blingalytics Documentation

Release 0.1.3

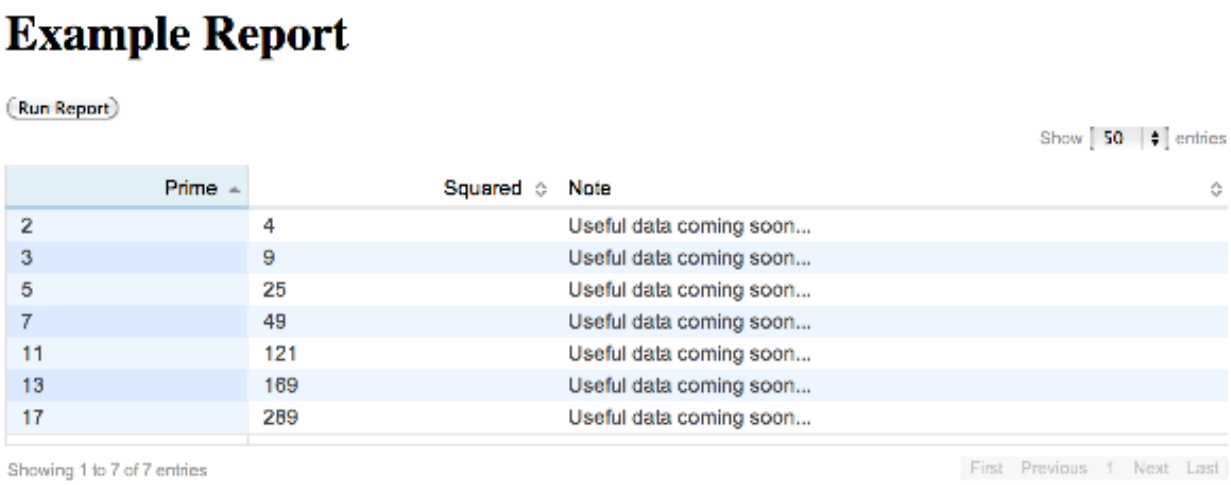
Jeff Schenck

December 10, 2013

Contents

CHAPTER 1

About



Blingalytics is an open-source Python project for building a flexible, powerful business reporting system.

- Define new reports quickly by writing simple Python
- Pluggable data source infrastructure — use the built-in modules or write one for your own particular needs
- Powerful caching system allows for fast and easy sorting and paging
- Drop-in web frontend for running and displaying reports to your users

Whatever business initiatives and performance indicators you want to look at, Blingalytics offers the tools you need to slice and dice your data. Maybe you want to see the number of retweets your clients get over time. Or maybe you simply want to track your revenue and other financial indicators. Perhaps you need to track how many virtual tractors your virtual farmers are purchasing. Blingalytics can probably help.

Thanks to Adly for fostering the development of Blingalytics as an open-source project. Blingalytics is released under the MIT License.

CHAPTER 2

Getting Help

If you've already blown through the docs and haven't found the answers you're looking for, there are a number of other channels where you can get help.

- **Mailing List**: If you need help with Blingalytics, you can post a question to our mailing list. The developers will do our best to reply promptly.
- **Twitter**: We maintain an official Twitter account, @blingalytics, where we'll announce official news. You can also tweet at us and see what happens!
- **IRC**: We'll hang out on the `#blingalytics` channel on Freenode, where we'll try to help answer any questions.

CHAPTER 3

Contents

3.1 Installation

3.1.1 Install with pip

Simply use pip to install the blingalytics package:

```
pip install blingalytics
```

3.1.2 Install from source

Download or clone the source from Github and run setup.py install:

```
git clone git@github.com:adly/blingalytics.git
cd blingalytics
python setup.py install
```

3.1.3 Requirements

For a very basic infrastructure, Python is the only dependency:

- Python >= 2.5

Some of the caches and sources have other dependencies, but they are only required if you're actually using those particular source and cache engines. The documentation for that module will specify what its dependencies are, which may include:

- Redis
- redis-py
- SQLAlchemy
- Elixir

3.2 Development

3.2.1 Get the code

Fork us on Github! You can find us at https://github.com/adly/blingalytics.

3.2.2 Contributing

There are a few ways you can get involved with the development of Blingalytics.

Use Blingalytics!

Use the code to build your business' reporting system and let us know how it goes. We want to know how you're using it, what problems you encountered, and what features you'd love to see.

Report bugs

If you think you've found a bug in the code, please check our ticket tracker to see if anyone else has reported it yet. If not, file a bug report! Standard issue reporting standards apply: please succinctly describe the steps to reproduce and what the issue is. For now, our issues are on our Github issues page.

Contribute patches and new features

If you've fixed a bug or have a feature that you think would be useful to the community, send us a pull request on Github.

3.2.3 Planned features

A list, in pseudo-random order, of some features we're planning. Some of these are already in use at Adly and just need some polish before they're ready to be included. Others are brand-new features the community would benefit from.

