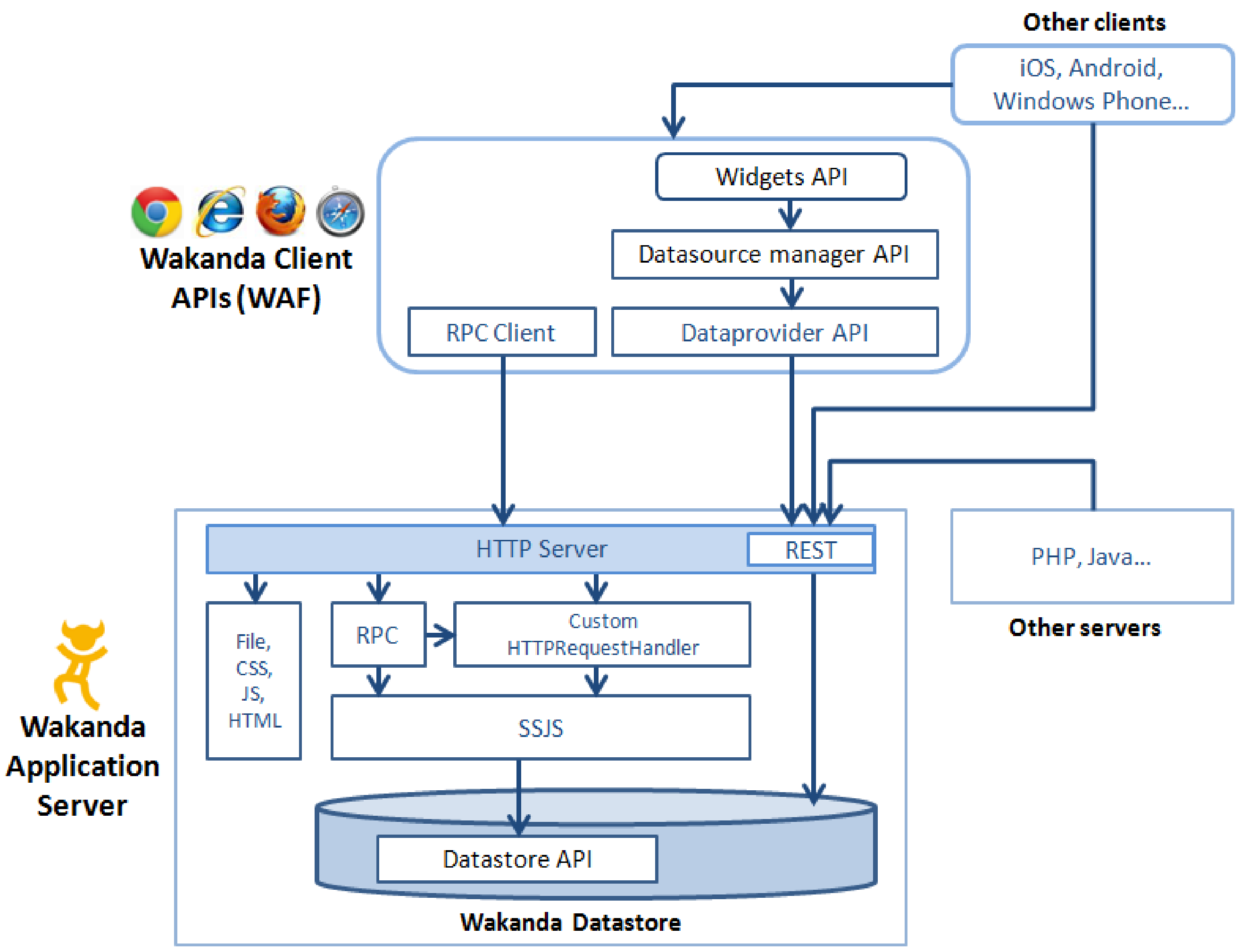
Note that there is one significant difference: you usually need to call methods asynchronously from the client in order to avoid freezing the browser. You can notice this difference in the syntax of the methods, which always work with blocks containing callback functions that are called when the server 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 Ajax Framework (WAF) API. You use the methods of the Dataprovider API to manage the entity collections and the entities of the current datastore 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 the client, according to the business logic of your application and your model-related needs. More specifically:

- When you need to control by programming the operation of datasources, 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 nor enabling the automatic datasource mechanisms, use the methods of the *Dataprovider* (client API);
- When you need to perform processing directly on server-side datastore models and data, use the *Datastore API* (server API).

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

Generally speaking, you must strive 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.

Access to datastore objects using 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.

Access to 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 of the 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
```

Access to 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 datas
```

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**.

Access to datastore class attributes

On the client side, 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 cla
  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 the *DatastoreClassAttribute* type 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. On entities, the attributes themselves can be directly accessed as properties (see **Using entity attributes**).

