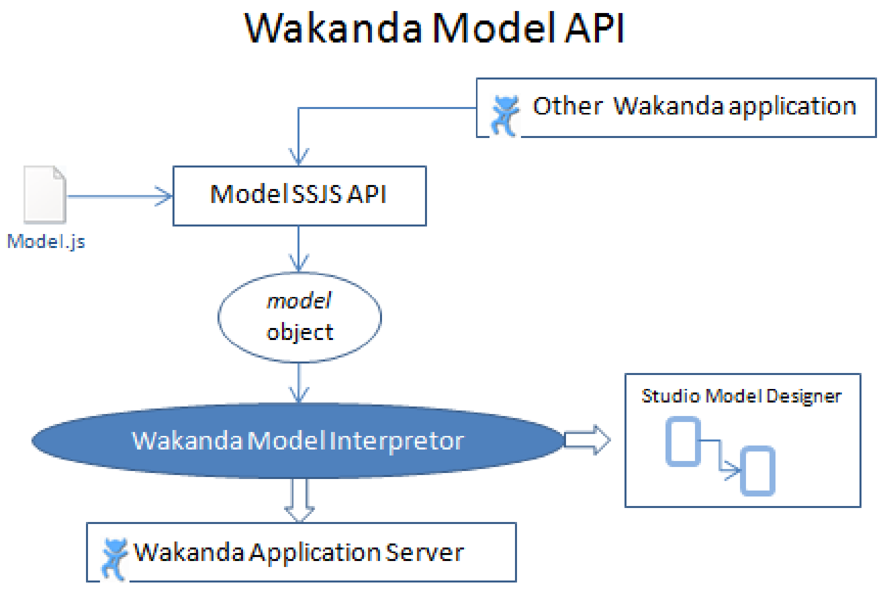
Model

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:


Wakanda Model API
Other Wakanda application
Model SSJS API
Model.js
model object
Wakanda Model Interpretor
Studio Model Designer
Wakanda Application Server

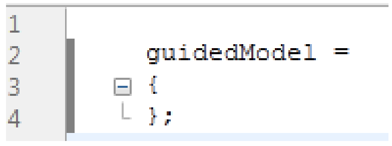
To activate the Wakanda procedural model generation, you just need to define a **model** global object in the **Model.js** file, either by using the **DataStoreCatalog( )** model constructor method or by simple assignment. For more information, please refer to the **Working with the Model API** section.

