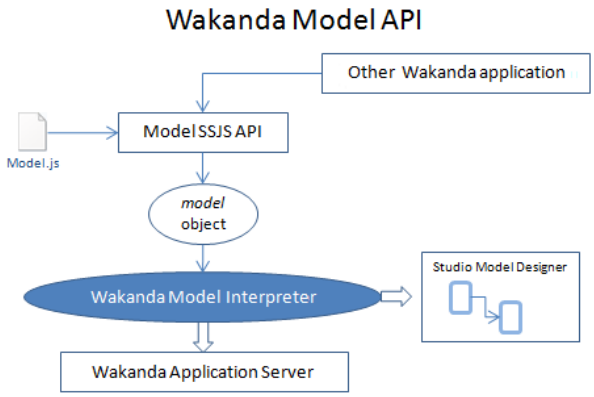
Model

The Wakanda Model server-side API allows you to build a model based on custom JavaScript code written in your Model.js file. You use the DataClass( ) and Attribute( ) constructors to create your datastore classes and attributes. You can also use the addClass( ) or addAttribute( ) methods. All of your model's objects can be defined using this API:

- **datastore classes** and their properties, including extended datastore classes,
- **attributes** (storage, calculated, and relation) along with their properties, including restricting queries and events, and
- **datastore class methods** and **events**.

At runtime, the model objects you created using the Model API will behave exactly as those you create using the Datastore Model Designer. There is no functional difference between how the model objects are created in a Wakanda project. This principle is the basis of the **Wakanda Model Architecture** where the active model can be built using different sources:



To enable the Wakanda procedural model generation, you just need to edit the **model** global object in the **Model.js** file by simple assignment. For more information, please refer to the Working with the Model API (v5) section.

Working with the Model API (v5)

About the model Object

The *model* object contains the JavaScript description of your Wakanda datastore model. It is automatically defined by Wakanda and can be modified in the "Model.js" file stored at the root of your Wakanda project. You can use **include( )** statements in the "Model.js" file to define JavaScript modules to load depending on your needs. Any JavaScript code stored in the "Model.js" file is executed by Wakanda to build or modify the datastore model.

Note:** If you defined your model using the Wakanda Studio's Model Designer, by default the* Model.js *object does not contain JavaScript code. The model definition is stored in json in the .waModel file (see* **Datastore Model Designer).*

Datastore classes, attributes or methods that have been defined (if any) using the Wakanda Studio's Model Designer can be modified using the Model.js code (the JavaScript code in Model.js is executed after the json-based model).

Writing Code for the model Object

Any JavaScript code that will define or modify your model need to be written or referenced in the "Model.js" at the root of your project. You can edit directly the Model.js file.

You can also add a first method from the Wakanda Studio's Datastore Model Designer: a dialog box is displayed, allowing you to define the location of the model files (with an automatic reference in the Model.js file). By default, existing datastore classes, attributes and methods are automatically stored in "Model" folder created at the root of the project and files such as "*dataClassName*-attributes.js" and "*dataClassName*-methods.js" are created.

You can put model code where you want once it is referenced in the "Model.JS" file using the [#cmd id=" 101065"/] method.

Using the Model Designer's Compliant Syntax

In Wakanda, several methods such as **addClass( )** or **addMethod( )** allows adding datastore classes and attributes as well as their properties (methods, restricting queries, event listener, etc.). In the current Wakanda release, these methods are still supported but are not recommended anymore, particularly if you want to use the Model Designer in conjunction with your procedural model declarations. Conversely, when you use code based on the **Attribute( )** or **DataClass( )** constructor methods, you can still have a link between the Datastore Model Designer and your code: you can use the  button to display the code for a datastore class method or an event.

We now recommend that you declare any datastore class or attribute element as properties of the **Attribute( )** and **DataClass( )** constructor method and to declare the associated code and methods directly. For example, if you want to be able to open the *onSet* method of a calculated attribute from the Model Designer directly, you must not use the **onSet( )** method but instead you should use the declaration syntax:

```
    //Supported but no longer suitable
    //No link with the Model Designer
var emp = model.addClass("Employee", "Employees");
    //...
emp.addAttribute("fullName", "calculated", "string");
emp.fullName.onSet = function(value)
{
    var names = value.split(' '); //split value into an array
    this.firstName = names[0];
    this.lastName = names[1];
}

    //Recommended code
    //Direct access from the Model Designer
model.Person.fullName = new Attribute("calculated", "string");
model.Person.fullName.onSet = function(value) {
    var names = value.split(' '); //split value into an array
    this.firstName = names[0];
    this.lastName = names[1];
}
```

Model Syntax

Once you have initialized a datastore class and its attributes using the **DataClass( )** and **Attribute( )** constructor methods, you need to use the following syntax to declare each object of the model that you want to create:

Syntax for a method applied to an entity:

```
model.className.entityMethods.method1 = function ();
model.className.entityMethods.method2 = function ();
model.className.entityMethods.method3 = function ();
```

Syntax for a method applied to a collection:

```
model.className.collectionMethods.method1 = function ();
model.className.collectionMethods.method2 = function ();
model.className.collectionMethods.method3 = function ();
```

Syntax for a method applied to a datastore class:

```
model.className.methods.method1 = function ();
model.className.methods.method2 = function ();
model.className.methods.method3 = function ();
```

Syntax for the scope of a datastore class method:

```
model.className.methods.methodName.scope ="public"
```

```
model.className.methods.methodName.scope ="publicOnServer"
```

Syntax for an attribute event:

```
model.className.attributeName.events.onInit = function()
model.className.attributeName.events.onLoad = function()
model.className.attributeName.events.onSet = function()
model.className.attributeName.events.onValidate = function()
model.className.attributeName.events.onSave = function()
model.className.attributeName.events.onRemove = function()
```

Syntax for a datastore class event:

```
model.className.events.onInit = function()
model.className.events.onLoad = function()
model.className.events.onValidate = function()
model.className.events.onSave = function()
model.className.events.onRemove = function()
model.className.events.onRestrictingQuery = function()
```

Syntax for a calculated attribute:

```
model.className.attributeName.onGet = function()
model.className.attributeName.onSet = function()
model.className.attributeName.onQuery = function()
model.className.attributeName.onSort = function()
```

***Note**: To see examples of the complete procedural syntax, you can convert an existing Studio-based model to a procedural model using the* 

button in the Model Designer and then take a look at the resulting code.

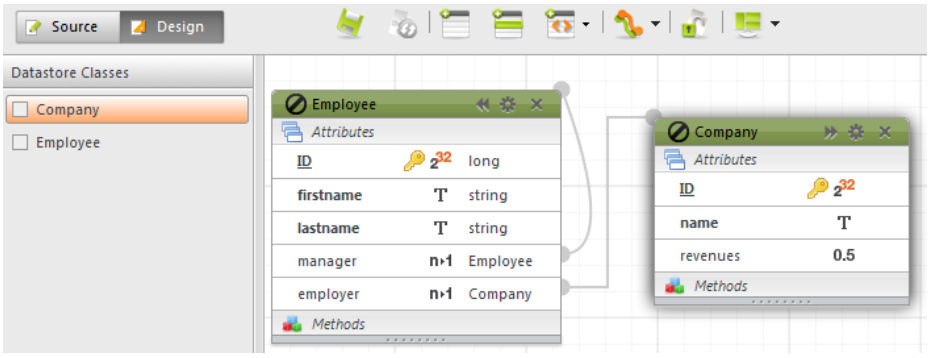
Viewing a Procedural Model in Wakanda Studio

You can view procedurally-built models in the Wakanda Studio Model Designer. This feature provides you with a visual representation of your datastore classes and their relations, exactly as if they were created through the Model Designer. You do not need to launch the server to see a procedural model: Wakanda Studio interprets and displays a model from the Model.js file exactly like the Wakanda Server does.

For example, if you write the following code in the Model.js file:

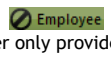
```
model.Employee = new DataClass("Employees" ,"public");

model.Employee.ID = new Attribute("storage", "long", "key auto");
model.Employee.firstname = new Attribute("storage", "string", "btree");
model.Employee.lastname = new Attribute("storage", "string", "btree");
model.Employee.manager = new Attribute("relatedEntity", "Employee", "Employee");
model.Employee.employer = new Attribute("relatedEntity", "Company", "Company");

model.Company = new DataClass("Companies" ,"public");
model.Company.ID = new Attribute("storage", "long", "key auto");
model.Company.name = new Attribute("storage", "string", "btree");
model.Company.revenues = new Attribute("storage", "number");
```

When you display the *Model.waModel* file in the Model Designer, you can see the following representation of the model:



***Note:** You may need to select datastore class names in the outline list on the left side in order to have them actually be drawn or refreshed in the designer.*

Any modification applied to the model definition in the Model.js file is immediately carried out in the Wakanda Studio as well.

You cannot edit procedural datastore classes through the Model designer. The locked icon 

shows that datastore classes are generated from the *model* JavaScrip object. Also, the **Source** button of the Model designer only provides access to extra properties of the model (json format).

We do not recommend adding standard datastore class through the Datastore Model Designer. Such operations would produce **hybrid models** (models containing both procedurally-based datastore classes and standard Studio-based datastore classes). Although supported, hybrid models require that you take extra care to manage potential naming conflicts and also add complexity for maintenance operations.

Attribute

The *Attribute* class contains properties and methods allowing you to add events to and to define your datastore class attributes.

Attribute model objects are referenced through the **model**.{className}.{attributeName} property, for example:

```
model.Employee.lastName.type = "string"; //lastName is an attribute of the Employee class
```

Alternatively, they can be created by the addAttribute( ), addRelatedEntities( ) and addRelatedEntity( ) methods.

onGet

Description

The **onGet** property contains the JavaScript function that describes how the calculated *Attribute* value will be evaluated. This property is only available for calculated attributes. For more information on calculated attributes, please refer to the Calculated Attribute paragraph.

When such a property is provided for a calculated attribute, Wakanda does not create the associated underlying storage space in the datastore but instead substitutes the function's code each time this attribute is accessed. If the attribute is not accessed, then the code never executes.

The JavaScript function stored in the **onGet** property will be called when the attribute is accessed. This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it.
Keep in mind that, within the **onGet** function, **this** represents the entity that is being processed.

Example

We define the 'onGet' method for the *hired* Boolean attribute:

```
model.Employee = new DataClass("Employees");
model.Employee.hiringDate = new Attribute("storage", "date");
model.Employee.hired = new Attribute("calculated", "bool");
    // onGet function
model.Employee.hired.onGet = function()
{
    return this.hiringDate != null;
}
```

onSet

Description

The **onSet** property contains the function define the JavaScript function that describes how a value entered in the calculated *Attribute* will be processed[#/descv]. This method is only available for calculated attributes. For more information on calculated attributes, please refer to the Calculated Attribute paragraph.

The JavaScript function stored in the **onSet** property is called when a value is entered in the attribute. This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it.
The function will receive the entered value as parameter. Keep in mind that, within the *jsCode* function, **this** represents the entity that is being processed.

Example

The *fullName* parameter is an enterable calculated attribute whose entered contents are processed and stored in other storage attributes:

```
model.Employee = new DataClass("Employees");
model.Employee.lastName = new Attribute("storage", "string");
model.Employee.firstName = new Attribute("storage", "string");
model.Employee.fullName = new Attribute("calculated", "string");
    // onSet function
model.Employee.fullName.onSet = function(value)
{
    var names = value.split(' '); //split value into an array
    this.firstName = names[0];
    this.lastName = names[1];
}
```

onQuery

Description

The **onQuery** property contains the JavaScript function to execute when the calculated *Attribute* is used in a query. This property is only available for calculated attributes. For more information on calculated attributes, please refer to the Calculated Attribute paragraph.

The JavaScript function stored in the **onQuery** property should actually return a string that represents a redirected query.
The function will receive two parameters: the comparison operator (e.g. ">", ">=", etc.) and the compared value.

This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it.

