

# Wakanda

## Server-Side Concepts



©2012 4D SAS. All rights reserved.

# Table of Contents

---

<b>Table of Contents</b> .....	<b>2</b>
<b>Wakanda Overview</b> .....	<b>4</b>
<b>Wakanda Datastore Model</b> .....	<b>4</b>
Relational Databases, Normalization vs. Denormalization .....	4
SQL Queries Return Row Sets not Records.....	5
Enter the Datastore Model Paradigm.....	5
Singular and Plural.....	6
Wakanda Terminology.....	7
<b>Datastore Classes</b> .....	<b>9</b>
<b>Entities</b> .....	<b>9</b>
<b>Entity Collections</b> .....	<b>9</b>
<b>Attributes</b> .....	<b>9</b>
Storage Attributes .....	10
Attribute Indexes.....	10
Custom Data Types .....	10
Calculated Attributes .....	10
Primary Relation Attribute.....	13
Alias Attributes.....	14
Dependent Relation Attributes and a More Sophisticated Example.....	15
Entity Collections and Attributes.....	17
<b>Inheritance</b> .....	<b>17</b>
Public, Public On Server, Protected and Private.....	18
Removing Inherited Attributes.....	19
Restricting Queries and Inheritance.....	19
On Restricting Query event.....	21
<b>Datastore Class Methods</b> .....	<b>21</b>
<b>Class Events and Attribute Events</b> .....	<b>28</b>
On Init.....	29
On Load.....	29
On Set.....	31
On Validate .....	32
On Save.....	34
On Remove .....	35
<b>Programming in Wakanda Server</b> .....	<b>36</b>
Datastore Classes.....	37
Datastore Class Attributes .....	37
Attribute Names and Attribute References .....	38
Creating Entities.....	38
Entity Collections.....	39
Queries and Finds.....	40
<i>Placeholder Query Syntax</i> .....	41
<i>Locating Entities by Key</i> .....	43
<i>Using Nulls in Queries</i> .....	43

<i>Conjunctions</i> .....	43
<i>Comparison Operators</i> .....	44
<i>Using JavaScript in Queries</i> .....	44
Relation Attributes and Entity Collections.....	44
Scalar Attributes and Entity Collections .....	45
Entity Collections and toArray() .....	45
Wakanda Transactions.....	48
Entity Locking.....	48
Testing Code .....	49
Wakanda Server Global Namespace and Objects.....	50
Wakanda Server File Management .....	55
Dedicated Workers and Shared Workers .....	55
Mutex() .....	61

## Wakanda Overview

Wakanda is a development and deployment environment for browser-based data applications. Wakanda is composed of three components: Wakanda Server, a data/HTTP/REST/web server, Wakanda Application Framework (WAF), a browser-based JavaScript framework used to simplify access to Wakanda Server, and Wakanda Studio, the development environment used to create both front- and back-end portions of an application. The native programming language for Wakanda is JavaScript. When designing a Wakanda application, you use JavaScript under Wakanda Server and also when designing web pages using WAF.

In order to make development as easy as possible, WAF programming is similar to programming under Wakanda Server. For example, the WAF provides access to datastore classes on the browser side in a manner substantially similar to accessing these same classes under Wakanda Server. This document however explores Wakanda Server concepts and code. Unless specified, programming examples in this document refer to server-side Wakanda programming.

## Wakanda Datastore Model

Built into Wakanda is a data management system that lets you easily manipulate information. The Wakanda datastore model uses a datastore class paradigm rather than a relational database methodology. Instead of representing information as tables, records, and fields, Wakanda uses a new approach that more accurately maps data to real world items and concepts.

## Relational Databases, Normalization vs. Denormalization

Relational databases are an efficient way to store and retrieve information. They achieve efficiency by storing information in lists called tables and linking tables together through values in specific fields. The highest level of efficiency is achieved by following a discipline called normalization in which information only appears in one location but is referenced wherever needed. This approach removes the need to update data in multiple places whenever a value is changed. However, normalization can add complexity to the development process and often results in a relational structure that obscures the real world concepts of the information it contains.

Take for example the following partial relational database structure:

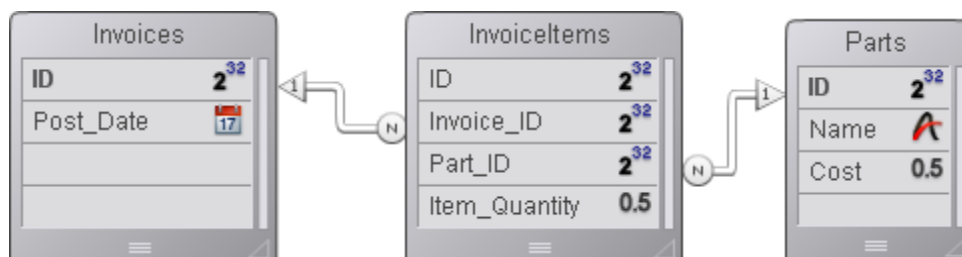


Fig 1

This structure describes Invoices, each of which can have many InvoiceItems. Each InvoiceItem refers to one Part that holds the name and individual cost. An Invoice total might

be calculated by summing the product of InvoiceItems.Item\_Quantity and Parts.Cost for all InvoiceItems for a given Invoice. Of course, each time the Invoice total is calculated, the relationships between all three tables must be understood and traversed correctly. This approach adds complexity to the development process. Often database designers will avoid this kind of complexity by incorporating some degree of “denormalization” in their data’s structure. For example, in the above structure one might be tempted to add a Cost field to InvoiceItems copied from the corresponding part or an InvoiceTotal field to the Invoices table where the sum of all its InvoiceItems is stored. Adding these two fields would certainly be more convenient and in some cases increase performance to one portion of the development process, but increase complexity in another. With a Cost field in InvoiceItems or an InvoiceTotal field in Invoices, developers must take care to recalculate and store these fields whenever a cost or quantity is changed. This perpetual battle between the need for efficiency via normalization and the need for convenience via denormalization is one of the issues that the Wakanda datastore model addresses.

## SQL Queries Return Row Sets not Records

Another struggle in relational databases is the difference between the records stored in tables and the rows returned by a query. Take for example the following query used with the relational structure above:

```
SELECT i.Post_Date, p.Name, p.Cost, ii.Item_Quantity, p.Cost * ii.Item_Quantity as Extended
From InvoiceItems ii
Join Parts p on p.ID = ii.Part_ID
Join Invoices i on i.ID = ii.Invoice_ID
Where i.Post_Date between '1/1/10' and '12/31/10'
```

The result of this query would be a list of post dates, part names, costs, quantities and extended costs for all invoice items on all invoices in the year 2010. Notice that the resulting rows are a collection of columns from all three tables and include a calculation. Each row in the resulting row set is neither an invoice, nor a part, nor an invoice item. In addition, each row is independent from the data where it is stored and does not “remember” to which record(s) it belongs.

## Enter the Datastore Model Paradigm

Imagine the ability to denormalize a relational structure yet not affect efficiency. Imagine describing everything about the business objects in your application in such a way that using the data becomes simple and straightforward and removes the need for a complete understanding of the relational structure. Imagine that the result of queries is an object that “knows” where it is stored and its relationship to other objects. These are some of the goals of the Wakanda datastore model.

In the Wakanda datastore model, a single datastore class can incorporate all the elements that make up a traditional relational database table but can also include calculated values, values from related parent entities, and direct references to related entities and entity collections.

When a Wakanda datastore model is complete, a query returns a list of entities called an Entity Collection, which fulfills the role of a SQL query’s row set. The difference is that each entity “knows” where it belongs in the data model and “understands” its relationships to all other entities. This means that a developer does not need to explain in a query how to relate

the various pieces of information nor in an update how to write modified values back to the relational structure.

So, in Wakanda, the structure can become:

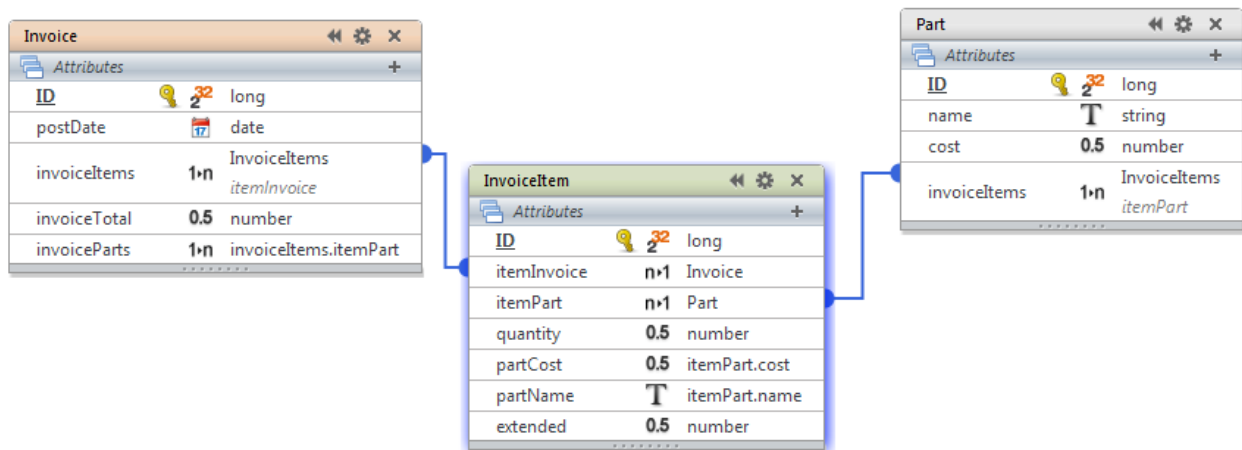


Fig 2

Notice that each datastore class (Invoice, InvoiceItem, and Part) has been “filled out” with every logical property and that each one reflects a complete picture of the entity and all its related entities. Invoice now contains an attribute (invoiceItems) that represents all the related InvoiceItems. InvoiceItem now contains an attribute (itemPart) that represents the corresponding part as well as all the attributes from the parent Part entity along with a calculated attribute (extended) that determines the extended cost.

## Singular and Plural

In a traditional database, a table has only one name. Many times developers name a table in the plural. For example, a typical name for a table might be “Cities” or “Companies”. Later during development, it often becomes necessary to refer to a single record in a table, particularly in association with relationships between tables. Sometimes developers name tables in the singular. For example, “City” or “Company” but then later it becomes necessary to refer to multiple records by some name. Wakanda addresses this need by providing a singular name for each datastore class and a plural name for entity collections of that class. For example, a class might be named “City” or “Company” while a collection of entities in those classes would be “Cities” or “Companies”. This duality helps clarify the relationships between datastore classes and provides a more natural set of expressions when referencing an entity and an entity collection. In the above diagram (Fig 2) note the *itemInvoice* attribute in the *InvoiceItem* class. This attribute is of type “Invoice” (note the singular) and makes it clear that it references a single entity of the Invoice class.

## Wakanda Terminology

Throughout the rest of this document we will refer to Wakanda items differently than relational database items so as not to confuse the two. Here is a brief list of the terminology used in Wakanda and the closest relational database equivalent.

Wakanda	RDBMS	Notes
Solution		A Wakanda solution is a collection of projects, each of which contains a single datastore model. A solution can contain any number of projects, all of which can be accessible at the same time.
Project		A Wakanda project is a design time concept that houses a single datastore model along with the other files that make up a single Wakanda application.
Application	Database	A Wakanda application is the runtime entity of a single Wakanda project.
Datastore Model	Database Structure	A complete set of all datastore classes for a single Wakanda project is referred to as a datastore model and is roughly equivalent to the relational concept of a single database structure. Datastore model is the top-level term for a Wakanda application's datastore and incorporates all the datastore classes for the project/application. The datastore model is sometimes referred to as the model.
Datastore Class	Table	The Wakanda datastore class is similar to a database table except a datastore class can be inherited and can contain many other types of properties (attributes & methods). In this document and elsewhere, datastore classes are sometimes referred to as classes.
Datastore Entity	Record or Row	In traditional relational databases a record is the logical unit of storage within a table that holds a value for each field in the table. In a SQL database, a row is one of the results of a query, which may not correspond directly to a record. Since a Wakanda datastore entity can represent any level of denormalization without its accompanying shortcomings, it fulfills both of these roles. In this document and elsewhere, datastore entities are often referred to as entities.
Entity Collection	Row Set (SQL) or Selection (4D)	These two concepts are only roughly similar. Whereas a row set is a list of rows, each of which contains values in columns an entity collection is a bundle of entities (objects), each of which contains a complete picture of the object's situation in the data. The 4D concept "Selection" is more similar to an entity collection except denormalization has its typical drawbacks on a relation database.
Attributes	Fields or Columns	A field is the smallest storage cell in a relational database. It is sometimes referred to as a column. However, the term "column" is more often associated with the results of a query where it may or may not be the value of a field. In Wakanda the collective term <i>attribute</i> is used to describe fields where information is stored as well as several other items that can return a value.
Storage Attribute	Field	A storage attribute (sometimes referred to as a scalar attribute) is the most basic type of attribute in a Wakanda datastore class and most directly corresponds to a field in a relational database. A storage attribute holds a single value for each entity in the class.
Calculated Attribute	Calculated Field	A calculated attribute doesn't actually store information. Instead, it determines its value based on other values from the same entity or from other entities, attributes or methods. When a calculated attribute is referenced, the underlying "calculation" is evaluated to determine the value. Calculated attributes may even be assigned

		values where user-defined code determines what to do during the assignment.
Scalar Attribute		A scalar attribute is a collective term to denote any attribute that returns a single value. All storage and most calculated attributes are scalar.
Relation Attribute	Relationship	Relation attribute is a collective term that exposes a relationship between two datastore classes and can be either N->1 or 1->N. Since they are attributes, they can be named and therefore become available as properties of the corresponding class. A relation attribute can result in an entity or an entity collection.
Primary Relation Attribute		A primary relation attribute is not dependent upon another relation attribute. Primary relation attributes are created by adding an attribute to a datastore class that refers to another class and either creates a new relationship between the classes or reverses the path of an existing N->1 primary attribute. Only primary relation attributes can be designated as a composition relationship.
Dependent Relation Attribute		A dependent relation attribute is one that is reliant on another relation attribute in the same datastore class.
Alias Attribute	Denormalized Field	An alias attribute builds upon a relation attribute. Once an N->1 relation attribute is defined, any of the attributes within the “parent” class can be directly referenced as attributes within the “child” class. The result is what appears to be denormalized data without the overhead of duplicating information. Alias attributes can reference any available attributes further up the relational tree.
Property		A general term for either an attribute or a method associated with a datastore class.
Relationship		Two datastore classes are associated via a relationship, which is an artifact of a datastore model. A relationship’s existence results in either one or two relation attributes, one in each of the related classes. Relation attributes are named and become a property of their respective classes.
Method		A collective term for a discreet portion of programming in Wakanda. Calculated attribute methods, class methods and event methods are all examples.
Datastore Class Method		In addition to attributes, each datastore class may also include methods. There are three different scopes to datastore class methods: <i>Class</i> , <i>Collection</i> , and <i>Entity</i> .
Calculated Attribute Method		A method that substitutes programming behavior in place of standard behavior for an attribute. There are four aspects to calculated attribute methods: <i>On Get</i> , <i>On Set</i> , <i>On Sort</i> , and <i>On Query</i> .
Event Method		A method triggered by either a datastore class event or an attribute event. These methods are sometimes referred to as simply events.
Events		Any of several built-in trigger points in a Wakanda datastore model. Events are available for both attributes and datastore classes. Wakanda supports six different events: <i>On Init</i> , <i>On Load</i> , <i>On Set</i> , <i>On Validate</i> , <i>On Save</i> , <i>On Remove</i> and <i>On Restricting Query</i> .
Inheritance		Similar to the like-named concept found in object oriented programming, a datastore class can inherit from another. The newly defined (derived) class may add additional properties or hide inherited properties. Datastore class inheritance may be defined by a query string stored in a project’s datastore model or by the <i>On Restricting Query</i> event



Query	Query	A query in a traditional database typically returns a row set, which can be thought of as a grid of values (rows and columns). A Wakanda query returns an entity collection, which is a bundle of entities, each of which “remembers” where it exists in the data.
Datastore	Database	The Wakanda term <i>datastore</i> is used to reference a single application’s set of data.

## Datastore Classes

In Wakanda a datastore class represents several things: it is the blueprint or template used when defining entities of its type, it is a named object with several attributes and methods, it is a visual construct used in the Wakanda Datastore Model Designer, and it is the object to which you add events, methods, and other properties. Each class has one attribute that is unique and represents its key. Commonly, the key attribute for a datastore class is set to generate an auto sequence value so that each entity has a unique key as it comes into being.

*Note: Future versions of Wakanda will support multi-attribute keys.*

## Entities

An entity is a reference to a single instance defined by a particular datastore class. Entities are the basic unit in a Wakanda application. When an entity reference is obtained by means of an entity collection, it retains information about the entity collection which allows iteration through the collection. Dataclass methods defined at the Entity level become properties of entities in the dataclass.

