Introduction to Wakanda Client-side Development

wakanda

This preliminary documentation does not reflect the final content and design.

This manual introduces you to client-side development for a Wakanda project. This material assumes that you have already read the *Wakanda Server-Side Concepts* manual and have familiarized yourself with the Wakanda's concept of datastore classes along with its unique and powerful capabilities as a database engine. You should be comfortable with the terminology introduced in that section and have already explored creating datastore classes in a model. We also assume that you have had some basic exposure to Wakanda Studio and have read the "GUI Designer" chapter in the *Wakanda Studio Reference Guide*.

Prerequisites

Here are the prerequisites for the developer who wants to begin working with Wakanda:

- JavaScript development background
- jQuery familiarity (preloaded with Wakanda)
- No advanced database knowledge necessary

About Wakanda

Wakanda is an open-source platform for creating "Web Applications" that have many of the same characteristics as desktop applications and yet run in the context of a web browser. It is an all-encompassing development and deployment solution that marries a powerful NoSQL database engine and web server with a widget-centric JavaScript framework on the client-side. Using only standards-based and open-source tools, Wakanda leverages the advancements in web browsers to empower developers to create web applications whose functionality is indistinguishable from a desktop application.

Wakanda is more than just a framework. It includes a web server with an integrated NoSQL database engine. The Wakanda Studio provides robust graphical tools for defining datastore classes, attributes, and methods at the heart of an application. The development environment is entirely based on HTML, CSS, and JavaScript, making it easily accessible to the majority of web developers.

As you work with and build solutions in Wakanda, you will find that they most closely resemble the Single-Page Application (SPA) method of development. While you are certainly able to transition the user from one page to the next, each page can be so richly interactive that the need to do so is greatly minimized. Once a page is delivered to a client, there is little need for postbacks to the server. Controls on your page will be refreshed as needed via AJAX calls based on user interaction. Using a robust set of widgets, you can easily build pages with complex interactions and very little code.

Wakanda's backend is made up of a web server, a database server, and a REST/JSON server that uses JavaScript as its programming language. In Wakanda, all the elements are designed to work in harmony for ease of development and deployment.

A Bit of Background

The "Web Application" has been a dream of developers for a number of years. However, limitations in the HTML specification and poor consistency in browser implementations have made it difficult to develop web-based solutions whose interface and behavior could approximate a desktop application.

A solution chosen by many developers to sidestep the variances between browsers was the "Rich Internet Application (RIA)." These solutions were developed in third-party (often commercial) applications like Flash or Silverlight in order to more closely approximate the feel of a desktop application in a web context. These solutions typically require the end-user to have a particular plug-in installed in order to "run" the application. As more and more web usage transitions to mobile devices (phones, tablets, and notebooks), these types of solutions are often not supported or require more bandwidth and/or horsepower than a solution built entirely on web standards.

With the advent of the HTML5 specification, we are finally seeing browser vendors take a keen interest in implementing features specific to the needs of both developers and designers. The browser makers across the board are making an unprecedented effort to assure their products render web content according to the specification.

When we talk about HTML5, we're including the advancements in CSS3 and JavaScript. The whole environment represents a set of standards all browser vendors are targeting in their most current releases. While there will always be small variances in browser implementations, the JavaScript tools today provide an easy way to test for compatibility on a per feature basis.

The Essential Parts of Wakanda

The Wakanda development environment is composed of two applications: **Wakanda Server** and **Wakanda Studio**. Wakanda Studio is the tool you will use to define every aspect of your project: from datastore classes, attributes, and methods to the client-side HTML pages, CSS files, and JavaScript code. Wakanda Server will act as both your database engine and web server during development as well as for your final deployment.

Using Wakanda Studio, you will create all the different aspects of your project. Some of these parts that you create will only run on the server; while other portions you will design specifically for client-side execution in a web environment. What follows is an introduction to the key concepts that we will be working with throughout this document.

Model

The model is the collection of all the datastore classes used by your web application. Everything you define as part of the model—datastore classes, attributes, and methods—are only executed by the server.

While it is certainly possible for client code to call a method on the server, the JavaScript in the method executes in the server environment and the results of the operation are returned to the client.

Data Provider (ds)

The data provider is a reference to the application's set of data as well as the datastore class's information and methods. You reference the data provider using the "ds" application-level object, which has a number of functions to facilitate accessing and manipulating data and/or classes. Within server-based code (i.e., datastore class methods) **ds** is the only way to interact with the data model.

The "ds" object is an interesting and potentially confusing feature in the Wakanda client environment. The datastore object exists in both the server and client (WAF) code; however, they are not exactly the same thing. In the server environment, the **ds** can perform many operations that simply wouldn't make sense from a web client perspective. For instance, the **ds** object on the server can start, commit, and rollback transactions.

When you write code that will be executed on the server (i.e., datastore class methods) the "ds" object refers to the datastore. You write code to be executed on the client references the Data

Provider even though you still use “ds” as the object. The Data Provider is the proxy of the datastore on the client.

Essential portions of the datastore have been proxied over to the WAF. In client-executed code, you will most often use data provider calls when you need to interact directly with the database. For instance, when a new query needs to be performed, the data provider will be used to create a new entity collection to change the data presented to the user. Using the data provider from within the client-executed code will initiate AJAX calls to the server.

Wakanda Application Framework (WAF)

The Wakanda Application Framework (WAF) is the name given to the client-side JavaScript framework. The WAF provides the client with the ability to access the backend datastore, obtain detailed information about a datastore class, manipulate entity collections and entities, interact with datasources, and handle all communication with the backend in an asynchronous manner.