Example

We define the 'onQuery' method for the *hired* boolean calculated attribute. Querying this attribute will actually return an appropriate query string applied to the *hiringDate* attribute:

```
model.Employee = new DataClass("Employees");
```

```
model.Employee.hiringDate = new Attribute("storage", "date");
model.Employee.hired = new Attribute("calculated", "bool");
    // onGet function
model.Employee.hired.onGet = function()
{
    return this.hiringDate != null;
}
    // onQuery function
model.Employee.hired.onQuery = function(compareOperator, compareValue)
{
    var newOper;
    if (compareOperator === "=" || compareOperator === "==")
    {
        if (compareValue === true)
            newOper = "is not";
        else
            newOper = "is";
    }
    else
    {
        if (compareValue === true)
            newOper = "is";
        else
            newOper = "is not";
    }
    return "hiringDate "+newOper+" null";
}
```

onSort

Description

The **onSort** property contains the JavaScript function that describes how the calculated *Attribute* must be sorted when an order by operation is triggered on it. This property is only available for calculated attributes. For more information on calculated attributes, please refer to the Calculated Attribute paragraph.

The JavaScript function stored in the **onSort** property must return a string that will be used as the sort operation criteria.
The function will receive a boolean value as parameter, that describes whether the sort is ascending or descending.

Keep in mind that, within the function, **this** represents the entity that is being processed.

This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it.

Example

This example shows how to sort an *age* calculated attribute on a *birthdate* storage value:

```
model.Employee = new DataClass("Employees");
model.Employee.birthdate = new Attribute("storage", "date", "btree");
model.Employee.age = new Attribute("calculated", "long");
    // onGet function
model.Employee.age.onGet = function()
{
    if (this.birthdate == null)
        return null;
    else
    {
        var today = new Date();
        var interval = today.getTime() - this.birthdate.getTime();
        var nbYears = Math.floor(interval / (1000 * 60 * 60 * 24 * 365.25));
        return nbYears;
    }
}
    // onSort function
model.Employee.age.onSort = function(ascending)
{
    if (ascending)
        return "birthdate";
    else
        return "birthdate desc";
}
```

events

Description

The **events** property contains an object that stores all the events for the attribute. Each property of this object is an event name and its value is the function. You can use this property to create or modify attribute events.

Properties of the **events** object are detailed in the Attribute Events chapter.

Note: *You can also use the* addEventListener( ) *method to install attribute events.*

type

Description

The **type** property defines the type of the Wakanda *Attribute*.

Available values depend on the attribute **kind**.

- For storage, calculated or alias attributes, **type** can be one of the standard supported Wakanda data types:
  "blob"
  "bool"
  "byte"
  "date"
  "duration"
  "image"
  "long"
  "long64"
  "number"
  "string"
  "uuid"
  "word"
- For relatedEntity attributes, **type** is the datastore class name corresponding to the 'one' class
- For relatedEntities attributes, **type** is the datastore class name corresponding to the 'many' class

Example

```
model.Employee.birthdate.type = "date";
```

kind

Description

The **kind** property defines the kind of the *Attribute*.

Available values are:

- "storage": to store simple scalar values such as strings, longs, etc.
- "calculated": to store scalar values based on a calculation, such as lastName+name
- "alias": an attribute built upon a relation attribute
- "relatedEntity": a N->1 relation attribute
- "relatedEntities": a 1->N relation attribute

Example

```
model.Employee.manager.kind = "relatedEntity"
```

path

Description

The **path** property defines the path used for related or alias *Attribute*.

For a detailed description of this property, please refer to the **Attribute( )** method.

Example

```
model.Employee.workingPlace = new Attribute("relatedEntity", "City");
model.Employee.workingPlace.path = "employer.location"; // relation to the City class
```

indexKind

Description

The **indexKind** property defines the index kind for a storage *Attribute*.

The following values are available:

- "btree": associates a B-Tree type index with the attribute. This multipurpose index type meets most indexing requirements.
- "cluster": associates a B-Tree type index using clusters with the attribute. This architecture is more efficient when the index does not contain a large number of keys, i.e., when the same values occur frequently in the data.

Example

```
model.Employee.lastname.indexKind = "btree";
```

primKey

Description

The **primKey** property contains **true** if the *Attribute* is the primary key of the datastore class.

Example

```
model.Company.name.primKey = true;
```

autosequence

Description

The **autosequence** property contains **true** if the *Attribute* has the Autosequence property on.

This property is available for numbers attributes only. When it is true, Wakanda automatically generates a new number in the attribute for each new datastore entity created following a sequence. This property is useful for primary key attributes.

Example

```
model.Company.ID.autosequence = true;
```

autogenerate

Description

The **autogenerate** property contains **true** if the *Attribute* has the Autogenerate property on.

This property is available for UUDI attributes only. When it is true, the UUID value will be generated automatically in the attribute by Wakanda for each new datastore entity created. If false, you must generate a valid UUID through your code. This property is useful for primary key attributes.

Example

```
model.Article.IDCode.autogenerate = true;
```

simpleDate

Description

The **simpleDate** property contains **true** if the date *Attribute* stores the date in "DD/MM/YYYY" format (e.g., "10/05/2013").

Otherwise, date values include the time, stored in UTC. The date is expressed in the following format: YYYY-MM-DDTHH:MM:SSZ (e.g., "2010-10-05T23:00:00Z" for October 5, 2010 in the Central European Timezone).

Example

```
model.Employee.birthdate.simpleDate = true;
```

scope

Description

The **scope** property defines the scope of the *Attribute*.

A "public" attribute can be used from anywhere, while a "public on server" attribute can only be accessed from the server. By default when the property is not set, attributes are "public".

Example

```
model.Employee.salary.scope = "publicOnServer";
```

limiting_length

Description

The **limiting_length** property defines the maximum number of characters enterable in the *Attribute*.

If the limiting length is set to 10, any longer text entered will be truncated to contain 10 characters.

Example

```
model.Article.code.limiting_length = 10;
```

blob_switch_size

Description

The **blob_switch_size** property defines the size in bytes below which the data of the BLOB *Attribute* will be stored within entities.

For example, if you enter 30 000, a 20 KB BLOB will be stored in the entity and a 40 KB BLOB will be stored outside the entitty. By default, the value is 0: all BLOB data are stored outside of entities.

Example

```
model.Article.rawData.blob_switch_size = 30000;
```

outside_blob

Description

The **outside_blob** property contains true if BLOB *Attribute* data are stored outside of the data file. By default (false), BLOB data are stored inside the data file.

Example

```
model.Article.techDoc.outside_blob = true;
```

unique

Description

The **unique** property contains true if Wakanda Server checks that values entered in the *Attribute* are unique. When this property is true, Wakanda returns an error when an entered value is duplicated.

```
model.Country.name.unique =  true;
```

not_null

Description

The **not_null** property contains true if Wakanda Server checks that a value is actually entered in the *Attribute*. When this property is true, Wakanda returns an error when you try to save an entity with the null value in the attribute.

Example

```
model.Country.name.nut_null = true;
```

autoComplete

Description

The **autoComplete** property contains true if Wakanda automatically builds a list of possible values based on existing values for the same *Attribute* during data entry. For example, if you enter “Ab” in a first name attribute, all the first names in the datastore class that start with “Ab” will appear in a list for you to select from.

This property is available for string attributes only.

Example

```
model.City.country.autoComplete = true
```

styled_text

Description

The **styled_text** property contains true if the *Attribute* stores styled text. Queries and sorts carried out in the data stored in the *Attribute* do not take any style tags into account.

This property is available for string attributes only.

Example

```
model.Employee.comments.styled_text = true
```

multiLine

Description

The **multiLine** property contains true if the *Attribute* stores multi-line text.

This property is only available for string attributes.

Example

```
model.Employee.comments.multiLine = true
```

readOnly

Description

The **readOnly** property contains true if the *Attribute* value cannot be set by user editing; it can only be set through the code.

Example

```
model.Company.ID.readOnly = true;
```

reversePath

Description

The **reversePath** property contains true if the relation attribute uses the reverse path of an existing relation.

For a detailed description of this property, please refer to the **RelatedEntity or relatedEntities attributes** in the **Attribute( )** method.

addEventListener( )

void **addEventListener**(String *event*, Function *jsCode*)

Parameter	Type	Description
event	String	Attribute event to listen to
jsCode	Function	JavaScript function to execute

Description

The **addEventListener( )** method allows you to associate an event listener function with the attribute.

***Note:** For more information about event listeners, please refer to the **Using Datastore Class Events** section.*

Using this method, you can define several event listeners for the same *event*.

In *event*, pass the name of the event to define. For attributes, available events are:

- "onInit"
- "onLoad"
- "onSet"
- "onValidate"
- "onSave"
- "onRemove"

For more information on these events, please refer to the **Description of events** section.

In *jsCode*, pass the JavaScript function to call when the event is generated. This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it.
Keep in mind that, within the *jsCode* function, **this** represents the entity that is being processed.
Unlike an event listener method associated with a datastore class (see **addEventListener( )** for datastore classes), the *jsCode* function associated with an attribute event listener will receive a parameter that is the name of the attribute. You can use this parameter within the function, for example to indicate the name of the attribute in an error message.

***Note:** You can also define events through the **events** property of the attribute.*

Example

We want to add events to the name and salary attributes:

```
var emp = model.addClass("Employee", "Employees");
emp.addAttribute("ID", "storage", "long", "key auto");
    //... add other attributes
var theName = emp.addAttribute("lastname", "storage", "string", "btree");
theName.addEventListener("onSet", setToCapitalize); // add an onSet event
    //you can also pass an event directly
emp.addAttribute("salary", "storage", "number", "cluster").addEventListener("onValidate", isInRange);

    // event functions
    // they can be used for different attributes
function setToCapitalize(attributeName)
{
    if (this[attributeName] != null)
    {
        this[attributeName] = this[attributeName].capitalize();
    }
}

function isInRange(attributeName)
{
    if (this[attributeName] < 1000 || this[attributeName] > 100000)
    {
        return { error: 1000, errorMessage: attributeName+" is out of range" };
    }
}
```

Attribute Constructor

Attribute( )

DatastoreClassAttribute **Attribute**(String | Null *kind*, String *type* [,String *indexOrPath* [,Object *options*]])

Parameter	Type	Description
kind	String, Null	Kind of the attribute (null = storage)
type	String	Type of the attribute
indexOrPath	String	Index kind (storage attributes) or relation path (relation attributes)
options	Object	Properties for the attribute
Returns	DatastoreClassAttribute	Reference to the created attribute

Description

The **Attribute( )** method is the constructor of the *DatastoreClassAttribute* type objects. It allows you to instantiate a new attribute in a *DatastoreClass* object. *DatastoreClassAttribute* objects are handled using the various properties and methods of the Attribute class.

Basically, this constructor method provides the same effect as the addAttribute( ) method. However, unlike addAttribute( ), **Attribute( )** allows you to maintain a link between the Wakanda Studio's Model Designer and the JavaScript code. For example, if you create a calculated attribute, you will be able to display the associated JavaScript code from the Model Designer by clicking the button. By consequent, **Attribute( )** is recommended to create an attribute.

To create an attribute object using the **Attribute( )** constructor, you must:

- use the **new** operator to create an instance of the object,
- assign the instance to a property of an existing *DatastoreClass* object; the name of the property will become the name of the attribute (see below).

For example, if you want to create a storage attribute named "lastName" in an existing "Student" datastore class, you must write:

```
model.Student.lastName = new Attribute("storage", "string");
```

In addition to the standard Reserved Keywords, the following words are reserved in procedural models and must not be used as attribute names:

Reserved keywords for attribute names in procedural models
properties
methods
collectionMethods
entityMethods
events
attributes

Once the attribute is instanciated, a prototyped *DatastoreClassAttribute* object is created and benefit from the various API methods of the Attribute class, such as addEventListener( ).

In *kind*, pass the kind property of the attribute to create. Available values are:

- "storage": to store simple scalar values such as strings, longs, etc.
- "calculated": to store scalar values based on a calculation, such as lastName+name
- "alias": an attribute built upon a relation attribute
- "relatedEntity": a N->1 relation attribute
- "relatedEntities": a 1->N relation attribute

For more information about attribute kinds, please refer to the Attribute Categories paragraph.

