Wakanda Security

or

"How to stop bad things from happening"

Table of Contents

Your tools in Wakanda 3
Concepts 3
Scope **4**
Privileges **6**
Restricting Queries **14**

Your Tools in Wakanda

1. Scope (Public, Public on Server, Protected, or Private)
2. Access control (groups and what they can be assigned to)
3. Setting up users and signing in
4. Inheritance, Restricting Queries and the onRestrictingQuery class event
5. Class methods
6. sessionStorage on the server
7. Calculated attributes
8. Transactions
9. HTTPS

Concepts

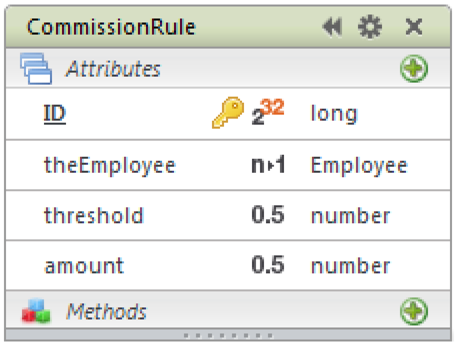
1. Setup your model so that items that should never leave the server are not available outside of the server regardless of access privileges.
2. Place as much business logic as possible in code that is only accessible to the server.
3. Use access control to ensure that users can only interact in a proper way with specific datastore classes and methods.
4. Wrap sensitive modifications into class methods and turn off direct access to insert, modify, and delete at the dataclass level. Use the 'promote' access privilege to allow the method to run and do its job.
5. Use either a restricting query or an onRestrictingQuery class event to restrict access to specific entities. The onRestrictingQuery event allows you to return your own entity collection so you can govern which entities are available. The onRestrictingQuery method has access to the authenticated user and sessionStorage.
6. If you want to provide read-only access to only some of your dataclass's attribute while allowing read/write access to others, you can use a calculated attribute that mirrors another attribute's value but doesn't allow you to set values.
7. NEVER trust information coming from the browser. Don't assume that attributes you provided for display-only purposes have not been altered.
8. Construct your UI code such that it mirrors the server-side restrictions but NEVER count on your client-side code to block a user's action.
9. Reference your pages through HTTPS so that the communication with the server is encrypted.

When designing a Wakanda application you have a variety of tools at your disposal to manage the information in a secure way. This document attempts to explain the various tools and how to apply them in a Wakanda application.

Scope

Your first, and probably most powerful, tool to 'lock down' a Wakanda application is **scope**. Scope is set during development and can be assigned individually for each dataclass, attribute and dataclass method. There are four scopes available in Wakanda: *Public*, *Public on Server*, *Protected*, and *Private*. Not all scopes are available for all elements. For example, a dataclass can only have a scope of either *Public* or *Public on Server* while attributes and dataclass methods can have any of the four scopes.

A dataclass that is *Public* is potentially available through the WAF and thus available from Interface pages served by Wakanda Server. A dataclass that is *Public on Server* is not available through the WAF nor through Wakanda Server's REST interface, and therefore is only available to code running on Wakanda Server. This setting is your first line of defense. In many applications, you may have support dataclasses or base dataclasses (more on this later) that you don't want users to even know about and certainly not have access to. When a dataclass is set to *Public on Server*, it is simply not available outside of the server regardless of access privileges. Moreover, regardless an attribute's or a method's scope, if it belongs to a dataclass that is *Public on Server*, it is not available outside the server.

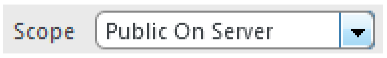

CommissionRule

Attributes

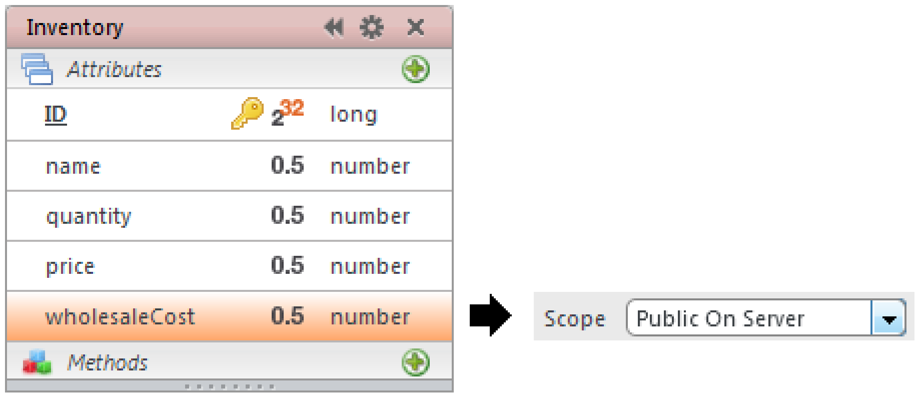
ID	2^{32}	long
theEmployee	n›1	Employee
threshold	0.5	number
amount	0.5	number

Methods

In the above dataclass, we are tracking commissions for employees based upon sales thresholds. We might not ever WANT this dataclass to be available to the browser, so we simply change its scope to *Public on Server*.


Scope: Public On Server

If a dataclass is set to *Public*, you may still want to restrict which attributes of the dataclass are available to the client. This is where the *Public on Server* scope at the attribute level is used. For example, in a dataclass of **Inventory**, you may have an attribute that tracks the wholesale cost. You may want to allow browser access to the **Inventory** dataclass, but not to the **wholesaleCost** attribute.


Inventory

Attributes

ID	2^{32}	long
name	0.5	number
quantity	0.5	number
price	0.5	number
wholesaleCost	0.5	number

Methods

Scope: Public On Server

Keep in mind that Scope is not something that can be changed per session. It is intended as a very secure mechanism to narrow client access regardless of user privileges. Items that are changed to *Public on Server* can still be accessed by server-side JavaScript.

