Trump Documentation

Release 0.0.5

Jeffrey McLarty

July 07, 2016

Contents

CHAPTER 1

Introduction

Trump is a framework for objectifying data, with the goal of centralizing the management of data feeds to enable quicker deployment of analytics, applications, and reporting. Munging data, common calculations, validation of data, can all be handled by Trump, upstream of any application or user requirement.

Inside the Trump framework, a symbol refers to one or more data feeds, each with their own instructions saved for retrieving data from a specific source. Once it's retrieved by Trump, depending on the attributes of the symbol, it gets munged, aggregated, checked, and cached. Downstream users are free to query the existing cache, force a re-cache, or check any property of the data prior to using it.

System Admins can systematically detect problems in advance, via common integrity checks of the data, then optionally schedule the re-cache by tag or symbol name. Users and admins have the ability to manually override problems if they exist, with a specific feed, in a way that is centralized, auditable, and backed-up efficiently.

With a focus on business processes, Trump's long run goals enable data feeds to be:

- **Prioritized**, *flexibly* - a symbol can be associated with multiple data source for a variety of reasons including redundancy, calculations, or optionality.
- **Modified**, *reliably* - a symbol's data feeds can be changed out, without any changes requiring testing to the downstream application or user.
- **Verified**, *systematically* - a variety of common data processing checks are performed as the symbol's data is cached.
- **Audited**, *quickly* - alerts and reports all become possible to assess integrity or inspect where manual over-rides have been performed.
- **Aggregated**, *intelligently* - on a symbol by symbol basis, feeds can be combined and used in an extensible number of ways.
- **Customized**, *dynamically* - extensibility is possible at the templating, munging, aggregation, and validity steps.

CHAPTER 2

Getting Started

2.1 Installation

2.1.1 Step 1. Install Package

SUMMARY OF STEP 1: Clone and install trump, from github.

```
git clone https://github.com/Equitable/trump.git + cd trump + python setup.py
install
```

Note: If you use any other installation method (Eg. `python setup.py develop`), you will need to manually create your own .cfg files, in step 2, by renaming the .cfg_sample files to cfg files.

Note: Trump is setup to work with `pip install trump`, however the codebase and features are being worked on very quickly right now (2015Q2). The version on pypi, will be very stale, very quickly. It's best to install from the latest commit to the master branch direct from GitHub.

2.1.2 Step 2. Configure Settings

SUMMARY OF STEP 2: Put a SQLAlchemy Engine String in trump/config/trump.cfg. Comment out all other engines.

Trump needs information about a database it can use, plus there are a couple other settings you may want to tweak. You can either follow the instructions below, or pass a SQLAlchemy engine/engine-string, to both SetupTrump() and SymbolManager() everytime you use them.

The configuration file for trump is in:

userbase/PythonXY/site-packages/trump/config/trump.cfg

or

yourprojfolder/trump/config/trump.cfg

Note: A sample config file is included, by the name trump.cfg_sample. Depending on your installation method, you may need to copy and rename it to trump.cfg. cfg files aren't tracked by git, nor does the installation do anything other than copy and rename the file extension. `pip` and `python setup.py install` will rename them for you. `python setup.py develop` won't rename them for you, you'll have to do it yourself.

Assuming you want to use a file based sqlite database (easiest, for beginners), change:

`engine: sqlite://` to `;engine: sqlite://` (notice the semi-colon, this just comments out the line)

and change this line:

`;engine: sqlite:////home/jnmclarty/Desktop/trump.db` to

`engine: sqlite:////home/path/to/some/file/mytrumpfile.db` (on linux) or

`engine: sqlite:///C:\path\to\some\mytrumpfile.db` (on windows)

The folder needs to exist in advance, the file should not exist. Trump creates the file.

2.1.3 Step 3. Adjust Existing Template Settings (Optional)

SUMMARY OF STEP 3: Adjust any settings for templates you intend you use.

Trump has template settings, stored in multiple settings files, using an identical method as the config file in Step 2. `pip` or `python setup.py install` would have created some from samples. Using any other installation methode, you would have to rename cfg_sample to cfg yourself.

The files are here:

userbase/PythonXY/site-packages/trump/templates/settings/

or

yourprojfolder/trump/templates/settings/

Edit trump/templating/settings cfg files, depending on the intended data sources to be used.

See the documentation section "Configuring Data Sources" for guidance.

2.1.4 Step 4. Run SetupTrump()

SUMMARY OF STEP 4: Run trump.SetupTrump(), to setup the tables required for Trump to work.

Running the code block below, will create all the tables required in the database provided in Step 2.

```
from trump import SetupTrump
SetupTrump()
# Or, if you skipped step 2 correctly, you could do:
SetupTrump(r'sqlite:////home/path/to/some/file/mytrumpfile.db')
```

If it all worked, you will see "Trump is installed @...". You're Done! Hard part is over.

You're now ready to create a SymbolManager, which will help you create your first symbol.

```
from trump import SymbolManager
sm = SymbolManager()
# Or, if you skipped step 2 correctly, you could do:
sm = SymbolManager(r'sqlite:////home/path/to/some/file/mytrumpfile.db')
...
mysymbol = sm.create('MyFirstSymbol') # should run without error.
```

2.1.5 Configuring Data Sources

Data feed source template classes map to their respective .cfg file in the templating/settings directory, as discussed in Step 3.

The goal of the files is to add a small layer of security. The goal of the template classes is to reduce code during symbol creation scripts. There is nothing preventing a password from being hardcoded into a template, the same way a tablename can be added to a .cfg file. It's only a maintenance decision for the admin.

The sections of the cfg files get used by the template's in their respective classes. The section of the config files' names are then either referenced at the symbol creation point, storing .cfg file info with the symbol in the database, or leaving Trump to query the attributes at every cache, from the source .cfg file.

Trump will use parameters for a source in the following order:

1. Specified explicitly when a template is used. (Eg. table name)

```
#Assuming the template doesn't clober the value.
myfeed = QuandlFT(authtoken='XXXXXXXX')
```

2. Specified implicitly using default value or logic derived in the template. (Eg. Database Names)

```
class QuandlFT(object):
   def __init__(authtoken ='XXXXXXXXX'):
      if len(authtoken) == 8:
         self.authtoken = authtoken
      else:
         self.authtoken = 'YYYYYYYYY'
```

3. Specified implicitly using read_settings(). (Eg. database host, port)

```
class QuandlFT(object):
   def __init__(**kwargs):
         autht = read_settings('Quandl', 'userone', 'authtoken')
      self.authtoken = autht
```

4. Specified via cfg section. (Eg. authentication keys and passwords)

```
class QuandlFT(object):
   def __init__(**kwargs):
      self.meta['stype'] = 'Quandl' #cfg file name
      self.meta['sourcing_key'] = 'userone' #cfg file section
```

contents of templating/settings/Quandl.cfg:

```
[userone]
authtoken = XXXXXXXXX
```

If the template points to a section of a config file, rather than reading in a value from a config file, (ie, #4), the info will not be stored in the database. Instead, the information will be looked up during caching from the appropriate section in the cfg file.

This means that the cfg file values can be changed post symbol creation, outside of Trump.

2.1.6 Testing the Installation

After Trump has been configured, and pointed at a database via an engine string using a config file, one can run the py.test enabled test suite. The tests require network access, but will skip certain tests without it. The testing suite makes a mess, and doesn't clean up after itself. So, be prepared to run it on a database which can be delete immediately after.

Insight into compatibility with databases other SQLite and PostGres, are of interest to the maintainers. So, if you run the test suite on some other database, and it all works, do let us know via a GitHub issue or e-mail. If it doesn't, please let us know that as well!

2.2 Uninstall

#. Delete all tables Trump created. (There is a script, which attempts to do that for you. See uninstall.py. This will (attempt to) remove all tables created by Trump. The file will likely require minor changes if you use anything other than PostgreSQL, or if it hasn't been updated to reflect newer tables in Trump.) #. Delete site-packages/trump and all it's subdirectories.

2.3 Basic Usage

These examples dramatically understate the utility of Trump's long term feature set.