For more information about the properties of *DatastoreClassAttribute* type objects, 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 the datastore classes and then send the result to the client or when you launch 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 **Datastore Class Events**
- 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	Advar
JSON-RPC Services	Installation of function files on the server. Initialization of classes, called	Optional: Synchronous or Asynchronous	Any type of use	Requi WAF a the RI client

	using client-side code			
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.	Manag power library the cl Possib functi restar
Datastore class method	Direct call using JavaScript (client or server)	Synchronous or Asynchronous (depends on the syntax)	Any operation on the datastore class	Direct data i class.
Datastore class event	Automatic on event call	Not applicable (no call on the client-side)	Applying or verifying datastore rules	Autom execu to the datast

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 Ajax 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 col
    }
    $("#display").html("selection : "+vcount);
        // we display the size of the entity collection in the container whose ID is "dis
});
```

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 col
        $("#display").html("selection : "+vcount);
            // we display the size of the entity collection in the container whose ID is
    }
});
```

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 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. 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,]
            }
            [{userData}]
         );
```

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},]
            [{userData}]
         );
```

- *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.

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 (*Dev branch only*)
catalog
delay
delayInfo
filterSet
first
forceReload (*Dev branch only*)
generateRESTRequestOnly (*Dev branch only*)
limit
method
onError
onSuccess
orderby
orderBy (*Dev branch only*)
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

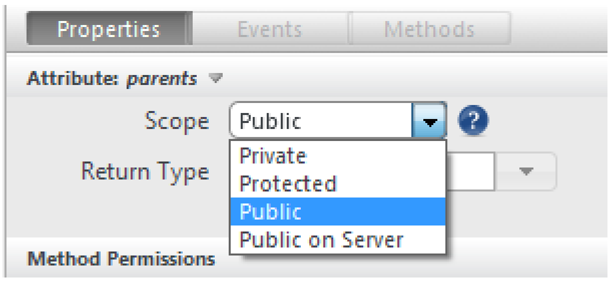
You can call any public datastore class method from the Dataprovider API. For more information about creating and specifying datastore class methods, refer to the **Datastore Class Methods** section in the *Wakanda Studio Reference Guide*.

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.

Handling Binary Return

Note: This feature is only available in the Development Branch of Wakanda.

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, from a datasource, a datastore class method named "getPict" which returns an image as binary data, you could write (synchronous call):

```
var urlPict = sources.person.getPict({generateRESTRequestOnly:true, callWithGet:true}, "S
```

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

Example

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

- On the server side, 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 side, 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, auto
  // myFunction and failure receive the resulting entity collection
  ```

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

  ```
  var myset = ds.Employee.callMethod({method:"moreThanAverage", pageSize: 60, onSucces
  // Exactly the same as the previous call
  ```

- On the client side, 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 que
      var nbEmp = myset.length; // we retrieve the number of entities found
      $("#display").html(nbEmp); // display in a container
  }
  ```

Error management

On the client, any errors generated during execution are usually retrieved in the callback functions specified during asynchronous calls (**onError** or event.error function in the case of a single function).

Errors are supplied as a stack that is an array. In the event of an error, the server returns the values in question.

- error.line = line number of the script that caused the error
- error.message = message of highest-level error.
- error.messages = array containing stack of error messages with element 0 containing the highest-level message
- error.sourceId = error code

Errors are usually characterized by a signature, a code and a message.

- Signatures: DBMGR = Wakanda database engine errors.
  - 0 to 1499 = errors related to the database
  - 1500 to 1799 = errors related to datastore classes
  - 1800 and 2099 = errors related to REST requests on datastore classes

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 attribute is the datastore class attribute on which you want to do the query. For example, "employee.name". This parameter can also be any valid attribute path such as "father.father.name".
- *comparator*: the comparator is the comparison that is made between *attribute* and *value*. The comparator is one of the symbols or keywords listed in the .
- *value*: the value is 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 for 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.

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'
```

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

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 nth 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
    }, 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
```

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 strictly equal data, ignores the wildcard (*)
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

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 refer
         { allowJavascript: true });   // to allow javascript execution
   return myCol;
}
model.MyClass.methods.mySearch.scope = "public"; // do not forget to make the method publ
```

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

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

DataClass

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

Working with datastore classes on the client

Access to datastore class methods

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

all( )

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

Parameter	**Type**	**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*])

Parameter	**Type**	**Description**
options	Object	Block 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 to which it was applied. 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 model 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.
The entities are returned in the default order, which is the order in which they were created.

Example

This example replaces the current entity collection of a datasource 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*])

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, of the call to the method 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 can create generic code: since the method name is passed as a string, it is easy to use variables.

Example

This example calls the datastore class method of the "class" type named *teaching* and passes it the name of a subject as parameter:

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

Example

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
}});
```

We execute the same method directly just like 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 datastore class to which it is applied. 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. This is the case, for instance, 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 type request. Any new request causes direct retrieval of entities on the server and updates the cache with the result.

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 the *attribute* attribute for the entity collection or datastore class to which it is applied. By default, the command takes all the entities of the datastore class or entity collection into account for calculating 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 taken into account.

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 generic **event.result** property.

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 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.

Example

In the following example, we fill a pop-up in an HTML page after a button click, using the distinct values stored in the 'country' attribute of 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 pop up with array va
               {
                    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	Object	Block 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].