- **Front-end implementation.** Internally we use an AJAXy front-end implementation built on top of YUI's DataTable. We'd like to make this generic enough that it's a snap to get your own project up and running with it.
- **Django app.** Having a quick and easy app to plug into your Django project would be super. This would involve a new source to interface with Django's ORM, perhaps a new cache interface to Django's caching framework, and probably some other niceties.
- **More included sources.** While sources are pluggable enough that you can write one for your own purposes, there are some common sources that we'd like to support. We've considered adding sources that pull from a MapReduce, from a public web API, and from various other datastores like MongoDB.
- **Memcache cache implementation.** We use Redis extensively at Adly, and it works incredibly well for caching and crunching report data. But I expect people are more familiar with Memcache and more likely to already have it set up in their infrastructure. It probably won't be as memory-efficient, but it should be possible to create an implementation if there's enough of a use case.
- **Others?** If you have other features you think would be useful, please mention them on the mailing list!

3.3 Sweet walkthrough

This intro should get you up and running with a basic Blingalytics installation in just a few minutes. Once you're done, you can check out the documentation for *Data Sources* and *Cache stores* to contemplate your options for snappier infrastructure.

If you haven't done so yet, you'll need to install the `blingalytics` package and requirements before beginning. See *Installation* for details.

3.3.1 One: define a report

With Blingalytics, you use a "report" definition to describe precisely what data you want to look at and how you want to slice it. In a report definition, you're only required to do two things:

- List the output columns and where they should get their data
- Define the key range for the report (explained below)

Baby's first report

So to start, let's put together a completely useless simplest-case report:

```
from blingalytics import base, formats
from blingalytics.sources import key_range, static

class LameReport(base.Report):
    keys = ('lame', key_range.SourceKeyRange)
    columns = [
        ('lame', static.Value(5, format=formats.Integer)),
    ]
```

So what does this report do? It provides one output column, whose value will always be `5`. However, that's not even the most useless property of this report, as this report will actually return zero rows. This is why a report's keys matter.

Key concept

The keys for a report determine what rows will be in the output. If your website sells doodads, you might want to see how many doodads you sell per day. In this case, the report keys would be the range of days you report on so that you get one row per day. If, on the other hand, you want to see how many doodads each user has bought, you would want one row per user. So the user ID is your report key, and the range would be all your users.

To specify the key range for your report, you set the keys attribute of your report class to a two-tuple. The first item is the label of your key column, and the second item is the type of key range. In our example, the `sources.SourceKeyRange` tells the report to only include rows returned by the source data. Other key ranges, such as a range of days, can be used to ensure that a row is returned for each key, even if there is no source data.

For advanced use cases, you can even have compound keys. For example, you could have a row per user per day. See *Creating and using reports* for more.

Build your columns

Columns are defined in a report as a list of two-tuples. Each two-tuple represents a column, in order, by defining a label and its data source. The label should be unique among the columns, and is used by keys and other options to

reference that column. The data source defines how that column's data should be computed, and is covered in more detail in *Data Sources*.

A slightly realer report

Now that we know a bit more about how this works, let's define a report that actually does something (to be fair, it's still pretty useless, but we're getting closer):

```
from blingalytics import base, formats
from blingalytics.sources import derived, key_range, static

class RealerReport(base.Report):
    keys = ('prime', key_range.IterableKeyRange([2, 3, 5, 7, 11, 13, 17]))
    columns = [
        ('prime', key_range.Value(format=formats.Integer)),
        ('squared', derived.Value(lambda row: row['prime'] ** 2, format=formats.Integer)),
        ('note', static.Value('Useful data coming soon...', format=formats.String)),
    ]
    default_sort = ('prime', 'asc')
```

So now we've got a report with three columns: prime is one of the prime numbers from the key range; note is simply a static string value; and squared is the square of the prime number. OK, time to run it!

3.3.2 Two: run the report

Once you've defined a report, such as `RealerReport`, you can instantiate the report and tell it where to cache the data:

```
from blingalytics.caches.local_cache import LocalCache
report = RealerReport(LocalCache())
```

Once you have a report instance, you can run the report:

```
report.run_report()
```

Retrieving report rows

Now that the report is cached, you can retrieve the data with limits, offsets, column sorting, and so on. But in the simplest case, you can just get all the rows:

```
rows = report.report_rows()
# rows = [
#     [1, '2', '4', 'Useful data coming soon...'],
#     [2, '3', '9', 'Useful data coming soon...'],
#     [3, '5', '25', 'Useful data coming soon...'],
#     [4, '7', '49', 'Useful data coming soon...']]
#     [5, '11', '121', 'Useful data coming soon...'],
#     [6, '13', '169', 'Useful data coming soon...'],
#     [7, '17', '289', 'Useful data coming soon...'],
# ]
```

Let's try sorting and limiting the data:

```
rows = report.report_rows(sort=('squared', 'asc'), limit=3)
# rows = [
#     [7, '17', '289', 'Useful data coming soon...'],
```

```
#     [6, '13', '169', 'Useful data coming soon...'],
#     [5, '11', '121', 'Useful data coming soon...'],
# ]
```

There are plenty more options for retrieving specific rows. See `Report.report_rows` for more.

3.3.3 Three: put it in the browser

Working in the Python interpreter is nice and all, but most of us want to insert this thing into a beautiful website and spread it around the web. Good news! Blingalytics comes with the tools to pull this off in just a few lines of code. (For example purposes, this will be shown as a Flask app, but it should be easy enough to insert this into your favorite Python web framework.)

The report app

Let's assume you've already *installed Blingalytics* and installed Flask.

