Webscavator Documentation

Release 0.0.1

Sarah Lowman

August 26, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

This documentation (apart form this page) has been completely auto-generated from the Python code docstrings, and describes how Webscavator works.

Any TODOs have the following format:

Note: This is a TODO. Please fix!

CHAPTER

TWO

WEBSCAVATOR API

2.1 launch.py

2.1.1 Starting Webscavator

To start webscavator:

```
> python launch.py runserver
```

Then go to your preferred web browser and go to http://localhost:5000

action_runtests(*functional=False, unit=False*)
Run tests by first calling *setup()* in *Utils* and then *runTests(unit, functional)* in *Testing*

make_app()
Runs webscavator. First runs *setup()* in *Utils* which loads the configuration file, and then runs *make_app()* in *application.py*.

2.2 application.py

This file stores the WSGI application class which takes in the URL requested, and responds with an html page.

class Application()
WSGI Application class. See the WerkZeug documentation for more details on how this works.

dispatch(*request, adapter, endpoint, vars*)
Dispatch the request to the correct endpoint, i.e. the controller method found in one of *Controllers*

load_session(*request*)
Load cookies from *session_store* and puts them into *request.session*.

static **make_url_map**()
Map that defines all the valid URLs and what endpoint they should be dispatched to. For example the index page, (which is just http://localhost:5000/) points to *generalController.index()* in *generalController*. The map uses *controller_lookup* dictionary in *Controllers*.

save_session(*request, response*)
Save cookies in *session_store* and *response*.

make_app()
Make the WGSI application. If *debug* is set to True in the configuration file, then an interactive debugger will be displayed in the web browser when there are errors. Otherwise a page 500 will be displayed.

2.3 Controllers

Package Contents:

2.3.1 baseController

baseController

Contains the *BaseController* which all other controllers inherit from and a couple of other useful functions for JSON-ifying responses.

class BaseController(*request, urls*)
BaseController contains useful methods for all the controllers, contains the return functions and cookie setters/getters.

addDefaultFilters()
Adds the default filters for the timegraph such as filtering by browser type, group, work hours, Google searches and local files. Gets called when a new case is being set up in *finish_wizard()* in *caseController*.

defaultResponse(*statuscode, *location, **vars*)
Return the rendered template with variables with the given *statuscode*.

renderTemplate(**location, **vars*)
Returns a rendered Mako HTML template.

return404()
Calls *defaultResponse* with a 404 page. This method is called by *application()* when none of the entries in the URL map match with any controller endpoints.

return500()
Calls *defaultResponse* with a 500 page. This method is called by *application()* when the endpoint has raised an exception or an error has happened.

returnResponse(**location, **vars*)
Calls *defaultResponse* with the expected page for the controller endpoint, with a 200 status code.

validate_form(*schema*)
Validates a form post against schema in *Forms*.

If no form was posted, returns *False*. If the form was posted and it is invalid, returns *False* and sets *self.form_error*, a dictionary with input names and their corresponding error messages. If the form validated correctly, returns *True* and sets *self.form_result* to the validated results.

write_log (*dbfile, msg*)
Computes a MD5 hash of the dbfile by calling *_computeHash(file)* and then appends the hash in a file for that dbfile by calling *_storeHash(hash, dbfile, msg)*. Returns the hash and the path to the folder where the hash file is kept.

class FormState (*obj, case, urls*)
Sometimes the validators in *Validators* need to know stateful information, such as the current case being edited to compare values with. This object is passed to the validators as *state*.

jsonify (*func*)
Wrap a function so the return value is 'JSON-ified' and wrapped in a Werkzeug response object. This essentially returns something an AJAX request will understand.

jsonifyfile (*func*)
Similar to *jsonify(func)*, but used for file uploads. They have to be put in textareas to work properly.

2.3.2 caseController

caseController

Contains the class that has all the endpoints for loading, adding and editing a case.

class CaseController (*request, urls*)
Controller for the set up and editing of cases.

addData (*entry*)
Given a validated form called *entry*, adds the group to the database, then calls *self.addEntry()* to convert the data in *entry['data']* to *Entry* objects. Returns the group if the adding of entries was successful, and otherwise returns *None*.

addEntry (*program, file, group*)
Calls the generator *convert_file()* found in *Converters* on each row of the file, and adds the result to the database. If an exception happens during the converting and adding of data, then the session is rolled back and *None* is returned. Otherwise *True* is returned.

Note: This had been optimised to make the adding of data as fast as possible, but has been slowed down again by adding search terms.

