WeaveAPI-0.7.0-eventMethods - Documentation for Weave events, 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 items stored on such a database. In Weave nomenclature, a database item is called an *event*.

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

Generics

The weave is a database, and the items on the database are called *events*. Each event is itself an anonymous offsider (ie, and offsider that does not by default have a named executable).

Normally, you would send a message to an event by passing the message through the weave.

In order to reduce duplication of code, the weave itself provides many methods that events on the weave can recognise. Of course, any particular event might recognise methods that over-ride the methods provided by the weave.

isNullEvent

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-existent event. Any attempt to send a message to a non-existent event will result in the message being sent to the `nullEvent`.

This method is used to test whether the target of the message is the `nullEvent`, or not.

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`.

newerThan

Test whether this event is newer than a certain time.

Syntax:

```
newerThan numberSeconds [ ts ]
```

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

If *ts* is given, tests against that timestamp, otherwise, tests against the current time.

Calculates the difference (in seconds) between the given timestamp and the event's `id`. If the difference is less than the number of seconds given, then returns the difference (in seconds). Otherwise, returns an empty string.

olderThan

Test whether this event is older than a certain time.

Syntax:

```
olderThan numberSeconds [ ts ]
```

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

If *ts* is given, tests against that timestamp, otherwise, tests against the current time.

Calculates the difference (in seconds) between the given timestamp and the event's `id`. If the difference is greater than the number of seconds given, then returns the difference (in seconds). Otherwise, returns an empty string.

Weave

Send a message to the event's weave.

syntax:

```
Weave [ message ]
```

If *message* is not specified, returns the weave's base directory.

Event methods for Presentation

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

asHtml

Show the contents of the event as a snippet of html.

Syntax:

```
asHtml
```

This method is specifically written to integrate with the standard weave http cgi interface.

asText

Show a summary of the contents of the event as text.

Syntax:

```
asText
```

fastHtml

Return a snippet of html to display the contents of the event.

syntax:

```
fastHtml
```

This method is specifically written to integrate with the standard weave http cgi interface.

navAsHtml

Show a navigation key as a snippet of html.

syntax:

```
navAsHtml key cgiurl
```

where *`key`* is the name of the navigation key to display, and *`cgiurl`* is the url of the cgi that displays an event.

Event Navigation

The `Weave` is a navigatible database. This means that you are able to navigate from one event in the database to another (similar to the way you can link from one page to another in the World Wide Web).

All navigation from one event to another is handled in the same way. The event contains a set of *navigation keys*, and each navigation key contains a reference to an event in the Weave. (Alternatively, a navigation key can contain `0`, which indicates that no event is referenced by that key.)

The following example illustrates the concept of navigation, using the default syntactic sugar:

```
this navKey
```

will return the internal reference to the event specified by the navigation key *`navKey`*.

```
this navKey message
```

sends the message *`message`* to the event specified by the navigation key *`navKey`*.

Some navigation keys have special meaning within the weave. You should not attempt to set the values of these keys directly. Use the appropriate methods, and the keys will be maintained in a meaningful and consistent manner.

`before`, `after` - Cronological ordering of events

As each event is created, it is given a timestamp as its `id`. The timestamp implies a cronological ordering of all the events on the weave.

From each event you can navigate to the events that lie just before and just after, according to this cronological ordering. By using this natural time order, it is possible to access every event on the weave. Either start with `Weave oldest` and follow the `after` navigation links, or start at `Weave newest`, and follow the `before` navigation links.

These two keys are set whenever an event is created. Also, they are checked again under certain circumstances, and corrected if they are found to be incorrect.

`parent`, `previous`, `next`, `oldestChild`, `newestChild` - Creating heirarchical structures.

You can create heirarchical structures within the set of weave events (see methods `isChildOf`, `isParentTo`, etc).

The relationships between events in any given heirarchical structure is provided by the navigation links `parent`, `previous`, `next`, `oldestChild` and `newestChild`.

By following these links, you can visit all of the events in any given heirarchical structure.

These keys are set as appropriate whenever a heirarchical structure is modified.

Ad-hoc linkages

You are free to devise your own navigation key names, to link from one event to another (similar to the way web pages link from one to another). Low-level methods are provided to set and follow navigation links.

Often, when you are designing a weave, you will decide that a class of events will all have the same set of navigation keys, which make sense for events in that class. It is recommended that you write special methods to handle each class of events. These methods should create and manage these keys so that they remain consistent and meaningful. The details of this are, of course, domain specific and application specific.

after

Send a message to the next event in the weave after the current one.

Syntax: $ after [*message*]

If *`message`* is not given, returns the internal reference contained in that key, otherwise, sends the message to the event referenced by that key.

before

Send a message to the next event in the weave before the current one.

syntax: $ before [*message*]

If *`message`* is not given, returns the internal reference contained in that key, otherwise, sends the message to the event referenced by that key.

fastList

Starting at this event, navigate to successive events using the specified navigation key. As each event is visited, send the specified message to it.

syntax:

```
fastList [ withRef ] nav max message
```

arguments are:

`nav` - the navigation key to use

max - the maximum number of events to navigate to.

