WebGrid

Release 0.5.6

unknown

May 12, 2023

CONTENTS

WebGrid is a datagrid library for Flask and other Python web frameworks designed to work with SQLAlchemy ORM entities and queries.

With a grid configured from one or more entities, WebGrid provides these features for reporting:

- Automated SQL query construction based on specified columns and query join/filter/sort options
- Renderers to various targets/formats
  - HTML output paired with JS (jQuery) for dynamic features
  - Excel (XLSX)
  - CSV
- User-controlled data filters
  - Per-column selection of filter operator and value(s)
  - Generic single-entry search
- Session storage/retrieval of selected filter options, sorting, and paging

Table of Contents

CHAPTER

ONE

GETTING STARTED

1.1 Installation

Install using *pip*:

```
pip install webgrid
```

Some basic internationalization features are available via extra requirements:

```
pip install webgrid[i18n]
```

1.2 Manager

Because WebGrid is generally framework-agnostic, a number of features are segmented into a grid framework manager. These include items like request object, session, serving files, etc.:

```
class Grid(webgrid.BaseGrid):
    manager = webgrid.flask.WebGrid()
```

The above may be specified once in the application and used as a base class for all grids.

Depending on the framework, setting up the manager also requires some additional steps to integrate with the application.

1.2.1 Flask Integration

In Flask, two integrations are necessary:

- Set up the connection to SQLAlchemy
- Integrate WebGrid with the Flask application

For a Flask app using Flask-SQLAlchemy for database connection/session management, the grids may use the same object that the rest of the app uses for query/data access. As an extension, the grid manager will register a blueprint on the Flask app to serve static files.

The following is an example of a minimal Flask app that is then integrated with WebGrid:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
import webgrid

class Grid(webgrid.BaseGrid):
    """Common base grid class for the application."""
    manager = webgrid.flask.WebGrid()

# Minimal Flask app setup
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)

# Integrate WebGrid as an extension
Grid.manager.init_db(db)
Grid.manager.init_app(app)
```

1.3 Basic Grid

Once the application's main *Grid* class has been defined with the appropriate manager, app grids may be created:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name)
    Column('Age', entities.Person.age)
    Column('Location', entities.Person.city)
```

The options available for setting up grids is truly massive. For starters, some good places to begin would be:

- *General Column Usage* for ways to configure the column layout
- *Grid Class* for grid options and query setup

1.4 Templates

If WebGrid is being used in an HTML context, some template inclusions must be made to support the header features, default styling, etc. The following will assume a Flask app with a Jinja template environment; customize to the needed framework and application.

Note, with the Flask grid manager, static assets are available through the blueprint the manager adds to the application.

1.4.1 CSS

Two CSS sheets are required and may be included as follows:

```
<link href="{{url_for('webgrid.static', filename='webgrid.css')}}" rel="stylesheet"␣
↪media="screen">
<link href="{{url_for('webgrid.static', filename='multiple-select.css')}}" rel=
↪"stylesheet" media="screen">
```

1.4.2 JS

WebGrid requires jQuery to be available. In addition, two JS assets are needed:

```
<script src="{{url_for('webgrid.static', filename='jquery.multiple.select.js')}}"></
↪script>
<script src="{{url_for('webgrid.static', filename='webgrid.js')}}"></script>
```

1.4.3 Rendering

Once the templates have all of the required assets included, rendering the grids themselves is fairly basic:

```
{{ grid.html() | safe}}
```

The *safe* filter is important for Jinja environments where auto-escape is enabled, which is the recommended configuration. The grid renderer output contains HTML markup and so must be directly inserted.

1.5 Internationalization

WebGrid supports *Babel*-style internationalization of text strings through the *morphi* library. To use this feature, specify the extra requirements on install:

```
pip install webgrid[i18n]
```

Currently, English (default) and Spanish are the supported languages in the UI.

CHAPTER

TWO

HELPFUL LINKS

- https://www.gnu.org/software/gettext/manual/html_node/Mark-Keywords.html
- https://www.gnu.org/software/gettext/manual/html_node/Preparing-Strings.html

CHAPTER

THREE

MESSAGE MANAGEMENT

The `setup.cfg` file is configured to handle the standard message extraction commands. For ease of development and ensuring that all marked strings have translations, a tox environment is defined for testing i18n. This will run commands to update and compile the catalogs, and specify any strings which need to be added.

The desired workflow here is to run tox, update strings in the PO files as necessary, run tox again (until it passes), and then commit the changes to the catalog files.

```
tox -e i18n
```

CHAPTER

FOUR

GRID CLASS

class webgrid.**BaseGrid**(*ident=None*, *per_page=<class 'webgrid._None'>*, *on_page=<class 'webgrid._None'>*, *qs_prefix=''*, *class_='datagrid'*, ***kwargs*)

WebGrid grid base class.

Handles class declarative-style grid description of columns, filterers, and rendering.

The constructor is responsible for:

- setting initial attributes
- initializing renderers
- setting up columns for the grid instance
- running the grid's *post_init* method

Args:

ident (str, optional): Identifier value for ident instance property. Defaults to None.

per_page (int, optional): Default number of records per page. Defaults to _None.

on_page (int, optional): Default starting page. Defaults to _None.

qs_prefix (str, optional): Arg name prefix to apply in query string. Useful for having multiple unconnected grids on a single page. Defaults to ''.

class_ (str, optional): CSS class name for main grid div. Defaults to 'datagrid'.

Class Attributes:

identifier (str): Optional string identifier used for the ident property.

sorter_on (bool): Enable HTML sorting UX. Default True.

pager_on (bool): Enable record limits in queries and HTML pager UX. Default True.

per_page (int): Default number of records per page, can be overridden in constructor or through query string args. Default 50.

on_page (int): Default page number, can be overridden in constructor or through query string args. Default 1.

hide_controls_box (bool): Hides HTML filter/page/sort/count UX. Default False.

session_on (bool): Enable web context session storage of grid filter/page/sort args. Default True.

subtotals (string): Enable subtotals. Can be none|page|grand|all. Default "none".

manager (Manager): Framework plugin for the web context, such as webgrid.flask.WebGrid.

allowed_export_targets (dict[str, Renderer]): Map non-HTML export targets to the Renderer classes.

enable_search (bool): Enable single-search UX. Default True.

unconfirmed_export_limit (int): Ask for confirmation before exporting more than this many records. Set to None to disable. Default 10000.

query_select_from (selectable): Entity, table, or other selectable(s) to use as the query from. If attributes like query_filter are used along with select_from, SQLAlchemy may require the select_from to precede the filtering.

query_joins (tuple): Tuple of joins to bring the query together for all columns. May have just the join object, or also conditions. e.g. [Blog], ([Blog.category],), or [(Blog, Blog.active == sa.true())] Note, relationship attributes must be referenced within tuples, due to SQLAlchemy magic.

query_outer_joins (tuple): Tuple of outer joins. See query_joins.

query_filter (tuple): Filter parameter(s) tuple to be used on the query. Note, relationship attributes must be referenced within tuples, due to SQLAlchemy magic.

query_default_sort (tuple): Parameter(s) tuple to be passed to order_by if sort options are not set on the grid. Note, relationship attributes must be referenced within tuples, due to SQLAlchemy magic.

apply_qs_args(*add_user_warnings=True*, *grid_args=None*)

Process args from manager for filter/page/sort/export.

Args:

add_user_warnings (bool, optional): Add flash messages for warnings. Defaults to True. grid_args (MultiDict, optional): Supply args directly to the grid.

apply_search(*query*, *value*)

Modify the query by applying a filter term constructed from search clauses.

Calls each filter search expression factory with the search value to get a search clause, then ORs them all together for the main query.

Args:

query (Query): SQLAlchemy query. value (str): Search value.

Returns:

Query: SQLAlchemy query

apply_validator(*validator*, *value*, *qs_arg_key*)

Apply a webgrid validator to value, and produce a warning if invalid.

Args:

validator (Validator): webgrid validator. value (str): Value to validate. qs_arg_key (str): Arg name to include in warning if value is invalid.

Returns:

Any: Output of *validator.to_python(value)*, or *None* if invalid.

before_query_hook()

Hook to give subclasses a chance to change things before executing the query.

build(*grid_args=None*)

Apply query args, run *before_query_hook*, and execute a record count query.

Calling *build* is preferred to simply calling *apply_qs_args* in a view. Otherwise, AttributeErrors can be hidden when the grid is used in Jinja templates.

build_qs_args(*include_session=False*)

Build a URL query string based on current grid attributes.

This is designed to be framework-agnostic and not require a request context. Usually the result would be used in a background task or similar (i.e. outside the flow of the rendered grid), so typically the session key is unnecessary.

Args:
include_session (bool, optional): Include session_key in the string. Defaults to False.

build_query(*for_count=False*)

Constructs, but does not execute, a grid query from columns and configuration.

This is the query the grid functions trust for results for records, count, page count, etc. Customization of the query should happen here or in the methods called within.

Build sequence: - *query_base* - *query_prep* - *query_filters* - *query_sort* - *query_paging*

Args:
for_count (bool, optional): Excludes sort/page from query. Defaults to False.

Returns:
Query: SQLAlchemy query object

can_search()

Grid *enable_search* attr turns on search, but check if there are supporting filters.

Returns:
bool: search enabled and supporting filters exist

check_auth()

For API usage, provides a hook for grids to specify authorization that should be applied for the API responder method.

If a 40* response is warranted, take that action here.

Note, this method is not part of normal grid/render operation. It will only be executed if run by a calling layer, such as the Flask WebGridAPI manager/extension.

clear_record_cache(*preserve_count=False*)

Reset records and record count cached from previous queries.

Args:
preserve_count (bool): Direct grid to retain count of records, effectively removing only the table of records itself.

column(*ident*)

Retrieve a grid column instance via either the key string or index int.

Args:
ident (Union[str, int]): Key/index for lookup.

Returns:
Column: Instance column matching the ident.

Raises:
KeyError when ident is a string not matching any column.

IndexError when ident is an int but out of bounds for the grid.

`export_as_response`(*wb=None*, *sheet_name=None*)

Return renderer response for view layer to provide as a file.

Args:
wb (Workbook, optional): XlsxWriter Workbook. Defaults to None. sheet_name (Worksheet, optional): XlsxWriter Worksheet. Defaults to None.