2.3.1 Tesla Closing Price from Multiple Sources

Adding the Symbol

```
from trump.orm import SymbolManager
from trump.templating import QuandlFT, GoogleFinanceFT, YahooFinanceFT,
                             DateExistsVT, FeedMatchVT

sm = SymbolManager()

TSLA = sm.create(name = "TSLA",
                 description = "Tesla Closing Price USD",
                 units = '$ / share')

TSLA.add_tags(["stocks","US"])

#Try Google First
#If Google's feed has a problem, try Quandl's backup
#If all else fails, use Yahoo's data...

# 'Close' is stored in the GoogleFinanceFT Template
TSLA.add_feed(GoogleFinanceFT("TSLA"))

TSLA.add_feed(QuandlFT("GOOG/NASDAQ_TSLA", fieldname='Close'))

# 'Close' is stored in the YahooFinanceFT Template
TSLA.add_feed(YahooFinanceFT("TSLA"))


#All three are downloaded, with every cache instruction
TSLA.cache()

# In the end, the result is one clean pandas Series representing
# TSLA's closing price, with source, munging, and validity parameters
# all stored persistently for future
# re-caching.

print TSLA.df.tail()

               TSLA
dateindex
2015-03-20  198.08
```

```
2015-03-23  199.63
2015-03-24  201.72
2015-03-25  194.30
2015-03-26  190.40

sm.finish()
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

TSLA = sm.get("TSLA")

#optional
TSLA.cache()

print TSLA.df.tail()

              TSLA
dateindex
2015-03-20  198.08
2015-03-23  199.63
2015-03-24  201.72
2015-03-25  194.30
2015-03-26  190.40

sm.finish()
```

2.3.2 Data From CSV, with a frequency-specified index

Adding the Symbol

```
from trump.orm import SymbolManager

#Import the CSV Feed Template
from trump.templating import CSVFT

#Import the Forward-Fill Index Template
from trump.templating import FFillIT

sm = SymbolManager()

sym = sm.create(name = "DailyDataTurnedWeekly")

f1 = CSVFT('somedata.csv', 'ColumnName', parse_dates=0, index_col=0)

sym.add_feed(f1)

weeklyind = FFillIT('W')
sym.set_indexing(weekly)

sym.cache()
```

```
sm.finish()
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

sym = sm.get("DailyDataTurnedWeekly")

#optional
oil.cache()

print sym.df.index
# <class 'pandas.tseries.index.DatetimeIndex'>
# [2010-01-03, ..., 2010-01-17]
# Length: 3, Freq: W-SUN, Timezone: None

sm.finish()
```

2.3.3 Tesla Closing Price from Two Sources, With Validity Checks

Adding the Symbol

```
from trump.orm import SymbolManager
from trump.templating import QuandlFT, GoogleFinanceFT,
                             DateExistsVT, FeedsMatchVT

sm = SymbolManager()

TSLA = sm.create(name = "TSLA",
                 description = "Tesla Closing Price USD",
                 units = '$ / share')

TSLA.add_feed(GoogleFinanceFT("TSLA"))
TSLA.add_feed(QuandlFT("GOOG/NASDAQ_TSLA", fieldname='Close'))

# Tell trump, to check the first and second feed,
# because they should be equal.

validity_settings = FeedsMatchVT(1, 2)
TSLA.add_validity(validity_settings)

# Tell trump, to make sure we have a data point for the current day
# any time we check validity.

validity_settings = DateExistsVT('today')
TSLA.add_validity(validity_settings)

# By default, the cache process checks the validity settings
# or will raise/log/warn/print/etc. based on the appropriate
# handler for validity.

# Since we're going to check validity, with a bit more
```

```
# granularity upstream/later, we can skip it during the cache process
# by setting it to False.

TSLA.cache(checkvalidty=False)

sm.finish()
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

TSLA = sm.get("TSLA")

#optional
TSLA.cache()

#There are a few options, to check the data...

#Individual validity checks can be ran, with the
# settings stored persistently in the object

# Eg 1
if TSLA.check_validity('FeedsMatch'):
   #do stuff with clean data

# Eg 2
if TSLA.check_validity('DateExists'):
   #do stuff with today's data point

# Or, all the validity checks with their
# respective settings can be ran with one simple
# property:

if TSLA.isvalid:
   #do stuff with knowing both feeds match, and
       # a datapoint for today exists.
```

2.3.4 Oil from Quandl & SQL Example

Adding the Symbol

```
from trump.orm import SymbolManager
from trump.templating import QuandlFT, SQLFT

sm = SymbolManager()

oil = sm.create(name = "oil_front_month",
                description = "Crude Oil",
                units = '$ / barrel')

oil.add_tags(['commodity','oil','futures'])

f1 = QuandlFT(r"CHRIS/CME_CL2",fieldname='Settle')
```

```
f2 = SQLFT("SELECT date,data FROM test_oil_data;")

oil.add_feed(f1)
oil.add_feed(f2)

oil.cache()

print oil.df.tail()

sm.finish()
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

oil = sm.get("oil_front_month")

#optional
oil.cache()

print oil.df.tail()

sm.finish()
```

2.3.5 Google Stock Price Daily Percent Change Munging

Adding the Symbol

```
from trump.orm import SymbolManager
from trump.templating import YahooFinaceFT

sm = SymbolManager()

GOOGpct = sm.create(name = "GOOGpct",
                    description = "Google Percent Change")

fdtemp = YahooFinanceFT("GOOG")

mgtemp = PctChangeMT()

GOOGpct.add_feed(fdtemp, munging=mgtemp)
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

GOOG = sm.get("GOOGpct")

#optional
```

```
GOOG.cache()

print GOOG.df.tail()

#              GOOGpct
# 2015-05-04  0.005354
# 2015-05-05 -0.018455
# 2015-05-06 -0.012396
# 2015-05-07  0.012361
# 2015-05-08  0.014170
```

CHAPTER 3

Object Model

3.1 Object-Relational Model

Trump's persistent object model, made possible by it's object-relational model (ORM), all starts with a `Symbol`, and an associated list of `Feed` objects.

An fragmented illustration of the ORM is presented in the three figures below.

Supporting objects store details persistently about error handling, sourcing, munging, and validation, so that a `Symbol` can `cache()` the data provided from the various `Feed` objects, in a single datatable or serve up a fresh pandas.Series at anytime. A symbol's it's `Index`, can further enhance the intelligence that Trump can serve via pandas.

Note: Trump's template system consists of objects, which are external to the ORM. Templates are used to expedite construction of ORM objects. Nothing about any template, persists in the database. Only instatiated ORM objects would do so. Templates, should be thought of as boilerplate, or macros, to reduce Feed creation time.

3.1.1 Symbol Manager

3.1.2 Conversion Manager

3.1.3 Symbols

Indices

A `Symbol` object's `Index` stores the information required for Trump to cache and serve data with different types of pandas indices.

Warning: A Trump `Index` does not contain a list of hashable values, like a pandas index. It should not be confused with the datatable's index, however it is used in the creation of the datatable's index. A more appropriate name for the class might be IndexCreationKwargs.

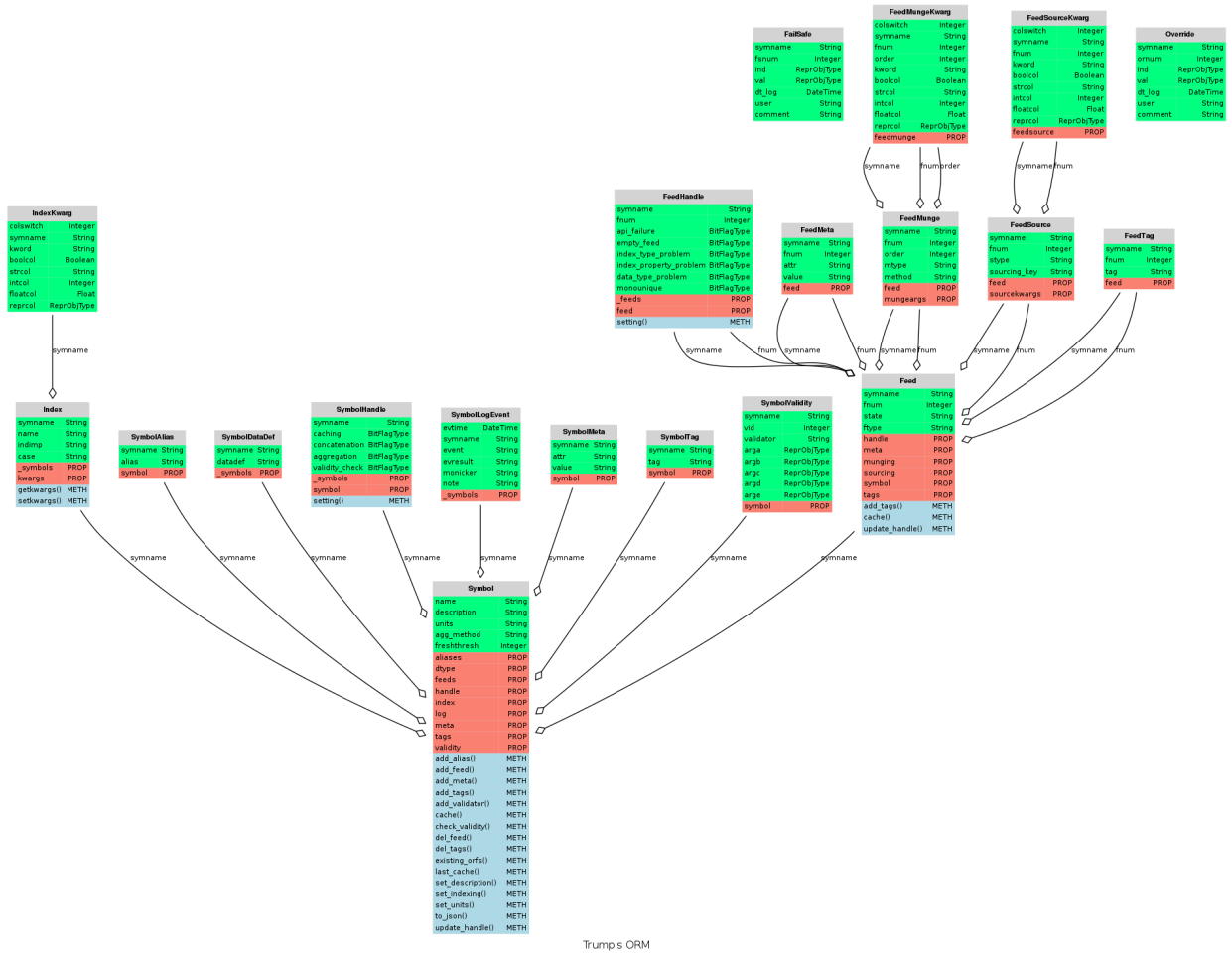

Fig. 3.1: The full ORM, excludes the symbol's datatable.

