WeaveAPI-0.7.0 - Documentation for Weave Version 0.7.0

The `Weave` is a navigatible object database, implemented using `Offsider` technology. It can be found at `http://weavedb.sourceforge.net/`.

This manual documents the methods understood by this offsider, as well as some of the related matters.

The API for the items stored in the database is documented seperately (WeaveAPI-0.7.0-eventMethods), as is the details of the web-based cgi interface (WeaveAPI-0.7.0-cgiInterface).

Generic methods and creating events

There are a number of methods that are used when creating events, and some of them have more general usefulness.

Perhaps methods documented in this section should be included elsewhere.

buildEvent

Build the infrastructure for a new event.

syntax:

```
    buildEvent ref
```

`ref` is the path to the event, relative to `$BASEDIRECTORY/events`.

closestAfter

Return the reference to the nearest event after the one given. Returns 0 if the weave doesn't have any events before the given one.

The given reference does not have to be a valid event. The one returned is.

syntax:

```
    closestAfter ref
```

where *`ref`* is the internal reference of the (hypothetical) event.

closestBefore

Return the reference to the nearest event before the one given. Returns 0 if the weave doesn't have any events before the given one.

The given reference does not have to be a valid event. The one returned is.

# syntax: # closestBefore *ref*

where `ref` is the internal reference to the (hypothetical) event.

deleteEvent

Remove an event from the Weave database.

Unlink it from any other events that it knows about. (Specifically, `parent`, `before` and `after`.)

syntax:

```
    deleteEvent $ref
```

WILL refuse to do it if the event has children (you must delete them first.)

NOTE: we cannot guarantee that other events don't still link to it.

TODO:

1. follow through each navigation to see whether it points to a relationship

2. check each navigation to see if the target event links back to this one a counter-argument to this one is that you should understand how your Weave is structured, and build that understanding into special methods for deleting special event types.

ensureAfterBefore

Starting at the current event, make sure that the before and after links are valid.

Keep working in both directions until everything is correct.

Syntax:

```
    ensureBeforeAfter ref
```

where `ref` is the internal reference of the event to start at.

hasEvent

A wrapper for `hasRef`.

Syntax:

```
    hasEvent ref
```

hasRef

Determine whether an event exists or not.

Syntax:

```
    hasRef ref
```

where `ref` is the internal reference to the hypothetical event.

returns the internal reference if the event exists, otherwise returns null string and generates an error.

id2dir

From a `unique.timestamp`, generate a base directory for an event.

The path returned is the base directory of an event that would have that timestamp as its id, even if such an event does not exist.

Syntax:

```
id2dir ts
```

where *`ts`* is in the form of a `unique.timestamp`.

id2ref

From a `unique.timestamp`, generate an internal reference for an event.

The reference returned is the internal reference for an event that would have that timestamp as its `id`, even if such an event does not exist.

Syntax:

```
id2ref ts
```

where *`ts`* is in the form of a `unique.timestamp`.

navigate

Send a message to an event that is referenced by a given navigation key.

syntax:

```
navigate key [ message ]
```

Where *`key`* is a weave key that contains the internal reference for an event on the weave, and *`message`* is a message to send to that event.

If *`message`* is not given, returns the internal reference for that key, otherwise sends the message to the event being referenced

If *`key`* is not found then an error is generated, and a reference of 0 is used.

If the key is found, but doesn't reference a valid event, then an error is generated, and a reference of 0 is used.

newest

Send a message to the newest event on the weave. (In otherwords, the event with the newest id)

Syntax:

```
newest [ message ]
```

where *`message`* is a message to send to the newest event.

If *`message`* is not specified, then the internal reference to the newest event is returned.

newestChild

Return the reference of the newest child of an event.

Syntax:

```
newestChild ref
```

where *`ref`* is the internal reference to an event.

newEvent

Create a new event.

Syntax:

```
newEvent [ id ]
```

Where *`id`* is in the form of a `unique.timestamp`. if *`id`* isn't specified, `unique.timestamp` is used.

Returns the internal reference of the new event.

oldest

Send a message to the oldest event on the weave. (In otherwords, the event with the oldest id)

Syntax:

```
oldest [ message ]
```

where *`message`* is a message to send to the oldest event.

If *`message`* is not specified, then the internal reference to the oldest event is returned.

prune

Remove an event, and all its children.

syntax:

```
prune ref
```

where *`ref`* is the internal reference to the event.

queueNewEvent

Queue the creation of a new event.

Syntax:

```
queueNewEvent [ ts ]
```

where *`ts`* is in the form of a `unique.timestamp`. if *`ts`* isn't specified, `unique.timestamp` is used.

NOTE: returns the **`id`** of the new event, rather than the internal reference, so that the caller can make use of `waitForEventId`.

ref2id

Convert an internal reference to an event `id`.

The internal reference does not have to point to a valid event.

syntax:

```
ref2id ref
```

where *`ref`* is the internal reference to the event.

setNavigation

Sets up a weave key to reference an event for navigation.

syntax:

```
setNavigate key ref
```

This will enable future navigation using `navigate`. *`key`* is the name of a key. It will be created if neccessary.

