WhooshES

Release 0.1+0.gc587f0e.dirty

unknown

Feb 03, 2022

CONTENTS

WhooshES is a toolkit for working with Whoosh and Elasticsearch using a standard API.

CHAPTER

ONE

INSTALLATION

This package can be installed from the official PyPI:

```
$ pip install whooshes
```

CHAPTER

TWO

DOCUMENTATION

The documentation can be found on Read the Docs.

2.1 User Guide (WIP)

2.2 API

2.2.1 Elasticsearch

This package contains the tools for providing compatibility between Whoosh and Elasticsearch.

class whooshes.elasticsearch.**ESQueryTransformer**(*schema: whoosh.fields.Schema*)

Query transformer implementation for Elasticsearch.

transform(*query: whoosh.query.qcore.Query*) → Dict[str, Any]

Transform a given query object as necessary for the target backend.

get_method(*tquery: Type[whoosh.query.qcore.Query]*) → Callable[[whoosh.query.qcore.Query], whooshes.transformer.T]

Get the particular method which should handle transforming the given query type. This method has a per-instance LRU cache applied during initialization to minimize overhead.

2.2.2 Whoosh Extensions

This package provides various extensions to the Whoosh library, such as new fields and analyzers, that may be convenient when interacting with Whoosh.

Note: While these aren't for providing compatibility with Elasticsearch and beyond, I didn't feel that they warranted their own package and repo, so here they are.

---

whooshes.ext.**SeparatedTokenizer**(*chars: str*)

Splits tokens by given characters.

Note that the tokenizer calls unicode.strip() on each match of the regular expression.

```
>>> cst = SeparatedTokenizer(';')
>>> [token.text for token in cst("hi there; what's ; up")]
['hi there', "what's", 'up']
```

class whooshes.ext.**UppercaseFilter**

Uses str.upper() to lowercase token text.

```
>>> from whoosh.analysis.tokenizers import RegexTokenizer
>>> rext = RegexTokenizer()
>>> stream = rext("This is a TEST")
>>> [token.text for token in UppercaseFilter()(stream)]
['THIS', 'IS', 'A', 'TEST']
```

class whooshes.ext.**SEPKEYWORD**(*sep: str*, *stored: bool = False*, *lowercase: bool = False*, *scorable: bool = False*, *unique: bool = False*, *field_boost: float = 1.0*, *sortable: bool = False*, *vector: Optional[Union[whoosh.formats.Format, bool]] = None*)

Whoosh field type for fields containing seperated keyword-like data. This behaves like the built-in keyword, except that the separator can be specified with the `sep` parameter.

Parameters

- **stored** – Whether to store the value of the field with the document.
- **commas** – Whether this is a comma-separated field. If this is False (the default), it is treated as a space-separated field.
- **scorable** – Whether this field is scorable.

clean()

Clears any cached information in the field and any child objects.

index(*value*, ***kwargs*)

Returns an iterator of (btext, frequency, weight, encoded_value) tuples for each unique word in the input value.

The default implementation uses the `analyzer` attribute to tokenize the value into strings, then encodes them into bytes using UTF-8.

parse_query(*fieldname*, *qstring*, *boost=1.0*)

When `self_parsing()` returns True, the query parser will call this method to parse basic query text.

parse_range(*fieldname*, *start*, *end*, *startexcl*, *endexcl*, *boost=1.0*)

When `self_parsing()` returns True, the query parser will call this method to parse range query text. If this method returns None instead of a query object, the parser will fall back to parsing the start and end terms using process_text().

process_text(*qstring*, *mode=''*, ***kwargs*)

Analyzes the given string and returns an iterator of token texts.

```
>>> field = fields.TEXT()
>>> list(field.process_text("The ides of March"))
["ides", "march"]
```

self_parsing()

Subclasses should override this method to return True if they want the query parser to call the field's `parse_query()` method instead of running the analyzer on text in this field. This is useful where the field needs full control over how queries are interpreted, such as in the numeric field type.

separate_spelling()

Returns True if the field stores unstemmed words in a separate field for spelling suggestions.

sortable_terms(*ixreader*, *fieldname*)

Returns an iterator of the "sortable" tokens in the given reader and field. These values can be used for sorting. The default implementation simply returns all tokens in the field.

This can be overridden by field types such as NUMERIC where some values in a field are not useful for sorting.

spellable_words(*value*)
Returns an iterator of each unique word (in sorted order) in the input value, suitable for inclusion in the field's word graph.

The default behavior is to call the field analyzer with the keyword argument `no_morph=True`, which should make the analyzer skip any morphological transformation filters (e.g. stemming) to preserve the original form of the words. Exotic field types may need to override this behavior.

spelling_fieldname(*fieldname*)
Returns the name of a field to use for spelling suggestions instead of this field.

Parameters **fieldname** – the name of this field.

subfields()
Returns an iterator of `(name_prefix, fieldobject)` pairs for the fields that need to be indexed when content is put in this field. The default implementation simply yields `("", self)`.

supports(*name*)
Returns True if the underlying format supports the given posting value type.

```
>>> field = TEXT()
>>> field.supports("positions")
True
>>> field.supports("chars")
False
```

to_bytes(*value*)
Returns a bytes representation of the given value, appropriate to be written to disk. The default implementation assumes a unicode value and encodes it using UTF-8.

to_column_value(*value*)
Returns an object suitable to be inserted into the document values column for this field. The default implementation simply calls `self.to_bytes(value)`.

tokenize(*value*, ***kwargs*)
Analyzes the given string and returns an iterator of Token objects (note: for performance reasons, actually the same token yielded over and over with different attributes).

2.3 License

HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PYTHON MODULE INDEX

W

INDEX