Burst Documentation

Release 0.5.10

scakemyer

Feb 24, 2019

Contents

A burst of providers.

CHAPTER 1

Features

- Fast, very fast
- No extra add-ons to install, all providers are included
- No extra service running in the background
- Easy settings to enable or disable providers and filters
- First-class support with Quasar, and only Quasar (don't even ask)
- Simple definitions-based architecture with overrides
- Clean, PEP8 compliant code

CHAPTER 2

Installation

Get the latest release from https://burst.surge.sh and *Install from zip* within Kodi, or install the add-on from the Quasar Repository if you already have Quasar installed.

CHAPTER 3

Topics

3.1 Using filters

If you go in the add-on's Advanced settings, you will notice an option named `Additional keyword filters (comma separated)`. Enabling this option will bring up three sub-settings: `Accept`, `Block` and `Require`. They all expect the same kind of parameters, which is a comma-separated list of keywords to respectively either accept, block or require. Although it's mostly self-explanatory, let's go over each of them to fully understand how they behave, and what kind of results you mind expect when using those settings.

3.1.1 Format

A comma-separated list is a standard way of defining multiple values. You can include spaces between keywords for readability, and Burst will work just the same. For example, those two settings will be equivalent: `HEVC,H265` vs `HEVC, H265`. They will both be understood as a list with the values `["HEVC", "H265"]`. Also note that uppercase or lowercase makes no difference, so both `hevc` and `HeVc` in a result name would also be considered a match.

The only special trick about the format of keywords is done by using underscores (_), which tell Burst to make sure there is a space, dot, dash, also an underscore, or other separator between your keyword and the other parts of the result's name. For example, if you want to match `ITA`, but not `italian`, you would use `_ITA_` as your keyword, which would match names like `A.Movie.2017.ITA.720p` but *not* `A.Movie.2017.Italian.720p`. A trailing underscore would also return a match, ie. `A.Movie.720p.ITA`. **Note that the 'Require' keyword treats underscores literally**, so using `_ITA_` in *Require* would only match names like `A.Movie_ITA_720p`.

3.1.2 Keyword types

Accept

The *Accept* setting will return results that include **any** of the keywords you specify. For example, `Italian, French` will return results that either include `italian` or `french`.

Block

The *Block* setting will block results that include **any** of the keywords you specify, and can be the most dangerous filter to use. For example, `ITA` would block every result that has `ita` anywhere in its name, regardless of delimiters like dots and dashes, so if you're looking for a movie named *My Inheritance*, you would get absolutely no result. For that reason, you should usually always add underscores around *Block* keywords to make sure there are delimiters around those keywords.

Require

The *Require* setting is also a dangerous filter to use, and will require **all** the keywords you specify to be included in the result names. For example, if you specify `ITA, _FR_`, you would only get results that include **both** `ITA` and `FR` (with delimiters), which will be very few if any. It can however be a very useful setting to get only results that include your preferred language.

3.2 Using overrides

Definitions are loaded from *burst/providers/definitions.py* starting with all the default providers in the *providers.json* file. You can override existing definitions using either an *overrides.py* or *overrides.json* file.

Using a Python file is deprecated but will remain supported.

Start by adding a new file named *overrides.json* in your userdata folder, ie. in *~/.kodi/userdata/addon_data/script.quasar.burst/overrides.json* and either paste the content of an existing provider or start with small overrides:

```
{
    'torlock': {
        'name': 'MyTorLock'
    }
}
```

If you are using the older Python format, put all your overrides in the *overrides* variable within that file, as such:

```
overrides = {
    'torlock': {
        'name': 'MyTorLock'
    }
}
```

3.3 Adding providers

Adding a custom provider is similar to using overrides, but using a JSON file for each of your providers, unless you add them all in your custom *overrides.json* file, which also works.