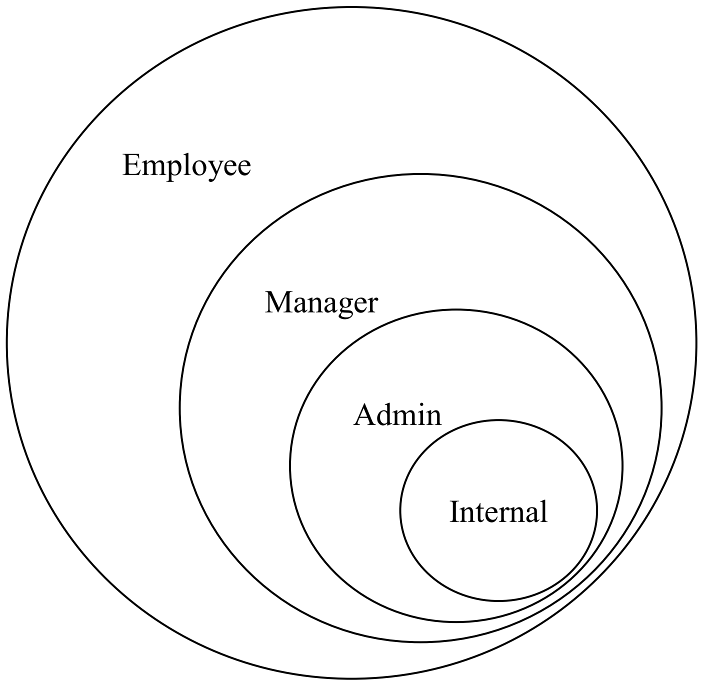
The default scope for dataclasses and attributes is *Public*. Remember to change the scope if your application requires it. The default scope for dataclass methods is *Public on Server*. If you want to allow a method to be called from the client, you must change its scope to *Public*.

Privileges

Your second, and probably most important, tool in Wakanda Security is **access control**. The purpose of the access control system in Wakanda is to govern what an authenticated user can do and what data they can access. Unlike Scope, access control causes the behavior of Wakanda items to change based on the currently authenticated user.

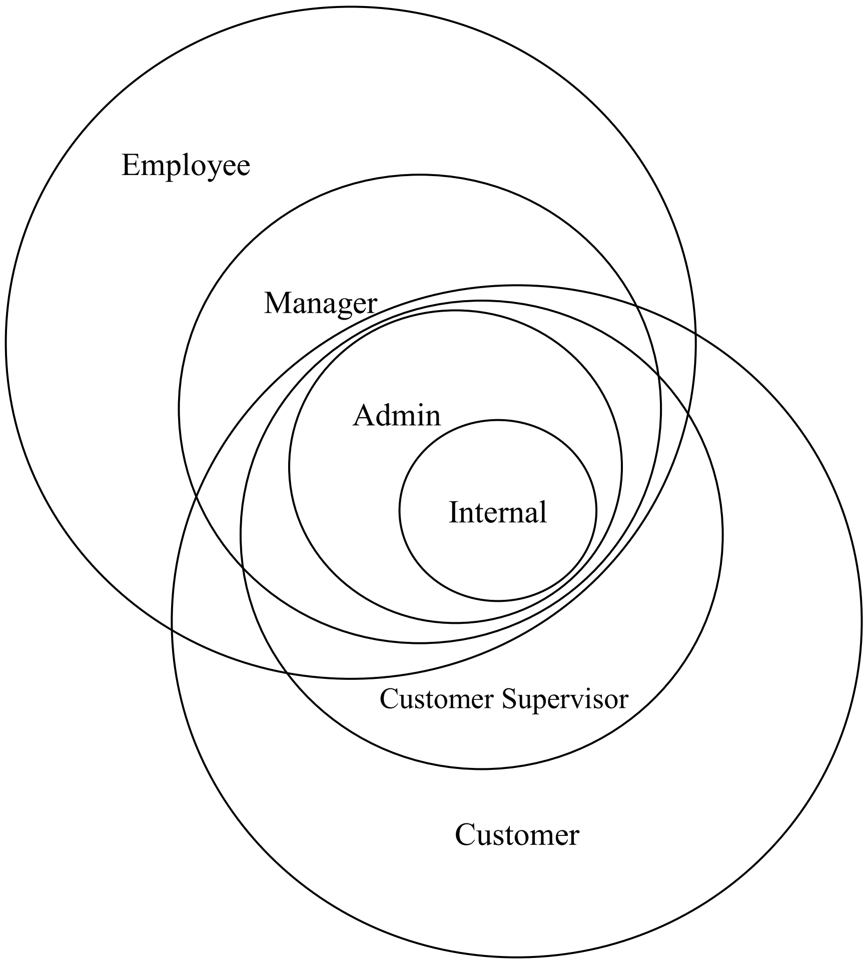
Access control starts at the group level. Groups are created and managed through the solution's Directory. You can access the Directory by opening the file named *SolutionName*.waDirectory. When you open this file, Wakanda Studio presents you with the Users and Groups editor. Keep in mind that the groups you define in the directory are available to ALL projects in the solution.

Groups have several purposes. They can be used as a way to bundle multiple users (more on this later). They can also be used as containers for other groups. This second ability provides a mechanism to build a hierarchy of access privileges. When a group is placed inside another group, the first group takes on all the access privileges of the containing group. For example, if you have both a group called *Manager* and a group called *Employee*, you may want to put the *Manager* group inside of the *Employee* group, thus allowing the *Manager* group to acquire all access privileges assigned to the *Employee* group. Later whenever the *Employee* group is allowed to do something, the *Managers* group will also be allowed to do it. At first, it may seem counter-intuitive that the contained group has the access privileges of the containing group, but consider the following group hierarchy diagram.