The values to pass in the *type* and *indexOrPath* parameters will depend highly on the attribute *kind*:

Storage and calculated attributes

Regarding storage, calculated or alias attributes:

- *type* can be one of the standard supported Wakanda data types:
  "blob"
  "bool"
  "byte"
  "date"
  "duration"
  "image"
  "long"
  "long64"
  "number"
  "string"
  "uuid"
  "word"
  For more information, please refer to the Storage Attribute Types section.
- *indexOrPath* is used either to define the index kind, to designate the primary key for a storage attribute, or the path for an alias attribute. You can pass one of the following values:

- "btree": associates a B-Tree type index with the attribute. This multipurpose index type meets most indexing requirements.
- "cluster": associates a B-Tree type index using clusters with the attribute. This architecture is more efficient when the index does not contain a large number of keys, i.e., when the same values occur frequently in the data.
- "key": designates the attribute as the primary key of the datastore class.
- "key auto": designates the attribute as the primary key of the datastore class with the *automatic* property:
  - for numeric types, the **autosequence** property is set
  - for uuid types, the **autogenerate** property is set

  ***Note:** A btree index is automatically set for primary key attributes.*
- a string providing the path for an alias attribute, for example "employees.location".

RelatedEntity or relatedEntities attributes

Regarding relation attributes:

- For "relatedEntity" attributes, the *type* is the datastore class name corresponding to the 'one' class. For example, in a classic *Employee->Company* relation, the *type* for an *employer* attribute added to the *Employee* class would be "Company".
  In the *indexOrPath* parameter, pass the path to the related entity.
  - For simple cases in a N->1 configuration, the path is implicit and built upon the *type*. You just need to pass the relation datastore class name. In our example, it would be "Company" again. This will create a foreign key in the *Company* datastore class.
  - In more complex cases, you may not want to create a foreign key but instead use existing relations. For example, if you have three datastore classes, *Employee - Company - City*, and an existing relation between *Company* and *City*, you can create a relation attribute in *Employee* based upon this existing relation in order to get the employee's work location. In this case, you will create an attribute named "workingPlace" in *Employee* of the *kind* "relatedEntity" and the *type* "City" and pass a custom path in the *indexOrPath* parameter, for example "employer.location" (*employer* is the N->1 relation attribute from *Employee* and *location* is the N->1 relation attribute from *Company*).

- For "relatedEntities" attributes, the *type* is the datastore class collection name corresponding to the 'many' class. For example, in the *Employee->Company* relation, the *type* for an *employees* attribute added to the *Company* class would be "Employees" (or "EmployeeCollection" if you left the default name).
  Regarding the *indexOrPath* parameter, two cases are to be considered:
  - you want to use the **reverse path** of an existing "relatedEntity" attribute. You just need to pass the "relatedEntity" attribute name (from the 'many' class) as the path in the *indexOrPath* parameter and add a {reverserPath: true} object as a 5th parameter. For example, you want to add in the *Company* datastore class a "relatedEntities" attribute named "employees" that will contain all employees working for the company. The *employer* N->1 relation attribute already exists in the *Employee* datastore class, so you can just use the reverse path to build the appropriate collection:

    ```
    emp.addAttribute("employees", "relatedEntities", "Employees", "employer", {reversePath:true});
    ```

    Setting the reverse path is necessary so that the existing foreign key of the 'many' class is used to establish the relation.
  - you want to use a **custom path** through several datastore classes and benefit from existing relations (which can be reverse paths). For example, if you have three datastore classes, *Employee - Company - City* with existing relations between *Company -> City*, and *Employee -> Company*, you can create a relation attribute in *City* based upon these existing relations in order to get the collection of employees working in the city. You could create the reverse path of the *workingPlace* attribute (see above). But, you can also decide to use a custom path for your attribute (both attributes would work the same way): create an attribute named "workForce" in *City* of the *kind* "relatedEntities" and the *type* "Employees" and pass a custom path in the *indexOrPath* parameter, for example "companies.employees" (*companies* is the 1->N relation attribute from *City* and *employees* is the 1->N relation attribute from *Employee*).
    In this case, you do not need to pass the "reversePath" option because the custom path already uses the reverse paths.

options

The *options* parameter is an object containing several key/value pairs allowing you to set various properties to the created storage or relation attribute.

The following properties are available for **storage** attributes:

Option	Type	Description
simpleDate	boolean	If true, the date is stored in "DD/MM/YYYY" format. Equivalent to the "Date only" Model Designer property.
scope	string	"public" (default) or "publicOnServer". A "public" attribute can be used from anywhere, while a "public on server" attribute can only be accessed from the server.
limiting_length	number	Limits the length of the text entered in the attribute. If you define the limiting length to be 10, any longer text entered will be truncated to contain 10 characters.
blob_switch_size	number	Size in bytes below which the data of the BLOB attribute will be stored within entities. For example, if you enter 30 000, a 20 KB BLOB will be stored in the entity and a 40 KB BLOB will be stored outside the entity. By default, the value is 0: all BLOB data are stored outside of entities.
outside_blob	boolean	If true, BLOB data will be stored outside of the data file. By default (false), BLOB data are stored inside the data file.
unique	boolean	If true, values entered in the attribute must be unique. If not, an error is returned.
not_null	boolean	If true, the attribute is mandatory; it cannot be null. Otherwise, an error is returned.
autosequence	boolean	For number attributes only. If true, Wakanda automatically generates a new number for each new datastore entity created following a sequence.
autogenerate	boolean	For UUID attributes only. If true, the UUID will be generated automatically by Wakanda for each new datastore entity created. If false, you must generate a valid UUID through the code.

autoComplete	boolean	For string attributes only. If true, Wakanda automatically builds a list of possible values based on existing values for the same attribute during data entry.
styled_text	boolean	If true, queries and sorts carried out in the data stored in the attribute do not take any style tags into account.
multiLine	boolean	If true, the attribute will appear by default as a multi-line widget.
readOnly	boolean	If true, the attribute value cannot be set by user editing; it can only be set through the code.
primKey	boolean	Sets the attribute as the new primary key(*).
indexKind	string	Sets the index kind for the attribute(*).
kind	string	Sets the kind for the attribute(*).
type	string	Sets the type for the attribute(*)

(*) These properties should usually be set through the **Attribute( )** method parameters.

The following property is available for **relation** attributes:

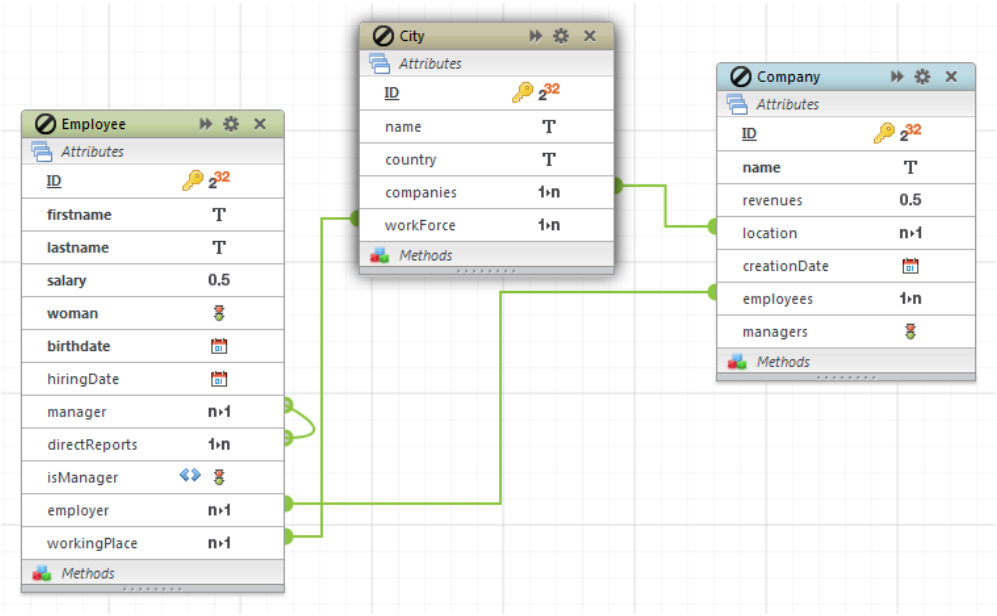
Option	Type	Description
reversePath	boolean	If true, the relation attribute will use the reverse path of an existing relation (for more information, please refer to the **RelatedEntity or relatedEntities attributes** paragraph.

Example

In this example, we will create a complete *Employee - Company - City* model to illustrate the various ways to add attributes in your model:

```
//Creating the Employee class
model.Employee = new DataClass("Employees");

model.Employee.ID = new Attribute("storage", "long", "key auto");
model.Employee.firstname = new Attribute("storage", "string", "btree");
model.Employee.lastname = new Attribute("storage", "string", "btree");
model.Employee.salary = new Attribute("storage", "number", "cluster");
model.Employee.woman = new Attribute("storage", "bool", "cluster");
model.Employee.birthdate = new Attribute("storage", "date", "btree");
model.Employee.hiringDate = new Attribute("storage", "date");
model.Employee.manager = new Attribute("relatedEntity", "Employee", "Employee");
model.Employee.directReports = new Attribute("relatedEntities", "Employees", "manager", {reversePath:true}
model.Employee.isManager = new Attribute("calculated", "bool"); //onGet method is defined below
model.Employee.employer = new Attribute("relatedEntity", "Company", "Company"); // relation to the Company
model.Employee.workingPlace = new Attribute("relatedEntity", "City", "employer.location"); // relation to

//onGet for the calculated atttibute
model.Employee.isManager.onGet = function()
{
    return this.directReports.length != 0;
}

    //Creating the Company class
model.Company = new DataClass("Companies");
model.Company.ID = new Attribute("storage", "long", "key auto");
model.Company.name = new Attribute("storage", "string", "btree");
model.Company.revenues = new Attribute("storage", "number");
model.Company.creationDate = new Attribute("storage", "date");
model.Company.location = new Attribute("relatedEntity", "City", "City");
model.Company.employees = new Attribute("relatedEntities", "Employees", "employer", {reversePath:true});
model.Company.managers = new Attribute("alias", "Employees", "employees.manager");

    //Creating the City class
model.City = new DataClass("Cities");
model.City.ID = new Attribute("storage", "long", "key auto");
model.City.name = new Attribute("storage", "string" , {autoComplete:true});
model.City.country = new Attribute("storage", "string", {unique:true , not_null : true });
model.City.companies = new Attribute("relatedEntities", "Companies", "location", { reversePath:true } );
model.City.workForce = new Attribute("relatedEntities", "Employees", "workingPlace", { reversePath:true }
    // this last relation could also have been defined like this:
    // new Attribute("relatedEntities", "Employees", "companies.employees")
```

You can preview the result in the Wakanda Studio Model Designer:



Example

You want to add an attribute with several options:

```
model.Employee.salary = new Attribute("storage", "number", null, {
    "minValue": "10000",
    "maxValue": "500000",
    "defaultFormat": {
        "format": "###,###,###.00",
        "presentation": "text"
    }
});
```

Attribute Events

The described event handler functions have to be set on an attribute through the following property:

model.{className}.{attributeName}.events

For example:

```
model.Employee.lastName.events.onInit = function(){
    this.lastName = " "; // assigns empty string to the attribute
}
```

onInit

Description

The **onInit** property contains the event function that is to be called just after a new entity is created in memory on the server. You can use this event to initialize attribute values, for example a custom ID.

This event is available at both the DatastoreClass and Attribute levels.

When this event is set for a datastore class and for one or more attribute(s), the calling order is as follows:

1. datastore class event
2. attribute event(s)

Example

You want to associate an *onInit* event to the "lastName" attribute:

```
model.Employee.lastName.events.onInit = function(){
    this.lastName = " "; // assigns empty string to the attribute
}
```

onLoad

Description

The **onLoad** property contains the event function to call just after an entity or an attribute is accessed.

This event is available at both the DatastoreClass and Attribute levels and is called differently depending on the level:

- on a datastore class: called each time an existing entity is loaded in memory on the server.
- on an attribute: called the first time an attribute is used (i.e., read or modified) for an entity, after this entity is loaded, and then is not called again for any subsequent times.

For more information, please refer to the Description of events and Class Events and Attribute Events sections.

