exhale Documentation

Release 1.0

Stephen McDowell

Aug 05, 2017

Contents

Automatic C++ library api documentation generation: breathe doxygen in and exhale it out. A *Quickstart* guide gives the bare minimum needed to get things working, please read the *Overview* section if you are new to Sphinx or Breathe.

This project revives the Doxygen style hierarchies in reStructuredText documents so that you can keep using the beautiful Sphinx websites, but have a more human-readable Library API without having to manually write it out yourself. Exhale is self-contained and easily portable to Read the Docs. You should be able to use any Sphinx theme of your choosing, though some of them likely work better than others.

A more comprehensive example repository (which was used for testing once upon a time) is located at the companion site.

CHAPTER 1

Overview

Exhale is an automatic C++ library API generation utility. It attempts to port the Doxygen hierarchy view presentations for classes and files into the Sphinx domain. See the *Quickstart* for the bare minimum you need to give to integrate it into your project.

What does it do?

Exhale is completely dependent upon valid Doxygen documentation, and a working interface with Breathe. More specifically exhale explicitly parses the Doxygen xml using Breathe. Once Breathe is done parsing everything, the problem basically becomes a graph traversal except that parts of the graph have been lost somewhere and need to be rediscovered.

Once the graph has been reconstructed and traversed, the API reStructuredText documents are generated and linked to one another, as well as the root library document. The intent of the root library document is for you to just include it in your top-level index `toctree` directive. Refer to *Additional Usage and Customization* for how the root library document will be presented as well as how to customize it.

The individual and root library page are an attempt to emulate the output of what Doxygen would for their html class and file hierarchies. Many similarities exist, such as the inclusion of `struct` before `class` in ranking over alphabetical. However, I took a few liberties to change the output to include things I felt were more useful, such as including `enum` in the Class Hierarchy.

Note: Every generated file has a reStructuredText label that you can use to highlight specific items such as an important class or a file. Refer to *Linking to a Generated File* for more information.

What does it need to do that?

Assuming you have Sphinx, Doxygen, and Breathe working with each other, exhale needs at least the following information:

1. The location of the output of the Doxygen xml **index.xml** file.
2. The name of the **folder** you want the generated files to be put in. You can give the current directory, but it will get *messy*.
3. The name of the root file **you** will be linking to from your reStructuredText. This file will be in the directory from 2.
4. The title of the document for 3, since this will appear in a `.. toctree::` directive.
5. The relative or absolute path to **strip** from the File Pages. If you follow the tutorials present on this site, this will always be `".."`. This may be removed in the future, but currently if this is not supplied then hosting on Read the Docs will break.

Warning: Order of execution matters: Doxygen must be executed / updated before exhale. If you are calling exhale from `conf.py`, then you must specify either an absolute path, or a path **relative to conf.py** for items 1 and 2 above.

I intentionally wrote exhale in one file (exhale) so that you can just drop that into your repository — e.g. right next to `conf.py`. If you are hosting on Read the Docs, make sure that `exhale.py` is being tracked by `git`.

So if your documentation directory structure looked like:

```
docs/
|   conf.py     # created by sphinx-quickstart
|   exhale.py   # placed here by you
|   index.rst   # created by sphinx-quickstart
|   about.rst   # created by you
|   Makefile    # created by sphinx-quickstart
|   Doxyfile    # created by you
--doxyoutput/ # output destination of Doxygen
    --xml/
           index.xml
           ... other doxygen xml output ...
```

Then you could answer 1-5 above with

1	Doxygen xml index location	`"./doxyoutput/xml/index.xml"`
2	Generated library containment folder	`"./generated_api"`
3	Generated library root file	`"library_root.rst"`
4	Generated library root file title	`"Library API"`
5	Path to strip from Doxygen output	`".."`

and the following directory structure would be produced:

```
docs/
|   conf.py     # created by sphinx-quickstart
|   exhale.py   # placed here by you
|   index.rst   # created by sphinx-quickstart
|   about.rst   # created by you
|   Makefile    # created by sphinx-quickstart
|   Doxyfile    # created by you
--doxyoutput/ # output destination of Doxygen
|   --xml/
|          index.xml
|          ... other doxygen xml output ...
--generated_api/
```

```
library_root.rst
... many other files ...
```

That is, all exhale is actually doing is creating a large number of independent reStructuredText documents that link between each other. Include the file from **3** in a `toctree` somewhere, and that file will link to every possible generated file in some way or another. These are also all searchable, since Sphinx is given control of the final setup and linking.

Note: The file in **3** should not have the path from **2** preceding, exhale does that.

CHAPTER 2

Usage

Using exhale can be simple or involved, depending on how much you want to change and how familiar you are with things like Sphinx, Breathe, Doxygen, and Jinja. At the top level, what you need is:

1. Your C++ code you want to document, with "proper" Doxygen documentation. Please read the *Doxygen Documentation Specifics* for common documentation pitfalls, as well as features previously unavailable in standard Doxygen.
2. Generating the API using Sphinx, Doxygen, Breathe already working.