Raises:
ValueError: No export parameter given.

Returns:
Response: Return response processed through renderer and manager.

`get_unique_column_key`(*key*)

Apply numeric suffix to a field key to make the key unique to the grid.

Helpful for when multiple entities are represented in grid columns but have the same field names.

For instance, Blog.label and Author.label both have the field name *label*. The first column will have the *label* key, and the second will get *label_1*.

Args:
key (str): field key to make unique.

Returns:
str: unique key that may be assigned in the grid's *key_column_map*.

`property grand_totals`

Executes query to retrieve subtotals for the filtered query.

For grand totals to be queried/returned, the grid's *subtotals* must be grand/all and one or more columns must have subtotals configured.

A single result record is returned, which will have fields corresponding to all of the grid columns (same as a record returned in the general records query).

Returns:
Any: Single result record, or None if grand totals are not configured.

`has_column`(*ident*)

Verify string key or int index is defined for the grid instance.

Args:
ident (Union[str, int]): Key/index for lookup.

Returns:
bool: Indicates whether key/index is in the grid columns.

`property has_filters`

Indicates whether filters will be applied in *build_query*.

Returns:
bool: True if filter(s) have op/value set or single search value is given.

`property has_sort`

Indicates whether ordering will be applied in *build_query*.

Returns:
bool: True if grid's *order_by* list is populated.

`iter_columns`(*render_type*)

Generator yielding columns that are visible and enabled for target *render_type*.

Args:
render_type (str): [description]

Yields:
Column: Grid instance's column instance that is renderable for *render_type*.

`property page_count`

Page count, or 1 if no *per_page* is set.

`property page_totals`

Executes query to retrieve subtotals for the filtered query on the current page.

For page totals to be queried/returned, the grid's *subtotals* must be page/all and one or more columns must have subtotals configured.

A single result record is returned, which will have fields corresponding to all of the grid columns (same as a record returned in the general records query).

Returns:
Any: Single result record, or None if page totals are not configured.

`post_init`()

Provided for subclasses to run post-initialization customizations.

`prefix_qs_arg_key`(*key*)

Given a bare arg key, return the prefixed version that will actually be in the request.

This is necessary for render targets that will construct ensuing requests. Prefixing is not needed for incoming args on internal grid ops, as long as the grid manager's args loaders sanitize the args properly.

Args:
key (str): Bare arg key.

Returns:
str: Prefixed arg key.

`query_base`(*has_sort*, *has_filters*)

Construct a query from grid columns, using grid's join/filter/sort attributes.

Used by *build_query* to establish the basic query from column spec. If query is to be modified, it is recommended to do so in *query_prep* if possible, rather than overriding *query_base*.

Args:
has_sort (bool): Tells method not to order query, since the grid has sort params. has_filters (bool): Tells method if grid has filter params. Not used.

Returns:
Query: SQLAlchemy query

`query_filters`(*query*)

Modify the query by applying filter terms.

Called by *build_query* to apply any column filters as needed. Also enacts the single-search value if specified.

Args:
query (Query): SQLAlchemy query object.

Returns:
Query: SQLAlchemy query

`query_paging`(*query*)

Modify the query by applying limit/offset to match grid parameters.

Args:
query (Query): SQLAlchemy query.

Returns:
Query: SQLAlchemy query

`query_prep`(*query*, *has_sort*, *has_filters*)

Modify the query that was constructed in *query_base*.

Joins, query filtering, and default sorting can be applied via grid attributes. However, sometimes grid queries need columns added, instance-time modifications applied, etc.

Called by *build_query*.

Args:
query (Query): SQLAlchemy query object. has_sort (bool): Tells method grid has sort params defined. has_filters (bool): Tells method if grid has filter params.

Returns:
Query: SQLAlchemy query

`query_sort`(*query*)

Modify the query by applying sort to match grid parameters.

Args:
query (Query): SQLAlchemy query.

Returns:
Query: SQLAlchemy query

`property record_count`

Count of records for current filtered query.

Value is cached to prevent duplicate query execution. Methods changing the query (e.g. *set_filter*) will reset the cached value.

Returns:
int: Count of records.

`property records`

Records returned for current filtered/sorted/paged query.

Result is cached to prevent duplicate query execution. Methods changing the query (e.g. *set_filter*) will reset the cached result.

Returns:
list(Any): Result records from SQLAlchemy query.

`property search_expression_generators`

Get single-search query modifier factories from the grid filters.

Raises:
Exception: filter's *get_search_expr* did not return None or callable

Returns:
tuple(callable): search expression callables from grid filters

property search_uses_aggregate

Determine whether search should use aggregate filtering.

By default, only use the HAVING clause if all search-enabled filters are marked as aggregate. Otherwise, we'd be requiring all grid columns to be in query grouping. If there are filters for search that are not aggregate, the grid will only search on the non-aggregate columns.

Returns:
bool: search aggregate usage determined from filter info

set_column_order(*column_keys*)

Most renderers output columns in the order they appear in the grid's `columns` list. When bringing mixins together or subclassing a grid, however, the order is often not what is intended.

This method allows a manual override of column order, based on keys.

set_export_to(*to*)

Set export parameter after validating it exists in known targets.

Args:
to (str): Renderer attribute if it is known. Invalid value ignored.

set_filter(*key*, *op*, *value*, *value2=None*)

Set filter parameters on a column's filter. Resets record cache.

Args:
key (str): Column identifier op (str): Operator value (Any): First filter value value2 (Any, optional): Second filter value if applicable. Defaults to None.

set_paging(*per_page*, *on_page*)

Set paging parameters for the main query. Resets record cache.

Args:
per_page (int): Record limit for each page. on_page (int): With *per_page*, computes the offset.

set_records(*records*)

Assign a set of records to the grid's cache.

Useful for simple grids that simply need to be rendered as a table. Note that any ops performed on the grid, such as setting filter/sort/page options, will clear this cached information.

Args:
records (list(Any)): List of record objects that can be referenced for column data.

set_renderers()

Renderers assigned as attributes on the grid instance, named by render target.

set_sort(**args*)

Set sort parameters for main query. Resets record cache.

If keys are passed in that do not belong to this grid, raise user warnings (not exceptions). These warnings are suppressed if the grid has a "foreign" session assigned (i.e. two grids share some of the same columns, and should load as much information as possible from the shared session key).

Args:
Each arg is expected to be a column key. If the sort is to be descending for that key, prepend with a "-". E.g. *grid.set_sort('author', '-post_date')*

CHAPTER

FIVE

FRAMEWORK MANAGERS

One design goal of the base grid class is that it be essentially framework-agnostic. That is, the grid, by itself, should not care if it is being run in Flask, BlazeWeb, or another web app framework. As long as it has a connection to the framework that provides required items with a consistent interface, the grid should interact with the framework through that connection.

Wrapped features available through the manager:

- SQLAlchemy connection and queries
- Request
- Session storage
- Flash messages
- File export in response

class webgrid.flask.**WebGrid**(*db=None*, *jinja_loader=None*, *args_loaders=None*, *session_max_hours=None*, *blueprint_name=None*, *blueprint_class=None*)

Grid manager for connecting grids to Flask webapps.

Manager is a Flask extension, and may be bound to an app via `init_app`.

Instance should be assigned to the manager attribute of a grid class:

```
class MyGrid(BaseGrid):
    manager = WebGrid()
```

Args:

db (flask_sqlalchemy.SQLAlchemy, optional): Database instance. Defaults to None. If db is not supplied here, it can be set via `init_db` later.

Class Attributes:

jinja_loader (jinja.Loader): Template loader to use for HTML rendering.

args_loaders (ArgsLoader[]): Iterable of classes to use for loading grid args, in order of priority

session_max_hours (int): Hours to hold a given grid session in storage. Set to None to disable. Default 12.

blueprint_name (string): Identifier to use for the Flask blueprint on this extension. Default "webgrid". Needs to be unique if multiple managers are initialized as flask extensions.

blueprint_class

alias of `Blueprint`

csrf_token()

Return a CSRF token for POST.

file_as_response(*data_stream*, *file_name*, *mime_type*)

Return response from framework for sending a file.

flash_message(*category*, *message*)

Add a flash message through the framework.

init_app(*app*)

Register a blueprint for webgrid assets, and configure jinja templates.

init_blueprint(*app*)

Create a blueprint for webgrid assets.

init_db(*db*)

Set the db connector.

persist_web_session()

Some frameworks require an additional step to persist session data.

request()

Return request.

request_form_args()

Return POST request args.

request_json()

Return json body of request.

request_url_args()

Return GET request args.

sa_query(**args*, ***kwargs*)

Wrap SQLAlchemy query instantiation.

static_url(*url_tail*)

Construct static URL from webgrid blueprint.

test_request_context(*url='/'*)

Get request context for tests.

web_session()

Return current session.

CHAPTER

SIX

ARGUMENTS LOADERS

Grid arguments are run-time configuration for a grid instance. This includes filter operator/values, sort terms, search, paging, session key, etc.

Arguments may be provided to the grid directly, or else it pulls them from the assigned framework manager. The most common use case will use the manager.

6.1 Managed arguments

The grid manager uses "args loaders" (subclasses of `ArgsLoader`) to supply grid configuration. These loaders each represent a source of configuration. For instance, a loader can pull args from the GET query string, a POSTed form, etc.

The first loader on the list gets a blank MultiDict as input. Then, results from each loader are chained to the next one on the list. Each loader may accept or override the values from the previous output. The last loader gets the final word on configuration sent to the grid.