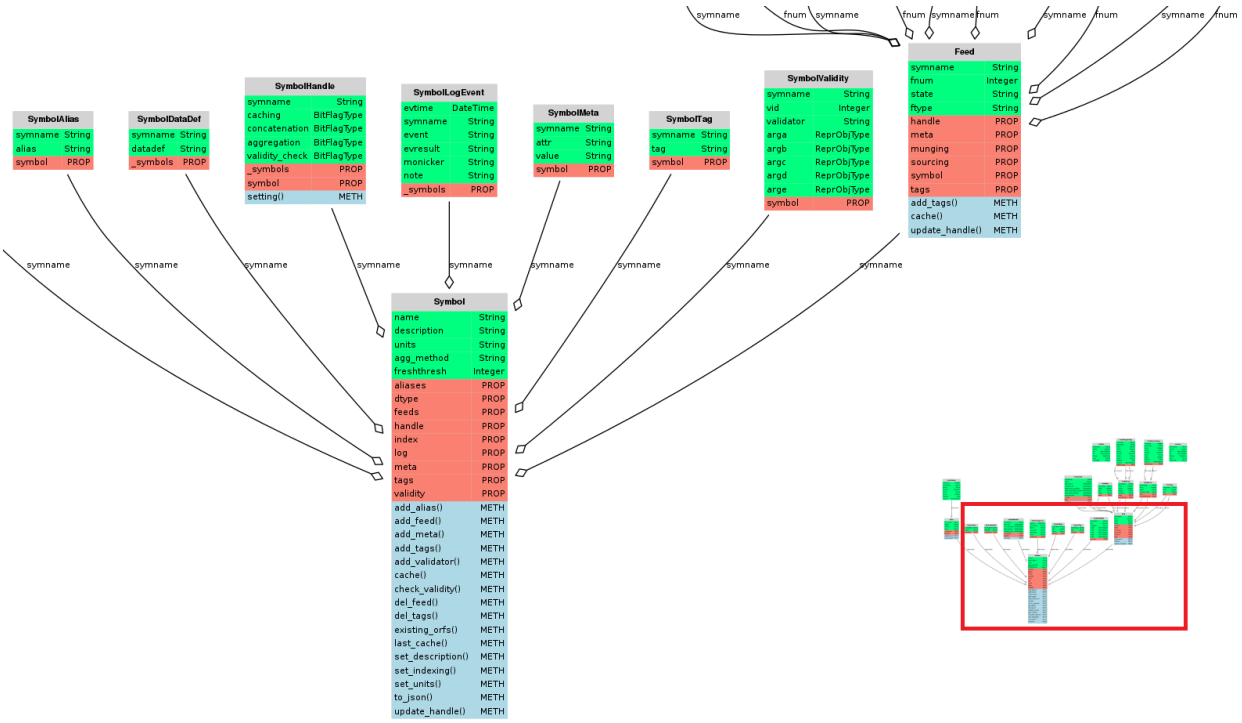

Fig. 3.2: The Symbol portion of the ORM, excludes the symbol's datatable.

Index Types

Validity Checking

3.1.4 Feeds

Feed Munging

3.1.5 Centralized Data Editing

Each trump datatable comes with two extra columns beyond the feeds, index and final.

The two columns are populated by any existing `Override` and `FailSafe` objects which survive caching, and modification to feeds.

Any `Override` will get applied blindly regardless of feeds, while the `FailSafe` objects are used only when data isn't availabe for a specific point. Once a datapoint becomes available for a specific index in the datatable, the failsafe is ignored.

3.1.6 Reporting

During the cache process, information comes back from validity checks, and any exceptions. This area of Trump's code base is currently WIP, however the basic idea is that the caching of a Feed, returns a `FeedReport`. For each cached Feed, there would be one report, all of which would get aggregated up into, and combined with the symbol-level information, in a `SymbolReport`. When the SymbolManager caches one or more symbols, it aggregates SymbolReports into one big and final `TrumpReport`.