Working with the Model API

Initializing the model Object

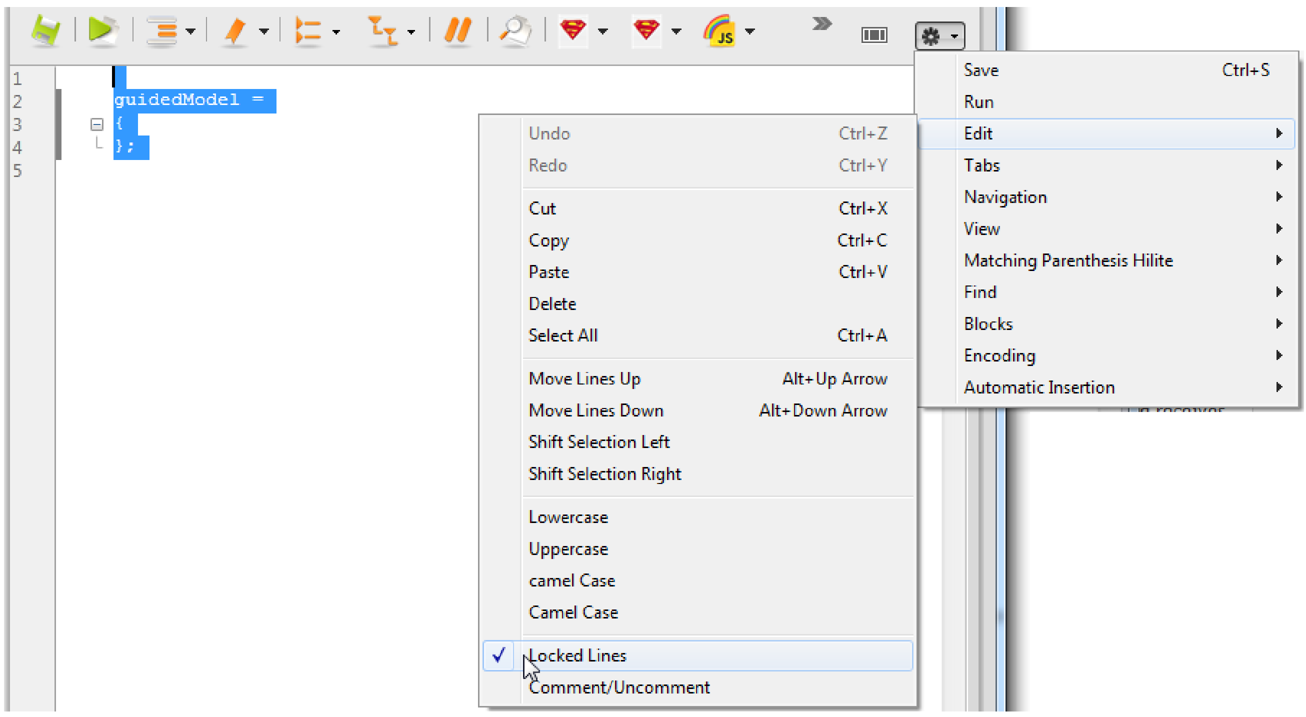
The **model** object contains the JavaScript model description. It must be defined in the **Model.js** file stored at the root of your Wakanda project. You can use **include( )** statements to define modules to load depending on your needs. When this object is found, Wakanda activates the procedural 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. This mode has to be set explicitly to the **Model.js** file. By default, the Model.js file is set to Guided model mode:



```
  guidedModel =
{
};
```

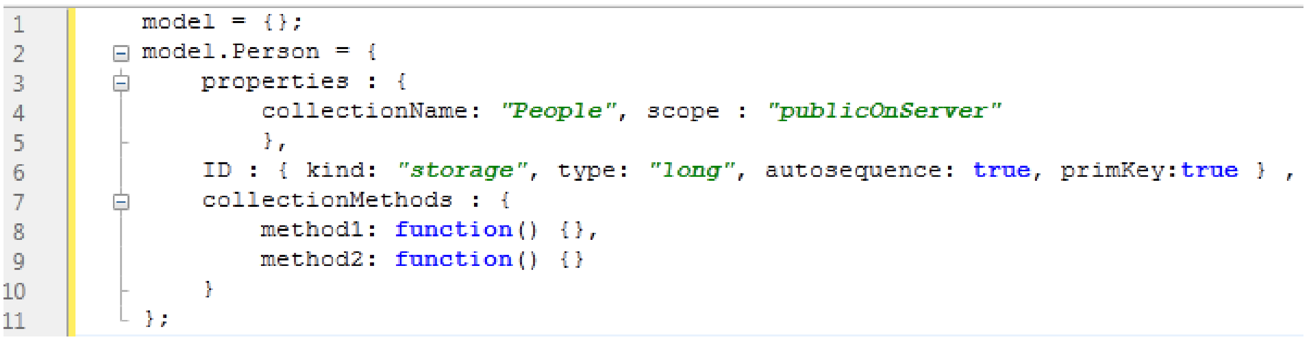
To be able to write the initialization code for the model object, you need to unlock and remove the "guidedModel" object definition. You can click on the Unlock Script button in the Datastore Model Designer toolbar or use the **Edit/Locked Lines** command of the designer's menu:



```
guidedModel =
{
};
```

Warning: Activating the Free form edition mode (also named "unguided model") will disable the automatic create/remove mechanisms between the Wakanda editors and the code editor. For more information, please refer to the Using the Free Form Edition Mode section.
Note also that existing datastore classes (if any) will be available and can be handled through the Model Designer only.