(typically, it is either `newestRef` or `oldestRef`)

`ref` is an internal reference to a valid Weave event. (but no check is done on validity)

`ref` defaults to 0, which indicates that the key does not yet reference a valid event.

Note: weave navigation keys are not the same as *named events* (see `namedEvents`).

Use of named events is preferable since it doesn't clutter the weave's key space, and because named events are accessable from the cgi interface. Both named events and weave navigation keys have the same syntactic sugar (ie, `this` *`key/name`* `[` *`message`* `]`).

Weave navigation keys are also different from (although serve a similar function) to event navigation keys.

waitForAttachment

Wait until a specific attachment for a specified event is created.

Syntax:

```
waitForAttachment id attachment
```

where *`id`* is the id of the event and *`attachment`* is the name of the attachment.

This method will block until the specified attachment is detected.

waitForEventId

Wait until a specific event is created

syntax:

```
waitForEvent id
```

where *`id`* is the `id` of the event that is being waited on.

This method will block until the specified event is added to the weave database.

waitForKey

Wait until a specific key for a specified event is created

Syntax:

```
    waitForKey id key
```

where *`id`* the `id` for the event (which may not exist yet), and *`key`* is the name of the key (which may not exist yet).

addChildDictionary

Create a new offsider, and implement it as a child of the weave.

syntax:

```
    addChildDictionary name [ offsider ]
```

where *`name`* is the name for the new child offsider.

if *offsider* is given (as either a base directory, or a named executable), then the child will be cloned from that offsider. otherwise the child will be is just a standard offsider (effectively empty).

This method overrides the standard `addChildDictionary` method, as supplied with the Offsider framework.

event

Send a message to an event in the weave.

Syntax:

```
    event ref [ message ]
```

if *`message`* is given, send the message to the event with internal reference *`ref`* otherwise, return that event's base directory.

if ref is `0`, send the message to the weave's `nullEvent`

eventId

Send a message to an event in the weave.

Syntax:

```
    eventId id [ message ]
```

if *`message`* is given, send the message to the event having *`id`* as its id. otherwise, return the event's internal reference.

if ref is `0`, send the message to the weave's `nullEvent`

makeExecutable

Return the source code for an executable that will send a message to this weave.

The executable is a script, which is written to stdout

syntax:

```
    makeExecutable [ baseDirectory ]
```

If *`baseDirectory`*, then use that as the base directory, otherwise, use the weave's base directory. (Why this is a useful option is anyone's guess!).

sugar

Provide syntactic sugar for messages to the weave.

if the message is in the form

```
navigationKey message
```

then convert it to $ navigate *navigationKey message*

if message is in the form

```
eventName message
```

then convert it to

```
namedEvent eventName message
```

syntax:

```
sugar text
```

This method overrides the default `sugar` method as supplied by the Offsider framework.

upgradeMe

A quick and dirty form of the standard `upgradeMe` method as supplied by the Offsider framework.

Upgrades from `pairs`, `keys/`, `methods/` and `eventMethods/`

Syntax:

```
upgradeMe
```

Weave

Send message to the Weave

syntax:

```
Weave [ message ]
```

In the case of the Weave, sends message back to itself, but keep in mind that child offsiders of the weave will also get copies of this method.

namedEvent

Send a message to a named event.

syntax:

```
namedEvent name [ message ]
```

`name` must be a key in the `namedEvents` subdictionary, and the key must reference an event.

newNamedEvent

Create a new named event.

syntax:

```
newNameEvent name [ description ]
```

`name` is the name to use for the event, `description` is an optional description, which is added as a key to the new event.

Returns the internal reference of the new event.

This method will fail if there is already a named event with this name.

Weave methods for Presentation

The following methods are understood by the Weave. They are methods to do with presenting information from the weave.

asText

Show the contents of a list of Weave events as text.

Syntax:

```
asText refs
```

where `refs` is a list of internal references.

Weave methods for html and cgi

The methods in this section are used to generate web pages for the web-accessible `cgi` interfaces.

They are methods that are understood by the Weave.

asHtml

Present a summary of an event as html.

Syntax:

```
asHtml ref
```

where `ref` is the internal reference for the event.

This Weave method is provided so that the generic `weave.cgi` can rely on a weave to present the html for an event as appropriate for that weave, (and that event).

This method can be overridden to provide custom web-page presentation appropriate for that weave.

The method **does not output a complete html page**. That is done by the cgi. It outputs a snippet of html that can be included inside the cgi-generated page.

fastEventHtml

Create a page of html to display the contents of an event

Syntax:

```
fastEventHtml ref
```

where `ref` is the internal reference of the event.

THIS IS A FAST METHOD which bypasses normal Offsider processing

fastHomeHtml

Output a snippet of html which provides basic entry points to the weave, for use by `weave.home.cgi`.

Syntax:

```
fastHomeHtml
```

THIS IS A FAST METHOD which bypasses normal Offsider processing.

fastNamedNodesHtml

Output a snippet of html which provides links to all the weave's named nodes for use by `weave.nodes.cgi`.