While there are frequently many similarities in the JavaScript you write for the backend and frontend, it is important to keep in mind that the WAF does not possess the same functionality in the backend datastore object. The WAF contains only those capabilities that are relevant to code being run in a client-side environment. Additionally, there are many cases where the syntax of commands differs between the server-side and client-side code.

Entity Collections & Entities

The overall concept of entity collections and entities are identical whether your code is server-side or client-side. There are however some differences in the syntax of commands you will use to work with these objects depending on the context: server or client. In a client environment, most of the functions that operate on an entity collection or entity will require a more complex syntax in order to perform their AJAX calls to the server in an asynchronous.

Datasources

A datasource is an object unique to the client-side development and has no server-side counterpart. Datasources are objects that provide a layer of communication between the widgets on an Interface page and the underlying data those widgets are meant to interact with.

Datasources are often created for you automatically when you drag a datastore class onto a widget, for example. You can have multiple widgets on an Interface page all subscribed to the same datasource and when the data changes in one widget, all the others will automatically refresh with the updated values.

Datasources never contain data themselves. Wakanda Studio will usually try to name the datasource the same as the object holding the actual data. In the case of a datastore class, the name will be the same although the first letter is in lowercase, e.g. “employee” for the “Employee” datastore class.

Widgets

Widgets are also a client-side only construct. Widgets can range from simple to complex: from a Button to a detailed Grid. A widget takes normal HTML controls and provides you with a way to bind attributes from the datastore class to be displayed or on which to perform specific actions. When you bind a datastore class to a widget, Wakanda Studio will create a datasource (if a matching one does not already exist) and the widget will be bound to that datasource.

Because widgets are bound to datasources, when the data changes in one widget, any other widget using the same datasource will be updated as well. In addition, if data is changed in the datasource itself, all widgets will be subsequently updated.

Widgets often provide a number of built-in events for you to provide your own code.

Flattening the Difference Between Server and Client

Wakanda does its best to flatten out the distinction between server code and client code. The general concepts of the datastore closely match the data provider and are the same in both environments.

Both environments use an object named "ds" to perform their actions, thus reducing the learning curve despite some differences in syntax.

Likewise, the concepts of entity collections and entities remain the same despite a few differences in syntax.

Hands-on Example

Wakanda development is heavily dependent on widgets, which are complex interface objects that have been built into the product. Widgets such as Grids, Text Inputs, Radio Buttons, Checkboxes, Combo Boxes and Buttons are all available in Wakanda Studio to be added to your Interface pages. Once a widget is placed on your Interface page, simply drag a datasource to the widget to bind the two together, which then simply and elegantly handle all the complexities normally associated data-driven web solutions.

As soon as you bind a datasource to a Grid, you can immediately run the page and view the data in your datastore class. Add an Auto Form widget bound to the same datasource and you can now edit entities for the selected element in the Grid. Add some Buttons to the Interface page that are bound to the datasource and you instantly have next/previous/first/last/add capabilities, still without having written any code yet.

As a developer, you'll find yourself concentrating on the more enjoyable parts of writing web applications rather than the underpinnings of shuffling data back and forth from datastore to client. You'll be able to focus more clearly on the end-user experience, on *designing* rather than implementing, and on utilizing the modern capabilities of HTML, CSS3, and JavaScript to create solutions that will rival desktop applications.

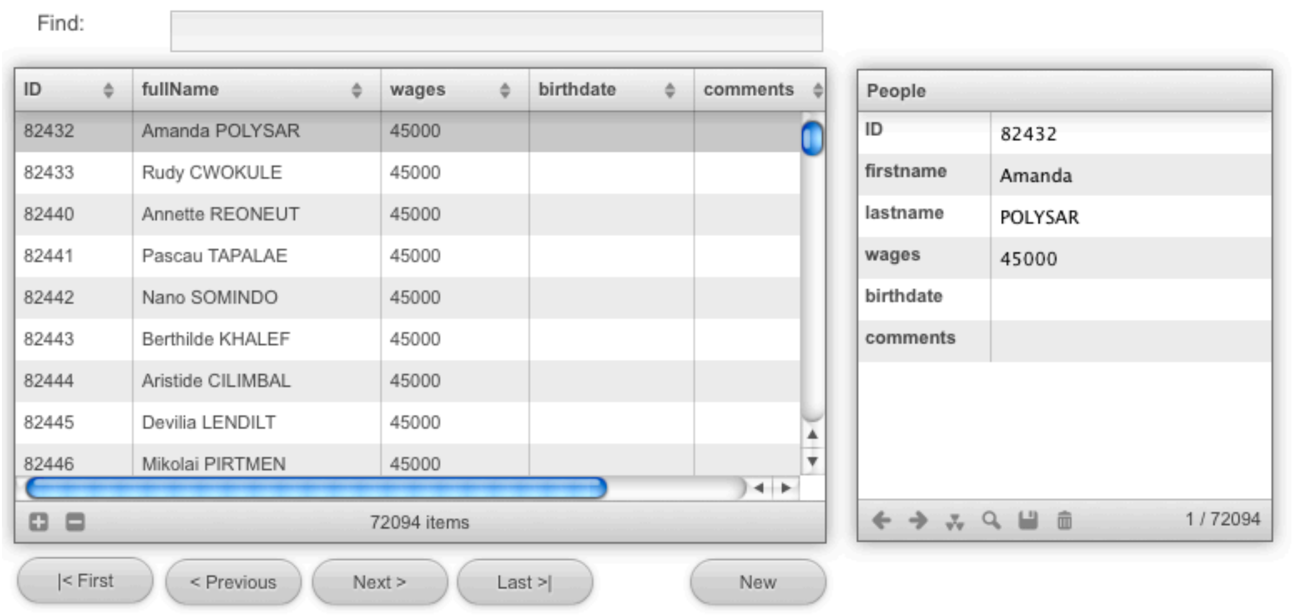
A Simple Example

If you've already gone through the *Quick Start* manual, you will know how to create the following:

1. Create a datastore class "Person" with the following attributes:
    a. firstName
    b. lastName
    c. wages
    d. birthdate
    e. comments
2. In an Interface page, create a Grid.
3. Drag and drop the "Person" datastore class on top of the Grid. The "person" datasource is created automatically.
4. Add an Auto Form to your Interface page and drag the "person" datasource on top of it.
5. Add five buttons to your Interface page and define the datasource to "person" and select the following actions:
    a. First

b. Previous
c. Next
d. Last
e. Create

Now that your Interface page is complete, let's run it. The end result should look something like this in your browser window:



Find:

ID	fullName	wages	birthdate	comments
82432	Amanda POLYSAR	45000		
82433	Rudy CWOKULE	45000		
82440	Annette REONEUT	45000		
82441	Pascau TAPALAE	45000		
82442	Nano SOMINDO	45000		
82443	Berthilde KHALEF	45000		
82444	Aristide CILIMBAL	45000		
82445	Devilia LENDILT	45000		
82446	Mikolai PIRTMEN	45000		

72094 items

People

ID	82432
firstname	Amanda
lastname	POLYSAR
wages	45000
birthdate	
comments	

1 / 72094

Click on the "New" button, which has the "Create" action. A blank entry in the Grid appears as well as an empty entry in the Auto Form in which you can add new entities.

Let's add one more thing to make this useful and to try out a bit of code. Add a Text Input widget, named "queryString".

In the "On Change" event, add the following code to query the datastore when the user changes the value in the "queryString" Text Input widget:

```
queryString.change = function queryString_change (event)
{
   var q = 'lastName="' + queryString + '@"';
   sources.people.query(q);
};
```

That was easy, but how does it work?

This might be a premature topic to cover and we don't mean to scare you off; but some people like to know where they're heading before they start the journey. Let's take a look at the HTML code generated by our Wakanda page and break down the pertinent parts. The source for our Interface page will look something like this:

```
<!DOCTYPE html><html>
    <head>
        <title>Simple Page</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <meta name="generator" content="Wakanda GUIDesigner"/>
        <meta http-equiv="X-UA-Compatible" content="IE=Edge"/>
       <meta name="build" content="102965"/>
       <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-
scale=10, user-scalable=1"/>
       <meta name="apple-mobile-web-app-capable" content="yes"/>
       <meta id="waf-interface-css" name="WAF.config.loadCSS"
content="styles/simplePage.css"/>
       <meta id="waf-palette" name="WAF.config.loadCSS"
content="/walib/WAF/widget/palette/css/widget-palette-default.css"/>
       <meta data-type="dataSource" data-lib="WAF" data-source="Person" data-source-
type="dataClass" data-dataType="string" data-autoLoad="true"
name="WAF.config.datasources" data-id="country" content="country"/>
<meta name="WAF.config.loadJS" id="waf-script" content="scripts/simplePage.js"/>
</head>
    <body id="waf-body" data-type="document" data-lib="WAF" data-platform="desktop"
data-rpc-activate="false" data-rpc-namespace="rpc" data-rpc-validation="false"
class="default" data-theme="default" style="visibility:hidden;">

>>> references to all of our widgets (removed for brevity)

       <script type="text/javascript" src="/waLib/WAF/Loader.js"></script>
   </body>
</html>
```

The first thing you may notice is the brevity of the underlying source. The Grid and Auto Form widgets are nothing but standard <div> tags with numerous custom attributes. In fact, all the objects on the Interface page are just standard HTML elements with custom attributes added to them.

In the <head> section of our form, there are a few <meta> tags worth mentioning. The "generator" meta tag identifies the page as a Wakanda document so Wakanda Studio knows to render it in the GUI Designer. There are additional <meta> tags for each of the datasources we created on our page. Additionally there is a <meta> tag reference to a CSS file that is associated with the Interface page.

The Wakanda "Loader"

So, how does all this come together as the complex objects, actions, and visual presentation experienced by the user? Near the bottom of the source of this Interface page, there is a reference to a WAF JavaScript file that is responsible for translating all these tags into functional objects at runtime.

The **Loader.js** file does many things. First, for each datasource on your page, it will construct a customized datasource object in memory. The object it builds for a datasource whose origin is a datastore class will have different methods and attributes from one whose origin is an array or variable. For a datastore class datasource, the Loader will create proxies for all the datastore class methods and it will create properties off the object to directly access the values in attributes of the currently selected entity.

The Loader then processes the page and instantiates all of the complex widget objects on the form. The creation of the widgets will cause a series of actions in which the widgets set up listeners for the various events each is interested in from its datasource. When the Loader is done, the user's web page will have been transformed into the useable interface you have created. The datasources will then begin firing off their respective events, beginning with making AJAX calls to retrieve any initial sets of data.

When all is complete, the Loader will have significantly modified the page's DOM. A "View Source" on the document will only show the sparse initial HTML, but the DOM in memory has been filled with numerous new objects. As a Wakanda developer you will need to get comfortable with working with and manipulating the DOM of your pages.