To add a provider, simply create a file with the `.json` extension under the `providers` folder in your userdata folder, ie. as `~/.kodi/userdata/addon_data/script.quasar.burst/providers/nice_provider.json`, and make sure it follows the format below (hopefully with `"subpage": false`):

```
{
    "1337x": {
        "name": "1337x",
```

(continues on next page)

(continued from previous page)

```
        "enabled": true,
        "private": false,
        "languages": "en",
        "anime_extra": "",
        "anime_keywords": "{title:original} {episode}",
        "anime_query": "EXTRA",
        "base_url": "https://www.1337x.to/search/QUERY/1/",
        "color": "FFF14E13",
        "general_extra": "",
        "general_keywords": "{title:original}",
        "general_query": "EXTRA",
        "language": null,
        "login_failed": "",
        "login_object": "",
        "login_path": null,
        "movie_extra": "",
        "movie_keywords": "{title:original} {year}",
        "movie_query": "EXTRA",
        "parser": {
            "infohash": "",
            "name": "item('a', order=2)",
            "peers": "item(tag='td', order=3)",
            "row": "find_once(tag='tbody').find_all('tr')",
            "seeds": "item(tag='td', order=2)",
            "size": "item(tag='td', order=5)",
            "torrent": "'https://www.1337x.to%s' % item(tag='a', attribute='href',␣
↪order=2)"
        },
        "season_extra": "",
        "season_extra2": "",
        "season_keywords": "{title:original} s{season:2}",
        "season_keywords2": "",
        "season_query": "EXTRA",
        "separator": "+",
        "show_query": "",
        "subpage": true,
        "tv_extra": "",
        "tv_extra2": "",
        "tv_keywords": "{title:original} s{season:2}e{episode:2}",
        "tv_keywords2": ""
    }
}
```

3.3.1 Provider fields

name

The provider's name as displayed to the user, typically with color.

color

The color of the provider name using Kodi's ARGB (alpha-red-green-blue) color format.

base_url

The `base_url` is the part of the provider's URL that is **always** found in your browser bar when you visit or more importantly, search the site. It may or may not contain the `QUERY` part (more on that later); it really only depends on the **common** part of the different search queries.

language

Forces a language preference for translations if they're available, eg. `es`

private

Boolean flag to mark this provider as private, see *PrivateProviders*.

separator

Space separator used in URL queries by this provider, typically `%20` for an encoded white-space or `+`

subpage

The most expensive boolean flag, to be avoided as much as possible. This tells Burst that we have no choice but to open **each and every** link to get to the torrent or magnet link. As it stands, we also waste the `torrent` (more on that later) definition under `parser`, which becomes the link to follow, and the page at that link gets automatically processed to find a magnet or torrent link in it.

*_query

Second part of the URL after `base_url` which will contain the `QUERY` keyword if it's not already in the `base_url`. This typically include category parameters specific to each provider, ie. `/movies/QUERY`

*_extra

The most confusing part of queries. Those will contain *extra* parameters, typically categories also, replacing the `EXTRA` keyword often found in the respective `*_query` definition, and often simply for the convenience of shorter `*_query` definitions. Note that this is mostly always just an empty string and not being used.

*_keywords

Keyword definitions for the different search types, with special placeholders like `{title}` for a movie or TV show title.

List of keyword types

- `{title}` Movie or TV show title
- `{year}` Release date, typically for movies only
- `{season}` Season number. Using `{season:2}` pads to 2 characters with leading zeros, eg. `s{season:2}` would become `s01` for an episode of season 1.

- `{episode}` Episode number, same formatting as `{season}` with regards to padding, ie. `{episode:2}`. Typically used with season as such: `s{season:2}e{episode:2}`

parser

This is the most important part of every provider, and tells Burst how to find torrents within search result pages. The first parser definition to be used is the `row`, and is also the "parent" to all to the others. It most usually ends with a `find_all('tr')`, and tells Burst which HTML tags, typically table rows, hold the results we're interested in. All other parser definitions will then look **within** each row for their respective information. Each other parser definition tells Burst what HTML tag has its information, for example `item(tag='td', order=1)` for `name` tells Burst that the torrent name is in the first table column of each row.