Example

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

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

        $("#display").html(html); // display in the container with the "display" ID (JQue
    }
});
```

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 the *attributeName* parameter as a string. It is mainly useful when implementing 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 **Access to datastore class attributes** paragraph.

The *DatastoreClassAttribute* type object returned contains numerous properties describing the attribute as it is specified in the datastore model (using the Datastore Model Designer). The number of properties varies according to the attribute type.

These properties can be useful for generic programming. More particularly, the available properties are:

- *kind*: nature 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 on the client side because calls and use of these attributes are identical and do not depend on this property.
- *name*: name of attribute
- *type*: type of 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 ou 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

You want to find out the value of the 'kind' property for an attribute:

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

getAttributes( )

Object **getAttributes**( )

Returns	Object	List of datastore class attributes

Description

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

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

Example

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

```
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	Number	Number of entities to keep in cache

Description

The **getCacheSize( )** method returns the current size of the entity cache on the client for the datastore class to which it is applied.

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

By default, the size is 300 entities. You can increase this value according to the needs of your application 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( )

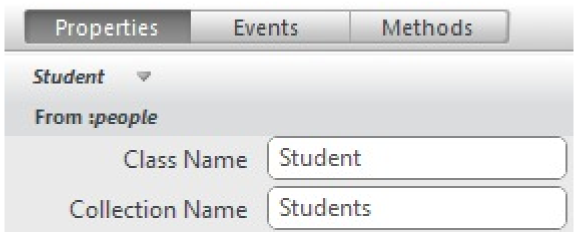
String **getCollectionName**( )

Returns	String	Name of an entity collection of the datastore class

Description

The **getCollectionName( )** method returns the name of an entity collection of 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 in the

Datastore class designer:



Class Name: Student
Collection Name: Students

You can just use the plural form of name of the datastore class, or use the suffix _collection, or again 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	Datastore	Datastore of class

Description

The **getDataStore( )** method returns the datastore of the datastore class to which it is applied.

By default, the current datastore is placed in the **ds** object (see **Access to 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 the *keyValue* parameter within the datastore class to which it is applied.

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.

Example

In this simplified example, we retrieve an entity by the value of its primary key 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
          {   myEnt=event.entity; // get the entity
              $("#display").html("result = "+myEnt.lastName.getValue());
          }});
    };
```

getName( )

String **getName**( )

Returns	String	Name of the datastore class

Description

The **getName( )** method returns, as a string, the name of the datastore class to which it is applied. This method is mainly useful for setting up generic code, for example, when you want to pass the name of the datastore class as a parameter.

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 the *EntityCollection* type attached to the datastore class to which it is applied. 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. Pass in *colRef* an entity collection server reference 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/or "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 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.

newEntity( )

Entity **newEntity**( )

Returns	Entity	New entity created in memory

Description

The **newEntity( )** creates a new entity in the datastore class to which it is applied 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 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
    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 of the datastore class or entity collection to which it is applied, and returns a new object of the *EntityCollection* type containing all the entities that are 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 is called successfully.

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 :n type; in this case, the values of the parameters must be passed in *options* through the **params** array (see below).

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 r
        {
            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 nota
        }
    });
};
```

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 el
            {
                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(e
                // search for an employee and retrieve his/her employer
    {
        var myset = event.entityCollection; // we retrieve the resulting entity collectic
        myset.getEntity(0, { onSuccess:function(event) // we load the first entity of the
                    // it is possible that this does not generate any server request beca
                    // has already been sent back by the server but using this syntax gua
            {
                var myEntity = event.entity;
                var html = ""; // initialize the display
                html += "Last Name: "+myEntity.lastName.getValue()+"<br/>"; // direct acc
                html += "First Name: "+myEntity.firstName.getValue()+"<br/>";
                myEntity.employer.getValue( { onSuccess: function(event) // access to a r
                {   // it is therefore necessary to nest an asynchronous call
                    // because of the autoExpand option, this access does not trigger a r
                    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 to which it is applied.

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

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

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

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 datastore class of the default 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 **Access to a datastore**). This reference is used in your client-side code as a shortcut to **WAF.ds** to reference the current datastore and access his datastore classes.

Example

Each datastore 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 datas
```

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* type object whose name is the same as the string passed in the *className* parameter in the current datastore. This method is mainly useful in the context of generic code.

Example

You have 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	Object	Datastore classes of current datastore

Description

The **getDataClasses( )** method returns all the datastore classes specified in the model of the current datastore, as well as 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 makes it easier to count, enumerate 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*)

Parameter	Type	Description
options	Object	Block 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 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 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.

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 the *Entity* type contain in the form of properties the different attributes that make up the datastore class.

However, there is one significant difference with respect to the server (see **Working with Entities**) concerning assigning and reading of values for attributes of the entity. On the server, you access these values through the 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 side (see **Using Server Datasources**), you can work on the Dataprovider with several entities in the same entity collection simultaneously. There is no notion of a current entity on the Dataprovider.

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) and pass the datastore class to it as a parameter:

```
// Create a new entity in the MyClass datastore class
var newEntity = ds.MyClass.newEntity();
// strictly 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 the *Entity* type, you can access datastore class methods of the "entity" type, specified in the Datastore class editor -- provided their scope is "Public". You cannot call Datastore class methods of the "entityCollection" or "class" type on these objects.

The call can be synchronous or asynchronous (see **Access to datastore objects using 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, of the call to the method 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 can create generic code: since the method name is passed as a string, it is easy to use variables.

getDataClass( )

DatastoreClass **getDataClass**( )

Returns	DatastoreClass	Datastore class to which the entity belongs

Description

The **getDataClass( )** method returns the *DatastoreClass* where the entity to which this method is applied belongs. This method is usually useful in the context of generic code.

getKey( )

String | Number **getKey**( )

Returns	Number, String	Primary key value of entity

Description

The **getKey( )** method returns the value of the primary key of the entity to which it is applied.