Example

You want to add an *onLoad* event to the Employee class:

```
model.Employee.events.onLoad = function(){
    if (this.photo.size == 0)  //no picture for the employee
        this.photo = loadImage('c:/Projects/Docs/WebFolder/NoPhoto.jpg'); //add a default one
}
```

onValidate

Description

The **onValidate** property contains the event function to call before saving an entity on the server, for example following a call to the save( ) method. It is automatically called before the **onSave** event. It is also generated when you call the validate( ) method. You can use this event to check the validity of the data entered compared with business rules that you have set. To reject the validation, you pass a specific object as the result of the function.

This event is available at both the DatastoreClass and Attribute levels. When this event is set for a datastore class and for one or more attribute(s), the calling order is as follows:

1. attribute event(s)
2. datastore class event

For more information, please refer to the Description of events and Class Events and Attribute Events sections.

Example

You want to add the *hasAtLeastOne* function as **onValidate** event:

```
model.Article.events.onValidate = function() {
    if (this.name == null) && (this.code == null) {
        return {error: 7, errorMessage: 'name or code must have a value'};
    }
    else {
        return {error: 0}; //Same as no error
```

```
    }
}
```

onSave

Description

The **onSave** property contains the event function to call just before saving an entity on the server, for example following a call to the **save( )** method and if the entity has passed validation. To reject the save, you pass a specific object as the result of the function.

This event is available at both the **DatastoreClass** and **Attribute** levels. When this event is set for a datastore class and for one or more attribute(s), the calling order is as follows:

1. datastore class event
2. attribute event(s)

When the event is set for one or more attribute(s), for better optimization, it is called only for each modified attribute. For this reason, unlike *onValidate*, the datastore class event is called before attribute events.

For more information, please refer to the **Description of events** and **Class Events and Attribute Events** sections.

Example

You want to set a datastore class method named *checkForUpdate* as the *onSave* event:

```
model.Item.events.onSave = function() {
    ds.Update.checkForUpdate(this);
}
```

onRemove

Description

The **onRemove** property contains the event function to be call just before an entity is to be deleted. It can be used for a variety of purposes including cleaning up related entities and validating the deletion. To reject the deletion, you pass a specific object as the result of the function.

This event is available at both the **DatastoreClass** and **Attribute** levels. When this event is set for a datastore class and for one or more attribute(s), the calling order is as follows:

1. attribute event(s)
2. datastore class event

Example

You want to add an *onRemove* event to check integrity:

```
model.Customer.events.onRemove = function(){
    if(this.allOrders.length > 0) { //reject the deletion
        return {errorCode: -123, errorMessage: "Cannot delete an entity with bound Orders"};
    }
}
```

onSet

Description

The **onSet** property contains the event function to call just after an attribute's value is set. In server-side code, this executes when the value is set. When an attribute is set in client-side code, this event executes when the entity is saved or the method **serverRefresh( )** is called, but only for each attribute that was modified.

This event is available only at the **Attribute** level.

Example

This event is useful in the situation where values from a parent entity need to be copied into a related entity, for example:

```
model.InvoiceItem.itemPart.events.onSet = function(){
    this.name = this.itemPart.name;
    this.cost = this.itemPart.cost;
}
```

DatastoreClass

The *DatastoreClass* class contains methods allowing you to add attributes and properties to the datastore class objects in your model reference.

DatastoreClass objects are referenced through the **model.**{className} property.

Alternatively, they can be created by the addClass( ) method.

{attributeName}

Description

Each datastore class attribute defined in the *DatastoreClass* dynamic object is available as a property of this object.

This property is a read-write object: you can modify or even create an attribute using the returned reference. In addition to the standard Reserved Keywords, the following words are reserved in dynamic models and must not be used as attribute names:

Reserved keywords for attribute names in dynamic models
properties
methods
collectionMethods
entityMethods
events
attributes

Note: *If you create a datastore class by building the object instead of using addAttribute( ) or its shortcut methods, you do not benefit from the instance methods of the Attribute class provided by the prototype.*

Example

The attribute name can be used to define events for calculated attributes:

```
var emp = model.addClass("Employee", "Employees");
emp.addAttribute("hiringDate", "storage", "date");
emp.addAttribute("hired", "calculated", "bool"); // add a calculated attribute

emp.hired.onGet = function() // use the attribute name property
{
    return this.hiringDate != null;
}
```

{methodName}

Description

Each datastore class method defined in the *DatastoreClass* dynamic object is available as a property of this object.

This property is a read-write object: you can modify or even create a method using the returned reference. If you create a method using this property, by default it is applied to "class" objects.

In addition to the standard Reserved Keywords, the following words are reserved in dynamic models and must not be used as method names:

Reserved keywords for attribute names in dynamic models
properties
methods
collectionMethods
entityMethods
events
attributes

Example

You want to create a new datastore class method in the Product class:

```
model.Product.addCountryCode = function(x,y) {
      this.name += x;
      this.version += y;
      this.save();
}
```

entityMethods

Description

The **entityMethods** property contains an object that stores each datastore class method applied to "Entity" objects of the datastore class. You can use this property to create or modify datastore class methods.

Example

Add an entity method to the Employee class:

```
model.Employee.entityMethods.getStaff = function() {
    return this.directReports.toArray("firstname,lastname");
};
```

collectionMethods

Description

The **collectionMethods** property contains an object that stores each datastore class method applied to "Collection" objects of the datastore class. You can use this property to create or modify datastore class methods.

Example

Add an entity collection method to the Employee class:

```
model.Employee.collectionMethods.getEmps = function(from, to) {
    from = from || 0;
    to = to || this.length;
    return this.toArray("firstname,lastname,salary", from, to);
        //return this.toArray(ds.Employee.firstname,ds.Employee.lastname,ds.Employee.salary, from, to);
};
```

methods

Description

The **methods** property contains an object that stores each datastore class method applied to the "Class" object itself. You can use this property to create or modify datastore class methods.

Example

Add a Class method to the Employee class:

```
model.Employee.methods.getFirstOnes = function(limit) {
    limit = limit || 2000;
    var col = ds.Employee.query("ID < :1", limit);
    return col;
};
```

events

Description

The **events** property contains an object that stores all the events for the datastore class. Each property of this object is an event name and its value is the function. You can use this property to create or modify datastore class events.

Properties of the **events** object are detailed in the DatastoreClass Events chapter.

addAttribute( )

DatastoreClassAttribute **addAttribute**(String *name*, String | Null *kind*, String *type*[, String *indexOrPath*][, Object *options*])

Parameter	Type	Description
name	String	Name of the attribute
kind	String, Null	Kind of the attribute (null = storage)
type	String	Type of the attribute
indexOrPath	String	Index kind (storage attributes) or relation path (relation attributes)
options	Object	Properties for the attribute
Returns	DatastoreClassAttribute	Reference to the created attribute

Description

The **addAttribute( )** method adds a new attribute to the datastore class. All Wakanda attributes are supported: storage, calculated, alias or relation.

In *name*, pass the name of the attribute to create. In addition to the standard Reserved Keywords, the following words are reserved in procedural models and must not be used as attribute names:

Reserved keywords for attribute names in procedural models
properties
methods
collectionMethods
entityMethods
events
attributes

In *kind*, pass the kind property of the attribute to create. Available values are:

- "storage": to store simple scalar values such as strings, longs, etc.
- "calculated": to store scalar values based on a calculation, such as lastName+name
- "alias": an attribute built upon a relation attribute
- "relatedEntity": a N->1 relation attribute
- "relatedEntities": a 1->N relation attribute

For more information about attribute kinds, please refer to the **Attribute Categories** paragraph.

The values to pass in the *type* and *indexOrPath* parameters will depend highly on the attribute *kind*:

Storage and calculated attributes

Regarding storage, calculated or alias attributes:

- *type* can be one of the standard supported Wakanda data types:
  "blob"
  "bool"
  "byte"
  "date"
  "duration"
  "image"
  "long"
  "long64"
  "number"
  "string"
  "uuid"
  "word"
  For more information, please refer to the **Storage Attribute Types** section.
- *indexOrPath* is used either to define the index kind, to designate the primary key for a storage attribute, or the path for an alias attribute. You can pass one of the following values:
  - "btree": associates a B-Tree type index with the attribute. This multipurpose index type meets most indexing requirements.
  - "cluster": associates a B-Tree type index using clusters with the attribute. This architecture is more efficient when the index does not contain a large number of keys, i.e., when the same values occur frequently in the data.
  - "key": designates the attribute as the primary key of the datastore class.
  - "key auto": designates the attribute as the primary key of the datastore class with the *automatic* property:
    - for numeric types, the **autosequence** property is set
    - for uuid types, the **autogenerate** property is set

    Note: *A btree index is automatically set for primary key attributes.*
  - a string providing the path for an alias attribute, for example "employees.location".

RelatedEntity or relatedEntities attributes

Regarding relation attributes:

- For "relatedEntity" attributes, the *type* is the datastore class name corresponding to the 'one' class. For example, in a classic *Employee->Company* relation, the *type* for an *employer* attribute added to the *Employee* class would be "Company".
  In the *indexOrPath* parameter, pass the path to the related entity.
  - For simple cases in a N->1 configuration, the path is implicit and built upon the *type*. You just need to pass the relation datastore class name. In our example, it would be "Company" again. This will create a foreign key in the *Company* datastore class.
  - In more complex cases, you may not want to create a foreign key but instead use existing relations. For example, if you have three datastore classes, *Employee - Company - City*, and an existing relation between *Company* and *City*, you can create a relation attribute in *Employee* based upon this existing relation in order to get the employee's work location. In this case, you will create an attribute named "workingPlace" in *Employee* of the *kind* "relatedEntity" and the *type* "City" and pass a custom path in the *indexOrPath* parameter, for example "employer.location" (*employer* is the N->1 relation attribute from *Employee* and *location* is the N->1 relation attribute from *Company*).
- For "relatedEntities" attributes, the *type* is the datastore class collection name corresponding to the 'many' class. For example, in the *Employee->Company* relation, the *type* for an *employees* attribute added to the *Company* class would be "Employees" (or "EmployeeCollection" if you left the default name).
  Regarding the *indexOrPath* parameter, two cases are to be considered:
  - you want to use the **reverse path** of an existing "relatedEntity" attribute. You just need to pass the "relatedEntity" attribute name (from the 'many' class) as the path in the *indexOrPath* parameter and add a {reverserPath: true} object as a 5th parameter. For example, you want to add in the *Company* datastore class a "relatedEntities" attribute named "employees" that will contain all employees working for the company. The *employer* N->1 relation attribute already exists in the *Employee* datastore class, so you can just use the reverse path to build the appropriate collection:

    ```
    emp.addAttribute("employees", "relatedEntities", "Employees", "employer", {reversePath:true});
    ```

    Setting the reverse path is necessary so that the existing foreign key of the 'many' class is used to establish the relation.
  - you want to use a **custom path** through several datastore classes and benefit from existing relations (which can be reverse paths). For example, if you have three datastore classes, *Employee - Company - City* with existing relations between *Company -> City*, and *Employee -> Company*, you can create a relation attribute in *City* based upon these existing relations in order to get the collection of employees working in the city. You could create the reverse path of the *workingPlace* attribute (see above). But, you can also decide to use a custom path for your attribute (both attributes would work the same way): create an attribute named "workForce" in *City* of the *kind* "relatedEntities" and the *type* "Employees" and pass a custom path in the *indexOrPath* parameter, for example "companies.employees" (*companies* is the 1->N relation attribute from *City* and *employees* is the 1->N relation attribute from

Employee).
In this case, you do not need to pass the "reversePath" option because the custom path already uses the reverse paths.

options

The *options* parameter is an object containing several key/value pairs allowing you to set various properties to the created storage or relation attribute.