## Entity Collections

An entity collection is a group of entity references of the same type that may or may not be sorted into a particular order. Normally, an entity reference only appears once in an entity collection. However, if an entity collection is sorted, it is possible that a single entity’s reference be a member of the entity collection more than once. Dataclass methods defined at the Collection level become properties of entity collections of the dataclass.

## Attributes

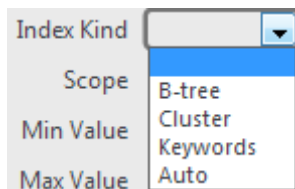
Attributes are named properties of Datastore Classes and come in a variety of kinds: Storage, Calculated, Relation, and Alias. Attributes that are scalar (i.e., provide only a single value) have one of the following data types: BLOB, bool, byte, date, duration, image, long, long64, number, string, uuid, and word.

## Storage Attributes

A storage attribute is equivalent to a field in a relational database and can be indexed. Values assigned to a storage attribute are stored as part of the entity when it is saved. When a storage attribute is accessed, its value comes directly from the datastore. Storage attributes are the most basic building block of an entity and are defined by name and data type. In addition, all datastore classes become a data type in a Wakanda application.

## Attribute Indexes

Wakanda allows you to index storage attributes. In Wakanda there are four different choices for index types: B-tree, cluster, keywords, and auto. B-tree is a general-purpose index that works well in most circumstances and is the most commonly chosen type. Cluster is an index that is best used on attributes with low cardinality, that is, when there are fewer unique values for an attribute. The keyword index type is used for large quantities of text and provides quick searches for individual words in the text. If you choose Auto, Wakanda will choose the index type (either B-tree or Cluster) based upon the data at the time the attribute is indexed. For classes with no saved entities, Auto will choose a B-tree index for all attribute types except Boolean, which will be set to Cluster.



## Custom Data Types

Any of the existing storage attributes can become the basis for new data types that encapsulate length controls, pattern matching and default formats. Once created, a custom data type is available for any attribute.

*Note: Custom data types are not yet available in Wakanda.*

## Calculated Attributes

A calculated attribute is a named property with a data type that masks a calculation. At the very minimum, a calculated attribute requires an *On Get* method that describes how its value will be calculated. When an *On Get* method is supplied for an attribute, Wakanda does not create the underlying storage space in the datastore but instead substitutes the method's code each time the attribute is accessed. If the attribute is not accessed, the code never executes.

A calculated attribute can also implement an *On Set* method, which executes whenever a value is assigned to the attribute. The *On Set* method describes what to do with the assigned value usually redirecting it to one or more storage attributes or in some cases other entities.

Just like storage attributes, calculated attributes may be included in queries. Normally, when a calculated attribute is used in a Wakanda query, the attribute is calculated once per entity examined. In many cases this is sufficient. However, calculated attributes can implement an *On Query* method that substitutes other attributes during the query. This allows calculated

attributes to be queried quickly by redirecting searches to other attributes, including storage attributes that may already be indexed.

Similarly, calculated attributes can be included in sorts. When a calculated attribute is used in a Wakanda sort, the attribute is calculated once per entity examined. Just like in queries, this is sufficient in many cases. However, calculated attributes can implement an *On Sort* method that substitutes other attributes during the sort, thus increasing performance.

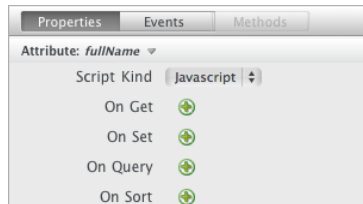


Fig 3 (the four different calculated attribute methods)

For an example of a calculated attribute, take the following datastore class:

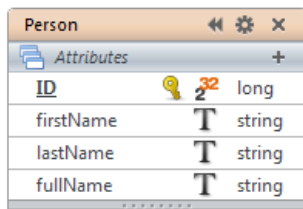


Fig 3.1

*ID*, *firstName*, and *lastName* are standard storage attributes. *fullName* is a calculated attribute with an *On Get* method as shown below:

```
onGet:function()  
{  
  return this.firstName + ' ' + this.lastName;  
}
```

Whenever the *fullName* attribute is accessed, Wakanda evaluates the expression and returns the resulting value. Since this code becomes part of the attribute's definition, it need not be referenced again in the application.

An implementation of the *On Set* calculated attribute method for *fullName* might be:

```
onSet:function(value)  
{  
  var names = value.split(' '); //split value into an array  
  this.firstName = names[0];  
  this.lastName = names[1];  
}
```

In this code, *value* is the parameter that receives the value assigned to the *fullName* attribute and is subsequently portioned out and assigned to the storage attributes *firstName* and *lastName*.

*On Sort* and *On Query* calculated attribute methods behave differently. Instead of simply performing an action, they return a string that is substituted during the operation. For example, the *On Sort* method for *fullName* might be:

```
onSort:function(ascending)
{
  if (ascending)
    return 'lastName, firstName';
  else
    return 'lastName desc, firstName desc';
}
```

The *On Sort* method receives a single Boolean value that describes whether the sort is ascending or descending. Using that parameter, the sort can be replaced with a string representation of other attributes. Notice that in the example above, *lastName* is sorted before *firstName*: the *fullName* attribute is calculated using `firstName + ' ' + lastName`. This illustrates that calculated attributes do not need to sort the same as the values they return.

Similar to the *On Sort* method, the *On Query* method returns a string that represents the redirected query. The *On Query* method receives two string parameters, the comparison operator (e.g. “>”, “>=”, etc.) and the compared value. It is the job of the *On Query* method to use these two values and construct a string that represents a substituted query fragment. Although more sophisticated than the other calculated attribute methods, the *On Query* usually follows a pattern that responds to the various kinds of comparison operators. Consider the following method:

```
onQuery:function(compOp, valueToCompare)
{
  var result = null;
  var pieces = valueToCompare.split(' '); //break into array
  var fname = pieces[0];
  var lname = ""; //not sure they provided a full name
  if (pieces.length > 1) //so check
    lname = pieces[1]; //2nd piece provided
  if (lname == "") { //only one piece was supplied
    if (compOp == '=') { //we'll take to mean special case
      //indicating very broad query
      result = '(firstName = "' + fname + "'';
      result += ' or lastName = "' + fname + "'';
    } //if
    else //we'll take this to mean comparison to lastName
      result = 'lastName ' + compOp + "' + fname + "'";
  }
  else { //two pieces were supplied
    switch (compOp) {
      case '=': //since no 'break' runs next case
      case '==':
        result = 'firstName ' + compOp + "' + fname + "'";
        result += ' and lastName ' + compOp + "' + lname + "'";
        break;
      case '!=': //since no 'break' runs next case
      case '!==':
        /* could use this but not as fast
        result = "(firstName "+ compOp + "' + fname + "'";
        result += "and lastName "+ compOp + "' + lname + "'";
```

```

        instead use the code below */
    result = 'not (';
    result += 'firstName '+compOp.substr(1)+ '''+fname+'''';
    result += 'and lastName '+compOp.substr(1)+ '''+lname+'''';
    break;
case '>': //all 4 handled in the case below
case '>=':
case '<':
case '<=':
    var compOper2 = compOp[0]; // get the first char
    result = '(lastName = "' + lname + '" and firstName '
    result += compOp + "' + fname + '")';
    result += 'or (lastName ' + compOper2 + "' + lname+ '")';
    break;
    } //switch
} //else
return result;
}

```

Notice that in all cases this function returns the string variable *result*. Although this method may seem somewhat daunting, this code is a realistic example of how you might handle the *fullName* calculated attribute's *On Query* method and it does quite a lot. This example handles queries on *fullName* where only one name part is supplied (e.g. "Smith") as well as full names (e.g. "John Smith"). In a query with a single part, it provides for a search on both *lastName* and *firstName* when the "=" operator is supplied. When using a not equals operator ("!=" or "!==") it substitutes a faster query that respects the operator used (see the comments). When a full name is supplied it expects the values to be in first name, last name order. It would be fairly easy to augment this code such that it handled full names in reverse order (e.g. "Smith, John").

## Primary Relation Attribute

A relation attribute provides access to other entities. Relation attributes can result in either a single entity (or no entity) or an entity collection (0 to N entities). It is important when discussing relationships between datastore classes that we indicate which "end" of the relationship represents the many (or N) and which represents the one (1). Sometimes developers refer to this as a parent/child relationship where there is one parent entity and multiple child entities. In other cases the relationships between datastore classes is described using terms "belongs to" and "has many." Regardless of how we choose to discuss relationships, the "point of view" used in the discussion is important. For example, consider the following partial Wakanda structure:

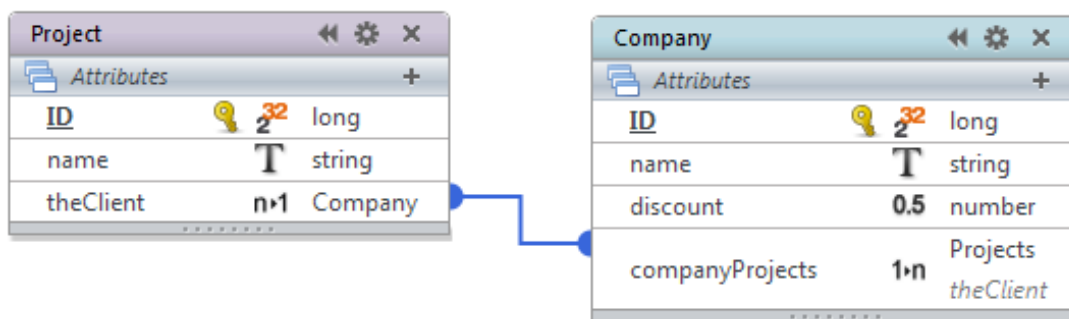


Fig 4

According to the single relationship in this model, there is at most one Company for each Project. Conversely, for each Company there is any number of related Projects. From a single project's point of view, there may or may not be one Company entity, considered the client. From a single Company, there can be any number of Projects, including none.

When you define a relation attribute in Wakanda, you usually start by creating an attribute in the many or child class, naming it and choosing as its type another datastore class. Doing so creates a relationship between the two classes. In the above diagram the *theClient* attribute in the Project class was added first. When this attribute was created, Wakanda automatically created the reciprocal attribute in Company and gave it a default name. In the above example, the reciprocal relation attribute was renamed *companyProjects*. It is possible to remove the 1->N attribute from the parent class if desired.

The resulting relation attribute (*theClient* in the above example) in the child datastore class references a single entity in the parent. The reciprocal relation attribute (*companyProjects* in the example above) in the parent class references an entity collection in the child. Typically you name each attribute in a way that describes it logically from the containing class's point of view. So, in the example above we named the relation attribute in the Project class *theClient* in order to indicate that the related company is the client for this project. In the Company class, the relation attribute is named *companyProjects* (notice the use of the plural) and when accessed results in an entity collection of related projects. In many cases, naming the attributes in a useful way is the only challenging task in relating datastore classes.

Both *theClient* and *companyProjects* in the above diagram are primary relation attributes and represent a direct relationship between the two entities. Below you will see how to build on these attributes using dependent relation attributes.

### Alias Attributes

Once a relationship exists between datastore classes, it becomes simple to add new attributes to child classes that refer to parent or even ancestral class attributes. Referred to as alias attributes, these properties store no data, but provide the convenience found in a denormalized relational structure. Consider the following partial Wakanda model:

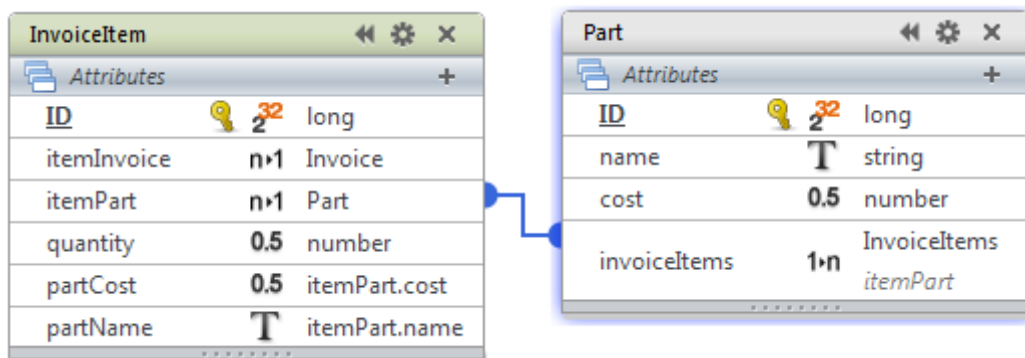


Fig 5

Notice that the InvoiceItem class has two attributes, *partCost* and *partName*, each of which refers to a storage attribute in the related class Part via the relation attribute *itemPart*. Because Wakanda traverses a specifically named relation attribute to determine each alias attribute's value, there is no ambiguity when it is accessed. Any number of alias attributes

can be included this way. The result is an InvoiceItem datastore class that incorporates all the pieces of information needed by the application.

## Dependent Relation Attributes and a More Sophisticated Example

Wakanda's relational abilities don't end with primary relation attributes. Wakanda also allows relation attributes that are dependent on other relation attributes. Called dependent relation attributes, they refer to other datastore classes via a path that always starts with another relation attribute in the same datastore class. So, consider the following Wakanda model:

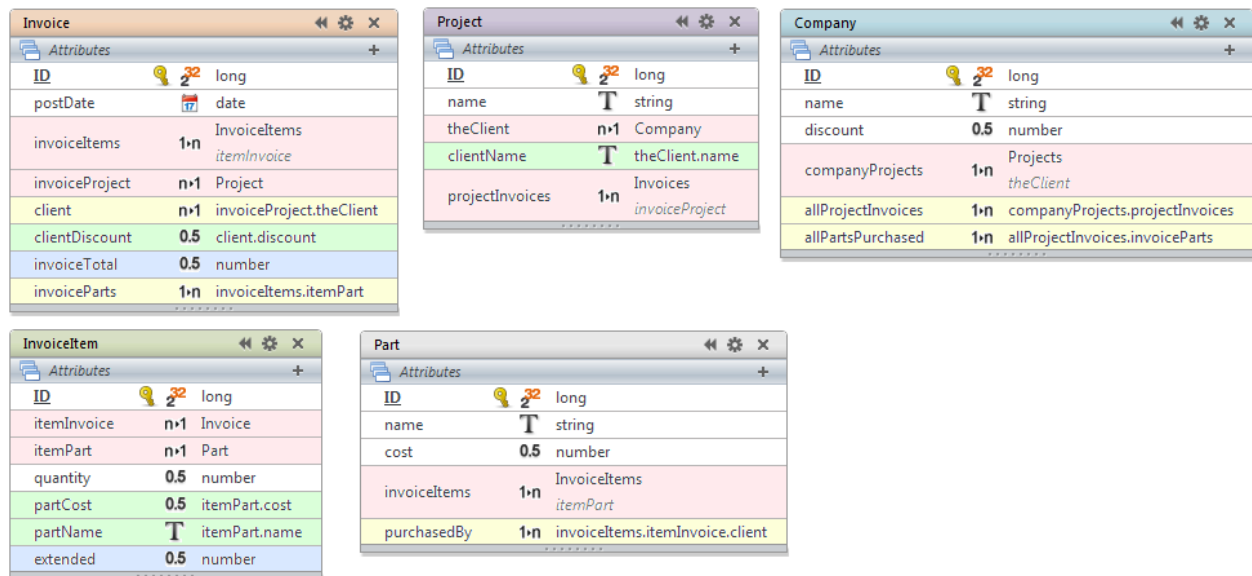


Fig 6 (relation lines between the datastore classes are hidden)

In the diagram above we show a variety of different kinds of attributes and have added color for clarity. The attributes are color-coded as follows:

- Storage attributes are white
- Calculated attributes are blue
- Primary relation attributes are pink
- Alias attributes are green
- Dependent relation attributes are yellow

When a dependent relation attribute is created, instead of a data type, you indicate a path. The path is composed of a list of other relation attributes separated by periods. Just like primary relation attributes, a dependent relation attribute can result in either a single entity or an entity collection.

Consider the *client* attribute in Invoice. This dependent relation attribute uses as its type the path of *invoiceProject.theClient*. It references the project's parent company and in use would return a single entity. Notice that the path starts with a reference to another attribute in the same entity (*invoiceProject*) and the path describes a chain of N->1 relation attributes. **When**

***a dependent relation attribute's path is composed of only N->1 relation attributes, the result is a single entity.***

Consider the *allProjectInvoices* attribute in Company. This dependent relation attribute is derived from the Company *companyProjects* attribute (which alone would result in an entity collection of related projects) combined with the Project relation attribute *projectInvoices*. When the *allProjectInvoices* attribute is accessed it will result in an entity collection of all invoices for all projects for a given Company entity. ***When a dependent relation attribute's path includes at least one 1->N relation attribute, the result is an entity collection.***

Consider the *invoiceParts* attribute in Invoice. This relation attribute uses the Invoice *invoiceItems* relation attribute combined with the InvoiceItem *itemPart* relation attribute. The path then includes both a 1->N and an N->1 relation attribute. In use, the *invoiceParts* relation attribute results in an entity collection of all parts used on an invoice. The resulting entity collection would contain each part once, regardless of whether the part was used on more than one invoice item.