Navigation will stop when the `nullEvent` is reached, even if the maximum has not yet been reached. `all` or `0` means don't stop until the `nullEvent` is reached.

message pass this message to each event in turn.

If the keyword `withRef` is given as the first argument, then the event's internal reference will be printed for each event visited.

hasNavigation

Does this event have the specified navigation key?

syntax:

```
hasNavigation key
```

where *key* is the name of the navigation key to use.

If the key exists, the full path to the key is returned.

me

Send a message to the current event.

You would not normally use this method to send a message, since the message can be sent directly.

This method is provided for completeness (it behaves the same as other navigation-specific methods)

syntax:

```
me [ message ]
```

When a message is specified, `this` *message* is the same as `this me` *message*, however note that:

`this me` returns the internal reference.

whereas

`this` returns the base directory.

navigate

Send a message to an event that is connected to this one by a navigation key.

syntax:

```
navigate key [ message ]
```

where *key* is the name of the navigation key to use, and *message* is the message to send to that event.

If *message* is not given, returns the internal reference contained in that key, otherwise, sends the message to the event referenced by that key.

If *key* is not found in `$BASEDIRECTORY/navigation`, then an error is generated, and a reference of `0` is used.

newestChild

Send a message to the most recently added child of this event.

syntax:

```
newestChild [ message ]
```

If *`message`* is not given, returns the internal reference contained in that key, otherwise, sends the message to the event referenced by that key.

next

Send a message to the next event in the weave having the same parent as this one. The *next* event is in the order that the children were added to the parent, not timestamp order.

syntax:

```
next [ message ]
```

If *`message`* is not given, returns the internal reference contained in that key, otherwise, sends the message to the event referenced by that key.

oldestChild

Send a message to the first child to be added to this event.

syntax:

```
oldestChild [ message ]
```

If *`message`* is not given, returns the internal reference contained in that key, otherwise, sends the message to the event referenced by that key.

parent

Send a message to the parent of this event.

syntax:

```
parent [ message ]
```

If *`message`* is not given, returns the internal reference contained in that key, otherwise, sends the message to the event referenced by that key.

previous

Send a message to previous event in the weave having the same parent as this one. The *previous* event is in the order that the children were added to the parent, not timestamp order.

syntax:

```
previous [ message ]
```

If *`message`* is not given, returns the internal reference contained in that key, otherwise, sends the message to the event referenced by that key.

renameNavigation

Change the name of an existing navigation key.

syntax:

```
renameNavigation oldname newname
```

where *oldname* is the existing name for the navigation key, and *newname* is the name to change it to.

setNavigation

Set a value for a navigation key. Once set, the key can be used to navigate to the event referenced by that key.

syntax:

```
setNavigate key [ reference ]
```

key is the name for the key. Avoid using standard names which have special meaning to weave events. Standard names are: `me after before parent next previous oldestChild newestChild`

reference is an internal reference to a Weave event. (no check is done on its validity)

If *reference* is not specified, then it will be taken to be `0`, which indicates that the key does not reference a valid event.

This can be used to disable a reference without deleting the key.

sugar

Provide syntactic sugar for navigation.

converts a message is in the form:

```
navigationKey message
```

to:

```
navigate navigationKey message
```

syntax:

```
sugar text
```

This method overrides the default `sugar` method provided by the Offsider framework.

Parents

You can set up heirarchical structures within the weave events by nominating various events as being the parents of other events. Each event have at most one parent, but a parent may have any number of children.

closestChildAfterId

Find the closest child of this event which comes after the given timestamp.

Syntax:

```
closestChildAfterId ts
```

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

Returns the reference to the closest child, if there is one. `0` otherwise.

closestChildAfterRef

Find the closest child of this event which comes after the given event reference.

Syntax:

```
closestChildAfterRef ref
```

where *`ref`* is an event reference.

Returns the reference to the closest child, if there is one. `0` otherwise.

closestChildBeforeId

Find the closest child of this event which comes before the given timestamp.

syntax:

```
closestChildBeforeId ts
```

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

Returns the reference to the closest child, if there is one. `0` otherwise.

closestChildBeforeRef

Find the closest child of this event which comes before the given event reference.

Syntax:

```
closestChildBeforeRef ref
```

where *`ref`* is an event reference.

Returns the reference to the closest child, if there is one. `0` otherwise.

isChildOf

Make this event the newest child of another event (its parent).

syntax:

```
isChildOf ref
```

where *`ref`* is the internal reference of the event that is to be the parent.

isParentOf

make this event the parent of another event. The other event becomes this event's newest child.

syntax:

```
isParentOf ref
```

where *`ref`* is the internal reference of the event that is to be the child.

orphan

Unlink this event from its parent, if any.

syntax:

```
orphan
```

prune

Remove this event and any children. Recursively prune the children as well.