ToDo: Optimise the adding of search terms.

case_details (*edit=False*)
First step of the wizard: add or edit case details.

check_details (*edit=False*)
Third step of wizard: user gets to check everything has been added correctly.

edit1 ()
Endpoint for editing a case. Returns *self.case_details(edit=True)*.

edit2 ()
Endpoint for editing CSV files. Returns *self.file_details(edit=True)*.

edit3 ()
Endpoint for checking details are correct. Returns *self.check_details(edit=True)*.

edit4 ()
Endpoint for the end of the wizard. Returns *self.finish_wizard(edit=True)*.

file_details (*edit=False*)
Second step of wizard: add the browser history files.

finish_wizard(*edit=False*)
Forth step of wizard: The wizard is complete. Creates a hash of the database file and adds message to log file by calling *self.write_log(dbfile, msg)* found in *baseController*.

jsonAddCase(**args, **kwds*)
Endpoint for wizard 1 form (adding case name and dbfile name). This is done via Ajax. If the form has errors, the error dictionary *self.form_errors* is jsonified and passed back to the webpage and the errors displayed to the user. If the form is valid, then the new case is created, the database initialised and data saved to the database. This will then return *True*.

jsonAddEntries(**args, **kwds*)
Endpoint for wizard 2 form (adding CSV/XML data). This is done via Ajax. If the form has errors, the error dictionary *self.form_errors* is jsonified and passed back to the webpage and the errors displayed to the user. If the form is valid, then *self.addData()* is called. If all goes well, this will then return *True*, otherwise if their are errors in the adding of the data in *self.addData()* then self.form_erros is returned.

jsonEditCase(**args, **kwds*)
Same as *self.jsonAddCase()* apart from the user is not allowed to edit the database name - this can be done manually be renaming the file when not in use.

jsonEditEntries(**args, **kwds*)
Same as *self.jsonAddEntries* but allows the editing of old entries and the addition of new entries.

load(***vars*)
Endpoint for the page to load an old case.

loaded()
Endpoint for when user has chosen to load a particular case. Checks whether the case is valid. If so, returns *finish_wizard()*. Otherwise, it returns *self.form_errors()*.

wizard()
Endpoint for the index page when a case is not yet loaded. Gives the user a choice whether to add a new case or load an old case.

wizard1()
Endpoint for adding a new case. returns *self.case_details()*.

wizard2()
Endpoint for adding new CSV files. returns *self.file_details()*.

wizard3()
Endpoint for checking details are correct. Returns *self.check_details()*.

wizard4()
Endpoint for the end of the wizard. Returns *self.finish_wizard()*.

2.3.3 generalController

generalController

Contains the class for the endpoints of general pages that do not need much complex processing (apart form the index page).

class GeneralController(*request, urls*)
Controller for general pages, e.g. the index page, about page and help pages.

about()
Endpoint for the about page.

guidelines()
Endpoint for guidelines on what data to add to cases.

help()
Endpoint for the index help page.

help_addfiles()
Endpoint for the help page on how to add new web history program file convertors

index()
Endpoint for index page of webscavator. If a case has been loaded, this will show information about the case and lead onto the visualisations. Otherwise, the index page will display a choice to either load a case or create a new case.

userguide()
Endpoint for the user documentation.

userguide_part1()
Endpoint for the user documentation on how to extract web history files and add/load/edit a case.

userguide_part2()
Endpoint for the user documentation on how to sue the visualisations.

2.3.4 visualController

visualController

Contains the class for the visualisation AJAX endpoints and the add filter form, as well as some helper functions.

class VisualController(*request, urls*)
Controller for filter pages and AJAX visualisation calls

addFilter()
Endpoint for the add filter pop-up page.

jsonAddFilter(**args, **kwds*)
Endpoint for the AJAX request to validate and add a new filter. If valid, the filter is added and a new hash is created and appended to the database file with the current date and time, returning *True*. Otherwise *self.form_error* is returned.

jsonGetDomains(**args, **kwds*)
Endpoint for the AJAX request to get the domain names. Calls *URL.getTop()* in *Database Models* which returns a list of domain name tuples in ascending order of number of visits.

jsonGetEntries(**args, **kwds*)
Endpoint for the AJAX request to get the timegraph. Calls *Case.getTimeGraph()* in *Database Models* with the date, time and filter options and returns a list of dictionaries that Flot can understand to plot. The dictionaries are those points to be highlighted, removed and visible.

jsonGetWordClouds(**args, **kwds*)
Endpoint for the AJAX request to get the search terms in the word cloud. Calls *Filter.onlineSearches()* in *Filters* for each search engine in the config file. Returns a dictionary of terms.

static **processOptions**(*opts*)
Process the options for each filterable table in *Database Models*. Returns a dictionary with the table attribute as keys and the operations, values and value type as the value tuple.

convertDates(*dates*)
Given a list of dates in JavaScript date format, returns a list of Python dates. JavaScript stores dates in milliseconds since the Epoch.

convertFilters(*request_args*)
For each request argument, if it is a filter, than put it in the appropriate filter list. The request arguments will either be "highlight" or "remove" and will subsequently be put in either *highlight_funcs* or *remove_funcs*. Returns those lists.

convertTimes(*times*)
Given a list of times stored as a fraction e.g. 15.5 is 3:30pm, returns a list of Python times.

2.3.5 Controllers

Controllers control the data between the database model and the html pages. Each controller inherits from *baseController* and consist of endpoints that requests are dispatched to and helper functions.

controller_lookup
controller_lookup is a dictionary used by *make_url_map()* in *application.py* to find the correct endpoints to dispatch the requests to. Any new controllers should have a line added to this dictionary.