TODO: A more detailed description of parser fields and a tutorial on how to actually create providers will soon be added.

3.3.2 Private providers

login_path

The `login_path` is the part of the URL used for logging in, typically something like `"/login.php"`. This can be found by inspecting the login form's HTML and taking its `action` attribute.

login_object

The `login_object` represents the form elements sent to the `login_path`. For built-in private providers, placeholders are used to replace setting values for the username and password (`USERNAME` and `PASSWORD` respectively). Custom providers cannot define new settings, and must therefore put the username and password in the `login_object` directly.

login_failed

String that must **not** be included in the response's content. If this string is present in the page when trying to login, it returns as having failed and no search queries will be sent.

3.4 Providers by media type

Using overrides, you can enable or disable providers depending on the type of media they contain. A *type* field can be set for this.

If a provider should be used for all media types, simply **do not** set its *type* field, which is the default behavior.

Available types:

- *movies*
- *shows* as alias for both - *episodes* - *seasons*
- *anime*

```
{
    'torlock': {
        'type': 'movies'
    }
}
```

If you are using the older Python format:

```
overrides = {
    'torlock': {
        'type': 'movies'
    }
}
```

3.5 burst package

3.5.1 Subpackages

burst.parser package

Submodules

burst.parser.HTMLParser module

A parser for HTML and XHTML.

exception burst.parser.HTMLParser.**HTMLParseError**(*msg*, *position=(None*, *None)*)
Bases: exceptions.Exception

Exception raised for all parse errors.

class burst.parser.HTMLParser.**HTMLParser**
Bases: *burst.parser.markupbase.ParserBase*

Find tags and other markup and call handler functions.

Usage: p = HTMLParser() p.feed(data) . . . p.close()

Start tags are handled by calling self.handle_starttag() or self.handle_startendtag(); end tags by self.handle_endtag(). The data between tags is passed from the parser to the derived class by calling self.handle_data() with the data as argument (the data may be split up in arbitrary chunks). Entity references are passed by calling self.handle_entityref() with the entity reference as the argument. Numeric character references are passed to self.handle_charref() with the string containing the reference as the argument.

CDATA_CONTENT_ELEMENTS = ('script', 'style')

reset()
Reset this instance. Loses all unprocessed data.

feed(*data*)
Feed data to the parser.

Call this as often as you want, with as little or as much text as you want (may include 'n').

close()
Handle any buffered data.

error(*message*)

get_starttag_text()
Return full source of start tag: '<...>'.

set_cdata_mode(*elem*)

clear_cdata_mode()

goahead(*end*)

parse_html_declaration(*i*)

parse_bogus_comment(*i*, *report=1*)

parse_pi(*i*)

parse_starttag(*i*)

check_for_whole_start_tag(*i*)

parse_endtag(*i*)

handle_startendtag(*tag*, *attrs*)

handle_starttag(*tag*, *attrs*)

handle_endtag(*tag*)

handle_charref(*name*)

handle_entityref(*name*)

handle_data(*data*)

handle_comment(*data*)

handle_decl(*decl*)

handle_pi(*data*)

unknown_decl(*data*)

entitydefs = None

unescape(*s*)

burst.parser.ehp module

" All the credit of this code to Iury de oliveira gomes figueiredo Easy Html Parser is an AST generator for html/xml documents. You can easily delete/insert/extract tags in html/xml documents as well as look for patterns. https://github.com/iogf/ehp

class burst.parser.ehp.**Attribute**
Bases: dict

This class holds the tags's attributes. The idea consists in providing an efficient and flexible way of manipulating tags attributes inside the dom.

Example: dom = Html().feed(' foo ')

for ind in dom.sail(): if ind.name == 'p': ind.attr['style'] = "color:blue"

It would change to color blue.

class burst.parser.ehp.**Root**(*name=None*, *attr=None*)

Bases: *list*

A Root instance is the outmost node for a xml/html document. All xml/html entities inherit from this class.

html = Html() dom = html.feed('<html> ... </body>')

dom.name == '' True type(dom) == Root True

sail()

This is used to navigate through the xml/html document. Every xml/html object is represented by a python class instance that inherits from Root.