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

getStamp( )

Number **getStamp**( )

Returns	Number	Current value of internal stamp of the entity

Description

The **getStamp( )** method returns the current value, on the client, of the internal *stamp* of the entity to which it is applied.

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

isNew( )

Boolean **isNew**( )

Returns	Boolean	True 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, the method returns False.

You can use this method to call specific code when an entity is created, for example.

isTouched( )

Boolean **isTouched**( )

Returns	Boolean	True if entity has been modified; otherwise, False

Description

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

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

remove( )

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

Parameter	Type	Description
options	Object	Block of options for asynchronous execution

Description

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

The **remove( )** method deletes from the datastore on the server the entity to which it is applied. Executing this method triggers a call to the **onRemove** event on the server if it is specified for the datastore class or one of the datastore class attributes of the entity. Deletion can be refused and an error returned by this event (see).

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

If an error occurs, for example if the entity has already been deleted, this error is returned in the *error* parameter of the "onError" callback function (or in **event.error** if you only use a single function). You can access the error stack through the *error* object, for example **error.message**. 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/or "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 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.

save( )

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

Parameter	Type	Description
options	Object	Block of options for asynchronous execution

Description

The **save( )** method saves, in the datastore, the modifications made to the entity to which it is applied. On the server, any code associated with the **onValidate** and **onSave** events of the datastore class is executed. You must call this method after creating or modifying each entity, if you want to save the changes.

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 attributes of the entity as they have been saved on the server, i.e. after application of the business rules defined on the server. This way you can retrieve calculated values. Any values that are modified and not null values are returned.

If an error occurs, for example if the entity was modified by another user between when it is loaded and when it is saved, the error is returned in the *error* parameter of the "onError" callback function (or in **event.error** if you only use a single function). You can access the error stack through the *error* object, for example **error.message**. 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, for example, so that you can display them (see **Locking Entities** inthe Datastore API manual).

Example

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

```
createBlankButton.click = function (event)
```

```
    {
        var e = ds.People.newEntity(); // creating of entity with attributes set to null
        e.lastName.setValue("Last name"); // entry of 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 c
                html += " fullName : "+event.entity.fullName.getValue()+"<br/>"; // conta
                $("#display").html(html); // display of values in the container with the
            },
            onError: function(error) // save has failed
            {
                var mess = error.message ;
                $("#display").html("error :"+ mess +"<br/>");
            }
            });
    };
```

touch( )

void **touch**( )

Description

The **touch( )** method indicates that the entity or one of its attributes to which this method is applied 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 detection of modifications is managed dynamically by the Dataprovider, 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 all of the entity).

Entity attribute

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

Using entity attributes

Access to 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 use the **getValue( )** and **setValue( )** methods on the attribute. For example:

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

Note: Do not confuse entity attributes with those of the datastore class (objects of the *Attribute* type specified in the datastore model of the application), described in **Access to datastore objects using the Dataprovider**.

getRelatedClass( )

DatastoreClass **getRelatedClass**( )

Returns	DatastoreClass	Datastore class of relation attribute

Description

The **getRelatedClass( )** method returns the datastore class related to the relation attribute to which it is applied.

This method is useful for creating generic code. For optimization reasons, when you access the attributes of a datastore class either directly or through the **getAttributeByName( )** method, Wakanda does not automatically retrieve the related datastore classes (the *relatedModel* property is null for relation attributes). If you need to access them, you must explicitly request their calculation using the **getRelatedClass( )** method.

getValue( )

Mixed **getValue**([Object *options*])

Parameter	**Type**	**Description**
options	Object	Block of options for asynchronous execution
Returns	Mixed	Value of entity attribute

Description

The **getValue( )** method gets the value of the attribute of the entity to which it is applied. You need to call this method in the Dataprovider API so as to ensure synchronization of access to values between the server and client.

- When you access a storage attribute, i.e. a simple type, there is no request sent to the server so you can retrieve the value of the attribute directly. The syntax to use is in the form:

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

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

  ```
  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 be sent to the server, which optimizes application operation. However, it is necessary to use the asynchronous syntax in this case.

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

Example

This example illustrates the principle of nesting asynchronous calls. It is the script of a button performing a query on the server and then accessing the values of the entity attributes, including relation attributes. The result is placed in a container displaying the values.
Note that you can only obtain a valid result by accessing it 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", // we precalculate relation attributes
        function(event) // callback function
        {
            event.entityCollection.getEntity(5, { // we access the 5th entity, in asynchr
                function(event) // callback function
                {
                    var myEntity = event.entity; // we retrieve the entity
                    var html = "";
                    html += "ID : "+myEntity.ID.getValue() + "<br/>"; // access to storag
                    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 rela
                    { // 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	Boolean	True if entity has been modified; otherwise, False

Description

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

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

setValue( )

void **setValue**(Mixed *value*)

Parameter	**Type**	**Description**
value	Mixed	New value of entity attribute

Description

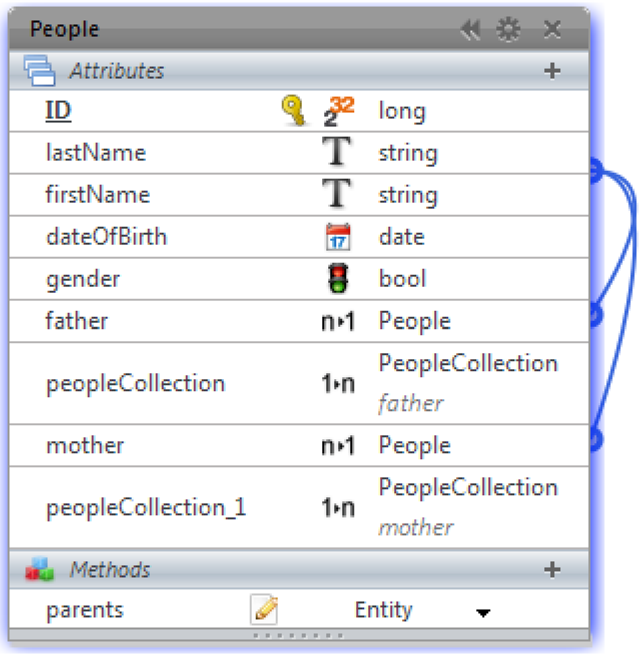
The **setValue( )** method modifies the attribute value of the entity to which it is applied. Its syntax is in the form:

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

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

Example

Given the following datastore class, storing people and their relatives:



People

Attributes

ID	2^{32}	long
lastName	T	string
firstName	T	string
dateOfBirth		date
gender		bool
father	n▸1	People
peopleCollection	1▸n	PeopleCollection *father*
mother	n▸1	People
peopleCollection_1	1▸n	PeopleCollection *mother*

Methods

parents	Entity

The following code, associated with a button, creates an entity and associates the entity of its father with it:

```
bindEntities.click = function (event)
{
    ds.People.find('lastName = "Lawson"', { onSuccess:function(event) // we look for
    {
        if (event.entity != null) // an entity is found
        {
            var theFather = event.entity; // we store the father in a variable
            var e = new WAF.Entity(ds.People); // we create a new blank entity
            e.name.setValue("Lawson"); // assigning of values
            e.firstname.setValue("Peter");
            e.father.setValue(theFather); // assigning relation attribute and passing
            e.save(); // call of save in synchronous mode
        }
    } });
};
```

touch( )

void **touch**( )

Description

The **touch( )** method indicates that the entity or one of its attributes to which this method is applied 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 detection of modifications is managed dynamically by the Dataprovider, 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 all of 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 using the Datastore API (see **Working with Entity Collections on the Server**).

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

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

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

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 = myEntSet[5]; // Server syntax, not possible on client
var myEntity6 = myEntSet.getEntity(5); // Same syntax on client
```

Note that in keeping with the principle of asynchronous calls on the client, the **getEntity( )** method must usually be executed with a callback function (see example of the **getEntity( )** method).

Access to entity collection methods

On objects of the *EntityCollection* type, you can access datastore class methods of the "entityCollection" type, specified in the Datastore class editor -- provided their scope is "Public". You cannot call datastore class methods of the "entity" or "class" type 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
```

length

Description

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

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

add( )

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

Parameter	**Type**	**Description**
entity	Entity	Entity to add

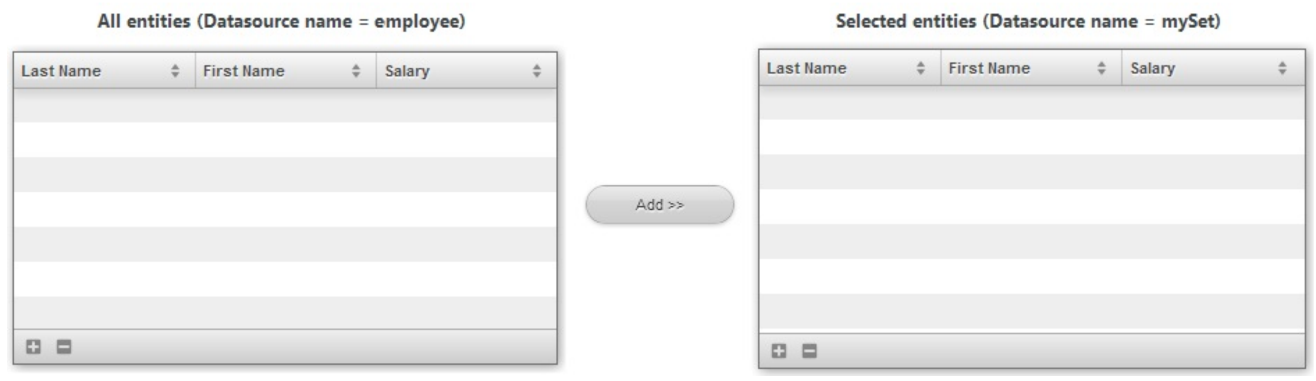
Description

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

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

Example

In this example, we are filling a datasource with entities selected from another datasource. In the GUI Designer, the Interface page appears as shown below:


All entities (Datasource name = employee)
Last Name
First Name
Salary
Add >>
Selected entities (Datasource name = mySet)
Last Name
First Name
Salary

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