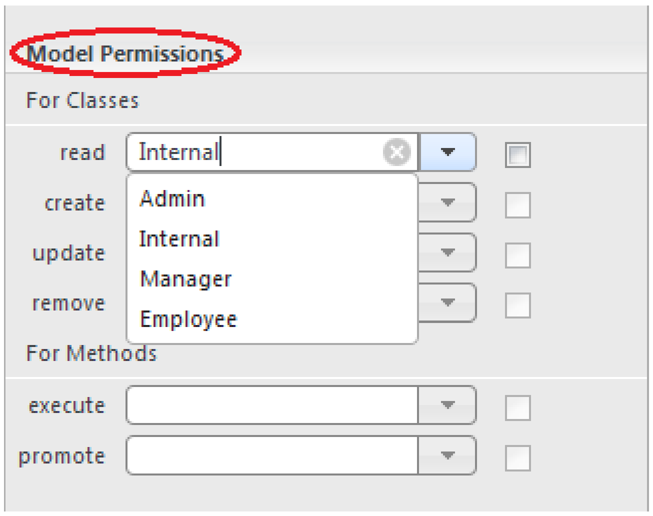
This diagram represents four groups and their relative access privileges. The *Employee* group contains the *Manager* group, which contains the *Admin* group, which contains the *Internal* group. It should be clear from this diagram that placing a user inside of the *Manager* group also places the same user inside of the *Employee* group.

You are not limited to strict hierarchy. For example, just because the *Manager* group has been placed inside of the *Employee* group, doesn't stop you from also placing *Manager* inside of another group. Access control is only limited by your imagination. For example, consider this diagram.

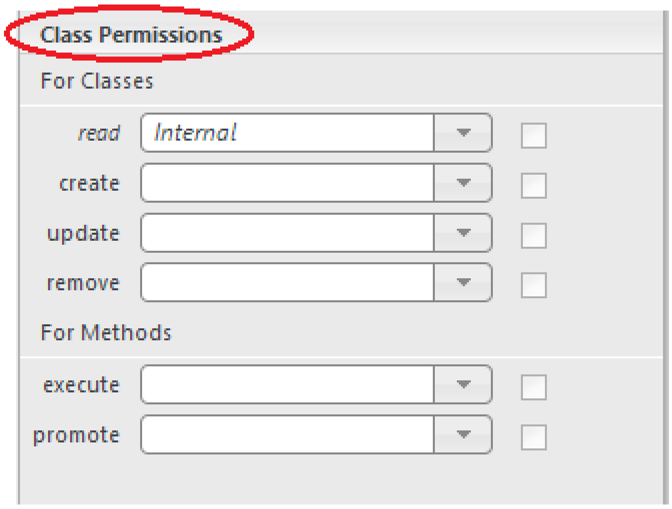


This diagram represents six groups and their relative access privileges. It includes the same hierarchy as above but adds two more groups intended for external users. The *Customer Supervisor* and *Customer* groups will be used to grant privileges to non-employee users. In this scheme, the *Admin* group has access rights that are equal to those of the *Manager* group plus the *Customer Supervisor* group. The *Customer Supervisor* group has the privileges of the *Customer* group. This diagram is for illustrative purposes. We do not show the extra two groups in the examples below. When you have defined your group's relative access privileges, you can begin to assign groups to model privileges.

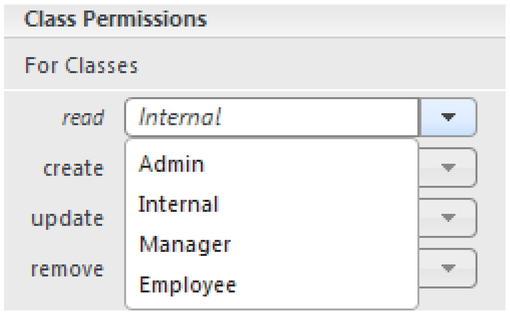
The last purpose of groups is to connect the privileges to a project's model. This is done by assigning a single group to specific control points in a Wakanda data model. These control points are available in an inheritance hierarchy starting with the model itself and then allowing finer grained control down to the dataclass and even method level. There are no control points at the attribute level in a Wakanda data model. If you assign a group to a control point at the model level, the group governs all the model's dataclasses and/or methods that have not been directly assigned to a group. To assign groups to the model's control point, make sure you have no dataclass selected in the editor and then assign the permissions to the model in the Properties panel.



In this example, we have assigned the group *Internal* to the default read permission of all dataclasses in the model. Of course, this can be overridden on a per dataclass basis. When permissions are inherited from the model, the group names appear in italics at the dataclass level.



There are six permission control points that you can assign to a group. Four of the control points govern access to entities and two govern access to methods. The first four control points are *read* (access an entity), *create* (save a new entity), *update* (save an existing entity) and *remove* (delete an existing entity). These control points let you indicate which group a user must be a member of to complete an operation. For example, using the two groups mentioned above, *Employee* and *Manager*, if we assign *Employee* to the *read* control point of a specific data class, then a user would have to be a member of either the *Employee* group or the *Manager* group to view entities in that dataclass. If we made this group assignment for the entire model, then only *Employees* and *Managers* would have access to ALL dataclasses provided we don't override this control point at the dataclass level.



Example of model level permissions being overridden at the dataclass level

If a user is not a member of the *read* group for a dataclass, any attempt to access entities in the dataclass will result in an error. This includes all functions that attempt to access an entity or an entity collection whether through the browser or on Wakanda server. It does not include the ability to create an empty entity collection using the class method createEntityCollection() nor the ability to create new entities. Since without *read* privileges, a user cannot access entities this control point also governs *update* and *remove*. After all, if you can't access an entity, you can't update or remove it.

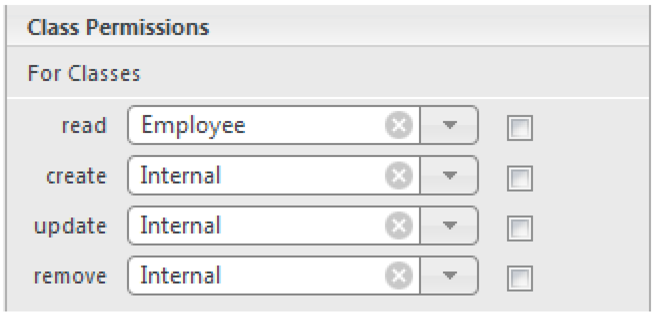
So far we have focused primarily on controlling access to entities. But there are circumstances in data applications where users may require *read* privileges, but should not have the ability to make modifications. You can control this using the other three dataclass control points: *create*, *update*, and *remove*. First, let's examine how these other three control points behave.