Consider the *purchasedBy* relation attribute in Part. This attribute uses the Part *invoiceItems* attribute, combined with the InvoiceItem *itemInvoice* attribute combined with the Invoice *client* attribute. Following the logic through you will see that the *purchasedBy* attribute results in all companies that have any invoices that sold the part. Conversely, the *allPartsPurchased* attribute in Company will result in all parts on all invoices for the company. The incorporation of sophisticated relation attributes at the datastore class level allows Wakanda applications to function without the need for explicit joins.

Notice that Invoice has an alias attribute named *clientDiscount* that results in the *discount* storage attribute in Company. *clientDiscount* uses the *client* dependent relation attribute in Invoice which references the *invoiceProject* attribute which references the related Project. Thus, alias attributes can reference values using any valid N->1 relation attribute in the same entity including dependent relation attributes.

There are a number of ways to achieve the same results in a Wakanda datastore model. For example, the *clientDiscount* attribute in Invoice could have been implemented using *invoiceProject.theClient.discount*. Similarly, the *allPartsPurchased* attribute in Company could have been implemented using *companyProjects.projectInvoices.invoiceItems.itemPart*. Although the entire path would be clear, the length of the path might become unmanageable. It is often easier to use intermediary relation attributes for convenience. Regardless of how you implement alias or relation attributes, if they traverse the same relational path, they are equivalent in efficiency. The main difference is in the readability of the datastore model.

The Wakanda model in Fig 6 also incorporates the calculated attributes *invoiceTotal* in Invoice and *extended* in InvoiceItem. The *invoiceTotal onGet* method could be as simple as:

```
onGet:function(){
    return this.invoiceItems.sum('extended');
}
```

Where the *extended onGet* method could be:

```
onGet:function(){
    return this.partCost * this.quantity;
}
```



## Entity Collections and Attributes

So far we have seen how to incorporate very sophisticated relationships between datastore classes and then use relation attributes to traverse the relational web of the data. But in Wakanda, the use of attributes is not limited to individual entities. Attributes are also used in conjunction with entity collections. When a scalar attribute is used with an entity collection, the result is an array of values each of which is the type of the scalar attribute. When a relation attribute is used in conjunction with an entity collection, the result is a new entity collection made up of all related entities. For example, consider the partial Wakanda model in Fig 5. Given an entity collection of invoice items, the attribute *itemPart* will project a collection of all parts for all invoice items in the original entity collection. This ability allows you to traverse relationships on multiple entities at a time (more on this later).

## Inheritance

Although common in object-oriented programming, inheritance is a concept not usually found in the data management world. However, there are some data management situations that could benefit from inheritance. Consider the following partial relational database structure:



Fig 7

This normalized structure describes People (inside and outside of the organization), Employees (only those inside the organization) and Salespeople (a specific subset of employees). Although shown in the above diagram using N->1 relationships, our business rules dictate that each salesperson represents exactly one employee, and each employee represents exactly one person. A salesperson cannot exist without a reference to an employee and an employee cannot exist without a reference to a person. Furthermore, employee records derive their key value from the corresponding person record and salespeople derive their key from the corresponding employee. In less formal terms, we can say that a salesperson is a kind of employee, and that an employee is a kind of person. Later when the data management system needs to address salespeople it can do so by operating on the Salespeople table but of course must join with the other tables to access a complete picture of the data.

Alternately, we could denormalize the data and combine all fields into one table, but then we would have to add extra fields that categorize the various record types of employees and salespeople and manage these types. Furthermore, if we combine all the fields into one table, each query that needs to operate on say Salespeople would have to include a means to identify only records of type salesperson. This places a greater burden on down stream business logic to identify the different types of people and their corresponding significant fields.

This situation is an example that is better addressed through inheritance. In Wakanda you do this by extending a datastore class to create a new class. So, in Wakanda the model can become:

The figure shows three screenshots of the Wakanda datastore class attribute lists:

- Person:** Attributes include ID (long, primary key), firstName (string), lastName (string), and fullName (string).
- Employee:** Attributes include hireDate (date), accessLevel (long), salary (number), and inherited attributes from Person (ID, firstName, lastName, fullName).
- Salesperson:** Attributes include territory (string), commissionType (string), hireDate (date), accessLevel (long), salary (number), and inherited attributes from Employee and Person (ID, firstName, lastName, fullName).

Fig 8

**Note:** In this version of Wakanda, you cannot yet add storage attributes to derived classes and are restricted to adding relation, alias and calculated attributes.

In the above Wakanda model, the Salesperson class inherits from the Employee class, which inherits from Person. Later during runtime each employee entity is composed of all properties from both Employee and Person and each salesperson entity is composed of properties from all three datastore classes. This includes all attributes and all methods associated with the extended classes. When a Wakanda application needs to work with just employees or just salespeople, it doesn't need to explicitly refine each query to "filter out" unneeded entities. Furthermore, when a derived class is created, Wakanda manages the underlying data and generates the inherited entities when needed. For example, if a new Salesperson entity is created and saved, Wakanda automatically manages the creation of the corresponding new Employee and Person. Furthermore, if a new attribute is added to the Person class, it becomes available in all derived datastore classes.

## Public, Public On Server, Protected and Private

Attributes in Wakanda can have a scope of *public*, *public on server*, *protected*, or *private*. The default scope is *public* which allows access to the attribute from anywhere, including the client-side framework (WAF). The scope of *public on server* allows access to the attribute from anywhere on Wakanda Server but the attribute is excluded from the WAF. If the scope is *protected* then the attribute is excluded from the WAF and is only available from within code attached to its class or to classes derived from its class. This includes class methods, class events and calculated attribute methods associated with the class or any inheriting classes. If an attribute is *private*, it is excluded from the WAF and can only be accessed by code attached to its own class. If a *protected* or *private* attribute is accessed outside of the proper context, Wakanda will generate an error.

## Removing Inherited Attributes

In some instances, inherited attributes are not appropriate for specific derived datastore classes. Consider the model in Fig 8. Let's assume that our application requires workPhone for People but employees and salespeople don't have individual work phone numbers and that we don't want to include *workPhone* as an attribute in either Employee or Salesperson. Wakanda allows inherited attributes to be removed in derived classes. In the Wakanda Datastore Model Designer, you do this by clicking the remove checkbox in the properties panel for the attribute in the derived class. The removed attribute appears "crossed out" in the section shown below and is not available in derived entities.

From People		
ID	<del>2</del> <sup>32</sup>	long
firstName	T	string
lastName	T	string
fullName	T	string
workPhone	T	string
cellPhone	T	string

Fig 8.1

## Restricting Queries and Inheritance

Without a restricting query, Wakanda manages inheritance classification. That is, if a derived entity is created, Wakanda assures that it is available as an entity in the derived dataclass. But this behavior can be overridden using a restricting query. When a restricting query is provided for a derived datastore class, the query becomes the basis for the inheritance classification. This allows the developer direct access to the entity's classification and provides a way to promote and demote entities up and down the inheritance chain.

For example, consider the following model:

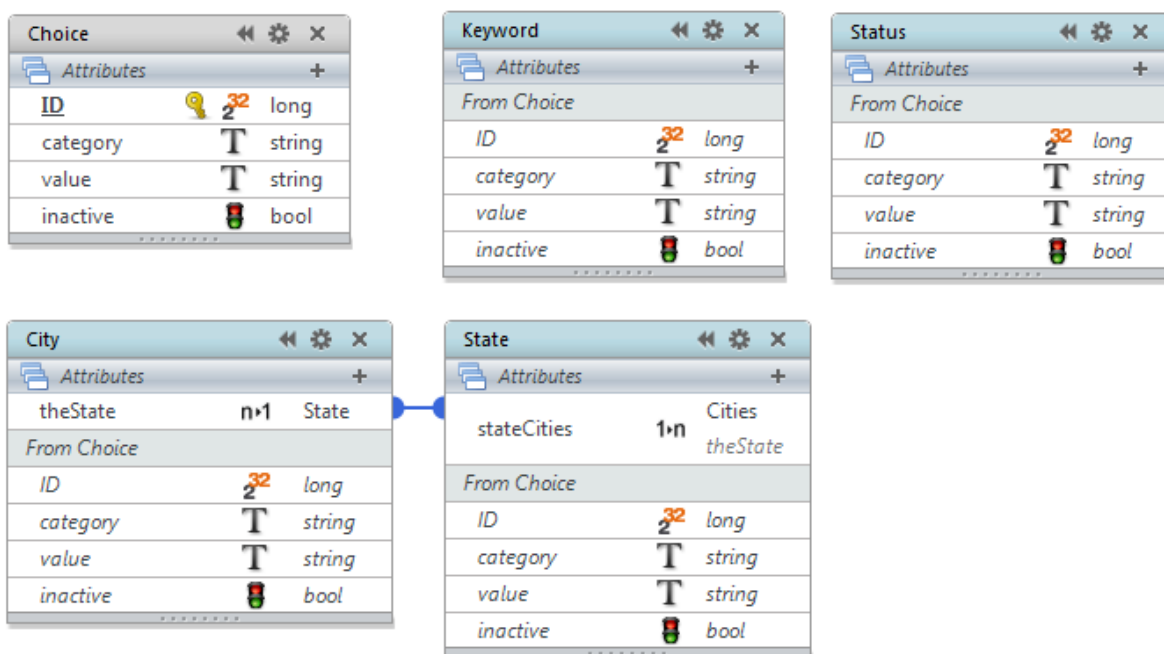


Fig 9

Typical of many data applications, this model denotes datastore classes used in support of user interface pick-list mechanisms. Choice is the base class for the other four. Keyword, Status, City, and State are all derived classes. Each of the derived classes includes a restricting query in the form of:



Notice that the *category* attribute is compared to the derived class's name. In use, each of the derived classes can be used without the need to include a filtering query at runtime. Should we need to add specific attributes to a derived class, we can do so without affecting the other types. However, if we add new attributes to the base class Choice, for example *sortOrder* or *abbreviation*, these attributes become available in all derived datastore classes. We can automate the classification of derived entities by adding a method for the *On Init* event of the Choice class as follows:

```
onInit:function()
{
    //get the name of the class of the entity
    var myType = this.getDataClass().getName();
    //store it in the category attribute
    this.category = myType;
}
```

The *On Init* event executes when an entity is created (more on this later). The above code determines the class's name and assigns it to *category*, thus auto categorizing the entity. Since all derived datastore classes inherit this behavior, the category is set when any of the derived types are created. If we create an entity in the base type Choice, it will be given a category of "Choice" but this could be changed before saving the entity. Also, notice that the derived classes State and City have a relationship between them. Derived classes can be related in ways that their corresponding base classes are not.

Of course, the inheritance scheme above implies that a single entity can be only one type such as Keyword, Status, City, State, or Choice. But Wakanda inheritance allows a single entity to be more than one inherited type. Consider the following model:

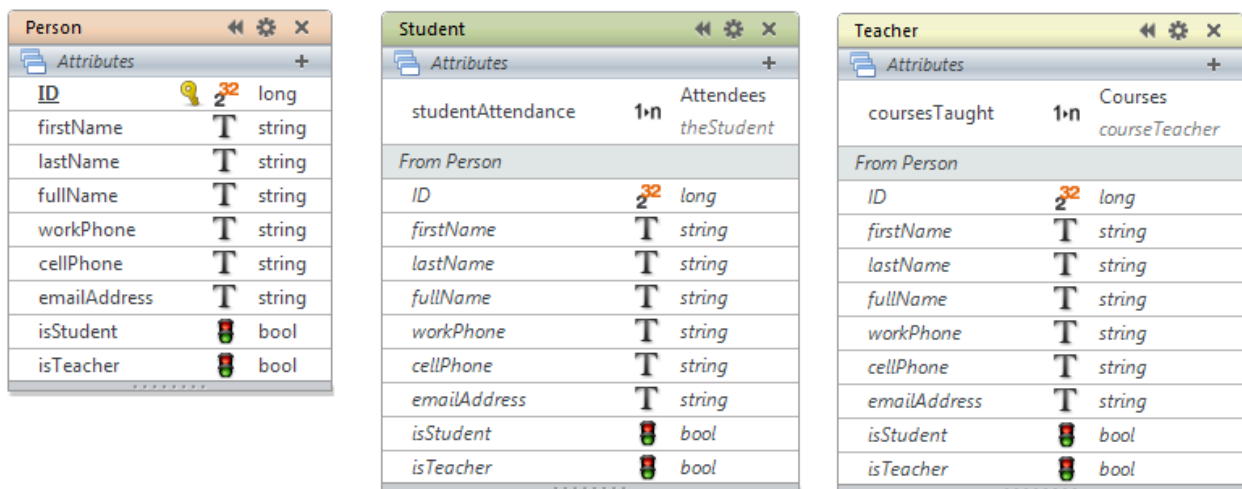


Fig 9.1

In this model, both Student and Teacher are derived from Person. Student is dependent upon the following restricting query `isStudent = True` while Teacher is dependent upon the restricting query `isTeacher = True`. Since these two values are independent, an entity in the Person class may be both a Student and a Teacher.

## On Restricting Query event

Sometimes, a restricting query string may not be sufficiently flexible. How would we allow access to multiple entities for some users but restrict access for others? A more flexible approach is to use the dataclass event *On Restricting Query*. This event's purpose is to return an entity collection that represents all entities in the dataclass. The advantage of this approach is that you can programmatically provide an entity collection which can be different for each authenticated user.

The On Restricting Query event can be used apart from inheritance and can be added to any datastore class. For example, a Sale dataclass can have as its restricting query logic that restricts sales to the current user. The advantage of doing so will ensure that no user can access sales of another user, regardless of the manner in which the information is retrieved.

## Datastore Class Methods

In addition to attributes, datastore classes may also include another type of user-defined property called methods. Methods are discreet portions of programming logic that execute when called. Datastore class methods can have one of three scopes: *Entity*, *Collection*, or *Class*. Each method's scope is defined in the Wakanda model. At runtime, class methods with a scope of *Entity* become a property of all entities of the class. In this scope, when the keyword `this` is used in the method it refers to the entity. Entity methods with a scope of *Collection* become properties of entity collections for the corresponding class. In this scope, when the keyword `this` is used it refers to the entity collection. The scope of *Collection* allows developers to build methods that operate over an entire entity collection. Datastore class methods with a scope of *Class* are essentially static methods and need no instance of an entity or entity collection to operate. In this scope, when the keyword `this` is used it refers to the class itself.

When you define a datastore class method, you can also indicate the return type using a popup in the Wakanda Datastore Model Designer. Choosing a type from the popup provides information to any mechanism that may call the method. Wakanda does not type check the results of the method against the type chosen in the popup.

## Entity Method

For an example of a method with the scope of Entity, consider the expanded model from Fig 9.1.