```
button1.click = function (event)
{
    var emp = sources.employee.getCurrentElement(); // retrieve the current element t
    var destSet = sources.mySet.getEntityCollection(); // retrieve the datasource's c
    if (emp != null) // if there is a selected entity
    {
        destSet.add(emp); // add the left entity to the entity collection
        source.mySet.setEntityCollection(destSet); // reapply the entity collection t
            // this 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/or "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:

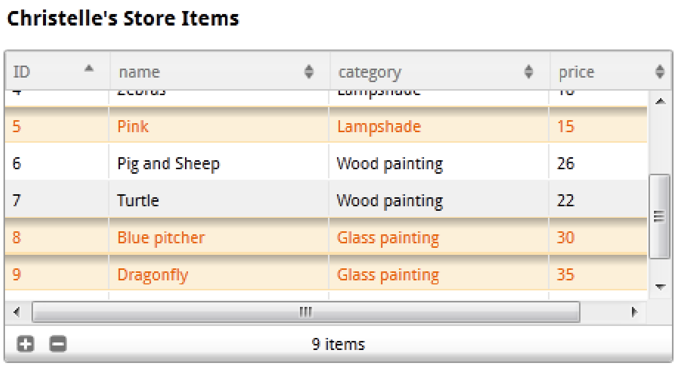
- **pageSize**: *number* (example: **pageSize: 60**)
  Number of entities per "page" returned by the server to the browser. By default, the value is 40: if the entity collection contains 200 entities, the server only returns the first 40 (for optimization reasons). Additional requests are triggered automatically 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 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.

Example

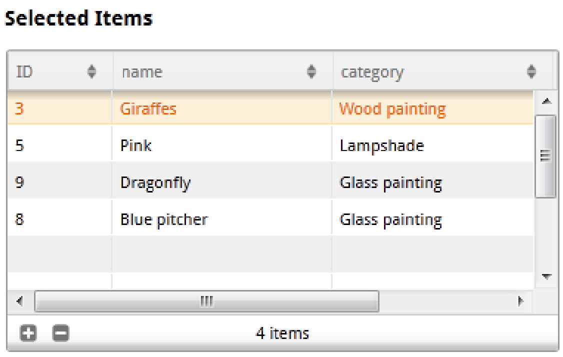
In a page from a hand-made items store, we want to display a list of items and a list of selected items. We add lists as grids, based on two different Datastore class datasources, "item" and "item2". The main list (at the left side) has the **Multiple** selection mode. At runtime, each time an item is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



Christelle's Store Items

ID	name	category	price
4	Zebras	Lampshade	10
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 ri
     {
         var collec = event.entityCollection; //gets the new entity collection from t
         sources.item2.setEntityCollection(collec); //assigns the collection to the d
            //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 mair
     col.buildFromSelection(sel, { onSuccess: buildsel }); // returns a new collectic
 };
```

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, of the call to the method 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 can create generic code: since the method name is passed as a string, it is easy to use variables.

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 the *attribute* attribute for the entity collection or datastore class to which it is applied. By default, the command takes all the entities of the datastore class or entity collection into account for calculating 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 taken into account.

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 generic **event.result** property.

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 of the entity whose primary key value is passed in the *key* parameter, in the entity collection to which it is applied.

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, the method returns -1.

forEach( )

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

Parameter	Type	Description
options	Object	Block 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 of the entity collection to which it is applied, in ascending order.

If the function modifies the entity, you must call the **save( )** method for each entity if you want 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 entity being processed within the entity collection.

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 t
                        // 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 coll
                 // 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* where the entity collection to which this method is applied belongs. It is mainly used in the context of 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 is passed in *position* parameter from the entity collection to which it is applied.

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.

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 >10
    'onSuccess':function(event) // first callback because query is asynchronous
    {
        event.entityCollection.getEntity(5, { // we request that the 5th entity of the en
            'onSuccess':function(event) // second callback because the getEntity method i
            {
                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
                html += "Last Name: "+myEntity.lastName.getValue() + "<br/>"; // with the
                html += "First Name: "+myEntity.firstName.getValue() + "<br/>";
                html += "wages: "+myEntity.wages.getValue() + "<br/>";

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

getReference( )

Object **getReference**( )

Returns	Object	Reference to the entity collection on the server

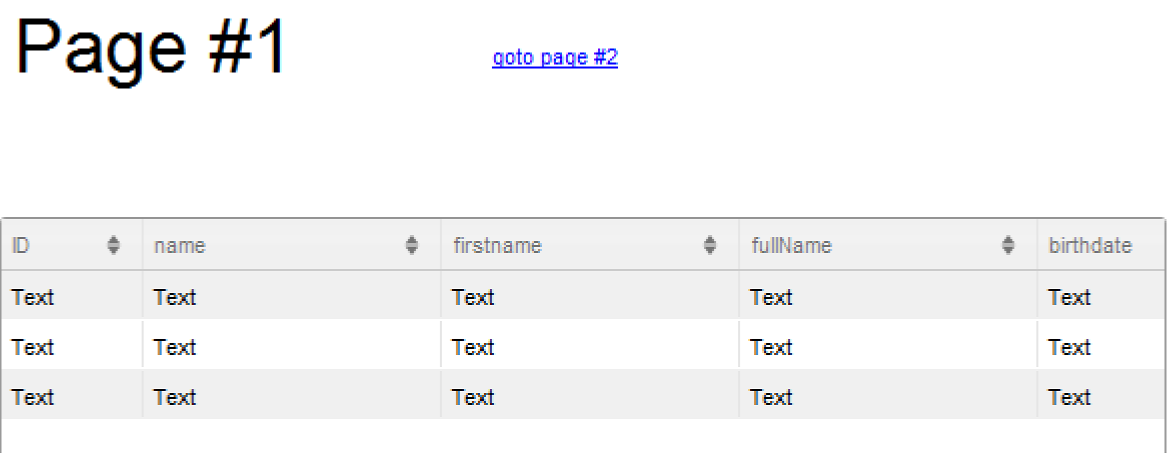
Description

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 with the **newCollection( )** method.

Example

In our application, 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 their navigation.



Page #1

goto page #2

ID	name	firstname	fullName	birthdate
Text	Text	Text	Text	Text
Text	Text	Text	Text	Text
Text	Text	Text	Text	Text

- On each page, the navigation link (a rich 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();  // get the r
    // the reference is an object, we must convert it as a string to be able to
    var savedRef = JSON.stringify(ref);
    sessionStorage.myCollectionSavedRef = savedRef; // stores locally the collec
};
```