You can then define the **model** object either by using the **DataStoreCatalog( )** model constructor method, or by simple assignment. If you use the constructor method, you benefit from the various methods of the Model API. However, since the *model* object is a standard JavaScript object containing objects referencing datastore classes, attributes and other properties, it can be defined using standard JavaScript object assignment syntax. For example, you can create a valid model using the following simplified code in the *Model.js* file:



```
model = {};
model.Person = {
    properties : {
        collectionName: "People", scope : "publicOnServer"
        },
    ID : { kind: "storage", type: "long", autosequence: true, primKey:true } ,
    collectionMethods : {
        method1: function() {},
        method2: function() {}
    }
};
```

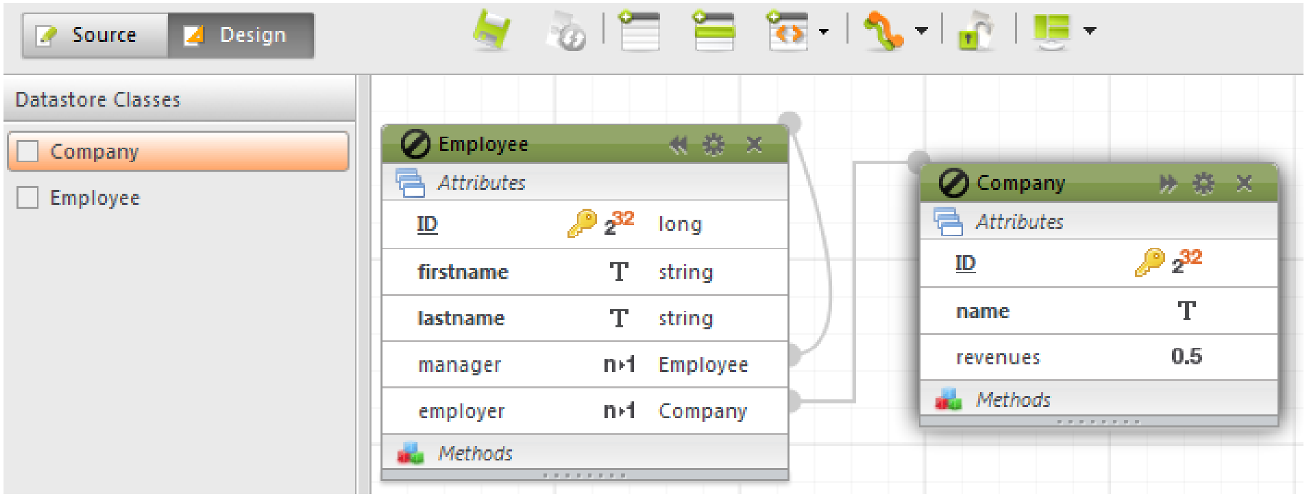
Viewing a Procedural Model in Wakanda Studio

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).

You can add datastore classes using the Model designer and, in this way, mix dataclasses built from the Model JS API and from the Model designer in the same project, but this configuration requires that you take extra care to manage potential naming conflicts.

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"