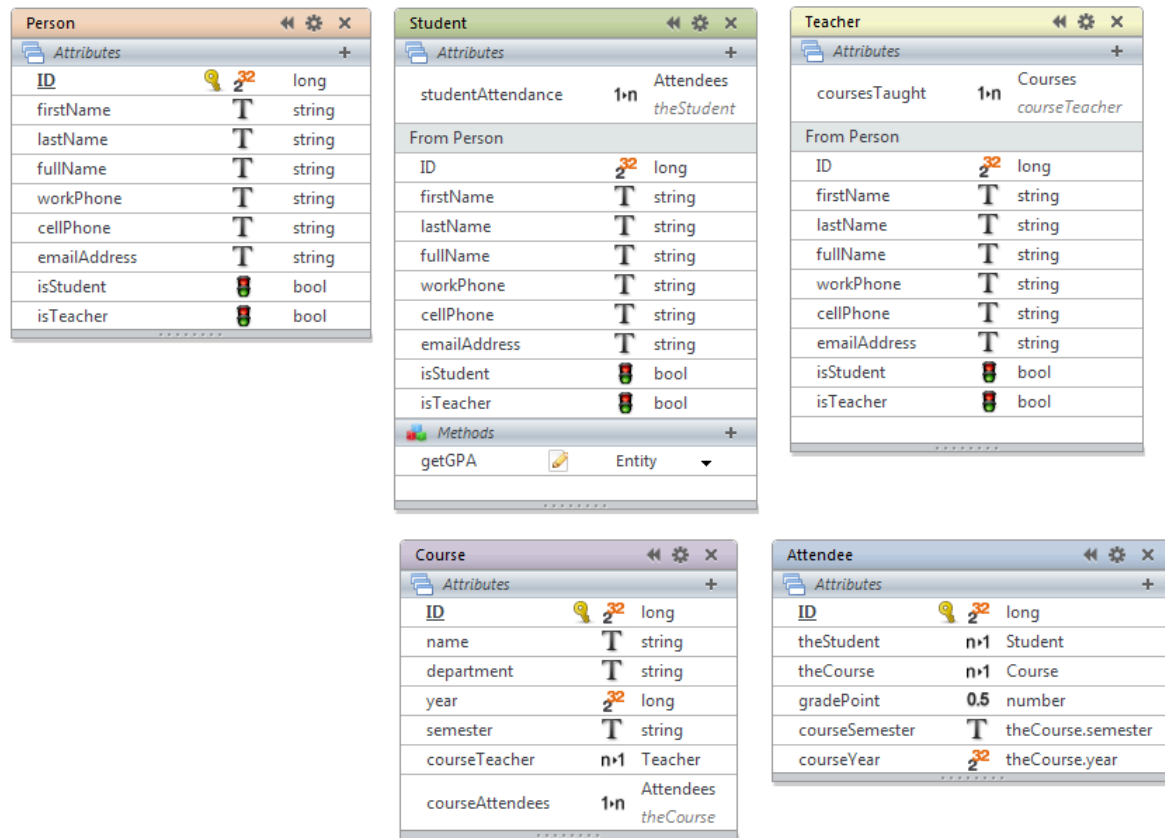


Fig 10

This model describes Student and Teacher, both derived from Person. It includes Course, the entities of which represent a specific subject and semester. Finally, it includes Attendee, each of which indicates that a particular student attended a course and received a specific grade. In this example, a method has been added to the Student class as shown. The method is named `getGPA` and its scope is *Entity*. The method is as follows:

```
getGPA:function(year)
{
    //for the current entity, get all related Attendees in year
    var courses = this.studentAttendance.query('courseYear = :1', year);
    //return the gradePoint average
    return courses.average('gradePoint');
}
```

This method accepts the parameter `year` and uses it to query the student's attendance for courses taken that year. It then averages the grades of the student. Note that using the keyword `this` refers to the student entity in which this method is called. The following code fragment shows how the `getGPA()` method might be called.

```
var theStudent = Student(321); //returns a single student entity
var theGPA = theStudent.getGPA(2011); //returns GPA for 2011
```

## Entity Collection Method

For an example of a method with a scope of Collection, we could create a new method that has code identical to the previous example. If we did so, the keyword `this` would refer to the entity collection and since relation attributes are available on entity collections as well as entities, `this.studentAttendance` would be just as valid. Instead, let's do something slightly more sophisticated. The following datastore class adds another method to Student called `listGPAs` and is given the scope of Collection:

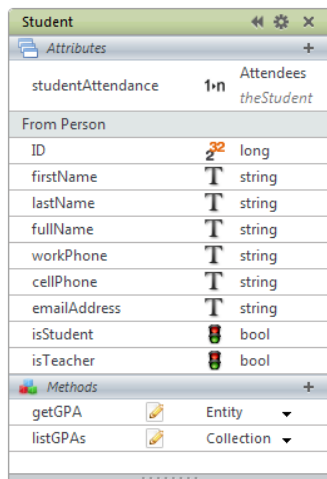


Fig 11

This new method has the following code:

```
listGPAs:function(year)
{
  var results = []; //empty array
  //in this context, this is an entity collection
  //so we can cycle through the entities using forEach
  this.forEach(function(theStudent)
  {
    results.push({
      name: theStudent.firstName + ' ' + theStudent.lastName,
      GPA: theStudent.getGPA(year));
  })
  return results; //empty array if no entities in collection
};
```

This method starts by declaring `results` as an empty array. It then uses the built-in Wakanda entity collection iterator `forEach` to cycle through all entities in the collection. For each entity, it defines an object with two attributes, `name` and `GPA` and then appends the object to the array. It fills the `GPA` for each student by calling the previously defined `getGPA` function. The result is an array of objects with two attributes each. The following code fragment shows the `listGPAs` method in action:

```
//locate all students that attended something in 2011
var stds2011 = Student.query('studentAttendance.year = 2011');
//list there 2011 GPAs
stds2011.listGPAs(2011);
```

When executed in a JavaScript file on Wakanda Server, the results look like figure 11.1.

<b>name:</b> "Elisa Getz"	<b>GPA:</b> 3.2118181818181815
<b>name:</b> "John George"	<b>GPA:</b> 2.8511111111111111
<b>name:</b> "Blanche Hughes"	<b>GPA:</b> 3.090909090909091
<b>name:</b> "Angelo Miller"	<b>GPA:</b> 3.1658333333333333

Fig 11.1

### Dataclass Method

When a method is given the scope of Class, it can be executed using a reference to the datastore class itself and requires neither an entity nor an entity collection. One good example of this is code that imports data into an empty Wakanda datastore. Consider the following Wakanda model and notice the Company method `importCompanies`.

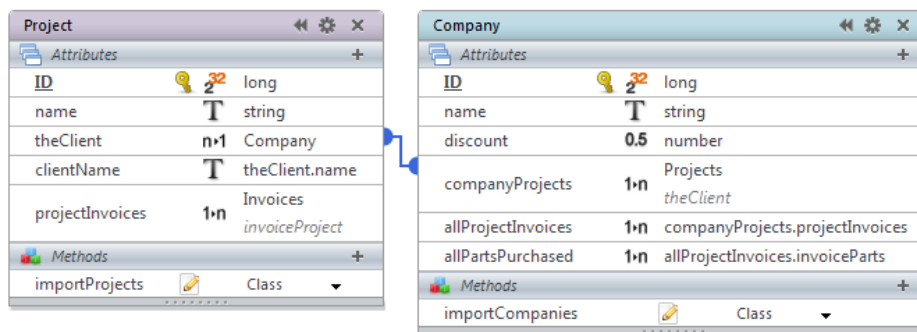


Fig 12

The `importCompanies` method might be something like this:

```
importCompanies:function(file)
{
  var importStream = TextStream(file, 'read'); //new text stream of file
  var record = importStream.read("\n"); //read first row
  if (record == 'ID\tClient\tDiscount') { //if it has the proper headers
    while (!importStream.end()) { //while not at the end of the file
      record = importStream.read("\n"); //read a row
      if (record != "") { //if we received something in record
        var fields = record.split('\t'); //split at tabs
        if (fields.length > 0) { //if row had any values
          var newCompany = new Company({ //create a new company
            ID : fields[0], //assign array elements to attributes
            name : fields[1],
            discount : fields[2]
          });
          newCompany.save(); //save the new company
        } //if
      } //if
    } //while
  }
}
```



```

    } //if
    importStream.close(); //close the text stream of the file
}

```

This method takes a file object as its parameter. It then opens a text stream to read from the file and reads the first line (“\n” is a line feed) where it expects to find column titles. If the column titles match, it reads one line at a time from the file and breaks the line into an array of values (“\t” is a tab). It then creates a new company entity and assigns values to the storage attributes. An example of this method in action might be the following:

```

var theFile = File('c:/Projects/companies.txt'); //get a file reference
Company.importCompanies(theFile); //import the file

```

The `importProjects` method of the `Project` class is similar, but must include a mechanism to find and designate the related company entity. So `importProjects` might be something like this:

```

importProjects:function(file)
{
    var importStream = TextStream(file, 'read'); //new text stream of file
    var record = importStream.read('\n'); //read first row
    if (record == 'Project ID\tProject\tProjClientID') { //check headers
        while (!importStream.end()) { //while not at the end
            record = importStream.read('\n'); //read a row
            if (record != "") { //if not empty
                var fields = record.split('\t'); //split at tabs
                if (fields.length > 0) { //if row had any values
                    var myClient = Company(fields[2]); //lookup parent company
                    var newProject = new Project({
                        ID : fields[0],
                        name : fields[1],
                        theClient : myClient //assign parent company to project
                    });
                    newProject.save(); //save the new project
                } //if
            } //if
        } //while
    } //if
    importStream.close(); //close the text stream of the file
}

```

This method is similar to the `importProjects` method except that it looks up the matching company and assigns it to the `theClient` attribute for each project. This establishes the relationship between the project and company. Notice that in order for this to work, the companies must be imported first so that they can be found during the project’s import.

But what if we have two datastore classes that are dependent on one another? For example, notice below the new attribute in `Company` named `defaultProject`.

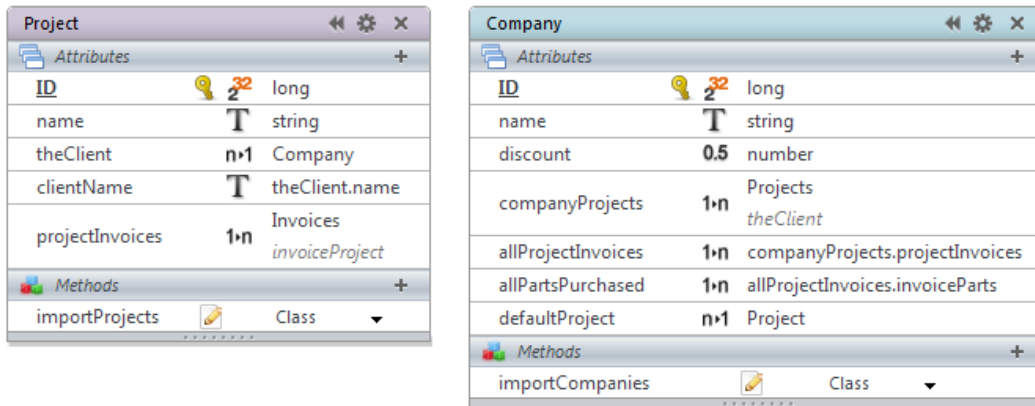


Fig 13 (screen shot will be updated)

In a relational database the *defaultProject* would typically be represented by a foreign key related to the Project *ID* attribute. Likewise *theClient* in Project would be a foreign key to the Company *ID* attribute. In this example, if we import companies first, there are no projects to be found. If we import projects first there are no companies to be found. How can we resolve this issue?

Wakanda solves this issue by allowing you to assign a key value in place of an entity. Instead of looking up an entity in another class, you can assign the imported foreign key directly to the

N->1 relation attribute. For example, the `importCompanies` method becomes (note the imported text file has an additional column representing the default project's key):

```
importCompanies:function(file)
{
  var importStream = TextStream(file, 'read'); //new text stream of file
  var record = importStream.read("\n"); //read first row
  if (record == 'ID\tClient\tDiscount\tDefProjID') { //note extra column
    while (!importStream.end()) { //while not at the end of the file
      record = importStream.read("\n"); //read a row
      if (record != "") { //if we received something in record
        var fields = record.split('\t'); //split at tabs
        if (fields.length > 0) { //if row had any values
          var newCompany = new Company({
            ID : fields[0],
            name : fields[1],
            discount : fields[2],
            defaultProject : fields[3], //note the new attribute
          });
          newCompany.save(); //save the new company
        } //if
      } //while
    } //if
    importStream.close();//close the text stream of the file
  }
}
```

Notice that the value in `fields[3]` is assigned directly to the *defaultProject* relation attribute. Wakanda converts this to an entity reference and stores the reference as part of the company. Wakanda allows this even though at the time of the assignment the corresponding

entity does not exist in the datastore. The `importProjects` method can be modified to use a similar approach.

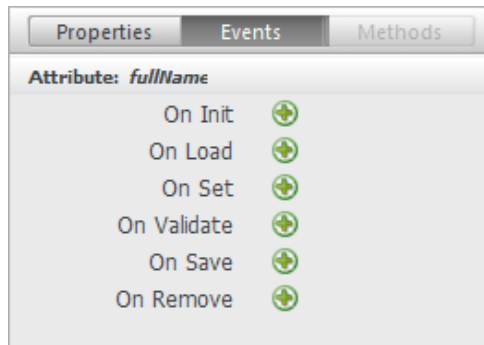
```
importProjects:function(file)
{
  var importStream = TextStream(file, 'read');    //new text stream of file
  var record = importStream.read('\n'); //read first row
  if (record == 'Project ID\tProject\tProjClientID') { //check headers
    while (!importStream.end()) { //while not at the end
      record = importStream.read('\n'); //read a row
      if (record != "") { //if not empty
        var fields = record.split('\t'); //split at tabs
        if (fields.length > 0) { //if row had any values
          var newProject = new Project({
            ID : fields[0],
            name : fields[1],
            theClient : fields[2] //assigns ID in place of entity
          });
          newProject.save(); //save the new project
        } //if
      } //if
    } //while
  } //if
  importStream.close(); //close the text stream of the file
}
```

The only issue this technique introduces is that we may create entity references where no entity exists. If we are unsure that every foreign key value has a matching record in our import file, we may want to locate orphaned entities and set their relation attributes to null.

Regardless of how we create entities in Wakanda, if we are assigning values directly to the key attribute of an entity, the auto sequence mechanism is not automatically informed of the largest key value. This implies that we need to reset the auto sequence mechanism so that the next value it produces is not a duplicate value of an existing entity. See the example in the “Programming in Wakanda Server” section for a way to address this.

## Class Events and Attribute Events

In addition to dataclass methods that are executed when called, Wakanda provides a series of events on both entities and attributes that can trigger specific code to execute. Not to be confused with the *On Get*, *On Set*, *On Sort*, and *On Query* methods of a calculated attributed, events are available on all attributes and datastore classes.



Class events operate on whole entities and are triggered by particular conditions. The following are datastore class level events:

Event	Description
<i>On Init</i>	Executes when a new entity comes into being after all auto sequence values are assigned.
<i>On Load</i>	Executes just after an entity is accessed from the datastore but before it is delivered to the calling routine.
<i>On Validate</i>	Executes when an entity is validated or saved but before the <i>On Save</i> event and can fail the validation if needed.
<i>On Save</i>	Executes before an entity is saved and can stop the save if needed.
<i>On Remove</i>	Executes before an entity is deleted and can stop the deletion if needed.
<i>On Restricting Query</i>	Executes when entities are accessed and returns an entity collection that represents all entities in the dataclass.

Like class events, attribute events run as the result of specific datastore actions. For each attribute event, Wakanda automatically passes a parameter, which is the name of the attribute affected. The following are attribute level events:

Event	Description
<i>On Init</i>	Executes when a new entity comes into being but after the entity's <i>On Init</i> event.
<i>On Load</i>	Executes the first time the attribute is accessed after the entity is loaded. If the attribute is not accessed, this event is not executed.
<i>On Set</i>	Executes after an attribute's value is set. In server-side code, this executes when the value is set. When an attribute is set in client-side code, this event executes when the entity is saved or the method <i>serverRefresh</i> is called, but only for each attribute that was modified.
<i>On Validate</i>	Executes just before the entity's <i>On Validate</i> event executes and can fail the validation if needed.
<i>On Save</i>	Executes just before the entity's <i>On Save</i> event executes and can stop the save if needed.
<i>On Remove</i>	Executes just before the entity's <i>On Remove</i> event and can stop the deletion if needed.

Regardless of whether an event method is executing for a datastore class or an attribute, the keyword `this` refers to the entity.

## On Init

The *On Init* event executes when an entity is created. In the following code fragment, we are creating a new `InvoiceItem`. The *On Init* event attached to `InvoiceItem` executes before the assignment to the variable `theItem`.

```
var theItem = new InvoiceItem(); //create a new item
```

However, when the *On Init* event executes any auto sequence assignment has already been made. In our example below, we attach an *On Init* event to a datastore class where the *ID* attribute is set to auto sequence:

```
onInit:function()
{
    this.name = 'my ID = ' + this.ID;
}
```

When an entity is created the *name* will include the *ID* since the auto sequence mechanism executes before the *On Init* event of the entity. The *On Init* event is the obvious place to initialize an entity. Typically you would assign default values to attributes or in the case of derived classes you may test for which type of entity is being created (see the inheritance restricting query example).

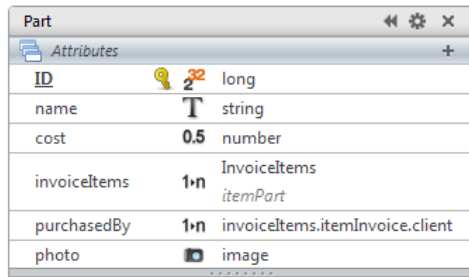
The *On Init* event for attributes is very similar and executes just after the entity's *On Init*. The main difference is that the event is provided a string parameter that is the name of the attribute. This allows developers to write modular code that can operate on a variety of attributes. Using a combination of the keyword `this` and the parameter `attributeName` you can indirectly reference the attribute. The syntax used to reference an entity's attribute by its name is shown below:

```
onInit:function(attributeName) //attributes receive the attributeName
{
    this[attributeName] = ""; //assigns empty string to the attribute
}
```

All attribute events are provided the `attributeName` parameter.

