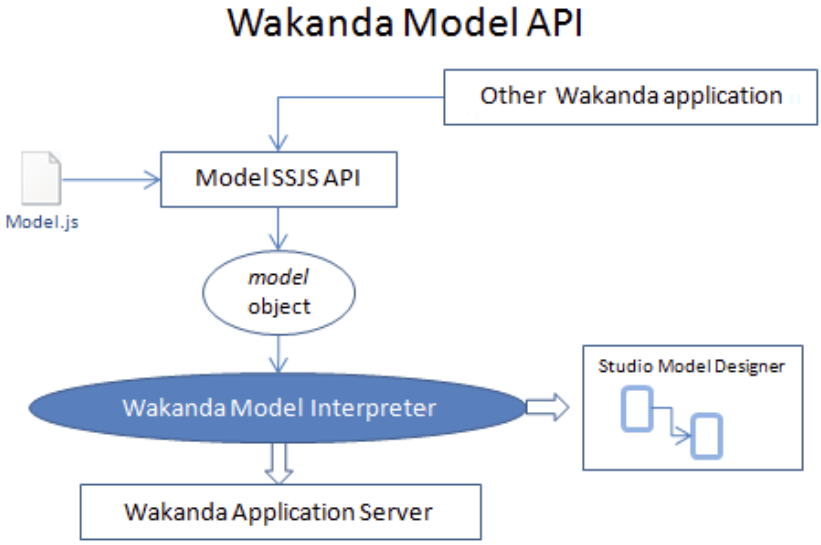
Model

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 addClass( ) and addAttribute( ) methods to define your datastore classes and attributes. 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 activate the Wakanda procedural model generation, you just need to define a **model** global object in the **Model.js** file, either by using the DataStoreCatalog( ) model constructor method or by simple assignment. For more information, please refer to the Working with the Model API section.

Working with the Model API

Initializing the model Object

The *model* object contains the JavaScript model description. It must be defined in the "Model.js" file stored at the root of your Wakanda project. You can use **include( )** statements to define modules to load depending on your needs. When this object is found, Wakanda activates the procedural model mode and uses it to build the datastore model based on JavaScript statements.

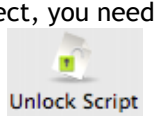
To initialize this object, you need to:

- activate the **Free form edition mode**,
- execute the **DataStoreCatalog( )** constructor method.

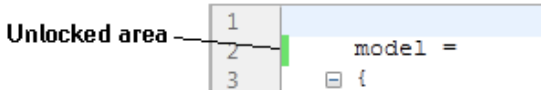
Activating the Free Form Edition Mode

The Free form edition mode must be set explicitely in your model (for more information, please refer to the **Using the Free Form Edition Mode** section). By default, the Model.js file is set to **guided model** mode:

```
1
2       guidedModel =
3     {
4     };
```

To be able to write the initialization code for the *model* JavaScript object, you need to unlock and remove the *guidedModel* object definition. To do this, just click on the 

button in the Datastore Model Designer toolbar. A warning dialog box is displayed because this action cannot be undone (see below). Click **OK** to switch your application model to free form edition mode. The "Model.js" file is then unlocked and the *model* object is defined:



Warning:

- *Activating the Free form edition mode (also named "unguided model") will disable the automatic create/remove mechanisms between the Wakanda Model Designer and the code editor and cannot be reverted. For more information, please refer to the **Using the Free Form Edition Mode** section.*
- *When you switch to the Free form edition mode in an existing project, previous datastore class definitions are not converted to JavaScript-based classes in the Model.js file. They are still available and can be used, but must be handled through the Model Designer only.*

Calling the model Constructor

Executing a **new DataStoreCatalog( )** call in the "Model.js" file is necessary to instantiate properly the model object. Once the *model* object is instantiated, you can call the various methods and properties of the **Model** API to build and control your model definition.