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* boo lean 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";
    }
```

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 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 *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:

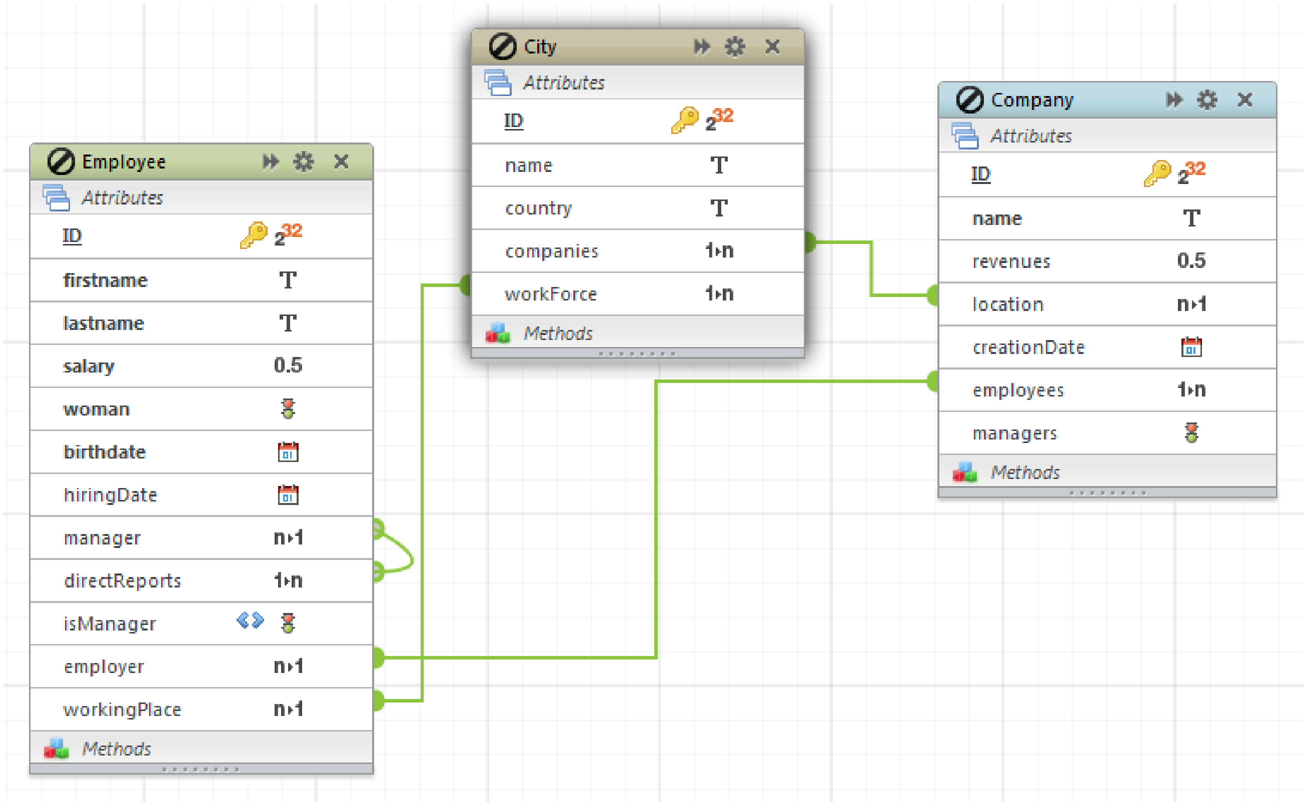
Option	**Type**	**Description**
reversePath	boolean	If true, the relation attribute will use the reverse path of an existing relation (for more information, please refer to the **RelatedEntity or relatedEntities attributes** paragraph.

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 defi
emp.addAttribute("employer", "relatedEntity", "Company", "Company"); // rela
emp.addAttribute("workingPlace", "relatedEntity", "City", "employer.locatio

    //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", "workingPlac
    // 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, pro
    include ("scripts/buildData.js"); // call an external script
    buildData(nbCompanies, progressRef); //execute the script with paramete
},  "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", "Person");
Emp.setRestrictingQuery("salary isnot 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 Datastore Class 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 *extendedClass*[, String | Null *scope*[, Object *properties*]]]])

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

Description

The **addClass( )** method adds a new datastore class to the current dynamic 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 *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 derived from another class.

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".

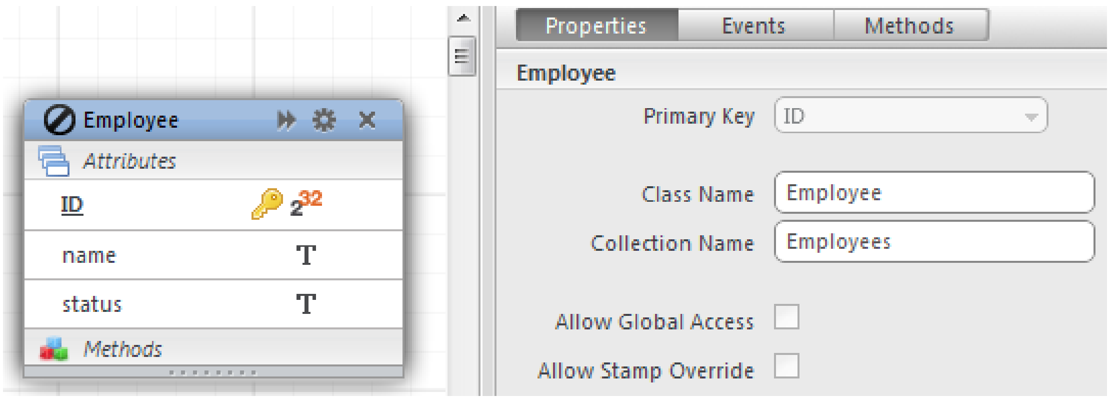
The optional *properties* parameter allows you to define any datastore class property within the **addClass( )** call. This parameter is an object containing property/value pairs. For example, you can pass *{ allowOverrideStamp : true }* in the *properties* parameter if you want to define the **Allow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



Employee

Primary Key: ID
Class Name: Employee
Collection Name: Employees
Allow Global Access: ☐
Allow Stamp Override: ☐

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

addOutsideCatalog( )

void **addOutsideCatalog**(String *localName*, String *hostName*, String *user*, String *password*[, Object *options*])

Parameter	**Type**	**Description**
localName	String	ID of the remote catalog
hostName	String	Address of the remote Wakanda Server
user	String	User name
password	String	User password

options	Object	Optional parameters

Description