The default setup provides request URL arguments to the first loader, and then applies session information as needed. Some cases where you might want to do something different from the default: - The grid has options filters with a large number of options to select - The grid has a lot of complexity that would be cleaner as POSTs rather than GETs

To use managed arguments with the default loaders, simply call `apply_qs_args` or `build` to have the grid load these for use in queries and rendering:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name)
    Column('Age', entities.Person.age)
    Column('Location', entities.Person.city)

grid = PeopleGrid()
grid.apply_qs_args()
```

Customizing the loader list on a managed grid requires setting the `args_loaders` iterable on the manager. This can be set as a class attribute or provided in the manager's constructor.

As a class attribute:

```
from webgrid import BaseGrid
from webgrid.extensions import RequestArgsLoader, RequestFormLoader, WebSessionArgsLoader
from webgrid.flask import WebGrid

class GridManager(WebGrid):
```

(continues on next page)

(continued from previous page)

```
    args_loaders = (
        RequestArgsLoader,    # part of the default, takes args from URL query string
        RequestFormLoader,    # use args present in the POSTed form
        WebSessionArgsLoader, # part of the default, but lower priority from the form␣
↪POST
    )

class Grid(BaseGrid):
    manager = GridManager()
```

Using the manager's constructor to customize the loader list:

```
from webgrid import BaseGrid
from webgrid.extensions import RequestArgsLoader, RequestFormLoader, WebSessionArgsLoader
from webgrid.flask import WebGrid

class Grid(BaseGrid):
    manager = WebGrid(
        args_loaders = (
            RequestArgsLoader,    # part of the default, takes args from URL query string
            RequestFormLoader,    # use args present in the POSTed form
            WebSessionArgsLoader, # part of the default, but lower priority from the␣
↪form POST
        )
    )
```

class webgrid.extensions.**ArgsLoader**(*manager*)

Base args loader class.

When a grid calls for its args, it requests them from the manager. The manager will have one or more args loaders to be run in order. Each loader fetches its args from the request, then ensuing loaders have the opportunity to modify or perform other operations as needed.

class webgrid.extensions.**RequestArgsLoader**(*manager*)

Simple args loader for web request.

Args are usually passed through directly from the request. If the grid has a query string prefix, the relevant args will be namespaced - sanitize them here and return the subset needed for the given grid.

In the reset case, ignore most args, and return only the reset flag and session key (if any).

class webgrid.extensions.**RequestFormLoader**(*manager*)

Simple form loader for web request.

Form values are usually passed through directly from the request. If the grid has a prefix, the relevant args will be namespaced - sanitize them here and return the subset needed for the given grid.

In the reset case, ignore most args, and return only the reset flag and session key (if any).

class webgrid.extensions.**RequestJsonLoader**(*manager*)

JSON loader for web request.

See *webgrid.types.GridSettings()* for the expected JSON structure. The parsed arguments are converted to the querystring arg format and merged with any previous args.

class webgrid.extensions.**WebSessionArgsLoader**(*manager*)

Session manager for grid args.

Args are assumed to have been sanitized from the request already. But, args may be combined from the request and the session store for a number of cases.

- If session_override or no filter ops, load args from session store
- Having filter ops present will reset session store unless session_override is present
- Page/sort args will always take precedence over stored args, but not reset the store
- Export argument is handled outside of the session store

In the reset case, ignore most args, and return only the reset flag and session key (if any). And clear the session store for the given grid.

apply_session_overrides(*session_args*, *previous_args*)

Update session args as needed from the incoming request.

If session override case, wholesale update from the incoming request. This is useful if a single filter needs to be changed via the URL, but we don't want to dump the rest of the stored filters from the session.

Otherwise, apply only page/sort if available in the request.

Export directive is passed through from the request, so a session store never triggers an export by itself.

Args:
session_args (MultiDict): Args loaded from the session store. previous_args (MultiDict): Args that came into this args loader.

Returns:
MultiDict: Args to be used in grid operations.

args_have_op(*args*)

Check args for containing any filter operators.

Args:
args (MultiDict): Request args.

Returns:
bool: True if at least one op is present.

args_have_page(*args*)

Check args for containing any page args.

Args:
args (MultiDict): Request args.

Returns:
bool: True if at least one page arg is present.

args_have_sort(*args*)

Check args for containing any sort keys.

Args:
args (MultiDict): Request args.

Returns:
List[str]: all args matching as sort args.

cleanup_expired_sessions()

Remove sessions older than a certain number of hours.

Configurable at the manager level, with the session_max_hours attribute. If None, cleanup is disabled.

get_args(*grid*, *previous_args*)

Retrieve args from session and override as appropriate.

Submitting the header form flushes all args to the URL, so no need to load them from session.

If that is not the path, then we either have filtering args on the URL, or not. Default behavior is currently to consider filtering args to be comprehensive and authoritative, UNLESS a session_override arg is present.

The special session_override arg causes the store to overlay request args against an existing session, and return the combination.

Args:
grid (BaseGrid): Grid used to get default grid key (based on grid class name). previous_args (MultiDict): Incoming args, assumed to be sanitized already.

Returns:
MultiDict: Args to be used in grid operations.

get_session_store(*grid*, *args*)

Load args from session by session_key, and return as MultiDict.

Args:
grid (BaseGrid): Grid used to get default grid key (based on grid class name). args (MultiDict): Request args used for session key.

Returns:
MultiDict: Args to be used in grid operations.

save_session_store(*grid*, *args*)

Save the args in the session under the session key and as defaults for this grid.

Note, export and reset args are ignored for storage.

Args:
args (MultiDict): Request args to be loaded in next session store retrieval.

6.2 Supplying arguments directly

Arguments may be provided directly to *apply_qs_args* or *build* as a MultiDict. If arguments are supplied in this fashion, other sources are ignored:

```
from werkzeug.datastructures import MultiDict

class PeopleGrid(Grid):
    Column('Name', entities.Person.name)
    Column('Age', entities.Person.age)
    Column('Location', entities.Person.city)