Now we build a very basic Flask app that has two URLs: a homepage, where we'll display our report; and an AJAX responder for our report JavaScript to talk to. The `report_response` helper function makes responding to the AJAX requests easy.

```
from blingalytics.helpers import report_response
from flask import Flask, request, render_template
from reports import RealerReport # import so it gets registered

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/report/')
def report():
    return report_response(request.args)

if __name__ == '__main__':
    app.run(debug=True)
```

The `index` page really just renders the template where you're going to implement the Blingalytics frontend. The `report` URL handles AJAX requests from the JavaScript frontend, and the `report_response` handles all the dirty work of parsing request parameters and interfacing with your report classes.

The report template

The template is even easier. Just include the appropriate CSS and JavaScript on your page, and invoke the Blingalytics jQuery plugin:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="/static/css/blingalytics.css" type="text/css" />
</head>
<body>
  <div id="report"></div>
  <script src="//ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js"></script>
```

```
  <script src="/static/js/jquery.dataTables.min.js"></script>
  <script src="/static/js/jquery.blingalytics.js"></script>
  <script>
    $('#report').blingalytics({
      'reportCodeName': 'realer_report'
    });
  </script>
</body>
</html>
```

With the CSS and JavaScript in place, you just have to pass in the `'reportCodeName'` option with the code name of the report you want to display.

Partake in the beauty

That's it! Run your web app. For Flask, this involves running `python app.py` at the command line, where `app.py` is the app file you created earlier. Then you should be able to point your browser to `localhost:5000` and play with the report in the browser.

3.3.4 Four: pull real data

Be patient... coming soon.

3.4 Creating and using reports

Reports are used to define a view into your data. At a basic level, reports are used to process a data set, format the data, and return tabular results. But reports can define many options, including:

- The name of the report
- The category of the report
- How long to cache the report
- What source data to pull
- How to manipulate the source data
- How the output should be formatted
- What options are given to the user to filter the data
- The default sorting

The next section describes in detail how to write and interact with your own reports. You will also want to check out *Data Sources* for details on how to pull data; *Column formatters* for more on formatting report output; and *User-input widgets* for details on accepting user filtering options.

3.4.1 Report objects

class `blingalytics.base.`**`Report`**(*cache*, *merge=False*)

To write a report, you subclass this base Report class and define your own attributes on it. The standard list of available attributes include:

category ***(optional)*** A string representing the category this report belongs to. The category is used by the get_reports_by_category function to group like reports together.

display_name ***(optional)*** The title of the report as a string. If not specified, this will be generated automatically based on the class name.

code_name ***(optional)*** The code name of the report as a string. The code name is used to identify this report, for example as part of a key name when caching the report's data. If not specified, this will be generated automatically based on the class name.

cache_time ***(optional)*** The number of seconds this report should remain valid in the cache. If not specified, defaults to `1800` (30 minutes).

keys If your report has just one key, this should be a two-tuple: the name of the key column as a string; and the desired key range class or instance. If you want a report with compound keys, you can specify them as a list of these two-tuples. This is described in detail in *Key range source*.

columns This should be a list of two-tuples describing your report's columns. The first item of each two-tuple is the name of the column as a string. The second item is a column class or instance. The column definitions do all the heavy lifting of pulling data from sources, manipulating it, and formatting the output. For more details, see *Data Sources*.

filters ***(optional)*** A list of two-tuples describing the filters and widgets to present to your users. The first item of each two-tuple is the name of the filter as a string. The second item is a filter instance. The types of filters you can use are specific to the sources you're using; see the relevant source's documentation in *Data Sources*. Filters will also generally specify a widget for collecting the user input; for more, see *User-input widgets*.

default_sort ***(optional)*** A two-tuple representing the column and direction that the report should be sorted by default. The first item is the name of the column to sort on, as a string. The second is the sort direction, as either `'asc'` or `'desc'`. If not specified, this defaults to sorting by the first column, descending.

Various sources used by the report may allow or require other attributes to be specified. This will be specified in the documentation for that source.

Here is a relatively simple example of a report definition:

```
from blingalytics import base, formats, widgets
from blingalytics.sources import database, derived, key_range

class RevenueReport(base.Report):
    display_name = 'Company Revenue'
    code_name = 'company_revenue_report'
    category = 'business'
    cache_time = 60 * 60 * 3 # three hours

    database_entity = 'project.models.reporting.RevenueModel'
    keys = ('product_id', key_range.SourceKeyRange)
    columns = [
        ('product_id', database.GroupBy('product_id',
            format=formats.Integer(label='ID', grouping=False), footer=False)),
        ('product_name', database.Lookup('project.models.products.Product',
            'name', 'product_id', format=formats.String)),
        ('revenue', database.Sum('purchase_revenue', format=formats.Bling)),
        ('_cost_of_revenue', database.First('product_cost')),
        ('gross_margin', derived.Value(
            lambda row: (row['revenue'] - row['_cost_of_revenue']) * \
            Decimal('100.00') / row['revenue'], format=formats.Bling)),
    ]
    filters = [
        ('delivered', database.QueryFilter(
```