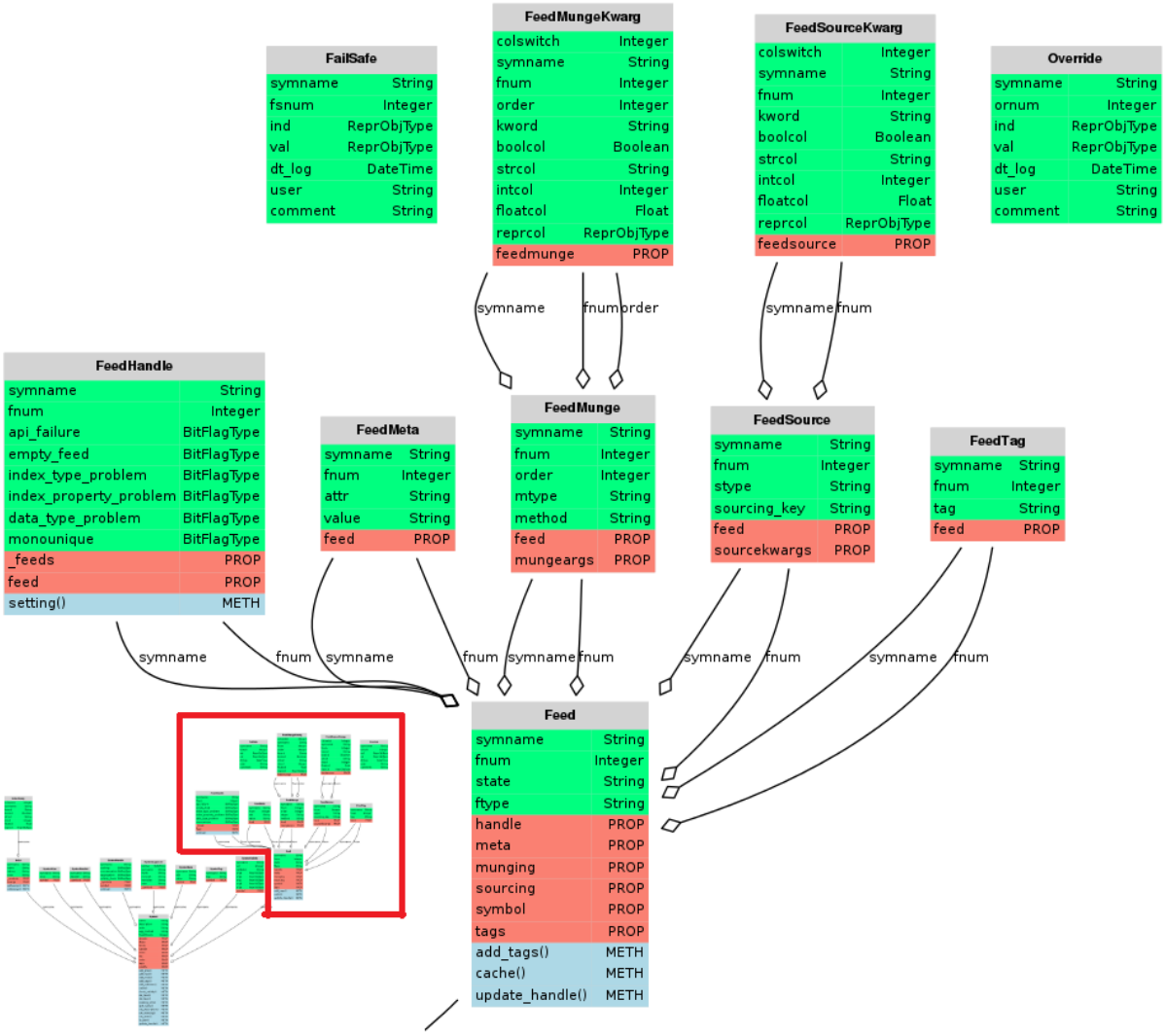

Fig. 3.3: The Feed, FailSafe & Override portion of the ORM

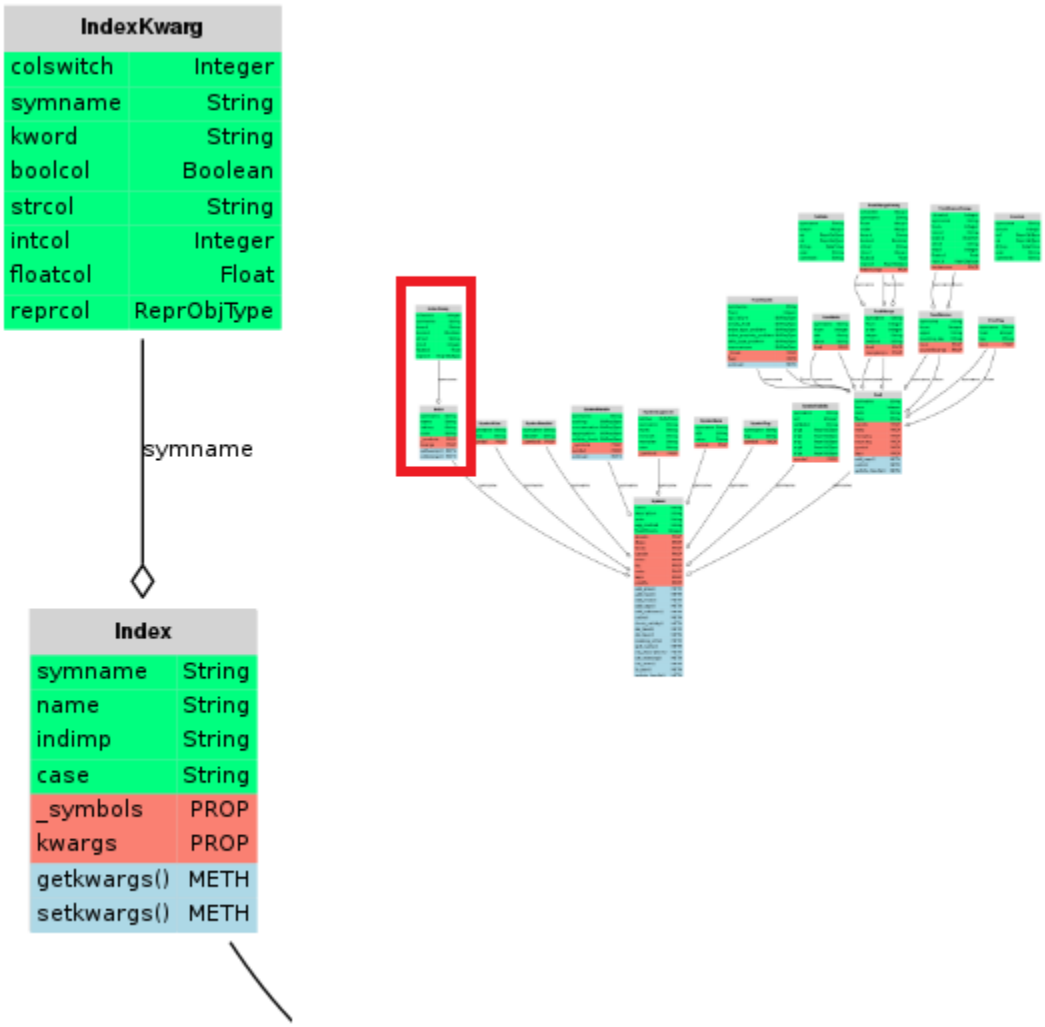

Fig. 3.4: The Index portion of the ORM.

Each of the three levels of reports, have the appropriate aggregated results, plus collections of their own HandlePointReport and ReportPoint objects.

3.1.7 Error Handling

The Symbol & Feed objects have a single SymbolHandle and FeedHandle object accessed via their .handle attribute. They both work identically. The only difference is the column names that each have. Each column, aside from symname, represents a checkpoint during caching, which could cause errors external to trump. The integer stored in each column is a serialized BitFlag object, which uses bit-wise logic to save the settings associated with what to do upon an exception. What to do, mainly means deciding between various printing, logging, warning or raising options.

The Symbol's possible exception-inducing handle-points include:

- caching (of feeds)
- concatenation (of feeds)
- aggregation (of final value column)
- validity_check

The Feed's possible exception-inducing handle-points include:

- api_failure
- feed_type
- index_type_problem
- index_property_problem
- data_type_problem
- monounique

For example, if a feed source is prone to problems, set the api_failure to print the trace by setting the BitFlag object's 'stdout' flag to True, and the other flags to False. If there's a problem, Trump will attempt to continue, and hope that there is another feed with good data available. However, if a source should be reliably available, you may want to set the BitFlag object's 'raise' flag to True.

BitFlags

Trump stores instructions regarding how to handle exceptions in specific points of the cache process using a serializable object representing a list of boolean values calleda BitFlag. There are two objects which make the BitFlag implementation work. There is the BitFlag object, which converts dictionaries and integers to bitwise logic, and then there is the BitFlagType which give SQLAlchemy the ability to create columns, and handle them appropriately, containing BitFlag objects.

The likely values assigned, will commonly be from the list below. Use Bitwise logic operators to make other combinations.

Desired Effect	BitFlag Instantiation	Description
Raise-Only	BitFlag(1)	Raise an Exception
Warn-Only	BitFlag(2)	Raise a Warning
Email-Only *	BitFlag(4)	Send an E-mail
DBLog-Only *	BitFlag(8)	Log to the Database
TxtLog-Only	BitFlag(16)	Text Log
StdOut-Only	BitFlag(32)	Standard Output Stream
Report-Only	BitFlag(64)	Report
TxtLog and StdOut	BitFlag(48)	Print & Log

- Denotes Features not implemented, yet.

The implementation is awkard, all in the name of speed. There are (4 + 7 x # of Feeds) BitFlags, per symbol. So they are serialized into integers, rather than having (4 + 7 x # of Feeds) x 7 boolean database columns.

3.2 Data Flow

Trump centralizes the flow of information using two concepts:

1. Objectification - the process of persistently storing information about data.
2. Caching - the process of fetching data, saving it systematically, and serving it intelligently.

3.2.1 Objectification

The objectification happens via an addition-like process entailing the instantiation of one or more symbols. The objectification enables downstream applications to work with symbol names in order to force the caching, and be served reliable data.

There are two approaches to perform the objectification instantiation of Symbols

1. First Principles (from ORM)
2. Template Based (from Special Python Classes + ORM)

First Principles

The first principles approach to using Trump is basically direct access to the SQLAlchemy-based object-relational model. It's time consuming to develop with, but necessary to understand in order to design new intelligent templates.

Using Trump's ORM, the process is something akin to:

For Every Symbol:

1. Instantiate a new `Symbol`
2. Optionally, add some `SymbolTag`
3. Optionally, adjust the symbol's `Index` case and type
4. Optionally, adjust the symbol's `SymbolHandle` handlepoints
5. Instantiate one ore more `Feed` objects
6. For each Feed, update `FeedMeta`, `FeedSource` details
7. Optionally, adjust each feed's `FeedMunge` instructions
8. Optionally, adjust each feed's `FeedHandle` handlepoints
9. Optionally, adjust each Symbol's `SymbolValidity` instructions

Template Based

By setting up, and using Trump template classes, the two steps below replace steps 1 to 8 of the first principles approach.

For Every Kind of Symbol:

1. Create custom templates for common sources of proprietary data.

For Every Symbol:

2. Instantiate a new `Symbol` using a template containing Tag, Feed, Source, Handle, Validity settings.
3. Tweak any details uncovered by the chosen templates for the symbol, or any of it's feeds.

In practice, it's inevitable that templates will be used where possible, and do the heavy lifting of instantiation, but tweaks to each symbol would be made post-instantiation.

3.2.2 Caching

The cache process, is more than just caching, but that's the main purpose. The cache process, essentially builds a fresh datatable. In order to cache a symbol, Trump performs the following steps:

For each Feed...

1. Fetches a fresh copy of each Feed, based on the `FeedSource` parameters.
2. Munges each Feed, based on the `FeedMunge` parameters.
3. Converts the datatype using a `SymbolDataDef`

Then...

4. Concatenates the data from each feed, into a dataframe.
5. Converts the index datatype using the Symbol's `Index` parameters.
6. Two columns are appended to the dataframe, one for overrides, one for failsafes. Any which exist, are fetched.
7. An aggregation method is used to build a final series out of the data from the feeds and any overrides/failsafes.
8. The dataframe is stored in the database, in it's own table, called a datatable.
9. Optionally, any validity checks, which are set up in `SymbolValidity`, are performed.