Compatitbility Note: *In Wakanda versions prior to v4, it was possible to handle the* model *object through direct assignments, without instantiation (see the **Free Form Syntax** paragraph). This syntax is now deprecated and should not be used anymore.*

Viewing a Procedural Model in Wakanda Studio

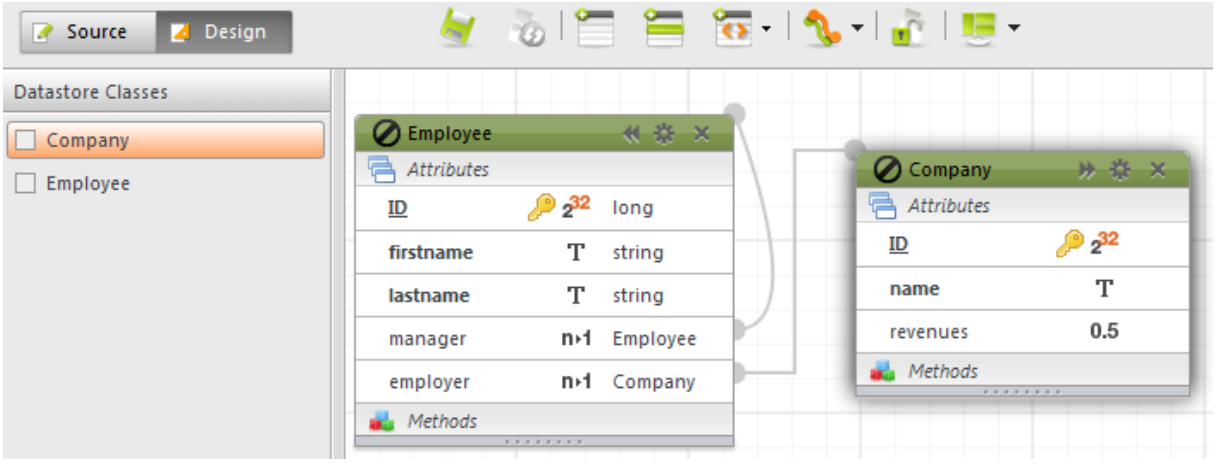
Procedurally built models can be viewed in the Wakanda Studio Model Designer. This feature allows

you to have a visual representation of your datastore classes as well as 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 = new DataStoreCatalog();

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("manager", "relatedEntity", "Employee", "Employee");
emp.addAttribute("employer", "relatedEntity", "Company", "Company");

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

Procedural datastore classes cannot be edited through the Model designer. The locked icon Employee shows that datastore classes are generated from the *model* JavaScrip object. Also, the **Source** button of the Model designer will only give access to extra properties of the model (json format).

About Hybrid Models

It is recommended to switch to Free form edition mode in new blank projects, where you can freely build your model and define your code using the Model API.

Switching to Free form edition mode in existing projects is not recommended. Such operation will 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 can bring some complexity for maintenance operations.

When you switch to Free form edition mode in existing projects, as explained above, no conversion is performed on existing datastore classes and attributes: they are still described in JSON (or XML in older projects) in the "Model.waModel" file. They have to be edited and maintained through the

Wakanda Studio's Model Designer, while procedurally-based datastore classes can be viewed in the Model Designer but have to be edited and maintained using the **Model** API.

Also, existing associated JavaScript code (calculated attributes, event and methods) is converted as standard JavaScript properties of the *model* object, which is a deprecated syntax (see **Free Form Syntax**):

```
model =
{
    Employee :
    {
        methods :
        {
            addEmp:function(fName, lName, compID)
            {
                var newEmployee = new ds.Employee ({
                    firstName : fName,
                    lastName : lName,
                    company : ds.Company.find("ID = :1", compID)
                });
                newEmployee.save();
                return newEmployee;
            }
        },
        fullName :
        {
            onSet:function(value)
            {
                var names = value.split(' ');
                this.firstName = names[0];
                this.lastName = names[1];
            },
            onGet:function()
            {
                return this.firstName + " " + this.lastName;
            }
        }
    }
};
```

Model.js converted to free form edition mode (deprecated syntax).

Working with the Model API (v5)

Initializing the model Object

The *model* object contains the JavaScript model description. It must be defined in the "Model.js" file stored at the root of your Wakanda project. You can use **include( )** statements to define modules to load depending on your needs. When this object is found, Wakanda activates the procedural model mode and uses it to build the datastore model based on JavaScript statements.

To initialize this object, you need to:

- switch to the **Advanced mode** in the Model Designer,
- execute the **DataStoreCatalog( )** constructor method.