The following properties are available for **storage** attributes:

Option	Type	Description
simpleDate	boolean	If true, the date is stored in "DD/MM/YYYY" format. Equivalent to the "Date only" Model Designer property.
scope	string	"public" (default) or "publicOnServer". A "public" attribute can be used from anywhere, while a "public on server" attribute can only be accessed from the server.
limiting_length	number	Limits the length of the text entered in the attribute. If you define the limiting length to be 10, any longer text entered will be truncated to contain 10 characters.
blob_switch_size	number	Size in bytes below which the data of the BLOB attribute will be stored within entities. For example, if you enter 30 000, a 20 KB BLOB will be stored in the entity and a 40 KB BLOB will be stored outside the entity. By default, the value is 0: all BLOB data are stored outside of entities.
outside_blob	boolean	If true, BLOB data will be stored outside of the data file. By default (false), BLOB data are stored inside the data file.
unique	boolean	If true, values entered in the attribute must be unique. If not, an error is returned.
not_null	boolean	If true, the attribute is mandatory; it cannot be null. Otherwise, an error is returned.
autosequence	boolean	For number attributes only. If true, Wakanda automatically generates a new number for each new datastore entity created following a sequence.
autogenerate	boolean	For UUID attributes only. If true, the UUID will be generated automatically by Wakanda for each new datastore entity created. If false, you must generate a valid UUID through the code.
autoComplete	boolean	For string attributes only. If true, Wakanda automatically builds a list of possible values based on existing values for the same attribute during data entry.
styled_text	boolean	If true, queries and sorts carried out in the data stored in the attribute do not take any style tags into account.
multiLine	boolean	If true, the attribute will appear by default as a multi-line widget.
readOnly	boolean	If true, the attribute value cannot be set by user editing; it can only be set through the code.
primKey	boolean	Sets the attribute as the new primary key(*).
indexKind	string	Sets the index kind for the attribute(*).
kind	string	Sets the kind for the attribute(*).
type	string	Sets the type for the attribute(*)

(*) These properties should usually be set through the **addAttribute( )** method parameters.

The following property is available for **relation** attributes:

Option	Type	Description
reversePath	boolean	If true, the relation attribute will use the reverse path of an existing relation (for more information, please refer to the RelatedEntity or relatedEntities attributes paragraph.

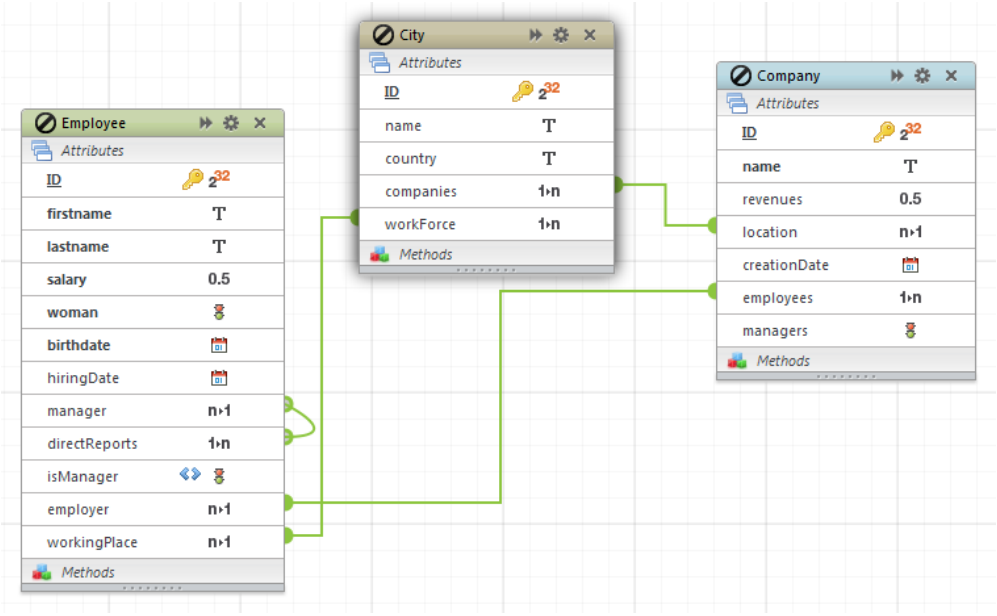
Example

In this example, we will create a complete *Employee - Company - City* model to illustrate the various ways to add attributes in your model:

```
model = new DataStoreCatalog();

    //Creating the Employee class
var emp = model.addClass("Employee", "Employees");

emp.addAttribute("ID", "storage", "long", "key auto");
emp.addAttribute("firstname", "storage", "string", "btree");
emp.addAttribute("lastname", "storage", "string", "btree");
emp.addAttribute("salary", "storage", "number", "cluster");
emp.addAttribute("woman", "storage", "bool", "cluster");
emp.addAttribute("birthdate", "storage", "date", "btree");
emp.addAttribute("hiringDate", "storage", "date");
emp.addAttribute("manager", "relatedEntity", "Employee", "Employee");
emp.addAttribute("directReports", "relatedEntities", "Employees", "manager", {reversePath:true});
emp.addAttribute("isManager", "calculated", "bool"); //onGet method is defined below
emp.addAttribute("employer", "relatedEntity", "Company", "Company"); // relation to the Company class
emp.addAttribute("workingPlace", "relatedEntity", "City", "employer.location"); // relation to the City cl

    //Creating the Company class
comp = model.addClass("Company", "Companies")
comp.addAttribute("ID", "storage", "long", "key auto");
comp.addAttribute("name", "storage", "string", "btree");
comp.addAttribute("revenues", "storage", "number");
comp.addAttribute("creationDate", "storage", "date");
comp.addAttribute("location", "relatedEntity", "City", "City");
comp.addAttribute("employees", "relatedEntities", "Employees", "employer", {reversePath:true});
comp.addAttribute("managers", "alias", "Employees", "employees.manager");

    //Creating the City class
```

```
var city = model.addClass("City", "Cities");
city.addAttribute("ID", "storage", "long", "key auto");
city.addAttribute("name", "storage", "string" , {autoComplete:true});
city.addAttribute("country", "storage", "string", {unique:true , not_null : true });
city.addAttribute("companies", "relatedEntities", "Companies", "location", { reversePath:true } );
city.addAttribute("workForce", "relatedEntities", "Employees", "workingPlace", { reversePath:true } );
    // this last relation could also have been defined like this:
    // city.addAttribute("workForce", "relatedEntities", "Employees", "companies.employees")

    //onGet for the calculated atttibute
emp.isManager.onGet = function()
{
    return this.directReports.length != 0;
}
```

You can preview the result in the Wakanda Studio Model Designer:



addEventListener( )

void **addEventListener**(String *event*, Function *jsCode*)

Parameter	Type	Description
event	String	Datastore class event to listen to
jsCode	Function	JavaScript function to execute

Description

The **addEventListener( )** method allows you to associate an event listener function with the datastore class.

***Note:** For more information about event listeners, please refer to the Using Datastore Class Events section.*

Using this method, you can define several event listeners for the same *event*.

In *event*, pass the name of the event to define. For datastore classes, available events are:

- "onInit"
- "onLoad"
- "onValidate"
- "onSave"
- "onRemove"
- "onRestrictingQuery"

In *jsCode*, pass the JavaScript function to call when the event is generated. This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it.
Keep in mind that, within the *jsCode* function, **this** represents the entity that is being processed.
Unlike an event listener method associated with an attribute (see addEventListener( ) for attributes), a datastore class event listener does not receive a parameter.

Example

We want to add a simple function on the *onRemove* event for the Company datastore class:

```
comp = model.addClass("Company", "Companies")
comp.addEventListener("onRemove", function()
{
    if (this.employees.length > 0)
        return { error : 1000, errorMessage:"The company is not empty" };
});
```

addMethod( )

void **addMethod**(String *name*, String *type*, Function *jsCode*[, String *scope*])

Parameter	Type	Description
name	String	Method name
type	String	Object to which to apply the method
jsCode	Function	JavaScript function to execute
scope	String	Scope for the method (default = "publicOnServer")

Description

The **addMethod( )** method allows you to define a datastore class method and add it to the current class.

In *name*, pass the name of the datastore class. This name must comply with the Reserved Keywords list.

In *type*, pass the object type on which the datastore class method must be applied. The following values are available:

- "entity": the method will be applied to single entities.
- "entityCollection": the method will be applied to entity collections.
- "dataClass": the method will be applied to all the entities of the datastore class.

In *jsCode*, pass the JavaScript function to execute as the datastore class method. This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it.

In *scope*, pass the scope of the datastore class method. The following values are available:

- "publicOnServer": a "public on server" datastore class method can only be invoked from the server (default if omitted).
- "public": a "public" datastore class method can be used from anywhere.

Example

We add an entity datastore class method that returns an array of entities:

```
model = new DataStoreCatalog();
var emp = model.addClass("Employee", "Employees"); // create the class
emp.addMethod("getStaff", "entity", function() {
    return this.directReports.toArray("firstname,lastname");
}, "public");
```

***Note:** See the example of the addAttribute( ) method to have an overview of the dynamic model.*

Example

We want to add the "buildData" import method to the Employee datastore class. In the Model.js file, we write:

```
model.Employee.addMethod("buildData", "dataClass", function(nbCompanies, progressRef) {
    include ("scripts/buildData.js"); // call an external script
    buildData(nbCompanies, progressRef); //execute the script with parameters
},  "public");
```

addRelatedEntities( )

void **addRelatedEntities**(String *name*, String *type*[, String *path*][, Object *option*])

Parameter	Type	Description
name	String	Name of the attribute
type	String	Type of the attribute
path	String	Relation path
option	Object	Reverse path option

Description

The **addRelatedEntities( )** method adds a new *relatedEntities* attribute to the datastore class. This method is a shortcut to the addAttribute( ) method with a predefined "relatedEntities" *kind* parameter.

In *name*, pass the name of the attribute to create. In addition to the standard Reserved Keywords, the following words are reserved in dynamic models and must not be used as attribute names:

Reserved keywords for attribute names in dynamic models
properties
methods
collectionMethods
entityMethods
events
attributes

In *type*, pass the datastore class collection name corresponding to the 'many' class. For example, in the *Employee->Company* relation, the *type* for an *employees* attribute added to the *Company* class would be "Employees" (or "EmployeeCollection" if you left the default name).

Regarding the *path* parameter, two cases are to be considered:

- you want to use the **reverse path** of an existing "relatedEntity" attribute. You just need to pass the "relatedEntity" attribute name (from the 'many' class) as path in the *indexOrPath* parameter and add a {reversePath: true} object as a 5th parameter. For example, in the

Company datastore class, you want to add a "relatedEntities" attribute named "employees" that will contain all the employees working for the company. The *employer* N->1 relation attribute already exists in the *Employee* datastore class, you can just use the reverse path to build the appropriate collection (see example).
Setting the reverse path is necessary so that the existing foreign key of the 'many' class is used to establish the relation.

- you want to use a **custom path** through several datastore classes and benefit from the existing relations (which can be reverse paths). For example, if you have three datastore classes, *Employee - Company - City*, with existing relations between *Company -> City*, and *Employee -> Company*, you can create a relation attribute in *City* based upon this existing relations to get the collection of employees working in the city. You could create the reverse path of the *workingPlace* attribute (see above). But, you can also decide to use a custom path for your attribute (both attributes would work the same way): create an attribute named "workForce" in *City* of the *kind* "relatedEntities" and the *type* "Employees" and pass a custom path in the *path* parameter, for example "companies.employees" (*companies* is the 1->N relation attribute from *City* and *employees* is the 1->N relation attribute from *Employee*).
In this case, you do not need to pass the "reversePath" option because the custom path already uses reverse paths.

The *option* parameter is an object that can contain the following property:

Option	Type	Description
reversePath	boolean	If true, the relation attribute will use the reverse path of an existing relation.

Example

```
//the following code:
emp.addRelatedEntities("employees", "Employees" , "employer" , {reversePath:true});
//is equivalent to:
emp.addAttribute("employees", "relatedEntities", "Employees", "employer", {reversePath:true});
```

addRelatedEntity( )

void **addRelatedEntity**(String *name*, String *type*[, String *path*][, Object *option*])

Parameter	Type	Description
name	String	Name of the attribute
type	String	Type of the attribute
path	String	Relation path
option	Object	Reverse path option

Description

The **addRelatedEntity( )** method adds a new *relatedEntity* attribute to the datastore class. This method is a shortcut to the **addAttribute( )** method with a predefined "relatedEntity" *kind* parameter.

In *name*, pass the name of the attribute to create. In addition to the standard **Reserved Keywords**, the following words are reserved in dynamic models and must not be used as attribute names:

Reserved keywords for attribute names in dynamic models
properties
methods
collectionMethods
entityMethods
events
attributes

In *type*, pass the datastore class name corresponding to the 'one' class. For example, in a classic *Employee->Company* relation, the type for an *employer* attribute added to the *Employee* class would be "Company".

