Wakanda Data Model

Table of Contents

Wakanda is a new environment for designing and deploying data-driven, web-based applications. At the heart of a Wakanda application is the **Data Model**. As the name implies, a data model describes the data classes and attributes that make up the information stored in your application. But it is more than that. If done correctly it also encapsulates your business rules and logic and provides elegant secure services to your application. This document assumes that you are familiar with the Wakanda Data Model basics as described in the Wakanda Server Side Concepts manual and the Wakanda Security Best Practices document. This is **not** an introductory-level discussion of the data model and it is presumed that you are already familiar with the basics of building a model and working with the various attribute types.

The purpose of this document is to familiarize you with the best practices and concepts involved with defining a data model in Wakanda. One of the best ways to understand and experience the Wakanda data model is by looking at model examples. This document relies heavily on this technique and will walk you through various types of models while explaining each one in detail. The examples discussed in this document are in the accompanying Wakanda solution.

In a Wakanda Data Model you can govern access to attributes and dataclasses using scope, such as **Private**, **Protected**, **Public**, and **Public on Server**. The Wakanda data model provides inherited dataclasses with additional attributes and methods. It provides restricting queries that govern which entities can be accessed from a dataclass and under which conditions. The model also includes events for entity initialization, loading, validating, and saving. In a Wakanda data model, you can define dataclass methods at the entity, collection, and dataclass levels, which become part of the model. All of this and more makes up a Wakanda data model.

When designing a Wakanda Data Model, keep the following concepts in mind:

1) Be aware of all your options including storage, calculated, relation, and alias attributes, as well as restricting queries, inheritance, events, and class methods.
2) Design your model so that it represents as much of your business logic and rules as possible.
3) Build security into your project at the data model level. See the Security Best Practices document.
4) Use restricting queries to assure control over which entities are available through a dataclass.
5) Use inherited dataclasses to create situation specific versions of your data.
6) Complete your dataclasses with all the needed elements so that they provide a complete and encapsulated picture of your data.
7) Look for examples of how you might structure your data.

As you begin the process of building your data model, keep the following "best-practices" in mind when you create attributes, define relations, and attempt to determine the appropriate inherited dataclasses:

1) Relational theories still apply in Wakanda. The big difference between Wakanda and other relational systems is that you build the relational hierarchy of information directly into the model. With the Wakanda data model you can traverse the information in your application easily and intuitively and use named attributes to query related information.
2) You do not need relation attributes to get to related data, but the biggest benefits do come from using relation attributes.
3) Determine the type of attribute to create for each discreet data point. Use storage attributes for the primary data that makes up an entity. If a value is always determined by the values in other attributes of the same entity, consider using a calculated attribute so that redundant information isn't stored. Use relation attributes to link dataclasses and alias attributes if you are regularly going to need values from parent entities.
4) Remember that when an entity is delivered to the browser, the values of all its calculated attributes are determined. If the code for a given calculated attribute is particularly complex and thus would be expensive to calculate each time the entity is returned to the browser, it would likely be best to use an entity method so that the code only runs when called.
5) Use inherited data classes to create situation specific versions of an existing class. You can use a derived class to limit which attributes are available or add new ones. You can use a derived class to limit which entities can be accessed. You can override methods and events so that your application provides different behavior for different classes.
6) Restricting queries return entity collections. In the case of derived classes, the collection returned can be a collection of the derived class or of the base class. When Wakanda returns the entity collection, it will convert it to a collection of the class in which the restricting query is run. When defining restricting queries to return entity collections for derived classes, it is more flexible to use criteria available to the base class. Otherwise, it will be difficult to promote and demote entities if you so desire.
7) If you define a restricting query for a base class, you will always return an entity collection from that class. In this case, it is best if you provide a way to access all the entities in the base dataclass as part of the restricting query. For example, you may want to return all the entities if the user is in the Admin group.
8) It might seem useful to create a base data class that all other data classes inherit from. For example, say you want to have dateCreated and createdBy attributes in all of your data classes, so you decide to create a base class with just those attributes and then inherit from this class for all other classes. In object-oriented programming, this might be useful but in Wakanda you would be putting all entities into the same dataclass, which might not provide the best performance.
9) Currently, Wakanda does not allow new storage attributes nor primary N -> 1 relation attributes to be added to derived data classes. So, if there is new information that must be stored, it will need to be added to the base class. Calculated attributes, alias attributes, dependent relation attributes, and 1 -> N primary relation attributes can be added to a derived class. If the base class doesn't need them, add them to the derived class.

When designing a Wakanda Data Model the order that you create items is constrained. For example, you can't complete a calculated attribute if it depends on storage attributes that you haven't yet created. When designing a Wakanda Data Model, you might want to follow these steps:

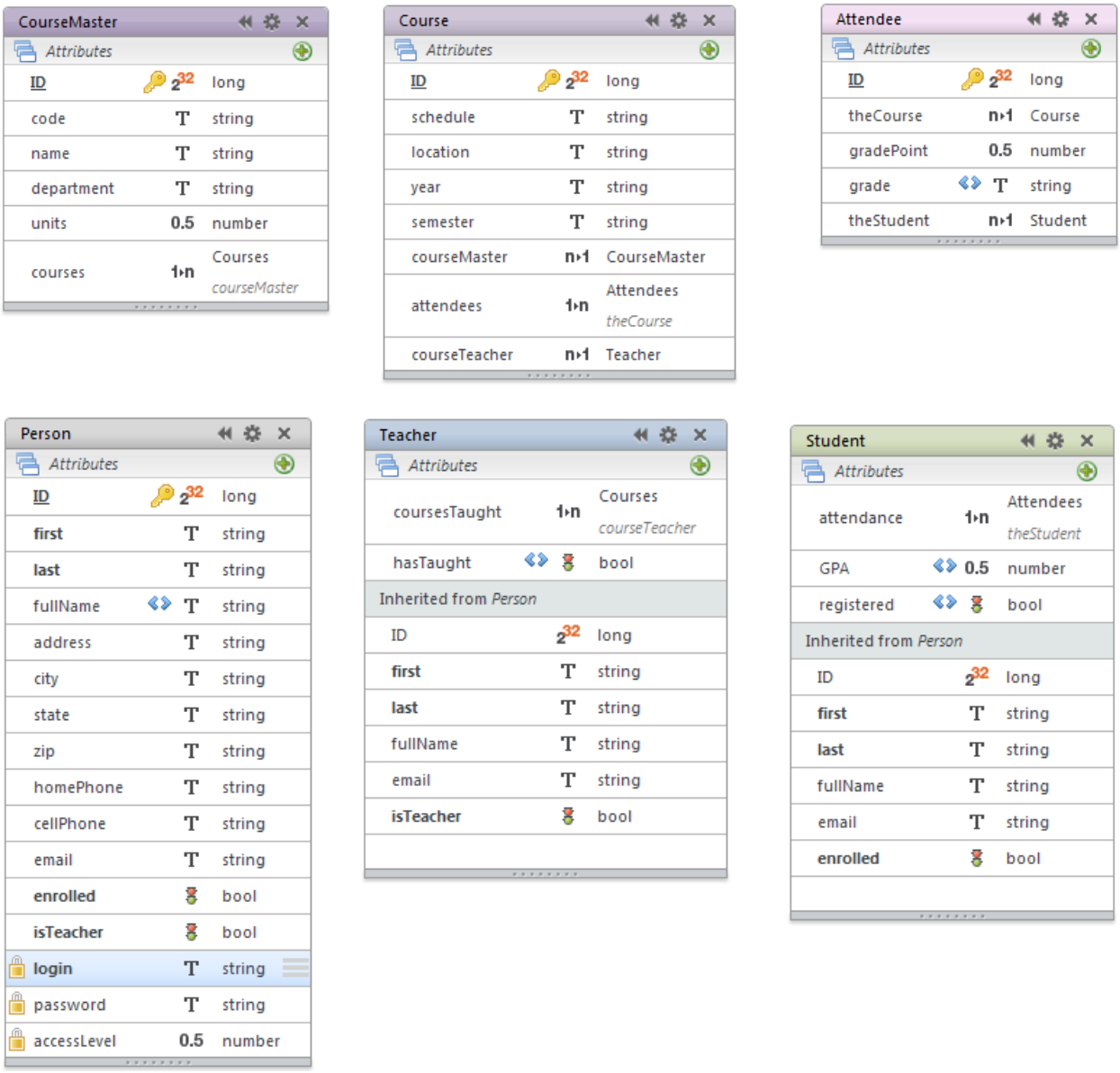
a) Create all the dataclasses you are confident the model will need and give each a well-named singular (for the datastore class) and plural name (for the collection).
b) Change the key type if necessary.
c) Choose one of the main data classes and add storage attributes and any simple calculated attributes (those based upon the storage attributes).
d) Do the same for each dataclass.
e) Start adding primary relation attributes on dataclasses that are at the bottom of the information hierarchy, that is, data classes that are dependent on others.
f) Do the same for each data class.
g) Add dependent relation attributes and alias attributes when needed.
h) Only after the above should you start adding events and methods.

When you create a new dataclass, Wakanda automatically creates an attribute named ID that is of type Long and set to auto sequence. As you create entities, this attribute will be assigned a unique number. Later, when entities are related, Wakanda uses the value in this attribute to link entities. You may change the type of this attribute if you prefer a different key type. For example, you can instead set the type to UUID where each value in this attribute is different from every other UUID. This can be useful if you want a unique identifier for every entity in your data model, not just within the dataclass. You may also remove the key from dataclasses that don't need it although this will prevent you from using relation attributes to reference the keyless dataclass. In many cases, leaving the default type of the ID attribute is sufficient.

The rest of this document explains Wakanda Data Models through many examples. All these models can be found in one Wakanda Solution. Note that we are focusing on the models and will provide little or no interface in the example solution.

Education

This example illustrates a data model for a school. It highlights a variety of Wakanda capabilities including inheritance, restricting queries, private attributes, 'removed' attributes in derived classes, and several calculated attributes. When you design a model, you do so with your application in mind. The interface for this application will need to provide several capabilities. Students will need to register for classes, see their class schedule, and consult their grades and GPA. Teachers will need to see the courses they are teaching, the students that are registered for those classes and be able to enter each attendee's grade. The registrar and staff will need to enter and enroll new students, add teachers, and create courses for the new school year. In addition, everyone should be able to update their contact information. The model is shown below.



CourseMaster is the dataclass that stores all the courses that are currently or historically provided by the school. Each entity in Course is related to a CourseMaster and stores information about the year, the semester, the schedule (e.g., Mon, Wed, and Fri) and location (e.g., Room 202). In this data model, each Course is taught by one teacher, indicated by *courseTeacher*. The Attendee data class stores information about each course a student is taking or has taken. In our model, the main purpose of the Attendee class is to relate a student to a course and to store the resulting grade. The Attendee *gradePoint* attribute stores the numeric representation of the grade (4, 3, etc.) whereas the calculated attribute *grade* provides the student's grade point as a letter grade (A-, B+, etc.). We start the discussion of this model with the *grade* attribute's On Get script as shown below.

```
onGet:function()
{
    var gp = {}; //could define this ahead of time but left here for clarity
    gp['0'] = 'F';
    gp['1'] = 'D';
    gp['1.33'] = 'D+';
    gp['1.67'] = 'C-';
    gp['2'] = 'C';
    gp['2.33'] = 'C+';
    gp['2.67'] = 'B-';
    gp['3'] = 'B';
    gp['3.33'] = 'B+';
    gp['3.67'] = 'A-';
    gp['4'] = 'A';
    if (gp[this.gradePoint] != null)
        return gp[this.gradePoint];
    else
        return '';
}
```

The simple code above defines a JavaScript object with properties that are named as '0', '1', '1.33' and so on. It then uses the object to convert the value for the current entity's *gradePoint* attribute into an alphabetic grade. The keyword **this** refers to the current entity in which this code is executed. The object *gp* could have been defined earlier but is shown here for illustration.

The same attribute's On Set script can be implemented as:

```
onSet:function(value)
{
    var gp = {}; //could define this ahead of time but left here for clarity
    gp['F'] = 0;
    gp['D'] = 1;
    gp['D+'] = 1.33;
    gp['C-'] = 1.67;
    gp['C'] = 2;
    gp['C+'] = 2.33;
    gp['B-'] = 2.67;
    gp['B'] = 3;
    gp['B+'] = 3.33;
```

```
    gp['A-'] = 3.67;
    gp['A'] = 4;
    value = value.toUpperCase();
    this.gradePoint = gp[value];
}
```

This is just the reverse of the On Get. With these two functions defined, we can display and assign to the *grade* attribute treating it like any other attribute. We complete the implementation of the calculated attribute by implementing the On Query and the On Sort scripts as shown below.

```
onQuery:function(compOperator, valueToCompare)
{
    valueToCompare = valueToCompare.toUpperCase();
    var gp = {};
    gp['F'] = 0;
    gp['D'] = 1;
    gp['D+'] = 1.33;
    gp['C-'] = 1.67;
    gp['C'] = 2;
    gp['C+'] = 2.33;
    gp['B-'] = 2.67;
    gp['B'] = 3;
    gp['B+'] = 3.33;
    gp['A-'] = 3.67;
    gp['A'] = 4;
    if (gp[valueToCompare] != null)
        return 'gradePoint ' + compOperator + gp[valueToCompare];
    else
        return 'gradePoint = -1'; //force a bad result so no entities are returned
}

onSort:function(ascending)
{
    if (ascending)
        return 'gradePoint';
    else
        return 'gradePoint desc';
}
```

Like all On Query and On Sort calculated attribute scripts, these two functions return strings that are substituted into queries and sorts when the calculated attribute is used.

The Person dataclass is used to store information about each individual associated with the school. This includes students and teachers as well as other staff members, such as support staff and registrars. In our model, we have extended Person to create two new dataclasses: Student and Teacher. Each of these derived data classes uses a restricting query based on a Boolean attribute in Person.

For Student, the restricting query is based on the *enrolled* attribute that defines whether a Person is enrolled in the school as a Student. The restricting query is quite simple.

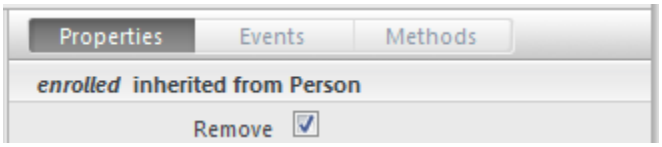
```
onRestrictingQuery:function()
    {
        return ds.Person.query('enrolled = true');
    }
```

The Teacher restricting query is very similar.

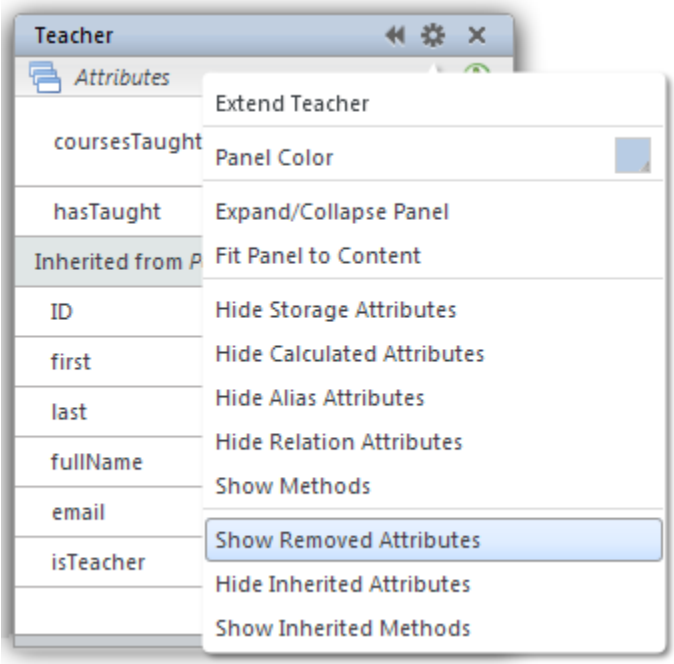
```
onRestrictingQuery:function()
    {
        return ds.Person.query('isTeacher = true');
    }
```

Both *enrolled* and *isTeacher* are indexed boolean storage attributes that can be set by a staff member indicating that a person is either an enrolled student, a teacher or both. Notice that the *enrolled* attribute is available in Student. If this attribute is set to **false** the person will cease to be a student and will no longer show up in the entities of the Student dataclass. Whether an attribute, like *enrolled*, is available in the Student data class is decided by your application's business rules. If we did choose to remove the *enrolled* attribute from Student, it could always be set in the corresponding Person entity to enroll a student in the school.

Notice that the *enrolled* attribute is not available in the Teacher dataclass and the *isTeacher* attribute is not available in the Student dataclass. Furthermore, the address and phone related attributes are not available in either derived class. These attributes have been removed from the derived classes using the Remove checkbox in the Properties tab.



With these attributes removed, Student and Teacher entities are greatly simplified with no extraneous information delivered to the browser during use. When a base attribute is removed, it is no longer part of the derived class but its display in the editor can be toggled using the **Show/Hide Removed Attributes** item.



The Student dataclass has some additional calculated attributes named *GPA* and *registered*. These attributes have been added to the derived Student data class and are only available in Student entities. Both *GPA* and *registered* would be unsuitable for the Teacher data class so it is appropriate that they only be added to Student instead of Person. The *registered* attribute returns true if the student is registered for any courses and contains the following On Get function.

```
onGet:function()
{
    return (this.attendance.length > 0);
}
```

With this attribute defined we can locate unregistered students with a simple query like this:

```
ds.Student.query('registered = false');
```

Student entities found by this query are enrolled but have not been assigned to any courses.

The Student *GPA* attribute is meant to return a student's overall grade point average. The function for this is quite simple.

```
onGet:function()
{
    return this.attendance.average('gradePoint');
}
```

Similarly, the Teacher dataclass has a calculated attribute named *hasTaught* that indicates whether the teacher has taught any courses.

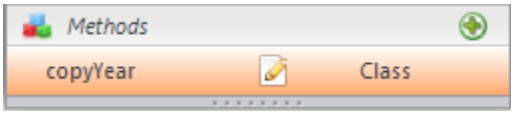
```
onGet:function()
{
    return (this.coursesTaught.length > 0);
}
```

In our application, we use the entities in the Person dataclass to authenticate users of our application. Notice the icon indicating that the *login*, *password*, and *accesslevel* attributes have a scope of *Private* and thus unavailable to the browser as well as the derived classes Student and Teacher. Our login listener method is shown below (see Wakanda Security Best Practices)

```
function myLogin(userName, password)
{
    var theUser = ds.Person({login:userName});
    if (theUser == null) // if no user was found
      return false; // let Wakanda try to find a user in the directory
    else
    {
        // see of the stored hash value is correct
        if (theUser.password == directory.computeHA1(userName, password))
        {
            var myLoginInfo = {myUserID: theUser.ID};
            var theGroups = [];
            if (theUser.enrolled)
                theGroups.push('Student');
            if (theUser.isTeacher)
                theGroups.push('Teacher');
            if (theUser.accessLevel == 1)
                theGroups.push('Registrar');
            if (theUser.accessLevel == 2)
                theGroups.push('Staff');
            return {ID: theUser.ID,
                    name: theUser.login,
                    fullName: theUser.fullName,
                    belongsTo: theGroups,
                    storage : {loginInfo: myLoginInfo}
                    }
        }
        else
            return { error: 1024, errorMessage:"invalid login" };
    }
}
```

With this login function, all we need to do is attach the groups ‘Student’, ‘Teacher’, ‘Registrar’ and ‘Staff’ to the appropriate control points in the model. See the Wakanda Security Best Practices document for more information on permissions.

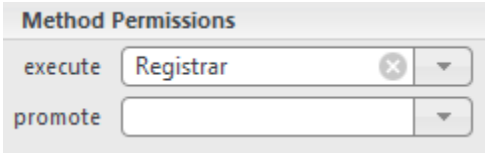
One of the common tasks in a school is creating the course schedule for the new school year. This is commonly done by the registrar or staff and usually uses the current year's courses as a basis. In our model, we have a method to copy an entire year's courses into another year. The registrar can then adjust the courses as needed. This method can be attached to the Course dataclass as a class method.



The method's code is fairly simple.

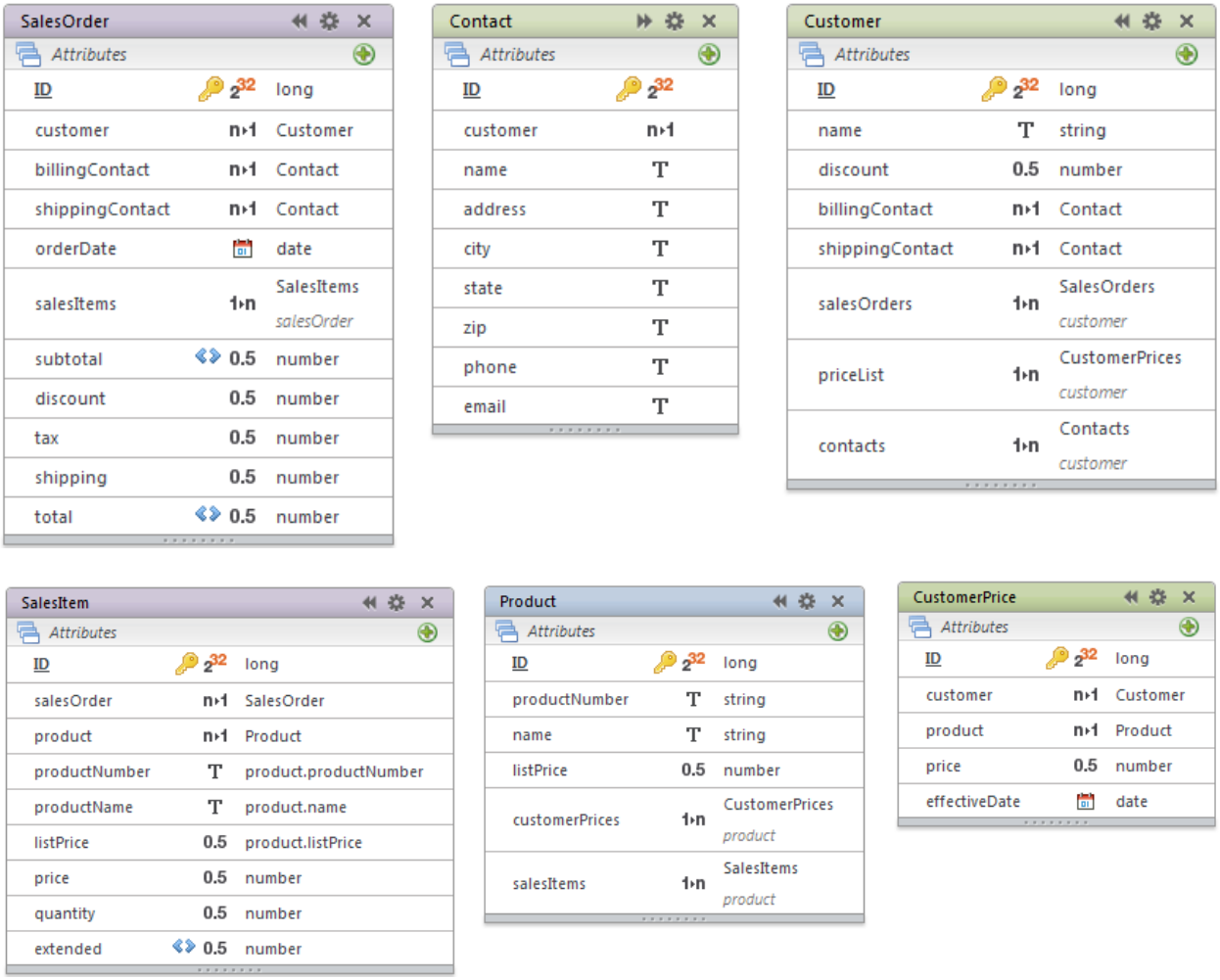
```
copyYear:function(fromYear, toYear)
{
    var fromCourses = ds.Course.query('year = :1', fromYear);
    fromCourses.forEach(function(loopCourse){
        new ds.Course({
            theCourse:  loopCourse.courseMaster,
            schedule: loopCourse.schedule,
            location: loopCourse.location,
            semester: loopCourse.semester,
            year: toYear,
            courseTeacher: loopCourse.courseTeacher
        }).save();
    });
}
```

This simple method copies an entire year's courses into another year. Of course, the registrar would need to update schedules, locations and teachers as needed but this method provides a starting point for the year. This type of method would commonly be restricted to users with specific credentials. In our case, only the registrar should run this so we assign the *registrar* group to the method's *execute* control point.



Sales

This example provides a data model that would be used in fulfilling customer orders where each product sold may have a different price per customer.



Each SalesOrder is related to one Customer. Each Customer has a default *billingContact* and *shippingContact* to be used when a customer is assigned to an order. In this project, our business rules dictate that an order's billing contact and shipping contact may be overridden for each sales order, but only with other contacts related to the customer. In order to accommodate this rule, SalesOrder also has primary relation attributes linking it to the Contact data class. Later, we will see how the model ensures that any contact assigned to a sales order is from the related customer.

One of the first things to note about this model is that the reciprocal 1 -> N relation attributes of the Contact data class have been deleted. When you define a relation attribute, Wakanda Studio automatically creates its reciprocal relation attribute in the related data class. In our application, these relation attributes (Contact to Customer and Contact to SalesOrder) are not needed.

If you will not be using the auto-generated 1 -> N relation attributes in your application, it is safe to delete them from the model. This not only clears up clutter in your data model, but it also streamlines entities in the data class.

Let's begin looking at code by examining what happens when a customer is assigned to a sales order. One of the events available at the attribute level is the On Set event. Do not confuse this with a similarly named calculated attribute script, which has a slightly different purpose. In this example, the On Set event for SalesOrder.customer is shown below.

```
onSet:function(attributeName)
{
    this.billingContact = this.customer.billingContact;
    this.shippingContact = this.customer.shippingContact;
}
```

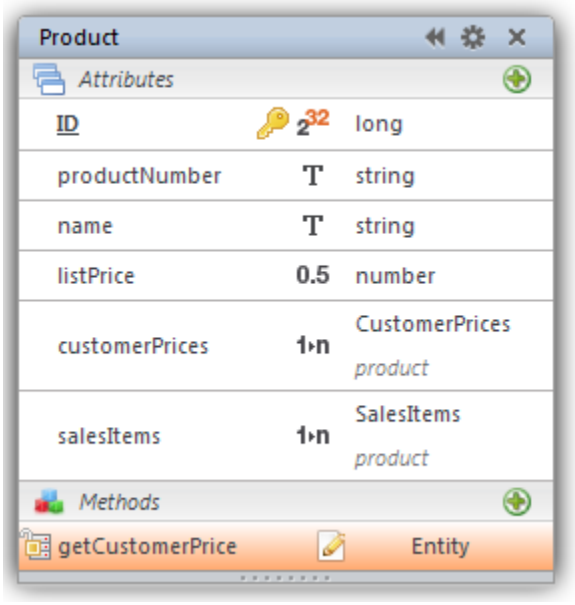
This code will run when a value is assigned to the *customer* attribute of a SalesOrder entity. Under Wakanda Server, this happens immediately when a value is assigned. Under the browser, this happens when the entity is delivered back to Wakanda Server and the value has been updated. Entities are delivered back to Wakanda Server during entity validation, entity saving and when **serverRefresh** is called.

Basic validation code for saving a SalesOrder is placed in the On Validate event as shown below.

```
onValidate:function()
{
    var error = {error: 0, errorMessage: ''};
    var billingContact = this.billingContact;
    var shippingContact = this.shippingContact;
    var theCustomer = this.customer;
    if (theCustomer == null)
        error = {error: 101, errorMessage: 'No assigned customer'};
    else if (this.orderDate == null)
        error = {error: 104, errorMessage: 'No order date'};
    else if (billingContact == null)
        error = {error: 102, errorMessage: 'No billing contact'};
    else if (shippingContact == null)
        error = {error: 103, errorMessage: 'No shipping contact'};
    else if (theCustomer.contacts.find('ID = :1', billingContact.ID) == null)
        error = {error: 105, errorMessage: 'Billing contact invalid for customer'};
    else if (theCustomer.contacts.find('ID = :1', shippingContact.ID) == null)
        error = {error: 106, errorMessage: 'Shipping contact invalid for customer'};
    return error;
}
```

This code returns an error if no customer, order date, billing contact or shipping contact is assigned, or if the assigned billing or shipping contact is not one of the contacts from the customer.

For each SalesOrder, there is any number of related SalesItems. Each SalesItem refers to a single Product and also tracks the price for each and the quantity sold. Relating Product to Customer is the CustomerPrice data class. This dataclass houses information on what price to charge a Customer for a given Product and on which date the price becomes effective (*effectiveDate*). If there is an effective CustomerPrice for a Product, then it is used to determine the price. If there is no effective CustomerPrice, then the customer is charged the product's *listPrice* modified by a customer *discount*. Notice that *price* is a storage attribute in SalesItem. Once *price* is determined it is saved into the sales item so that future changes to the customer and customer prices do not affect existing orders. This is different from a calculated attribute that would be recalculated upon each use. Let's look at how we might determine the price of a Product for a SalesItem. To do this, we create a data class entity method for Product named *getCustomerPrice*.

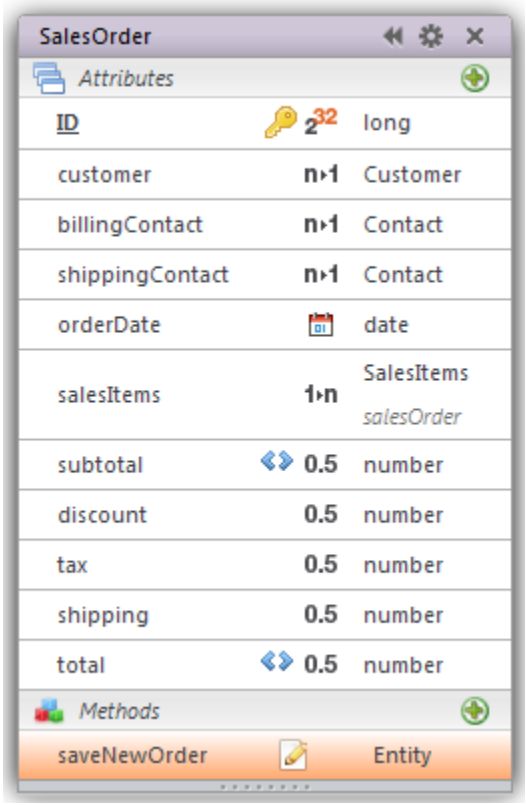


This function's purpose is to return a specific price for a product on a given sales order. This function will only be used on the server so its scope is set to "Public on Server".

```
getCustomerPrice:function(salesOrder)
{
    thePrices = this.customerPrices; // get the customer prices for this product
    var theCustomer = salesOrder.customer; // get the customer of the sales order
    var orderDate = salesOrder.orderDate; // get order date of the sales order
    var qString = 'customer = :1 AND effectiveDate <= :2'; // build a query string
    qString += ' order by effectiveDate desc'; // notice the built in sort
    var customerPrice = thePrices.find(qString, theCustomer, oderDate);
    if (customerPrice != null)
        return customerPrice.price;
    else
        return this.listPrice - (this.listPrice * theCustomer.discount);
}
```

This function requires a sales order entity as an argument. The sales order must have a valid customer and order date to be used with this function that we verify in the calling method. The query string locates entities that have a matching customer and an *effectiveDate* before or equal to the *orderDate*. The query string also includes an **order by** clause that sorts the results in descending order by *effectiveDate*. Since the query string is used with the collection's *find* function, only the most recently active CustomerPrice entity is returned. Queries and finds on a collection only consider entities that are in the collection. If no customer price is found, *find* returns null. The rest is just a matter of deciding which price to use.

Next, let's examine how this function is used in conjunction with a sales order. As you design your model you will often decide between allowing direct saves of entities from the browser (*entity*.save()) vs. controlling entity saves by using data class methods. In our example, the application's business rules dictate that we create a sales order and its items in one logical step. And that once saved, a sales order and its items cannot be modified. We handle this by disabling direct updates of SalesOrder and SalesItem by assigning the appropriate groups to the *create*, *update* and *remove* control points of the SalesOrder data class (see Wakanda Security Best Practices guide). Then we add an entity level data class method to SalesOrder, named *saveNewOrder*, that has the proper privileges to save sales orders and items.



The *saveNewOrder* method works on a single SalesOrder, but expects an array of objects as an argument. The object in each element holds a product ID and a quantity. No other information passed to this call is considered.

```
saveNewOrder:function(salesItems)
{
    var salesItems = salesItems || null; // in case this method was called with no argument
    try
    {
        this.validate(); // run the On Validate event
        if (!this.isNew())
            throw 107; //not a new sales order
        else if (salesItems == null)
            throw 108; //no sales items
        else if (!Array.isArray(salesItems))
            throw 109; //sales items not an array
        else if (salesItems.length == 0)
            throw 110; //array has no elements
        else
        {
            ds.startTransaction();
            try
            {
                for (index in salesItems)
                {
                    var item = salesItems[index];
                    if ((item.productID == null) || (item.quantity == null))
                        throw 111; //incorrect sales item properties
                    else if (!item.quantity > 0)
                        throw 112; //no quantity specified
                    else
                    {
                        var theProduct = ds.Product(item.productID);
                        if (theProduct == null)
                            throw 113; //no product found for sales item
                        else
                            new ds.SalesItem({
                                salesOrder: this,
                                quantity: item.quantity,
                                product: theProduct,
                                price: theProduct.getCustomerPrice(this)
                            }).save();
                        }
                    }
                        this.save(); //save the sales order
                        ds.commit(); //commit the transaction
                }
            catch (e) //if there is any errors
            {
                ds.rollBack(); //rollback the transaction
                throw e; //re-throw the error
            }
        }
    }
    catch (e)
    {
```

```
        throw e; //re-throw the error
    }
}
```

Since this method is an entity level data class method, the keyword **this** represents the sales order for which this method is executed. One of the first things this method does is to call **this.validate()**. This causes the On Validate event of the entity to execute and returns an error object if any basic validation isn't met. It is advantageous to isolate basic validation code in the On Validate event so that it can be used independently to validate an entity without having to save it. Next, this method checks to make sure this is a new sales order since we don't allow sales orders to be modified. It then runs some basic validation tests on the salesItems argument.

After passing validation, this method starts a transaction and begins going through the elements of the array, producing a new sales item for each one. Notice the use of *getCustomerPrice* and the keyword **this** to determine the price for each sales item. The method finishes by saving the sales order itself and committing the transaction. If an error occurs, the commit code will not be reached and instead the transaction is rolled back and the error is re-thrown.

Much of the rest of the model is fairly simple. The attribute *extended* in SalesItem is a simple calculated attribute that multiplies the price times the quantity.

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

Back in SalesOrder, the calculated attribute *subtotal* returns the sum of the *extended* attribute for all sales items on the order. Since this value is calculated, it is always up to date when examined.

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

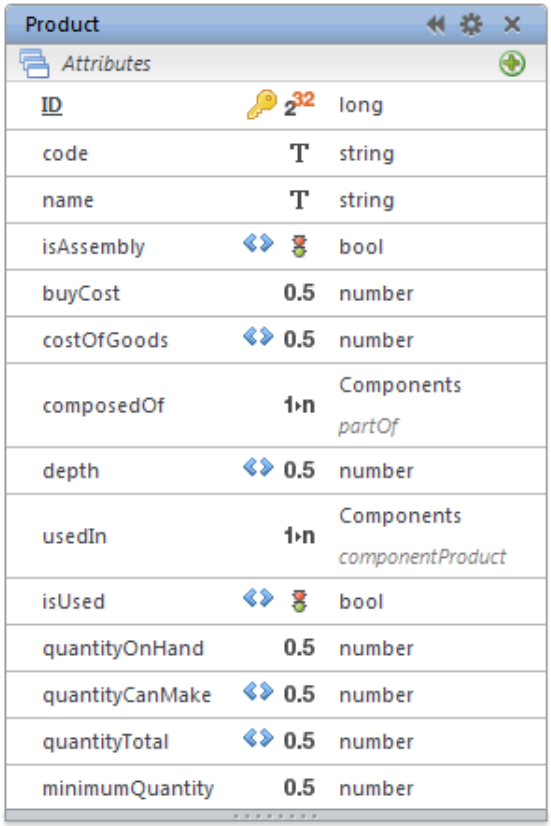
Lastly, the calculated attribute *total* returns the sum of the *subtotal*, *shipping*, and *tax* attributes.

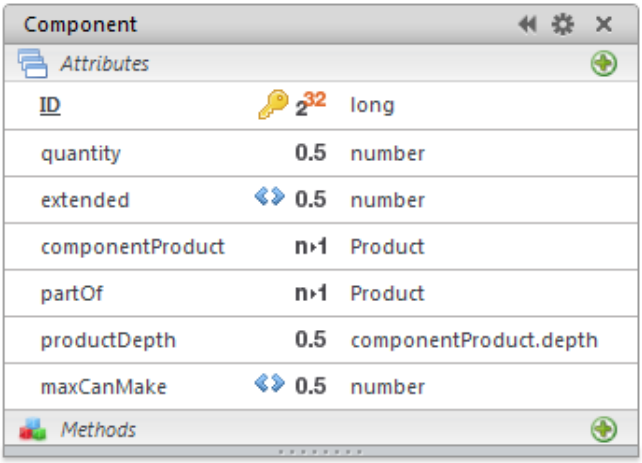
```
onGet:function()
{
    return this.subtotal + this.shipping + this.tax;
}
```

Bill of Materials

This example provides a data model that might be used in manufacturing. It shows how elegant a data model can be for a very messy task. In manufacturing there is a need to allow products to be composed of other products which are composed of other products, etc. This can be quite difficult to handle in a traditional relational database.

The example's model has two dataclasses named Product and Component. Each Product is an item that we buy or make. For each Product that we make, we track the product's components. Each Product that we make has a collection of Components, each of which describes a quantity of a component Product. This results in a functionally recursive relation between Product and Component and allows a way to describe the entire hierarchy of a Product. First, examine the model.





Product has a relation attribute named *composedOf*. This relation attribute returns a collection of components that represent a product's parts. The reciprocal for *composedOf* is the Component relation attribute named *partOf*. The dataclass Component has another relation attribute named *componentProduct* that relates back to Product and during use would refer to a different Product entity. The reciprocal of *componentProduct* is the Product attribute *usedIn*. If you follow the logic through you will see that you can traverse the hierarchy of products in both directions, down the hierarchy from a product to its components or up the hierarchy from component products to the products they comprise.

Typically, in an application of this type there are certain functions that need to be supported. One such function is the ability to determine the cost of goods for any given Product. The cost of goods for a product is either its buy cost (if we buy it) or the cost of its constituent products (if we make it). Let's take a look at how this might be handled in a Wakanda Data Model. First, note that the Product dataclass has a calculated attribute named *isAssembly* that is used to determine if a product is something we make. It returns true if a product has components or false if it doesn't. The On Get function is shown below.

```
onGet:function()
    {
        return (this.composedOf.length > 0);
    }
```

We could have chosen to place this value in a storage attribute so that it could be indexed and optimized but in this model we chose to make it a calculated attribute.

Next, Product has a storage attribute named *buyCost*. This attribute stores the cost of each Product that we purchase but has no meaning to product's we make. The Product dataclass also has a calculated attribute named *costOfGoods* which returns the buy cost for products we purchase or the assembly cost for those we make. The On Get for *costOfGoods* is shown below.

```
onGet:function()
    {
        var theCost = 0;
        if (this.isAssembly)
            theCost = Math.round(this.composedOf.sum('extended') * 100) / 100;
        else
            theCost = this.buyCost;
        return theCost;
    }
```

This function uses the *isAssembly* attribute to decide whether to return the *buyCost* or to sum the *extended* attribute of the *composedOf* components. The *extended* attribute in Component is itself calculated as shown below.

```
onGet:function()
    {
        var theCost = 0;
        var theQuantity = 0;
        if (this.componentProduct!= null)
            theCost = this.componentProduct.costOfGoods;
        if (this.quantity != null)
            theQuantity = this.quantity;
        return (Math.round(theCost * theQuantity * 100)/100);
    }
```

This function's goal is to multiply the part's quantity by the cost of the component product and return the extended value. Notice that this function uses componentProduct.costOfGoods and is therefore recursively determining the cost. The value that this calculated attribute returns is determined **interactively** at the time it is used. The result is up to the minute costing information on any Product we examine.

Next, let's look at the depth attribute in Product. This calculated attribute is used to determine the deepest part of a given Product's hierarchical list of components. For Product's that we buy, it returns 1. For Product's that we make, it returns 1 more than the deepest component. For example, products that are composed only of other products that we buy, it returns 2. The function for this attribute is below.

```
onGet:function()
    {
        if (!this.isAssembly)
            return 1;
        else
            return (this.composedOf.max('productDepth') + 1);
    }
```

This function relies on an alias attribute in Component named *productDepth* that refers to the component products depth. Like *costOfGoods*, this value is calculated interactively when used.

Further down the Product dataclass is the *quantityOnHand* attribute. This storage attribute is intended to contain the quantity of each Product that we have in inventory. Following *quantityOnHand* is the *quantityCanMake* attribute. This calculated attribute is intended to calculate the quantity that can be made of any given Product. Here is the On Get function for *quantityCanMake*.

```
onGet:function()
    {
        var theQuantity = 0;
        if (this.isAssembly)
            theQuantity = this.composedOf.min('maxCanMake');
        return theQuantity;
    }
```

This function uses the *maxCanMake* attribute of Component which is calculated using the following function.

```
onGet:function()
    {
        return Math.floor(this.componentProduct.quantityTotal / this.quantity);
    }
```

The *maxCanMake* attribute of Component uses the *quantityTotal* attribute of Product which is calculated using this function.

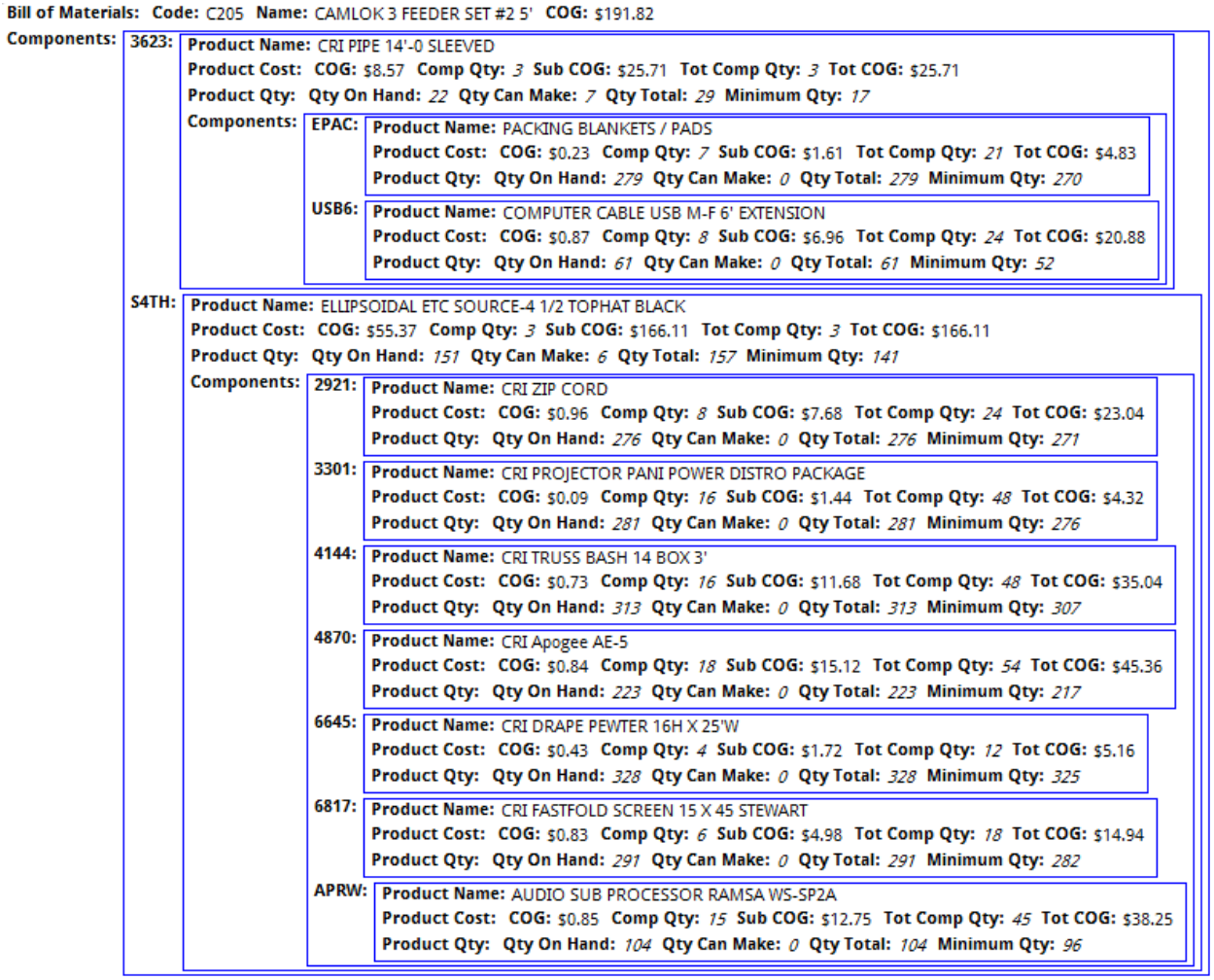
```
onGet:function()
    {
        return this.quantityOnHand + this.quantityCanMake;
    }
```

Let's follow the chain of logic through calculating the maximum quantity that we can make for one Product entity. When we access Product.quantityCanMake it runs the first function above. The function returns 0 if the product is not an assembly (we can't make something we buy). Otherwise, it asks Wakanda to determine the minimum value of the *maxCanMake* attribute for all components of the original product entity. In order to determine the minimum value for the components, Wakanda runs the On Get function (second function) associated with *Component.maxCanMake* for each of the components. The *Component.maxCanMake* function asks Wakanda to determine the component product's *quantityTotal* value and divides that value by the quantity needed as indicated by *Component.quantity* (shown as this.quantity in the code). Accessing the component product's *quantityTotal* executes its On Get function (third function) which adds the *quantityOnHand* to the *quantityCanMake*. Of course, referencing *quantityCanMake* executes its On Get function and around we go until we get to the bottom of the component hierarchy. The result is that we can determine the quantity that we can make of any given Product by simply accessing the calculated attribute.

As you design your model you will often consider whether to use an entity method in place of a calculated attribute. The advantage of an entity method is that it only executes when called. On Wakanda Server this is also true of calculated attributes. However, when an entity is delivered to the browser, Wakanda Server sends the value of all attributes that have a scope of Public, thus running the On Get function for calculated attributes. If it is uncommon that you will need the value on the browser or if the calculated attribute's code may take too much time, you should consider using an entity method instead.

Of course, there is a tradeoff. Entity methods are not attributes and may require extra handling on the browser. They cannot be queried or sorted like attributes. In addition, they cannot be bound to many of the Wakanda widgets as attributes can.

Let's take a look at some entity methods that might be used in the Bill of Materials model. One common need in a system that handles a hierarchy of products is a type of report called an exploded bill of materials. This is typically a hierarchical list for a given product that breaks down its sub-assemblies and their sub-assemblies all the way to the bottom of the product hierarchy. A visual representation of an exploded BOM is shown below.



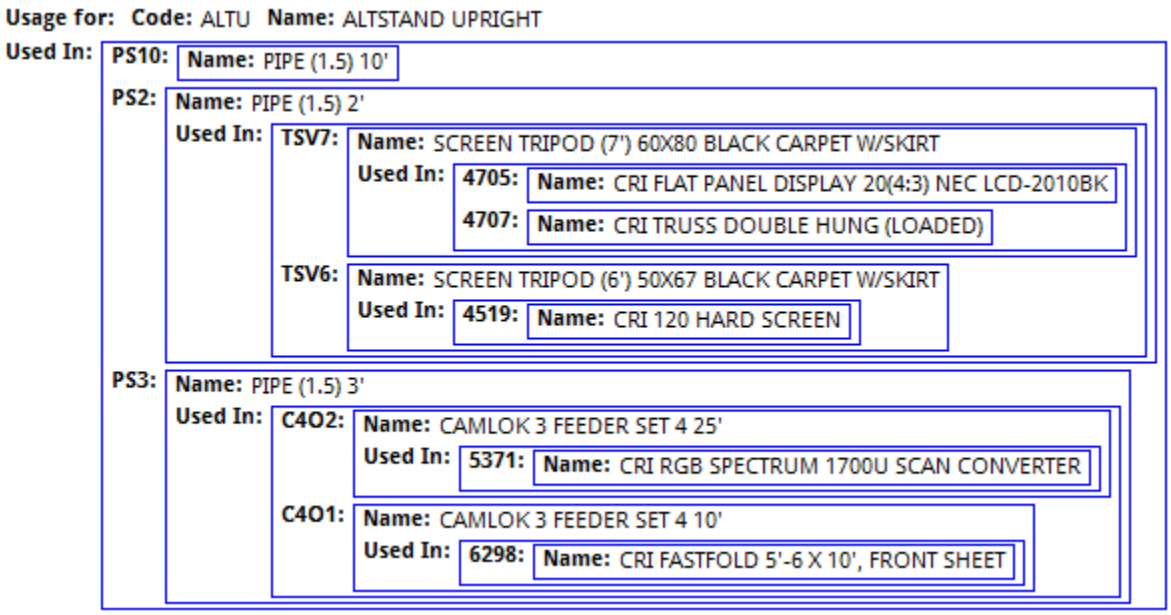
To create this in the Bill of Materials module we introduce a dataclass method named *explodedBOM* that is defined at the entity level. When executed, this function returns a javascript object of the product's hierarchy of components. This method can then be called from either the server or the browser to provide the bill of materials information for a Product entity. The code for the *explodedBOM* function is shown below.

```
explodedBOM:function(levelQuantity)
{
    var levelQuantity = levelQuantity || null; // makes levelQuantity safe to access
    var partsObject = {};  // object to store description at this level
    var topObject = null; // object for top of BOM
    if (levelQuantity == null) // if the function was called with no parameters
    {
        levelQuantity = 1; // then we are at the top
        topObject = {}; // build an object for the top
        topObject['Bill of Materials'] = [{
            Code: this.code,
            Name: this.name,
            COG: '$' + this.costOfGoods.toFixed(2)}]; // single element array for display
    }
```

```
    var theParts = this.composedOf; // get its components
    theParts.forEach(function(loopPart) { // and loop through them
        var theCode = loopPart.componentProduct.code; // get the components code
        partsObject[theCode] = {}; // use the code as a property name
        partsObject[theCode]['Product Name'] = loopPart.componentProduct.name;
        var info = {}; // build an object for component cost/qty info
        info['COG'] = '$' + loopPart.componentProduct.costOfGoods.toFixed(2);
        info['Comp Qty'] = loopPart.quantity;
        info['Sub COG'] = '$' + loopPart.extended.toFixed(2);
        info['Tot Comp Qty'] = loopPart.quantity * levelQuantity;
        var totCOG = Math.round(loopPart.extended * levelQuantity * 100);
        info['Tot COG'] = '$' + (totCOG/100).toFixed(2);
        partsObject[theCode]['Product Cost'] = [info];
        var inventoryInfo = {}; // build an object for component product info
        inventoryInfo['Qty On Hand'] = loopPart.componentProduct.quantityOnHand;
        inventoryInfo['Qty Can Make'] = loopPart.componentProduct.quantityCanMake;
        inventoryInfo['Qty Total'] = loopPart.componentProduct.quantityTotal;
        inventoryInfo['Minimum Qty'] = loopPart.componentProduct.minimumQuantity;
        partsObject[theCode]['Product Qty'] = [inventoryInfo];
        if (loopPart.componentProduct.isAssembly) // if this component is an assembly
        { // then recursively call the explodedBOM method for this component's product
          var subQty = levelQuantity * loopPart.quantity;
          partsObject[theCode].Components = loopPart.componentProduct.explodedBOM(subQty);
        }
    });
    if (topObject != null) // indicates we are at the top of the BOM
    {
      if (this.isAssembly) // if there were any components
        topObject.Components = partsObject; // put all the components into one property
      return topObject; // return the top object
    }
    else // indicates we are in a recursive call
        return partsObject; // so just return the parts
}
```

This method builds a complex bill of materials (BOM) object for a product. It starts by checking whether *levelQuantity* was passed to this function. If it wasn't, then the method assumes that this is the first call in a recursive calling chain and thus the top product in the BOM. If this is the top of the BOM, it defines *topObject* to store information about the product for which it is building the BOM. Next it loops through all of the product's components building up objects with information about the component and its corresponding product. It then tests to see if the corresponding component product is itself an assembly, and if so, it calls *explodedBOM* recursively passing the aggregate quantity needed at that particular level in the hierarchy. It finishes by checking to see if whether the top object was defined. If it was then it appends the list of any components to the top object and returns it. If there is no top object, then this is a recursive call of the *explodedBOM* method and it returns only the parts object. This method starts at a given Product and traverses the hierarchy of component products to the bottom of the bill of materials.

A similar method can be defined for exploded usage which shows where a product is used in other products. The result is hierarchical list of other products that are dependent on the first. A visual representation might look like this.



The *explodedUsage* method is shown below.

```
explodedUsage:function(levelQuantity)
{
    var levelQuantity = levelQuantity || null;
    var topObject = null; // object for top of usage
    if (levelQuantity == null) // if the function was called with no parameters
    {
        levelQuantity = 1; // then we are at the top
        topObject = {}; // build an object for the top
        topObject['Usage for'] = [{Code: this.code, Name: this.name}];
    }
    var partsObject = {};
    var theParts = this.usedIn; // get the components where this product is used
    theParts.forEach(function(loopPart) { // and loop through them
      var theCode = loopPart.partOf.code; // get the parents code
      partsObject[theCode] = {}; // use the code as a property name
      partsObject[theCode].Name = loopPart.partOf.name;
      if (loopPart.partOf.isUsed) // if the parent is itself used in something
      { //then recursively call the explodedUsage method for the parent Product
        var subQty = levelQuantity * loopPart.quantity;
        partsObject[theCode]['Used In'] = loopPart.partOf.explodedUsage(subQty);
      }
    });
```

```
    if (topObject != null) // indicates we are at the top of the usage information
    {
        if (this.isUsed) // if this product was used in anything
            topObject['Used In'] = partsObject; // attach the usage info
      return topObject;
    }
    else // indicates we are in a recursive call
        return partsObject; // return the usage info
}
```

This method is similar to the previous one. It starts by checking whether levelQuantity was passed to this function. If it wasn't, then the method assumes that this is the first call in a recursive calling chain and thus the top product for exploded usage. If this is the top, it defines *topObject* to store information about the product for which it is building the usage information. Next it loops through all of the components that use the product. Note the use of **this.usedIn** which provides a collection of Component entities where the Product is used. The rest of this method functions similarly to the *explodedBOM* method.

Another task that might be needed is the ability to make a complex product from its constituent products, and if needed, their constituent products, down to the bottom of the product hierarchy. The goal is to use up already existing inventory (quantityOnHand) of any given product before causing assemblies further down the product hierarchy to be built. We can already determine the maximum quantity of a product that can be made by examining the *quantityCanMake* attribute but what if we actually want to build some products and recursively decrement inventory down through the BOM? For this, we introduce two entity level dataclass methods named *buildProduct*, which has a scope of Public so it can be called on the browser, and *buildAssembly*, which is a support method to *buildProduct* and has a scope of Public on Server that protects it from being called by the browser. First, let's look at *buildProduct*.

```
buildProduct:function(quantityToBuild)
{
    var quantityToBuild = quantityToBuild || null;
    if (quantityToBuild == null)
        quantityToBuild = 1; // if they didn't specify, we will assume one
    else if (quantityToBuild < 1)
        quantityToBuild = 1; // if they gave us 0 or negative

    if (this.isAssembly) // if it isn't an assembly, we can't build it
    {
        if (this.quantityCanMake >= quantityToBuild) // do we have enough
        {
            ds.startTransaction();
            try
            {
                var theParts = this.composedOf;
                theParts.forEach(function(loopPart) {
                    loopPart.componentProduct.buildAssembly(quantityToBuild * loopPart.quantity);
                });
                this.quantityOnHand += quantityToBuild;
                this.save();
```

```
                ds.commit();
            }
            catch (e)
            {
                ds.rollback();
            }
        }
        else
            throw 100;
    }
    else
    {
        throw 90;
    }
}
```

This method accepts a quantity to be built. If no quantity is supplied, or if the value supplied is 0 or less, then we assume the user wants to build only one. The method tests to see if the current Product is an assembly. If it doesn't have component products, then this is something we buy, not make. Next it tests to see if we even have enough inventory to meet the build demand. If all these tests are passed, it starts a transaction before it begins modifying the inventory of products. If anything goes wrong during the execution of this method or the subsequent *buildAssembly* method, then we will roll back the transaction so that everything is left in its original state. Notice that the bulk of the code is placed inside of a **try..catch** block and the result of any error is the execution of ds.rollback. The code loops through all of the components and calls the *buildAssembly* method for each component product, passing in the quantity needed. The *buildAssembly* method is shown below.

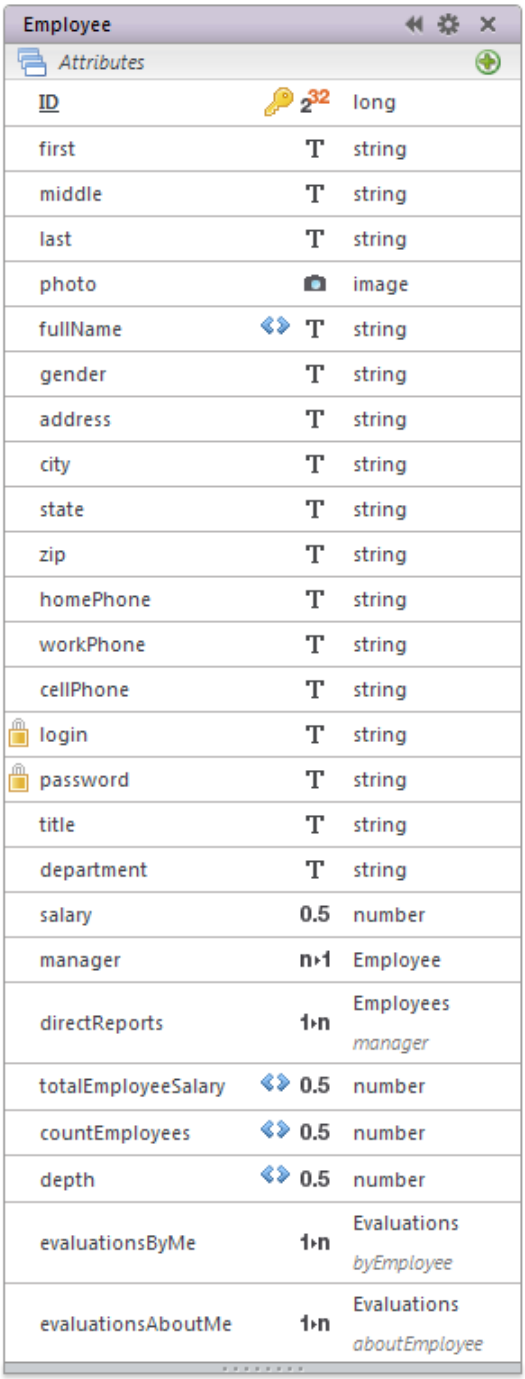
```
buildAssembly:function(quantityNeeded)
{
    if (this.isAssembly)
    {
        if (this.quantityTotal >= quantityNeeded)
        {
            var makeQuantity = 0;
            if (quantityNeeded > this.quantityOnHand)
                makeQuantity = quantityNeeded - this.quantityOnHand;
            if (makeQuantity > 0)
            {
                var theParts = this.composedOf;
                theParts.forEach(function(loopPart) {
                    loopPart.componentProduct.buildAssembly(makeQuantity * loopPart.quantity);
                });
                this.quantityOnHand += makeQuantity;
            }
            this.quantityOnHand -= quantityNeeded;
            this.save();
        }
```

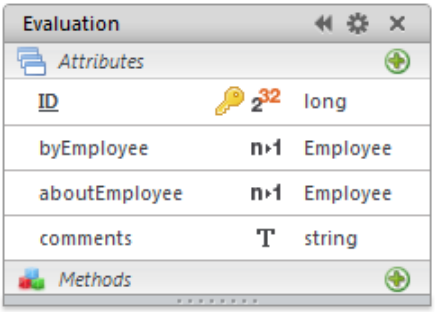
```
        else
        {
            throw 110;
        }
    }
    else
    {
        this.quantityOnHand -= quantityNeeded;
        this.save();
    }
}
```

This method uses up existing inventory (*quantityOnHand*) before issuing further *buildAssembly* calls. If a product is not an assembly this function simply decrements the quantity on hand. We need not worry that the quantity on hand is insufficient to meet demand because our original *quantityCanMake* was tested at the top level of the calling chain. After several tests, this method checks to see if any subassemblies need to be built. If so, it issues recursive calls to *buildAssembly* with the quantity that needs to be made at that level. It then increments and decrements quantity on hand as needed. In an actual application this routine would typically issue requests to manufacturing to build the assemblies instead of just decrementing inventory.

Organization

This project's data model deals with employees and evaluations. Since many applications in the business world track employee information it is a good place to start. In this example each employee is able to evaluate every other employee for which they have a relation. This includes their manager, their peers and their direct reports. Let's start with the following data model. The evaluation is nothing more than tracking comments by someone, about someone.





Most of the Employee attributes are self-explanatory. The *fullName* calculated attribute's code is the concatenation of *first*, *middle* and *last* as below. In our code we assume that *first* and *last* are not null.

```
onGet:function()
    {
        var middle = '';
        if (this.middle != null)
            middle = (this.middle.length > 0 ? ' ' + this.middle : '');
        return this.first + middle + ' ' + this.last;
    }
```

Scanning further down the list of attributes in Employee, nothing else is of particular interest until we get to the *manager* relation attribute and its reciprocal *directReports*. These attributes represent a recursive relation between employee entities. In use, the *manager* attribute will produce an Employee entity while the *directReports* attribute will produce an entity collection of Employees.

Next we have some calculated attributes that build on *directReports*. The *totalEmployeeSalary* sums the total salary of all direct or indirect reports, providing a departmental salary total.

```
onGet:function()
    {
        return this.directReports.sum('totalEmployeeSalary') + this.salary;
    }
```

The *countEmployees* attribute also uses *directReports* and is intended to return the total number of people who directly or indirectly report to an individual. Its code is similar to the *totalEmployeeSalary* attribute.

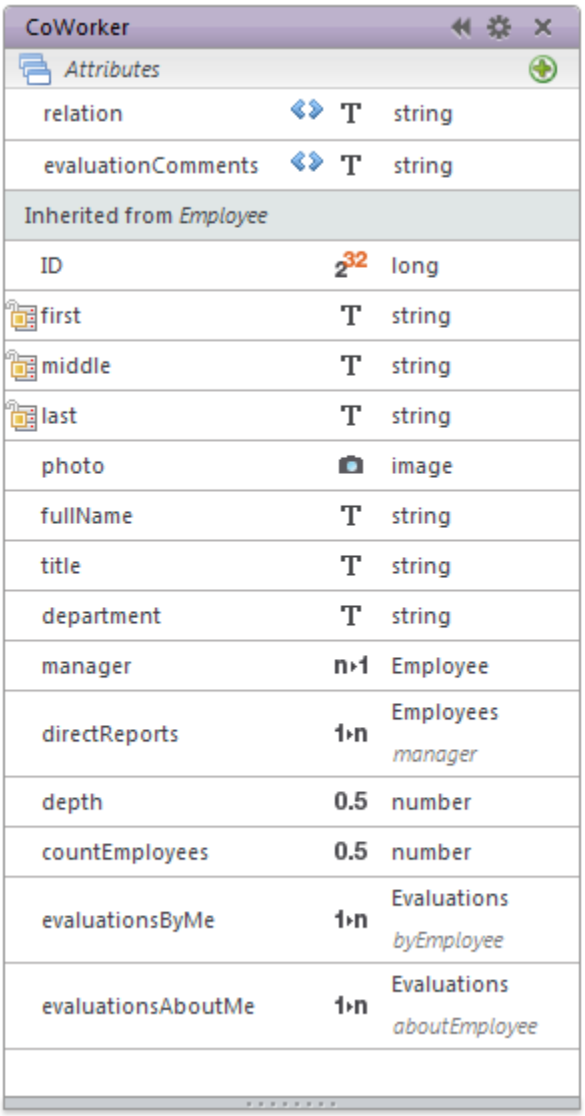
```
onGet:function()
    {
        return this.directReports.sum('countEmployees') + 1;
    }
```

Lastly, the code for the *depth* attribute is shown below.

```
onGet:function()
    {
        return this.directReports.max('depth') + 1;
    }
```

This attribute returns a number corresponding to level of the individual in the organizational hierarchy. For individual contributors, those without any direct reports, this attribute returns 1. For those with direct reports composed of individual contributors, it returns 2, and so on. Notice that this calculated attribute's code is basically recursive and determines the depth interactively.

The last two attributes are *evaluationsByMe* and *evaluationsAboutMe*. Both of these attributes provide an entity collection from the Evaluation data class. In this example, evaluations are very simple. They represent comments made by an employee about another employee. Our business rules dictate that employees can only make comments about other related employees. That is, an employee can only comment on their manager, their peers or their reports. How can we insure this at the model level? One way to do this is to extend the Employee data class to create a new derived data class. In our example, we name this data class CoWorker. The CoWorker data class will have a restricting query that controls which entities can be accessed through the data class as well as two new calculated attributes named *relation* and *evaulationComments*.



First, let's examine the code in the On Restricting Query event.

```
onRestrictingQuery:function()
{
    var myInfo = sessionStorage.loginInfo;
    var resultCollection = ds.Employee.createEntityCollection();
```

```
    if (myInfo != null)
    {
        var queryString = '';
        if (myInfo.myManagerID != null)
        {
            queryString = 'ID = ' + myInfo.myManagerID + ' OR '; // my boss
            queryString += '(manager.ID = ' + myInfo.myManagerID; // my peers
            queryString += ' AND ID != ' + myInfo.myEmployeeID + ') OR '; // not me
        }
        queryString += ' manager.ID = ' + myInfo.myEmployeeID; // and my reports
        resultCollection = ds.Employee.query(queryString);
    }
    return resultCollection;
}
```

The only purpose of the derived CoWorker dataclass is to provide a collection of specialized entities that represents the current user's co-workers. The restricting query assumes that the current user's ID and manager ID were placed in *sessionStorage* during login (see Wakanda Security Best Practices for an example). If the information has NOT been added to *sessionStorage*, then the restricting query returns an empty entity collection. If there is valid login information in *sessionStorage*, we use *myEmployeeID* and *myManagerID* to construct a query string to locate the current user's manager, peers and reports, while excluding the current user. The result is that any time this data class is accessed it will be limited to showing the co-workers of the current user.

You might also have noticed that we have chosen to remove various storage attributes such as *address*, *city*, *state*, *zip* and *salary* from the derived class. These attributes are not needed when accessing co-workers. Furthermore, we only want to provide access to *fullName* in the CoWorker data class, not its individual components. But if we remove *first*, *middle* and *last* from the data class, it will thus invalidate the On Get code for *fullName* which depends on these attributes. Instead, we set *first*, *middle* and *last* to have a scope of Public on Server which allows these attributes to be used in the On Get of *fullName* but they are not delivered to the browser.

Next, let's examine the *relation* calculated attribute and its On Get method:

```
onGet:function()
{
    var myInfo = sessionStorage.loginInfo;
    var result = '';
    if (this.ID == myInfo.myManagerID)
        result = 'Manager'
    else if (this.manager != null)
    {
        if (this.manager.ID == myInfo.myEmployeeID)
            result = 'Direct Report';
        else if (this.manager.ID == myInfo.myManagerID)
            result = 'Peer';
    }
    return result;
}
```

This script also uses the same login information from *sessionStorage* as the code above. But unlike the restricting query code, it doesn't need to check whether there is valid login info because this code will only run for entities found by the original restricting query, which only returns entities if there is valid login information.

The last calculated attribute in the CoWorker data class is *evaluationComments*. The On Get code for this attribute is shown below.

```
onGet:function()
{
    var result = '';
    var myInfo = sessionStorage.loginInfo;
    var queryString = 'byEmployee.ID = :1 AND aboutEmployee.ID = :2';
    var theEval = ds.Evaluation.find(queryString, myInfo.myEmployeeID, this.ID);
    if (theEval != null)
        result = theEval.comments;
    return result;
}
```

Once again, there is no need to check for valid login information. This calculated attribute looks up an Evaluation entity that is **about** the co-worker (notice the use of **this.ID**) and **by** the current user. The resulting comments are returned as the value for the attribute. This means that this value adjusts based upon the current user. The same CoWorker entity for two different users will return a different value in this attribute. We take this even further by using the same calculated attribute's On Set method, as shown below.

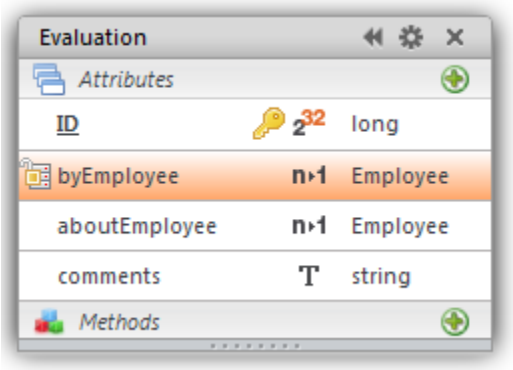
```
onSet:function(value)
{
    var myInfo = sessionStorage.loginInfo;
    var queryString = 'byEmployee.ID = :1 AND aboutEmployee.ID = :2';
    var theEval = ds.Evaluation.find(queryString, myInfo.myEmployeeID, this.ID);
    if (theEval == null) // if it doesn't already exist
        theEval = new ds.Evaluation({aboutEmployee: this, byEmployee: myInfo.myEmployeeID});
    theEval.comments = value;
    theEval.save();
}
```

This code is essentially the reverse of the On Get. It locates the correct Evaluation. If it doesn't find one, it creates a new Evaluation. Notice that you can assign either an entity (**aboutEmployee: this**) or a key value (**byEmployee: myInfo.myEmployeeID**) to a relation attribute. Next it passes the value that was assigned to the *evaluationComments* attribute on to the *comments* attribute of the Evaluation. Lastly it saves the Evaluation.

Using the *evaluationComments* attribute in CoWorker, the current user need never deal with evaluations directly when making comments about coworkers.

But what about viewing comments made **about** the current user **by** their co-workers. If our business rules dictate that all comments are to be anonymous, then the originator of the comment

will need to be unavailable to the current user. This is accomplished by setting the scope of the *byEmployee* attribute of the Evaluation data class to *Public on Server*.



With this scope, no browser can access this value. We then attach a restricting query to the Evaluation data class as below.

```
onRestrictingQuery:function()
{
    var myInfo = sessionStorage.loginInfo;
    var resultCollection = ds.Evaluation.createEntityCollection();
    if (myInfo != null)
    {
        var session = currentSession();
        if (session.belongsTo('Admin'))
            resultCollection = ds.Evaluation.all();
        else if (myInfo.myEmployeeID != null)
        {
            var queryString = 'byEmployee.ID = :1 OR aboutEmployee.ID = :1';
            var myID = myInfo.myEmployeeID;
            resultCollection = ds.Evaluation.query(queryString, myID);
        }
    }
    return resultCollection;
}
```

This restricting query constrains evaluations to those by or about the current user, unless the current user is in the Admin group. This restricting query combined with the scope of the *byEmployee* attribute insures that the current user cannot access other people's evaluations nor see who wrote the evaluations about them.

Notice that in all the code aboveabove, we are not counting on controlling what a user can do on the client side of the application. We are building all logic and business rules into the model itself.

A Wakanda data model can also include dataclass methods. Dataclass methods can be attached at the dataclass, collection or entity level. In our example we add a method named *getOrgChart* attached at the entity level of the Employee dataclass. Its purpose is to return a javascript object that describes direct and indirect reports of a given employee to a particular depth, along with a

brief bit of information about each. In essence, this function returns an org chart. We do this with the following method.

```
getOrgChart:function(depth)
{
    var depth = depth || null; //in case depth wasn't passed
    if (depth == null)
        depth = 100; //effectively infinite if not specified
    else
        depth -= 1;     //decrement the depth
    var org = {}; //will hold the org chart for this level
    var info = {Title: this.title, Salary: '$' + this.salary.toFixed()};
    info['Total Employees'] = this.countEmployees; //add Total Employees attribute
    info['Total Salary'] = '$' + this.totalEmployeeSalary; //add Total Salary
    org[this.fullName] = [info];  //put entire info object as new attribute
    if ((this.directReports.length > 0) && (depth > 0))
    {
        var attName = this.directReports.length + ' Reports';
        org[attName] = {};
    var theEmployees = this.directReports;
    var i = 0;
    theEmployees.forEach(function(loopEmployee) {
        i++;
        org[attName][i] = loopEmployee.getOrgChart(depth);
    });
    }
    return org;
}
```

Before examining this routine, look at a sample of the result on the next page. The sample is a visual representation of the org chart object returned for the CEO of an organization, to a depth of 3. This gives us an example of our goal. The *getOrgChart* method is actually quite simple. It first checks whether the depth parameter was included, assumes a depth of 100 if it wasn't, else it decrements the depth value. It then creates an object to store the org chart. It adds the current employee to the org chart object and then if the employee has direct reports AND we still haven't satisfied the depth, we cycle through the direct reports and recursively call the same routine. Calling this routine is as simple as locating an entity and executing the getOrgChart method.

Bill Costa: **Title:** CEO **Salary:** $400000 **Total Employees:** *1607* **Total Salary:** $122860995

6 Reports:

1: **Albert Barnette:** **Title:** Vice President Engineering **Salary:** $218592 **Total Employees:** *188* **Total Salary:** $14447987

3 Reports:

1: **Dave Mei:** **Title:** Engineering Director **Salary:** $134246 **Total Employees:** *33* **Total Salary:** $2525205

2: **C. Preston Meyer:** **Title:** Engineering Director **Salary:** $142014 **Total Employees:** *90* **Total Salary:** $6828691

3: **Iris Wong:** **Title:** Engineering Director **Salary:** $145019 **Total Employees:** *64* **Total Salary:** $4875499

2: **Hazel Lintner:** **Title:** Vice President Sales **Salary:** $218062 **Total Employees:** *377* **Total Salary:** $28683997

5 Reports:

1: **Jing Biggar:** **Title:** Sales Director **Salary:** $135657 **Total Employees:** *54* **Total Salary:** $4110706

2: **Dennis Pine:** **Title:** Sales Director **Salary:** $136785 **Total Employees:** *58* **Total Salary:** $4385093

3: **Steven Kato:** **Title:** Sales Director **Salary:** $144652 **Total Employees:** *82* **Total Salary:** $6195572

4: **Peggy Peth:** **Title:** Sales Director **Salary:** $142929 **Total Employees:** *77* **Total Salary:** $5877828

5: **Diana Weil:** **Title:** Sales Director **Salary:** $146512 **Total Employees:** *105* **Total Salary:** $7896736

3: **Dave Miller:** **Title:** Vice President Marketing **Salary:** $201150 **Total Employees:** *312* **Total Salary:** $23644880

4 Reports:

1: **Pam Arens:** **Title:** Marketing Director **Salary:** $135307 **Total Employees:** *94* **Total Salary:** $7057486

2: **Lee Breen:** **Title:** Marketing Director **Salary:** $138081 **Total Employees:** *62* **Total Salary:** $4644783

3: **William Melling:** **Title:** Marketing Director **Salary:** $138067 **Total Employees:** *79* **Total Salary:** $5971834

4: **Duffy Freiberg:** **Title:** Marketing Director **Salary:** $138116 **Total Employees:** *76* **Total Salary:** $5769627

4: **Charles W. Wohl:** **Title:** Vice President Accounting **Salary:** $200858 **Total Employees:** *405* **Total Salary:** $30660756

6 Reports:

1: **Wayne Hill:** **Title:** Accounting Director **Salary:** $137125 **Total Employees:** *79* **Total Salary:** $5916810

2: **Bob Maderious:** **Title:** Accounting Director **Salary:** $133739 **Total Employees:** *38* **Total Salary:** $2917841

3: **Joseph Coghlan:** **Title:** Accounting Director **Salary:** $141114 **Total Employees:** *101* **Total Salary:** $7566268

4: **Steve Orlando:** **Title:** Accounting Director **Salary:** $134183 **Total Employees:** *52* **Total Salary:** $3914370

5: **Jeff Willmore:** **Title:** Accounting Director **Salary:** $143038 **Total Employees:** *69* **Total Salary:** $5254832

6: **John Gehres:** **Title:** Accounting Director **Salary:** $141114 **Total Employees:** *65* **Total Salary:** $4889777

5: **Larry Bowen:** **Title:** Vice President Support **Salary:** $202452 **Total Employees:** *265* **Total Salary:** $20069274

3 Reports:

1: **Harry Nochimson:** **Title:** Support Director **Salary:** $145328 **Total Employees:** *89* **Total Salary:** $6691706

2: **Mr. Ken Tyrrell:** **Title:** Support Director **Salary:** $145122 **Total Employees:** *43* **Total Salary:** $3262419

3: **Liane Girouard:** **Title:** Support Director **Salary:** $136101 **Total Employees:** *132* **Total Salary:** $9912697

6: **Winnie Larkin:** **Title:** Vice President Development **Salary:** $202996 **Total Employees:** *59* **Total Salary:** $4954101

5 Reports:

1: **Lewis F. Hoffman:** **Title:** Development Director **Salary:** $140941 **Total Employees:** *54* **Total Salary:** $4189305

2: **John W. Ericson:** **Title:** Development Director **Salary:** $137285 **Total Employees:** *1* **Total Salary:** $137285

3: **Ernest Seedorf:** **Title:** Development Director **Salary:** $140354 **Total Employees:** *1* **Total Salary:** $140354

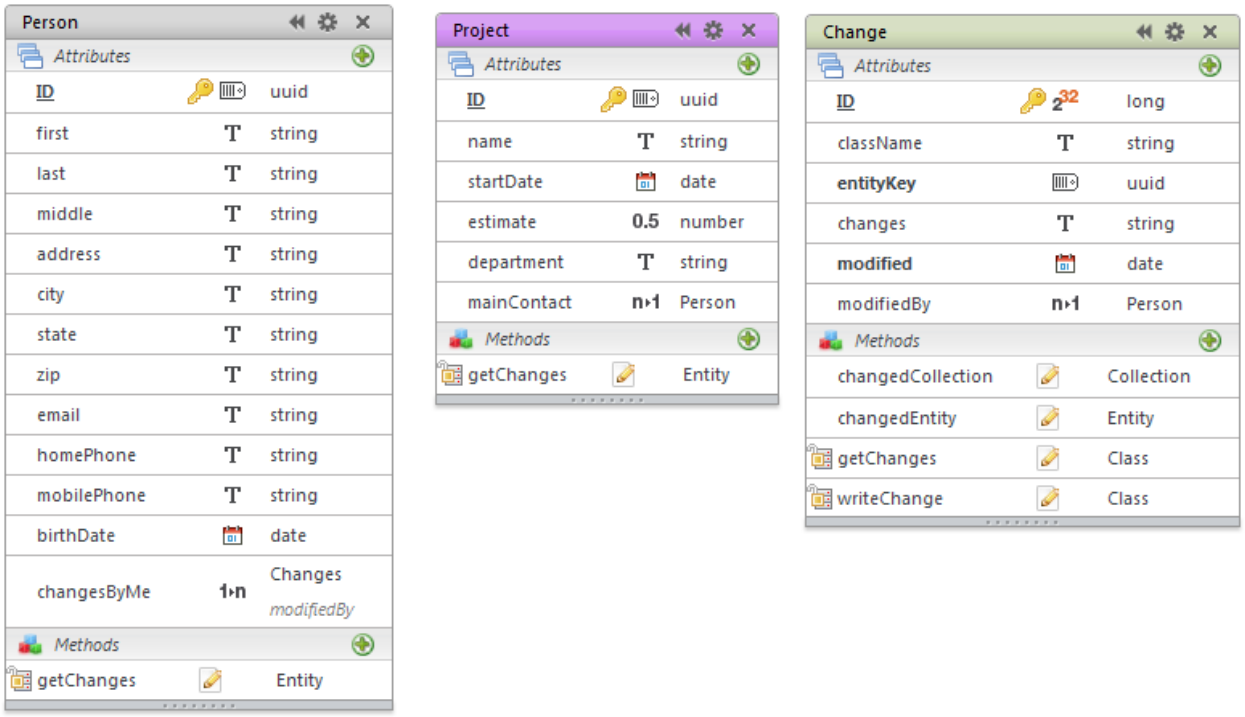
4: **Mike Fenton:** **Title:** Development Director **Salary:** $138437 **Total Employees:** *1* **Total Salary:** $138437

5: **Arnaldo Wright:** **Title:** Development Director **Salary:** $145724 **Total Employees:** *1* **Total Salary:** $145724

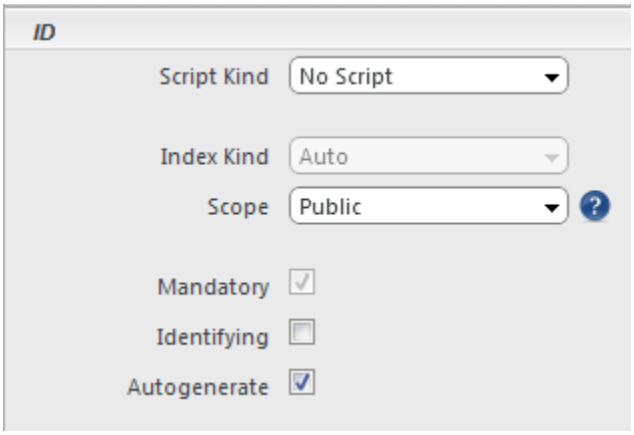
Change Tracking

This example provides a data model that you might use to track changes to entities in a variety of data classes. In this example, a 'change' is defined as an update to an entity where one or more storage or relation attributes have been modified. Our goal is to record what the previous value was in each of the altered attributes and store the results in the *changes* attribute of a Change entity. This model contains two additional dataclasses, Person and Project, so that we can illustrate how changes would be tracked. The Change dataclass and methods could be added to any project.

Notice that there are no relation attributes linking the Change dataclass to the entity that was changed. The information about which entity was changed is stored in *entityKey* and *className*. This example shows that you can associate dataclasses without relation attributes.



One of the first things to note about this model is that the keys for both Person and Project are UUIDs. This insures that the key value for each entity in Person matches no entities in Project. As we will see, using UUIDs as keys makes it easy to locate the changes that go with any particular entity. If you do decide to use UUIDs, you can choose to generate the value yourself or have Wakanda do it. To have Wakanda auto generate the UUID keys, select the auto generate checkbox in the properties panel.



The idea for change tracking is simple. We want to add code to the On Save event of dataclasses for which we want to track changes. This includes both Person and Project in this example. The On Save event for both Person and Project are identical and contain the following code.

```
onSave:function()
{
    ds.Change.writeChange(this);
}
```

Like in all dataclass events, the keyword **this** refers to the entity in which it is called. At the time the On Save executes, **this** holds an entity that may have modifications. Since this code is in the On Save, we are assured that the entity has passed validation as defined by the class's On Validate event. In the model, notice that the scope icon for *writeChange* is Public on Server and it therefore cannot be called directly from the browser. The code for *writeChange* is shown below.

```
writeChange:function(theEnt)
{
    if (!theEnt.isNew()) //if theEntity is not new
    {
        var theClass = theEnt.getDataClass(); //get its class
    //using its key, get a reference to the entity before it was updated
    var oldEnt = theClass(theEnt.getKey());
    var changedFrom = '';     //will hold a list of modifications
    for (var e in theClass.attributes)
    {    //cycle through all attributes
        var theAttrib = theClass.attributes[e];
        var kind = theAttrib.kind;
        if ((kind == 'storage') || (kind == 'relatedEntity'))
        {
            if (oldEnt[e] == null) //formerly null
            {
                if (theEnt[e] != null) // but not now
                    changedFrom += e + ' : null\r'; // so record null
            }
                else if (kind == 'relatedEntity')
            {  // is it now null or has the key changed
```

```
                if ((theEnt[e] == null) || (theEnt[e].getKey() != oldEnt[e].getKey()))
                    changedFrom += e + ': ' + oldEnt[e].getKey() + '\r';//append to changes
            }
            else //must be storage attribute
            {  // is it now null or has it changed
                if ((theEnt[e] == null) || (theEnt[e].toString() != oldEnt[e].toString()))
                    changedFrom += e + ': ' + oldEnt[e] + '\r'; //append to changes
            }
        }
    }
    if (changedFrom.length > 0)
    {   //if anything was modified
        var myInfo = sessionStorage.loginInfo; //placed in login listener
        new ds.Change({ //keep a record of the changes
            className: theClass.getName(),
            entityKey: theEnt.getKey(),
            modified: new Date(),
            modifiedBy: myInfo.myEmployeeID,
            changes: changedFrom.substr(0, changedFrom.length - 1)
        }).save();
    }
    }
}
```

With this code in place, every time a Person or Project is updated, the changes are recorded in a Change entity. In our example, we merely record the former value of each changed attribute, but you could expand this example to individually record each attribute change in a separate dataclass.

This method first checks to see if *theEnt* is new. We only track changes and not the original values in a new entity. If *theEnt* is not new, we get its dataclass and then lookup the corresponding entity in the datastore. Since this is a fresh query, the entity returned will be a copy of what the entity looked like before being modified. Next, we cycle through all of the entity's attributes, only taking action for storage and relation attributes. If the value in any of these attributes has been changed, we record the change in the variable *changedFrom*. After cycling through all the attributes, we test to see if there has been any modifications to record and if so, we create a new Change entity and save it. Notice that we are assuming some information placed into sessionStorage prior to this routine executing.

Now that we have all the changes recorded, how do we retrieve the changes for a given entity? To do this, we use the *getChanges* method of either Person or Project. Notice that these methods are defined at the entity level and their purpose is to return an entity's changes sorted in descending order of creation. If there are no changes, then this method returns an empty entity collection. The code for *getChanges* is identical for both Person and Project.

```
getChanges:function()
{
    return ds.Change.getChanges(this);
}
```

Notice that this calls a class level dataclass method in Change which is also called *getChanges* and uses as its argument the current entity **this**. The code for the Change dataclass *getChanges* method is shown below.

```
getChanges:function(theEntity)
{
    // return the changes that go with theEntity
    return ds.Change.query('entityKey = :1 order by ID desc', theEntity.getKey());
}
```

This method locates all changes for a given entity and doesn't include the dataclass name in the query. This operates properly because the key type of the classes we are tracking are UUID. If you use long integers as keys, you would need to add a test for the dataclass name to this query to differentiate between entities that have the same key but are in different classes.

The collection returned by this query is sorted in descending order by ID so that the most recent modification will be listed first.

Now that we can track changes and determine which changes go with any entity, we may need a way to do the reverse, that is, determine which entity goes with a change. This is the function of the *changedEntity* method of the Change dataclass. Here is the code.

```
changedEntity:function()
{
    // return the parent entity
    var theClass = ds.dataClasses[this.className];
    return theClass(this.entityKey);
}
```

This entity level dataclass method uses the stored class name (**this.className**) to determine the class and then returns an entity based on its key. In this example, the entity returned may be either a Person or a Company. It would be up to the routine calling this method to determine what type of entity has been returned.

Last, let's look at how we might use a collection of Changes. Since we are recording both the date of the change (Change.modified) and the person who made the change (Change.modifiedBy) it might be necessary to query for the changes made by an individual from a specific date forward. For example, consider this partial method:

```
var qString = 'modified >= :1 AND modifiedBy = :2';
var theChanges = ds.Change.query(qString, theDate, thePerson);
```

This code would locate all Change entities made by *thePerson* on or after *theDate*. Of course, this would be a collection of Change entities and not the entities the changes are about. This is where the *changedCollection* method is used.

```
changedCollection:function(className)
{
    var theClass = ds.dataClasses[className]; //get the class
    var classChanges = this.query('className = :1', className); //Change collection
```

```
    if (classChanges.length = 0)
        return theClass.createEntityCollection(); //empty collection
    else
    {
        var theIDs = classChanges.entityKey; //produces a scalar array
        return theClass.query('ID in :1', theIDs); //target class collection
    }
}
```

This collection level dataclass method takes a class name as an argument. Since a collection of Changes may be the modifications to entities from a variety of dataclasses, this parameter is used to indicate what type of collection to return. Since this is a collection level method the keyword **this** refers to the collection of Changes. As always, a query on a collection is restricted to entities already in the collection. So the query using *className* returns the entities from the original collection that are all for the same data class. If no entities in the original collection were for the specified class, then an empty collection is returned. Otherwise, we use Wakanda's ability to return a scalar array from a storage attribute to product an array of IDs (theIDs) that are then used to query the specified class. The result is a collection of entities in the specified class that had any changes in the original Changes collection. To illustrate how this method would be used, we add one line to the code above.

```
var qString = 'modified >= :1 AND modifiedBy = :2';
var theChanges = ds.Change.query(qString, theDate, thePerson);
var modifiedProjects = theChanges.changedCollection('Project');
```

At this point, *modifiedProject* would be an entity collection of the Project dataclass that had any changes made by *thePerson* since *theDate*.

This example has illustrated a quick and easy way to implement change-tracking in a generic way that, with little effort, could be added to any of your projects.

Hopefully this document has helped you to understand your options in defining a Wakanda Data Model and provided inspiration in your own applications.