Activating the Advanced Mode

The Advanced edition mode must be set explicitly in your model (for more information, please refer to the **Using the Free Form Edition Mode** section). By default, the Model.js file is set to **guided model** mode, where only the methods and events related to the model are stored (your model definition is stored in a separate JSON file):

```
guidedModel =
{
    Company :
    {
        collectionMethods :
```

To be able to write the initialization code for the JavaScript *model* object, you need to unlock and remove the *guidedModel* object definition. To do this, just click on the **Go to advanced model** button  in the Datastore Model Designer toolbar. You must confirm the warning that appears because this action cannot be undone.

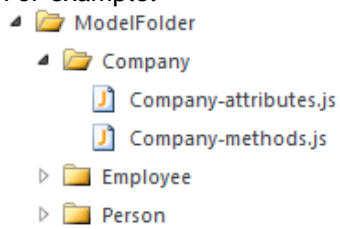
Switching to the advanced model permanently removes the guided model from your project. Click **OK** to switch your application model to advanced edition mode. The "Model.js" file is then unlocked, the *model* object is defined and **include( )** statements are inserted to reference converted datastore classes:

```
// converted from guided model

model = new DataStoreCatalog();

include("ModelFolder/Company/Company-attributes.js");
include("ModelFolder/Company/Company-methods.js");
```

Existing datastore classes and attributes are automatically converted using the following mechanism:

- A "ModelFolder" is created at the root of the project.
- A "dataClassName" subfolder is added for each existing datastore class
- Each "dataClassName" subfolder contains two files:
  - "*dataClassName*-attributes.js": JavaScript code creating the datastore class, attributes and events
  - "*dataClassName*-methods.js": JavaScript code gathering datastore class methods

For example:



Note that in the advanced mode, there are no longer any code locked areas. You can store your code wherever you want. However, you can still have a link between the Datastore Model Designer and your code (see below).

Calling the model Constructor

You must execute a **new** DataStoreCatalog( ) call in the "Model.js" file in order to instantiate the model object properly. Once the *model* object is instantiated, you can call the various methods and properties of the Model API to build and control your model definition.

***Compatibility Note**: In Wakanda versions prior to v4, it was possible to handle the* model *object through direct assignments, without instantiation (see the **Free Form Syntax** paragraph). This syntax is now deprecated and should no longer be used.*

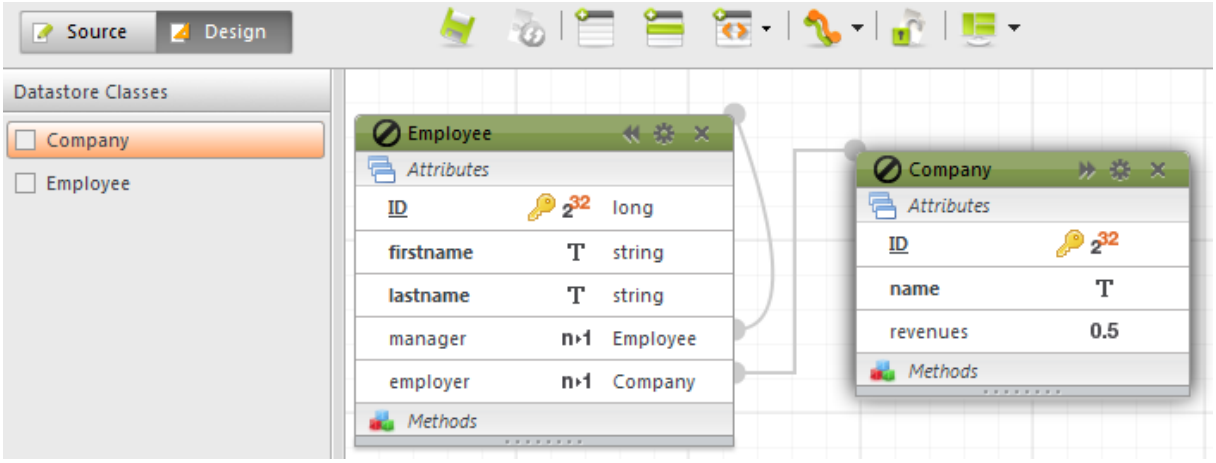
Viewing a Procedural Model in Wakanda Studio