In the *path* parameter, pass the path to the related entity:

- For simple cases in a N->1 configuration, the *path* is implicit and built upon the type. You just need to pass the relation datastore class name. In our example, it would be "Company" again. This will create a foreign key in the *Company* datastore class.
- In more complex cases, you may not want to create a foreign key but instead to use existing relations. For example, if you have three datastore classes, *Employee - Company - City*, and an existing relation between *Company* and *City*, you can create a relation attribute in *Employee* based upon this existing relation in order to get the employee's work location. In this case, you will create an attribute named "workingPlace" in *Employee* of the kind "relatedEntity" and the type "City" and pass a custom path in the *path* parameter, for example "employer.location" (*employer* is the N->1 relation attribute from *Employee* and location is the N->1 relation attribute from *Company*).

The *option* parameter is an object that can contain the following property:

Option	Type	Description
reversePath	boolean	If true, the relation attribute will use the reverse path of an existing relation.

removeAttribute( )

void **removeAttribute**(String *name*)

Parameter	Type	Description
name	String	Name of the attribute

Description

The **removeAttribute( )** method removes the *name* attribute from the datastore class for the **model** object.

Pass in *name* the name of the attribute to remove from the datastore class.

Once an attribute has been removed from the **model**, it cannot be accessed.

This method can be combined with the derived datastore class feature for optimization reasons, or to control data available to clients.

Example

You want to create a derived datastore class from the Employee class and remove some attributes from it:

```
    // Employee is an existing dataclass, it is extended
model.PublicEmp  = new DataClass("Employees", "public", "Employee") // PublicEmp is the derived dataclass
model.PublicEmp.removeAttribute("salary"); // remove some attributes
model.PublicEmp.removeAttribute("category");
```

setProperties( )

void **setProperties**(Object *properties*)

Parameter	Type	Description
properties	Object	Properties to set for the class

Description

The **setProperties( )** method allows you to define one or several properties for the datastore class. You can call the **setProperties( )** method as many times as necessary for a datastore class.

In *properties*, pass an object containing the properties you want to set in the form of *{property_string:value}*. The following properties are available:

Property name	Value type	Default	Description
publishAsJSGlobal	Boolean	False	If true, the datastore class is exposed at the global object level, e.g. "Emp"; if False (default), the datastore class is available through the datastore name, e.g. "ds.Emp" (default datastore).
allowOverrideStamp	Boolean	False	If true, an entity can be modified regardless of its internal stamp if you have also passed true for the 'overrideStamp' option to the save( ) (Datasource) and save( ) (Dataprovider) functions.
defaultTopSize	Number	40	Default top size for requests made to the server to retrieve the entities for the datastore class.
restrictingQuery	Object		{queryStatement : string} A query that restricts the entities returned for a datastore class. For better clarity, you should define this property using the setRestrictingQuery( ) method {top : number} Top number of entities returned by the restricting query.
properties	Object		{collectionName : string, scope : string , extends : string}. Additional properties. These properties can also be set using the addClass( ) method

Example

You create a new datastore class and define its properties:

```
model = new DataStoreCatalog();
var city = model.addClass("City", "Cities");
city.addAttribute("ID", "storage", "long", "key auto");
city.addAttribute("name", "storage", "string");
city.setProperties ({allowOverrideStamp : true, defaultTopSize : 50});
```

setRestrictingQuery( )

void **setRestrictingQuery**(String *queryStatement*)

Parameter	Type	Description
queryStatement	String	Restricting query for the class

Description

The **setRestrictingQuery( )** method allows you to associate a restricting query with the datastore class. A restricting query is a query that is automatically applied whenever all the entities of the datastore class are accessed. For more information about restricting queries, please refer to the Programming Restricting Queries section.

In *queryStatement*, pass a string containing the restricting query to associate with the class. The query must be written using a direct syntax (e.g. *"status == 'Manager'"*); you cannot use ":n" placeholders.

***Note**: If you want to set a custom "top" property for the query (maximum number of returned entities), you must use the setProperties( ) instead.*

Example

You want to derive a new Employee class from the Person datastore class and use a restricting query to define the Employee's default collection:

```
model = new DataStoreCatalog();
... // define the Person datastore class
var Emp = model.addClass("Employee", "Employees", "public", "Person");
```

```
Emp.setRestrictingQuery("salary isnot null");
```

DatastoreClass Constructor

DataClass( )

DatastoreClass **DataClass**([String *collectionName* [,String | Null *scope* [,String | Null *extendedClass* [,Object *properties*]]]])

Parameter	Type	Description
collectionName	String	Collection name of the datastore class
scope	String, Null	Scope of the class: "public" (default) or "publicOnServer"
extendedClass	String, Null	Parent class (if any) of the datastore class
properties	Object	Additional properties
Returns	DatastoreClass	New datastore class

Description

The **DataClass( )** method is the constructor of the *DatastoreClass* type objects. It allows you to instanciate a new datastore class in the current procedural *Model* (also called datastore catalog) of your Wakanda application. *DatastoreClass* objects are handled using the various properties and methods of the DatastoreClass class.

Basically, this constructor method provides the same effect as the addClass( ) method. However, unlike addClass( ), **DataClass( )** allows you to maintain a link between the Wakanda Studio's Model Designer and the JavaScript code. For example, if you create a calculated attribute, you will be able to display the associated JavaScript code from the Model Designer by clicking the button. By consequent, **DataClass( )** is recommended to create a datastore class.

To create a datastore class object using the **DataClass( )** constructor, you must:

- use the **new** operator to create an instance of the object,
- assign the instance to a property of the datastore **model** object; the name of the property will become the name of the datastore class. This name must comply with the general rules defined in the Reserved Keywords section.

For example, if you want to create a public datastore class named "Student", you must write:

```
model.Student = new DataClass("Students");
```

Once the datastore class is instanciated, a prototyped *DatastoreClass* object is created and benefit from the various API methods of the DatastoreClass class, such as setProperties( ).

In *collectionName*, you can pass the name of entity collection for the new datastore class. If omitted, by default the "Collection" suffix is added to *className* to get the collection name. For example, if "MyClass" is the *className*, "MyClassCollection" will automatically be defined as the collection name.

In *scope*, pass the scope of the added datastore class. Two values are accepted:

- "public": the datastore class can be accessed from anywhere.
- "publicOnServer": the datastore class can only be accessed on the server (no client-side access is allowed).

By default, if this parameter is omitted or if you pass **null**, the datastore class scope will be "public".

In *extendedClass*, pass the name of an existing datastore class from which you want the new datastore class to be derived. Pass **null** if the added datastore class should not derive from another class.

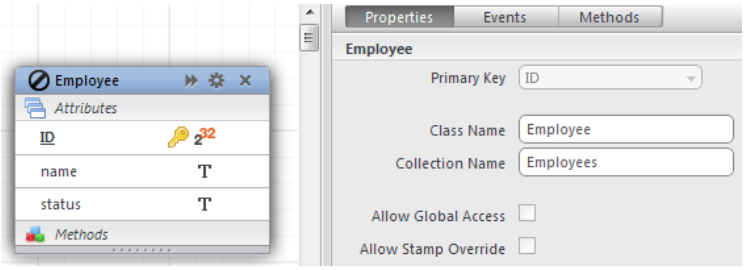
The optional *properties* parameter allows you to define any datastore class property within the **DataClass( )** call. This parameter is an object containing property/value pairs. For example, you can pass *{ allowOverrideStamp : true }* in the *properties* parameter if you want to define the **Allow Stamp Override** option for the datastore class.
Although you can set properties with the **DataClass( )** method, for better clarity it is recommended to use the setProperties( ) method to define appropriate properties. For more information about available properties, please refer to this method description.

Example

You want to create a simple Employee datastore class. In the Model.js file, you write:

```
model.Employee = new DataClass("Employees");
model.Employee.ID = new Attribute( "storage", "long", "key auto");
model.Employee.name = new Attribute( "storage", "string");
model.Employee.status = new Attribute( "storage", "string");
```

The datastore class is created and can be viewed in the Wakanda Studio's Model Designer:



Example

You want to create a Manager datastore class that is derived from the Employee class and set some properties:

```
model.Manager = new DataClass ("Managers", "public", "Employee", {
```

```
    restrictingQuery: { queryStatement: "status == 'manager'" },
    allowOverrideStamp : true
});
```

Example

You want to add a datastore class working with a resticting query:

```
model.Employee = new DataClass("EmployeeCollection" ,"public", "Person", {
    restrictingQuery: {
        "queryStatement": "employer is not null"
    }
});
```

DatastoreClass Events

The described event handler functions have to be set on a datastore class through the following property:

model.{className}**.events**

For example:

```
model.Item.events.onSave = function() {
    ds.Update.checkForUpdate(this);
}
```

onInit

Description

The **onInit** property contains the event function that is to be called just after a new entity is created in memory on the server. You can use this event to initialize attribute values, for example a custom ID.

This event is available at both the DatastoreClass and Attribute levels.

When this event is set for a datastore class and for one or more attribute(s), the calling order is as follows:

1. datastore class event
2. attribute event(s)

Example

You want to associate an *onInit* event to the "lastName" attribute:

```
model.Employee.lastName.events.onInit = function(){
    this.lastName = " "; // assigns empty string to the attribute
}
```

onLoad

Description

The **onLoad** property contains the event function to call just after an entity or an attribute is accessed.

This event is available at both the DatastoreClass and Attribute levels and is called differently depending on the level:

- on a datastore class: called each time an existing entity is loaded in memory on the server.
- on an attribute: called the first time an attribute is used (i.e., read or modified) for an entity, after this entity is loaded, and then is not called again for any subsequent times.

For more information, please refer to the Description of events and Class Events and Attribute Events sections.

Example

You want to add an *onLoad* event to the Employee class:

```
model.Employee.events.onLoad = function(){
    if (this.photo.size == 0)  //no picture for the employee
        this.photo = loadImage('c:/Projects/Docs/WebFolder/NoPhoto.jpg'); //add a default one
}
```

onValidate

Description

The **onValidate** property contains the event function to call before saving an entity on the server, for example following a call to the save( ) method. It is automatically called before the **onSave** event. It is also generated when you call the validate( ) method. You can use this event to check the validity of the data entered compared with business rules that you have set. To reject the validation, you pass a specific object as the result of the function.

This event is available at both the DatastoreClass and Attribute levels. When this event is set for a datastore class and for one or more attribute(s), the calling order is as follows:

1. attribute event(s)
2. datastore class event

For more information, please refer to the Description of events and Class Events and Attribute Events sections.

Example

You want to add the *hasAtLeastOne* function as **onValidate** event:

```
model.Article.events.onValidate = function() {
    if (this.name == null) && (this.code == null) {
        return {error: 7, errorMessage: 'name or code must have a value'};
    }
    else {
        return {error: 0}; //Same as no error
```

```
    }
}
```

onSave

Description

The **onSave** property contains the event function to call just before saving an entity on the server, for example following a call to the save( ) method and if the entity has passed validation. To reject the save, you pass a specific object as the result of the function.

This event is available at both the DatastoreClass and Attribute levels. When this event is set for a datastore class and for one or more attribute(s), the calling order is as follows:

1. datastore class event
2. attribute event(s)

When the event is set for one or more attribute(s), for better optimization, it is called only for each modified attribute. For this reason, unlike *onValidate*, the datastore class event is called before attribute events.

For more information, please refer to the Description of events and Class Events and Attribute Events sections.

Example

You want to set a datastore class method named *checkForUpdate* as the *onSave* event:

```
model.Item.events.onSave = function() {
    ds.Update.checkForUpdate(this);
}
```

onRemove

Description

The **onRemove** property contains the event function to be call just before an entity is to be deleted. It can be used for a variety of purposes including cleaning up related entities and validating the deletion. To reject the deletion, you pass a specific object as the result of the function.