2.4 Converters

Package Contents:

2.4.1 xmlConverter

Children:

Web Historian Converter

class WebhistorianConverter(*xml_file*)
Converts Web Historian XML output to normalised format. The toplevel element for a web history entry is *UrlHistoryItem*.

process_element(*data*)
process_row takes in an xml element that contains one web history entry, and returns a normalised dictionary as output.

XML Convertors

Module contains abstract class that all web history programs that output XML data must inherit from.

All children must return in *process_element(row)*:

```
return {
    'type': type of entry,
    'url': URL entry,
    'modified_time': modified time of entry,
    'access_time': access time of entry,
    'filename': entry file name,
    'directory': entry directory,
    'http_headers': HTTP headers of entry,
    'title': entry page title,
    'deleted': if entry is deleted,
    'content_type': entry content type,
```

```
    'browser_name': browser used, one of 'Firefox', 'Internet Explorer', 'Safari', 'Chrome', 'Opera',
    'browser_version': browser version,
    'source_file': profile or location/name of file
}
```

All entries in the dictionary may be *None* apart from *url* and *access_time*.

class **XMLConverter**(*xml_file*)

process()
Reads in the XML file. Finds all the top level web history elements and returns a generator which can be looped over to get the normalised element entries.

2.4.2 csvConverter

Children:

Chrome Cache Viewer Converter

class **ChromecacheviewerConverter**(*csv_file*)
Converts Chrome Cache Viewer CSV output to normalised format.

To obtain CSV data from Chrome Cache Viewer, select all the entries, and go to *File* > *Save Selected Items*. Save as a *Tab-delimited Text File*, and then change the extension to *csv*. Make sure the file is in UTF-8 encoding, otherwise the CSV converter will throw an exception.

process_row(*row*)
process_row takes in a row from the CSV file, and returns a normalised dictionary as output.

One row of Chrome Cache Viewer's CSV data looks like so:

filename, url, content_type, file_size, access_time, server_time, modified_time, expired_time, server_name, http_headers, content_encoding, cache_name, cache_control, e_tag, _

Fox Analysis Converter

class **FoxanalysisConverter**(*csv_file*)
Converts Fox Analysis CSV output to normalised format.

To obtain CSV data from Fox Analysis click on *File* > *Export to* > *CSV File*. Please load in the *Website.csv* file produced.

This is comma delimited by default and has 1 line at the top for titles and headers.

process_row(*row*)
process_row takes in a row from the CSV file, and returns a normalised dictionary as output.

One row of Fox Analysis CSV data looks like so:

id, fromvisit, datevisited, url, host, totalvisitcount, type, frecency, title

Pasco Converter

class **PascoConverter**(*csv_file*)
Converts Pasco CSV output to normalised format.

To obtain CSV data from Pasco run:

```
pasco index.dat > output.csv
```

This is tab delimited by default and has 3 lines at the top for titles and headers.

process_row(*row*)
process_row takes in a row from the CSV file, and returns a normalised dictionary as output.

One row of Pasco CSV data looks like so:

type, url, modified_time, access_time, filename, directory, http_headers

Net Analysis Converter

class NetanalysisConverter(*csv_file*)
Converts Net Analysis CSV output to normalised format.

To obtain CSV data from Net Analysis, click on *File* > *Export History As* > *Tab Delimited Text*. Rename the extension to *csv* instead of *txt*.

process_row(*row*)

process_row takes in a row from the CSV file, and returns a normalised dictionary as output.

Net Analysis produces the same output regardless of what type of browser was used, producing *a lot* of columns to process. One row of Fox Analysis CSV data looks like so:

type, tag, last_visited_utc, access_time, hits, user, url, host, title, abs_path, query, fragment, port, url_category, username, password, redirect_url, feed_url, referral_url, favicon_url, cache_folder, cache_file, extension, length, exists, http_response, cache_entry_type_flag, content_type, content_length, content_encoding, active_bias, date_first_visited, date_expiration, modified_time, date_index_created, date_added, date_last_sync, source_file, source_offset, index_type, browser_version, ie_type, status, bookmark, urn

Note: Net Analysis allows users to move columns around and export the data in it's new format. Currently this convertor assumes the columns are in the correct format and does not work when they are moved / there are less columns.

ToDo: See if there is a work around for this.

CSV Converters

Module contains abstract class that all web history programs that output CSV data must inherit from.

All children must return in *process_row(row)*:

```
return {
    'type': type of entry,
    'url': URL entry,
    'modified_time': modified time of entry,
    'access_time': access time of entry,
    'filename': entry file name,
    'directory': entry directory,
    'http_headers': HTTP headers of entry,
    'title': entry page title,
    'deleted': if entry is deleted,
    'content_type': entry content type,
    'browser_name': browser used, one of 'Firefox', 'Internet Explorer', 'Safari', 'Chrome', 'Opera',
```

```
    'browser_version': browser version,
    'source_file': profile or location/name of file
}
```

All entries in the dictionary may be *None* apart from *url* and *access_time*.

class **CSVConverter**(*csv_file*)

Children classes must override delimiter and skip:

delimiter the CSV file delimiter.

skip how many lines to skip at the top of the CSV file (including header lines).

process()

Reads in the CSV file, skips *skip* number of lines and then for each line in the CSV file calls *self.process_row()* in the child class.