## On Load

The *On Load* event executes when an entity is loaded from the datastore. This provides a control point where the entity attribute's values can be modified before the entity is delivered to the requesting mechanism. For example, consider the following datastore class:



Part		
Attributes		
ID	232	long
name	T	string
cost	0.5	number
invoiceItems	1-n	InvoiceItems itemPart
purchasedBy	1-n	invoiceItems.itemInvoice.client
photo		image

Fig 14

In this example from Figure 6, a new attribute of type Image has been added to the Part entity. This attribute is intended to display a picture of the part, but in our example application not all parts have photos. The *On Load* method might look like this:

```
onLoad:function()
{
  if (this.photo.size == 0)
    this.photo = loadImage('c:/Projects/Docs/WebFolder/NoPhoto.jpg');
}
```

In this code, the “NoPhoto.jpg” image file is used for parts that have no photo. The `loadImage()` function is a Wakanda global method that loads an image from a file. Since this code runs before the entity is delivered, parts without photos will have the default photo when accessed.

For attributes the *On Load* event executes when the attribute is first accessed. To see this in action, consider the following Wakanda Server code fragment:

```
var myParts = Part.query('ID < 10'); //locates all parts with ID < 10
var myEntity = myParts[0]; //On Load for entity executes
var myName = myEntity.name; //On Load for attribute executes
var myName2 = myEntity.name; //On Load does NOT execute again
```

The first line of code assigns an entity collection of parts to `myParts`. Since no particular entity was accessed, the *On Load* entity event for Part does not execute. The second line of code assigns a specific part to `myEntity`. Since a specific entity was accessed, the *On Load* event for the entity is executed but no *On Load* events occur for the entity’s attributes. The third line of code assigns the value in the entity’s name attribute to `myName`. Since a specific attribute was accessed the *On Load* event for that attribute executes. The fourth line of code will not cause the *On Load* event for `name` to execute again because the attribute has already been loaded for the entity.

**Note:** Even assignments to attributes will cause the *On Load* event to execute on that attribute. For example:

```
myEntity.name = 'Part 1'; //On Load for attribute executes
```

This code will cause the *name On Load* event to execute before its *On Set* event.

## On Set

The *On Set* event is only available for attributes. For an example of an attribute event consider the following classes from Figure 15.

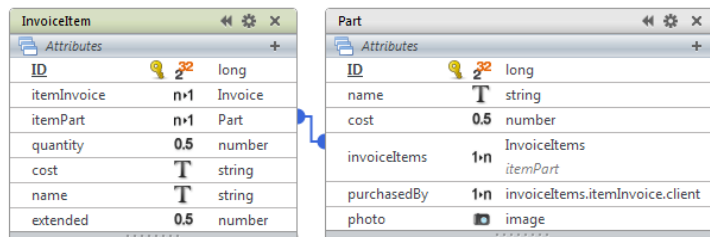


Fig 17

Notice that the alias attributes *partCost* and *partName* in the *InvoiceItem* class have been replaced with the storage attributes *cost* and *name*. In our example application, this is not an instance of denormalization because our business rules state that an invoice item acquires the part name and cost at the time the part is specified and thereafter is independent from the part. The rules indicate that future invoice items can reference the same part with a different name or cost. This is a common application situation where values from a parent entity need to be copied into a related entity. This can be accomplished using the *On Set* event for *itemPart* in *InvoiceItem*.

The method might look something like this:

```
onSet:function(attributeName)
{
    this.name = this.itemPart.name;
    this.cost = this.itemPart.cost;
}
```

Later, when a part is assigned to *itemPart*, the values will be copied from the part into the invoice item. For example, consider the following code fragment:

```
var theItem = new InvoiceItem(); //create a new item
var thePart = Part(324); //find part with a key of 324
theItem.itemPart = thePart; //entity is assigned so On Set runs
//at this point the new item has copied the name and cost
theItem.save();
```

In an attribute's event method, the keyword **this** refers to the entity, not the attribute. In the case of the *On Set* event, the method runs after the attribute's value has been assigned therefore the attribute *itemPart* is a valid entity reference during the *On Set* event code.

All entity and attribute events are processed on Wakanda Server. This means that assignments made in code running on the browser will not cause *On Set* events until the entity is returned to the server by either `save()` or `serverRefresh()`. When this happens, only the attributes that have had values assigned will receive *On Set* events.

Attribute events execute even when values are assigned inside of other attribute events. In the above *On Set* example, if there is an *On Set* method for either the invoice item's name or cost, the events for those attributes are executed.

The same is true for datastore class level events; code in one event may cause another event to execute. For example, if the class *On Init* event contains the following code:

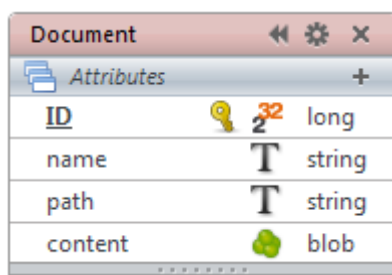
```
onInit:function()
{
    this.postDate = new Date();
}
```

In this code the assignment to *postDate* will cause other events to execute, namely the *postDate* attribute's *On Load* and *On Set* events. These additional events will complete before the original entity's *On Init* event. It is not hard to envision a situation where an event's code ends up causing the same event's code to run again. Wakanda Server manages this by ensuring that each chain of event calls never executes the same event twice for the same entity or attribute.

## On Validate

The *On Validate* event occurs when an entity is validated or saved but before the *On Save* event. The purpose of *On Validate* is to provide a mechanism to test whether an entity is complete and qualified to be saved before proceeding. This can be particularly useful when your *On Save* event contains code that accesses external services or external files. Without this event, the *On Save* event may proceed and complete part of its intended function before running into a validation issue, leaving the data in an inconsistent state. It is good practice to put all business validation logic in the *On Validate* and reserve the *On Save* for actions on the entity being saved, writing to other entities or external files, calling external services and reporting on the most egregious errors.

For example, consider the following structure:



Document			
Attributes			
ID	Key	32	long
name	T		string
path	T		string
content	blob		blob

Fig 15

Let's assume that our business rules dictate that a Document entity cannot be saved without a name and content, but when it is saved the content is written to an external file and not stored in the datastore. Our *On Validate* might be something like this:

```
onValidate:function()
{
    if (this.name == null) //no name
        return {error: 3, errorMessage: 'no name specified'};
    else if (this.content.length == 0) //no content
```



```

    return {error: 4, errorMessage: ' document contains no content'};
  else {
    var modelPath = ds.getModelFolder().path; //path to project folder
    var docFolder = Folder(modelPath + 'Documents');
    if (!docFolder.exists) //check for subfolder named Documents
      try {
        docFolder.create(); //if not there, create it
      }
      catch (e) { //if you can't create it, then error out
        return {error: 5, errorMessage: ' could not create doc folder'};
      }
    if (docFolder.exists) {
      if (docFolder.getFreeSpace()/1024 < 100) //check for drive space
        return {error: 6, errorMessage: ' drive space is low'};
    }
  }
}

```

This code tests if the entity meets some simple business rules (name must be supplied and document must have something in it) and then checks to make sure the folder where documents are to be written exists and that there is a minimum amount of hard disk space. If anything in the validation goes wrong, the validation fails. This event would typically be paired with an *On Save* event where the document is written to the “Documents” folder, the path to the document is stored in the *path* attribute, and the *content* attribute is cleared and therefore not stored in the datastore. The code to save a document entity might look something like this:

```

var newDoc = new Document()
//some code
try {
  newDoc.save(); //try to save the document
}
catch (e) {
  //may be either a validation or save error
}

```

Keep in mind that the *save()* method may produce errors from either the *On Validate* event or the *On Save* event so your code must handle exceptions from both.

An attribute’s *On Validate* event executes before the entity’s *On Validate* event and is supplied the attribute’s name as a string parameter. For example:

```

onValidate:function(attributeName)
{
  return hasSomeText(this, attributeName);
}

```

Where the function *hasSomeText()* holds the following code:

```

hasSomeText:function(entity, attributeName)
{
  if (entity[attributeName] == null) || (entity[attributeName].length == 0) {
    return {error: 7, errorMessage: '' + attributeName + ' has no value'};
  }
  else {

```

```

        return {error: 0}; //Same as no error
    }
}

```

Note that returning an object where the `error` attribute equals zero is the same as returning no error.

## On Save

The *On Save* event functions similar to the *On Validate* but only executes if the entity passes validation. If an entity has both *On Validate* and *On Save*, then the *On Validate* executes first. If an entity has no *On Validate* method then the *On Save* will execute whenever the `save()` method is called on an entity. However, the *On Save* event can still reject the save by returning an error object just like the *On Validate*. Designers should typically reserve the *On Save* for work needed in tandem with saving an entity. Consider this datastore model:

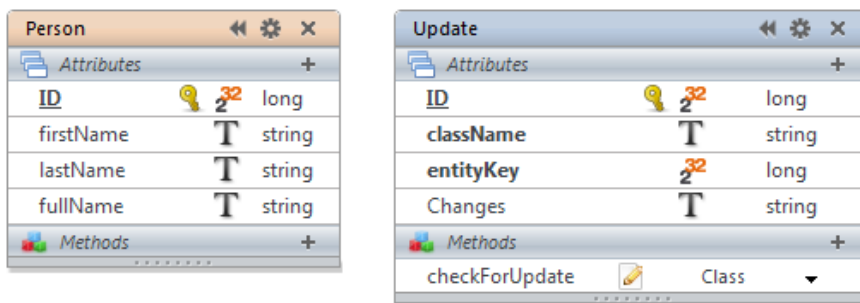


Fig 16

The `checkForUpdate()` method has the scope of `Class` and the following code:

```

checkForUpdate:function(theEntity)
{
    if (!theEntity.isNew()) { //if theEntity is not new
        var theClass = theEntity.getDataClass(); //get its class
        //using its key, get a reference to the entity before it was updated
        var formerEntity = theClass(theEntity.getKey());
        var changedFrom = ""; //will hold a list of modifications
        for (var e in theClass.attributes){ //cycle through all attributes
            if (theEntity[e] != formerEntity[e]) //if the attribute changed
                //append to the list of changes the previous value
                changedFrom += e + ' : ' + formerEntity[e] + '\r';
        }
        if(changedFrom.length > 0) { //if anything was modified
            new Update({ //keep a record of the changes
                className: theClass.getName(),
                entityKey: theEntity.getKey(),
                changes: changedFrom
            }).save();
        }
    }
}

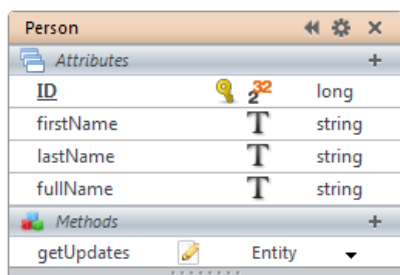
```

This class method is a generic routine for recording all changes to an entity. It checks to see if the entity being saved is new. If not, it looks up the same entity in the datastore, which

returns a copy of the entity before it was modified. It then cycles through all the entity's attributes testing each one to see if it has been changed and writing the changes to a variable. If there are any changes, it creates a new Update entity to record the modifications along with the class name and key. Incorporating it into the Person's *On Save* event is as simple as:

```
onSave:function()
{
    Update.checkForUpdate(this);
}
```

We can even add a new method to the Person class with a scope of *Entity* that returns the corresponding updates. For example, here is the same class with a new method named `getUpdates()`.



Notice that the new method has the scope of *Entity*. The `getUpdates()` method returns an entity collection of Updates and has the following code:

```
getUpdates:function()
{
    var className = this.getDataClass().getName(); //returns 'Person'
    //build a query string using the key and class name
    var qString = 'entityKey = :1 AND className :2 order by ID desc';
    //find the Updates for the current entity
    var theUpdates = Update.query(qString, this.getKey(), className);
    //return the entity collection of updates
    return theUpdates;
}
```

To use this new method you would call it as a property of an entity. The result would be an entity collection of updates that show the modifications in reverse order of creation so that the latest modification comes first. For example:

```
var somePeople = Person.query('lastName = "Brown"');
var onePerson = somePeople[0];
var theUpdates = onePerson.getUpdates(); //returns sorted collection
```

Of course this code assumes that we have at least one person with a last name of "Brown."

## On Remove

The *On Remove* event executes just before an entity is to be deleted. It can be used for a variety of purposes including cleaning up related entities and validating the deletion. To reject the deletion, you pass a specific object as the result of the function. Take the Part class from Fig 14 and the following *On Remove* event method.

```

onRemove:function()
{
    if (this.invoiceItems.length != 0)
        return {error: 1, errorMessage: ' Part in use by invoice items'}
}

```

In this example, our business rules dictate that we cannot delete a part that is still in use. This method rejects the deletion of a part that is still being referenced by an invoice item. You reject the deletion by returning an object with at least one property named *error*. The *error* property should be a number greater than zero. If it is greater than zero, then the deletion will not occur. The number you choose is up to you. Optionally you may include the *errorMessage* property, which is a string to describe the error.

But what if we wanted to just delete the corresponding invoice items when the part is deleted (unlikely, but let's use it for an example)? Our *On Remove* code might look something like this:

```

onRemove:function()
{
    if (this.invoiceItems.length != 0){
        try {
            ds.startTransaction();    //start a transaction
            this.invoiceItems.remove(); //attempt to delete the invoiceItems
            ds.commit();              //if we get here then all went well, so commit
        }
        catch (e){
            var theError = {
                error: 2,
                errorMessage: ' Related invoice items could not be deleted'
            };
            ds.rollback(); //rollback the transaction
            return theError; //return the error
        }
    }
}

```

Notice the use of the `try...catch` error-trapping block to attempt to delete the related invoice items. In this example we start a transaction before we attempt to delete the related entities. If deleting the invoice items generates an error, then the line `ds.commit()` will be skipped and control will pass directly to `catch` where we roll back the transaction. In this way, the invoice items deletion is an all or nothing mechanism. If there is an error deleting the invoice items, then our part's deletion generates an error as well.

## Programming in Wakanda Server

As you may have noticed, Wakanda Server uses JavaScript as its programming language. This allows developers to utilize a single language for both browser-based and server-based code. The Wakanda Server JavaScript framework provides a variety of services to manipulate datastore classes, entity collections, entities, and many other server-side constructs.

When code runs on Wakanda Server it does so in the context of a Wakanda application. In this context, `Application` is a global object so all of its properties are available with no

prefix. One of the application object's properties is `ds`, which represents the default datastore of the application. When programming in Wakanda server, many references begin with `ds`.

## Datastore Classes

All datastore classes in an application are available as a property of `ds`. For example,

```
var personClass = ds.Person;
```

This code assigns to `personClass` a reference to the `Person` datastore class. For dataclasses set as global access in the data model, you need not include the `ds`. For example,

```
var personClass = Person;
```

As you may have noticed, most of the method examples above referenced dataclasses as global objects. From here on we will reference dataclasses as properties of `ds`.

A list of all datastore classes is also available as a property of `ds`. For example,

```
function resetAllAutoSequences(){ //datastore class method
  for (var e in ds.dataClasses){ //cycle through all classes in ds
    var theClass = ds.dataClasses[e]; //get a reference to a class
    var maxID = theClass.all().max('ID'); //find the largest value
    class.setAutoSequenceNumber(maxID + 1); //reset the auto sequence
  }
}
```

This code resets all auto sequence values for all datastore classes, provided the key for each class is left at the default name of "ID." The method `setAutoSequenceNumber` is a standard method of datastore classes.

## Datastore Class Attributes

Datastore class attributes are available as properties of their respective classes. For example:

```
var lastNameAttribute = ds.Person.lastName; //reference to class attribute
var firstNameAttribute = ds.Person['firstName']; //alternate way
```

This code assigns to `lastNameAttribute` and `firstNameAttribute` references to the `lastName` and `firstName` attribute of the `Person` class. This syntax does not return values held inside of the attribute, but instead returns references to the attributes themselves. Note the two different ways to reference class attributes. You can cycle through all the attributes of a datastore class like this:

```
var personClass = ds.Person; //reference to class, not a collection
var allPeople = personClass.all(); //reference to entity collection
var attribCount = {}; //empty object
for (var e in personClass.attributes){ //cycle through class attributes
  var attribute = personClass.attributes[e]; //reference to class attribute
  //use class attribute and allPeople to count non-nulls
  attribCount[attribute.getName()] = allPeople.count(attribute);
}
```

This code first gets a reference to the Person class and then a reference to an entity collection of all people. It then creates an empty object that will house the non-null counts of each attribute in people. It then cycles through all the attributes in the Person class and uses the name of the attribute to implicitly add a new property for each attribute that stores the value of the count. Note that the `count()` function can take either a reference to an attribute or a string name of an attribute.