The method sail is used to return an iterator for these objects.

Example: data = '<a>   '

html = Html() dom = html.feed(data)

for ind in dom.sail(): print type(ind),',', ind.name

It would output.

<class 'ehp.Root'> , a <class 'ehp.Root'> , b

index(*item*, ***kwargs*)

This is similar to index but uses id to check for equality.

Example:

data = '<a>' html = Html() dom = html.feed(data)

for root, ind in dom.sail_with_root(): print root.name, ind.name, root.index(ind)

It would print.

a b 0 a b 1 a 0

The line where it appears ' a 0' corresponds to the outmost object. The outmost object is an instance of Root that contains all the other objects. :param item:

remove(*item*)

This is as list.remove but works with id.

data = '<a>' html = Html() dom = html.feed(data) for root, ind in dom.sail_with_root(): if ind.name == 'b': root.remove(ind)

print dom

It should print.

<a >

find(*name=''*, *every=1*, *start=1*, **args*)

It is used to find all objects that match name.

Example 1:

data = '<a>' html = Html() dom = html.feed(data)

for ind in dom.find('b'): print ind

It should print.

Example 2.

data = ‘<body> alpha.   beta. </body>’ html = Html() dom = html.feed(data)

for ind in dom.find(‘p’, (‘style’, ‘color:green’)): print ind

Or

for ind in dom.find(‘p’, (‘style’, [‘color:green’, ‘color:red’])): print ind

Output.

 beta.

find_once(*tag=None*, *select=None*, *order=1*)
” It returns the nth (order) ocurrence from the tag matching with the attributes from select

find_all(*tag=None*, *select=None*, *every=1*, *start=1*)
” It returns all ocurrences from the tag matching with the attributes from select

find_with_root(*name*, **args*)
Like Root.find but returns its parent tag.

from ehp import *

html = Html() dom = html.feed(‘’‘<body> alpha   beta </body>’‘’)

for root, ind in dom.find_with_root(‘p’): root.remove(ind)

print dom

It would output.

<body > </body>

by_id(*id_value*)
It is a shortcut for finding an object whose attribute ‘id’ matches id.

Example:

data = ‘<a>’ html = Html() dom = html.feed(data)

print dom.byid(‘foo’) print dom.byid(‘bar’)

It should print.

 None

take(**args*)
It returns the first object whose one of its attributes matches (key0, value0), (key1, value1), ...

Example:

data = ‘<a>’ html = Html() dom = html.feed(data)

print dom.take((‘id’, ‘foo’)) print dom.take((‘id’, ‘foo’), (‘size’, ‘2’))

take_with_root(**args*)
Like Root.take but returns the tag parent.

match(**args*)
It returns a sequence of objects whose attributes match. (key0, value0), (key1, value1), ...

Example:

data = ‘’ html = Html() dom = html.feed(data)

for ind in dom.match((‘size’, ‘1’)): print ind

It would print.

match_with_root (**args*)
Like Root.match but with its parent tag.

Example:

from ehp import *

html = Html() dom = html.feed(‘‘‘<body> xxx   mmm </body>’‘’)

for root, ind in dom.match_with_root((‘style’, ‘color:black’)): del ind.attr[‘style’]

item = dom.fst(‘body’) item.attr[‘style’] = ‘color:black’

print dom

Output.

<body style=”color:black” > xxx   mmm </body>

join (*delim*, **args*)
It joins all the objects whose name appears in args.

Example 1:

html = Html() data = ‘<a> This is cool. That is. ’ dom = html.feed(data)

print dom.join(‘’, ‘b’) print type(dom.join(‘b’))

It would print.