- *Update* control governs whether an existing entity can be saved. If the current user is not a member of the group attached to a dataclass's *update* control point, an error is thrown when *entity*.save() is called for an existing entity.
- *Remove* control is similar to *update* control. If the current user is not a member of the group assigned to the *remove* control point, an error is thrown when *entity*.remove() is called. If the remove() function of the *entityCollection* is called, a different error is thrown.
- *Create* control governs whether a new entity can be saved. If the current user is not a member of the group attached to the dataclass's *create* control point, an error is thrown when *entity*.save() is called for a new entity. *Create* control is unlike *update* and *remove* in that it does not pre-suppose *read* access to the dataclass.

Use these control points to restrict user actions on all entities of a dataclass. All privilege errors can be trapped using a standard Try-Catch block. All permission control points govern access whether the dataclass is being accessed on the browser or on the server.

The two control points that govern methods are named *execute* and *promote*, and like the other control points they can be assigned at the model level, and then overridden at the dataclass level. But unlike the others, they can be further overridden on a per method basis. The *execute* control point is just what it sounds like. Users will need to be members of the group assigned to this control point in order to execute the method. The *promote* control point isn't really a control point. It is a companion piece of information to the *execute* control point. It tells the model under what privileges the method should run.

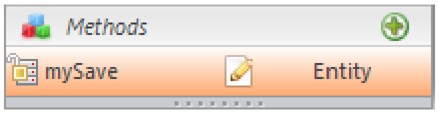
For example, if you don't want anyone to directly modify entities in a specific dataclass, you assign the *Employee* group to the read control point but assign an empty group named *Internal* to the *create*, *update*, and *remove* control points. Since the *Internal* group has no users, no one can update the entities directly.


Class Permissions

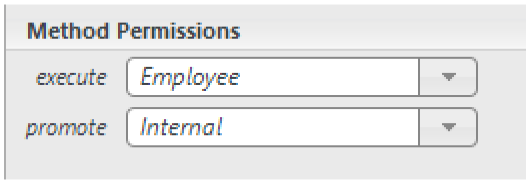
For Classes

read	Employee
create	Internal
update	Internal
remove	Internal

Instead, you create a dataclass entity method (one that operates on individual entities) to update the entity.



You use this method to verify that the entity is valid and that the attributes you didn't want changed were not. After checking the entity, you save it (more on this below). In order to allow users to call this method, you assign the *Employee* group to the *execute* control point. But wait, if the user is in the *Employee* group, he/she can't update entities because we assigned the group *Internal* to the other three dataclass control points. If we don't do something, the method will fail with a privileges error when it attempts to save the entity. This is where the *promote* privilege is used. When the method executes, it temporarily runs under the permissions of the group assigned to the *promote* control point. So in our example, we would assign the *Internal* group to the *promote* control point and this would allow the method to modify entities in this dataclass.


Method Permissions

execute: Employee

promote: Internal

The promotion is only temporary for the duration of the method call.

Except for *promote*, permission control points that haven't been assigned a group are left 'wide open'. So you may want to adopt a strategy of assigning groups at the model level in order to restrict access and override the groups on a per dataclass basis. For example, you may want an *Internal* group with no users that you then assign at the model level where appropriate.

Notice how we have completely avoided discussing users until now. This is because users become members of groups in a variety of ways. The simplest way to create and manage users is through the solution's directory which is stored as part of the solution and can be managed by the Users and Groups editor in Wakanda Studio. Here, you can create users and assign them to groups. In many applications, this simple approach is flexible enough. In addition, various server-side objects let you manipulate and modify the directory's users and groups. Wakanda solutions have a default user (admin) and group (Admin) when first created. The Admin group has access to various Wakanda functions and represents system administrators. You may want to assign the Admin group some permissions in the data model and provide access to all the entities in your dataclasses by using restricting queries.

But what if we want to define some users 'on the fly' and place them in groups. For example, say you have a dataclass in your model named *Employee* and you want to base access to your application on this dataclass. Wakanda handles this situation quite elegantly. To do this, you add a login listener to your application by using the setLoginListener() method of the directory like this:

```
directory.setLoginListener('myLogin');
```

You place this code in a JavaScript file designated as a bootstrap file for the project. Then, in the JavaScript file located at *projectfolder*/scripts/required.js (or a JavaScript file referenced by an include in *required.js*), you define the function *myLogin*. This function will be automatically called to authenticate a user and will be passed the userName and password entered by the user from the browser or sent by the Wakanda Server function loginByPassword(). The method might look something like this:

```
function myLogin(userName, password)
{
var theEmployee = ds.Employee({login:userName});
  if (theEmployee == 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 (theEmployee.password == directory.computeHA1(userName, password))
      {
            var theGroups = [];
            switch (theEmployee.accessType){
                case 1:
                    theGroups = ['Internal'];
                    break;
                case 2:
                    theGroups = ['Admin'];
                    break;
                case 3:
                    theGroups = ['Manager'];
                    break;
                case 4:
                    theGroups = ['Employee'];
                    break;
            }
            return {
    ID: theEmployee.ID,
    name: theEmployee.login,
    fullName: theEmployee.fullName,
    belongsTo: theGroups};
        }
        else
            return { error: 1024, errorMessage:"invalid login" };
  }
};
```

This example function assumes that we have four groups in the solution and that we have assigned them to control points where appropriate. The function installed by setLoginListener is provided two parameters, *username* and *password*, and can return three possible values.

- If the login listener returns **false**, then control is passed back to Wakanda Server to authenticate the user using the directory. This would be the case if we logged in as a predefined system administrator user in the directory. This would work even when the application has no data in the Employee dataclass.
- If the login listener returns an **error object**, composed of two attributes named *error* and *errorMessage*, then the authentication fails.

Otherwise, the login listener needs to return an object with four attributes named *ID*, *name*, *fullName*, and *belongsTo*. This object will represent the user in the newly created session and will be available elsewhere through the directory object's currentUser() method. When you do access currentUser(), the userID will have been converted into an UUID so you may want to force the ID property into an UUID format during the assignment in the login listener. Group membership for the session is set via an array of either group names or group IDs in the *belongsTo* property. The example above uses group names.