Returns a generator which can be looped over to get the normalised row.

2.4.3 Converters

All the modules in this package represent a particular web history program which gives an output such as a CSV file, XML file or text file.

Each module takes in a row from the output and returns a particular dictionary which is used to add that row to the webscavator database. see *csvConverter* for details of adding CSV data and *xmlConverter* for details of adding XML data.

2.4.4 Variables

program_lookup is a dictionary of all the currently supported programs: key = Program long name and value is name the short name stored in the database. *program_lookup* is accessed through the methods below.

program_info is a dictionary of useful information for each of the programs supported. The minimum entries for each dictionary should be *image* - what icon is associated with the program (default is application.png) and *description* - how to produce a valid file form the program (use HTML). Other entries can be added and used when needed. *program_info* is also accessed through the methods below.

Add to these two dictionaries when adding a new program converter.

2.4.5 Functions

convert_file(*type, file*)

Given a file, locates the correct converter and returns a generator which produces a normalised dictionary of data for each row in the file.

get_file(*name*)

Given a full name, return the file type.

get_name(*id*)

Given the name of the python file the converter is stored in, return the full name of the converter.

get_names()

Return a list of all the full names of the converters.

get_program(*name*)

Given a full name, return the python file name.

get_program_info(*program*)
Given a program name, return the dictionary of information about it.

get_program_infos()
Return the dictionary program_info with information about each of the supported programs.

get_programs_files()
Return a list of (full name, file type) tuples

2.5 Forms and Validators

Package Contents:

2.5.1 Forms

Each class represents the fields present on a particular web form, and how it should be validated. Specific validators can be found in *Validators*. All forms inherit from Schema.

If the form has an error, the validator will throw an exception and add the name of the form field and the error message to a dictionary *form_error*, which is passed back to the web page for the user to correct. If the form is valid, the endpoint will process the form, and return the successful web page.

Note: Most of these forms were created before it was realised that files other than CSV were needed (such as XML). Therefore a lot of form field names are 'csv_file' rather than just generic 'file'.

ToDo: Change any field names that refer to csv data to be more generic. Applies to Mako templates too.

class AddData(**args, **kw*)
Form to add a new file. Inherits from *Data()*

Attributes

program – web history program used to create file.

name – name of this dataset.

desc – optional description of dataset.

data – the web history file.

Messages

badDictType: The input must be dict-like (not a `%(type)s`: `%(value)r`)

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field `%(name)s` was not expected.

class Case(**args, **kw*)
Case Schema.

Messages

badDictType: The input must be dict-like (not a `%(type)s`: `%(value)r`)

badType: The input must be a string (not a `%(type)s`: `%(value)r`)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class Data(**args, **kw*)
Data Schema.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class EditData(**args, **kw*)
Form to edit file. Inherits from *Data()*

Attributes

group – the current data group being edited. Validator checks this exists.

program – web history program used to create file.

name – name of this data.

desc – optional description of data.

keepcsv – user chooses to keep the current file or upload another.

data – the new file if keepcsv is *False*.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class Filter(**args, **kw*)
Represents one filter line. *add_filter_form* can have many filter lines.

Attributes

cls – The Class the filter applies to.

attribute – the Class's attribute the filter applies to.

function – the operation to do on the attribute.

value – the value the attribute will be compared with.

value_list – the file of items to compare the attribute to.

Either *value* or *value_list* will be selected, the other will be *None*.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class add_filter_form(**args, **kw*)
Form to add a filter.

Attributes

filter – list of *Filter()*.

name – name of the filter.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class edit1_form(**args, **kw*)
Form to edit case. Inherits from *Case()*

Attributes

name – new name of case.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class edit2_form(**args, **kw*)
Form to edit multiple files.

Attributes

csv_entry – list of *EditData* objects.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class load_form(**args, **kw*)
Form to load a case Sqlite database file.

Attributes

case – the name of a case. Validator checks case exists on disk.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class wizard1_form(**args, **kw*)
Form to add a new case. Inherits from *Case()*.

Attributes

name – name of case.

dbfile – file to save case to. Validator checks this file doesn't already exist.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

class wizard2_form(**args, **kw*)
Form to add multiple files.

Attributes

csv_entry – list of *AddData* objects.

Messages

badDictType: The input must be dict-like (not a %(type)s: %(value)r)

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

missingValue: Missing value

noneType: The input must be a string (not None)

notExpected: The input field %(name)s was not expected.

2.5.2 Validators

Each class validates a certain field or group of fields within a form.

If the field(s) have errors, the validator will throw an *Invalid* exception and add the name of the form field and the error message to a dictionary, which is passed back to the web page for the user to correct.