grid = PeopleGrid()
grid.apply_qs_args(grid_args=MultiDict([
    ('op(name)', 'contains'),
    ('v1(name)', 'bill'),
]))
```

CHAPTER

SEVEN

TYPES

Types are defined for grid JSON input/output. These can be mirrored on the consumer side for API integrity (e.g. in TypeScript).

Typically, in API usage, the consumer app will be building/maintaining a `GridSettings` object to send to the API, and accepting a `Grid` in response.

7.1 Settings Types

class webgrid.types.**Filter**(*op: str*, *value1: Union[str, List[str]]*, *value2: Union[str, NoneType] = None*)

class webgrid.types.**Paging**(*pager_on: bool = False*, *per_page: Union[int, NoneType] = None*, *on_page: Union[int, NoneType] = None*)

class webgrid.types.**Sort**(*key: str*, *flag_desc: bool*)

class webgrid.types.**GridSettings**(*search_expr: Union[str, NoneType] = None*, *filters: Dict[str, webgrid.types.Filter] = <factory>*, *paging: webgrid.types.Paging = <factory>*, *sort: List[webgrid.types.Sort] = <factory>*, *export_to: Union[str, NoneType] = None*)

classmethod from_dict(*data: Dict[str, Any]*) → *GridSettings*

Create from deserialized json

to_args() → Dict[str, Any]

Convert grid parameters to request args format

7.2 Grid Types

class webgrid.types.**ColumnGroup**(*label: str*, *columns: List[str]*)

class webgrid.types.**FilterOperator**(*key: str*, *label: str*, *field_type: Union[str, NoneType]*, *hint: Union[str, NoneType] = None*)

class webgrid.types.**FilterOption**(*key: str*, *value: str*)

class webgrid.types.**FilterSpec**(*operators: List[webgrid.types.FilterOperator]*, *primary_op: Union[webgrid.types.FilterOperator, NoneType]*)

class webgrid.types.**GridSpec**(*columns: List[Dict[str, str]], column_groups: List[webgrid.types.ColumnGroup], column_types: List[Dict[str, str]], export_targets: List[str], enable_search: bool, enable_sort: bool, sortable_columns: List[str], filters: Dict[str, webgrid.types.FilterSpec] = <factory>*)

class webgrid.types.**GridState**(*page_count: int*, *record_count: int*, *warnings: List[str]*)

class webgrid.types.**Grid**(*settings:* webgrid.types.GridSettings, *spec:* webgrid.types.GridSpec, *state:* webgrid.types.GridState, *records: List[Dict[str, Any]]*, *totals: webgrid.types.GridTotals*, *errors: List[str]*)

CHAPTER

EIGHT

COLUMNS

8.1 Base Column and ColumnGroup

class webgrid.**Column**(**args*, ***kwargs*)

Column represents the data and render specification for a table column.

Args:

label (str): Label to use for filter/sort selection and table header.

key (Union[Expression, str], optional): Field key or SQLAlchemy expression. If an expression is provided, column attempts to derive a string key name from the expression. Defaults to None.

filter (FilterBase, optional): Filter class or instance. Defaults to None.

can_sort (bool, optional): Enables column for selection in sort keys. Defaults to True.

xls_num_format (str, optional): XLSX number/date format. Defaults to None.

render_in (Union(list(str), callable), optional): Targets to render as a column. Defaults to _None.

has_subtotal (Union(bool,str,callable), optional): Subtotal method to use, if any. True or "sum" will yield a sum total. "avg" maps to average. Can also be a callable that will be called with the aggregate expression and is expected to return a SQLAlchemy expression. Defaults to False.

visible (Union(bool, callable), optional): Enables any target in *render_in*. Defaults to True.

group (ColumnGroup, optional): Render grouping under a single heading. Defaults to None.

Class Attributes:

xls_width (float, optional): Override to autocalculated width in Excel exports.

xls_num_format (str, optional): Default numeric/date format type.

apply_sort(*query*, *flag_desc*)

Query modifier to enable sort for this column's expression.

extract_and_format_data(*record*)

Extract a value from the record for this column and run it through the data formaters.

extract_data(*record*)

Locate the data for this column in the record and return it.

format_data(*value*)

Use to adjust the value extracted from the record for this column. By default, no change is made. Useful in sub-classes.

new_instance(*grid*)

Create a "copy" instance that is linked to a grid instance.

Used during the grid instantiation process. Grid classes have column instances defining the grid structure. When the grid instantiates, we have to copy those column instances along with it, to attach them to the grid instance.

render(*render_type*, *record*, **args*, ***kwargs*)

Entrypoint from renderer.

Uses any renderer-specific overrides from the column, or else falls back to the output of *extract_and_format_data*.

Renderer-specific methods are expected to be named *render_<type>*, e.g. *render_html* or *render_xlsx*.

property render_in

Target(s) in which the field should be rendered as a column.

Can be set to a callable, which will be called with the column instance.

Returns:
tuple(str): Renderer identifiers.

property visible

Enables column to be rendered to any target in *render_in*.

Can be set to a callable, which will be called with the column instance.

Returns:
bool: Enable render.

xls_width_calc(*value*)

Calculate a width to use for an Excel renderer.

Defaults to the *xls_width* attribute, if it is set to a non-zero value. Otherwise, use the length of the stringified value.

class webgrid.**ColumnGroup**(*label*, *class_=None*)

Represents a grouping of grid columns which may be rendered within a group label.

Args:
label (str): Grouping label to be rendered for the column set. class_ (str): CSS class name to apply in HTML rendering.

8.2 Built-in Specialized Columns

class webgrid.**LinkColumnBase**(**args*, ***kwargs*)

Base class for columns rendering as links in HTML.

Expects a subclass to supply a *create_url* method for defining the link target.

Notable args:
link_label (str, optional): Caption to use instead of extracted data from the record.

Class attributes:
link_attrs (dict): Additional attributes to render on the A tag.

create_url(*record*)

Generate a URL from the given record.

Expected to be overridden in subclass.

link_to(*label*, *url*, ***kwargs*)

Basic render of an anchor tag.

render_html(*record*, *hah*)

Renderer override for HTML to set up a link rather than using the raw data value.

class webgrid.**BoolColumn**(**args*, ***kwargs*)

Column rendering values as True/False (or the given labels).

Notable args:

reverse (bool, optional): Switch true/false cases.

true_label (str, optional): String to use for the true case.

false_label (str, optional): String to use for the false case.

format_data(*data*)

Use to adjust the value extracted from the record for this column. By default, no change is made. Useful in sub-classes.

class webgrid.**YesNoColumn**(**args*, ***kwargs*)

BoolColumn rendering values as Yes/No.

Notable args:

reverse (bool, optional): Switch true/false cases.

class webgrid.**DateColumnBase**(**args*, ***kwargs*)

Base column for rendering date values in specified formats.

Designed to work with Python date/datetime/time and Arrow.

Notable args/attributes:

html_format (str, optional): Date format string for HTML.

csv_format (str, optional): Date format string for CSV.

xls_num_format (str, optional): Date format string for Excel.

xls_width_calc(*value*)

Determine approximate width from value.

Value will be a date or datetime object, format as if it was going to be in HTML as an approximation of its format in Excel.

class webgrid.**DateColumn**(**args*, ***kwargs*)

Column for rendering date values in specified formats.

Designed to work with Python date and Arrow.

Notable args/attributes:

html_format (str, optional): Date format string for HTML.

csv_format (str, optional): Date format string for CSV.

xls_num_format (str, optional): Date format string for Excel.

class webgrid.**DateTimeColumn**(**args*, ***kwargs*)

Column for rendering datetime values in specified formats.

Designed to work with Python datetime and Arrow.

Notable args/attributes:

html_format (str, optional): Date format string for HTML.

csv_format (str, optional): Date format string for CSV.

xls_num_format (str, optional): Date format string for Excel.

class webgrid.**TimeColumn**(**args*, ***kwargs*)

Column for rendering time values in specified formats.

Designed to work with Python time and Arrow.

Notable args/attributes:

html_format (str, optional): Date format string for HTML.

csv_format (str, optional): Date format string for CSV.

xls_num_format (str, optional): Date format string for Excel.

class webgrid.**NumericColumn**(**args*, ***kwargs*)

Column for rendering formatted number values.

Notable args:

format_as (str, optional): Generic formats. Default "general". - general: thousands separator and decimal point - accounting: currency symbol, etc. - percent: percentage symbol, etc.

places (int, optional): Decimal places to round to for general. Default 2.

curr (str, optional): Currency symbol for general. Default empty string.

sep (str, optional): Thousands separator. Default empty string.

dp (str, optional): Decimal separator. Default empty string.

pos (str, optional): Positive number indicator. Default empty string.

neg (str, optional): Negative number indicator for general. Default empty string.

trailneg (str, optional): Negative number suffix. Default empty string.

xls_neg_red (bool, optional): Renders negatives in red for Excel. Default True.

Class attributes:

xls_fmt_general, *xls_fmt_accounting*, *xls_fmt_percent* are Excel number formats used for the corresponding *format_as* setting.

get_num_format()

Match format_as setting to one of the format strings in class attributes.

html_decimal_format_opts(*data*)

Return tuple of options to expand for decimalfmt arguments.

places, *curr*, *neg*, and *trailneg* attributes are passed through unless *format_as* is "accounting".

render_html(*record*, *hah*)

HTML render override for numbers.

If format is percent, the value is multiplied by 100 to get the render value.

Negative values are given a "negative" CSS class in the render.

xls_construct_format(*fmt_str*)

Apply places and xls_neg_red settings to the given number format string.

property xlsx_style

Number format for XLSX target.

class webgrid.**EnumColumn**(**args*, ***kwargs*)

This column type is meant to be used with python *enum.Enum* type columns. It expects that the display value is the *value* attribute of the enum instance.

format_data(*value*)

Use to adjust the value extracted from the record for this column. By default, no change is made. Useful in sub-classes.

8.3 General Column Usage

Columns make up the grid's definition, as columns specify the data, layout, and formatting of the grid table. In WebGrid, a column knows how to render itself to any output target and how to apply sorting. In addition, the column is responsible for configuration of subtotals, filtering, etc.

The most basic usage of the column is to specify a heading label and the SQLAlchemy expression to be used. With this usage, sorting will be available in the grid/column headers, and the column will be rendered on all targets:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name)
```

The grid will have a keyed lookup for the column as it is defined. In the above case, the grid will pull the key from the SQLAlchemy expression, so the column may be referred to in surrounding code as:

```
grid.column('name')
```

8.3.1 Filtering

When defining a column for a grid, a filter may be specified as part of the spec:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name, TextFilter)
```

In the above, filtering options will be available for the *name* column. Because *TextFilter* supports the single-search UI, the column will also be automatically searched with that feature.

While the most common usage of filters simply provides the filter class for the column definition, a filter instance may be provided instead. Filter instances are useful when the column being filtered differs from the column being displayed:

```
class PeopleGrid(Grid):
    query_joins = ([entities.Person.location], )

    class LocationFilter(OptionsIntFilterBase):
        options_from = db.session.query(
            entities.Location.id, entities.Location.label
        ).all()
```

(continues on next page)

(continued from previous page)

```
    Column('Name', entities.Person.name, TextFilter)
    Column('Location', entities.Location.name, LocationFilter(entities.Location.id))
```

A number of things are happening there:

- The grid is joining two entities
- A custom filter is provided for selecting locations from a list (see *Custom Filters*)
- The location column renders the name, but filters based on the location ID

8.3.2 Sorting

Some columns are display-only or filter-only and do not make sense as sorting options. For these, use the *can_sort* option (default is True):

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name, can_sort=False)
```

More advanced sort customization is available for column subclasses. See *Custom Columns* for more information.

8.3.3 Visibility

WebGrid allows columns to be "turned off" for the table area (i.e. sort/filter only):

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name, visible=False)
```

Also, a column may be designated as being present for specific renderers. This can be helpful when a width-restricted format (like HTML) needs to leave out columns that are useful in more extensive exports:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name, render_in=('xlsx', 'csv'))
```

8.3.4 Subtotals

Useful for numeric columns in particlar, subtotals options may be specified to provide a way for the grid query to aggregate a column's data. Grids then have the option to turn on subtotals for display at the page or grand level (or both).

The most basic subtotal specification is simply turning it on for a column, which will use the SUM function:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name, has_subtotal=True)
```

The same result may be achieved with one of the string options recognized:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name, has_subtotal='sum')
```

The other string option recognized applies an average on the data:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name, has_subtotal='avg')
```

For greater customization, a callable may be provided that takes the aggregated expression and returns the function expression to use in the SQL query:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name,
           has_subtotal=lambda col: sa.sql.func.count(col))
```

Finally, a string may be provided for output on the totals row(s) instead of aggregated data:

```
class PeopleGrid(Grid):
    Column('Name', entities.Person.name, has_subtotal="What's in a name?")
```

8.4 Custom Columns

The basic WebGrid Column is flexible enough to handle a great deal of data. With the other supplied built-in columns for specific data types (boolean, date, float/decimal, int, etc.), the most common scenarios are covered. However, any of these supplied column classes may be extended for application-specific scenarios.

Below are some examples of common customizations on grid columns.

Rendered value:

```
class AgeColumn(Column):
    def extract_data(self, record):
        # All rendered targets will show this output instead of the actual data value
        if record.age < 18:
            return 'Under 18'
        return 'Over 18'
```

Render specialized for single target:

```
class AgeColumn(Column):
    def render_html(self, record, hah):
        # Only the HTML output will show this instead of the actual data value.
        if record.age < 18:
            # Add a CSS class to this cell for further styling.
            hah.class_ = 'under-18'
            return 'Under 18'
        return 'Over 18'
```

Sorting algorithm:

```
class ShipmentReceived(Column):
    def apply_sort(self, query, flag_desc):
        # Always sort prioritized shipments first
        if flag_desc:
            return query.order_by(
                priority_col.asc(),
                self.expr.desc(),
```

(continues on next page)

(continued from previous page)

```
            )
        return query.order_by(
            priority_col.asc(),
            self.expr.asc(),
        )
```

XLSX formula:

```
class ConditionalFormulaColumn(Column):
    xlsx_formula = '=IF(AND(K{0}<>"",C{0}<>""),(K{0}-C{0})*24,"")'

    def render_xlsx(self, record, rownum=0):
        return self.xlsx_formula.format(rownum)
```

Value links to another view:

```
class ProjectColumn(LinkColumnBase):
    def create_url(self, record):
        return flask.url_for(
            'admin.project-view',
            objid=record.id,
        )
```

CHAPTER

NINE

FILTERS

9.1 Base Filters

class webgrid.filters.**Operator**(*key*, *display*, *field_type*, *hint=None*)

Filter operator representing name and potential inputs.

See webgrid.filters.ops for a collection of predefined operators.

Args:

key (str): Internal identifier to be used in code and in request args.

display (str): Label for rendering the operator.

field_type (str): Input field spec. Can be: - None: no input fields - input: single freeform text input - 2inputs: two freeform text inputs - select: single select box - select+input: select box as first input, freeform text as second

hint (str, optional): Input field hint to show in UI. Defaults to None.

class webgrid.filters.**FilterBase**(*sa_col=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *dialect=None*)

Base filter class interface for webgrid filters.