The Loader also pre-loads a number of useful frameworks for you to use: jQuery and jQuery UI. The jQuery framework is an indispensible part of the WAF, and is readily available for use in your code. The Loader handles the loading of the JavaScript and CSS files associated with your Interface page. The Loader makes a number of optimizations in the way all these files are loaded to reduce the number of file requests to the server and minimize the amound of data to be transferred.

Any CSS or JS files that you want loaded, should be specified in the Properties tab of your Interface page via the Wakanda Studio so that the Loader can take responsibility for adding them in the proper sequence. Because the Loader's actions are taking place **after** the initial page has been delivered to the client, you cannot count on the page objects being defined until the Loader has completed its tasks. Therefore, the following would be invalid:

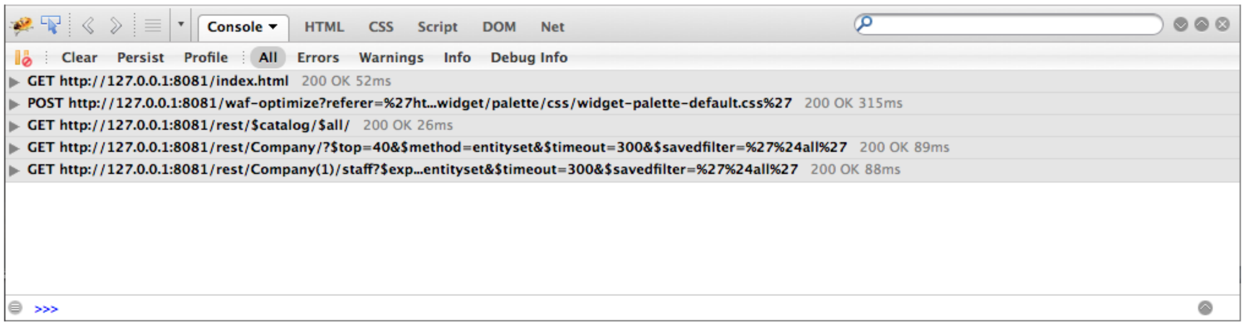
```
<head>
    <script>
        $(document).ready(function(){
            $("#wakandaobject").css("background-color","red");
        });
    </script>
</head>
```

Instead of placing your startup functions in the head of the document, there is an "On Load" event for the Wakanda page that will be run **after** the Loader has completed its initial setup.

Debugging

All modern browsers have some form of debugger that will let you explore the underlying DOM, put breakpoints in your JavaScript code, and evaluate the state of objects as you step line-by-line through code.

Using the JavaScript debugger in your browser allows you to find out what is happening as the code on your Interface page is executed.



```
GET http://127.0.0.1:8081/index.html   200 OK 52ms
POST http://127.0.0.1:8081/waf-optimize?referer=%27ht…widget/palette/css/widget-palette-default.css%27   200 OK 315ms
GET http://127.0.0.1:8081/rest/$catalog/$all/   200 OK 26ms
GET http://127.0.0.1:8081/rest/Company/?$top=40&$method=entityset&$timeout=300&$savedfilter=%27%24all%27   200 OK 89ms
GET http://127.0.0.1:8081/rest/Company(1)/staff?$exp…entityset&$timeout=300&$savedfilter=%27%24all%27   200 OK 88ms
```

Introduction to Datasources

The cornerstone of Wakanda client development is the datasource object. A **datasource** is a Wakanda object that manages information and acts as a dispatcher for events. Its main purpose is to provide data to widgets and to inform widgets when values have been modified.

A Wakanda datasource does not contain the data that it provides. Instead, it relies upon another construct to house the data. A Wakanda datasource can be based on five different data origins:

- datastore class,
- relation attribute,
- variable,
- array, or
- object.

When a datasource is based on a datastore class or relation attribute, it uses the data provider to interact with Wakanda Server. However, when a datasource is based on a variable, array, or object, its origin is a local JavaScript construct.

The datasource purpose is twofold: provide a number of conveniences to ease developing with Wakanda and to act as a dispatcher between the actual data-bound objects and the widgets on your Interface page. Multiple widgets can "subscribe" to the same datasource. When the content of the datasource has changed, it will notify all subscribing widgets so they may each take appropriate action.

Widgets depend on a datasource to provide them with the data they display. Datasources are either an entity collection (datastore class or relation attribute) or a browser-side item such as a variable, array, or object.

Datasources Based on Datastore Classes and Arrays

Conceptually, a datasource based on a datastore class or relation attribute is a mash-up combining the concept of an entity collection that expresses its currently selected element as an addressable entity.

Datasources based on arrays try to behave as similarly as possible to those based on a datastore class since both are representing multiple elements. Regardless of whether the origin of the datasource is an array, datastore class, or relation attribute, the role of the datasource is to keep track of the data and maintain a pointer to one of its items as the "current element." Additionally, the datasource promotes the attributes of the current element to the datasource's properties, allowing simple references such as:

```
var currentPersonName = sources.people.fullName;
```

The ability of a datasource to track both the collection of elements as well as the "current" one is a powerful convenience. This capability lets you place on your page a Grid widget displaying a selection of elements and an Auto Form widget with the details of the currently selected element. When both the Grid and Auto Form are bound to the same datasource, changing a value in the Auto Form will automatically update the value in the Grid. Add a Button widget to your Interface page bound to the same datasource with its automatic action set to "next" and each time the user clicks on it, the position of the currently selected element will advance by one and automatically refresh both the element displayed in the Grid and Auto Form.