Syntax:

```
prune
```

Named Events

A named event is an event that can be referenced by name, using a syntax like

```
weave name message
```

where *weave* is the named executable for the weave, *name* is the name of the event and *message* is the message to send to the event.

isChildOfNamedEvent

Make this event a child of a named event.

syntax:

```
isChildOfNamedEvent name
```

where *name* is the name of the named event.

nameMe

Make this event into a named event.

syntax:

```
nameMe name
```

name is the name to use for this event.

Templates

An event can *subscribe* to zero or more *tempates*. When it is searching for a method, it will look to its templates as part of that search.

A template therefore provides a mechanism somewhat similar to the concept of a class in certain (multiple inheritance) object-oriented programming languages.

addTemplates

Add to the list of templates for this event

syntax:

```
addTemplates templates
```

where *templates* is a list of template names.

The new template names are added at the head of the existing list, so they will be searched in first.

externalMethods

Provide a list of all methods that the event understands, which are not owned by the event.

syntax:

```
externalMethods [ fullPath ]
```

If `fullPath` is specified as a keyword, then the full path of each executable is returned, otherwise just the method names are returned.

This method overrides the default `externalMethods` method, as provided by the Offsider framework.

isMethod

Determine whether this event responds to the named method.

syntax:

```
isMethod methodName
```

Returns the full path to the executable that implements the method.

resolve in this order:

method owned by the event: *methodName*

method owned by a template: *methodName*

method supplied by the weave: `offsider.`*methodName*

method supplied elsewhere: `offsider.`*methodName*

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

setTemplates

Set the list of templates for this event.

syntax:

```
setTemplates templates
```

where *templates* is a list of template names.

templateMethods

List all the methods provided by this event's templates.

Syntax:

```
templateMethods
```

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 relationship, and attach this event to it.

syntax:

```
newRelationship name [ summary ]
```

name is the name of the navigation key from this event to the relationship.

summary is used as a the `summary` key to the parent of the relationship chain.

relationshipNext

Send a message to the next event in a relationship.

syntax:

```
relationshipNext name [ message ]
```

`name` is the name of the key that points to the relationship. *`message`* is the message to send to the next event.

relationshipPrevious

Send a message to the previous event in a relationship.

syntax: $ relationshipPrevious *name* [*message*]

`name` is the name of the key that points to the relationship. *`message`* is the message to send to the next event.

setRelationship

Attach this event to a pre-existing relationship chain.

syntax: $ setRelationship *name previous previousName*

`name` is the relationship name to be used by this event.

`previous` is the internal reference of another event already in the relationship chain.

`previousName` is the relationship name being used by the previous event.

Event Flow

Because the weave is a navigatible database, and because each event on the weave is an object, we have the possibility for a novel form of programming called *event flow*.

Consider using the weave to store a program. Each step in the program is stored in each own event.

In a normal imperative programming language, the sequence of execution flows naturally from one step to the next. To alter that sequence of execution, we have various (standard) control structures. The simplest imperative languages have `goto` and `conditional goto` statements to over-ride this normal sequential flow.

The concept of *event flow* removes the idea of an encasing or enclosing control structure, and instead gives each event the ability to decide the next action in the sequence.

Thus, at every point, the event not only provides the action at that point, it also provides the decision for which event to go to next.

Under the current model for event flow, we have two possible directions in which the flow can continue. These are called *down* and *across*, and they correspond to *if true* and *if false* in a conditional goto.

This concept is still being worked out, and the methods documented in this section will probably undergo a redesign in the future.

flow

Perform this event's part in a flow sequence.

syntax:

```
flow
```

The sequence is:

Perform the flow rule.

If the rule returns an empty string, flow across, otherwise flow down.

flowAction

What to do when the rule has been satisfied, prior to flowing down to the next event.

syntax:

```
flowAction
```

By default, performs `this NOP`.

flowDown

The rule was satisfied, flow down to the next event

syntax:

```
flowDown
```

flowEntry

set up ready for the flow decision in this event

syntax:

```
flowEntry
```

By default, performs `this NOP`.

flowNext

The rule was not satisfied, flow to the next event

syntax:

```
flowNext
```

flowNextAction

what to do when the rule has NOT been satisfied, prior to flowing down to the next event.

syntax:

```
flowNextAction
```

By default, performs `this NOP`.

flowRule

A rule to decide whether to flow across or down.

Syntax:

```
flowRule
```

If this event satisfies the rule, return a non-empty string If this event does not satisfy the rule, return an empty string.

By default, performs `this NOP`. The default action will stop the flow at the `newestChild`.

flowStop

What to do when the flow stops at this event

syntax:

```
flowStop
```

By default, performs `this NOP`.

startFlow

Start a flow from this event.

syntax:

```
startFlow
```

Various environment variables can be used to specify the various parts of the flow mechanism at each event that it flows to.

`FLOWENTRY` message on entry to the event

`FLOWRULE` message to decide whether to flow across or down.

`FLOWACTION` message to perform before flowing down

`FLOWDOWN` message to flow down

`FLOWNEXTACTION` message to perform before flowing across

`FLOWNEXT` message to flow across

`FLOWSTOP` message to cease the flow.

If any of these are not set, they will get set to the appropriate default method name. For example `$FLOWENTRY` will get set to `flowEntry`.