Web Workers

Wakanda Server uses Web workers to provide you with JavaScript multithreading capabilities. A Web worker is created by referencing an individual JavaScript file. When the worker is instantiated, it then becomes an object residing in memory waiting to be called.

On the server side, Wakanda fully implements the Web Workers W3C specifications, originally designed for the client side, except that:

- the **wait( )** Wakanda-specific method has been implemented to allow a parent thread to stay alive after it is executed.
- the *WorkerUtils API* is partially implemented:
  - WindowTimers are supported, see **setTimeout( )** and **setInterval( )**).
  - the **importScripts( )** API is also implemented.

There are two types of Web workers:

- **Dedicated workers** have one "parent" and can only communicate with their parent or the workers they spawn. They are always part of an execution tree. Dedicated Web Worker instances are created with the **Worker( )** constructor method. Dedicated workers are described in the **Worker Instances** section.
- **Shared workers** can be shared among several callers. They don't have a specific "parent". They are uniquely identified by the name of their JavaScript files and a given name. Shared Web Worker instances are created with the **SharedWorker( )** constructor method. Shared workers are described in the **SharedWorker Instances** section.

For a detailed overview of Web workers, please refer to the **Dedicated Workers and Shared Workers** section in the "Wakanda Server-Side Concepts" manual.

Web Worker Constructors

SharedWorker( )

void **SharedWorker**(String *scriptPath* [, String *workerName*])

Parameter	Type	Description
scriptPath	String	Pathname to JavaScript file
workerName	String	Name of the worker to execute

Description

The **SharedWorker( )** method is the constructor of the *SharedWorker* type class objects. It allows you to create new **SharedWorker Instances** on the server.

Shared workers are Web workers that can be addressed from any thread, while dedicated workers are Web workers that can only be addressed from the parent thread that created them. Dedicated workers end when the parent thread ends, while shared workers continue to exist even if the thread that spawned them ends. For more information, refer to the **Worker Instances** class description.

Shared workers are uniquely identified by their script file names and a given name. The constructor will spawn a new shared worker thread if it does not exist yet.

In the *scriptPath* parameter, pass a path to a project-specific JavaScript file. If you pass the file with a relative path, Wakanda assumes that the project folder is the default folder. The referenced file must have valid statements that result in a worker.

Note: *If the worker's JavaScript file has any code outside of all its function declarations, Wakanda considers it as initialization code for the worker and executes it when the worker is created.*

In *workerName*, pass the name of the shared worker you want to create (if you omit the *workerName* parameter, the shared worker will be created with an empty string as its name). This shared worker name will be used to reference the shared worker for all the threads. When other threads want to interact with an already existing shared worker, they do so by executing the same code as if they are creating it, but instead receive a reference to this existing shared worker.

Example

Download the example solution

This shared worker creates an entity every second for 5 seconds, and sends info to the log. Here is the launcher function:

```
function doTestSharedWorker()
{
    var theWorker = new SharedWorker("SendRequestsWorker.js", "SendRequests");
    var thePort = theWorker.port; // MessagePort
    thePort.onmessage = function(evt)
    {
        var message = evt.data;
        switch(message.type)
            {
                case 'error':
                    debugger;
                    break;
            }
      }
     wait(); //waits for new messages in onmessage
}
doTestSharedWorker();
```

Here is the code of the SendRequestsWorker.js file:

```
function doSendRequests()
{
      count++;
      console.log('Count: ' + count);

      var theDate = new Date();
      if((theDate - startDate) < theDuration) {
            console.log('creating');
            var z = new ds.Util({
                              testValue    : count,
                              dateValue     : theDate
                        });
            z.save();
            console.log('' + ds.Util.length);
      } else {
            console.log('closing');
            close();
      }
}

onconnect = function(msg)
{
    var thePort = msg.ports[0];
    console.log('In onconnect');
    thePort.postMessage("OK");
}
console.log('Start of test...');
```

```
var count = 0;
var startDate = new Date();
var theDuration = 5000;

setInterval(doSendRequests, 1000) //Run every second
```

Example

Here is a basic example of creating a shared worker: the purpose of this datastore class method is to respond to a browser-side request for information on the status of the "TaskMgr" shared worker.

```
getTaskManagerStatus:function()
{
    var tmRef = 0;
    var tmInfo = {taskCount:0, errorCode:0};
    var taskMgr = new SharedWorker('WorkersFolder/TaskMgr.js', 'TaskMgr');
    var thePort = taskMgr.port; //MessagePort
    thePort.onmessage = function(event)
    {
        var message = event.data;
        switch (message.type)
        {
            case 'connected':
                tmRef = message.ref;
                thePort.postMessage({type: 'report', ref: tmRef});
                break;

            case 'update':
                tmInfo.taskCount = message.count;
                thePort.postMessage({type: 'disconnect', ref: tmRef});
                return tmInfo;
                close();
                break;

            case 'error':
                tmInfo.errorCode = message.errorCode;
                return tmInfo;
                close();
                break;
        }
    }
    wait();//waits until a call to close() in this thread
    //allows to handle incoming messages on the onmessage
    //at this point, this thread is about to end but the shared
    //worker continues on
}
```

The corresponding "TaskMgr" worker might be something like this:

```
function doSomeWork()
{
    try {
        // do something
        tmCount += 1;
    }
    catch(e){
        tmError = 1;
    }
}

onconnect = function(msg) // called when a new SharedWorker is created
{
    var thePort = msg.ports[0];
    tmKey += 1;
    tmConnections[tmKey] = thePort;
    thePort.onmessage = function(event)
    {
        var message = event.data;
        var fromPort = tmConnections[message.ref];
        switch (message.type)
        {
            case 'report':
                if (tmError!= 0)
                {
                    fromPort.postMessage({type: 'error', errorCode: tmError });
                    close();
                }
                else
                {
                    fromPort.postMessage({type: 'update', count: tmCount});
                }
                break;
```

```
            case 'disconnect':
                tmConnections[message.ref] = null;
                break;
        }
    }
    thePort.postMessage({type: 'connected', ref: tmKey});
}

var tmCount = 0;
var tmKey = 0;
var tmError = 0;
var tmConnections = [];
setInterval(doSomeWork, 1000) //Run every second
```

Worker( )

void **Worker**(String *scriptPath*)

Parameter	Type	Description
scriptPath	String	Pathname to JavaScript file

Description

The **Worker( )** method is the constructor of the class objects of the dedicated *Worker* type. It allows you to create new **Worker Instances** on the server. The proxy object allows the parent to exchange data with a dedicated worker.

Dedicated workers are Web workers that can only be addressed from the parent thread that created them, while Shared workers are Web workers that can be addressed from any thread. Dedicated workers end when the parent thread ends, while shared workers continue to exist even if the thread that spawned them ends. For more information, refer to the **Worker Instances** class description.

In the *scriptPath* parameter, pass a path to a project-specific JavaScript file. If you pass the file with a relative path, Wakanda assumes that the project folder is the default folder. The referenced file must have valid statements that result in a worker.

Note: *If the worker's JavaScript file has any code outside of all its function declarations, Wakanda considers it as initialization code for the worker and executes it when the worker is created.*

Example

The following is a simple example of a parent and child exchanging messages. Below is the **parent.js** script:

```
// A dedicated web worker is created by calling the Worker constructor
// with the name of the JavaScript file to execute (located in the default folder).
// This will return a proxy object (worker) so that it will be possible to communicate with the child.
var worker = new Worker('child.js');

// Define the message callback that will be triggered each time the child sends a message.
var state = 0;
worker.onmessage = function (event)
{
    if (state == 0) {
            // Child has received our initial message and this is its reply.
        console.log(event.data);    // "Child started".
            // Send a message to request termination.
        worker.postMessage('Please quit.');
            // Go back to idle, waiting for reply from child.
        state = 1;

    } else {    // state == 1
            // Child has terminated.
        console.log(event.data);    // Child finished.
            // We can terminate by calling close(), which will exit the wait().
        close();
    }
}

// Send a message to the child to trigger message exchange.
worker.postMessage("Go ahead.");

// Asynchronous execution
wait();

// After close() is called in callback, we are done.
console.log('Parent has terminated.');
```

Here is the **child.js** script:

```
// Child execution is asynchronous.
// onmessage is a global attribute containing the callback to trigger each time a message is received.
var state = 0;

onmessage = function (event) {
    if (state == 0) {
            // Waiting for a message from parent, just received it
```

```
        console.log(event.data);    // "Go ahead"
            // Reply to parent. Note that postMessage() is a global method. It will send a message to the parent's
            // worker proxy object onmessage attribute.
        postMessage("Child started");

            // Go back to idle, waiting for next message.
        state = 1;

    } else {    // state == 1
            // Waiting for a message from parent to terminate, just received it.
        console.log(event.data);    // "Please quit".
            // Sends a message back to parent, we're done.
        postMessage("Child finished");
            // Terminate.
        close();
    }
}
```

Note that there is no call to **wait( )**. By default, the child will wait until the end of the script, and service asynchronous callbacks.
In this example, the JavaScript code does nothing except for defining a callback. Everything is done by the callback.

SharedWorker Instances

Constructor: SharedWorker( )

Web workers instantiated with the **SharedWorker( )** constructor are **Shared workers**. The *SharedWorker* proxy object is defined in the parent thread; it allows worker threads to communicate.

Shared workers can be shared among several callers. They don't have a specific "parent". They are uniquely identified by the name of their JavaScript files and a given name. There is no "child" level for shared Web workers because they don't have a specific "parent". Once created, a shared worker can be shared among several callers. There is no special link between the shared worker and the thread that created it.

For a detailed example of how to use a shared Web worker, please refer to the **SharedWorker( )** method description.

ports

Description

The **ports** property contains the **MessagePort** object of the *SharedWorker* in its element 0. This object contains the tools for the *SharedWorker* proxies to communicate with the shared Web workers. When creating or connecting to a shared worker through the *SharedWorker* object, the whole **ports** object will be sent to the shared worker.

Here are the contents of the **ports[0]** object:

```
{
    attribute onmessage;      // Callback to trigger when a message is received
    attribute onerror;            // Callback function to trigger when an error is received
    void postMessage(in messageData);    // Method to send a message to a worker
};
```

For more information about these members, refer to the corresponding sections in the **Worker Instances** chapter:

- **onmessage**
- **onerror**
- **postMessage( )**

Two additional properties are available in the **ports[0]** object for *SharedWorker* objects used in the context of server-side WebSocket "proxy" object:

- **binaryType**: This property defines the type of data sent by the client.
- **onclose**: This property defines the function to call when the client socket is closed.

These properties are documented in the **Worker Instances** chapter.

onconnect

Description

The **onconnect** property contains the function to call when a thread creates a new *SharedWorker* proxy object to connect to the current shared worker.

The function defined will receive a single object as a parameter, which contains a copy of the **ports** object created in the "parent" *SharedWorker*. This object will have a property named *ports*, which is an array of message ports. *ports[0]* is the message port to use for communication. You can access the **ports** properties through *ports[0]*.

For example, if you declare "event" as a parameter to the **onconnect** function, you must first access the ***event.port[0]*** property to access the data posted in the **ports** property of the *SharedWorker* proxy object:

```
onconnect = function (msg)
{
    var msgPort = msg.ports[0];   // access the communication port
    msgPort.onmessage = function(event)   // access the port (MessagePort) data
    {
        var message = event.data;
        ...
    }
}
```

close( )

void **close**()

Description

The **close( )** method ends the thread from which it is called.

This method can be called:

- From a *Worker* or a *SharedWorker* parent thread where only the parent thread is closed.
  If you want to close a dedicated child worker from the parent thread, you can call the **terminate( )** method. If you call **close( )** on a waiting parent thread, all the dedicated workers spawned from that thread will receive a message to terminate (their internal "close" flag is set to **true**). If **close( )** is called during a callback in a **wait( )**, this will exit the **wait( )**.
- From a child thread.
  In this case, the internal "close" flag is set to **true**. The **wait( )** event loop is exited and the thread is closed.

The **close( )** method effect is not immediate: the JavaScript interpreter will continue until the current execution (exiting the current callback) is finished. All resources will then be freed up.

wait( )

Boolean **wait**([Number *timeout*])

Parameter	Type	Description
timeout	Number	Timeout in milliseconds
Returns	Boolean	True if the worker is terminated; False otherwise

Description

The **wait( )** method allows a thread to handle events and to continue to exist after the complete code executes.

In the context of a Web worker, the **wait( )** method allows a parent Web worker thread to handle child worker events. Since the parent-child worker communication is asynchronous (based on callbacks), this method is necessary in the parent script to allow the thread to keep from terminating after the code execution and to listen for callbacks. During the waiting time, asynchronous callback events from Web workers are handled. When this method has been called, the thread stays alive until you call **close( )**.

***Note:** The **wait( )** method is also available for child workers although it is usually not necessary in this context. Child worker scripts always implicitly call the wait mechanism.*

The **wait( )** method can also be used in the context of the main thread to allow asynchronous communication, for example when using or **System Workers**. In this context, to stop the **wait( )** loop, you need to use **exitWait( )**.

Note that while executing, the **wait( )** method blocks the thread but still handles callbacks.

If you specify a value (in milliseconds) in the optional ***timeout*** parameter, **wait( )** will run only during the time specified and then give the control back after this time, returning *false* if the worker is not terminated.

Example

Download the example solution

This shared worker creates an entity every second for 5 seconds, and sends info to the log. Here is the launcher function:

```
function doTestSharedWorker()
{
    var theWorker = new SharedWorker("SendRequestsWorker.js", "SendRequests");
    var thePort = theWorker.port; // MessagePort
    thePort.onmessage = function(evt)
    {
        var message = evt.data;
        switch(message.type)
            {
                case 'error':
                    debugger;
                    break;
            }
      }
     wait(); //waits for new messages in onmessage
}
doTestSharedWorker();
```

Here is the code of the SendRequestsWorker.js file:

```
function doSendRequests()
{
      count++;
      console.log('Count: ' + count);

      var theDate = new Date();
      if((theDate - startDate) < theDuration) {
            console.log('creating');
            var z = new ds.Util({
                              testValue    : count,
                              dateValue     : theDate
                        });
            z.save();
            console.log('' + ds.Util.length);
      } else {
            console.log('closing');
            close();
      }
}

onconnect = function(msg)
{
    var thePort = msg.ports[0];
    console.log('In onconnect');
    thePort.postMessage("OK");
}
console.log('Start of test...');

var count = 0;
var startDate = new Date();
var theDuration = 5000;

setInterval(doSendRequests, 1000) //Run every second
```

Worker Instances

Constructor: Worker( )

Web workers instantiated with the **Worker( )** constructor are **dedicated workers**. They have one "parent" and can only communicate with their parent or the workers they spawn. They are always part of an execution tree. The *Worker* proxy object is defined in the parent thread; it allows parent and child threads to communicate.

The child level is the JavaScript thread where the Web worker script is running.

For a detailed example of how to use a dedicated Web worker, please refer to the **Worker( )** method description.

onmessage

Description

The **onmessage** property contains the function to call when a message is received.

This property is available:

- in a parent *Worker* or *SharedWorker* (through the **ports** property) object: the function will be called when the child thread calls the parent thread.
- in the child Web worker thread (global application level): the function will be called when the parent *Worke*r or *SharedWorker* calls the child worker.

The function defined will receive a single object as a parameter. This object will have a property named *data*, which will contain a copy of any object passed by the child or parent **postMessage( )** call. For example, if you pass "event" as a parameter to the **onmessage** function in the parent process, the *event.data* property will receive any object posted as a parameter by the child **postMessage( )** callback.

Note: *When modifying the* ***onmessage*** *attribute, you must be careful not to have a race condition because there is no synchronization mechanism.*

Example

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

```
var myWorker = new Worker('WorkersFolder/dedicatedWorker.js');
var myWorker.onmessage = function(event)
{
    var message = event.data;
    if (message.type == 'stopped')
    {
        close();
    }
}
myWorker.postMessage({type: 'process'});
wait();
```

The contents of the *dedicatedWorker.js* file is shown below:

```
function doSomeWork()
{
    // do some work, such as writing files, sending e-mails, etc.
}

function onmessage(event)
{
    var message = event.data;
    if (message.type == 'process')
    {
        doSomeWork();
        postMessage({type: 'stopped'});
        close();
        break;
    }
}
```

onerror

Description

The **onerror** property contains the function to call when an uncaught runtime script error occurs in one of the *Worker*'s scripts.

This function will receive a single object, which has the following three attributes, as a parameter:

- *message*: represents the error message.
- *filename*: represents the absolute URL of the script where the error originally occurred.
- *lineno*: represents the line number where the error occurred in the script.

Note: *When modifying the* ***onerror*** *attribute, you must be careful not to have a race condition because there is no synchronization mechanism.*

binaryType

Description

Preliminary Note: *This property is only available in the context of a worker used to handle the "proxy" server object of a client WebSocket. For more information, please refer to the paragraph.*

This property is available:

- at the global application level in a dedicated *Worker*
- through the **ports** property object of a *SharedWorker*

The **binaryType** property can be used to define the type of the *message.data* property of a WebSocket "proxy" object corresponding to the message actually received from the client. *data* is a property of the *message* parameter passed to the **onmessage** callback function.

The following values are accepted for the **binaryType** property:

Value	Type description
"buffer" (default)	See **Buffer** in the Wakanda documentation
"blob"	See **BLOB** in the Wakanda documentation
"string"	See String on MDN

For small data volume, the "string" **binaryType** is the easiest to use. For large data volume, the "buffer" type is recommended. By default, if you do not set the **binaryType** property, the "buffer" data type is used.

onclose

Description

Preliminary Note: *This property is only available in the context of a worker used to handle the "proxy" server object of a client WebSocket. For more information, please refer to the paragraph.*

The **onclose** property contains the function to call when the client WebSocket handled by the worker has just been closed.

The function defined will be called without any parameters. It can be used to clean up variables or to close associated workers on the server side.

close( )

void **close**()

Description

The **close( )** method ends the thread from which it is called.

This method can be called:

- From a *Worker* or a *SharedWorker* parent thread where only the parent thread is closed.
  If you want to close a dedicated child worker from the parent thread, you can call the **terminate( )** method. If you call **close( )** on a waiting parent thread, all the dedicated workers spawned from that thread will receive a message to terminate (their internal "close" flag is set to **true**). If **close( )** is called during a callback in a **wait( )**, this will exit the **wait( )**.
- From a child thread.
  In this case, the internal "close" flag is set to **true**. The **wait( )** event loop is exited and the thread is closed.

The **close( )** method effect is not immediate: the JavaScript interpreter will continue until the current execution (exiting the current callback) is finished. All resources will then be freed up.

postMessage( )

void **postMessage**(Mixed *messageData*)

Parameter	Type	Description
messageData	Mixed	Message to send to the dedicated worker

Description

The **postMessage( )** method allows you to exchange data between a parent *Worker* proxy and a dedicated Web worker.

This method is available:

- in a parent *Worker* object: a **postMessage( )** call from the parent *Worker* will be received as an object in the [#cmd id="103006"/] function of the worker script (in the *.data* property).
- in the child Web worker thread (global application object): a **postMessage( )** call from the child script will be received as parameter to the function called by the **onmessage** attribute of the parent worker (in the *.data* property).

In the *messageData* parameter, pass a value containing the data to exchange. Note that this data will be copied (cloned) before being exchanged. This value can be of any primitive type. It can also be an object: supported data types depend on the *structured clone algorithm*. This algorithm itself is defined in the HTML 5 specifications.

- Wakanda currently supports the following object types as values in *messageData*: Boolean, Number, String, Date, RegExp, Array, Object, and Functions.
- The following object types are currently not supported in *messageData* objects: ImageData, File, Blob, and FileList.
- Wakanda native objects, such as Entity collections or Datastore classes, are not supported in *messageData* objects.

Example

See the full example for the **onmessage** property in the **Worker Instances** section.

Example

This dedicated worker can be used to create a large number of entities in the background.

The parent worker code:

```
var worker = new Worker("create_entities.js");
worker.postMessage(100000); // we want to create 100,000 entities
```

The contents of the *create_entities.js* dedicated worker file is shown below:

```
function createEntities (n)
{
    var    i;
    for (i = 0; i < n; i++)
    {
        var d = new Date(); //gets data
        var entry  = new Timestamp(); //creates a new entity in the Timestamp model

        entry.testValue= d.toString(); //stores data
        entry.save(); // saves the entity
    }
    return i;
}

onmessage = function (event)
{
    if (event.data >= 0)  // event.data gets the current number of entities sent by the parent worker
    {
        createEntities(event.data);
    }
    close(); // closes the worker
}
```

terminate( )

void **terminate**()

Description

The **terminate( )** method allows you to terminate the dedicated worker execution. It closes the parent worker and sets the dedicated child worker "close" flag to **true**.

Note: *You can also end a dedicated worker child thread by calling the [#cmd id="103008"/] method from inside the thread.*

This method will allow the worker to complete its currently running code and, at the next point where it can process a new event, it will close, ignoring any queued events. If you want to allow the worker to complete all its queued events, simply pass it a new event telling it to close (see **close( )**).

When a parent worker finishes its execution for any reason, the **terminate( )** method is automatically called on all its dedicated child workers.

Since **terminate( )** waits until the currently running worker code ends, the effect may not be immediate (in particular, if the dedicated worker is executing a loop). Therefore, it is very important to avoid using "while (true) {}" loops in dedicated workers. We recommend that you always execute asynchronous calls (using **onmessage** callbacks).

wait( )

Boolean **wait**([Number *timeout*])

Parameter	Type	Description
timeout	Number	Timeout in milliseconds
Returns	Boolean	True if the worker is terminated; False otherwise

Description

The **wait( )** method allows a thread to handle events and to continue to exist after the complete code executes.

In the context of a Web worker, the **wait( )** method allows a parent Web worker thread to handle child worker events. Since the parent-child worker communication is asynchronous (based on callbacks), this method is necessary in the parent script to allow the thread to keep from terminating after the code execution and to listen for callbacks. During the waiting time, asynchronous callback events from Web workers are handled. When this method has been called, the thread stays alive until you call **close( )**.

Note: *The **wait( )** method is also available for child workers although it is usually not necessary in this context. Child worker scripts always implicitly call the wait mechanism.*

The **wait( )** method can also be used in the context of the main thread to allow asynchronous communication, for example when using or **System Workers**. In this context, to stop the **wait( )** loop, you need to use **exitWait( )**.

Note that while executing, the **wait( )** method blocks the thread but still handles callbacks.

If you specify a value (in milliseconds) in the optional ***timeout*** parameter, **wait( )** will run only during the time specified and then give the control back after this time, returning *false* if the worker is not terminated.