When executed, data from each Feed is queried, and munged according to predefined instructions, on a per-feed basis. The feeds are joined together, each forming columns of a pandas Dataframe. A `IndexImplementor` corrects the index. An aggregation method converts the Dataframe into a single, final, Series. Depending on the aggregation method, any single values are overrode, and blanks get populated, based on any previously defined `Override` and `FailSafe` objects associated with the symbol being cached.

The Datatable & Aggregation

Steps #6, #7 & #8 above are easiest to understand, with a graphical look at the final product: a cached Symbol's datatable.

An example of a datatable, is in the figure below. This, is a simple table, common to anybody with SQL knowledge.

The example datatable, seen above, is one symbol with two feeds, both of which had problems. One of the feeds stopped completely on the 11th, the other had a missing datapoint. Plus, a previous problem, looks like it was manually overrode on the 6th, but then later, the feed started working again. The overrides and failsafes were applied appropriately on the 6th, and the 12th, while the failsafe on the 10th, was ignored after the feed #2 started working again.

It's easy to imagine the simple Dataframe after step #5 of the cache process. It would have a single index, then a column for every Feed. #6, appends the two columns mentioned, along with any individual datapoints. Then an aggregation method creates the 'final' column. Details about the specific aggregation method are defined at, or updated after,

	indx	final	override_feed000	feed001	feed002	failsafe_feed999
1	2015-03-02 ...	43.88	{null}	43.88	43.88	{null}
2	2015-03-03 ...	43.28	{null}	43.28	43.28	{null}
3	2015-03-04 ...	43.06	{null}	43.06	43.06	{null}
4	2015-03-05 ...	43.11	{null}	43.11	43.11	{null}
5	2015-03-06 ...	42.37	42.37	42.36	42.36	{null}
6	2015-03-09 ...	42.85	{null}	42.85	42.85	{null}
7	2015-03-10 ...	42.03	{null}	42.03	42.03	{null}
8	2015-03-11 ...	41.98	{null}	{null}	41.98	{null}
9	2015-03-12 ...	41.02	{null}	{null}	{null}	41.02
10	2015-03-13 ...	41.38	{null}	{null}	41.38	41.44

Fig. 3.5: Example of a symbol's datatable, with two feeds of data, both with problems.

Symbol instantiation. Up to and including the aggregation, all operations are simply changing the dataframe of feeds, overrides, and failsafes.

After the final is calculated, the dataframe is stored until the next cache, as a table - the datatable, illustrated in the figure above. It can then be quickly checked for validity and served to applications.

3.3 Aggregation Methods

Trump currently has two types of aggregation methods:

1. Apply-Row
2. Choose-Column

As the names infer, the apply-row methods have one thing in common, they build the final data values by looking at each row of the datatable, one at a time. The choose-column methods, compare the data available in each column, then return an entire series. Row-apply methods all take a pandas Series, and return a value. Column-choose methods all take a pandas Dataframe, and return a series.

Row-apply functions are invoked using the pseudo code below:

```
df['final'] = df.apply(row_apply_method, axis=1)
```

Column-choose functions are invoked using the pseudo code below:

```
df['final'] = column_choose_method(df)
```

Both methods have access to the data in the override, and failsafe, columns so it's technically possible to create a method which overloads the behaviour of these columns. It is the responsibility of each method to implement the override, and failsafe, logic.

3.3.1 Apply-Row Methods

Each of these methods, can be thought of as a for-loop that looks at each row of the datatable, then decides on the correct value for the final column, on a row by row basis.

The datatable, as a Dataframe, gets these methods applied. The columns are sorted prior to being passed. So, the value at index 0, is always the override datapoint, if it exists, and the value at index -1, is always the failsafe datapoint, if it exists. Everything else, that is, the feeds, are in columns 1 through n, where n is the number of feeds.

Note: The aggregation methods are organized in the code using private mixin classes. The FeedAggregator object handles the implementation of every static method, based solely on it's name. This means that any new methods added, must be unique to either mixin.

3.3.2 Choose-Column Methods

Each of these methods, can be thought of as a for-loop that looks at each column of the datatable, then chooses the appropriate feed to use, as final. They all still apply overrides and failsafes on a row-by-row basis.

The datatable, as a Dataframe, is passed to these methods in a single call.

Note: See the note in the previous section about custom method naming.

CHAPTER 4

Templating

4.1 Template Base Classes

4.2 Template Classes

4.2.1 Tag Templates

4.2.2 Munging Templates

4.2.3 Source Templates

4.2.4 Feed Templates

4.2.5 Index Templates

4.2.6 Validity Templates

CHAPTER 5

Source Extensions

5.1 Creating & Modifying Source Extensions

This section of the docs is really only intended for those who want to write, or modify, their own source extensions. But, it can be helpful to understand how they work, even for those who don't want to write an extension.

Trump's framework enables sources of varying, dynamic, and proprietary types. A source extension is basically a generalized way of getting a pandas Series out of an existing external API. For instance examples include, the pandas datareader, a standardized DBAPI 2.0 accessible schema, a proprietary library, or something as simple as a CSV file. At a high level, each symbol's feed's source's kwargs are passed to the appropriate source extension, based on the defined source type.

When each symbol is cached, it loops through each of it's feeds. Each feed's source is queried, using four critical python lines in orm.Feed.cache():

```
if stype in sources:
   self.data = sources[stype](self.ses, **kwargs)
else:
   raise Exception("Unknown Source Type : {}".format(stype))
```

The important line, is the second one. 'sources', is a dictionary loaded every time trump's orm.py is imported. The key's are just strings representing the "Source Type", eg. "DBAPI", "Quandl", "BBFetch" (Example of a proprietary source). The values of the sources dictionary are SourceExtension objects. The SourceExtension objects wrap modules discovered dynamically when loader.py scans the source extension folder. The code for the SourceExtension is below:

```
class SourceExtension(object):
   def __init__(self, mod):  #instantiated only once per import of trump.orm
      self.initialized = False
      self.mod = mod
      self.renew = mod.renew
      self.Source = mod.Source
   def __call__(self, _ses, **kwargs):  #called each symbol's feed's cache (in the second line above)
      if not self.initialized or self.renew:
         self.fetcher = self.Source(_ses, **kwargs)
         self.initialized = True
      return self.fetcher.getseries(_ses, **kwargs)
```

A SourceExtension is instantiated only once, when loader.py passes a module it discovered. The modules, are the "source extension", which are just simply python files, required to be created in a standard way. The standard can be illustrated with an example. Below, is an example csv-file source extension (which may be stale, compared to the actual csv extension).

See trump/extensions/source for more examples.

```
stype = 'PyDataCSV'
renew = False

class Source(object):
   def __init__(self, ses, **kwargs):
      from pandas import read_csv
      self.read_csv = read_csv

   def getseries(self, ses, **kwargs):

      col = kwargs['data_column']
      del kwargs['data_column']

      fpob = kwargs['filepath_or_buffer']
      del kwargs['filepath_or_buffer']

      df = self.read_csv(fpob, **kwargs)

      data = df[col]

      return data
```

Noticed that the two variables, stype & renew, as well as the Source class, are used in the SourceExtension instantiation.

5.1.1 Source Extension Standard Form

Any extension module needs 3 things; an stype variable, renew variable, and Source class.

stype (str)

stype is the string used in the 'sources' dictionary mentioned earlier, and must match the the stype set in the corresponding Source template(s).

renew (boolean)

renew is a boolean, which determines if the Source object is reinstantiated on every use. For instance, one might create a source, which sets up a database connection, which stays open for the life of any script using trump's orm, but only if that specific source is used at least once. Renew would be set to False, and the connection logic, would be put in Source.__init__. Alternatively, if a new connection would be required on every symbol's cache, renew would be set to True. The tradeoffs, are speed and resource constraints. Both __init__ and getseries get the same arguments. The current live trump session, and the symbol's feed's source kwargs.

Source (class)

Source is an an object with one other method, getseries, other than the constructor (__init__). Both take the same arguments: the trump session, and the Symbol's Feed's Source's kwargs. getseries, returns a dataframe.

5.2 Pre-Installed Source Extensions

5.2.1 BBFetch

```
# the directory is tx-bbfetch
stype = 'BBFetch'
renew = True
```

Required kwargs:

- 'elid'
- 'bbtype' = ['COMMON', 'BULK'], then a few relevant kwargs depending on each.

Optional kwargs:

- 'duphandler' - 'sum'
- 'croptime' - boolean

5.2.2 DBAPI

```
# the directory is tx-dbapi
stype = 'DBAPI'
renew = True
```

The DBAPI driver, will use by default the same driver SQLAlchemy is using for trump. There is currently no way to change this default. It's assumed that the driver is DBAPI 2.0 compliant.