```
            lambda entity: entity.is_delivered == True)),
        ('online_only', database.QueryFilter(
            lambda entity, user_input: entity.is_online_purchase == user_input,
            widget=widgets.Checkbox(label='Online Purchase'))),
    ]
    default_sort = ('gross_margin', 'desc')
```

Once you have defined your report subclass, you instantiate it by passing in a cache instance. This tells the report how and where to cache its processed data. For more, see *Cache stores*. Once you have a report instance, you use the following methods to run it, manipulate it, and retrieve its data.

classmethod render_widgets()
Returns a list of this report's widgets, rendered to HTML.

classmethod get_widgets()
Returns a list of this report's widget instances. Calling the widgets' `render()` method is left to you.

clean_user_inputs(***kwargs*)
Set user inputs on the report, returning a list of any validation errors that occurred.

The user input should be passed in as keyword arguments the same as they are returned in a GET or POST from the widgets' HTML. The widgets will be cleaned, converting to the appropriate Python objects.

If there were errors, they are returned as a list of strings. If not, returns `None` and stores the user inputs on the report. Note that this effectively changes the `unique_id` property of the report.

If your report has user input widgets, this should be called before run_report; if the report has no widgets, you don't need to call this at all.

run_report()
Processes the report data and stores it in cache.

Depending on the size of the data and the processing going on, this call can be time-consuming. If you are deploying this as part of a web application, it's recommended that you perform this step outside of the request-response cycle.

is_report_started()
If `run_report()` is currently running, this returns `True`; otherwise, `False`.

is_report_finished()
If `run_report()` has completed and there is a current cached copy of this report, returns `True`; otherwise, `False`.

kill_cache(*full=False*)
By default, this removes or invalidates (depending on the cache store being used) this cached report. If there are other cached versions of this report (with different user inputs) they are left unchanged.

If you pass in `full=True`, this will instead perform a full report-wide cache invalidation. This means any version of this report in cache, regardless of user inputs, will be wiped.

report_header()
Returns the header data for this report, which describes the columns and how to display them.

The header info is returned as a list of dicts, one for each column, in order. Certain column types may add other relevant info, but by default, the header dicts will contain the following:

- `label`: The human-readable label for the column.
- `alignment`: Either `'left'` or `'right'` for the preferred text alignment for the column.
- `hidden`: If `True`, the column is meant for internal use only and should not be displayed to the user; if `False`, it should be shown.

- `sortable`: If `True`, this column can be sorted on.

The first column returned by a Blingalytics report is always a hidden column specifying the row's cache ID. The first column header is for this internal ID.

`report_rows`(*selected_rows=None*, *sort=None*, *limit=None*, *offset=0*, *format='html'*)
Returns the requested rows for the report from cache. There are a number of arguments you can provide to limit, order and format the rows, all of which are optional:

- `selected_rows`: If you want to limit your results to a subset of the report's rows, you can provide them as a list of internal cache row IDs (the ID of a row is always returned in the row's first column). Defaults to `None`, which does not limit the results.
- `sort`: This is a two-tuple to specify the sorting on the table, in the same format as the `default_sort` attribute on reports. That is, the first element should be the label of the column and the second should be either `'asc'` or `'desc'`. Defaults to the sorting specified in the report's `default_sort` attribute.
- `limit`: The number of rows to return. Defaults to `None`, which does not limit the results.
- `offset`: The number of rows offset at which to start returning rows. Defaults to `0`.
- `format`: The type of formatting to use when processing the output. The built-in options are `'html'` or `'csv'`. Defaults to `'html'`. This is discussed in more detail in *Column formatters*.

The rows are returned as a list of lists of values. The first column returned by Blingalytics for any report is always a hidden column specifying the row's internal cache ID.

`report_footer`(*format='html'*)
Returns the computed footer row for the report. There is one argument to control the formatting of the output:

- `format`: The type of formatting to use when processing the output. The built-in options are `'html'` or `'csv'`. Defaults to `'html'`. This is discussed in more detail in *Column formatters*.

The footer row is returned as a list, including the data for any hidden columns. The first column returned by Blingalytics for any report is reserved for the row's internal cache ID, so the first item returned for the footer will always be `None`.

`report_timestamp`()
Returns the timestamp when the report instance was originally computed and cached, as a `datetime` object.

`report_row_count`()
Returns the total number of rows in this report instance.

3.4.2 Utility functions

The `blingalytics` module provides a few utility methods that are useful for retrieving your reports.

Note: The report registration method is done using a metaclass set on the `base.Report` class. This means that the modules where you've defined your report classes have to be imported before the methods below will know they exist. You can import them when your code initializes, or right before you call the utility functions, or whatever — just be sure to do it.

`blingalytics.`**`get_report_by_code_name`**(*code_name*)
Returns the report class with the given `code_name`, or `None` if not found.

blingalytics.**get_reports_by_category**()
Returns all known reports, organized by category. The result is returned as a dict of category strings to lists of report classes.

3.5 Data Sources

In Blingalytics, sources implement an interface between the report and the original source data.

For example, the database source provides an interface for doing sums and counts over database columns; the derived source allows you to perform calculations over columns from other sources; and the merge source allows you to pull data from other reports and produce a sort of meta-report.

When writing a report, the sources you choose to use will provide you with some or all of the following:

- **Column types**, which define how and where the source gets the data for that column.
- **Filters**, which you or the end user (through user input widgets) can use to filter the source data for the report.
- **Key ranges**, which the report uses to determine which rows to produce.

The documentation for each of the sources provides more in-depth information.

3.5.1 Standard column options

Most columns will require or accept certain arguments specific to their use case, but all column types accept the following optional standard arguments:

- `format`: A format class or instance that should be used to determine the formatting and display options for this column. If omitted, the column defaults to being hidden. See *Column formatters* for more.
- `footer`: Most numeric data columns will calculate a sum for the footer by default. String data columns have no footer by default. Each column's documentation will specify if it has any non-standard footer handling. For any column, if you wish to disable the footer, you can set this to `False`.

3.5.2 Standard filter options

Most filters also require or accept arguments specific to their use case, but all filters accept the following optional standard arguments:

- `columns`: If one or more columns are provided, the filter should only be applied to the given columns. Other columns should remain unaffected. The columns may be specified as a string (for just one column) or a list of strings, specifying the column names to filter. Defaults to `None`, which applies the filter to all columns in the report.
- `widget`: A widget class or instance that defines the widget type the user should be shown to input a filter argument. Defaults to `None`. See *User-input widgets* for more.

3.5.3 Available sources

Key range source

The key range source provides many of the standard key ranges you'll use frequently. It also provides a column for outputting the values returned by any key range.

Column types

class blingalytics.sources.key_range.**Value**(*format=None*, *footer=True*)
Occasionally you may want to simply have a column show the values returned by a given key range, such as with an `IterableKeyRange`. This column requires no special options, and returns the key value from the key whose name matches the name of this column.

Key ranges

class blingalytics.sources.key_range.**SourceKeyRange**
This key range doesn't actually ensure any keys. It simply allows each key value returned by the sources to be a new row. If it isn't returned by a source column, it won't be a row.

class blingalytics.sources.key_range.**IterableKeyRange**(*iterable*, *sort_results=True*)
Ensures every value returned by the iterable is in the key range. It takes one required argument, which is the iterable to use.

Note that this iterable must be returned in sorted order. By default, this key range will sort the iterable for you before it is returned. However, if your iterable is already in sorted order and you want to avoid the overhead of resorting the list, can pass in `sort_results=False`.

class blingalytics.sources.key_range.**EpochKeyRange**(*start*, *end*)
Ensures a key for every day between the start and end dates.

This key range takes two positional arguments, start and end, which are used to determine the range of days. These arguments can be datetimes, which will be used as-is; or they can be strings, which will be considered as references to named widgets, and the user input from the widget will be used for the date.

The values of the keys returned by this key range are in the form of an integer representing the number of full days since the UNIX epoch (Jan. 1, 1970). This is ideal for use with the `Epoch` formatter.

class blingalytics.sources.key_range.**MonthKeyRange**(*start*, *end*)
Ensures a key for every month between the start and end dates.

This key range takes two positional arguments, start and end, which are used to determine the range of months. These arguments can be datetimes, which will be used as-is; or they can be strings, which will be considered as references to named widgets, and the user input from the widget will be used for the date.

The values of the keys returned by this key range are in the form of an integer representing the number of full days since the UNIX epoch (Jan. 1, 1970). This is ideal for use with the `Epoch` formatter.

Database source

Column types

Filter types

Key ranges

Static source

The static source provides a column for inserting static data. It's incredibly simple and not terribly useful, but it can come in handy from time to time. For example, if you want to fill a column with a 'coming soon...' message, you could use a static column.

Column types

class blingalytics.sources.static.**Value**(*value*, ***kwargs*)

Returns a given value for each row in the report. In addition to the standard column options, it takes one positional argument, which is the static value to return for every row.

Derived source

The derived source allows you to perform arbitrary operations using the data returned in columns from other sources.

For example, if you have a report with a column for gross revenue and a column for net revenue that are both pulled from the database, you could have a derived column to display the gross margin by performing the operation `(net revenue / gross revenue * 100)`.

Column types

class blingalytics.sources.derived.**Value**(*derive_func*, ***kwargs*)

A column that derives its value from other columns in the row. In addition to the standard column options, this takes one positional argument: the function used to derive the value.

The function you provide will be passed one argument: the row, as pulled from other data sources but before the `post_process` step. The row is a dict with the column names as keys. Your function should return just the derived value for this column in the row. The function is often provided as a lambda, but more complex functions can be defined wherever you like.

Continuing the example from above:

```
derived.Value(lambda row: row['net'] / row['gross'] * Decimal('100.00'))
```

By default, the footer for this column performs the same operation over the appropriate footer columns. This is generally the footer you want for a derived column, as opposed to simply summing or averaging the values in the column. If one of the columns involved in the derive function does not return a footer, this will return a total.

Merge source

The merge source provides a mechanism for merging and filtering the data resulting from two or more other "sub-reports". This can be useful if you need to combine results from two different databases or with two different key ranges into one report.

When building a merged report, you must specify the reports to merge in a report attribute:

- `merged_reports`: Specifies the sub-reports you want to merge. Provide the reports as a dict with the keys naming the sub-reports and the values being the sub-report classes themselves. For example:

```
merged_reports = {
    'revenue': ProductRevenueReport,
    'awesome': UserAwesomenessReport,
}
```

All merge columns take the same positional arguments, which are used to specify which columns from sub-reports should be combined into the merge column. You can specify the merge columns as follows:

- If no argument is provided, the merge report will merge any columns found with the same name from all sub-reports.

- If you provide a single string as an argument, the merge report will merge any columns from sub-reports that have that name.
- To merge columns of various names from various sub-reports, you can specify as many as you like as dotted strings. Each string should be in the form of `'report_name.column_name'`.

As a merged report is processed, it will actually run the full end-to-end `run-report` process for each of its sub-reports. It will then aggregate the results together based on the columns and filters in the merge report.

Column types

class `blingalytics.sources.merge.`**`Sum`**(**args*, ***kwargs*)
Merges sub-report columns by summing the values returned by each sub-report. Takes the standard merge column arguments, as described above.

class `blingalytics.sources.merge.`**`First`**(**args*, ***kwargs*)
Merges sub-report columns by keeping the first value returned by any sub-report. Takes the standard merge column arguments, as described above.

class `blingalytics.sources.merge.`**`BoolAnd`**(**args*, ***kwargs*)
Merges sub-report columns by performing a boolean-and operation over the values returned by the sub-reports. This returns `True` if *all* the values from sub-reports are truthy; otherwise, it returns `False`. A `None` value is considered `True` here so that they essentially are ignored in determining the result. Takes the standard merge column arguments, as described above.

class `blingalytics.sources.merge.`**`BoolOr`**(**args*, ***kwargs*)
Merges sub-report columns by performing a boolean-or operation over the values returned by the sub-reports. This returns `True` if *any* of the values from sub-reports are truthy; otherwise, it returns `False`. A `None` value is considered `False` here so that they essentially are ignored in determining the result. Takes the standard merge column arguments, as described above.

Filter types

class `blingalytics.sources.merge.`**`DelegatedFilter`**(**args*, ***kwargs*)
Allows you to display one widget from the merge report, then supply the user input to each of the sub-reports to process as they normally would using the filters defined on each sub-report.

This filter simply takes the standard widget keyword argument. You also need to ensure that the name of this filter matches the name of any sub-report filters you want to pick up the user input.

class `blingalytics.sources.merge.`**`PostFilter`**(*filter_func*, ***kwargs*)
This filter can be used to exclude entire rows from the output, based on the data in the row from the sub-reports.

This takes one positional argument, which should be a filtering function. The function takes the merged row as a dict, and optionally the user input if a widget is specified. If the function returns a truthy value, the row will be included; otherwise, it will be skipped. For example:

```
merge.PostFilter(
    lambda row, user_input: row['revenue'] >= user_input,
    widget=widgets.Select(choices=MIN_REVENUE_CHOICES))
```

class `blingalytics.sources.merge.`**`ReportFilter`**(*report_name*, ***kwargs*)
This filter allows you to selectively include or exclude specific sub-reports from being processed at all. It takes one positional argument: the name of the sub-report from the `merged_reports` report attribute from the merge report.

This filter requires a widget (it would be pretty silly to use this one without a widget). It will include the specified sub-report in the merged output only if the user input is truthy. Generally, a `Checkbox` widget is appropriate for this.

3.6 Cache stores

Cache engines are used as an intermediate store for computed report data. Since pulling and processing the data for a report can be computationally intensive, this cached storage allows us to sort and page through the resulting report very quickly and easily.

Currently, the easiest to get up and running is *Filesystem caching*, as it has no dependencies outside of Python and simply caches to the local filesystem. However, it cannot handle concurrent connections and is generally a poor choice outside of the development environment. At the moment, the preferred choice for deployment is *Redis caching*.

3.6.1 Available cache engines

Redis caching

The Redis-based cache engine provides flexible, powerful storage for computed reports. It is currently the recommended caching option for anything beyond a simple dev environment.

Note: The Redis cache requires Redis and its Python bindings to be installed. See *Installation* for details.

---

class `blingalytics.caches.redis_cache.`**`RedisCache`**(***kwargs*)

Caches computed reports in Redis. This takes the same init options as the redis-py client. The most commonly used are:

- `host`: The host to connect to the redis server on. Defaults to `'localhost'`.
- `port`: The port to use when connecting. Defaults to `6379`.
- `db`: Which Redis database to connect to, as an integer. Defaults to `0`.

Filesystem caching

The local filesystem cache engine provides the simplest, no-fuss option for setting up Blingalytics. It stores report caches in a file using SQLite, which is included in Python.

While the ease of use is great, the SQLite solution presents a number of limitations:

- *Poor concurrency support*. While you can have multiple clients reading from a SQLite database, any write connection requires a lock on the entire database. If you have more than one end user, use *Redis caching* instead.
- *Poor network access*. It's technically possible to set up remote access to your SQLite databse, but it's really not recommended. So if you wanted to have multiple machines accessing the cache, use *Redis caching* instead.

You've been warned. Great for dev, poor for everything else.

class `blingalytics.caches.local_cache.`**`LocalCache`**(*database='/tmp/blingalytics_cache'*)

Caches the files locally on the filesystem. Takes just one optional argument:

- `database`: This is the file where the cache database will be created. Defaults to `/tmp/blingalytics_cache`. Note that this cache will not work with SQLite's in-memory database option.

3.7 User-input widgets

Widgets provide a mechanism for displaying basic HTML inputs to the user and then cleaning the input and passing it into a report.

All widgets accept the following parameters as keyword arguments. Some widgets may accept or require additional arguments, which will be specified in their documentation.

- `label`: The label the user sees for this widget. Defaults to `'Filter'`.
- `default`: A default value that is initially displayed in the widget. This can be a callable, which will be evaluated lazily when rendering the widget. Defaults to `None`.
- `required`: Whether the input can be left blank. If `True`, a blank value will be added to the errors produced by the report's `clean_user_inputs` method. By default, user input is not required, and a blank value will be returned as None.
- `extra_class`: A string or list of strings that will add extra HTML classes to the rendered widget. Defaults to no extra classes.
- `extra_attrs`: A dict of attribute names to values. These will be added as attributes on the rendered widget. Defaults to no extra attributes.

3.7.1 Widget types

class `blingalytics.widgets.`**`Checkbox`**(*label=None*, *default=None*, *required=False*, *extra_class=None*, *extra_attrs=None*)

Produces a checkbox user input widget. The widget's default value will be evaluated as checked if it's truthy and unchecked if it's falsy.

class `blingalytics.widgets.`**`DatePicker`**(*date_format='%m/%d/%Y'*, *end_of_day=False*, ***kwargs*)

Produces a text input to build into a datepicker widget. It accepts one additional optional argument:

- `date_format`: This is the string passed to the `datetime.strptime` function to convert the textual user input into a datetime object. Defaults to `'%m/%d/%Y'`.

Note that this widget is rendered simply as an HTML text input with a class of `'bl_datepicker'`. It is left up to you to throw a JavaScript datepicker widget on top of it with jQuery or whatever.

For this widget, the `default` argument can be:

- A string in the format specified by the `date_format` option.
- A `date` or `datetime` object.
- One of the following special strings: `'today'`, `'yesterday'`, `'first_of_month'`.
- A callable that evaluates to any of the previous options.

class `blingalytics.widgets.`**`Select`**(*choices=[]*, ***kwargs*)

Produces an select input widget. This takes one additional optional argument:

- `choices`: A list of two-tuples representing the select options to display. The first item in each tuple should be the "cleaned" value that will be returned when the user selects this option. The second item should be the label to be displayed to the user for this option. This can also be a callable. Defaults to `[]`, an empty list of choices.

For the `default` argument for this type of widget, you provide an index into the choices list, similar to how you index into a Python list. For example, if you have three select options, you can default to the second option by passing in `default=1`. If you want the last selection to be default, you can pass in `default=-1`.

class blingalytics.widgets.**Autocomplete**(*multiple=False*, ***kwargs*)

Produces a text input for an autocomplete widget. Unlike most widgets, this **does not** accept the `default` argument. It takes the following extra argument:

- `multiple`: Whether the autocomplete should accept just one or multiple values. When set to `True`, this will add a class of `'bl_multiple'` to the widget. Defaults to `False`.

Note that this widget is rendered simply as an HTML text input with a class of `'bl_autocomplete'`, and optionally a class of `'bl_multiple'`. It is left up to you to throw a JavaScript autocompletion widget on top of it with jQuery or whatever.