Contains filter operators, inputs, render information, and the inner workings to apply the filter to the database query as needed.

Args:

sa_col (Expression): SQLAlchemy expression to which we apply operator/values.

default_op (str, optional): UI shortcut to enable filter with the specified op if none is given by the user. Defaults to None.

default_value1 (str, optional): Use with *default_op* to set input. Defaults to None.

default_value2 (str, optional): Use with *default_op* to set input. Defaults to None.

dialect (str, optional): DB dialect in use, for filters supporting multiple DBMS platforms. Defaults to None.

Class attributes:

operators (tuple(Operator)): Available filter operators in the order they should appear.

primary_op (str, optional): Key of operator to be selected automatically when the filter is added in UI. Defaults to first available filter op.

init_attrs_for_instance (tuple(str)): Attributes to set when copying filter instance. Should not normally need to be set.

input_types (tuple(str)): Possible input types for renderer to make available. Can be "input", "input2", and/or "select". Defaults to *("input",)*.

receives_list (bool): Filter takes a list of values in its *set* method. Defaults to False.

is_aggregate (bool): Filter applies to HAVING clause instead of WHERE

Instance attributes:

op (str): Operator key set by the user or programmatically.

value1 (Any): First input value following validation processing.

value2 (Any): Second input value following validation processing.

value1_set_with (str): First input value raw from *set* call.

value2_set_with (str): Second input value raw from *set* call.

error (bool): True if input processing encountered a validation error.

`apply`(*query*)

Query modifier to apply the needed clauses for filter inputs.

`format_invalid`(*exc*, *col*)

Wrapper for generating a validation error string.

`get_search_expr`()

Filters can be used for the general "single search" function on the grid. For this to work in SQL, the grid needs to pull search expressions from all filters and OR them together.

Return value is expected to be a callable taking one argument (the search value). E.g. *lambda value: self.sa_col.like('%{}%'.format(value))*

Return value of *None* is filtered out, essentially disabling search for the filter.

`property is_active`

Filter is active if op is set and input requirements are met.

`property is_display_active`

Filter display is active (i.e. show as a selected filter in UI) if op is set.

`new_instance`(***kwargs*)

Note: Ensure any overrides of this method accept and pass through kwargs to preserve compatibility in future

`property op_keys`

List of Operator keys used by this filter.

`process`(*value*, *is_value2*)

Process the values as given to .set(), validating and manipulating as needed.

`raise_unrecognized_op`()

Specified operator was not in the filter's list.

`set`(*op*, *value1*, *value2=None*)

Set filter operator and input values.

Stores the raw inputs, then processes the inputs for validation. Applies the default operator if needed.

Args:

op (str): Operator key. value1 (Any): First filter input value. Pass None if the operator takes no input. value2 (Any, optional): Second filter input value. Defaults to None.

Raises:
validators.ValueInvalid: One or more inputs did not validate.

class webgrid.filters.**OptionsFilterBase**(*sa_col*, *value_modifier='auto'*, *default_op=None*, *default_value1=None*, *default_value2=None*)

Base class for filters having a list of options.

UI for these is a single select box. By default, these are all shown with a search box within the filter, and checkboxes to pick multiple items.

Notable args:
value_modifier (Union(str, callable, Validator), optional): modifier to apply to the input value(s) in the request. Generally, the options list in the filter will have the "true" type in the identifier, but the request will come in with strings. If *value_modifier* is "auto", we'll check one of the options IDs with some known types to pick a webgrid validator. Or, a webgrid validator can be passed in. Or, pass a callable, and it will be wrapped as a validator. Defaults to "auto".

Class attributes:
options_from (tuple): Iterable of options available to the filter. Defined here as a class attribute, but can be overridden as an instance attribute, property, or method. Options are expected to be tuples of the form (key, value). *key* is the part that will be validated on input and used in the filter query clause. *value* is displayed in UI.

apply(*query*)

Query modifier to apply the needed clauses for filter inputs.

get_search_expr()

Match up a search value to option display, grab the corresponding keys, and search.

match_keys_for_value(*value*)

Used for single-search to match search value to part of an option's display string.

new_instance(***kwargs*)

Note: Ensure any overrides of this method accept and pass through kwargs to preserve compatibility in future

property option_keys

Extract a keys list from the options tuples.

property options_seq

Resolver for *options_from* that caches the options values.

Tries to treat *options_from* as a callable first, and if that fails, refers to it as an attribute/property value instead.

process(*value*)

Apply the *value_modifier* to a value.

set(*op*, *values*, *value2=None*)

Set the filter op/values to be used by query modifiers.

Since this type of filter has only one input box, *value2* is ignored (present here for consistent API). Each of the items passed in the first filter value is processed with the selected/given validator. Any items that do not pass validation are ignored and left out of the query.

Args:
op (str): Operator key. values (iterable): List/tuple/iterable of values to be validated. value2 (Any, optional): Ignored. Defaults to None.

setup_validator()

Select a validator by type if *value_modifier* is "auto", or wrap a callable.

class webgrid.filters.**OptionsIntFilterBase**(*sa_col*, *value_modifier=<class 'webgrid.validators.IntValidator'>*, *default_op=None*, *default_value1=None*, *default_value2=None*)

Base class for filters having a list of options with integer keys.

Shortcut for using *OptionsFilterBase* and supplying *webgrid.validators.IntValidator* as the *value_modifier*.

9.2 Built-in Filters

class webgrid.filters.**TextFilter**(*sa_col=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *dialect=None*)

Filter with single freeform text input.

apply(*query*)

Query modifier to apply the needed clauses for filter inputs.

property comparisons

Handles the dialect-specific job of text comparisons.

In a functional text filter, we want to be case-insensitive. The default behavior of some databases (such as postgresql) is to be case-sensitive for LIKE operations. We work around that for dialects known to have that complexity, and compare upper-case values or use the ILIKE operator.

Some, like mssql, are normally case-insensitive for string comparisons, and we assume that is the case here. Obviously, the database can be set up differently. If it is, that case would need to be handled separately in a custom filter.

get_search_expr()

Filters can be used for the general "single search" function on the grid. For this to work in SQL, the grid needs to pull search expressions from all filters and OR them together.

Return value is expected to be a callable taking one argument (the search value). E.g. *lambda value: self.sa_col.like('%{}%'.format(value))*

Return value of *None* is filtered out, essentially disabling search for the filter.

class webgrid.filters.**IntFilter**(*sa_col=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *dialect=None*)

Number filter validating inputs as integers.

validator

alias of IntValidator

class webgrid.filters.**AggregateIntFilter**(*sa_col=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *dialect=None*)

Number filter validating inputs as integers, for use on aggregate columns.

class webgrid.filters.**NumberFilter**(*sa_col=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *dialect=None*)

Same as int filter, but will handle real numbers and type everything as a decimal.Decimal object

process(*value*, *is_value2*)

Process the values as given to .set(), validating and manipulating as needed.

validator

alias of FloatValidator

class webgrid.filters.**AggregateNumberFilter**(*sa_col=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *dialect=None*)

Number filter validating inputs as Decimal, for use on aggregate columns.

class webgrid.filters.**DateFilter**(*sa_col*, *_now=None*, *default_op=None*, *default_value1=None*, *default_value2=None*)

Complex date filter.

Depending on the operator, inputs could be one or two freeform, or a select and freeform.

Notable args:

_now (datetime, optional): Useful for testing, supplies a date the filter will use instead of the true *datetime.now()*. Defaults to None.

apply(*query*)

Query modifier to apply the needed clauses for filter inputs.

check_arrow_type()

Verify that the expression given to the filter is not ArrowType. If it is, cast it to a date to avoid type problems in the date filter

process(*value*, *is_value2*)

Process the values as given to .set(), validating and manipulating as needed.

set(*op*, *value1*, *value2=None*)

Set filter operator and input values.

Stores the raw inputs, then processes the inputs for validation. Applies the default operator if needed.

Args:

op (str): Operator key. value1 (Any): First filter input value. Pass None if the operator takes no input. value2 (Any, optional): Second filter input value. Defaults to None.

Raises:

validators.ValueInvalid: One or more inputs did not validate.

class webgrid.filters.**DateTimeFilter**(*sa_col*, *_now=None*, *default_op=None*, *default_value1=None*, *default_value2=None*)

Complex datetime filter.

Depending on the operator, inputs could be one or two freeform, or a select and freeform.

Notable args:

_now (datetime, optional): Useful for testing, supplies a date the filter will use instead of the true *datetime.now()*. Defaults to None.

check_arrow_type()

DateTimeFilter has no problems with ArrowType. Pass this case through.

get_search_expr()

Filters can be used for the general "single search" function on the grid. For this to work in SQL, the grid needs to pull search expressions from all filters and OR them together.

Return value is expected to be a callable taking one argument (the search value). E.g. *lambda value: self.sa_col.like('%{}%'.format(value))*

Return value of *None* is filtered out, essentially disabling search for the filter.

process(*value*, *is_value2*)

Process the values as given to .set(), validating and manipulating as needed.

class webgrid.filters.**TimeFilter**(*sa_col=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *dialect=None*)

Time filter with one or two freeform inputs.

apply(*query*)

Query modifier to apply the needed clauses for filter inputs.

get_search_expr(*date_comparator=None*)

Filters can be used for the general "single search" function on the grid. For this to work in SQL, the grid needs to pull search expressions from all filters and OR them together.

Return value is expected to be a callable taking one argument (the search value). E.g. *lambda value: self.sa_col.like('%{}%'.format(value))*

Return value of *None* is filtered out, essentially disabling search for the filter.

process(*value*, *is_value2*)

Process the values as given to .set(), validating and manipulating as needed.

class webgrid.filters.**YesNoFilter**(*sa_col=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *dialect=None*)

Simple "bool" filter designed for use with the YesNoColumn.

No inputs, just the all/yes/no operators.

apply(*query*)

Query modifier to apply the needed clauses for filter inputs.

get_search_expr()

Filters can be used for the general "single search" function on the grid. For this to work in SQL, the grid needs to pull search expressions from all filters and OR them together.