If the field is valid, the validator will return the value. The validators may also process the value and return the processed value, e.g. the value is a Case id, and the processor returns the Case object.

Note: Most of these forms were created before it was realised that files other than CSV were needed (such as XML). Therefore a lot of form field names are 'csv_file' rather than just generic 'file'. Also, since XML (and any other added convertor) files are allowed, the uploaded file is no longer checked to see if it ends with CSV. This should be fixed to check against the type of file it expects.

ToDo: Change any field names that refer to csv data to be more generic. Applies to Mako templates too.

ToDo: Validate the uploaded data against what it should be. E.g Webscavator = XML and Net Analysis = CSV. PErhaps add a column to *program_lookup* or *program_info* in *Converters*.

ToDo: Add more checks for duplication, e.g. not allowing groups or filters with same name.

class CheckAllowed(**args, **kw*)
Check the value is within a set of predefined values.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class CheckAllowedAttributes(**args, **kw*)
Checks that the attribute belongs to the class in the same filter.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class CheckAllowedClass(**args, **kw*)
Checks to see if the class is one of a predefined set of allowed classes.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: You have chosen an invalid selection

noneType: The input must be a string (not None)

class CheckAllowedFunctions(**args, **kw*)
Checks that the operation is allowed on the attribute in the same filter.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class CheckAllowedProgram(**args, **kw*)
Checks that the web history program selected is within a set of predefined programs.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: You have chosen an invalid program

noneType: The input must be a string (not None)

class CheckAllowedValues(**args, **kw*)
Checks that the value is allowed for the attribute in the same filter.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class CheckCaseDoesntExist(**args, **kw*)
Checks that case database files do not already exist on disk.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

duplication: This database file has already been used

empty: Please enter a value

noneType: The input must be a string (not None)

class CheckCaseExists(**args, **kw*)
Checks that case database files does exist.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: You have chosen a case that does not exist

noneType: The input must be a string (not None)

class CheckHasItems(**args, **kw*)
Checks the value is not empty / *None* / *false* etc.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: Please select an item.

noneType: The input must be a string (not None)

class File(**args, **kw*)
Checks whether an uploaded file is empty or not.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class GroupInDatabase(**args, **kw*)
Checks that when editing a group of data, that the group is the correct one.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: An invalid id for these web browser files has been given

noneType: The input must be a string (not None)

class NoDuplicates(**args, **kw*)
Checks there are no duplicates in the inputed list.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: You selected the same value more than once.

noneType: The input must be a string (not None)

class NonAscii(**args, **kw*)
Checks value is alpha numeric.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: Please remove any spaces and non ASCII characters.

noneType: The input must be a string (not None)

class NotInDatabase(**args, **kw*)
Checks the data supplied is not already in database when adding a new object. If editing an object, then it will already be in the database, so must supply *currentObj* in *state*. If they match then allowed.

state is a *FormState* object which is populated in *validate_form* in *baseController*.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

duplication: This has already been used

empty: Please enter a value

noneType: The input must be a string (not None)

valueNotAllowed(*obj*)
Override to check if values are not in a list of disallowed values.

class **RemoveEmpties**(**args, **kw*)
Removes empty items from a list.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class **RequireIfCondition**(**args, **kw*)
RequireIfCondition requires 'requireds' fields if 'field' [condition] 'value'. [condition] is defined by *RequireIfCondition*'s children as can be equals or not equals.

E.g. if a user selects the option 'other' in a select box, then it is required that they fill in a value in a text input called 'Other'.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class **RequireIfEquals**(**args, **kw*)
Child of *RequireIfCondition*, the condition is ==

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

condition()
eq(a, b) – Same as a==b.

class **RequireIfNotEquals**(**args, **kw*)
Child of *RequireIfCondition*, the condition is *!=*

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

condition()
ne(a, b) – Same as a!=b.

class **StringNone**(**args, **kw*)
Checks to see if value has a string value 'None', if so, convert this to a Python *None*.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: The text you have entered is invalid

noneType: The input must be a string (not None)

class **Upload**(**args, **kw*)
Upload a file. The value returned is the file stream.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

class **UploadData**(**args, **kw*)
Checks the file uploaded is a CSV file.

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

invalid: You did not upload a CSV file

noneType: The input must be a string (not None)

class **ValidCSVData**(**args, **kw*)
Checks that the data in a CSV file can be processed correctly

Messages

badType: The input must be a string (not a %(type)s: %(value)r)

empty: Please enter a value

noneType: The input must be a string (not None)

2.5.3 Forms and Validators

Forms and Validators use the FormEncode library.

Every HTML form in Webscavator is validated using *Forms* and *Validators*. Form data is kept in *self.request.form*, e.g.:

```
request.form['date'] = '1/1/2010'
```

Each endpoint that wants to validate a form calls *self.validate_form(schema)* defined in *baseController*, where *schema* is one of the classes in *Forms*.