- 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 collect
    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 t
        ds.Person.newCollection(ref, function(event) // asynchronous call for th
        {
            var newcol = event.entityCollection; // get the collection saved on
            sources.person.setEntityCollection(newcol); // assigns the collectio
        });
    }
};
```

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 of the entity collection or datastore class to which it is applied and returns a new entity collection containing sorted data.

The values are sorted according to the attribute(s) specified in the *attributeList* parameter. You can pass from 1 to X attributes, either in the form of attribute references separated by commas, or a single string containing attribute names and directions, separated 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 perform an ascending sort or "desc" for a descending one (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 is already sorted. For more information, refer to Building a query.

Example

The following example performs the order by after 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 of the datastore class or entity collection to which it is applied, and returns a new object of the *EntityCollection* type containing all the entities that are 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 is called successfully.

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 :n type; in this case, the values of the parameters must be passed in *options* through the **params** array (see below).

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	Object	Block of options for asynchronous execution

Description

The **toArray( )** method creates and returns a JavaScript array where each element is an object containing a set of properties and values corresponding to the attribute names and values of the datastore class to which the method is applied. 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 X relation levels.

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

- a string containing the names or paths of attributes belonging to the datastore class to which the method is applied, 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 (): in this case, all the datastore class attributes of the datasource are returned. If the datastore class contains related attributes, you automatically retrieve for each of them the internal ID of each related entity (primary key + internal *stamp*, see below).

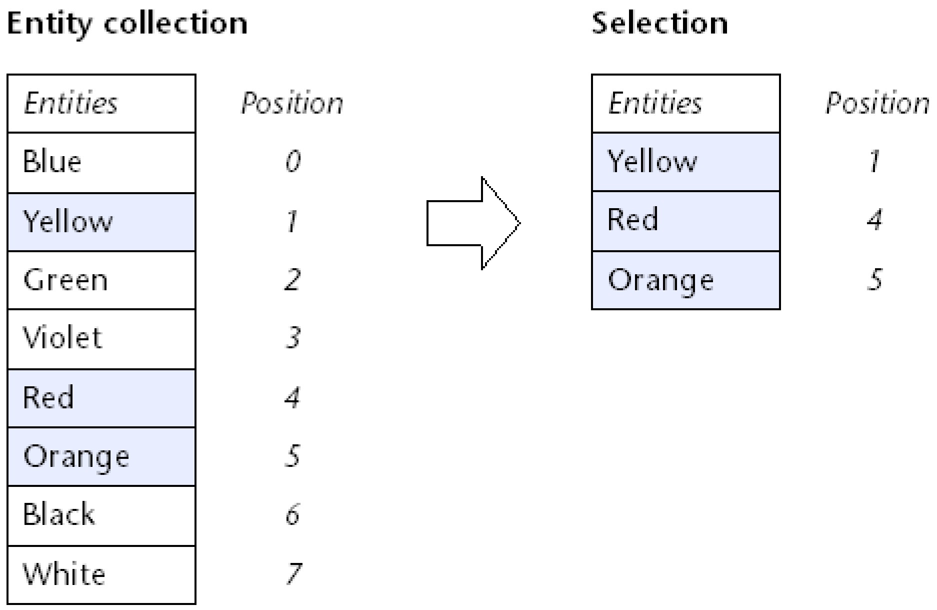
When it is called from a client machine, the method automatically performs two operations:

- It adds, for each array element, the internal ID of each entity. This ID is made with the primary key and the internal *stamp* of the entity. 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 fetch the 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.

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	Number	Number 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 to which the selection is attached.

forEach( )

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

Parameter	**Type**	**Description**
options	Object	Block of options for asynchronous execution

Description

Note (DP): This method is currently not yet implemented.

The **forEach( )** method executes a function on each entity of the *Selection* to which it is applied, in ascending order.

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

In the functions called back by the server, you can retrieve the selection, the entity collection and the entity being processed through the following properties:

- **event.entityCollection**: current entity collection
- **event.selection**: current selection
- **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 entity being processed within the entity collection.

getSelectedRows( )

Array **getSelectedRows**( )

Returns	Array	Positions of selected entities

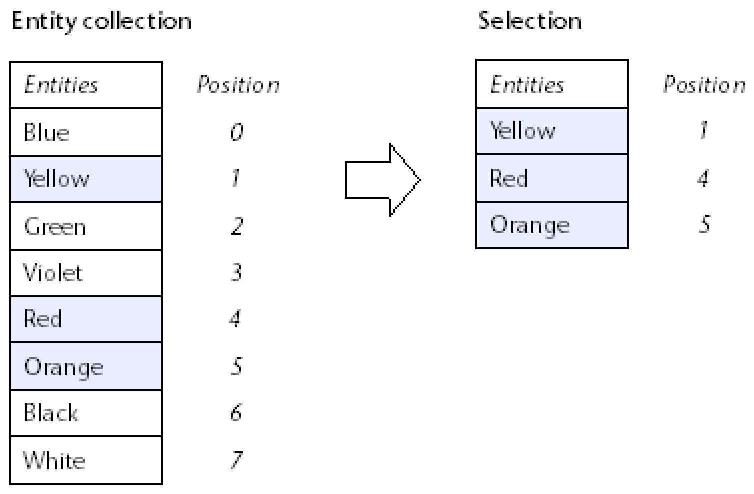
Description

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

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

Example

Based on the following situation:



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

isMultipleMode( )

Boolean **isMultipleMode**( )

Returns	Boolean	true if the selection is in multiple mode, false otherwise

Description

The **isMultipleMode( )** method returns **true** if the *Selection* works in multiple selection mode and **false** otherwise. 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 of Wakanda studio.

isSelected( )

Boolean **isSelected**(Number *pos*)

Parameter	**Type**	**Description**
pos	Number	Entity position in the parent entity collection
Returns	Boolean	true if the entity is selected in the collection, false otherwise

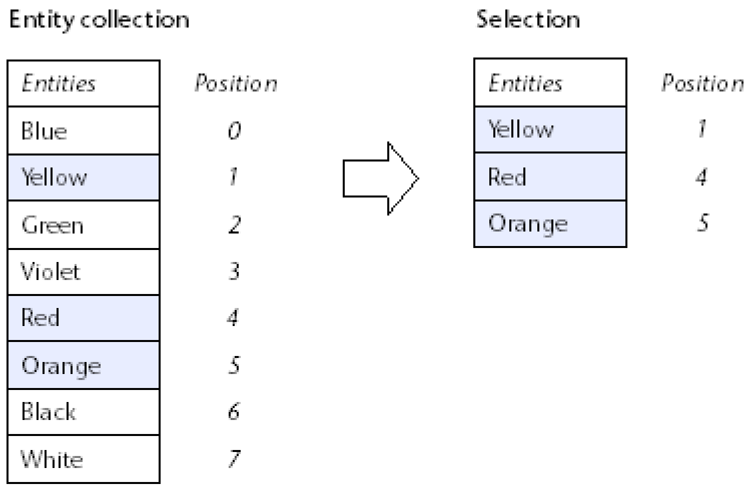
Description

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	Boolean	true if the selection is in single mode, false otherwise

Description

The **isSingleMode( )** method returns **true** if the *Selection* works in single selection mode and **false** otherwise. By default, widgets and, by consequent, selections work in single selection mode. The selection mode for a widget can be defined in the GUI Designer of Wakanda studio.

select( )

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

Parameter	Type	Description
pos	Number	Position of entity to select
addToSel	Boolean	true = add to selection, false = replace selection

Description

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

When the *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** in 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

We want to select the last entity of the datasource current selection:

```
var theSel = sources.item.getSelection(); // gets the current selection
var theLast = sources.item.length; // size of the entity collection
theSel.select(theLast-1); // selects the last entity (-1 because pos starts from 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 the *Selection* the entities whose range you defined using the *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** in the *addToSel* parameter. In this case, if an entity was already included in the *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

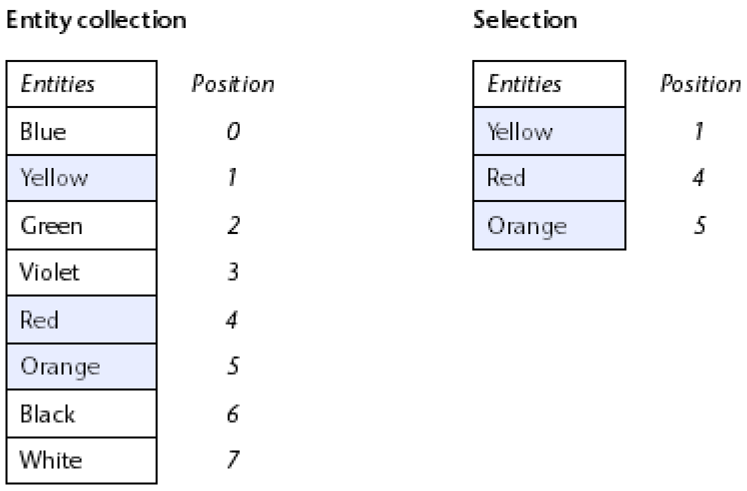
The **setSelectedRows( )** method allows you to set the selected entities in the parent entity collection, thus to modify the associated *Selection*.

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

If the *Selection* is in single selection mode, only the first entity position in array 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.colors.getSelection(); //gets the associated selection
col.setSelectedRows([0,2,6]); //sets 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*