Required kwargs include:

- 'dbinsttype' which must be one of 'COMMAND', 'KEYCOL', 'TWOKEYCOL'
- 'dsn', 'user', 'password', 'host', 'database', 'port'

Optional kwargs include:

- duphandler ['sum'] which just groups duplicate index values together via the sum.

Additional kwargs:

Required based on 'dbinsttype' chosen:

'COMMAND' : - 'command' which is just a SQL string, where the first column becomes the index, and the second column becomes the data.

'KEYCOL' : - ['indexcol', 'datacol', 'table', 'keycol', 'key']

'TWOKEYCOL' : - ['indexcol', 'datacol', 'table', 'keyacol', 'keya', 'keybcol', 'keyb']

5.2.3 psycopg2

```
# the directory is tx-psycopg2
stype = 'psycopg2'
renew = True
```

Started extension for a Postgres-specifc source.

Not fully implemented.

5.2.4 PyDataCSV

```
# the directory is tx-pydatacsv
stype = 'PyDataCSV'
renew = False
```

All kwargs are passed to panda's read_csv function.

Additional required kwargs:

- 'filepath_or_buffer' - should be an absolute path. Relative will only work, if caching is only

performed by a python script which can access the relative path.

- 'data_column' - the specific column required, so to turn the dataframe into a series.

5.2.5 PyDataDataReaderST

```
# the directory is tx-pydatadatareaderst
stype = 'PyDataDataReaderST'
renew = True
```

This uses pandas.io.data.DataReader, all kwargs get passed to that.

start and end are optional, but must be of the form 'YYYY-MM-DD'.

Will default to since the beginning of available data, and run through "today".

data_column is required to be specified as well.

5.2.6 Quandl

```
# the directory is tx-quandl
stype = 'Quandl'
renew = True
```

All kwargs are passed to Quandl's API quandl.get()

An additional 'fieldname' is available to select a specific column if a specifc quandl DB, doesn't support quandl's version of the same feature.

5.2.7 SQLAlchemy

```
# the directory is tx-sqlalchemy
stype = 'SQLAlchemy'
renew = True
```

a SQLAlchemy based implementation...so an engine string could be used.

Not fully implemented

5.2.8 WorldBankST

```
# the directory is tx-worldbankst
stype = 'WorldBankST'
renew = False
```

Uses pandas.io.wb.download to query indicators, for a specific country.

country, must be a world bank country code.

Some assumptions as implied about the indicator and the first level of the index. This may not work for all worldbank indicators.

CHAPTER 6

User Interface

6.1 UI Prototype

A preliminary user interface for Trump is being prototyped.

6.1.1 Web Interface

The web UI was born out of Flask, Jinja2 and Bootstrap "hello world".

Some screen shots, of the beginning, are below.

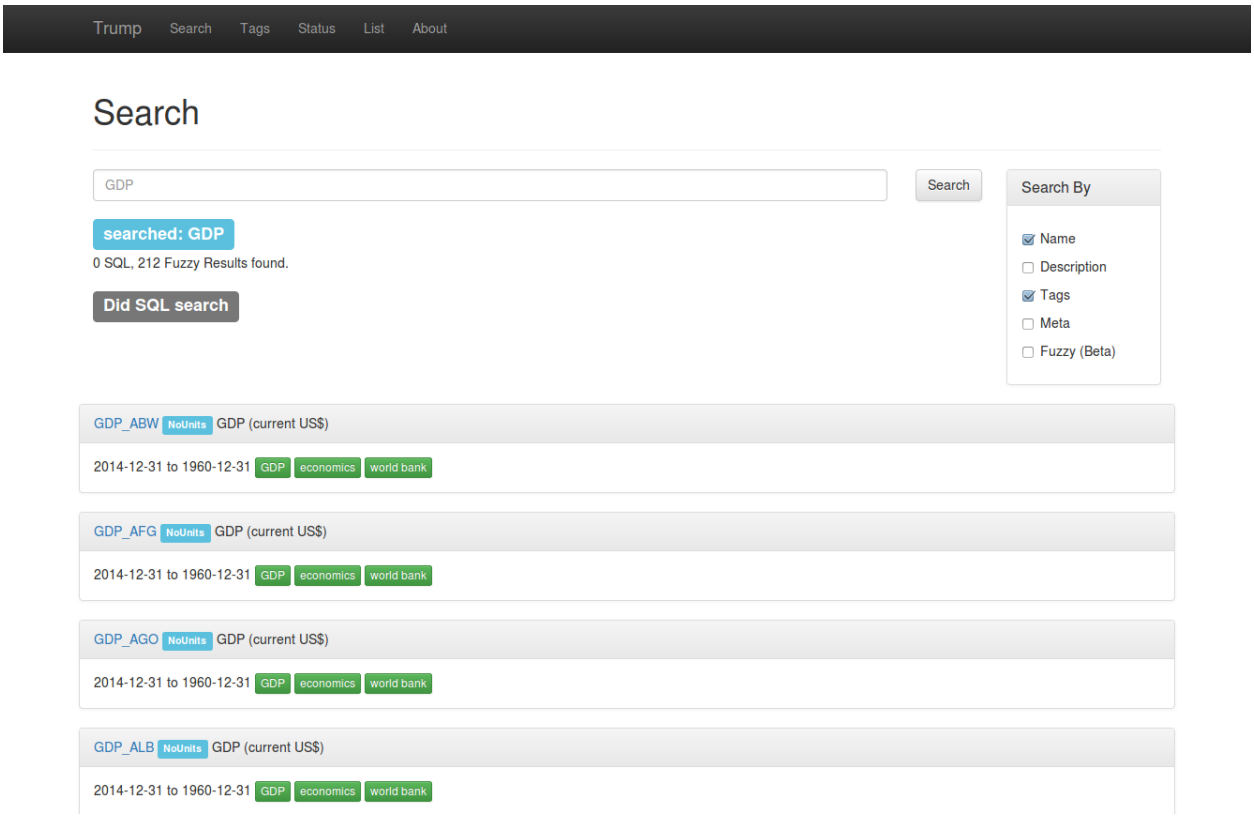

Fig. 6.1: SQL-like search, is straight forward and as expected.

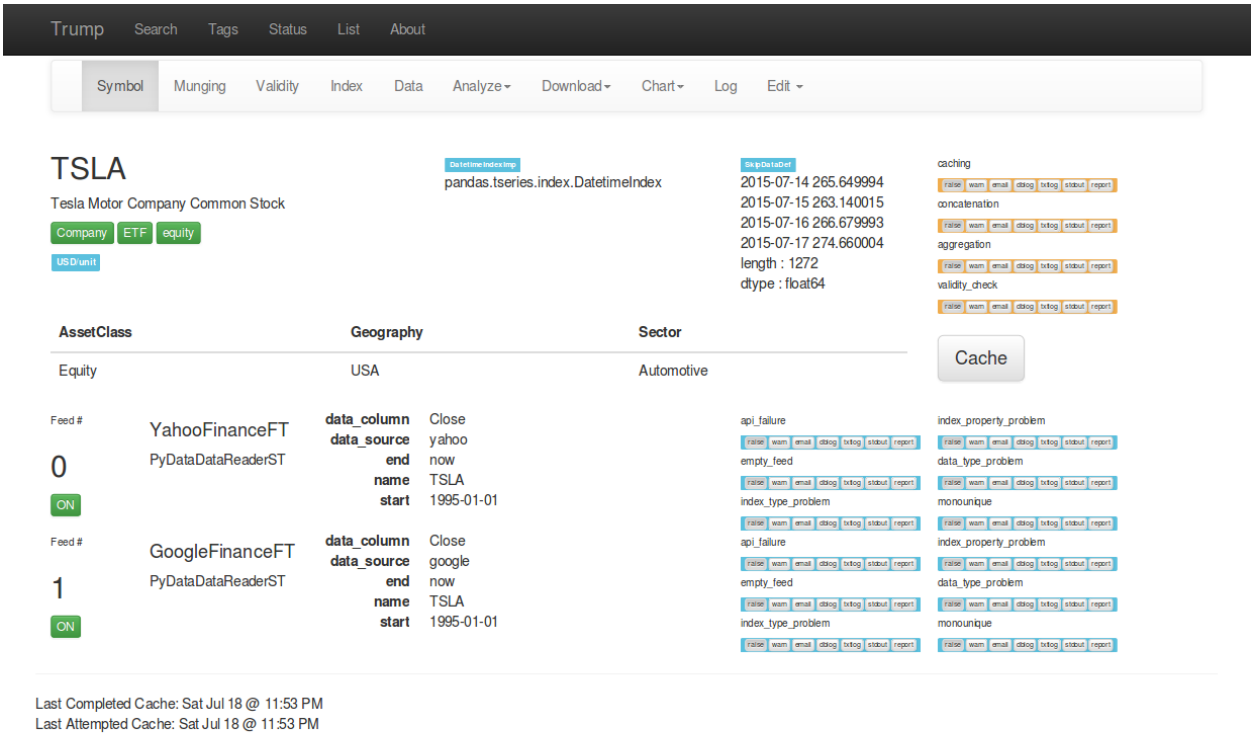

Fig. 6.2: An example of symbol page, for a symbol with two feeds.