The data is validated according to the schemas validators, most of which come from FormEncode, but others are defined in *Validators*. If there is an error, the validator will raise an exception which gets stored in *self.form_errors*, a dictionary of form names and their error messages, e.g.:

```
form_errors['date'] = "The date must be in dd/mm/yyyy format."
```

If there are no errors, *self.form_result* gets populated with the validated results, e.g.:

```
form_result['date'] = datetime(2010, 1, 1)
```

2.6 Models

Package Contents:

2.6.1 Database Models

Models

This is where all the database models are defined. Each database table is represented by a class, and each object in that class is a table row. SQLAlchemy converts the objects into rows and any manipulation on them as SQL queries to abstract the model from the actual database implementation used.

class `Browser`(*name, version, source*)
Class that stores browser types.

id browser id

name browser name

version browser version

source browser source. this is either a profile for Firefox/Chrome or the file location for IE, e.g. history or cache

entries list of entry items that were made using this browser

static `getPercentages`()
Used in the overview statistics. Returns a list of *(browser object, percent)* tuples for each browser where percent is the percentage this browser is used in all the entries.

`id`
Public-facing descriptor, placed in the mapped class dictionary.

`name`
Public-facing descriptor, placed in the mapped class dictionary.

`source`
Public-facing descriptor, placed in the mapped class dictionary.

`version`
Public-facing descriptor, placed in the mapped class dictionary.

class `Case`(*name*)
Class that stores the case. Each database will only have one case.

id case id

name case name

date date this case was created

groups A list of group objects that belong to this case

static `create_database`(*dbfile*)
Creates the database file, initialises all the models in this file and binds the database to webscavator's session.

`date`
Public-facing descriptor, placed in the mapped class dictionary.

`dblocation`(*dbfile*)
Return the absolute location of a given database file.

static `filter_queries`(*q, remove_funcs, highlight_funcs*)
This expects *q* to have entry, browser, url, group and search_terms available for filtering. Returns three queries with the correct entries put in each according to the filters: *q_removed*, *q_highlighted* and *q_not_highlighted*.

formatted_date
A property for formatted_date. Returns the date in a normalised format.

static **getAllEntries**(*case*)
Returns all the entries for this case where the access time and date are not *None*.

static **getMinMax**(*latest, case*)
Return the minimum and maximum dates for Flot. Given a date, returns minimum which is 1 month prior on the 1st of the month, to the last day of the month for the given date. E.g. given 12th May, returns (1st April, 31st May). This is returned in JavaScript milliseconds since the Epoch.

static **getNewestEntry**(*case*)
Returns the latest Entry for this case.

static **getOldestEntry**(*case*)
Returns the oldest Entry for this case.

static **getTimeGraph**(*startdate, enddate, starttime, endtime, case, remove_funcs, highlight_funcs, remove_duplicates=False, duplicate_time=0*)
Given a start and end date and time and filters, returns three lists of entry points: *highlighted*, *not_highlighted* and *removed* by calling *filter_queries()* Each entry will be in one of those lists for Flot to draw.

static **get_case**()
Get the current case.

id
Public-facing descriptor, placed in the mapped class dictionary.

static **load_database**(*dbfile*)
Given a database file, connects to the database and binds it to webscavator's session.

name
Public-facing descriptor, placed in the mapped class dictionary.

class Group(*name, desc, case, program*)

Class that stores information about a particular uploaded file.

id group id

name group name

description group description

csv_name the filename of the uploaded file

program the program used to make the file

case the case object this group belongs to

entries list of entry objects for this group

Note: Most of these forms were created before it was realised that files other than CSV were needed (such as XML).

ToDo: change *csv_name* to *file_name*.

case
Public-facing descriptor, placed in the mapped class dictionary.

case_id
Public-facing descriptor, placed in the mapped class dictionary.

csv_name
Public-facing descriptor, placed in the mapped class dictionary.

description
Public-facing descriptor, placed in the mapped class dictionary.

getEndDate()
Get the ending date of all the group's entries.

getNumEntries()
Get amount of entries for this group.

getStartDate()
Get the starting date of all the group's entries.

id
Public-facing descriptor, placed in the mapped class dictionary.

name
Public-facing descriptor, placed in the mapped class dictionary.

program
Public-facing descriptor, placed in the mapped class dictionary.

program_icon
Return the icon of the web history program used to create the file. See vizzieweb/convertors/__init__.py for details.

program_name
Return the full name of the web history program used to create the file. See vizzieweb/convertors/__init__.py for details.

class Entry(**vars*)
Class that stores a particular row in the file.

id entry id

type entry type e.g. URL

access_date entry access date

access_time entry access time

modified_date entry modified date

modified_time entry modified time

url entry url

filename entry filename on disk

directory entry folder on disk

http_headers entry url http headers

title entry url title

deleted whether entry was originally deleted or not

content_type type of content, usually only index.dat files store this

group group object this entry belongs to

parsedurl url object this entry has

browser browser object this entry belong to

access_date
Public-facing descriptor, placed in the mapped class dictionary.