 This is cool. That is. <type ‘str’>

Example 2:

html = Html() data = ‘<a> alpha<c>beta</c> gamma’ dom = html.feed(data)

print dom.join(‘’, ‘b’, ‘c’)

It would print.

 alpha<c >beta</c>gamma

Example 3:

html = Html() data = ‘<a>alpha<c>beta</c>gamma’ dom = html.feed(data)

print dom.join(‘n’, DATA)

It would print.

alpha beta gamma

fst (*name*, **args*)
It returns the first object whose name matches.

Example 1:

html = Html() data = ‘<body> Cool. </body>’ dom = html.feed(data)

print dom.fst(‘em’)

It outputs.

 Cool.

Example 2:

data = '<body> alpha.   beta. </body>' html = Html() dom = html.feed(data)

for ind in dom.find('p', ('style', 'color:green')): print ind

print dom.fst('p', ('style', 'color:green')) print dom.fst_with_root('p', ('style', 'color:green'))

Output:

 beta. beta. (<ehp.Tag object at 0xb7216c0c>, <ehp.Tag object at 0xb7216d24>)

fst_with_root (*name*, **args*)
Like fst but returns its item parent.

Example:

html = Html() data = '<body> Cool. </body>' dom = html.feed(data)

root, item dom.fst_with_root('em') root.insert_after(item, Tag('p')) print root

It outputs.

<body > Cool. </body>

For another similar example, see help(Root.fst)

text ()
It returns all objects whose name matches DATA. It basically returns a string corresponding to all asci characters that are inside a xml/html tag.

Example:

html = Html() data = '<body>This is all the text.</body>' dom = html.feed(data)

print dom.fst('em').text()

It outputs.

This is all the text.

Notice that if you call text() on an item with children then it returns all the *printable* characters for that node.

write (*filename*)
It saves the structure to a file.

sail_with_root ()
This one works like sail(), however it yields the tag's parents as well as the child tag.

For an example, see help(Root.remove).

walk ()
Like sail but carries name and attr.

Example:

html = Html() data = '<body> This is all the text.</body>' dom = html.feed(data)

for ind, name, attr in dom.walk(): print 'TAG:', ind print 'NAME:', name print 'ATTR:', attr

It should print.

TAG: NAME: 1 ATTR: TAG: This is all the text. NAME: 1 ATTR: TAG: This is all the text. NAME: em ATTR: TAG: <body > This is all the text.</body> NAME: body ATTR:

walk_with_root()
Like walk but carries root.

Example:

html = Html() data = '<body>alpha</body>' dom = html.feed(data)

for (root, name, attr), (ind, name, attr) in dom.walk_with_root(): print root, name, ind, name

Output:

alpha 1 alpha 1 <body >alpha</body> em alpha em <body >alpha</body> body <body >alpha</body> body

insert_after(*y*, *k*)
Insert after a given tag.

For an example, see help(Root.fst_with_root).

insert_before(*y*, *k*)
Insert before a given tag.

For a similar example, see help(Root.fst_with_root).

parent(*dom*)
Find the parent tag

list(*text=''*)

select(*text=''*)

get_attributes(*text*)

class burst.parser.ehp.**Tag**(*name*, *attr=None*)
Bases: *burst.parser.ehp.Root*

This class's instances represent xml/html tags under the form: <name key="value" ...> ... </name>.

It holds useful methods for parsing xml/html documents.

class burst.parser.ehp.**Data**(*data*)
Bases: *burst.parser.ehp.Root*

The pythonic representation of data that is inside xml/html documents.

All data that is not a xml/html token is represented by this class in the structure of the document.

Example:

html = Html() data = '<body>alpha</body>' dom = html.feed(data)

x = dom.fst('em')

x holds a Data instance.

type(x[0]) print x[0]

Output:

<class 'ehp.Data'> alpha

The Data instances are everywhere in the document, when the tokenizer finds them between the xml/html tags it builds up the structure identically to the document.

text()
It returns all objects whose name matches DATA. It basically returns a string corresponding to all asci characters that are inside a xml/html tag.

Example:

html = Html() data = '<body>This is all the text.</body>' dom = html.feed(data)

print dom.fst('em').text()

It outputs.