The cleaned user input will be coerced to a list of integer IDs.

3.8 Column formatters

Formats define how values in a report should be output for display to the user. They generally take the underlying Python value and convert it into a string appropriate for display in the given medium. For example, an internal value of `Decimal('9.99')` might be converted to a display value of `'$9.99'` for monetary values.

All formatters accept the following optional keyword arguments:

- `label`: A string that will be used as the label for the column, as returned by the `report_header` method. By default, this will be automatically generated based on the column name.
- `align`: Either `'left'` or `'right'`, used to determine the preferred alignment of the column data. If this is not supplied, the formatter will use its default alignment. Generally, number-type columns default to right-aligned, while other columns default to left-aligned.

Note: Many of the formatters rely on Python's `locale` module to perform locale-dependent formatting. For example, if your locale is set to `'en_US'`, monetary values will be formatted as `'$1,234.00'`. If your locale is set to `'en_GB'`, monetary values will be formatted as `'£1,123.00'`. Your locale is set per Python thread, and should be set somewhere in your code that guarantees it will be run prior to any formatting operations. See Python's documentation for the `locale` module for more.

3.8.1 Output formats

As a base, all formats have formatter functions defined for two types of output: HTML and CSV. You can pass in `'html'` (the default) or `'csv'` as the format option to your report's `report_rows` and `report_footer` methods to format the output appropriately.

But you are not limited to HTML and CSV output formatting. If you want to format values differently for another medium, you can subclass any formats you use and add a `format_NAME` method. For example, you could add a `format_pdf` method. You would then pass in `'pdf'` as the format option to your report's `report_rows` and `report_footer` methods. See the docstring and code of the base `Format` class for more.

3.8.2 Formatter types

class blingalytics.formats.**Hidden**(*label=None*, *align=None*, *sortable=True*)

No particular formatting is performed, and this column should be hidden from the end user. This column will be marked as hidden in the header data returned by your report's `report_header` method.

A hidden column is used for the internal ID returned with each row. You could also use these yourself to, for example, pass along a URL that gets formatted into a nice-looking link with some front-end JavaScript.

class blingalytics.formats.**Bling**(*label=None*, *align=None*, *sortable=True*)
Formats the column as monetary data. This is formatted as currency according to the thread-level locale setting. If being output for HTML, it will add grouping indicators (such as commas in the U.S.). If being output for CSV, it will not. By default, the column is right-aligned.

For example, in the `'en_US'` locale, numbers will be formatted as `'$1,234.56'` for HTML or `'$1234.56'` for CSV.

class blingalytics.formats.**Integer**(*grouping=True*, ***kwargs*)
Formats the data as an integer. This formatter accepts one additional optional argument:

- `grouping`: Whether or not the formatted value should have groupings (such as a comma in the U.S.) when output for HTML. For example, when representing the number of pageviews per month, you would probably want separators; however, for an database ID you probably don't. Defaults to `True`.

This formatting is based on the Python thread-level `locale` setting. For example, in the `'en_US'` locale, numbers will be formatted as `'1,234'` for HTML or `'1234'` for CSV. By default, the column is right-aligned.

class blingalytics.formats.**Percent**(*precision=1*, ***kwargs*)
Formats the data as a percent. This formatter accepts one additional optional argument:

- `precision`: The number of decimal places of precision that should be kept for display. Defaults to `1`.

This is formatted as a decimal number with a trailing percent sign. For example, numbers will be formatted as `'12.3%'` with a precision of `1`. By default, this column is right-aligned.

class blingalytics.formats.**String**(*title=False*, *truncate=None*, ***kwargs*)
Formats column data as strings. Essentially, this will simply coerce values to strings. It also accepts a couple optional formatting parameters:

- `title`: If `True`, will title-case the string. Defaults to `False`.
- `truncate`: Set this to an integer value to truncate the string to that number of characters. Adds an ellipsis if truncation was performed. Defaults to not performing any truncation.

This column is left-aligned by default.

class blingalytics.formats.**Boolean**(*terms=('Yes', 'No', '')*, ***kwargs*)
Formatter for boolean data. This coerces the value to a boolean, and then presents a string representation of whether it's true or false. It accepts one optional argument:

- `terms`: This is a tuple of three strings that determine how true, false, and null values will be represented, in that order. By default, it uses `('Yes', 'No', '')`.

This column is left-aligned by default.

class blingalytics.formats.**Epoch**(*label=None*, *align=None*, *sortable=True*)
Formats the column as a date. Expects the underlying data to be stored as an integer number of days since the UNIX epoch. (This storage method works great in conjunction with a `database.ColumnTransform` filter for doing timezone offsets.)

This date is formatted according to the Python thread's `locale` setting. For example, in the `'en_US'` locale, a date is formatted as '01/23/2011'. By default, the column is left-aligned.

class blingalytics.formats.**Date**(*format=None*, ***kwargs*)
Formats the column as a date. Expects the underlying data to be stored as a Python `date` or `datetime` object. This date is formatted according to the Python thread's `locale` setting. For example, in the `'en_US'` locale, a date is formatted as '01/23/2011'. By default, the column is left-aligned.

class blingalytics.formats.**Month**(*label=None*, *align=None*, *sortable=True*)
Formats the column as a month. Expects the underlying values to be Python `datetime` or `date` objects. For example, `'Jan 2011'`.

3.9 Frontend Implementation

The most common use case for Blingalytics is to be displayed on a web page, so we wanted to provide a pre-baked solution that can get you up and running in minutes. You're welcome to tweak it or even roll your own, but this is a great starting point.

3.9.1 In the HTML

To implement the Blingalytics frontend on your site, the first thing you'll need to do is include the appropriate CSS and JavaScript files on the page. These static files are included under `blingalytics/statics/css/` and `blingalytics/statics/js/` and should be made available by your server.

CSS to include:

```
<link rel="stylesheet" href="/static/css/blingalytics.css" type="text/css" />
```

JavaScript to include:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js"></script>
<script src="/static/js/jquery.dataTables.min.js"></script>
<script src="/static/js/jquery.blingalytics.js"></script>
```

Once you've included the static dependencies on the page, you can use the blingalytics jQuery plugin to insert a report table anywhere on your page:

```
<script>
    jQuery('#selector').blingalytics({'reportCodeName': 'report_code_name'});
</script>
```

For now the plugin only accepts two options:

`reportCodeName` ***(optional)*** The `report class code_name` attribute. This specifies which report should be displayed on the page. Defaults to `'report'`.

`url` ***(optional)*** The URL to hit when contacting the server for an AJAX response. Defaults to `'/report/'`.

3.9.2 On the backend

The blingalytics jQuery plugin inserts a bunch of HTML and JavaScript that talks over AJAX with your server. So your server, at the `url` you specify in the plugin options, should respond appropriately. To make this easy, a Python helper function is provided.

`blingalytics.helpers.`**`report_response`**(*params*, *runner=None*, *cache=<LocalCache /tmp/blingalytics_cache>*)

This frontend helper function is meant to be used in your request-processing code to handle all AJAX responses to the Blingalytics JavaScript frontend.

In its most basic usage, you just pass in the request's GET parameters as a `dict`. This will run the report, if required, and then pull the appropriate data. It will be returned as a JSON string, which your request-processing code should return as an AJAX response. This will vary depending what web framework you're using, but it should be pretty simple.

The function also accepts two options:

`runner` ***(optional)*** If you want your report to run asynchronously so as not to tie up your web server workers while processing requests, you can specify a runner function. This should be a function that will initiate your async processing on a tasks machine or wherever. It will be passed two arguments: the report

code_name, as a string; and the remaining GET parameters so you can process user inputs. By default, no runner is used.

`cache` ***(optional)*** By default, this will use a local cache stored at `/tmp/blingalytics_cache`. If you would like to use a different cache, simply provide the cache instance.

Python Module Index

b