access_time
Public-facing descriptor, placed in the mapped class dictionary.

access_time_str
A property for access_time_str. Returns the date in a normalised format.

static **averagePages**()
Returns the average number of websites visited in one day, i.e. only http and http entries.

browser
Public-facing descriptor, placed in the mapped class dictionary.

browser_id
Public-facing descriptor, placed in the mapped class dictionary.

browsername
A property for browsername.

content_type
Public-facing descriptor, placed in the mapped class dictionary.

deleted
Public-facing descriptor, placed in the mapped class dictionary.

directory
Public-facing descriptor, placed in the mapped class dictionary.

filename
Public-facing descriptor, placed in the mapped class dictionary.

static **filesAccessed**()
Returns a file directory tree and the total amount of files accessed. The tree is a dictionary of drives, each drive letter is the key and the value is another dictionary with the keys being file types. For each file type, the value is a tree structure with the end point being a file name, the number of times the file was accessed and a list of access dates.

Note: This does not allow for Linux etc drives, only Windows one letter drives.

ToDo: re-optimise this query, more things were done making it slow again for larger datasets.

ToDo: Allow other operating systems file systems.

static **generateHeatMap**()
Returns heatmap things for the overview: the table headers, the heatmap table and the highest and lowest values (used to calculate heatmap colour).

group
Public-facing descriptor, placed in the mapped class dictionary.

group_id
Public-facing descriptor, placed in the mapped class dictionary.

http_headers
Public-facing descriptor, placed in the mapped class dictionary.

id
Public-facing descriptor, placed in the mapped class dictionary.

modified_date
Public-facing descriptor, placed in the mapped class dictionary.

modified_time
Public-facing descriptor, placed in the mapped class dictionary.

modified_time_str
A property for modified_time_str. Returns the date in a normalised format.

static **peakTime**()
Returns the peak time of web browser usage. Similar to the heatmap, although doesn't do for each day of the week - sums up each hour and returns the highest.

timeline_date
A property for timeline_date. Returns the date in a normalised format for Flot.

title
Public-facing descriptor, placed in the mapped class dictionary.

type
Public-facing descriptor, placed in the mapped class dictionary.

url
Public-facing descriptor, placed in the mapped class dictionary.

class URL(*url=None*)
Class that stores a URL for an entry divided up into its different parts.

entry_id url id

scheme url scheme

netloc url network location

path url path

params url params

query url query strings

fragment url fragment

username url username

password url password

hostname url hostname

port url port

domain normalised hostname i.e without the www

search the search engine string (if any)

entry entry object. one-to-one join.

asDict()
Used for optimising adding URLs to database when converting files.

domain
Public-facing descriptor, placed in the mapped class dictionary.

entry
Public-facing descriptor, placed in the mapped class dictionary.

entry_id
Public-facing descriptor, placed in the mapped class dictionary.

fragment
Public-facing descriptor, placed in the mapped class dictionary.

static **getTop**(*num=100, highlight_funcs=, [], remove_funcs=, []*)
Get the top [amount] filtered URLS for a case. This has been optimised as one query with a subquery.

hostname
Public-facing descriptor, placed in the mapped class dictionary.

netloc
Public-facing descriptor, placed in the mapped class dictionary.

params
Public-facing descriptor, placed in the mapped class dictionary.

password
Public-facing descriptor, placed in the mapped class dictionary.

path
Public-facing descriptor, placed in the mapped class dictionary.

port
Public-facing descriptor, placed in the mapped class dictionary.

query
Public-facing descriptor, placed in the mapped class dictionary.

scheme
Public-facing descriptor, placed in the mapped class dictionary.

search
Public-facing descriptor, placed in the mapped class dictionary.

setDomain()
Removes www from front of any hostnames to normalise the hostnames and then tries to remove subdomains by working backwards to find the domain.

static **setDomain_manual**(*group*)
Manually set the domain names. Used in the optimised adding of URLs which bypasses creating URL objects using *__init__()*

urlstrip(*url*)
Some urls from IE appear like [username]@[url]. urlstrip() gets rid of the [username]@ bit. Returns the shortened URL

username
Public-facing descriptor, placed in the mapped class dictionary.

class Filter(*label, text, fillColor='#33A1C9'*)
Class that stores a filter object.

id filter id

fillcolor the color of flot's markers

label the name of the filter

text the long name of the filter, what the user sees

query pickled Filter object which stores the filter parts see model/filters.py

fillColor
Public-facing descriptor, placed in the mapped class dictionary.

id
Public-facing descriptor, placed in the mapped class dictionary.

label
Public-facing descriptor, placed in the mapped class dictionary.

static **onlineSearches**(*engine, highlight_funcs=, [], remove_funcs=, [], num=20*)
Given a search engine, amount and filters, returns a word cloud dictionary of search terms. The keys are the terms and the values are a (ratio, search phrases term was in, length of search phrases) tuple. Also returns total amount of terms, total number of unique terms and the smallest ratio (used to make the sizes of the words relative to smallest).

query
Public-facing descriptor, placed in the mapped class dictionary.