This is all the text.

Notice that if you call text() on an item with children then it returns all the *printable* characters for that node.

class burst.parser.ehp.**XTag**(*name*, *attr=None*)
Bases: *burst.parser.ehp.Root*

This tag is the representation of html's tags in XHTML style like It is tags which do not have children.

class burst.parser.ehp.**Meta**(*data*)
Bases: *burst.parser.ehp.Root*

class burst.parser.ehp.**Code**(*data*)
Bases: *burst.parser.ehp.Root*

class burst.parser.ehp.**Amp**(*data*)
Bases: *burst.parser.ehp.Root*

class burst.parser.ehp.**Pi**(*data*)
Bases: *burst.parser.ehp.Root*

class burst.parser.ehp.**Comment**(*data*)
Bases: *burst.parser.ehp.Root*

class burst.parser.ehp.**Tree**
Bases: object

The engine class.

clear()
Clear the outmost and stack for a new parsing.

last()
Return the last pointer which point to the actual tag scope.

nest(*name*, *attr*)
Nest a given tag at the bottom of the tree using the last stack's pointer.

dnest(*data*)
Nest the actual data onto the tree.

xnest(*name*, *attr*)
Nest a XTag onto the tree.

ynest(*data*)

mnest(*data*)

cnest(*data*)

rnest(*data*)

inest(*data*)

enclose(*name*)
When found a closing tag then pops the pointer's scope from the stack so pointing to the earlier scope's tag.

class burst.parser.ehp.**Html**
Bases: *burst.parser.HTMLParser.HTMLParser*

The tokenizer class.

fromfile(*filename*)
It builds a structure from a file.

feed(*data*)

Return type *Root*

handle_starttag(*name*, *attr*)
When found an opening tag then nest it onto the tree

handle_startendtag(*name*, *attr*)
When found a XHTML tag style then nest it up to the tree

handle_endtag(*name*)
When found a closing tag then makes it point to the right scope

handle_data(*data*)
Nest data onto the tree.

handle_decl(*decl*)

unknown_decl(*decl*)

handle_charref(*data*)

handle_entityref(*data*)

handle_pi(*data*)

handle_comment(*data*)

burst.parser.markupbase module

Shared support for scanning document type declarations in HTML and XHTML.

This module is used as a foundation for the HTMLParser and sgmllib modules (indirectly, for htmllib as well). It has no documented public API and should not be used directly.

class burst.parser.markupbase.**ParserBase**
Parser base class which provides some common support methods used by the SGML/HTML and XHTML parsers.

error(*message*)

reset()

getpos()
Return current line number and offset.

updatepos(*i*, *j*)

parse_declaration(*i*)

parse_marked_section(*i*, *report=1*)

parse_comment(*i*, *report=1*)

unknown_decl(*data*)

burst.providers package

Submodules

burst.providers.definitions module

3.5.2 Submodules

3.5.3 burst.burst module

3.5.4 burst.client module

3.5.5 burst.filtering module

3.5.6 burst.ordereddict module

class burst.ordereddict.**OrderedDict**(**args*, ***kwds*)

Bases: dict, UserDict.DictMixin

Backport of collections.OrderedDict for Python 2.6 (Kodi 16)

clear() → None. Remove all items from D.

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

keys() → list of D's keys

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

values() → list of D's values

items() → list of D's (key, value) pairs, as 2-tuples

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

iteritems() → an iterator over the (key, value) items of D

copy() → a shallow copy of D

classmethod fromkeys(*S*[, *v*]) → New dict with keys from S and values equal to v.
v defaults to None.

3.5.7 burst.provider module

3.5.8 burst.utils module

- modindex

CHAPTER 4

Credits

- @elgatito for all the updates with Elementum
- @mancuniancol for all his work on Magnetic, this add-on wouldn't have been possible without him.
- All the alpha and beta testers that led to the first stable release.

Python Module Index

b

Index

V

W

X

Y