Return value is expected to be a callable taking one argument (the search value). E.g. *lambda value: self.sa_col.like('%{}%'.format(value))*

Return value of *None* is filtered out, essentially disabling search for the filter.

class webgrid.filters.**OptionsEnumFilter**(*sa_col*, *value_modifier=None*, *default_op=None*, *default_value1=None*, *default_value2=None*, *enum_type=None*)

Options filter that pulls options from a python Enum.

Most filters can be used in column definitions as a class or an instance. With this filter, an instance must be given, so the *enum_type* can be specified.

Notable args:
enum_type (Enum): Python Enum type to use for options list.

default_modifier(*value*)

Generic *value_modifier* that validates an item in the given Enum.

new_instance(***kwargs*)

Note: Ensure any overrides of this method accept and pass through kwargs to preserve compatibility in future

options_from()

Override as an instance method here, returns the options tuples from the Enum.

process(*value*)

Validate value using *value_modifier*.

9.3 Custom Filters

The basic requirements for a custom filter are to supply the operators, the query modifier for applying the filter, and a search expression for single-search. A few examples are here.

Simple custom filter:

```
class ActivityStatusFilter(FilterBase):
    # operators are declared as Operator(<key>, <display>, <field-type>)
    operators = (
        Operator('all', 'all', None),
        Operator('pend', 'pending', None),
        Operator('comp', 'completed', None),
    )

    def get_search_expr(self):
        status_col = sa.sql.case(
            [(Activity.flag_completed == sa.true(), 'completed')],
            else_='pending'
        )
        # Could use ilike here, depending on the target DBMS
        return lambda value: status_col.like('%{}%'.format(value))

    def apply(self, query):
        if self.op == 'all':
            return query
        if self.op == 'comp':
            return query.filter(Activity.flag_completed == sa.true())
        if self.op == 'pend':
            return query.filter(Activity.flag_completed == sa.false())
        return super().apply(self, query)
```

Options filter for INT foreign key lookup:

```
class VendorFilter(OptionsIntFilterBase):
    def options_from(self):
        # Expected to return a list of tuples (id, label).
        # In this case, we're retrieving options from the database.
        return db.session.query(Vendor.id, Vendor.label).select_from(
            Vendor
        ).filter(
            Vendor.active_flag == sa.true()
        ).order_by(
            Vendor.label
        ).all()
```

Aggregate filters, i.e. those using the HAVING clause instead of WHERE, must be marked with the *is_aggregate* flag. Single-search via expressions will only address aggregate filters if all search filters are aggregate. Using an aggregate filter will require a GROUP BY clause be set.

class AggregateTextFilter(TextFilter):
is_aggregate = True

CHAPTER

TEN

RENDERERS

10.1 Base Renderer

class webgrid.renderers.**Renderer**(*grid*)

Abstract interface for a WebGrid renderer.

If the renderer has an *init* callable, it will be called by the constructor.

Renderers are callable, which will trigger the *render* method:

```
renderer = HTML(my_grid)
output = renderer()
```

Args:

grid (BaseGrid): Parent grid of this renderer instance.

can_render()

Guard method for preventing a renderer from overflowing the target format.

For instance, spreadsheets have limitation in the number of possible rows. A renderer to that format should check that the record count does not exceed that limit.

Returns:

bool: True if the renderer can proceed.

property columns

Cache a set of columns from the grid that will render on this target.

abstract property name

Identifier used to find columns that will render on this target.

abstract render()

Main renderer method returning the output.

10.2 Built-in Renderers

class webgrid.renderers.**HTML**(*grid*)

Renderer for HTML output.

buffer_th(*colspan*, ***kwargs*)

Render a placeholder TH tag for spacing between column groups.

can_render()

Guard method for preventing a renderer from overflowing the target format.

For instance, spreadsheets have limitation in the number of possible rows. A renderer to that format should check that the record count does not exceed that limit.

Returns:
bool: True if the renderer can proceed.

property columns

Cache a set of columns from the grid that will render on this target.

confirm_export()

Export confirmation data as a JSON object for use by the JS asset.

current_url(***kwargs*)

Generate a URL from current request args and the given kwargs.

export_url(*renderer*)

Generate a URL that will trigger an export to one of the grid's renderers.

Args:
renderer (str): Export key (e.g. xlsx, csv) for rendering target.

filtering_add_filter_select()

Render the select box for adding a new filter. Used by the filter template.

filtering_col_inputs1(*col*)

Render the first input, which can be freeform or select.

filtering_col_inputs2(*col*)

Render the second filter input, currently only a freeform.

filtering_col_label(*col*)

Label getter for filter column.

filtering_col_op_select(*col*)

Render select box for filter Operator options.

filtering_fields()

Table rows for the filter area.

filtering_filter_options_multi(*filter*, *field_name*)

Render the multiselect options.

filtering_json_data()

Export certain filter data as a JSON object for use by the JS asset.

filtering_multiselect(*field_name*, *current_selected*, *options*)

Almost all selects are rendered with multiselect UI. Render that here.

Structure is based on the jQuery Multiselect plugin. For efficiency of render with large numbers of options, we customized the plugin for WebGrid use and offloaded the main render/transform here.

filtering_session_key()

Hidden input to preserve the session key on form submission.

filtering_table_attrs(***kwargs*)

HTML attributes to render on the grid filter table element.

filtering_table_row(*col*)

Single filter row with op and inputs.

footer()

Render the grid footer area from template.

form_action_method()

Detect whether the header form should have a GET or POST action.

By default, we look at the grid manager's args_loaders for RequestFormLoader. If it is present, the form will be POST.

form_action_url()

URL target for the grid header form.

get_add_filter_row()

Render just the Add Filter area on a row.

get_group_heading_colspans()

Computes the number of columns spanned by various groups.

Note, this may not be the number of columns in the group in the grid definition, because some of those columns may not render in this target.

get_search_row()

Render the single-search input, along with filter select.

grid_attrs()

HTML attributes to render on the main grid div element.

group_th(*group*, *colspan*, ***kwargs*)

Render a column group heading with the needed span.

Note, will render an empty placeholder if the column has no group.

has_groups()

Returns True if any of the renderer's columns is part of a column group.

header()

Return content for the grid header area. Used by the grid template.

header_filtering()

Return content for the grid filter area. Used by the header template.

header_form_attrs(***kwargs*)

HTML attributes to render on the grid header form element.

header_paging()

Render the paging area of the grid header.

header_sorting()

Render the sort area. Used by the header template.

load_content(*endpoint*, ***kwargs*)

Load content via Jinja templates.

Gives the grid manager a chance to render the template, in order to allow for application-level overrides on the grid templates. Otherwise, defaults to the internal Jinja environment set up for this renderer.

property name

Identifier used to find columns that will render on this target.

no_records()

Render a message paragraph indicating the current filters return no records.

paging_img_first()

Render the footer icon for the first page of the grid.

paging_img_first_dead()

Render the footer disabled icon for the first page of the grid.

paging_img_last()

Render the footer icon for the last page of the grid.

paging_img_last_dead()

Render the footer disabled icon for the last page of the grid.

paging_img_next()

Render the footer icon for the next page of the grid.

paging_img_next_dead()

Render the footer disabled icon for the next page of the grid.

paging_img_prev()

Render the footer icon for the previous page of the grid.

paging_img_prev_dead()

Render the footer disabled icon for the previous page of the grid.

paging_input()

Render the per-page input.

paging_select()

Render the page selection input.

paging_url_first()

Generate a URL for the first page of the grid.

paging_url_last()

Generate a URL for the last page of the grid.

paging_url_next()

Generate a URL for the next page of the grid.

paging_url_prev()

Generate a URL for the previous page of the grid.

`render`()

Main renderer method returning the output.

`render_select`(*options*, *current_selection=None*, *placeholder=('', Markup(' '))*, *name=None*, *id=None*, ***kwargs*)

Generalized select box renderer.

Args:

options (iterable): Option tuples (value, label) or (value, label, data). If the data piece is present, it will be rendered as the value of a "data-render" attribute on the option tag.

current_selection (iterable, optional): Option values to be marked as selected. Defaults to None.

placeholder (tuple(str), optional): Option to use as a "blank" value.

name (str, optional): Value for HTML name attribute. Defaults to None.

id (str, optional): Value for HTML id attribute. Defaults to a sanitized value derived from *name*.

kwargs: Passed as HTML attributes on the select tag.

`reset_url`(*session_reset=True*)

Generate a URL that will trigger a reset of the grid's UI options.

`sorting_select`(*number*)

Render the dropdown select of sorting options.

Args:

number (int): Priority of ordering option.

Returns:

str: Jinja-rendered string.

`sorting_select1`()

Render the first sort select.

`sorting_select2`()

Render the second sort select.

`sorting_select3`()

Render the third sort select.

`sorting_select_options`()

Generate list of tuple pairs (key, label) and (-key, label DESC) for sort options.

Returns:

list: List of tuple pairs.

`table`()

Render the table area of the grid from template.

`table_attrs`(***kwargs*)

Apply default HTML table attributes to the supplied kwargs.

Returns:

dict: keys/values to be rendered as attributes

`table_column_headings`()

Combine all rendered column headings and return as Markup.

table_grandtotals(*rownum*, *record*)

Render a Grand totals row based on subtotal columns defined in the grid.

table_group_headings()

Combine all rendered column group headings and return as Markup.

table_pagetotals(*rownum*, *record*)

Render a Page totals row based on subtotal columns defined in the grid.

table_rows()

Combine rows rendered from grid records, return as Markup.

Page/Grand totals are included here as rows if enabled in the grid.

table_td(*col*, *record*)

Render a table data cell.

Value is obtained for render from the grid column's *render* method. To override how a column's data is rendered specifically for HTML, supply a *render_html* method on the column.

table_th(*col*)

Render a single column heading TH tag.

Sortable columns are rendered as links with the needed URL args.

table_totals(*rownum*, *record*, *label*, *numrecords*)

Render a totals row based on subtotal columns defined in the grid.

table_tr(*rownum*, *record*)

Generate rendered cells and pass to table_tr_output for rendered result.

table_tr_output(*cells*, *row_hah*)

Combine rendered cells and output a TR tag.

table_tr_styler(*rownum*, *record*)

Compile the styling to be used on a given HTML grid row.

Applies odd/even class based on the row number. Adds in any row stylers present in grid configuration.