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 = new DataStoreCatalog();

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", "Employe
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");
```

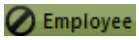
When you display the *Model.waModel* file in the Model Designer, you can see the following representation of the model:



***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.

Using the Model Designer's Compliant Syntax

In previous Wakanda versions, several methods such as addClass( ), addMethod( ) or onGet( ) were provided for 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 deprecated, particularly if you want to use the Model Designer in conjunction with your procedural model declarations. When you use 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:

```
    //Deprecated code
    //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:

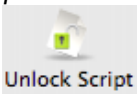
```
model.className.attributeName.events.onInit = function()
model.className.attributeName.events.onLoad = function()
model.className.attributeName.events.onSet = function()
model.className.attributeName.events.onValidate = function()
model.className.attributeName.events.onSave = function()
model.className.attributeName.events.onRemove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

Attribute

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

Attribute model objects can be created by the **addAttribute( )**, **addRelatedEntities( )** and **addRelatedEntity( )** methods or referenced through the **model.datastoreClass.{attributeName}** property.

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.

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(

    // 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'
    }
}
```

onGet( )

void **onGet**(Function *jsCode*)

Parameter	Type	Description
jsCode	Function	JavaScript function to execute

Description

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

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

In *jsCode*, pass the JavaScript function to call 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 *jsCode* function, **this** represents the entity that is being processed.

Example

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

```
var emp = model.addClass("Employee", "Employees");
emp.addAttribute("hiringDate", "storage", "date");
emp.addAttribute("hired", "calculated", "bool");
    // onGet function
emp.hired.onGet = function()
{
    return this.hiringDate != null;
}
```

onQuery( )

void **onQuery**(Function *jsCode*)

Parameter	Type	Description
jsCode	Function	JavaScript function to execute

Description

The **onQuery( )** method allows you to define the JavaScript function to execute when the calculated *Attribute* is used in a query. This method is only available for calculated attributes. For more

information on calculated attributes, please refer to the **Calculated Attribute** paragraph.

The **onQuery( )** method should actually return a string that represents a redirected query. In *jsCode*, pass the JavaScript function to call when the calculated attribute is queried. This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it.
The *jsCode* function will receive two parameters: the comparison operator (e.g. “>”, “>=”, etc.) and the compared value.

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:

```
var emp = model.addClass("Employee", "Employees");
emp.addAttribute("hiringDate", "storage", "date");
emp.addAttribute("hired", "calculated", "bool");
    // onGet function
emp.hired.onGet = function()
{
    return this.hiringDate != null;
}
    // onQuery function
emp.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";
}
```

onSet( )

void **onSet**(Function *jsCode*)

Parameter	Type	Description
jsCode	Function	JavaScript function to execute

Description

The **onSet( )** method allows you to define the JavaScript function that describes how a value entered in the calculated *Attribute* will be processed. This method is only available for calculated attributes. For more information on calculated attributes, please refer to the **Calculated Attribute** paragraph.

The **onSet( )** method is usually associated with enterable calculated attributes. In *jsCode*, pass the JavaScript function to call 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 *jsCode* 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:

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

onSort( )

void **onSort**(Function *jsCode*)

Parameter	Type	Description
jsCode	Function	JavaScript function to execute

Description

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

The **onSort( )** method must return a string that will be subsituted during the sort operation. In *jsCode*, pass the JavaScript function to call when the calculated attribute is sorted. This code will not be invoked (i.e., executed) when the Model.js file is interpreted, but stored with a pointer to it. The *jsCode* function will receive a value as parameter, which is the sort order. Keep in mind that, within the *jsCode* function, **this** represents the entity that is being processed.

Example

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

```
var emp = model.addClass("Employee", "Employees");
emp.addAttribute("birthdate", "storage", "date", "btree");
emp.addAttribute("age", "calculated", "long");
    // onGet function
emp.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
emp.age.onSort = function(ascending)
{
    if (ascending)
```

```
        return "birthdate";
    else
        return "birthdate desc";
}
```

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 or to designate the primary key for a storage attribute (it is ignored for calculated or alias attributes). 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.*

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", "empl
    ```

    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 entitty. 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.