Another type of datasource can be created from a 1->N relation attribute of a datasource you already have defined on the Interface page. This is called a **relation attribute datasource** and will expose an entity collection that represents the related entities of the current item of its parent datasource. This allows you to display “children” collection of entities in a second Grid as you step through the “parent” entities.

Datasources Based on Variables and Objects

Datasources based on variables or objects, on the other hand, are based on individual elements and simply reference the data contained in those objects.

Regardless of its type, a datasource does **NOT** contain data; it only points to the object containing the data. Consider the datasource to be a “wrapper” for your data and the wrapper has been extended with additional properties and methods to ease development and communication with your widgets. This is an important distinction to make and can be particularly confusing when the datasource’s origin is a JavaScript variable. For instance, if you create a datasource whose ID is “myVar” and whose origin is “myVar,” you will end up with two distinct items: “sources.myVar” representing the datasource object and “myVar” representing a JavaScript variable that stores the data. If you want to change the value of the JavaScript variable, you would need to do the following:

```
myVar = ‘new value’
sources.myVar.sync(); //advise the datasource of our change

//or since array-based datasources are JavaScript objects…
friends.push({Name:'Dan’, Age: 47, Gender:'Male', Birthdate:'09/09/1963'});
friends.push({Name:’Dave', Age: 50, Gender:'Male', Birthdate:'03/24/1960'});
friends.push({Name:'Melinda', Age: 41, Gender:'Female', Birthdate:''});
friends.push({Name:'Wendee', Age: 41, Gender:'Female', Birthdate:’’ });
sources.friends.sync(); // tell the datasource to update our widgets
```

Any time you *programmatically* change the values of JavaScript objects being referenced by a datasource, you must inform the datasource of the change by issuing a **sync()** command. This call will force the datasource to reload its data from the underlying source and notify all the widgets subscribed to it. You do not need to do this if the end-user changed the data from within the widget. Widgets typically handle advising their datasource of any changes made by the end user, which in turn will cause the datasource to update the original variable with the updated value.

By default, Wakanda Studio will often set the datasource ID to be the same as the underlying source object. However, the datasource ID can be renamed without any issues. If you are comfortable with the fact that “sources.myVar” is a different object than “myVar” then keep them the same. We’ve found that for datasources whose origin is a JavaScript variable/object that maintaining the same ID is often convenient. However, for datasources whose origin holds multiple elements (i.e., whose origin is a datastore class, relation attribute, or array), we often rename our datasource ID to be descriptive of how we intend to use it on our Intreface page.

So, for a datasource based on our “Person” datastore class we may give the ID “people” because this more clearly expresses the multiplicity of our data. There may also be cases in which you have multiple datasources defined for the same class, in which case you do need to come up with a unique naming convention. For example, you may want to have two Grids on your page in which your “Person” entities are divided between “friends” and “enemies”; each Grid can be based on a separate datasource, but share a common datastore class (though presumably different selection criteria).

Referencing Datasources

The WAF takes the set of datasources defined on an Interface page and attaches each one as an attribute of an object named "sources". As an example, a datasource whose ID is "people" will be referred to in code as *sources.people*.

Datasources can also be used apart from widgets when you want to interact with data directly.

When referring to the attributes of a datastore class datasource, you can refer to them as follows:

```
sources.people.firstName
```

Synchronous vs. Asynchronous Coding

In Wakanda, all client-based functions can be called either synchronously or asynchronously. On the other hand, server-based calls are synchronous by nature. Only when we consider executing code in a web-based client-environment does it become necessary to consider the impact to the user of executing code in a synchronous fashion.

In a web environment, synchronous calls to a backend database can be very disruptive to the user's experience. While a synchronous call is executing, all user-interaction with the Interface page is suspended until the operation concludes. If the user clicks a button, causing a lengthy search, the screen will appear frozen and nothing is redrawn until the results were returned.

The Wakanda Application Framework (WAF) provides an asynchronous way to execute every method that involves communication with the backend server. The synchronous mode of executing calls is still supported though discouraged in a production environment.

Wakanda determines if a method will execute asynchronously based on the signature of the parameters passed to the call. If a set of "Options" is provided to a function, it will execute asynchronously.

For some methods, there is no asynchronous support because the nature of the method does not require additional communication with the backend server. For instance, **getCurrentElement()**, **getPosition()**, and **isNewElement()** can each determine their return value from the data already loaded as part of the datasource.

Each command that supports asynchronous calling allows for two parameters in addition to others specific to the given function: "options" and "userData".

options

The *options* parameter allows for a set of values and/or functions to be provided to the called method. The exact list of options varies per method; however, they all support the passing of a JavaScript function to handle the "onSuccess" and "onError" events. Let's examine the simple command to save changes to the current entity in a datasource:

```
sources.people.save({onSuccess: function(event) {
    //the save was successful
    //include any subsequent code here
},
onError: function(event){
    //the entity did not save
    //you can get the error from event.message
    alert(event.message);
}});
```

In the case of "onSuccess" and "onError", a JavaScript function must be provided. The function passed will be called back when the given operation completes its task. In our example above, we included an anonymous function. Anonymous functions can be a useful shortcut if you have a modest amount of code to execute for the event. If you have more elaborate code or code that needs to be consolidated and used across a number of events, then you can pass named functions in it as well:

```
function handlePeopleSave {
       // do a bunch of complex stuff here
};

sources.people.save({onSuccess: handlePeopleSave,
onError: function(event){
   //the entity did not save
   //you can get the error from event.message
   alert(event.message);
}});
```

The "event" object that is returned to the function handling "onSuccess" or "onError" will contain a variety of attributes that are relative to the context. For instance, in the *onSuccess* of a query you will obtain your results from "event.entityCollection". Likewise, a call to ds.getEntity will see its successful results in **event.entity**.

userData

An additional parameter in most asynchronous calls is an object we'll refer to as *userData*. The userData parameter is optional and its exclusion will not affect whether the command is executed asynchronously. You can pass any JavaScript object type as a valid parameter: from a simple string to a complex array of objects.