And, much, much more, coming soon...

6.1.2 Search

Trump's SymbolManager object, has basic/expected SQL-enabled search functionality.

The Trump UI prototype is boosted by an ElasticSearch server with a single index consisting of symbol, tag, description, and meta data. To add a symbol to the index, use the json created from Symbol.to_json().

6.1.3 Background Caching

Trump's caching process isn't blazing fast, which means using the UI to kick off caching of one or more symbols, requires a background process in order for the web interface to stay responsive.

A very simple RabbitMQ consumer application, is included with the UI, which listens for the instruction to cache. The python pika package is required.

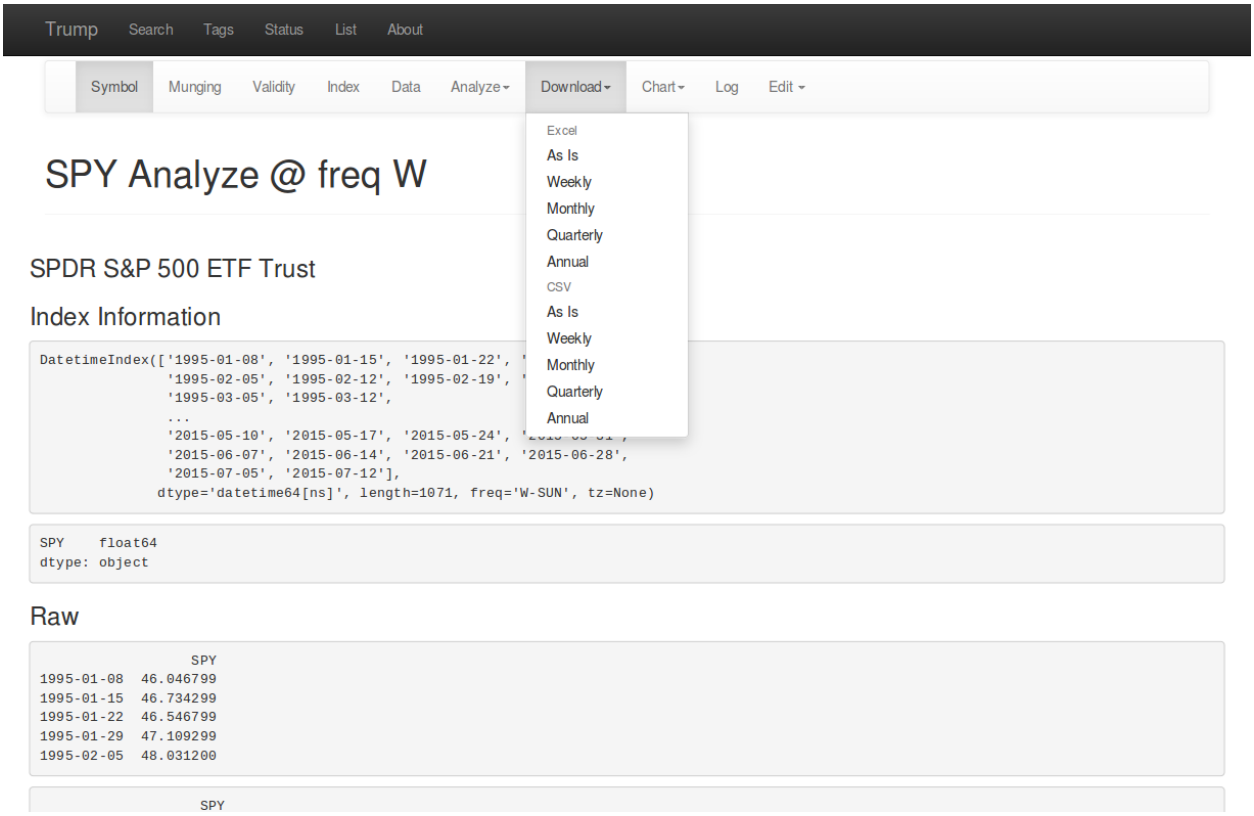

Fig. 6.3: View the index, data, and do common analysis. Or, download to excel/csv...

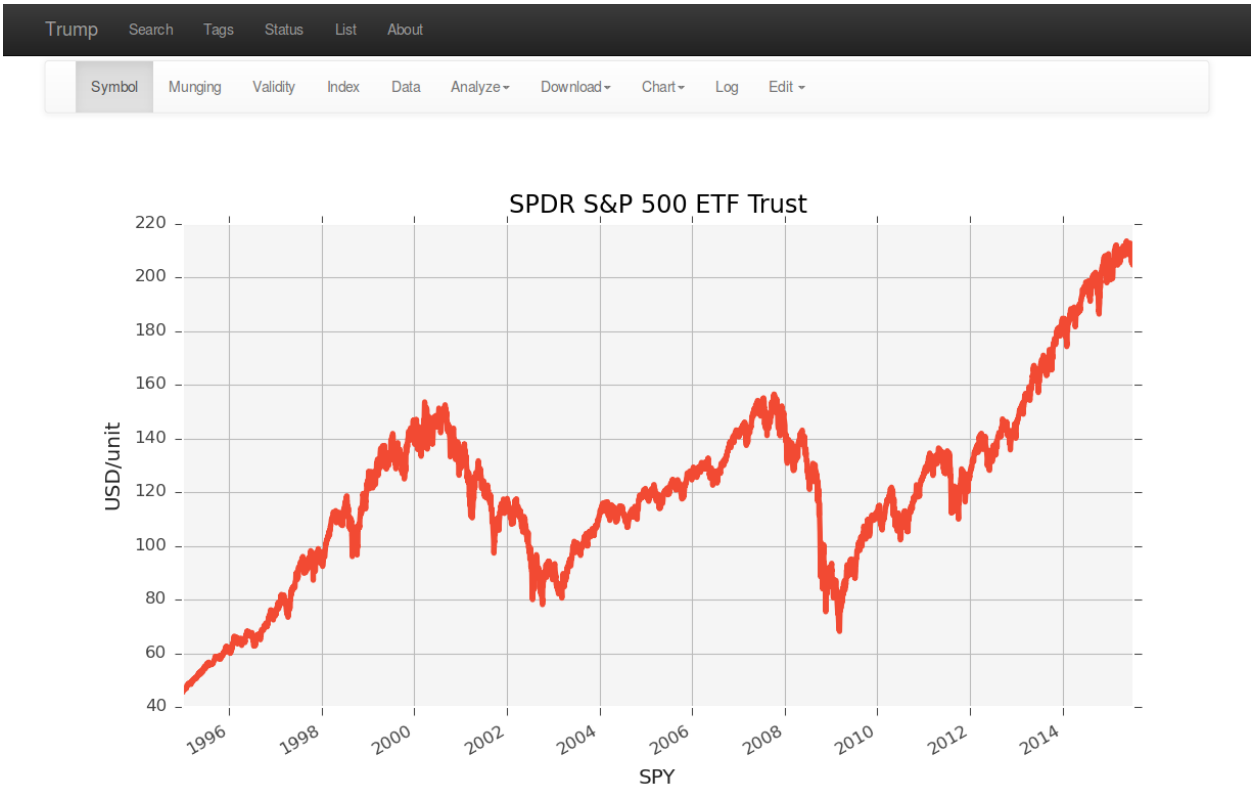

Fig. 6.4: Histograms and basic charting are available.