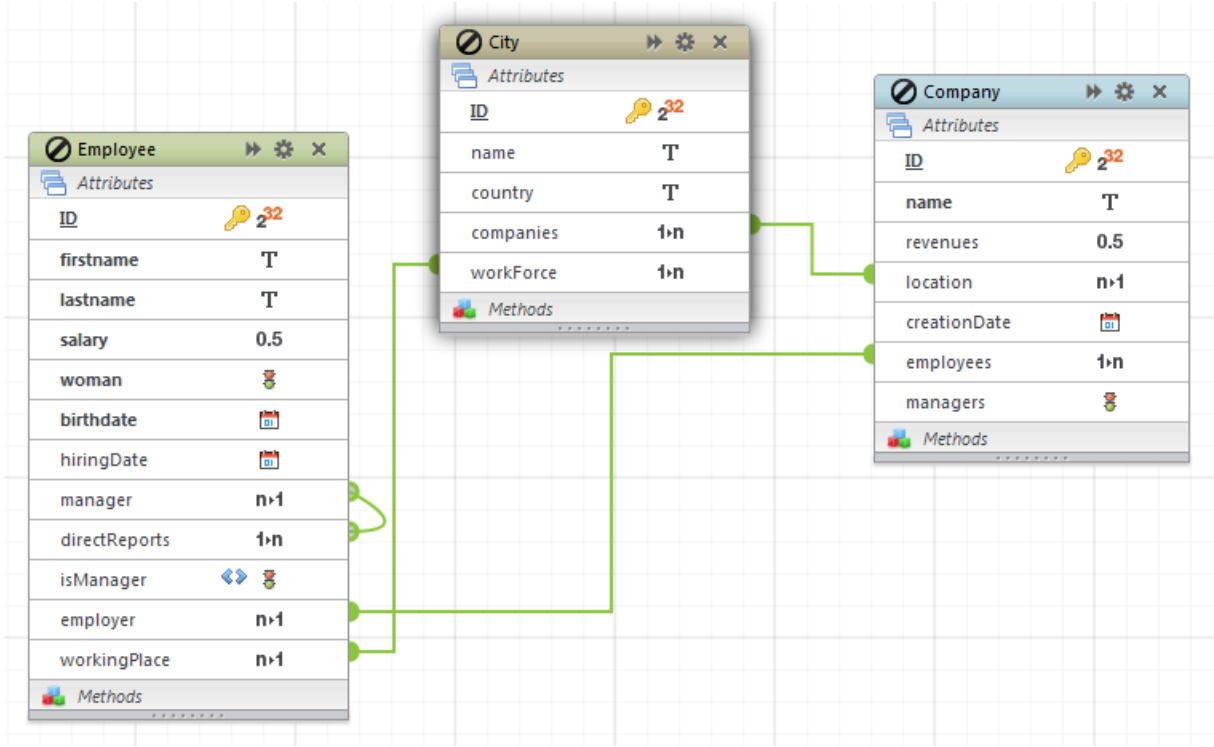
Example

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

```
model = new DataStoreCatalog();

    //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", "Employe
model.Employee.directReports = new Attribute("relatedEntities", "Employees",
model.Employee.isManager = new Attribute("calculated", "bool"); //onGet meth
model.Employee.employer = new Attribute("relatedEntity", "Company", "Company
model.Employee.workingPlace = new Attribute("relatedEntity", "City", "employ

//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", "emp
model.Company.managers = new Attribute("alias", "Employees", "employees.mana

    //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_
model.City.companies = new Attribute("relatedEntities", "Companies", "locati
model.City.workForce = new Attribute("relatedEntities", "Employees", "workin
    // 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"
    }
});
```

DatastoreClass

The *DatastoreClass* class contains methods allowing you to add attributes and properties to the datastore class objects in your model reference. *DatastoreClass* objects can be created by the **addClass( )** method or referenced through the **model.{className}** property.