text
Public-facing descriptor, placed in the mapped class dictionary.

class SearchTerms(*term, engine, engine_long*)
Class that stores a search term.

id search term id

term the term

occurrence number of times term has appeared in searches

engine what search engine this term appeared in

engine_long full name of search engine

entries list of entries search term belongs to

engine
Public-facing descriptor, placed in the mapped class dictionary.

engine_long
Public-facing descriptor, placed in the mapped class dictionary.

entries
Public-facing descriptor, placed in the mapped class dictionary.

static **getTerms**(*query*)
Given the query part of a url, extracts the serach terms or phrases (those surrounded by quotes). Returns the search string and a list of terms in that string.

id
Public-facing descriptor, placed in the mapped class dictionary.

occurrence
Public-facing descriptor, placed in the mapped class dictionary.

term
Public-facing descriptor, placed in the mapped class dictionary.

get_plotable(*d, t, u, bn, bv, bs, p, ti*)
Returns (date, time, URL, browser name, browser version, browser source, program name, title) in a format Flot (the JavaScript timeline library) will understand. Only (date, time) needs to be sent in order for the graph to be displayed properly. The others are used for hover over and clickable text.

Note: Currently this function uses a hack to get British Summer Time dates to be displayed properly as GMT in Flot by adding an hour on to any dates after March 28th 2010.

ToDo: Fix this urgently!!

getFilter(*highlight_funcs=, [], remove_funcs=, []*)
Get the right filter. This is used to filter domain names and search terms correctly. *Case.filter_queries* is called, and if *highlight_funcs* != [] then return the highlighted query, otherwise return the non-highlighted query.

2.6.2 Filters

Filters

FilterQuery objects save the filter details and are stored pickled in the *Filter* class as *Filter.query* in *Database Models*.

class **FilterQuery**()
Stores the tables, attributes, operations and values of a query, e.g.

```
params = (Entry, title, Is, 'Test')
```

which equates to *WHERE Entry.title = 'Test'* in SQL.

FilterQuery objects get stored inside a *Filter* object in *Database Models*. The object stored inside the *Filter* object gets pickled thereby preserving the filter information.

add_element(*cls, attr, func, val, val_list*)
Add a new filter line to the *FilterQuery* object *params* list attribute.

query()
Called by *Case.filter_queries()* in *Database Models* to construct the filter clause. Returns an ANDed list of filters e.g. *Entry.title = "test" AND Entry.url <> "http://example.org"*

regexp(*expr, item*)
Enables regular expressions for filter queries.

2.6.3 Models

All the Python code to do with the database is kept here. This uses the SQLAlchemy ORM library. The underlying database is sqlite, but could very easily be modified.

2.7 Utility functions

Package Contents:

2.7.1 Utils

Useful variables and methods. Includes all the database connection functions and access to the configuration file.

Variables

ROOT_DIR the root directory of Webscavator.

CASE_FILE_DIR where case files are kept.

CONFIG_PATH where the config file is kept.

FILE_TYPES dictionary of file extensions and what type of file they are. Used in displaying file accesses in the 'files' tab.

session the sqlite session. Used to access the database. Useful methods include session.add(obj)', *session.flush()* and *session.commit()*.

config the config parser. To get an option do *config[section][option]*

Functions

bind(*db*)
This binds the database to the current session.

checkDebugging(*testing*)
If debug is set to *True* in the config file, a default database is loaded called *debug.db*. This is so that every time webscavator is restarted, there is a database pre-loaded. The Python web server automatically restarts when it detects a change to the code, so when debugging webscavator lots of restarts will happen.

connect(*dbfile*)
Given a database file, create an SQLAlchemy database engine which connects to the database.

getCases()
Gets the case names (.db files) stored in the case file directory.

getLists()
Gets the list names (.txt files) stored in the case lists directory.

init_database(*db*)
Initialises the database by creating all the tables in webscavator.models.models.py.

multidict_to_dict(*md*)
Turns a Werkzeug multidict into a dictionary with lists in entries which have more than one value.

setup(*testing=False*)
This is run when webscavator is started in *launch.py*. Finds the config file and loads it into the global variable config using *ConfigParser()*. Then runs *checkDebugging(testing)* with the config option for testing.

2.7.2 Utilities

All utility libraries should go in here.

2.8 Testing

Package Contents:

2.8.1 Testing Utilities

Some utilities for testing.

class FakeFileUpload(*filename, content_type, location*)
A fake File Upload to mimic uploading an actual file.

2.8.2 Functional Testing

Package Contents:

Case Controller Testing

General Controller Testing

Visual Controller Testing

2.8.3 Unit Testing

Package Contents:

Forms Testing

Validator Testing

Model Testing

This is where all the unit and functional tests are kept.

runTests(*unittests=True, functionaltests=True*)

Tests are run from *launch.py* by typing in the command line:

```
python launch.py runtests functional=True unit=True
```

CHAPTER

THREE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

L

W

INDEX

D

E

F

G

H

I

J

L

M

N

V

W

X