The *myLogin* function above uses the *Employee* dataclass to validate the *username* and *password* parameters. In this example, we locate an *Employee* entity with userName and then verify that the previously-stored encrypted password matches.

You might notice a potential issue in the above code. What would happen if we have already assigned access privileges to the *Employee* dataclass such that it requires membership in the *Manager* group to read entities. How would we allow our login listener to acquire the needed privileges? After all, it is this routine that is determining the privileges for each user. This is where an optional second parameter of the setLoginListener() function plays a role. For example:

```
directory.setLoginListener('myLogin', 'Manager'); // run with Manager permissions
```

The second parameter can be a group object, a group ID, or a group name. The example above uses the group's name. This code will allow the login listener to run with the privileges of the group *Manager*. So if the group *Manager* is assigned to the *read* control point for the *Employee* dataclass our method will function correctly.

You might also notice that the only thing we return in the login listener to authenticate access is a simple object with four properties: *ID*, *name*, *fullName*, and *belongsTo*. This object need not represent a user in the solution's directory, but it should not share an ID with one either. You may want to store more information during authentication. For example, since we have already identified the *Employee* entity in the above code, we may want to keep its ID and the ID of the employee's manager in the sessionStorage object. To do this, we add a fifth property, named *storage*, to the object that we return. Whatever you assign to *storage* will become part of the sessionStorage object. In the example, we assign an object with one property *loginInfo*, which is itself an object with two properties: *myEmployeeID* and *myManagerID*.

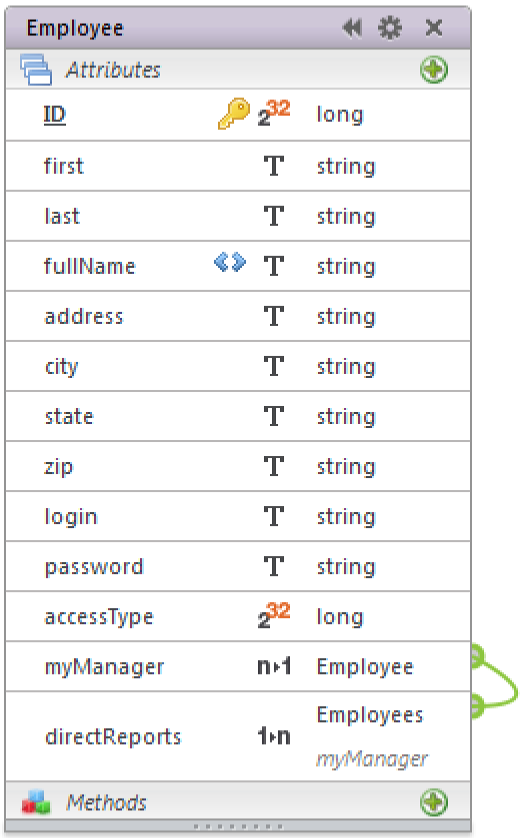
```
        var forStorage = {
myEmployeeID: theEmployee.ID,
myManagerID: theEmployee.myManager.ID
};
        return {
ID: theEmployee.ID,
name: theEmployee.login,
fullName: theEmployee.fullName,
belongsTo: theGroups,
storage: {loginInfo: forStorage}
}; // will become part of new session's storage
```

With this change, any downstream function that needs to determine which employee is currently signed in can simply examine the sessionStorage.loginInfo.myEmployeeID property. This is particularly useful when using On Restricting Query events for dataclasses.

***Note**: You should not place complex objects managed by Wakanda Server into sessionStorage, such as entities, entity collections, files, and folders. Instead, you can store the information needed to reference the object.*

Restricting Queries

So far, we have examined how to lock down client access to datastore model items and to control access to those items on a per user basis. But what if we want to allow a specific user to access only specific entities? If the current user **does not** belong to a group with *read* access to a dataclass, they can **never** see entities in that dataclass. If the current user **does** belong to a group with *read* access, they can see **all** the entities. This can be a problem where sensitive information is concerned. Using the *Employee* dataclass implied by the example login listener method above, we may want to allow some users to access only their own *Employee* entity while other users access multiple *Employee* entities. In order to do so, all users will require *read* access to the *Employee* class. But if all users have *read* access, how can we restrict which entities the members of the *Employee* group can see? We suggest using Restricting Queries. Let's use the following model as a reference:



ID	2^{32}	long
first	T	string
last	T	string
fullName	T	string
address	T	string
city	T	string
state	T	string
zip	T	string
login	T	string
password	T	string
accessType	2^{32}	long
myManager	n›1	Employee
directReports	1›n	Employees *myManager*

There are two types of restricting queries: Restricting Query property or the On Restricting Query event.

One uses a restricting query string entered into the dataclass directly as part of the model. The string has syntax similar to the standard query string syntax. Special variables are available in restricting query strings to allow access to the current user's attributes. For example, 'fullName = $userName' is a valid restricting query string that would restrict the entities of a derived class to only those entities where the *fullName* matches the currently authenticated user's name. Restricting query strings can include any combination of attributes in the class combined with comparison operators and values.

However, using 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 *onRestrictingQuery*. This event's function is called whenever all the dataclass's entities are accessed, including the dataclass method all() as well as query() and find(). It also includes all direct means of accessing an entity such as locating an entity by key.