{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 attribu

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

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 or to designate the primary key for a storage attribute (it is ignored for calculated or alias attributes). 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.*

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 *Employe*e 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", "empl
    ```

    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 entitty. 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:

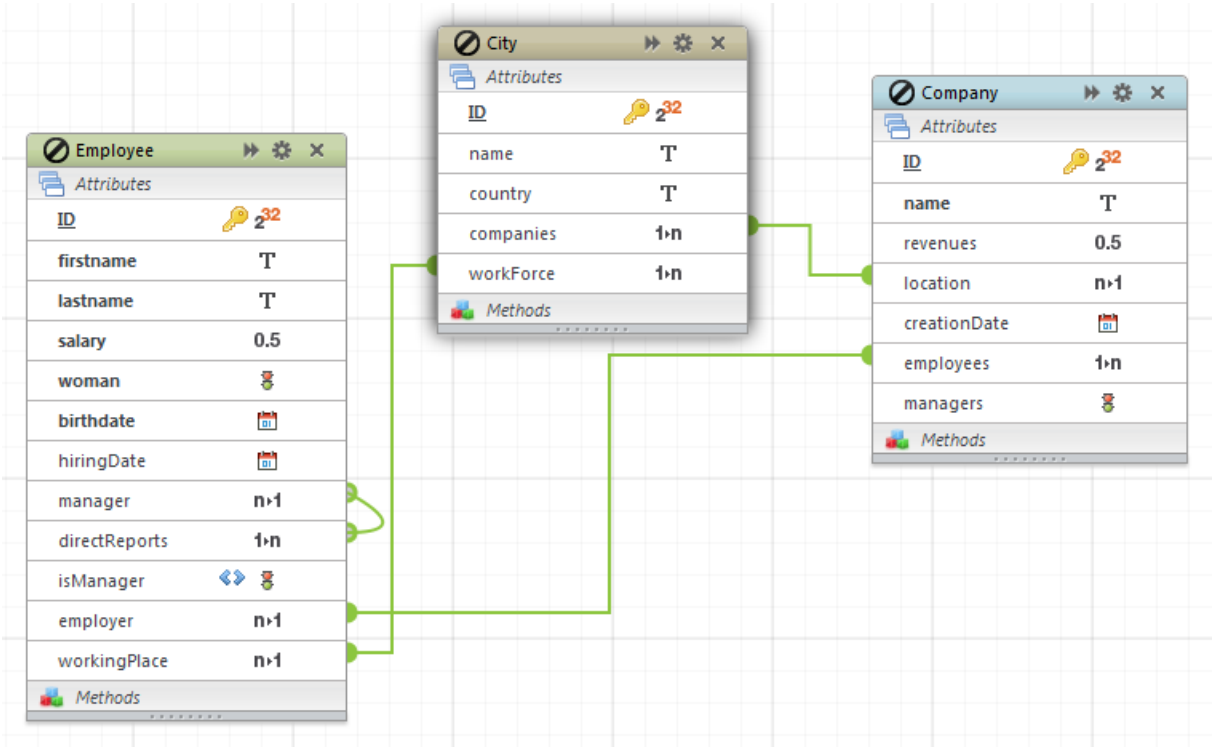
Option	Type	Description
reversePath	boolean	If true, the relation attribute will use the reverse path of an existing relation (for more information, please refer to the **RelatedEntity or relatedEntities attributes** paragraph.

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",
emp.addAttribute("isManager", "calculated", "bool"); //onGet method is define
emp.addAttribute("employer", "relatedEntity", "Company", "Company"); // rela
emp.addAttribute("workingPlace", "relatedEntity", "City", "employer.location

    //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", .
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 :
city.addAttribute("companies", "relatedEntities", "Companies", "location", .
city.addAttribute("workForce", "relatedEntities", "Employees", "workingPlace
    // this last relation could also have been defined like this:
    // city.addAttribute("workForce", "relatedEntities", "Employees", "compa

    //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, pr
    include ("scripts/buildData.js"); // call an external script
```