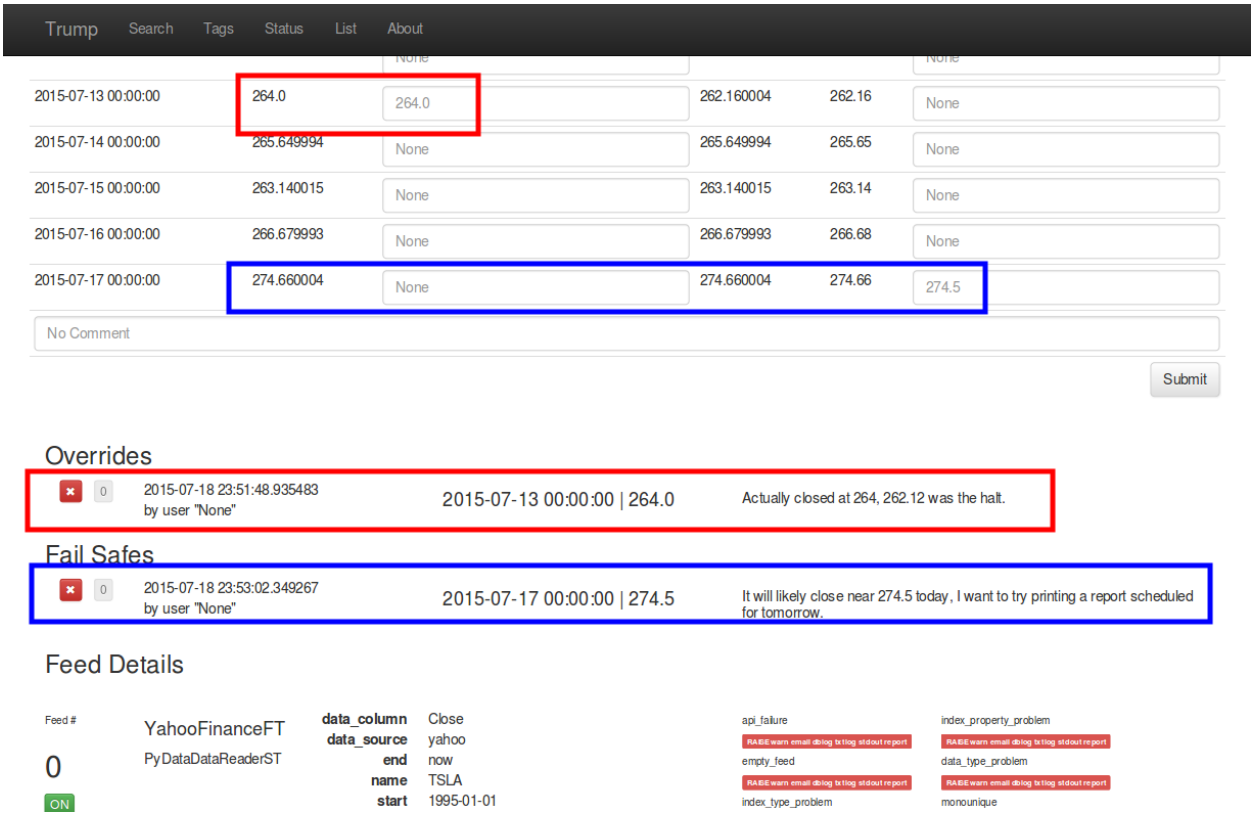

Fig. 6.5: Overrides and failsafes, are what makes Trump amazing for business processes.

Fig. 6.6: The last time a symbol was attempted, and successfully cached, are available.

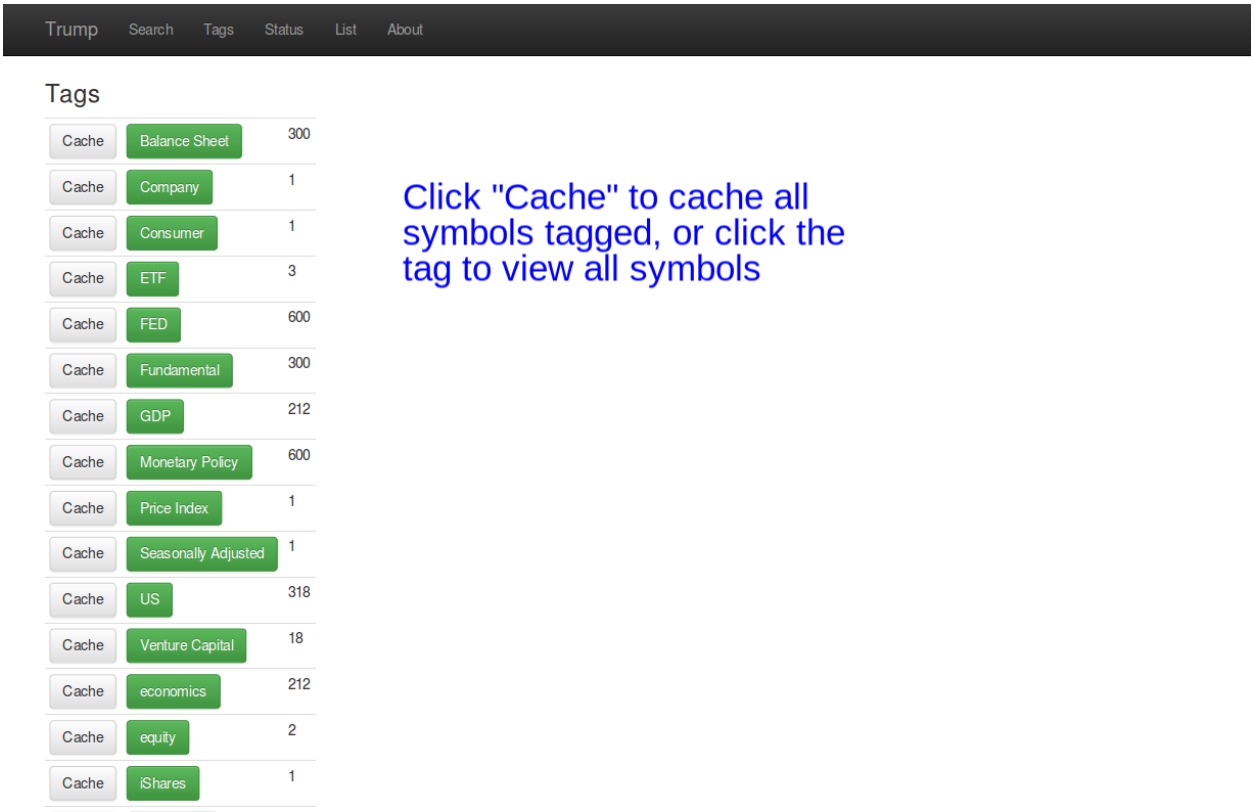

Fig. 6.7: Browse and cache sets of symbols, based on tags...

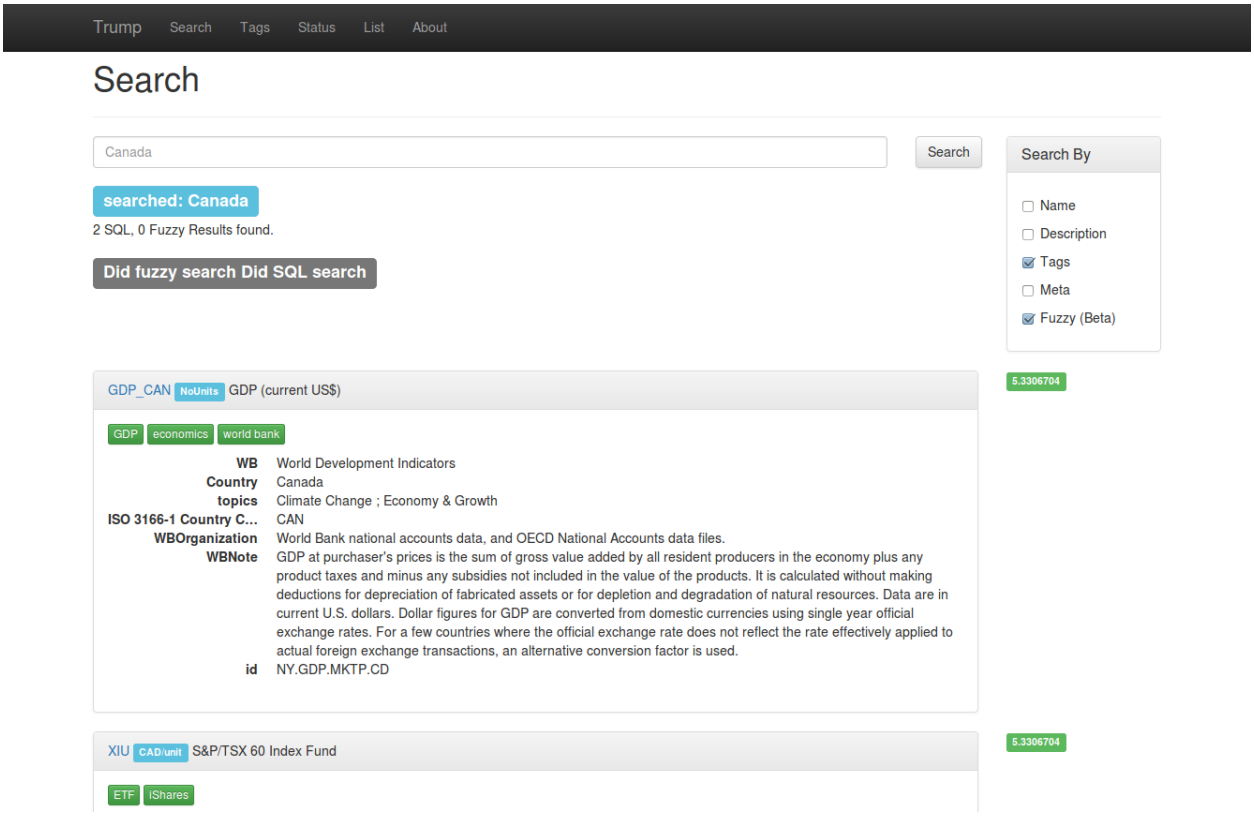

Fig. 6.8: ElasticSearch, makes searching much cooler...