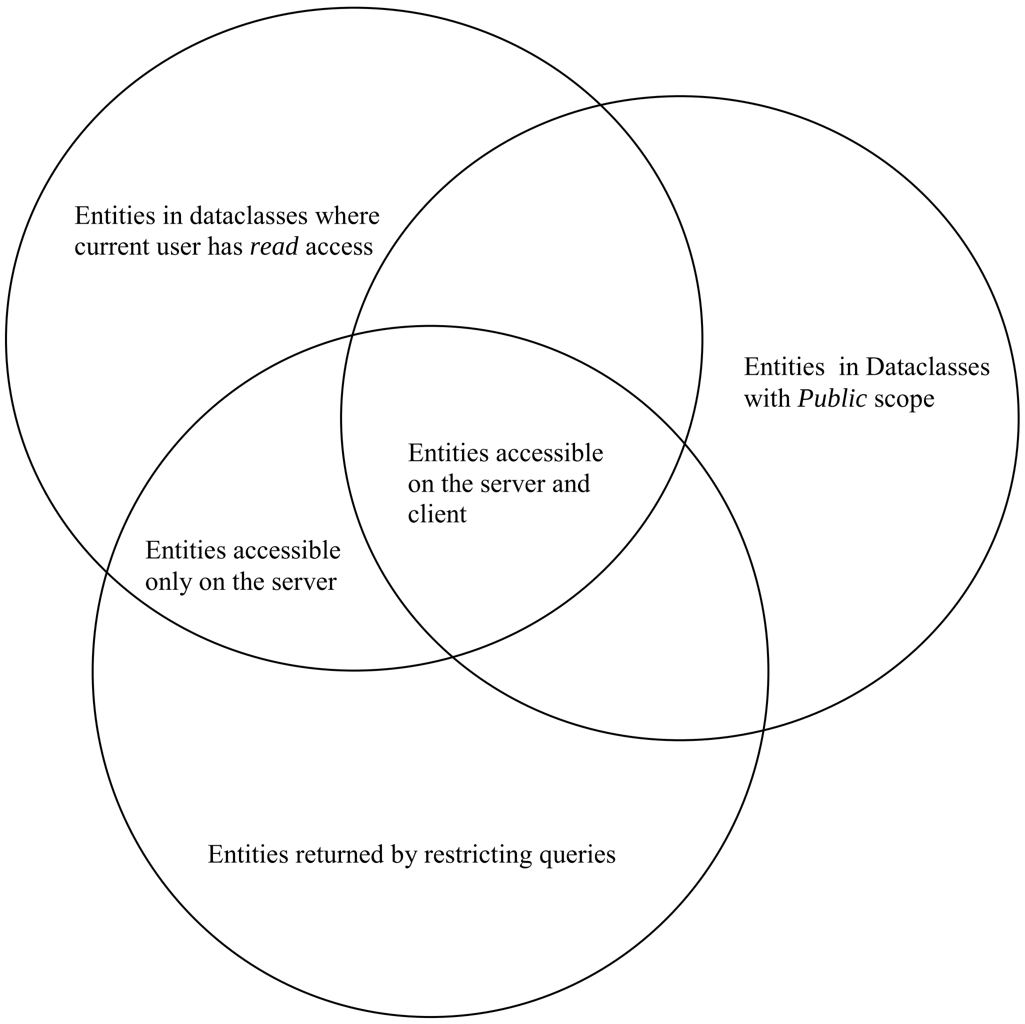
The goal of the *onRestrictingQuery* event is to return an entity collection that represents all entities in the dataclass that are accessible under current conditions. The advantage of this approach is that you can programmatically provide an entity collection, which can be different for each authenticated user. Using the example model above, we may want our *Employee* dataclass's *onRestrictingQuery* event to be something like this:

```
onRestrictingQuery:function()
{
var result = ds.Employee.createEntityCollection(); // empty collection
var session = currentSession(); // get the current session
var loginInfo = sessionStorage.loginInfo; // get the login info
  if (session.belongsTo('Admin')) // if current user is in the Admin group
result = ds.Employee.all(); // return all entities
  else if (loginInfo != null) // if session has valid login info
  {
var myID = loginInfo.myEmployeeID; // get the Employee ID
result = ds.Employee.query('ID = :1 OR myManager.ID = :2', myID, myID);
// the query will return the current user and all of the direct reports
}
  return result; //return the collection
};
```

This code is the *onRestrictingQuery* event. Here we have chosen to first examine if the current user is in the Admin group, granting access to all the entities in the dataclass. It is common to provide this failsafe mechanism so that at least one user can get to all the entities. For all other users, we return an entity collection for the current user plus any employees that are direct reports of the current user. Notice that we use the ID in sessionStorage placed there by our login listener. Having the employee ID of the current user available in sessionStorage can be very convenient when other events, methods, and restricting queries need information about the authenticated user.

You may have noticed a potential issue in the above code. We are querying the *Employee* dataclass inside the restricting query of that same dataclass. Won't this cause the restricting query to run again, and then again, etc. in a recursive loop? Wakanda handles this automatically. When you query a dataclass from its own restricting query, Wakanda allows the query to access ALL entities and does not run the restricting query code again.

Access to entities in a Wakanda application can be illustrated by the following diagram:


Entities in dataclasses where current user has *read* access
Entities in Dataclasses with *Public* scope
Entities accessible on the server and client
Entities accessible only on the server
Entities returned by restricting queries

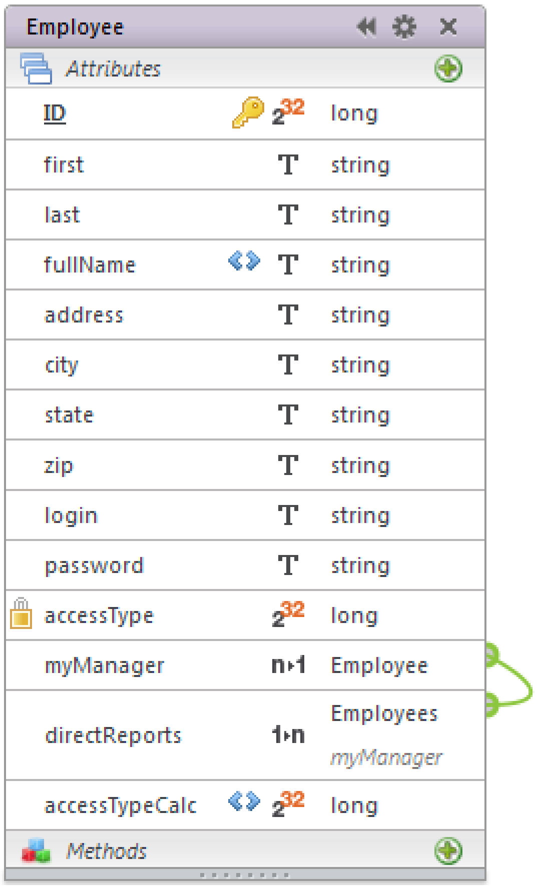
The behavior of Permissions control points is an all or nothing proposition. For example, if a group has *update* privileges, it can modify and save any existing entities that it can access. But what if you want finer control over validating the update, create, or remove process? For example, say you want to allow updates to only some of the entities that can be accessed or you want to allow some users to update particular attributes while others cannot. There are several ways you can approach this situation:

- Use calculated attributes to provide versions of other attributes that are validated upon change.
- Add code in the *onSave* or *onRemove* dataclass events to reject inappropriate actions.
- Turn off *create*, *update*, and *remove* privileges by assigning a group to these control points. Instead, create entity level dataclass methods that replace the .save() and .remove() methods. Promote these methods so that they have the access privileges

needed to operate on the entities. Use the methods to validate the saves. Then, on the client, call these entity methods instead.

- Turn off *create*, *update*, and *remove* privileges by assigning a group to these control points. Instead, create dataclass methods that will handle these operations. Allow the methods to accept data as parameters that are not entities. Double check the values, locate the entities and update or delete them.

Let's explore each of these possibilities by first creating a calculated attribute. Using the *Employee* dataclass on page 14, let's say you don't want just anyone to change the *accessType* attribute. So instead we set *accessType* to a scope of *Private* and create a new calculated attribute to replace it.



Employee

Attributes

ID	2^{32}	long
first	T	string
last	T	string
fullName	T	string
address	T	string
city	T	string
state	T	string
zip	T	string
login	T	string
password	T	string
accessType	2^{32}	long
myManager	n▸1	Employee
directReports	1▸n	Employees *myManager*
accessTypeCalc	2^{32}	long

Methods

The new attributes in the `On Get` function would be simple.

```
onGet:function()
{
  return this.accessType;
}
```

Next, we add an On Set function that controls the conditions that must be met to allow a value to be assigned. In this example we only allow members of the *Admin* group to make this assignment.

```
onSet:function(value)
{
  var theSession = currentSession(); // get the current session
  if (theSession.belongsTo('Admin')) // if user is in the Admin group
      this.accessType = value; // allow the assignment
}
```

Users that are not in the *Admin* group will not be able to set the attribute and no error is returned. This simple and convenient method is practical when only a few attributes need this type of validation. If we have many attributes to test, it is better to use one of the following methods.

The next method is the onSave and/or onRemove dataclass events. We can use these events to verify that the action should be allowed. Say we have an *Invoice* entity that requires an invoice date, but can't be saved if the date is earlier than today unless the current user is in the *Admin* group. Our On Save method might look like this:

```
onSave:function()
{
  var theError = {errorCode: 0, errorMessage: ''}; // same as no error
  var theSession = currentSession(); // get the current session
  if (!theSession.belongsTo('Admin')) // if user is not in the Admin group
  {
var today = new Date(); // get today's date
// make sure the date portion of the invoice date matches today's date
      if (this.invoiceDate.toLocaleDateString() != today.toLocaleDateString())
      {
          theError = {
              errorCode: 20,
              errorMessage: 'You cannot postdate invoices'};
          }
      }
  }
  return theError; //always return whatever is in theError
}
```

In this example code, we first define a variable to hold an error object. If the method ends up returning an error object with an *errorCode* of 0, it is the same as returning no error. Next we check if the current session belongs to the *Admin* group. If it does, the save is permitted because members of the *Admin* group are allowed to save invoices with postdated invoice dates. Note how easy it is to verify that the current session is a member of the *Admin* group. If the current session isn't in the *Admin* group, we verify that the invoice date is today.

While the previous coding example provides a good method in validating an entity, how can we be sure that attributes we provided to the browser, but didn't want modified, were left unchanged? Below is one possible technique.

First, we assign a group to the *create* and *update* permission control points. This prevents entities from being directly saved. Next, we create an object, named *entityPattern* in the code below, that holds information on which attributes can be modified during an update and create. We do this for all dataclasses whose scope is *Public*. By default, this object includes all storage attributes and all relation attributes whose scope is *Public* and also allow you to assign an entity/value. We define this object in the login listener method and include it in the resulting object's storage attribute. The code below would be added to the login listener method just before the resulting object is returned.

```
var entityPattern = {};
for (var e in ds.dataClasses){   //cycle through all dataclasses
  var theDataClass = ds.dataClasses[e];
  if (theDataClass.getScope() == 'public') { //if the dataclass is public
      entityPattern[e] = {modifyAttributes: {}, createAttributes: {}};
      for (var a in theDataClass.attributes){
          var theAttribute = theDataClass.attributes[a];
          if (theAttribute.scope == 'public') { //if the attribute is public
              var attKind = theAttribute.kind;
              if ((attKind == 'storage') || (attKind == 'relatedEntity')) {
                  if (theAttribute.name != 'ID')
                      entityPattern[e].modifyAttributes[a] = theDataClass.attributes[a];
                  entityPattern[e].createAttributes[a] = theDataClass.attributes[a];
              }
          }
      }
  }
}
var forStorage = {myEmployeeID: theEmployee.ID, entityPattern: entityPattern};
```

Without any change to the code above, the *entityPattern* object will contain all qualified public attributes for all public dataclasses. But you will probably want to control the construction of this object based upon the authenticated user, removing any attributes that shouldn't be changed during creation and modification. For example, we may only want to allow updates to the *accessType* attribute of the Employee dataclass for users that are administrators.

```
var entityPattern = {};
var session = currentSession();
var isAdmin = session.belongsTo('Admin');
for (var e in ds.dataClasses){   //cycle through all dataclasses
  var theDataClass = ds.dataClasses[e];
  if (theDataClass.getScope() == 'public') { //if the dataclass is public
      entityPattern[e] = {modifyAttributes: {}, createAttributes: {}};
      for (var a in theDataClass.attributes){
          var theAttribute = theDataClass.attributes[a];
          if (theAttribute.scope == 'public') { //if the attribute is public
              var attKind = theAttribute.kind;
              if ((attKind == 'storage') || (attKind == 'relatedEntity')) {
                  if ((e != 'Employee') || (a != 'accessType') || isAdmin)) {
                      if (theAttribute.name != 'ID')
```

```
                            entityPattern[e].modifyAttributes[a] = theDataClass.attributes[a];
                        entityPattern[e].createAttributes[a] = theDataClass.attributes[a];
                    }
                }
            }
        }
    }
}
var forStorage = {
myEmployeeID: theEmployee.ID,
myManagerID: theEmployee.myManager.ID,
entityPattern: entityPattern
};
```

Once the *entityPattern* object is defined and placed in sessionStorage, the rest becomes fairly simple.