Args:
rownum (int): row number in the rendered grid. record (Any): result record.

Returns:
HTMLAttributes: attributes collection to be applied on a TR.

class webgrid.renderers.**JSON**(*grid*)

Renderer for JSON output

as_response()

Return a response via the grid's manager.

can_render()

Guard method for preventing a renderer from overflowing the target format.

For instance, spreadsheets have limitation in the number of possible rows. A renderer to that format should check that the record count does not exceed that limit.

Returns:
bool: True if the renderer can proceed.

property columns

Cache a set of columns from the grid that will render on this target.

property name

Identifier used to find columns that will render on this target.

render()

Main renderer method returning the output.

serialized_columns()

Usually we would use the renderer's column list. For JSON, though, we want to supply any labels possible for use in a front-end app. The front-end needs to know names for filters, for example.

class webgrid.renderers.**XLSX**(*grid*)

Renderer for Excel XLSX output.

adjust_column_widths(*writer*, *wb*)

Apply stored column widths to the XLSX worksheet.

as_response(*wb=None*, *sheet_name=None*)

Return an attachment file via the grid's manager.

body_headings(*xlh*, *wb*)

Render group and column label rows.

Args:

xlh (WriterX): Helper for writing worksheet cells. wb (Workbook): xlsxwriter Workbook object for direct usage.

body_records(*xlh*, *wb*)

Render records and totals rows.

Args:

xlh (WriterX): Helper for writing worksheet cells. wb (Workbook): xlsxwriter Workbook object for direct usage.

build_sheet(*wb=None*, *sheet_name=None*)

Create and populate a worksheet for the current grid.

Args:

wb (Workbook, optional): xlsxwriter Workbook. Defaults to None (create one). sheet_name (str, optional): Sheet name. Defaults to None (use grid identity).

Raises:

ImportError: No suitable XLSX library installed. RenderLimitExceeded: Too many records to render to the target.

Returns:

Workbook: Created/supplied workbook with the rendered worksheet added.

can_render()

Guard method for preventing a renderer from overflowing the target format.

For instance, spreadsheets have limitation in the number of possible rows. A renderer to that format should check that the record count does not exceed that limit.

Returns:

bool: True if the renderer can proceed.

property columns

Cache a set of columns from the grid that will render on this target.

file_name()

Return an output filename based on grid identifier.

A random numeric suffix is added. This is due to Excel's limitation to having only one workbook open with a given name. Excel will not allow a second file with the same name to open, even if the files are in different paths.

get_group_heading_colspans()

Computes the number of columns spanned by various groups.

Note, this may not be the number of columns in the group in the grid definition, because some of those columns may not render in this target.

has_groups()

Returns True if any of the renderer's columns is part of a column group.

property name

Identifier used to find columns that will render on this target.

record_row(*xlh*, *rownum*, *record*, *wb*)

Render a single row from data record.

Value is obtained for render from the grid column's *render* method. To override how a column's data is rendered specifically for XLSX, supply a *render_xlsx* method on the column.

Args:

xlh (WriterX): Helper for writing worksheet cells. rownum (int): Not used by default, but helpful for style overrides. record (Any): Object containing row data. wb (Workbook): xlsxwriter Workbook object for direct usage.

render()

Main renderer method returning the output.

sanitize_sheet_name(*sheet_name*)

Work around Excel limitations on names of worksheets.

sheet_body(*xlh*, *wb*)

Render the headings/records area of the worksheet.

Args:

xlh (WriterX): Helper for writing worksheet cells. wb (Workbook): xlsxwriter Workbook object for direct usage.

sheet_footer(*xlh*, *wb*)

Placeholder method for app-specific sheet footer rendering.

Args:

xlh (WriterX): Helper for writing worksheet cells. wb (Workbook): xlsxwriter Workbook object for direct usage.

sheet_header(*xlh*, *wb*)

Placeholder method for app-specific sheet header rendering.

Args:

xlh (WriterX): Helper for writing worksheet cells. wb (Workbook): xlsxwriter Workbook object for direct usage.

totals_row(*xlh*, *rownum*, *record*, *wb*)

Render a totals row based on subtotal columns defined in the grid.

update_column_width(*col*, *data*)

Compute and store a column width from length of current data.

class webgrid.renderers.**CSV**(*grid*)

Renderer for CSV output.

as_response()

Return an attachment file via the grid's manager.

body_headings()

Render the column headers.

Note, column groups do not have real meaning in the CSV context, so they are left out here.

body_records()

Render all rows from grid records.

Value is obtained for render from each grid column's *render* method. To override how a column's data is rendered specifically for CSV, supply a *render_csv* method on the column.

build_csv()

Render grid output as CSV and return the contents in an IO stream.

can_render()

Guard method for preventing a renderer from overflowing the target format.

For instance, spreadsheets have limitation in the number of possible rows. A renderer to that format should check that the record count does not exceed that limit.

Returns:
bool: True if the renderer can proceed.

property columns

Cache a set of columns from the grid that will render on this target.

file_name()

Return an output filename based on grid identifier.

A random numeric suffix is added. This is due to Excel's limitation to having only one workbook open with a given name. Excel will not allow a second file with the same name to open, even if the files are in different paths.

property name

Identifier used to find columns that will render on this target.

render()

Main renderer method returning the output.

CHAPTER

ELEVEN

TESTING

11.1 Test Helpers

A collection of utilities for testing webgrid functionality in client applications

class webgrid.testing.**GridBase**

Base test class for Flask or Keg apps.

Class Attributes:

grid_cls: Application grid class to use during testing

filters: Iterable of (name, op, value, expected) tuples to check for filter logic, or a callable returning such an iterable. *name* is the column key. *op* and *value* set the filter parameters. *expected* is either a SQL string or compiled regex to find when the filter is enabled.

sort_tests: Iterable of (name, expected) tuples to check for sort logic. *name* is the column key. *expected* is a SQL string to find when the sort is enabled.

assert_in_query(*look_for*, *grid=None*, *_query_string=None*, ***kwargs*)

Verify the given SQL string is in the grid's query.

Args:

look_for (str): SQL string to find.

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

assert_not_in_query(*look_for*, *grid=None*, *_query_string=None*, ***kwargs*)

Verify the given SQL string is not in the grid's query.

Args:

look_for (str): SQL string to find.

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

assert_regex_in_query(*look_for*, *grid=None*, *_query_string=None*, ***kwargs*)

Verify the given regex matches the grid's query.

Args:

look_for (str or regex): Regex to search (can be compiled or provided as string).

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

check_filter(*name*, *op*, *value*, *expected*)

Assertions to perform on a filter test.

Args:

name (str): Column key to filter. op (str): Filter operator to enable. value (Any): Filter value to assign. expected (str or regex): SQL string or compiled regex to find.

check_sort(*k*, *ex*, *asc*)

Assertions to perform on a sort test.

Args:

k (str): Column key to sort. ex (str or regex): SQL string to find. asc (bool): Flag indicating ascending/descending order.

expect_table_contents(*expect*, *grid=None*, *_query_string=None*, ***kwargs*)

Run assertions to compare rendered data rows with expected data.

Args:

expect (list): List representation of expected table data.

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

expect_table_header(*expect*, *grid=None*, *_query_string=None*, ***kwargs*)

Run assertions to compare rendered headings with expected data.

Args:

expect (list): List representation of expected table data.

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

get_grid(*grid_args*, **args*, ***kwargs*)

Construct grid from args and kwargs, and apply grid_args.

Args:

grid_args: grid query args

Returns:

grid instance

get_pyq(*grid=None*, *_query_string=None*, ***kwargs*)

Turn provided/constructed grid into a rendered PyQuery object.

Args:

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

Returns:

PyQuery object

get_session_grid(**args*, *_query_string=None*, ***kwargs*)

Construct grid from args and kwargs, and apply query string.

Args:

_query_string: URL query string with grid query args

Returns:

grid instance

query_to_str(*statement*, *bind=None*)

Render a SQLAlchemy query to a string.

test_filters()

Use filters attribute/property/method to run assertions.

test_search_expr_passes(*grid=None*, *_query_string=None*)

Assert that a single-search query executes without error.

test_sort()

Use sort_tests attribute/property to run assertions.

class webgrid.testing.**MSSQLGridBase**

MSSQL dialect produces some string oddities compared to other dialects, such as having the N'foo' syntax for unicode strings instead of 'foo'. This can clutter tests a bit. Using MSSQLGridBase will patch that into the asserts, so that look_for will match whether it has the N-prefix or not.

assert_in_query(*look_for*, *grid=None*, *context=None*, *_query_string=None*, ***kwargs*)

Verify the given SQL string is in the grid's query.

Args:

look_for (str): SQL string to find.

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

assert_not_in_query(*look_for*, *grid=None*, *context=None*, *_query_string=None*, ***kwargs*)

Verify the given SQL string is not in the grid's query.

Args:

look_for (str): SQL string to find.

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

assert_regex_in_query(*look_for*, *grid=None*, *context=None*, *_query_string=None*, ***kwargs*)

Verify the given regex matches the grid's query.

Args:

look_for (str or regex): Regex to search (can be compiled or provided as string).

grid (BaseGrid, optional): Grid to use instead of *self.get_session_grid*. Defaults to None.

kwargs (dict, optional): Additional args passed to *self.get_session_grid*.

query_to_str_replace_type(*compiled_query*)

Same as query_to_str, but accounts for pyodbc type-specific rendering.

webgrid.testing.**assert_list_equal**(*list1*, *list2*)

A list-specific equality assertion.

This method is based on the Python *unittest.TestCase.assertListEqual* method.

Parameters

- **list1** –
- **list2** –

Returns

webgrid.testing.**assert_rendered_xlsx_matches**(*rendered_xlsx*, *xlsx_headers*, *xlsx_rows*)

Verifies that *rendered_xlsx* has a set of headers and values that match the given parameters.

NOTE: This method does not perform in-depth analysis of complex workbooks!

Assumes header rows and data rows are contiguous. Multiple worksheets or complex layouts *are not verified!*

Parameters

- **rendered_xlsx** – binary data passed to openpyxl as file contents
- **xlsx_headers** – list of rows of column headers
- **xlsx_rows** – list of rows in order as they will appear in the worksheet

webgrid.testing.**query_to_str**(*statement*, *bind=None*)

returns a string of a sqlalchemy.orm.Query with parameters bound

WARNING: this is dangerous and ONLY for testing, executing the results of this function can result in an SQL Injection attack.