If we were to run this directly on Wakanda Server and display the `attribCount` object, we might see the following result:

```
ID: 5000
firstName: 5000
lastName: 5000
address: 5000
city: 5000
state: 5000
zip: 5000
phoneNumber: 5000
isStudent: 2275
isTeacher: 70
```

## Attribute Names and Attribute References

Many Wakanda methods accept string references as attribute names and can also accept attribute references. For example, consider the following:

```
var localPeople = ds.Person.query('zipCode = 95113');
var lastNames = localPeople.toArray('lastName');
```

This code defines an entity collection of people in the 95113 zip code. It then produces an array of last names. In place of a string value representing an attribute, you may also use an attribute reference like this:

```
var lastNameAtt = ds.Person.lastName; //reference to class attribute
var localPeople = ds.Person.query('zipCode = 95113');
var lastNames = localPeople.toArray(lastNameAtt); //uses attribute ref
```

## Creating Entities

New entities can be created using the datastore class name. For example:

```
var newPerson = new ds.Person();
```

This code creates a new entity of the Person datastore class. Alternatively, the same entity could be created using the following syntax:

```
var newPerson = ds.Person.createEntity();
```

This allows a more generic form of entity creation. For example:

```
var name = 'Person';
var newPerson = ds.dataClasses[name].createEntity();
```

Of course, the entity is still to be saved. A more complete example might look like:

```
var newPerson = new ds.Person();
newPerson.first = 'Fred';
newPerson.last = 'Williams';
newPerson.save();
```

Or you can pass an object as a value to the constructor, like this:

```
var newPerson = new ds.Person({
    first : 'Fred',
    last : 'Williams'
});
newPerson.save();
```

You need not even assign the entity to a variable if all you want to do is create a new one:

```
new ds.Person({first : 'Fred', last : 'Williams'}).save();
```

Or similarly:

```
ds.Person.createEntity({first : 'Fred', last : 'Williams'}).save();
```

## Entity Collections

Entity collections are usually created using a query or returned from a relation attribute. For example:

```
var brokers = ds.Person.query('personType = broker');
```

This code returns into `brokers` all the people of type `broker`. To access an entity of the collection, use syntax similar to accessing an element in an array. For example:

```
var theBroker = brokers[0]; //Entity Collections are 0 based
```

The entity collection method `orderBy()` returns a new entity collection according to the supplied sort criteria. For example:

```
brokers = brokers.orderBy('name'); // returns a sorted collection
```

This code returns into `brokers` the same collection of person entities but sorted by name. Alternatively, you can use a relation attribute to return an entity collection. For example:

```
var brokers = ds.Person.query('personType = broker');
var brokerCompanies = brokers.myCompany;
```

This code assigns to `brokerCompanies` all related companies of the people in the entity collection `brokers` assuming the relation attribute `myCompany`. Using relation attributes on

entity collections is a powerful and easy way to navigate up and down the chain of related entities.

## Queries and Finds

A Wakanda query returns an entity collection. An entity collection may have no entities. A Wakanda find returns a single entity, the first one found if more than one matches the criteria or `null` if no entities match the query. Both use the same syntax and both can be very simple. For example:

```
var lowParts = ds.Part.query('ID < 100');
```

This query returns into `lowParts` an entity collection of all parts with *ID* less than 100. Notice that `query()` is a method of the Part datastore class. When a query is performed on a datastore class, all entities of that class are included in the query. But consider the following code:

```
var lowParts = ds.Part.query('ID < 100');
var lowWidgetParts = lowParts.query('Name = Widget');
```

The first line returns an entity collection that is queried further in the second line. A query performed on an entity collection only considers entities that are included in the collection. Thus, `lowWidgetParts` would be an entity collection of all parts named “Widget” with an *ID* of less than 100. Notice that the value “Widget” is not in quotes in the query string. This is allowed provided the value contains no spaces or other characters that would cause ambiguity. For example, we would need to write the following if the part’s name is “Large Widget”:

```
var lowWidgetParts = lowParts.query('Name = "Large Widget"');
```

Of course, queries can be more complex than this. But since datastore classes can include alias and relation attributes, even complex queries are easy to write and understand. For example, using the Wakanda model from Fig 6 consider the following query examples:

```
var entColl = ds.Project.query('ID < 1000 and clientName = "Brown"');
```

This finds all projects with an *ID* less than 1000 where the client’s name is “Brown.” Notice the use of the alias attribute *clientName*, which behaves as any other attribute in Project. Wakanda manages the relation between the two classes. Similarly, we can use relation attributes in queries. For example:

```
var entColl = ds.Invoice.query('client.discount < 30');
```

This query returns all invoices where the client’s discount is less than 30. The relation attribute *client* represents the parent project’s parent company. Notice how simple this query is to its SQL equivalent where we would have to incorporate two joins in our statement.

Even more interesting are queries using 1->N relation attributes. For example,

```
var coll = ds.Company.query('name = "B*" and companyProjects.name = "Green"');
```



This query returns all companies that have a name that starts with “B” and have at least one related project with the name of “Green.” The “\*” character is the Wakanda wildcard character. When you include more than one reference to a 1->N relation attribute in a query then the conjunctions OR and AND control whether the criteria is applied to the same related entity or independently. For example:

```
var qString = 'companyProjects.name = "Green";  
qString += ' AND companyProjects.ID < 100';  
var result = ds.Company.query(qString);
```

This query returns all companies that have at least one project with the name of “Green” and an *ID* below 100. One single related entity must meet both criteria for the company to be included in the resulting collection. If we replace the “AND” with “OR” then the query finds all companies that have at least one project with the name of “Green” or another project with an *ID* less than 100.

Wakanda queries also allow for keyword searching. For example:

```
var myDocs = ds.Document.query('docText %% "Project"');
```

This query will return into `myDocs` an entity collection of documents where the *docText* attribute contains the individual word “Project.” Note the keyword comparison operator “%%”. This type of query will work correctly on the *docText* attribute regardless of whether it was indexed or not. But if we add a Keywords index to the *docText* attribute, the query will operate much faster. Using keyword queries is different from using a simple contains query. For example, consider this similar query:

```
var myDocs = ds.Document.query('docText = "*Project*"');
```

This query will not only find documents with the word “Project,” but also those with the word “Projection.” In fact, it will locate all documents where the string of characters is found anywhere in *docText*.

Wakanda interprets query strings from left to right. This means that neither ANDs nor ORs take precedence. Use parenthesis in queries to control the order of execution. For example:

```
var qString = '(cost > 50 and cost < 100) or name = Widget'  
var myParts = ds.Part.query(qString);
```

The above code might return a different entity collection than the following:

```
var qString = 'cost > 50 and (cost < 100 or name = Widget)'
```

### *Placeholder Query Syntax*

As an alternative, queries can be written with placeholder syntax. For example,

```
var myComps = ds.Company.query('name = :1', name);
```

Note “:1” in the query string. Each placeholder is included as a colon followed by a digit. Up to nine placeholders can be included in the query string and then supplied as parameters to the query method. There are several advantages to this syntax:

- 1) You don't have to build a complex query string that includes values from other variables or objects.
- 2) You don't have to put quotes around string values that have spaces or other characters.
- 3) This syntax is necessary if you wish to use an entity as a value in the query (as opposed to one of its attributes).

For an example of using an entity as a value in a query consider the following:

```
var myPart = ds.Part(100);
var myPartItems = ds.InvoiceItem.query('itemPart = :1', myPart);
```

Note how `myPart` (an entity) is used as the value in the query. After the query, `myPartItems` would be an entity collection of all invoice items related to `myPart` through the `itemPart` relation attribute.

You may also query using 1->N relation attributes. For example:

```
var myPart = ds.Part(100);
var invoices = ds.Invoice.query('invoiceItems.itemPart = :1', myPart);
```

This query will return an entity collection into `invoices` containing all invoices that have at least one invoice item whose `itemPart` is `myPart`.

The `find()` method uses an identical syntax to the `query()` but differs in what it returns. For example,

```
var Fred = ds.Person.find('name = "fred"); //Returns one entity
var Freds = ds.Person.query('name = "fred"); //Returns a collection
var aFred = Freds[0] //Returns an entity, but remembers the collection
```

The first line returns only one entity. If there are more than one “fred” in the example above then `find()` returns the first one. You should not depend on which entity will be returned by `find()` when more than one entity matches the criteria.

The second line returns an entity collection of all people named “fred.” It then assigns to the variable `aFred` a reference to the first entity in the collection. Note that functionally the variable `Fred` and `aFred` are different. `Fred` has no underlying reference to an entity collection while `aFred` is a reference to a specific entity in the `Freds` entity collection. This allows the entity `aFred` to iterate through the `Freds` entity collection using the method `next()`.

**Note:** Using an entity reference as a value in a query is not completed.

## Locating Entities by Key

Similar to the `find()` method a single entity may be looked up by key value. For example,

```
var thePerson = ds.Person(100);
```

This code assigns to `thePerson` an entity that has the key of 100. The converse of this is the `getKey()` function for entities. For example,

```
var mike = ds.Person.find('name = "mike"); //Returns one entity
var mikeKey = mike.getKey() //Returns the entity key
```

You can also locate an entity by supplying an object as a value. For example:

```
var aPerson = ds.Person({firstName: 'Brad', lastName: 'Wilson'});
```

This code assigns to `aPerson` a person whose first name is “Brad” and whose last name is “Wilson.” This is functionally identical to

```
var aPerson = ds.Person.find('firstName = Brad & lastName = Wilson');
```

Just like `find()`, if more than one entity matches the criteria then the first one is returned.

*Note: Wakanda will support multi-attribute keys in the future. When `getKey()` is called on an entity with a multi-attribute key, it will return an array of values that make up the key. To locate an entity with a multi-attribute key, you will provide either an array of values or a series of parameters.*

## Using Nulls in Queries

As you would expect the keyword *Null* plays an important role in Wakanda queries. For example:

```
var badInvoices = ds.Invoice.query('invoiceItems = Null');
```

This query returns a collection of all invoices that have no invoice items. Similarly:

```
var orphanItems = ds.InvoiceItem.query('itemPart = Null');
```

This query returns all invoice items that have no related *itemPart*. This includes items that have never been assigned a related part as well as items whose part has been deleted.

## Conjunctions

Wakanda lets you build queries using the following conjunctions.

Conjunction	Aliases	Notes
AND	& or &&	Logical AND
OR	or	Logical OR
NOT	!	Logical negation. Operates on next criteria or group of criteria if inside of parenthesis
EXCEPT	^	Equivalent to AND NOT

## Comparison Operators

Wakanda supports the following comparison operators. Generally, all Wakanda queries are case insensitive.

Operator	Aliases	Notes
=	eq or like	Allows for wildcard support using “*”
!=	#	Not like, allows for wildcard support
==	is or eqeq	Exactly equal, no wildcard support
!=	nene, isnot, or ##	Not equal, no wildcard support
>	gt	Greater than
>=	gteq or gte	Greater than or equal to
<	lt	Less than
<=	lteq or lte	Less than or equal to
%		Contains keyword, may use a keyword index
=%	matches or %*	Matches a JavaScript regex expression
!=%	!%*	Does not match regex expression

## Using JavaScript in Queries

Most queries can be handled by the syntaxes above. But in some cases a query is needed that doesn't follow the *attribute/comparison/value* methodology. What if we want to find all entities with an even numbered ID? Or what if we want to find all the people who have a last name with a length of 10? To do this, we can incorporate JavaScript expressions that return a Boolean value in our queries. For example:

```
var evenIDs = ds.Part.query('$(this.ID % 2)= 0');
var lastName10 = ds.Person.query('$(this.lastName.length = 10)');
```

To inform Wakanda that a portion of a query is to be treated as a JavaScript expression, you prefix it with a dollar sign (\$) and enclose it in parenthesis. In order to reference any attributes of the query's datastore class you need to use the keyword `this` so that Wakanda will correctly interpret identifiers. If you don't use the keyword `this`, then Wakanda interprets the identifier as a JavaScript variable. For example:

```
var divisor = 2;
var someParts = ds.Part.query('$(this.ID % divisor)= 0');
```

Notice that the variable `divisor` is assigned before the query and is then used as part of the JavaScript expression.

## Relation Attributes and Entity Collections

In addition to the variety of ways you can query, you can also use relation attributes as properties of entity collections to return new entity collections. For example:

```
var myParts = ds.Part.query('ID < 100'); //Return parts with ID less than 100
var myInvoices = myParts.invoiceItems.itemInvoice;
// All invoices with at least one line item related to a part in myParts
```

The last line will return in `myInvoices` an entity collection of all invoices that have at least one invoice item related to a part in the entity collection `myParts`. When a relation attribute is used as a property of an entity collection, the result is always another entity collection, even

if only one entity is returned. When a relation attribute is used as a property of an entity collection and no entities are returned, the result is an empty entity collection, not null.

## Scalar Attributes and Entity Collections

All scalar attributes are available as properties of entity collections as well as entities. When used in conjunction with an entity collection a scalar attribute returns an array of scalar values. For example,

```
var locals = ds.Person.query('city = "San Jose"); //collection of people
var localEmails = locals.emailAddress; //array of email addresses (strings)
```

This code returns into `localEmails` an array of email addresses as strings.

## Entity Collections and toArray()

All entity collections include a function called `toArray()`. This flexible function provides a way to return an array of objects of varying complexity. For example, assume the following structure:

The image shows three screenshots of entity models in a software interface. Each screenshot displays a list of attributes for a specific entity, including their names, data types, and relationships to other entities.

Entity	Attribute Name	Cardinality	Target Entity	Data Type
Attendee	ID	1		long
	theStudent	n:1	Student	
	theClass	n:1	Class	
	gradePoint	0.5		number
	grade	T		string
	studentName	T		string
	Class	ID	1	
theCourse		n:1	Course	
schedule		T		string
location		T		string
classTeacher		n:1	Teacher	
classAttendees		1:n	Attendees	theClass
semester		T		string
courseCode		T		theCourse.code
courseName		T		theCourse.name
courseDepartment		T		theCourse.department
GPA		0.5		number
teacherName		T		string
Course	ID	1		long
	name	T		string
	department	T		string
	courseClasses	1:n	Classes	theCourse
	code	T		string
	courseTeachers	1:n	courseClasses.classTeacher	

This model indicates that a course entity may have many classes (*courseClasses*) and that a class entity may have many attendees (*classAttendees*). If we query for classes and apply the `toArray()` function to the resulting collection without specifying a parameter then we receive back an array of objects, one object for each entity in the collection.

The structure of each object is the default for the datastore class and follows these rules:

- Storage, calculated, and alias attributes are returned as properties on each object.
- N->1 relation attributes are returned as a property, which itself has one property called `__KEY` made up of two properties: `ID` and `__Stamp`. In future versions of Wakanda, `__KEY` may have more properties.
- 1->N relation attributes are returned as a property having one property named `__Count`.

These three rules result in a visual representation like this:

<b>ID:</b> 4251	<b>theCourse:</b> <div style="border: 1px solid blue; padding: 5px; margin: 5px;"> <b>__KEY:</b> <div style="border: 1px solid blue; padding: 5px; margin: 5px;"> <b>ID:</b> 20  <b>__STAMP:</b> 1 </div> </div>	<b>classAttendees:</b> <div style="border: 1px solid blue; padding: 5px; margin: 5px;"> <b>__COUNT:</b> 198 </div>
<b>ID:</b> 4252	<b>theCourse:</b> <div style="border: 1px solid blue; padding: 5px; margin: 5px;"> <b>__KEY:</b> <div style="border: 1px solid blue; padding: 5px; margin: 5px;"> <b>ID:</b> 26  <b>__STAMP:</b> 1 </div> </div>	<b>classAttendees:</b> <div style="border: 1px solid blue; padding: 5px; margin: 5px;"> <b>__COUNT:</b> 189 </div>

So, referring to the model above, consider this code:

```

var myClasses = ds.Class.query('ID < 5'); //Class is reserved so need ds
var classesArray = myClasses.toArray(); //default values
var oneClass = classesArray[0] //one class object (not an entity)
var theID = oneClass.ID //scalar value of one object
var theCourseCode = oneClass.courseCode //value from alias attribute
var courseKey = oneClass.theCourse.__Key //key object of related entity
var courseID = courseKey.ID //ID of related entity
var courseStamp = courseKey.__STAMP //Mod stamp of related entity
var attendeeCount = oneClass.classAttendees.__COUNT //Just the count

```

First thing to note is that the datastore class named `Class` is prefixed with `ds`. Most datastore classes can be set for global access without a prefix but in the case of one named “Class” you might need to reference it as a property of `ds` since `Class` is a reserved word. This example illustrates the default values returned by `toArray()`. In most cases you will specify a parameter or parameters for the `toArray()` method and this will affect the objects returned. So using the same structure let’s look at the following code:

```

var myClasses = ds.Class.query('ID < 5');
var classesArray = myClasses.toArray("ID, courseCode");
var oneClass = classesArray[0]; //one class object (not an entity)
var theID = oneClass.ID; //scalar value of one object
var theCourseCode = oneClass.courseCode; //value from alias attribute

```

Here the `toArray()` function is provided with two scalar attribute names: `ID` and `courseCode`. Even though `courseCode` is referenced from another entity, it is treated as any other scalar attribute in `Class`.