```
    buildData(nbCompanies, progressRef); //execute the script with parameter
},  "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:
//is equivalent to:
emp.addAttribute("employees", "relatedEntities", "Employees", "employer", {r
```

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.

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

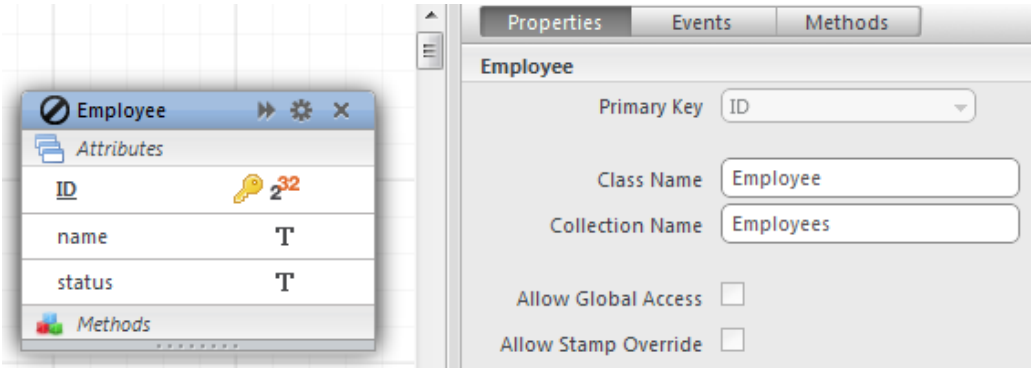
use the setProperties( ) method to define appropriate properties. For more information about available properties, please refer to this method description.

Example

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

```
model = new DataStoreCatalog();
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 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"
    }
});
```

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.

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

Example

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

```
model.Company.addAttribute("location", "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

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.

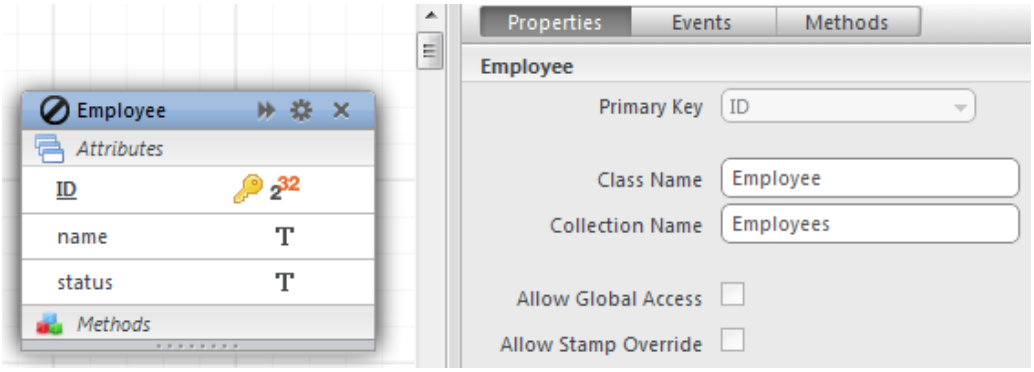
The optional *properties* parameter allows you to define any datastore class property within the **addClass( )** call. This parameter is an object containing property/value pairs. For example, you can pass *{ allowOverrideStamp : true }* in the *properties* parameter if you want to define the **Allow 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:

```
model = new DataStoreCatalog();
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
        });
```

Model Constructor

DataStoreCatalog( )

Model **DataStoreCatalog**( )

Returns	Model	Model for the application

Description

The **DataStoreCatalog( )** method is the constructor of the *Model* type objects. It allows you to create a model procedurally (also called a datastore catalog) for your Wakanda application. Model objects are handled using the **Model** class.

To activate the procedural model mode and create the *model* object, you must:

- call this method in the "Model.js" file located at the root of your Wakanda project,
- reference a new global object named "model",
- use the **new** operator to create an instance of the object.

Once your model is instantiated with this method, it is automatically loaded at Wakanda server launch and its contents, including datastore class methods, are available in all JavaScript contexts. Note also that the model can also be loaded in Wakanda Studio's Model Designer.

***Note**: You can initialize a JavaScript* model *object in your Wakanda application by creating a standard JavaScript global object named "**model**" in the Model.js file, for example:*

```
model = {}; //creates a dynamic model
```

*This code activates the procedural model mode. However, when using the **DataStoreCatalog( )** constructor, you create a prototyped Model object and benefit from the various API methods of the Model class, such as **addClass( )**.*

Example

You want to initialize a model in your project and call a model definition that is defined in other external files. You write the following code in the *Model.js* file at the root of your project:

```
model = new DataStoreCatalog(); //initializes model instance in the project
include("classes/emp.js"); //file containing Employee class description
include("classes/comp.js"); //file containing Company class description
```