This event is available at both the DatastoreClass and Attribute levels. When this event is set for a datastore class and for one or more attribute(s), the calling order is as follows:

1. attribute event(s)
2. datastore class event

Example

You want to add an *onRemove* event to check integrity:

```
model.Customer.events.onRemove = function(){
    if(this.allOrders.length > 0) { //reject the deletion
        return {errorCode: -123, errorMessage: "Cannot delete an entity with bound Orders"};
    }
}
```

onRestrictingQuery

Description

The **onRestrictingQuery** property contains the event function to call whenever all the entities in a datastore class are accessed either by locating an entity by key or by using one of these dataclass methods: all( ), query( ), or find( ).

A restricting query function **must return** a valid collection from the datastore class. If you omit to return a collection, or if the returned collection is invalid (undefined), the server will not send any entities.

This event is only available on datastore classes.

For more information, please refer to the Description of events and Class Events and Attribute Events sections.

Example

The following restricting query event automatically gives access to the contacts belonging to the currently logged in user:

```
model.Contact.events.onRestrictingQuery = function(){
    var col = ds.Contact.query("owner = :$userID");
    return col; // a collection must be returned
}
```

Method

The *DatastoreClassMethod* class contains read/write properties that allow you to get or set the properties of your datastore class methods.

DatastoreClassMethod objects can be referenced through the **model.{className}.{methodName}** property.

For example:

```
model.Product.addCountryCode.applyTo = "entity";
```

scope

Description

The scope property contains the current scope value for the datastore class method.

Two values are available:

- "public": the datastore class method can be used from anywhere.
- "publicOnServer" (default): the datastore class method can be used only from the server.

Example

You want to set to "public" a method of the Student datastore class:

```
model.Student.myFirstMethod.scope = "public";
```

applyTo

Description

The **applyTo** property contains the kind of the method, that is, the objects to which the method will be applied.

Three values are available:

- "class" (default): the method applies to all the entities stored in the datastore class
- "entity": the method applies only to an entity collections of the datastore class
- "entityCollection": the method applies only to entity collections of the datastore class

Example

You want to create a new datastore class method in the Product class and apply it to entities:

```
model.Product.addCountryCode = function(x,y) {
      this.name += x;
      this.version += y;
      this.save();
}
model.Product.addCountryCode.applyTo = "entity";
```

Model

The *Model* class contains methods allowing you to add datastore classes to a model reference. You create a model reference by using the DataStoreCatalog( ) constructor method.

{className}

Description

Each datastore class defined in the *dataStoreCatalog* object is available as a property of the **model** object.

This property is a read-write object: you can modify or even create a class using the returned reference.

Example

You can use an existing datastore **{className}** reference to add an attribute:

```
model.Company.location = new Attribute ("relatedEntity", "City", "City");
```

Example

You can create and define a new datastore class by building and assigning the corresponding object:

```
model.Job = {  // creates the Job datastore class
    properties: { collectionName: "Jobs", scope : "publicOnServer" },
    ID: { kind: "storage", type: "long", autosequence: true, primKey:true } ,
    name: { kind: "storage", type: "string"} ,
    collectionMethods : {
        method1: function() {},
        method2: function() {},
                // ...
    }
}
```

addClass( )

DatastoreClass **addClass**(String *className* [, String *collectionName*[, String | Null *scope*[, String | Null *extendedClass*[, Object *properties*]]]])

Parameter	Type	Description
className	String	Name of the datastore class to create
collectionName	String	Collection name of the datastore class
scope	String, Null	Scope of the class: "public" (default) or "publicOnServer"
extendedClass	String, Null	Parent class (if any) of the datastore class
properties	Object	Additional properties
Returns	DatastoreClass	New datastore class

Description

***Note**: This method is not recommended if you want to keep the link between the Wakanda Studio's Model Designer and your code. It is recommended to use the DataClass( ) constructor instead to create a datastore class.*

The **addClass( )** method adds a new datastore class to the current procedural model.

In *className*, pass the name of the datastore class to create. This name must comply with the general rules defined in the Reserved Keywords section.

In *collectionName*, pass the name of entity collection for the new datastore class. If omitted, by default the "Collection" suffix is added to *className* to get the collection name. For example, if "MyClass" is the *className*, "MyClassCollection" will automatically be defined as the collection name.

In *scope*, pass the scope of the added datastore class. Two values are accepted:

- "public": the datastore class can be accessed from anywhere.
- "publicOnServer": the datastore class can only be accessed on the server (no client-side access is allowed).

By default, if this parameter is omitted or if you pass **null**, the datastore class scope will be "public".

In *extendedClass*, pass the name of an existing datastore class from which you want the new datastore class to be derived. Pass **null** if the added datastore class should not derive from another class.

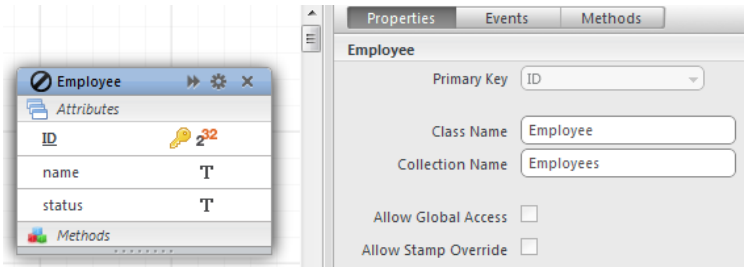
The optional *properties* parameter allows you to define any datastore class property within the **addClass( )** call. This parameter is an object containing property/value pairs. For example, you can pass *{ allowOverrideStamp : true }* in the *properties* parameter if you want to define the **Allow Stamp Override** option for the datastore class.
Although you can set properties with the **addClass( )** method, for better clarity it is recommended to use the setProperties( ) method to define appropriate properties. For more information about available properties, please refer to the setProperties( ) method description.

Example

You want to create a simple Employee datastore class. In the Model.js file, you write:

```
var emp = model.addClass("Employee", "Employees");
emp.addAttribute("ID", "storage", "long", "key auto");
emp.addAttribute("name", "storage", "string");
emp.addAttribute("status", "storage", "string");
```

The datastore class is created and can be viewed in the Model Designer:



Example

You want to create a Manager datastore class that is derived from the Employee class and set some properties:

```
var manager = model.addClass("Manager", "Managers", "public", "Employee", {
        restrictingQuery: { queryStatement: "status == 'manager'" },
        allowOverrideStamp : true
        });
```

mergeOutsideCatalog( )

void **mergeOutsideCatalog**(String *localName*, String | Object *hostName | mergeInfo* [,String *user* ,String *password*])

Parameter	Type	Description
localName	String	Local ID of the remote catalog
hostName | mergeInfo	String, Object	Address of the remote Data Server, or Object containing the connection information
user	String	User name
password	String	User password

Description

The **mergeOutsideCatalog( )** method allows you to reference and use a remote catalog in your current Wakanda *model* reference. This method allows you to:

- use a 4D database (tables, fields and project methods) within your Wakanda application,
- share catalogs between several Wakanda applications.

Note: *A* catalog *is similar to a* model, *but only refers to datastore class definitions, excluding extra or graphical properties.*

Actually, this method initializes a REST session on the remote server and allows any subsequent query (sent from the Wakanda Server) that needs access to outside data to open and use a client connection on the remote server. Of course, the remote server must have been configured to allow REST sessions (see "Connecting a 4D Database" and "Connecting to a Wakanda Server Catalog" below).

Once the remote catalog has been referenced within the current *Model* object, all its datastore classes, attributes, properties, methods... are available locally as standard objects and can be used as if they were defined in the actual model. For example, if you referenced a 'Company' datastore class referenced from an outside catalog, you can access it through the 'ds.Company' statement. You can also modify locally the outside datastore classes by creating derived datastore classes.
Data relative to the remote catalog is also available (provided access is allowed to this data); it can be handled, modified and saved on the remote server transparently.

The **mergeOutsideCatalog( )** method accepts two syntaxes:

- **direct syntax**, for basic access:

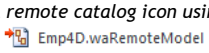
```
model.mergeOutsideCatalog(localName, hostname, user, password)
```

- **object-based syntax**, providing additional options:

```
model.mergeOutsideCatalog(localName, {
        hostname: string 'hostname',
        user: string 'userName',
        password: string 'password',
        jsFile: string 'configuration JavaScript file',
        timeout:number 4D session timeout } )
```

In *localName*, pass a name to identify the remote catalog in your project. This name is used locally for information, it is not called in the JavaScript code. It can be used to define the .js customization file is the *jsFile* property is omitted (see below). Also, if you use Wakanda Studio, the *localName* is used to display the remote catalog reference in the Explorer:

A remote catalog icon using the local name in Wakanda Studio:



Whatever the syntax you used, you need to enter the following parameters; either directly, either as object properties:

- In *hostName*, pass the address of the remote server (4D or Wakanda) from which you want to replicate the catalog. It can be a hostname (i.e. "http://www.mySharedApp") or an IP address with a port (i.e. "http://123.45.67.89:7070"). Secured connections (https) are supported and advised for applications in production.

- In *user* and *password*, pass the user name and password required to login to the remote server for opening a REST session. If the server REST access is not protected, you can just pass empty strings ("").
  Note that these ids will be used to initialize and use the REST session on the server and that any subsequent request will be executed with the attached access rights. You do not control privileges individually for each user, it is shared by all the threads connecting to the server. Consequently, you need to control the data exposed at the remote server level and/or use Wakanda local features such as object scope,

derived datastore classes orcalculated attributes to control data available to each user.

- *jsFile* (object-based syntax only): you can pass in the *jsFile* property the relative path name of a JavaScript file located in the same folder as the model file, for example "custom.js". This file can contain JavaScript code to modify the local reference to the outside model for customizing, optimizing or security needs. The file is executed by Wakanda just after the outside catalog is merged. Within this file, you can modify attribute properties of the datastore class, such as events or scope, add calculated and alias attributes, remove attributes or create local datastore class derived from tables in the external catalog (see examples).

***Note:** By default, if you omit the jsFile property or use the direct syntax, Wakanda will automatically look for a file with same name as the localName and with the .js suffix. For example, si the local catalog name is Emp4D.waRemoteModel, Wakanda will automatically use Emp4D.js if it exists.*

- *timeout* (object-based syntax only): timeout of client connections on the 4D server (pass a number expressing minutes). Each client query that requires a REST access to the external 4D database will create or use a client connection (process) on the 4D server side, keeping the client context. By default, the connection is closed after 60 minutes of inactivity. You can reduce this timeout up to 15 minutes, depending on your needs.

Connecting to a 4D Database

In order a Wakanda application to be able to connect to a 4D database and to use its tables, fields and methods, you need to have the appropriate configuration on the 4D side:

- You must use at least a v14 version of 4D Server or 4D (professional edition).
- The 4D Web Server and the REST services must be started and responding on the defined TCP port (in case of trouble, pay attention to the fact that some third-party applications such as IMs may use the default HTTP port 80).
- REST session opening queries must be filtered by assigning a 4D group to the REST access or by using the On REST authentication database method (optional)
- You must set appropriate property to each table, field or project method that you want to be exposed by REST. By default, tables and fields are exposed, methods are not exposed. You also need to declare the attached table and the scope for each exposed project method, so that it can be handled through JavaScript.

For more information about how to configure a 4D database, please refer to the 4D documentation.

Connecting to a Wakanda Server Catalog

There are no specific REST access rights on a Wakanda Server. A **mergeOutsideCatalog( )** login request is handled as a standard login request. User and password values must be validated by the Wakanda Server to allow access to the catalog.

Only 'public' classes and attributes from the outside catalog can be seen: if you do not want to share some attributes from a datastore class, set their scope to 'publicOnServer'. If you set a datastore class scope to "publicOnServer", none of its attributes will be shared, regardless of their own scope.

Example

You want to merge your catalog with a table from a 4D database and modify some attributes. In 4D, the Students table looks like:

Or course, the database is configured to accept REST queries (Web Server and REST services are launched and the table is exposed).