A visual representation of the array might be:

<b>ID:</b> 4251	<b>courseCode:</b> "ICB-2"
<b>ID:</b> 4252	<b>courseCode:</b> "NS-2"
<b>ID:</b> 4253	<b>courseCode:</b> "ICB-3"
<b>ID:</b> 4254	<b>courseCode:</b> "CC2-2"

But we are not limited to attributes of the entity collection. For example:

```
var myClasses = ds.Class.query('ID < 5');
var classesArray = myClasses.toArray("ID, theCourse.name, theCourse.code");
var oneClass = classesArray[0]; //one class object (not an entity)
var theID = oneClass.ID; //scalar value
var myCourse = oneClass.theCourse.name; //attributes from the same entity
var myCode = oneClass.theCourse.code; //are grouped into one object
```

Here we are requesting a scalar attribute from the collection `myClasses` and two scalar attributes from the related course. When multiple attributes from the same related entity are included in the `toArray()`, they are grouped together. A visual representation might be:

<b>ID:</b> 4252	<b>theCourse:</b> <div style="border: 1px solid blue; padding: 5px;"> <b>code:</b> "NS-2"  <b>name:</b> "Neuropsychological Screening Section 2" </div>
<b>ID:</b> 4253	<b>theCourse:</b> <div style="border: 1px solid blue; padding: 5px;"> <b>code:</b> "ICB-3"  <b>name:</b> "Intervention: Cognitive-Behavioral Therapy Section 3" </div>
<b>ID:</b> 4254	<b>theCourse:</b> <div style="border: 1px solid blue; padding: 5px;"> <b>code:</b> "CC2-2"  <b>name:</b> "Case Conference II Section 2" </div>

If we reference a relation attribute that returns an entity collection, an array is returned. For example:

```
var myClasses = ds.Class.query('ID < 5');
var a = "ID, classAttendees.grade, classAttendees.studentName";
var classesArray = myClasses.toArray(a);
var oneClass = classesArray[0]; //one class object (not an entity)
var theID = oneClass.ID; //scalar attribute
var allAttendees = oneClass.classAttendees; //classAttendees is an array
var oneAttendee = allAttendees[0]; //get one of the array's elements
var oneGrade = oneAttendee.grade; //get the attendees grade
var oneName = oneAttendee.studentName; //get the attendees name
```

A visual representation of `classesArray` might be:

<b>ID:</b> 4251	<b>classAttendees:</b>	<b>grade:</b> "A"	<b>studentName:</b> "Mike Burke"
		<b>grade:</b> "B"	<b>studentName:</b> "Mr. Van Linge"
		<b>grade:</b> "B"	<b>studentName:</b> "George Wu"
		<b>grade:</b> "A"	<b>studentName:</b> "Gordon Klemens"
		<b>grade:</b> "A-"	<b>studentName:</b> "John Wong"
		<b>grade:</b> "C+"	<b>studentName:</b> "John Wong"
		<b>grade:</b> "C"	<b>studentName:</b> "Harry S."

## Wakanda Transactions

A transaction is a way to bundle several data operations (saves, updates, deletes) into an all or nothing package. Wakanda implements transactions as a method of the datastore called `startTransaction()`. Once this method is called, all subsequent data operations are included in the transaction. When all the data operations are complete, a matching `commit()` or `rollback()` method is called. If you roll back a Wakanda transaction, all the data operations are reversed. Code that is executing between the `startTransaction()` and either the `commit()` or `rollback()` is said to be “in transaction.” Code running inside of a transaction sees all the data operations as completed until and unless there is a roll back.

Wakanda supports nested transactions. A nested transaction occurs when you start a transaction while already in a transaction. Like a set of matching parenthesis, the first occurrence of either a `commit()` or `rollback()` applies to the most recently created transaction and all the data operations that were performed while it was in place.

## Entity Locking

Record or row locking is a methodology used in relational databases to avoid inconsistent updates to data. The concept is to either lock a record upon read so that no other process can update it or alternately to check when saving a record to verify that some other process hasn't modified it since it was read. The former is sometimes referred to as pessimistic record locking and it ensures that a modified record can be written at the expense of locking records to other users. The latter is sometimes referred to as optimistic record locking and it trades the guarantee of write privileges to the record for the flexibility of deciding write privileges only if the record needs to be updated. In pessimistic record locking, the record is locked even if there is no need to update it. In optimistic record locking, the validity of a record's modification is decided at update time. Wakanda uses optimistic locking in that when Wakanda attempts to save an entity, it checks to see if the entity has been modified by another mechanism since it was loaded. Each Wakanda entity has an additional attribute not



shown in the Wakanda Datastore Model Designer. This additional attribute is changed at each save of the entity. When Wakanda attempts to save an entity, it checks to make sure that the value in memory matches the one stored in the datastore. If it doesn't, then the save is not completed and an error is thrown. For example, consider the following:

```
var x = ds.Person(1); //Reference to entity
var y = ds.Person(1); //Separate reference to same entity
x.firstName = 'Bill';
x.save();
y.firstName = 'William';
y.save(); //Throws error
```

In this example, we assign to `x` a reference to the person entity with a key of 1. Then, we assign another reference of the same entity to variable `y`. Using `x`, we change the first name of the person and save the entity. When we attempt to do the same thing with `y`, Wakanda checks to make sure the entity on disk is the same as when the reference in `y` was first assigned. Since it isn't the same, Wakanda reports an error and doesn't save the second modification. If we want to handle the error ourselves, we can add a `try...catch` block to the code like this:

```
var x = ds.Person(1); //Reference to entity
var y = ds.Person(1); //Separate reference to same entity
x.firstName = 'Bill';
x.save();
y.firstName = 'William';
try {
    y.save(); //Throws error
}
catch (e){
    //do something here
}
```

One exception to Wakanda's optimistic entity locking mechanism happens if an entity is modified and saved while inside of a transaction that is still open. When this occurs the entity is locked to all other processes until the transaction is completed. Attempting to save an entity in this state will result in an error.

You can turn off the Wakanda entity locking mechanism on a per dataclass basis by using the **Allow Stamp Override** checkbox in the Properties panel. When the mechanism is turned off for a dataclass, Wakanda no longer checks whether the entity has been modified and always overwrites it.

## Testing Code

An easy way to test out server-side code is to create a JavaScript file in your project and run it individually using Wakanda Studio. The scope of the file will be the Application where the file resides and the last line of code that returns a value will be echoed back. This allows you to work through and debug server-side code independent of the front-end browser. This technique was used in several examples above to show the results of functions.

## Wakanda Server Global Namespace and Objects

The above examples have introduced Wakanda Server objects such as entities, entity collections, and datastore classes. Each of these types of objects has predefined methods and attributes that are available in Wakanda server-side programming. Below is a brief description of some of the other Wakanda Server objects.

In code running under Wakanda Server, the default object is the Application. The application object represents one Wakanda project and it has several attributes and methods. All the application object's attributes and methods can be accessed without a qualifier in code running under Wakanda Server. For example:

```
Application.console.log('Processed %d items', numItems);
```

The code above is equivalent to:

```
console.log('Processed %d items', numItems);
```

Some of the more commonly used application attributes are `sessionStorage`, an object that provides a general-purpose storage mechanism for session based information and `ds` an object that provides access to the application's model and a variety of other services. For example, say our application needs to track user-specific information. We define a class scope method named `setSessionInfo` with the following code:

```
setSessionInfo: function(name, value)
{
    sessionStorage.lock(); //will pause if already locked
    sessionStorage[name] = value;
    sessionStorage.unlock();
}
```

This method takes two parameters: the name of what is to be stored and the value. It then attempts to lock the `sessionStorage` object. Each session has a unique `sessionStorage` object but since the browser-side code can run multiple asynchronous threads, we must ensure that two threads don't update the session storage object at the same time. This is easily done using the `lock()` method, which will wait for the `sessionStorage` object to be unlocked before the code proceeds. You should keep the code between lock and unlock as streamlined as possible.

With this one method, browser-side code can identify name/value pairs that can be passed to the server for safekeeping. Each browsing session has a default timeout of 30 minutes, which is refreshed upon each use. The thirty-minute timeout can be adjusted as needed.

To retrieve the stored value, we create the following method:

```
getSessionInfo: function(name)
{
    return sessionStorage[name];
}
```

This method can retrieve any saved session value. This includes complex JavaScript objects.

**Note:** The two methods above are for illustration purposes only. It is unlikely that you would have a session storage mechanism with no control under what conditions an item is stored. More likely, you would attach items to `sessionStorage` using server side constructs driven by controlled input from the client.

One of the most common properties of the `Application` object is the attribute `ds`. This attribute provides access to the application's model and datastore. All datastore classes can be accessed as a property of `ds`. Some examples are:

```
var dClass = ds.Invoice; //assigns the class itself
```

Assigns to `dClass` the datastore class named `Invoice`.

**Note:** Wakanda allows access to multiple datastores at the same time. `ds` is simply the reference to the default datastore of the application.

Alternately, we can refer to classes indirectly by name such as:

```
var className = 'Invoice';
var dClass = ds[className];
```

As you may have noticed in other examples, if the class has been set for global access, it can be accessed without the `ds` prefix. For example,

```
var dClass = Invoice; //Don't need ds for classes set for global access
```

If we need to cycle through all datastore classes, we use the `ds.dataClasses` attribute like this:

```
for (var i = 0; i < ds.dataClasses.length; i++){
    var theClass = ds.dataClasses[i]; //assigns each class
    //Do something with the class here
}
```

Or alternatively like this:

```
for (var e in ds.dataClasses){
    var dClass = ds.dataClasses[e];
    //Do something with the class here
}
```

Also available for datastore classes is the `setAutoSequenceNumber()` method. This method takes one integer parameter, which sets the value for the class's sequence number. This method is particularly useful after assigning values directly to the key attribute of entities. Resetting a datastore class's auto sequence number will ensure that no new entity will repeat a key value.

The `ds` object also provides methods to access transactions (see the *On Remove* event example), project specific folders (`getModelFolder()`, `getDataFolder()`), and to manipulate the server's data cache for the application (`getCacheSize()`, `setCacheSize()`).

Once we have a reference to a datastore class, a variety of attributes and methods become available. Consider the following code:

```

var dClass = ds.Invoice; //Invoice class of default datastore
var endDt = new Date(); //Date for today
var startDt = new Date(); //Date for todaybut changed below
startDt.setMonth(startDt.getMonth()-1); //subtract a month
var eCol = dClass.query('postDate >=:1 AND postDate<=:2', startDt, endDt);
var eCount = dClass.length; //how many entities are in the class
var message = eCol.length + ' invoices of ' + eCount + ' in last month';

```

This code fragment starts by assigning the Invoice class to the `dClass` variable. It then creates two date variables for use in the query. When a query is performed on a datastore class, all entities of the class are considered. The next line counts the number of entities in the datastore class. When the attribute length is applied to a datastore class, it returns the total number of entities in the class. If we want to retrieve a collection of all the entities, we use the datastore class method `all()` like this:

```

var allInvoices = ds.Invoice.all();

```

This brings us to one of the most important objects in Wakanda: entity collections. An entity collection represents a list of entities and provides many different methods for manipulating data. For example, let's break down the following code fragment:

```

var brokers = ds.Person.query('personType = :1', 'broker');
var brokerCnt = brokers.count('rank'); //Count brokers with a rank
var westBrokers = brokers.query('region = "West"'); //West brokers
var rankedWestBrokers = westBrokers.orderBy('rank'); //Sorted west brokers
var topWestBroker = null; //will be used below
if (rankedWestBrokers.length > 0){
    topWestBroker = rankedWestBrokers[0]; //Highest ranked west broker
}
var topWestBrokers = ds.Person.createEntityCollection(); //Empty collection
rankedWestBrokers.forEach(function(theBroker){
    if (theBroker.rank == topWestBroker.rank) //same rank as topWestBroker
        topWestBrokers.add(theBroker); //so include them
});
var topWestCompanies = topWestBrokers.myCompany (); //project to Company

```

This code uses a variety of entity collection abilities. It performs a query on a datastore class and assigns the resulting collection to `brokers`. It then counts the number of entities in the collection that have a value in the `rank` attribute, excluding nulls. Note that `count()` is a method and not an attribute. Using the entity collection now residing in `brokers`, it creates a new collection by further filtering the people to those in the “West” region. A query on an entity collection only considers entities that are already members of the collection. It then sorts the entities in `westBrokers` in order of the attribute `rank` and produces a new, sorted entity collection assigned to `rankedWestBrokers`. Next, it tests to see if there are any entities in `rankedWestBrokers` by using the attribute `length`. If there are, it assigns the first person to `topWestBroker`. To access a specific entity in an entity collection use `[e]` where `e` is an integer from 0 to the total number of entities in the collection minus one. It then uses the Person class to create a valid, but empty entity collection in `topWestBrokers` that will eventually hold all other brokers whose `rank` ties with the rank of the `topWestBroker`. Next, it cycles through all entities in `rankedWestBrokers` using the entity collection iterator `forEach()`, which takes as its first parameter a function to be applied to each entity in the collection. In our example, the function is defined in-line but you may also reference a JavaScript function elsewhere in your code. The function that is specified must accept as its

first parameter a reference to the correct entity type. In our example, this parameter is named `theBroker`, but you may choose any name for this argument. For each entity in the collection, we test to see if the rank equals the `topWestBroker`. If it does, we add the entity to `topWestBrokers` using the entity collection method `add()`. When an entity collection is unsorted, `add()` will not add the same entity twice. If an entity collection is sorted, then `add()` will append the entity reference to the collection, even if the entity is already elsewhere in the collection. The method `add()` can add either a single entity or an entire entity collection. When the `forEach()` is finished, the `topWestBrokers` collection contains the top west broker and all those with the same rank. Of course, we could have performed a query in place of the `forEach()`, but this code illustrates one of the ways to traverse an entity collection. Lastly, we use the `myCompany` relation attribute of the `Person` class to return an entity collection of the companies related to the `topWestBrokers`.

Let's look at some other methods of entity collections. For example,

```
//get an array of all unique people types
var peopleTypes = ds.Person.all().distinctValues('personType');
var results = {};
for (var e in peopleTypes){
    //find all people of each type and average their salary
    results[e] = ds.Person.query('personType = :1', e).average('salary');
}
```

This code fragment starts by using the entity collection method `distinctValues()` to retrieve an array of unique values in the `personType` attribute from all entities in the `Person` class. It then cycles through each element of `peopleTypes` and assigns to a newly defined attribute of `results` the average salary of all people with that specific person type. When this code fragment is done, `results` will have one attribute per person type and the value of each attribute will be the average salary of all people of that type.

Another entity collection method is `remove()`. For example,

```
//find people that have no company
var orphanPeople = ds.Person.query('myCompany = null');
orphanPeople.remove(); //delete them
```

This code will delete all people that have no related companies, assuming there is a relation attribute `myCompany`. If the `Person` class has an *On Remove* event method, the event will execute once per person being deleted. If there is no *On Remove* method and we are deleting all entities in the class, Wakanda will truncate the datastore class's data, which is faster than deleting individual entities.

Individual entities also have methods in addition to the user-defined ones. For example:

```
onSave:function()
{
  if (this.isNew()){
    this.creationDate = new Date(); //today's date
  }
}
```

This *On Save* event tests to see if the entity in question is new (never been saved) and if so it writes the creation date into the entity. A similar function `isModified()` is also available for entities. It returns `true` when an attribute of an entity has been changed, but the entity has yet to be saved. Once the entity is saved, `isModified()` becomes `false`. If we wish to test whether an entity will pass validation, we could use the following:

```
var newPerson = new ds.Person({
  firstName: 'Fred',
  lastName: 'Williams'
});
try {
  newPerson.save(); //try to save the item
}
catch (e) {
  //may be either a validation or a save error
}
```

*Note: In a derived entity there will be an attribute that provides access to the corresponding extended entity. This item is not available in this version.*

## Wakanda Server File Management

Included in Wakanda Server are many utility methods, including file management methods and objects accessible at the global level. For example:

```
var projectFolder = ds.getModelFolder(); //default project folder
//our import folder should be inside of the project's folder
var importFolder = Folder(projectFolder.path + 'ImportData/');
//if it exists
if (importFolder.exists()){
    //cycle through each file in the folder
    importFolder.forEachFile(function(file){
        var filename = file.nameNoExt; //get the plain file name
        var theClass = ds.dataClasses[filename]; //get the class
        if (theClass != undefined) //if there is a class with that name
            theClass.import(file); //import the file into the class
    });
}
```

This code starts by using `getModelFolder`, which is a method of `ds`. This method returns the folder containing the Wakanda project in which the code is executed. It then defines a new variable named `importFolder` to hold a reference to a subfolder named “ImportData.” Next, it checks to see if the `importFolder` exists and if it does, it cycles through all the files it contains. In our example, the code is expecting to find text files in this folder that have the same names as datastore classes in our model. For each file, the code determines the file name and uses it to locate a reference to the corresponding datastore class. On the off chance that a file in the folder is named something other than a datastore class, we check to see if the reference `theClass` is undefined. If not, the code expects a method named `import` to exist for each class and passes it the file reference.

## Dedicated Workers and Shared Workers

Wakanda Server provides multi-threaded capabilities using workers. A worker is created by referencing an individual JavaScript file. When the worker is instantiated, it then becomes an object residing in memory waiting to be called. A worker can have any number of internal methods; however, it must implement an `onmessage` function to respond to outside events if there are any.

The main difference between a dedicated worker and a shared worker is scope. A dedicated worker can only be addressed from the parent thread that created it while a shared worker can be addressed from any thread. Dedicated workers terminate when the parent thread ends while shared workers continue to exist even if their spawning thread ends.

Here is a basic example of how to create a dedicated worker:

```
//the default root folder is the project's folder
//the code inside of dedicatedWorker.js is used to create the worker
var myWorker = new Worker('WorkersFolder/dedicatedWorker.js');
//once we have the worker we attach a function in this thread
//that the worker can call from its thread
var myWorker.onmessage = function(event)
{
    var message = event.data;
```

```

    //the .data attribute will be passed a copy of the object
    //sent from inside of the worker thread
    if (message.type == 'stopped') //if the data has type = 'stopped'
    {
        close(); //we close THIS thread
    }
}
//now we tell the worker to do something by passing an object
myWorker.postMessage({type: 'process'});
wait(); //and then we pause

```

In the above method, a new worker is created by passing a path to a project-specific JavaScript file, named “dedicatedWorker.js”. By passing the file with its relative path, Wakanda uses the project folder as the default, and expects the referenced file to result in a worker.

To create the worker, Wakanda Server parses the JavaScript file and creates an object in memory with all the functions defined in the file as properties. The code above goes on to define a method named `onmessage` that is attached to the new dedicated worker’s reference. The name of this method must be `onmessage` if it is to respond to the `postMessage` calls from inside the worker.

The code above continues by calling `postMessage` to which it passes a simple object with one property named `type` that has the value “process.” Calling `postMessage` is the standard way to communicate with a worker. It takes a single object as a parameter. The object can be as complex as you would like; the values in the object are copied so that they can be used inside of the worker. The code then calls `wait()`, which allows the parent thread to continue to exist, and is ended with a call to `close()`.

A simple example of the `dedicatedWorker.js` file is shown below:

```

function doSomeWork()
{
    // do some work, such as writing files, sending emails, etc.
}

function onmessage(event)
{
    //this function is executed when the parent thread
    //calls postMessage()
    var message = event.data;
    if (message.type == 'process')
    {
        doSomeWork(); //call another function
        postMessage({type: 'stopped'}); //tell the parent we are done
        close(); //close this worker
        break;
    }
}
//Initialize global variables here

```

The first thing to note is that any code executed by parsing the JavaScript file may result in variables and objects that continue for the life of the worker.



Notice that the worker's code also implements a method named `onmessage`. When the parent thread calls the worker with `postMessage`, the `onmessage` function inside of the worker is called. When the worker calls the parent thread with `postMessage`, the `onmessage` function attached to the worker's reference in the parent is called. While both these methods must accept a single object parameter, what they do is completely independent.

In either case the `onmessage` function receives a single object as a parameter. In the above example, the name of the parameter is `event`, but you can choose another name. Regardless of what you call this parameter, it will have a property named `data`. The `data` property will contain a copy of any object passed via the corresponding `postMessage` call. Since the values in `data` are copies, you cannot use any reference types, such as entities, entity collections, or datastore classes. Instead you must pass whatever information you need to identify Wakanda datastore items, such as arrays of key values or datastore class names.

In our worker example, we assign to the `message` variable the values passed from the parent code and then test the `type` property to see if the message is "process." If it is, we run the method `doSomeWork`. In our example, `doSomeWork` is just a stub, but it could do a variety of tasks, such as poll a local directory for files, process entities, send emails, or clean up data. When `doSomeWork` completes, our worker uses `postMessage` to tell the parent thread that it has stopped and then calls `close()` to terminate.

Between the calling parent thread and the worker, here is the logical order of execution in our example:

- 1) At the `new Worker` call, Wakanda Server parses the file which sets up the worker's `onmessage` function. The new worker is now waiting for messages even though it hasn't been told to do anything yet.
- 2) The new worker's reference is returned into `myWorker` in the parent thread.
- 3) A new method named `onmessage` is attached to the worker's reference (`myWorker`) in the parent thread, thus preparing the `myWorker` reference to receive a message from inside the worker.
- 4) The parent thread calls `postMessage` on `myWorker` by passing `{type: 'process'}`. This call causes the `onmessage` function inside the worker to run or queue up a message if the worker is still executing code.
- 5) The parent thread calls `wait()`, which keeps it from terminating, thus turning the parent thread into a worker. When you call this method, the thread stays alive until you call `close()`.
- 6) The worker receives a message through the `onmessage` function as a result of step 4. If it has been told to "process," it calls `doSomeWork`.
- 7) When `doSomeWork` is finished, the worker messages the parent by calling `postMessage` and passing the object `{type: 'stopped'}`.
- 8) The worker then calls `close()`, which ends the worker and frees its resources. The message sent to the parent thread is preserved because the values were copied into the

data property in step 7. Notice that the worker may close before the message it sent to the parent is handled. This is why all worker event data is copied and object references cannot be used.

- 9) The parent thread receives the message from the worker. If the message indicates that the worker has “stopped,” the parent thread closes itself.

The order in which some of these steps execute is ambiguous. For example, steps 2 through 5 may execute before step 1 ends because they are executing in separate threads. But this won't cause a problem. If `postMessage` is called for a worker or parent thread that is currently running code, the resulting event is queued. This includes parent threads that haven't yet executed `wait()`. If you find that you need to control the order of execution between threads, use the methods `mutex()` or `syncEvent()` described below.

If you call `postMessage` on a worker that is currently executing code, the event is stacked and will be executed when the worker finishes its current operation. It is possible that several `postMessage` calls be stacked so that multiple `onmessage` events are waiting to execute. When this happens, the `onmessage` events will execute in the order they were called.

To end a thread, you call `close()` from inside the thread. You can force a dedicated worker to close from the parent thread by calling the worker's `terminate()` method. The `terminate()` method will allow the worker to complete its currently running code and at the next point where it can process a new event, it will close, ignoring any queued events. If you call `close()` on a waiting parent thread, all the dedicated workers spawned from that thread will receive a message to terminate. If you want to allow a worker to complete all its queued events, simply pass it a new event telling it to close.

Shared workers are similar to dedicated workers in that they are created using a reference to a JavaScript file, but unlike dedicated workers, shared workers can continue to exist even after their spawning parent thread ends. When a shared worker is created, you provide a key value in addition to the file reference. When other threads want to interact with an already existing shared worker, they do so by executing the same code as if they are creating it, but instead receive a reference to the existing shared worker.

Here is a basic example of creating a shared worker using a datastore class method:

```
getTaskManagerStatus:function()
{
    //tmRef is a value to uniquely identify this parent thread
    //it is returned by the shared worker when this thread connects
    var tmRef = 0;
    //tmInfo is a simple object to be returned to the client browser
    var tmInfo = {taskCount:0, errorCode:0};
    //create or connect to a shared worker using the reference 'TaskMgr'
    var taskMgr = new sharedWorker('WorkersFolder/TaskMgr.js', 'TaskMgr');
    var thePort = taskMgr.MessagePort;    //attach onmessage to MessagePort
    thePort.onmessage = function(event)
    {
        var message = event.data;
        //decide what the message is telling us to do
        switch (message.type)
        {
```

```

    case 'connected':
        //in this example, this value is returned when this thread
        //connects to the shared worker (see below)
        tmRef = message.ref; //store the reference
        //tell the taskMgr to 'report' and pass this parent
        //threads unique reference
        taskMgr.postMessage({type: 'report', ref: tmRef});
        break;

    case 'update':
        //when the shared worker sends this message it is
        //reporting a count that we assign to tmlInfo.taskCount
        tmlInfo.taskCount = message.count;
        //next we tell the worker that we are disconnecting
        taskMgr.postMessage({type: 'disconnect', ref: tmRef});
        //then we return an object to the browser
        return tmlInfo;
        close(); //and close this thread
        break;

    case 'error':
        //when the shared worker sends this message, it is
        //reporting an error that we assign to tmlInfo.errorCode
        tmlInfo.errorCode = message.errorCode;
        //return tmlInfo to the browser
        return tmlInfo;
        //no need to tell TaskMgr we are disconnecting since
        //it closes when there is an error
        close();
        break;
    }
}
wait(); //waits until a call to close() in this thread
//at this point, this thread is about to end but the shared
//worker continues on
}

```

The purpose of this datastore class method is to respond to a browser-side request for information on the status of the “TaskMgr” shared worker. At the beginning, it defines a variable to hold a number that will be used to inform the shared worker which thread is requesting an update. It then creates a basic object with which to report to the browser-side request and attempts to connect to the shared worker referenced as “TaskMgr.” If there is no shared worker with this reference, this call creates it. If there is already a shared worker that uses the reference “TaskMgr,” this call connects to it.

The code attaches an `onmessage` function, which is expecting a message type of “connected,” “update,” or “error.” Finally, it waits for a response. Since the browser-based call to this method would typically be asynchronous, it is reasonable that this method wait for a response from the shared worker. The corresponding “TaskMgr” worker might be something like:

```

function doSomeWork()
{
    try {
        // do something
        tmCount += 1;
    }
}

```

```

    }
    catch(e){
        tmError = 1;
    }
}

function onconnect (event)
{
    //for a shared worker you use the array ports and attach
    //onmessage to ports[0];
    var thePort = event.ports[0];
    tmKey += 1; //each new connection will receive a unique number
    tmConnections[tmKey] = thePort; //keep a reference to the port
    thePort.onmessage = function(event)
    {
        var message = event.data; //same as dedicated worker
        //the parent always passes its unique reference in message.ref
        //so we know who we are talking to
        var fromPort = tmConnections[message.ref];
        switch (message.type)
        {
            case 'report':
                if (tmError!= 0) //something went wrong so...
                {
                    //tell the parent thread
                    fromPort.postMessage({type: 'error', errorCode: tmError });
                    close(); //and close
                }
                else
                {
                    //nothing wrong, so update the parent with the count
                    fromPort.postMessage({type: 'update', count: tmCount});
                }
                break;

            case 'disconnect':
                //parent is disconnecting, so remove its reference
                tmConnections[message.ref] = null;
                break;
        }
    }
    thePort.postMessage({type: 'connected', ref: tmKey});
}

var tmCount = 0; //count of something going on in doSomeWork
var tmKey = 0; //increments for each new connection to this worker
var tmError = 0; //error code set in doSomeWork
var tmConnections = []; //array of ports (connections) to parent threads
setTimer(doSomeWork(), 1000) //Run every second

```

When the task manager's shared worker is created, the file is parsed, executing the code at the bottom. This code sets up some variables, including the array `tmConnections` that will be used to track all the threads connected to the task manager. The last line of initialization code sets up a timer that will execute the method `doSomeWork` every second (1000 milliseconds). The `setTimer` method is available only inside of a worker or waiting parent

thread. Each time the `doSomeWork` method is executed, the `tmCount` variable is incremented unless there is an error.

Notice the `onconnect` function defined for the task manager. This function is automatically called whenever another thread acquires a reference to the shared worker. This includes the thread that creates the shared worker as well as threads that connect to it when it is already running. When the `onconnect` method is called, the `event` parameter will have a property of type `Array` named `ports`. The value in `ports[0]` is a reference to the parent thread that caused the `onconnect` function to execute. Notice that in our code we increment the variable `tmKey` and then store the reference to the connecting thread in the `tmConnections` array. Also notice that an `onmessage` function is attached to the port. This method will be called when the parent thread posts messages to the shared worker. In our example, we are attaching the same `onmessage` function to each of the connecting parent threads, but we could also attach a variety of `onmessage` functions depending on the situation.

Our code then calls `postMessage` on the newly connected parent thread to inform the parent that the connection was made and returns the value in the `tmKey` variable. Back in the parent thread, this value is stored and used when the parent thread requests that the task manager “report.” This in turn causes the `onmessage` function, attached to the parent’s port inside of the shared worker, to execute. Notice that the shared worker’s `onmessage` function uses the key to determine which parent thread sent a message. If the message type is “report,” the task manager either responds with a count of how many times `doSomeWork` has successfully executed or it responds with an error code, if something went wrong, and then closes. If the task manager successfully “reports” to the parent thread, the parent thread lets the task manager know that it is disconnecting and closes. In the task manager, the “disconnect” causes it to clear out the port reference from the tracking array. This last step is not strictly necessary in our example code. But if we were to augment the task manager so that it responds to all attached parent threads, it might become necessary to track which parent threads are still valid. So, it is good practice to have the parent thread inform the shared worker when it is “disconnecting.” There is no built-in event to inform the shared worker which parent threads are no longer connected.

## Mutex()

With the ability to use multiple threads comes the need to control the order and concurrency of code execution. To do this, Wakanda Server provides a global object called `Mutex`. To create a mutex, you pass it a string that acts as the key for the mutex. Any other thread that uses the same key will interact with the same mutex. A mutex has three methods `lock()`, `unlock()` and `tryToLock()`. The function `tryToLock()` attempts to lock the mutex and returns `true` if it could be locked or `false` if it couldn’t. Unlike the `lock` method (see below), `tryToLock` doesn’t pause execution. For example:

```
var writeMutex = Mutex('writeMutex');
if (writeMutex.tryToLock())
{
    //code here executes if tryToLock was able to lock the mutex
    writeMutex.unlock();
}
//code here executes even if tryToLock was NOT able to lock the mutex
```

A mutex also provides a way to pause execution in one thread until a condition is met in another. In the example below, we want to create a re-usable log file object for use in various server-side functions. To do this, we create a new JavaScript file (named `myLog.js` in our example) with the following code:

```
function Log(file)    //will use this as the constructor
{
    this.logFile = null;
    this.init(file);
    return this;
}

Log.prototype.init = function(file) //add to the prototype chain of Log
{
    if(typeof file == 'string')    //if we pass a string
        file = File(file);    //assume it to be a path
    this.logFile = file; //otherwise assume it to be a file
    if (!file.exists)    //if it doesn't exist
        file.create(); //create the empty file
}

Log.prototype.append = function(message)    //add another function to Log
{
    if(this.logFile != null){
        //if the Log references a valid file, get a Mutex
        //using the files path as its name
        var logMutex = Mutex(this.logFile.path);
        //attempt to lock the Mutex
        //if it is already locked, the next line pauses execution
        //until it becomes unlocked
        logMutex.lock();
        //from here until logMutex.unlock() we know only one thread
        //can be writing to the log file
        var logStream = TextStream(this.logFile, 'write');
        var today = new Date();
        var stamp = today.toString() + ' ' + today.toLocaleTimeString();
        logStream.write(stamp + ': ' + message + '\n');
        logStream.close();
        logMutex.unlock();
    }
}
```

The code above creates a new `Log` object and then adds two functions to its prototype chain. The first function named `init` takes either a Wakanda file object or a path to a file as a string. The code then checks that the file exists and if it doesn't, creates it. The next function, named `append`, takes a string as a parameter. The `append` function double-checks that the `Log` object has been initialized (i.e., references a valid log file) and then creates an object by calling the Wakanda global method `Mutex()`. In our example we use the log file's path as our key with the goal that only one thread can write to a given file at the same time. We then include the above JavaScript file in our project's code by adding an `include` statement to our project's main JavaScript file (i.e., `projectname.waModel.js`) like this:

```
include('myLog.js');
```

This project's main JavaScript file is executed for all calls involving the project so the Log object will be available throughout.

To see an example of the Log object in action we will employ the Wakanda Server method `verify`, which verifies the data and indices of a Wakanda datastore. Consider the following class method:

```
function verifyData()
{
    var projectLog = new Log('./log.txt');
    //our projects main log file
    var noProblems = true; //flag to let us know if there were any problems
    //the object below will provide the call back function(s)
    var problemHandler = {
        addProblem: function(problem){
            projectLog.append('Verify: ' + JSON.stringify(problem));
            noProblems = false;
        }
    }
    projectLog.append('Start Verify');
    ds.verify(problemHandler); //will make periodic call backs
    if (noProblems)
        projectLog.append('Verify found no problems');
    projectLog.append('End Verify');
    return noProblems;
}
```

The `ds.verify` method's single parameter is an object that has functions as attributes. As the verification progresses, Wakanda Server calls back to the appropriate function(s) with information. In the example below, we are only interested in the call back for `addProblem`, which is called as the verification process discovers individual issues with the data or indices. When `addProblem` is called, it is provided a problem object containing information about the issue. In our example, we convert the object to a string value and write it to the project's log file after the heading "Verify: " so that when examining the log file later we can clearly see items concerning this routine. Notice that the log file used is simply "log.txt." In our project, we use this log file for a variety of tasks and therefore require the services of a mutex to make sure updates to the log are done in an atomic way.

There are several other potential call back methods supported by `verify` and we could have included these in `problemHandler`. These include `openProgress(title, maxElements)`, `setProgressTitle(title)`, `progress(currentElementNumber, maxElements)` and `closeProgress()`. For more information see the Wakanda Server API documentation.