11.2 Test Usage

What follows is a brief example of setting up filter/sort/content tests using *GridBase*:

```
class TestTemporalGrid(webgrid.testing.GridBase):
    grid_cls = TemporalGrid

    sort_tests = (
        ('createdts', 'persons.createdts'),
        ('due_date', 'persons.due_date'),
        ('start_time', 'persons.start_time'),
    )

    @property
    def filters(self):
        # This could be assigned as a class attribute, or made into a method
        return (
            ('createdts', 'eq', dt.datetime(2018, 1, 1, 5, 30),
            "WHERE persons.createdts = '2018-01-01 05:30:00.000000'"),
            ('due_date', 'eq', dt.date(2018, 1, 1), "WHERE persons.due_date = '2018-01-01
↪'"),
            ('start_time', 'eq', dt.time(1, 30).strftime('%I:%M %p'),
            "WHERE persons.start_time = CAST('01:30:00.000000' AS TIME)"),
        )

    def setup_method(self, _):
        Person.delete_cascaded()
        Person.testing_create(
            createdts=dt.datetime(2018, 1, 1, 5, 30),
            due_date=dt.date(2019, 5, 31),
            start_time=dt.time(1, 30),
        )
```

(continues on next page)

(continued from previous page)

```
    def test_expected_rows(self):
        # Passing a tuple of tuples, since headers can be more than one row (i.e.␣
↪grouped columns)
        self.expect_table_header((('Created', 'Due Date', 'Start Time'), ))

        self.expect_table_contents((('01/01/2018 05:30 AM', '05/31/2019', '01:30 AM'), ))
```

CHAPTER

TWELVE

COMMON GOTCHAS

Following are some common scenarios encountered in application development with WebGrid. Generally, there is a reason why the given behavior occurs. If you run into a problem that is not listed here but seems to be wrong or counterintuitive behavior, feel free to create an issue for it.

12.1 Grid

Grid identifier

Every grid instance has an identifier: the *ident* property. This value is used for session lookups to grab/store the most recent set of request args applied to that grid.

The identifier is set in one of three ways, in order of priority:

- Value passed in grid constructor:

```
class PeopleGrid(BaseGrid):
    Column('Name', Person.name)

grid = PeopleGrid(ident='my-awesome-grid')
```

- Class attribute assigned:

```
class PeopleGrid(BaseGrid):
    identifier = 'my-awesome-grid'
    Column('Name', Person.name)
```

- Default derived from class name, e.g. *people_grid*

Query usage

Grid classes are generally loaded at import time. Because Column instances attached to a grid class can refer to SQLAlchemy expressions, some frameworks have limitations in what can be declared.

Most entity attributes used directly will be fine. However, if a grid refers to attributes in a SQLAlchemy query, under Flask-SQLAlchemy you will see some import-time issues. Creating queries within that framework requires the app context to be present, but at import time, that is usually not the case.

To resolve this, currently, the grid class would need to be defined in a factory method:

```
def people_grid_factory():
    query = db.session.query(Person.name).subquery()

    class PeopleGrid(BaseGrid):
```

(continues on next page)

(continued from previous page)

```
        Column('Name', query.c.name)

    return PeopleGrid
```

12.2 Request Management

Multiple grids on a page

Because grids use URL arguments to configure page/filter/sort, having more than one grid on a page requires some additional setup. Use the qs_prefix on one of the grids to differentiate it in session/request loads:

```
class PeopleGrid(BaseGrid):
    Column('Name', Person.name)

class EquipmentGrid(BaseGrid):
    Column('Model', Equipment.model)

people_grid = PeopleGrid()
equipment_grid = EquipmentGrid(qs_prefix='equip')
```

Query string too long

Submitting the header form typically directs to a GET request with args for all page/filter/sort information. While we trim out any filter inputs we can to keep the string size manageable, with enough filters selected, the query can overflow the limits of the target server.

This scenario comes up most commonly with multiselect filters having lots of options. Each option selected results in the full filter value arg being added to the query string, and so it will easily cause the overflow.

The solution is to use the form POSTed args loader (`RequestFormLoader`), which will direct the UI form to use a POST instead of a GET. This is not the default due to backwards-compatibility, since many views using WebGrid are GET-only routes. However, the default may change in the future.

Please refer to *Arguments Loaders* for details about customizing the loading of grid configuration.

An alternative solution would be to configure your nginx/apache/etc. parameters accordingly for maximum request size.

Overriding the session

Session storage in WebGrid is generally all-or-nothing for filter information. That is, we have two basic scenarios:

- *session_key* URL argument is present without any filter arguments
  - Filter information will be pulled from session and applied
- Filter arguments are present along with the *session_key*
  - Session information is discarded, and the new filter information stored in its place

Page/sort arguments do not trigger the session discard.

In some situations, it can be helpful to provide a single filter in a request without discarding session information. To do this, include a *session_override* arg (properly prefixed). The following example will retain any other filters in the session, and override only the name filter:

```
https://mysite.com/mypage?session_key=12345&session_override=1&op(name)=eq&v1(name)=steve
```

Sharing session between grids

Sometimes, grids are similar enough that a common set of filters will apply to multiple grids. For instance, a tabbed report of similar data may show varying levels of detail, but have the same top-level filters.

To share filter values across such a set of grids, you can pass the *session_key*. The target grid will recognize that a foreign session has loaded and ignore any filters that don't match up on the target.

Note, when doing this sort of operation, the session will update to reflect the new grid.

12.3 Column

Multiple columns with the same name

WebGrid does not enforce unique column key names in the grid definition. However, these keys are made to be unique at run time, to preserve the ability to refer to any column by its key. For example:

```
class PeopleGrid(BaseGrid):
    query_joins = ([entities.Person.location], )

    Column('Name', entities.Person.name, TextFilter)
    Column('Location', entities.Location.name, TextFilter)
```

In this example, both columns would be keyed as 'name'. To make this unique, WebGrid will find unique keys at run time. *Person.name* will have the *name* key, but *Location.name* will be assigned *name_1*. Further *name* columns would get *name_2*, *name_3*, etc.

Keep in mind, filter/sort arguments must follow the key. If we try to set a filter on location name in the view, that would become:

```
people_grid.set_filter('name_1', 'eq', 'Paris')
```

To apply a specific key in this scenario rather than accepting the one generated, simply label one of the columns:

```
class PeopleGrid(BaseGrid):
    query_joins = ([entities.Person.location], )

    Column('Name', entities.Person.name, TextFilter)
    Column('Location', entities.Location.name.label('location'), TextFilter)
```

Subclassing Column with a new constructor

In many cases, creating a subclass of *Column* for app-specific behavior is not a problem (see *Custom Columns*). If you need to put in a custom constructor, though, beware, for here be monsters.

In WebGrid, with the declarative grid setup, *Column* instances are created and attached to a grid class definition. When the grid class is instantiated, these column instances must be copied to new instances for the grid instance to use.

That instance copy assumes a certain arrangement of constructor arguments. The first four arguments must be in the same order: *label*, *key*, *filter*, and *can_sort*. The remaining arguments should also be present in the new constructor, or else they will likely not be carried over to the new instance (unless the custom constructor sets them).

This is a known limitation to the way that columns are instantiated for grids. Because the need for custom constructors is minimal in practice, this arrangement will likely stay in place for the foreseeable future.

Column only for filter, no display

For summary-level tables, it can be desirable to filter the recordset on values that are not in the rendered result. One must be careful when dealing with aggregations and grouping, however, because having the column in the SELECT list may require grouping and affect data granularity for the result.

To avoid inclusion in SELECT, pass the column expression directly to the filter rather than the column itself:

```
Column('Last Name', 'no_expr', TextFilter(Person.lastname), visible=False, can_
↪sort=False)
```

In the above case, *Person.lastname* will not be in SELECT. But, it will be included in the WHERE clause if the filter is set or search is used. The other keyword arguments remove the column from rendering and sorting, so the column is useful only for filtering.

12.4 Filter

Setting a filter

Filters may be set directly:

```
grid.column('name').filter.set('eq', 'steve')
```

Or, through the grid:

```
grid.set_filter('name', 'eq', 'steve')
```

Some filters have two values:

```
grid.column('date').filter.set('selmonth', '3', value2='2020')
```

The first value of a filter is required when setting, even if the filter does not take any values:

```
grid.column('name').filter.set('empty', None)
```

OptionsEnumFilter

Most filters can be assigned to a column as a class or an instance. One, *OptionsEnumFilter* currently requires an instance, so the *enum_type* can be passed in the constructor. This means the column also must be passed to the filter:

```
EnumColumn(
    'Type', Project.type, OptionsEnumFilter(Project.type, enum_type=ProjectType),
)
```

Aggregate columns

When constructing a grid column from an aggregate, remember that filters for that column will not be allowed in the SQL WHERE clause. Instead, they need to be in the HAVING clause.

Because of this, use the *Aggregate* filters instead of *IntFilter* and *NumberFilter*.

Using aggregate filters will require having a GROUP BY clause set on the grid query.

Aggregate filters and search

Search will only include aggregate filters if all searchable filters are aggregate.

The search box assumes that all search expressions can be combined with OR to generate a complete search expression. Because of this, search can use the WHERE clause or the HAVING clause, but not both. Using columns in HAVING requires they be in the GROUP BY, which will affect data granularity for some reports.

If search should include columns computed with aggregate functions, build a wrapping select query that includes the necessary aggregation and grouping in a nested select or CTE. Then, build the grid with the results of the wrapping query, which will not involve the need for aggregate filters.

12.5 File Exports

Excel sheet name limitations

Excel has a limit of 30 characters on worksheet names. If a name is provided that will exceed that limit, it will be truncated with a trailing ellipse.

Excel records limitations

Excel file formats have limits of how many rows can be included. This was a bigger issue when XLS was the common format, but XLSX does have limits as well.

The XLSX renderer will raise a *RenderLimitExceeded* exception if the query result is too large.

Excel file naming

Excel will not load multiple files with the same filename, even though they are in different directories. For this reason, we append a random numeric suffix on the filenames, so Excel will see them as different workbooks.

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y