Syntax:

```
fastNamedNodesHtml
```

THIS IS A FAST METHOD which bypasses normal Offsider processing.

Templates

A template is an offsider that resides within the weave. It is able to provide methods for events on the weave. If an event *subscribes* to a template, then it will recognise the methods which that template has.

The weave has a child offsider called `templates`. All the templates are children of that offsider.

For example:

List all the templates in the weave:

```
weave templates childDictionaries
```

List all the methods of template *template*:

```
weave templates template methods
```

addTemplate

Add a child offsider which events can use as a template. A template can provide implementations of methods for events.

syntax:

```
addTemplate name [ offsider ]
```

`name` is the name for the template.

`offsider` is either the named executable or the base directory of an offsider from which the new template will be cloned.

Collections

A *collection* is a mechanism for grouping events together in ordered lists.

(The `previous` and `next` navigation links can do this too, but any event can only have one set of such links.)

On the other hand, and event can belong to any number of different collections.

For each collection that an event belongs to, it has a navigation key that points to one link in that collection's chain.

Each event in the collection points to its link in the chain using a navigation key. The name of this key can be anything, and it is not neccessary that all events in the collection use the same navigation key.

Each collection is thus composed of a number of links, and each link points to an event that belongs to the collection. The navigation key used by the link is `pointsTo`. The event that the link points to is said to belong to the collection. The link does not belong to the collection, it is just part of the mechanism for implementing the relationships between all the events in the collection. Each link also has navigation keys `previous` and `next`, which point to the neighbouring links, and a navigation key `parent`, which points to the common parent of all the links.

Note: Currently, *collections* are known as *relationships*

Each link in the chain has a common parent. This parent event is known as the collection's parent.

newRelationship

Create a new event which will act as a link in a collection chain.

syntax:

```
    newRelationship parent child name
```

`parent` is the relationship parent.

`child` is the event for which the relationship was created

`name` is the navigation name being used by the child to point to this relationship (redundant, but might be useful)

returns the internal reference to the new event.

newRelationshipParent

Create a new event which will act as a parent to a collection chain.

syntax:

```
    newRelationshipParent [ summary ]
```

returns the internal reference to the new event.

Method API for `nullEvent`

nullEvent

The `nullEvent` is a child offsider of the weave, but is not an event on the weave.

The `nullEvent` is used to detect when an attempt is made to navigate to a non-existen t event. Any attempt to send a message to a non-existent event will result in the message being sent to the `nullEvent`.

The `nullEvent` can be accessed with:

```
weave nullEvent message
```

isNullEvent

This method is used to test whether the target of the message is the `nullEvent`, or n ot.

Events on the weave have their own version of the `isNullEvent` method. The `nullEvent` has its version.

When sent to an event on the weave, the message `isNullEvent` will return an empty string, indicating that it is **not** the `nullEvent`.

Syntax:

```
isNullEvent
```

navigate

Respond to the navigate message, but do not navigate from here.

syntax:

```
navigate key [ message ]
```

where *`key`* is the name for a (supposed) navigation key and *`message`* is a message intended for the (non-existent) event referenced by that key.

If *message* is not given returns `0` otherwise performs `this` *`message`*

In effect, *key* is ignored completely. This method is used to capture invalid attempts at navigation from an event in the weave.

noMessage

This method is performed if the `nullEvent` is envoked, but no message is provided.

Returns `0`, which represents the `nullEvent`'s internal reference.

Syntax:

```
noMessage
```

Usage:

```
weave nullEvent
```

(which is equivalent to *`weave`* `nullEvent noMessage`).

noCommand

`noCommand` is a deprecated name for `noMessage`

Weave QueueReader

The weave has its own `QueueReader`, which can handle incoming requests so as to help insure database integrity.

You can send a message to the QueueReader using:

```
    weave QueueReader message
```

where *weave* is the named executable to the weave, and *message* is the message to send to the QueueReader.

When it processes requests from the queue, the QueueReader sends the contents of the request back to the weave as a message.

In other words,

```
    weave QueueReader queue message
```

is equivalent to

```
    weave message
```

once the request has been processed.

performRequest

What the weave's `QueueReader` finally does with the request. The contents of the request are sent to the weave as a message.

Syntax:

```
    performRequest requestFilename
```

where *requestFilename* is the file that contains the request. The filename is relative to the QueueReader's base directory.

This method overrides the default method, as supplied with the `QueueReader` framework.

Weave QueueReader

The weave has its own `QueueReader`, which can handle incoming requests so as to help insure database integrity.

Queueing a request to the weave's QueueReader has the effect of postponing the request for some later time. It also provides a mechanism for ensuring that incoming asynchronous database update requests can be queued for sequential processing.

The message

```
    weave queue message
```

is equivalent to

```
    weave message
```

The message will not be sent until the QueueReader processes it, but it is more safe.

queue

Place a request on the weave's QueueReader.

When the QueueReader processes the request, it will send the contents of the request back to the weave as a message.

Syntax:

```
    queue message
```

where `message` is a message that will be processed at some later time.