Next, we create an entity method with promoted privileges as described on page 11. The promoted privileges allow this method to create and update entities even when the user's privileges do not allow it.

```
mySave:function()
{
return entitySave(this);
}
```

The *entitySave* method is defined in the *required.js* file as:

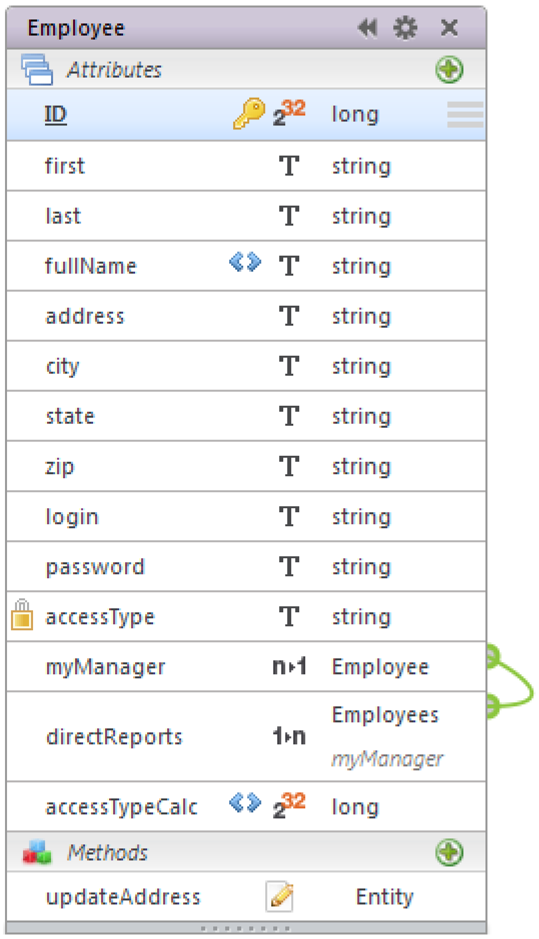
```
function entitySave(theEntity)
{
  var result = {saved: false, errorCode: 0, errorMessage: ''};
  var entityPattern = sessionStorage.loginInfo.entityPattern;
  if (!this.isNew()) { //if this is not a new entity
      var theClass = theEntity.getDataClass(); //get the dataclass of the entity to save
      var theClassName = theClass.getName(); //get the dataclass name
      var storedEntity = theClass(theEntity.getKey()); //find the same entity on disk
      if (storedEntity != null){
          if (this.getStamp() == storedEntity.getStamp()){ //do the stamps match?
              for (var e in entityPattern[theClassName].modifyAttributes) {
                  //cycle through entityPattern.modifyAttributes
                  storedEntity[e] = theEntity [e]; //copy value into old entity
              }
              try {
                  storedEntity.save();
                  result.saved = true;
              }
              catch (e) {
                  result.errorCode = e.errorCode;
                  result.errorMessage = e.errorMessage;
              }
          }
          else { //entity was modified by another user
```

```
                result.errorCode = 20;
                result.errorMessage = 'Entity already modified'
            }
        }
        else {
            result.errorCode = 10;
            result.errorMessage = 'Entity not found'
        }
    }
    else { //new entity being saved, different list of attributes
        var newEntity = theClass.createEntity();
        for (var e in entityPattern[theClassName].createAttributes) {
            //cycle through entityPattern.createAttributes
            newEntity[e] = theEntity [e]; //copy into new entity
        }
        try {
            newEntity.save();
            result.saved = true;
        }
        catch (e) {
            result.errorCode = e.errorCode;
            result.errorMessage = e.errorMessage;
        }
    }
    return result;
}
```

Notice that this last method is generic and works for all entities. It looks up and copies the appropriate values from the modified entity into the original one stored in the datastore. This method stops any values not described by *entityPattern* from being saved. Once this technique is coded, the bulk of the work is setting up the *entityPattern* object correctly during authentication. This mechanism puts the effort on back-end code and allows client-side entities to be easily saved.

The last method avoids the issue of whether an entity's attributes were modified on the client by only allowing parameter values to be written to the entity. This simplifies back-end code at the cost of client-side effort.

First, we assign a group to the *update* permission control point so that entities cannot be directly saved. Then, we create an entity level method with promoted privileges as described above. In our example, this method is named *updateAddress* and its purpose is to provide a method to update an Employee's address, city, state, and zip.


Employee

Attributes

ID		long
first		string
last		string
fullName		string
address		string
city		string
state		string
zip		string
login		string
password		string
accessType		string
myManager	n›1	Employee
directReports	1›n	Employees *myManager*
accessTypeCalc		long

Methods

updateAddress	Entity

The *updateAddress* method is an entity level method, but we don't save the entity directly. Instead we locate the same entity in the datastore, check that its stamp still matches, and then assign the parameters passed to the method directly to entity attributes. Afterwards, we save the entity. Notice that we only allow the values passed as parameters to make their way into the entity.

```
updateAddress:function(address, city, state, zip)
{
  var storedEntity = theClass(theEntity.getKey()); //find the entity in the datastore
  if (storedEntity != null){
      if (this.getStamp() == storedEntity.getStamp()){ //do the stamps match?
          storedEntity.address = address;
          storedEntity.city = city;
          storedEntity.state = state;
          storedEntity.zip = zip;
          storedEntity.save();
      }
  }
}
```