The userData parameter provides a means for you to pass information on to the "onSuccess" or "onError" handler methods. Why would you need to do this? Well, keep in mind that your command is executing asynchronously and the user has been returned control to continue interacting with your Interface page. Suppose that the user move ahead and changes the currently selected entity in your datasource before the "onSuccess" event of your save operation is called. If your onSuccess method needs to refer to anything relative to the context at the time the save was invoked, you would need to provide that information (or a way to access that information) via the userData parameter.

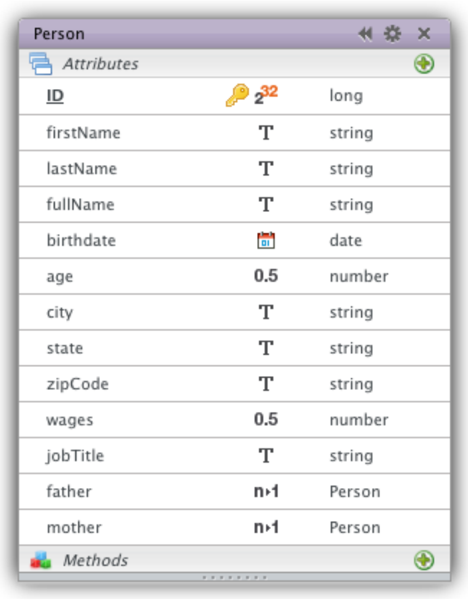
The data you pass in userData will be available inside the callback function through the **event.userData** object afterwards.

```
// theAge, theWage and theState are JavaScript variables
ds.Person.query("age > :1 and wages > :2 and state =
:3",{params:[theAge,theWage,theState]}, userData);
```

Querying

Wakanda projects are data-driven solutions. As such, one of the more common tasks you will perform is searching your data and presenting the results to your users. In this section, we are going to explore the various search-related functions and the scope of entities that each acts upon.

Most of the examples in this section are based on the following datastore class named Person. This class has two relationships to itself through the “father” and “mother” relation attributes that will help us highlight how related entities are loaded.



Person

Attributes

ID	2^{32}	long
firstName	T	string
lastName	T	string
fullName	T	string
birthdate		date
age	0.5	number
city	T	string
state	T	string
zipCode	T	string
wages	0.5	number
jobTitle	T	string
father	n›1	Person
mother	n›1	Person

Methods

The primary function used for searching is **query()**. The query() function can be called on those objects referencing a collection of entities and is usually applied to the datastore class. Queries can be performed at the data provider level, on either datasources or entity collections. For instance, the following example will search at the data provider level (all entities in the class) and return a collection of Person entities matching our criteria.

```
ds.Person.query("lastName='Smith'");
```

If you come from an SQL background, you will find querying and working with the return results a bit different in Wakanda. Queries in Wakanda are quite powerful, allowing you to include criteria across many relation attributes in datastore classes. However, in Wakanda all those relationships are defined at the datastore class level. Where SQL uses ad hoc joins between classes in a query, Wakanda depends on declared relationships. This can hardly be viewed as a limitation when the Wakanda search language so easily enables you to implement complex joins with little effort. For instance, the following line of code will find all the people whose grandfather is named “Joe.”

```
ds.Person.query("father.father.firstName='Joe'")
```

In an SQL query you express your SELECT list along with your search criteria and the result returned to you is a set of data that has been decoupled from the underlying entities. A Wakanda query, on the other hand, always returns a collection of entities, representing all the attributes of the underlying datastore class. When that entity collection is coupled with a datasource on an

Interface page, your users are able to freely navigate the entity collection with access to all the attributes you have exposed to them.

The "Wakanda Server-Side Concepts" manual covers many aspects of querying the datastore on the backend and most of these concepts also apply to querying in the WAF. Instead, in this manual we will focus on the ways in which querying behaves differently on the client.

There are a number of factors in executing code on the client that necessitate different behavior than the same code executed server-side. For instance, a query executed on the server that returns thousands of results is easily manageable because you are working directly with server-based objects. However, the same search from a client-environment suffers the extra burden of transmitting those results from server to client.

When a web client (via the WAF) executes a query, the entire entity collection is generated on the server but by default only the first 40 entities (or the value defined in the datastore class's **Default Top Size** property) are transmitted to the client. While this number can be customized with each query, the goal is to quickly return data to present to the user. You might think that this would create great hardship for the developer in tracking which collections of entities have been retrieved, but it does not. The developer can simply code the application as though all the entities have been loaded.

When the code performs any operation on an entity not yet received by the client, the WAF will execute a new REST call and obtain the next set of entities containing the entity needed. A useful way to see this behavior in action is to watch the Console in Firebug as you page through entities in a Grid. As you scroll through the Grid, you will see subsequent REST calls executed to retrieve additional collections of entities. If you drag the scrollbar down to the bottom of your list in a large collection of entities, you will see that only a single REST call is made to pull the data for those entities in your display, not the data for all the entities in between.

Once received by the WAF, the entities are cached on the client.

Data-retrieval optimization is not the only means by which Wakanda addresses the needs of client-executed code. Like most commands in the WAF, queries can be performed asynchronously so that control is returned immediately to the user while the server is executing and sending the results. The following line will execute a query synchronously and the end user will not be able to perform any action while the query executes:

```
var myCollection = ds.Person.query("lastName = 'S@'");
```

If this code were executed as part of a button script, the user would find their screen frozen until the results were returned. Using the standard methods discussed earlier for executing code asynchronously, we can perform this search and give immediate control back to the user:

```
ds.Person.query("firstName='@a@'", { onSuccess:function(event) {
    var myCollection = event.entityCollection;
});
```

This method of execution greatly enhances the user experience in a web environment. When the asynchronous calls are combined with the optimized data-retrieval built into the WAF, the developer is able to build solutions that are highly responsive to user interaction.

As previously mentioned, queries can be performed on the data provider, on datasources, and on entityCollection objects. The behavior of each type of query differs slightly.

Querying on the Data Provider

In the "Wakanda Server-Side Concepts" manual, you were introduced to the datastore (referenced as **ds**) along with all its methods and attributes. The WAF has a proxy of the datastore called the Data Provider, which is also referenced using "ds". One of the services that the data provider offers in class-scope functions is querying. It should be noted that the "ds" reference on the server is a completely different object than the "ds" in the WAF. The WAF's version of "ds" is limited to those operations necessary from a client-development viewpoint and as such is a subset of the functionality found on the server. Additionally, same-named functions in the "ds" may have different and more complex parameters in the WAF to account for things like asynchronous execution.

In the WAF, a query can be performed on the data provider much the same way it is executed on the server, with the return result being an entity collection. Queries on the data provider assume all entities of the datastore class are the basis for the search: *ds.Person.query(searchCriteria)* is equivalent to *ds.Person.all().query(searchCriteria)*.

When executed asynchronously, the resulting entity collection can be obtained in the **event.entityCollection** attribute as follows:

```
ds.Person.query("lastName='Smith'", { onSuccess:function(event) {
    var myCollection = event.entityCollection;
    sources.people.setEntityCollection(myCollection);
});
```

The above example assumes that the Interface page has a datasource named "people". A query of this type is executed on all the entities in the datastore class. In the code above, the results are then assigned to the datasource's entityCollection (which in turn would propagate the changes across all widgets subscribed to the same datasource).

Querying on an entityCollection Object

A query performed directly on a pre-existing entityCollection object will apply its criteria only to those entities already in the entity collection. In the following example, let's assume the Person datastore class contains 1,000 entities, sequentially numbered:

```
var myCollection = ds.Person.query('ID<=100'); //searches the datstore class
var myResult = myCollection.query('ID>95'); //searches only within the collection
```

The resulting entity collection in myResult will only contain those five entities that formed the intersection of the two queries. Just keep in mind that queries based on entity collections behave like a subquery.

A query based on an entity collection fully supports the asynchronous method as follows:

```
//get a handle to the datasource's entityCollection
var currentSet = sources.people.getEntityCollection();

//perform a query only against those items in our current collection
currentSet.query("age>40",{onSuccess: function(event){
    //assign the results back into our datasource
    sources.people.setEntityCollection(event.entityCollection);
},
onError: function(event){
    //handle any error condition here
}});
```

Querying on a Datasource

The most common way you are likely to interact with data on your Interface pages is through datasources. Since all the widgets on your Interface page are subscribed to a datasource for their information, operating directly with the datasource is the most expedient way to perform searches.

Datasources have a lot of conveniences built-in for the developer and querying is no exception. When a query is performed on a datasource object, the WAF assumes that the results should replace the datasource's current entity collection. Therefore, rather than calling it as a function that returns an entity collection, it can be called as a method:

```
sources.people.query('firstName="John"');
```

This synchronous example finds all the people named "John" and replaces the datasource's current entity collection with the new values. If you have a Grid whose datasource is "people", the entity collection will be automatically updated.

The scope of a query performed on a datasource is all the entities in the datastore class. This may seem counterintuitive since the user is often viewing a subset of entities within datasource-bound objects like Grids. However, with a datasource query performing much like a dataprovider (ds) query, the developer is given the easiest shortcut for the most common type of action.

Note that when Wakanda creates a datasource for a datastore class, it names the datasource the same as the class (transformed in lower camelcase, e.g., the "LineItems" datastore class creates a datasource named "lineItems"). You can, of course, modify the datasource name to better describe its purpose if you'd like.

Datasources have a function named **filterQuery()** that can be used to perform a subquery on the current entity collection:

```
sources.people.filterQuery('age>40');
sources.people.filterQuery('age>40',{onSuccess:function(event){
   //do something here
}});
```

Queries on a datasource can be executed asynchronously as well. Since the assignment of the results to the current entity collection is handled automatically, you might find that your *onSuccess* call has nothing to do:

```
sources.people.query('firstName="John"', {onSuccess:function(){}});
```

It is not necessary to actually do anything in the *onSuccess* function since its mere presence as part of how you execute the statement will set the function to execute asynchronously.

Sorting in a Query

The syntax of the query string you provide to the **query()** function supports some additional capabilities in how you express your request. You can also specify an **order by** clause as part of your query:

```
var myCollection = ds.Person.query('lastName = "S@" and age < 50 Order by age desc');
```

The benefit of expressing your sort criteria as part of your query is the optimization you achieve by having the server perform both operations before any data is returned to the client. The above statement produces the same results as:

```
var myCollection = ds.Person.query('lastName = "S@" and age<50');
myCollection.orderBy('age desc');
```

However, in the second example, the server would have initially sent the client the first set (40 or as defined by the datastore class's **Default Top Size** property) of entities of the resulting query. When the subsequent **orderBy()** function is called, the server will order the entities and transmit the first set of entities in the resulting entity collection again.

Extra Options

When a query is executed asynchronously, there are a number of additional options that may be included as part of the syntax. The basic format of the **query()** command is as follows:

```
query(queryString, {options}, userData)
```

As you've seen in previous examples, the "options" common to most asynchronous calls are:

```
onSuccess: function(event){/*handle onSuccess*/},
onError: function(event){/*handle onError*/}
```

Additional options are:

```
params: [x,y,z],
autoExpand: String,
queryPlan: Boolean,
pageSize: Number (defaults to 40 unless otherwise defined),
progressBar: ProgressBarID
```

***Note**: For more information regarding these options, refer to Defining Queries (Client-side) in the Client-Side API's Dataprovider chapter.*

Params

Wakanda allows for up to nine placeholder values to be inserted into your query string. Too often developers need to build query strings by concatenating backend entity references with user-supplied data. By using placeholders, you can write more generic code with fewer errors. Assuming you have a JavaScript variable named *findName*, the following query can be applied:

```
var myCollection = ds.Person.query('lastName= :1',{params:[findName]}, userData);
```

Wakanda will provide all the proper escaping of characters like quotes within the parameters.

The placeholder syntax is optional except for the specific case when you want to use an entity as your search criteria. As an example, suppose we want to find all the children of the currently selected Person in our datasource:

```
var theFather = sources.people.getCurrentElement();
sources.people.query("father = :1", {params: [theFather],
onSuccess:function(event){
   //handle results
},onError: function(event){

}});
```

In this example we've taken an entire Person entity as assigned to the *theFather* variable and used the JavaScript object itself as our placeholder value.

autoExpand

When a query is performed in Wakanda, an entity collection is returned with all the attributes of the datastore class. However, only those attributes that can return scalar values actually have data returned. In our People class, all the attributes will have data values except for the "father" and "mother" relation attributes. Rather than pull the data from the corresponding related entities, Wakanda will instead return an object that indicates that the attribute is "deferred". Only if a subsequent request tries to access the related entity will Wakanda issue the additional REST call to obtain the data.

At the time you execute a query, if you know you will need access to the related entities, you can use the *autoExpand* option. AutoExpand will force Wakanda to pre-load and provide data for the related entities specified. In the following example, we will apply the autoExpand option on the "father" attribute, but not on the "mother" attribute. The "mother" entity will need to be manually loaded:

```
sources.people.query('ID<100',{autoExpand: 'father'});

//The datasource now has an entity collection with an autoExpanded father

//When the current entity of the datasource changes, let's display
//the mother and father

PeopleEvent.currentEntity = function (event)
{
      var html = '' ;
      html += sources.people.fullName + ' is the son of
'+sources.people.getAttributeValue('father.fullName')+ '<br/>';
   sources.people.mother.load({onSuccess:function(event)
   {
      var motherEntity = event.entity;
      html+= 'this mother is '+motherEntity.fullName.getValue();
      $('#display').html(html);
   } });

};
```

AutoExpand is not necessary for alias attributes in your class. These attributes are obtained by the server and treated as scalar values in the data returned by the query. If you find yourself frequently using the same relation attribute, you might consider creating an alias attribute.

It is not uncommon to need the same attribute auto-expanded every time a search is performed. There is a special function available on datasources allowing you to indicate which related datastore classes you always want to auto-expand. The **declareDependencies()**function allows you to specify one or more datastore classes to auto-expand in any subsequent query. This function is typically called in the onLoad event for the page:

```
sources.people.declareDependencies("father","mother");
```

When a widget, such as a Grid, is configured to display data from a datasource's related entity, it will automatically declare a dependency for the source to assure that the data is always available. For example, suppose your “people” datasource is bound to a Grid and configured to display fullName, age, and father.fullName. The Grid widget will see that you intend to include the father.fullName value and will issue a declareDependencies(‘father’) for the datasource on your behalf.

If you know the specific attribute in the related datastore class that you need expanded, you can specify it directly for better server optimization:

```
sources.people.declareDependencies("father.fullName","mother");
```

queryPlan

If the **queryPlan** option is set to true, the query returns its queryPlan along with the resulting entityCollection object. The queryPlan will be returned as an attribute of the entityCollection object (i.e., event.entityCollection.queryPlan).

pageSize

The **pageSize** option provides the developer with a way to change the default number of entities to batch send to the client for each REST call. By default, this value is set to 40 unless you have specified another value in the datastore class’s **Default Top Size** property.

progressBar

If you expect your queries to take a large amount of time to execute, you may want to include a Progress Bar widget on your Interface page. Once you have added this widget to your Interface page, you can pass pass the its ID as an optional parameter to the **query()** command:

```
sources.people.query('firstName = "S@"',{progressBar: myProgressBarID});
```

The server will establish a unique progress indicator specific to the current user’s session and the Progress Bar on the Interface page will automatically start its periodic polling of the server to update its value. This whole process functions asynchronously so the user is able to perform other operations on the page while the search continues to be executed.