The **addOutsideCatalog( )** method allows you to reference and use a remote catalog in your current Wakanda *model* reference. This method allows you to share catalogs between several Wakanda applications.

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 an outside 'Company' datastore class, you can access it through the 'ds.Company' statement.
Data relative to the remote datastore model is also available (provided access is allowed to this data); it can be handled, modified and saved on the remote server transparently.

The remote catalog must be published by Wakanda Server. You can share catalogs between projects published by the same Wakanda Server or different Wakanda Servers. 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.

In *localName*, pass a name to identify the remote catalog in your code.

In *hostName*, pass the address of the Wakanda application 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.

In *user* and *password*, pass the user name and password required to login to the remote application. If the application is in free access mode (see **Free Admin Access vs Controlled Admin Access**), you can just pass empty strings ("") or omit the parameters.

The *options* object can be used to set additional parameters. Actual parameters depend on the remote catalog type. Currently, only a single parameter is available:

- **sharedAccess**: *boolean* (example: **'sharedAccess': true**)
  When you connect to a remote server using **addOutsideCatalog( )**, by default the user/password access is shared by all the threads connecting to the server. In this mode, you do not control access rights for each user. It means that users with lower level rights could have access to data they should not be allowed to.
  To apply actual user access rights to remote data, you must activate the 'passthru' mode. In this mode, you will neither pass *user* nor *password* parameters and Wakanda will automatically pass the initial user login to the remote server. To activate this mode, pass **false** to the **sharedAccess** property.
  Implementation Note: In the current version of Wakanda, the 'passthru' mode is not available and sharedAccess is always true.

Example

You want to reference a catalog from another project in the same solution, so you can write:

```
model = new DataStoreCatalog(); // create a model reference
model.addOutsideCatalog("myOtherProject","http://127.0.0.1:8082"); //other
```

Example

You want to reference the catalog from another Wakanda Server through its host name:

```
model = new DataStoreCatalog();
model.addOutsideCatalog("books", "http://books.mylibrary.com", "admin", "nir
```

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 various properties and methods of 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
```