- In the Model.js file, you write:

```
model.mergeOutsideCatalog("my4DBase", {
    hostname: "192.168.90.94:80, //address of the the 4D database Web server
    user: "admin", // user and password must be evaluated on the 4D side using either
    password: "nimda", // REST Access property or On REST Authentication db method}
);
```

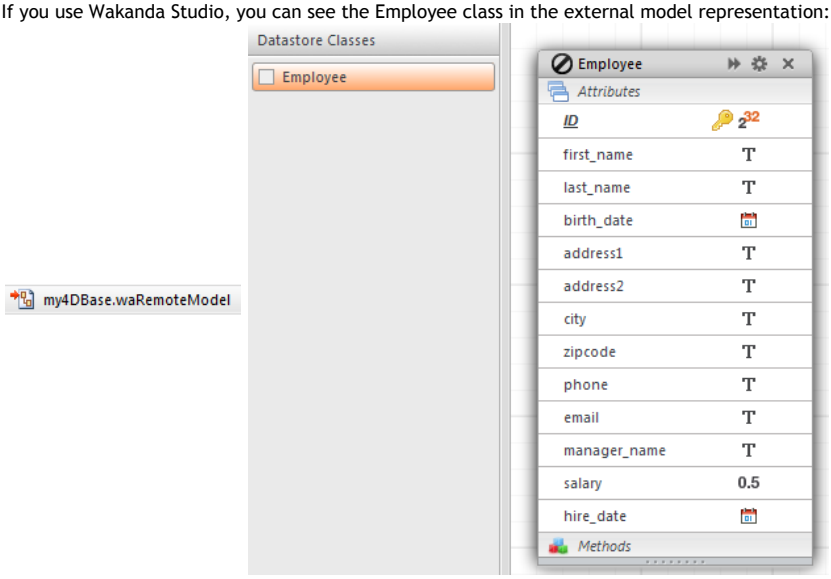
- When the solution is started, you are able to call the [Employee] table as a standard datastore class and thus, as a datasource. You can check that the Employee datastore class is published using the basic **$catalog** Rest query:

```
http://127.0.0.1:8081/rest/$catalog/
```

It should return a result such as:

```
{

    "dataClasses": [
        {
            "name": "WakandaLocalClass",
            "uri": "/rest/$catalog/WakandaLocalClass",
            "dataURI": "/rest/WakandaLocalClass"
        },
        {
            "name": "Employee",
            "uri": "/rest/$catalog/Employee",
            "dataURI": "/rest/Employee"
        }
    ]

}
```

If you use Wakanda Studio, you can see the Employee class in the external model representation:



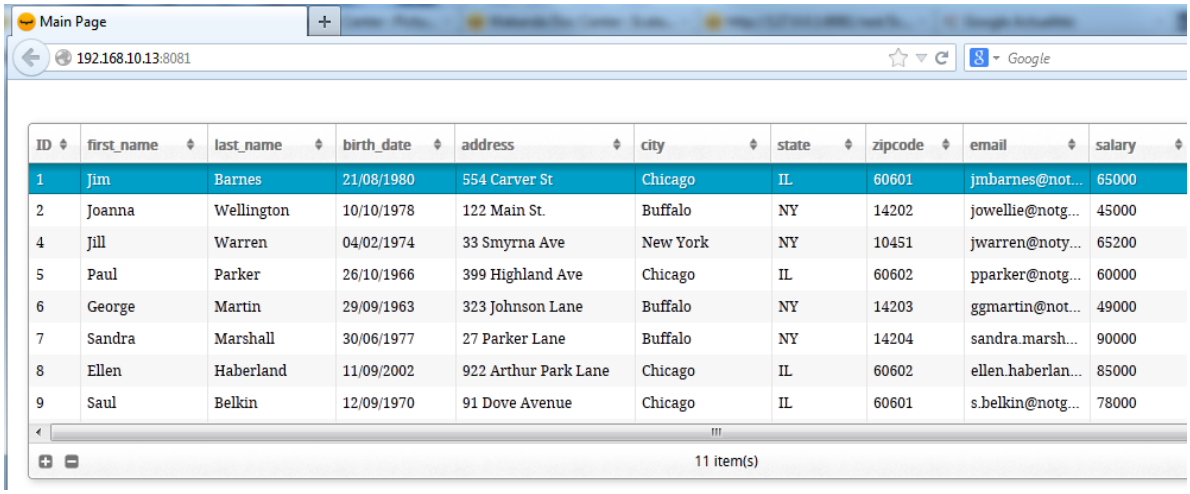
- Then on the server side, you can execute for example:

```
var ds.Employee.all(); //select all records from the 4D table
```

- You can also create a datasource over the datastore class and display 4D data on the Web:
4D side:

ID	first_name :	last_name :	birth_date :	address1 :	address2 :	city :	zipcode :	email :	salary :
1	Jim	Barnes	21/08/1980	554 Carver St	IL	Chicago	60601	jmbarnes@notyahoo.com	65000
2	Joanna	Wellington	10/10/1978	122 Main St.	NY	Buffalo	14202	jowellie@notgmail.com	45000
4	Jill	Warren	04/02/1974	33 Smyrna Ave	NY	New York	10451	jwarren@notyahoo.com	65200
5	Paul	Parker	26/10/1966	399 Highland Ave	IL	Chicago	60602	pparker@notgmail.com	60000
6	George	Martin	29/09/1963	323 Johnson Lane	NY	Buffalo	14203	ggmartin@notyahoo.com	49000
7	Sandra	Marshall	30/06/1977	27 Parker Lane	NY	Buffalo	14204	sandra.marshall65@notgmail.com	90000
8	Ellen	Haberland	11/09/2002	922 Arthur Park Lane	IL	Chicago	60602	ellen.haberland@notyahoo.com	85000
9	Saul	Belkin	12/09/1970	91 Dove Avenue	IL	Chicago	60601	s.belkin@notgmail.com	78000

Wakanda side:



Example

Using the same 4D database and Wakanda solution as above, you want to customize the available information on the Wakanda side:

- you want to create a datastore class derived from Employee in order to have full control over the published information
- for performance or security reasons, you want to remove some attributes from the derived class and add a calculated attribute.

To do this, you will use the additional *jsFile* parameter in the **mergeOutsideCatalog( )** call. In this file, you will add JavaScript code to execute after the outside model is called.

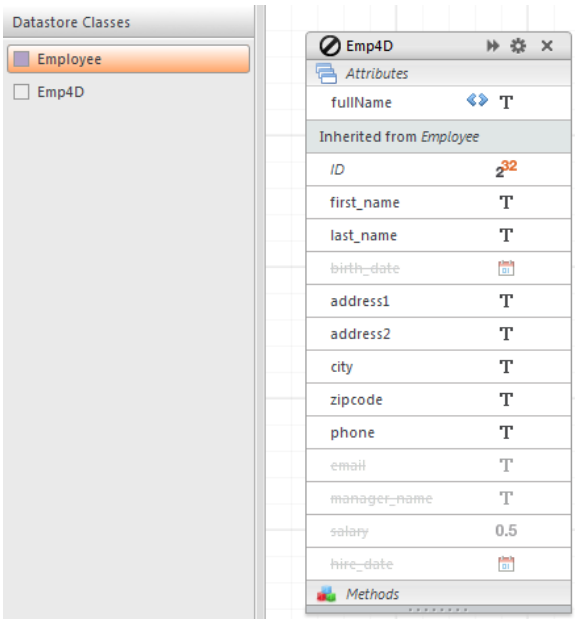
Write the following code at the end of the Model.js file:

```
model.mergeOutsideCatalog("my4DBase", {
    hostname: "192.168.90.94:80, //address of the the 4D database Web server
    user: "admin",
    password: "nimda",
    jsFile: "my4DBase.js", // Name of the file where local model is customized
    timeout: 15}
);
```

Then, in the my4DBase.js, located in the same folder as the model file, you can write:

```
    //Restricting the scope of the Employee extended class
model.Employee.scope="publicOnServer";

    // Creating the extended datastore class
model.Emp4D = new DataClass("Employees", "public", "Employee")

    //Adding a calculated attribute
model.Emp4D.fullName = new Attribute("calculated","string");

model.Emp4D.fullName.onGet=function(){  //onGet function, based on the extended data class
    return this.first_name+" "+this.last_name;
}
model.Emp4D.fullName.onSet=function(value){ //onSet function, based on the extended data class
    var names = value.split(' '); //split value into an array
    this.first_name= names[0];
    this.last_name= names[1];
}

    //Removing unwanted attributes
model.Emp4D.removeAttribute("birth_date");
model.Emp4D.removeAttribute("email");
model.Emp4D.removeAttribute("manager_name");
model.Emp4D.removeAttribute("salary");
model.Emp4D.removeAttribute("hire_date");
```

After you restart the solution, if you use Wakanda Studio, you can see the derived class in the model editor:



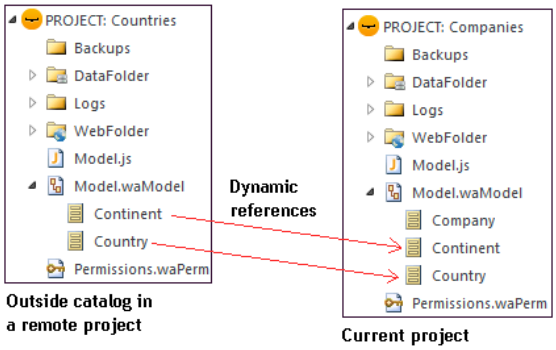
You can use the new "Emp4D" derived datastore class in your application, for exemple through a datasource:

ID	fullName	address1	state	city	zipcode
1	Jim Barnes	554 Carver St	IL	Chicago	60601
2	Joanna Wellington	122 Main St.	NY	Buffalo	14202
4	Jill Warren	33 Smyrna Ave	NY	New York	10451
5	Paul Parker	399 Highland Ave	IL	Chicago	60602
6	George Martin	323 Johnson Lane	NY	Buffalo	14203
7	Sandra Marshall	27 Parker Lane	NY	Buffalo	14204
8	Ellen Haberland	922 Arthur Park Lane	IL	Chicago	60602
9	Saul Belkin	91 Dove Avenue	IL	Chicago	60601

12 item(s)

Example

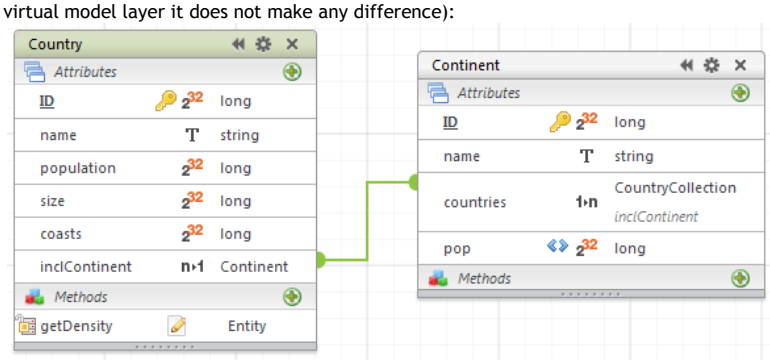
You have two projects in two different Wakanda solutions, named *Companies* and *Countries*. You want to reference the *Countries* catalog from the *Companies* model.



- The *Companies* project has a procedurally based model, defined in the "Model.js" file. To keep it simple, we just create a single datastore class with basic attributes:

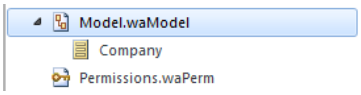
```
var comp = model.addClass("Company", "Companies");
comp.addAttribute("ID", "storage", "long", "key auto");
comp.addAttribute("name", "storage", "string");
comp.addAttribute("country", "storage", "string");
```

- The *Countries* project has Studio-based model defined in the Model Designer (it could be procedurally based, thanks to the Wakanda virtual model layer it does not make any difference):



This project is published by another Wakanda Server on port 8082.

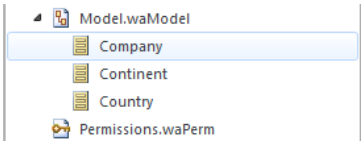
At this point, the *Companies* project model only contains a single datastore class:



To reference the *Countries* catalog, you just need to add to the *Companies* Model.js file:

```
model.mergeOutsideCatalog("addCountries","http://123.45.67.89:8082"); // address of the Countries project
    // addCountries is an arbitrary name, it is not used
```

Then, the *Companies* model will reference the *Countries* datastore classes:



Then, for example, you could call from within the *Companies* project:

```
var coll = ds.Company.query("country == :1", ds.Country(3).name);
    // access to the Countries catalog and data
    // from the Companies project
```

Outside classes are also available in the GUI designer as standard classes; you can create datasources on them: