Storage

The Wakanda ***Storage*** object provides an interface for persistent data storage of key-value pair data in Wakanda Server. It fully implements the Web Storage W3C specification, originally designed for the client, on the server.

In addition, the following Wakanda-specific methods have been implemented for multi-thread handling:

- **lock( )**
- **unlock( )**
- **tryLock( )**

Note: *The localStorage feature is not implemented on Wakanda Server.*

Storage objects

Storage objects are automatically available through two properties of the global **Application** object:

- **sessionStorage**: stores and maintains data during each user HTTP session. Each session has its own **sessionStorage** property available and editable while the session is alive.
- **storage**: stores and maintains data while the project (application) is launched. Each application has its own **storage** property available until it is closed. The stored data are shared between all user HTTP sessions while the application is alive.

A ***Storage*** object is also available as a property of a **User** object:

- *user*.**storage**: stores and maintains user-related data as long as the server is alive (the object is not session-related).

In all cases, the ***Storage*** object has the same properties and methods available.

Storage contents

As defined in the W3C specifications, each ***Storage*** object provides access to a list of key/value pairs (also named *items*):

- Keys are strings. Any string (including an empty string) can be a valid key.
- Values can be of any primitive types. They can also be objects; in this case, supported data types depend on the *structured clone algorithm*. This algorithm itself is defined in the HTML 5 specifications. It is used internally when a ***Storage*** object value is set.
  - Wakanda currently supports the following object types as values in ***Storage*** objects: Boolean, Number, String, Date, RegEx, Array, Object, and Functions.
  - The following object types are currently not supported for ***Storage*** objects: ImageData, File, Blob, and FileList.
  - Wakanda native objects, such as Entity collections or Datastore classes, are not supported in ***Storage*** objects.

Each ***Storage*** object is associated with a list of key/value pairs when it is created. Multiple separate objects implementing the ***Storage*** interface can all be associated with the same list of key/value pairs simultaneously.

Storage Class

length

Description

The **length** property returns the number of key/value pairs (or items) currently available in the *Storage* object.
This property is readOnly. If you try to write it, a *TypeError* exception will occur.

clear( )

void **clear**( )

Description

The **clear( )** method removes all key/value pairs from the *Storage* object.
This method closes the session opened on the server with the first **sessionStorage** call.
If the *Storage* object is already empty, this method does nothing.

getItem( )

Mixed **getItem**(String *key*)

Parameter	Type	Description
key	String	Name of the key to get
Returns	Mixed	Copy of the value

Description

The **getItem( )** method returns a copy of the value stored with the given ***key*** in the *Storage* object.
This method is similar to reading a value directly from the object:

```
var x = sessionStorage.getItem("username");
    // is exactly the same as
var x = sessionStorage.username;
```

Whichever method you use, getting a value from a *Storage* object returns a "structured clone" (i.e., a copy) of the value from the original object. If you modify this value, only the copy will be affected, not the original *Storage* object. For more information about structured clones, refer to the **Storage contents** paragraph.
If the given ***key*** does not exist in the *Storage* object, a Null object is returned.

key( )

String **key**(Number *keyIndex*)

Parameter	Type	Description
keyIndex	Number	Index of the key to retrieve
Returns	String	Name of the key at the keyIndex position

Description

The **key( )** method returns the name of the key stored at the ***keyIndex*** position in the *Storage* object. The key position may change when you add or remove key/value pairs.
If ***keyIndex*** is greater than or equal to the number of key/value pairs in the *Storage* object, this method returns Null.
The **key( )** method should be used in specific cases only. Because it accesses the ***keyIndex*** value sequentially, its execution may take some time.

lock( )

void **lock**( )

Description

The **lock( )** method locks the *Storage* object to which it is applied, so that only the thread that placed it can read or modify it.
This method allows you to ensure an exclusive access for a specific JavaScript thread. While the locked status is on, no other access can be done from another thread. If another thread tries to read from or write data to the object, it will be blocked. If it tries to do a **lock( )**, it will also be blocked.
Note: *To avoid blocking thread execution, you can use the* ***tryLock( )*** *method.*
The object will be unlocked when:

- either the **unlock( )** method is called; note that if you called the **lock( )** method several times on the same thread, you should call the same number of **unlock( )** methods for the thread to be unlocked (there is an internal counter);
- or the JavaScript thread execution is terminated.

In any case, it is a good habit to call **unlock( )** once locking is no longer necessary.

removeItem( )

void **removeItem**(String *key*)

Parameter	Type	Description
key	String	Name of the key to remove

Description

The **removeItem( )** method allows you to remove an item from the *Storage* object.

In the *key* parameter, pass the name of the item to remove. If the *key* exists in the *Storage* object, the corresponding key/value pair is removed from the object.

If the *key* value is not found in the *Storage* object, the method does nothing.

setItem( )

void **setItem**(String *key*, Mixed *value*)

Parameter	Type	Description
key	String	Name of the key to set
value	Mixed	Value to set or to update

Description

The **setItem( )** method allows you to create or update an item in the *Storage* object.

In the *key* parameter, pass the name of the item to set:

- If the given key already exists in the *Storage* object, its current value is updated with the *value* you passed.
- If the given key does not exist in the *Storage* object, a new key/value pair is added to the object, with the given *key* and *value*.

This method is similar to directly assigning a value to the object:

```
sessionStorage.setItem ("username", "Bob") ;
     // is exactly the same as
sessionStorage.username = "Bob" ;
```

Whichever method you use, setting a value to a *Storage* object first creates a "structured clone" (i.e., a copy) of the value before assigning it to the *Storage* object. If you modify it later, the *Storage* object will not reflect the change. For more information about structured clones, refer to the **Storage contents** paragraph.

tryLock( )

Boolean **tryLock**( )

Returns	Boolean	True if object was locked successfully, false if object was already locked

Description

The **tryLock( )** method tries to lock the *Storage* object to which it is applied; it returns *true* in case of success and *false* otherwise.

Locking *Storage* objects allows you to ensure an exclusive access for the current JavaScript thread (see **lock( )**). But if you try to lock a *Storage* object that is already locked by another thread, the second JavaScript thread execution is blocked. To avoid this situation, the **tryLock( )** method allows you to check if the object can be locked before you try locking it:

- If the *Storage* object is not locked, the method locks it for all the other threads and returns *true*.
- If the *Storage* object is already locked, the method does nothing and returns *false*.

unlock( )

void **unlock**( )

Description

The **unlock( )** method removes a lock that was previously put on the *Storage* object. *Storage* objects are locked by using the **lock( )** method.

Once a locked *Storage* object is unlocked, all threads can have access to it.

You should execute one **unlock( )** method per **lock( )** execution.

In all cases, the object will be unlocked when the execution of the JavaScript thread that placed the lock is terminated.

Storage Objects

storage

Description

The **storage** property returns the project *Storage* object for the current application.

Data stored in **storage** is alive while the project (application) is running. This data is shared between all user sessions. You can use locking methods to handle multiple access.

Note: *You can also have access to* ***Storage*** *objects that are available for each user session by using the* ***sessionStorage*** *property.*

Storage objects have specific properties and methods, listed in the class description.

sessionStorage

Description

The **sessionStorage** property is the *Storage* object available for each HTTP session in the current application.

The **sessionStorage** property gives you an easy way to handle user sessions and to keep session-related data on the server. This data is accessible at the session level (each session has its own **sessionStorage** object).

A new session is opened on the server when you make a call to **sessionStorage** for the first time. A special cookie is then sent to the browser with a reference to the *Storage* object. Data stored in **sessionStorage** is alive while the HTTP session exists.
A session is closed when the cookie expires or when you call the **clear( )** method on the *Storage* object.

Note: *You can also access the* ***Storage*** *object that is available for the entire Wakanda project by using the* ***storage*** *property.*

Storage objects have specific properties and methods, listed in the **Storage Class** description.

user.storage

Description

The **user.storage** property returns the *Storage* object associated with the *User*. This object is automatically available for each user defined in the solution and is maintained as long as the Wakanda server is alive (it is not stored after the server shuts down). You can use it for example to write initialization data at startup or to count the number of times a user logs in or out.

Note that this property is user-related and not session-related. The **user.storage** object is available even when the user is not logged.

Example

We want to store the number of times each user connects to the application. In a custom login method, we increment the user log account (see full example in the **loginByPassword( )** method description):

```
model.Person.methods.login = function(userName, password)  //  the function gets name and password
    {
        var result =loginByPassword(userName, password, 60*60); // session is  created in case of success
        if (result) // user identified successfully
            {
                var user = currentUser(); //gets the user running the session
                var logs = user.storage.logs; //gets the user log count
                if (logs == null) // logs key does not exist, it is the first login
                    logs = 0; // initialization

                var newLog = logs + 1; // incrementation  otherwise
                user.storage.logs = newLog; //stores the new log
            };
        return result; // result is sent to the client
    }
```