Quickstart

In your `conf.py`

```
# setup is called auto-magically for you by Sphinx
def setup(app):
    # create the dictionary to send to exhale
    exhaleArgs = {
        "doxygenIndexXMLPath"  : "./doxyoutput/xml/index.xml",
        "containmentFolder"    : "./generated_api",
        "rootFileName"         : "library_root.rst",
        "rootFileTitle"        : "Library API",
        "doxygenStripFromPath" : ".."
    }

    # import the exhale module from the current directory and generate the api
    sys.path.insert(0, os.path.abspath('.')) # exhale.py is in this directory
    from exhale import generate
    generate(exhaleArgs)
```

In your `index.rst`, you might have something like

Note: The above assumes that your Doxygen xml tree has already been created. The *Fully Automated Building*

section provides additional steps to do this all at once.

Lastly, you will likely want to add these two lines somewhere in `conf.py` as well:

```
# Tell sphinx what the primary language being documented is.
primary_domain = 'cpp'

# Tell sphinx what the pygments highlight language should be.
highlight_language = 'cpp'
```

The full documentation for the only (official) entry point is: *exhale.generate()*.

Additional Usage and Customization

The main library page that you will link to from your documentation is laid out as follows:

1	{{ Library API Title }}	Heading
2	{{ after title description }}	Section 1
3	Class Hierarchy	Section 2
4	File Hierarchy	Section 3
5	Full API Listing	Section 4
6	{{ after body description }}	Section 5

1. The dictionary key `rootFileTitle` passed to *exhale.generate()* function is what will be the Heading title.
2. Section 1 can optionally be provided by the dictionary key `afterTitleDescription` in the argument to *exhale.generate()*.
3. The class view hierarchy (including namespaces with class-like children).
4. The file view hierarchy (including folders).
5. An ordered enumeration of every Breathe compound found, except for groups.
6. Section 5 can optionally be provided by the dictionary key `afterBodySummary` in the argument to *exhale.generate()*.

Clickable Hierarchies

While I would love to automate this for you, it is not possible to do so very easily. If you would like to have a more interactive hierarchy view (instead of just bulleted lists), you will need to add some extra files for it to work. There are a lot of different options available, but I rather enjoy Stephen Morley's collapsibleLists: it's effective, easily customizable if you know front-end, and has a generous license.

You will need

1. The javascript library.
2. The css stylesheet and its associated images.
3. A sphinx template override.

I have taken the liberty of adding these files to the exhale repository, just clone exhale and move the files to where you need them to go. Specifically, the exhale repository looks like this:

```
exhale/
    |   README.md
    |   exhale.py # put next to conf.py
    --treeView/
        --_static/
        |       --collapse/
        |              CollapsibleLists.compressed.js # (1) library
        |              tree_view.css                 # (2) stylesheet
        |              button-closed.png             #  v  associated images
        |              button-open.png
        |              button.png
        |              list-item-contents.png
        |              list-item-last-open.png
        |              list-item-last.png
        |              list-item-open.png
        |              list-item-root.png
        |              list-item.png
        --_templates/
                layout.html                          # (3) MUST be layout.html
```

You then just need to to move the folder `collapse` to your `_static` directory, and move `layout.html` to your `_templates` directory. So your `docs` folder might look something like:

```
docs/
    |   conf.py     # created by sphinx-quickstart
    |   exhale.py   # placed here by you
    |   index.rst   # created by sphinx-quickstart
    |   about.rst   # created by you
    |   Makefile    # created by sphinx-quickstart
    --_static/
    |   --collapse/
    |          ... everything from above ...
    --_templates/
           layout.html # copied from above
```

Sphinx will make everything else fall into place in the end. If you already have your own `layout.html`, you know what you are doing — just look at mine and add the relevant lines to yours.

You can now add the key value pair `createTreeView = True` to the dictionary you are passing to *exhale.generate()*.

Warning: If you are hosting on Read the Docs, you will need to make sure you are tracking all of those files with git!

Linking to a Generated File

Every file created by exhale is given a reStructuredText label that you can use to link to the API page. It is easiest to just show how the labels are created.

```
def initializeNodeFilenameAndLink(self, node):
    html_safe_name = node.name.replace(":", "_").replace("/", "_")
    node.link_name = "{}_{}".format(qualifyKind(node.kind).lower(), html_safe_name)
```

The parameter `node` is an *exhale.ExhaleNode* object. So if the node being represented is a `struct some_thing` in `namespace arbitrary`, then

```
node.name      := "arbitrary::some_thing"
node.link_name := "struct_arbitrary__some_thing"
```

Noting that there are **two** underscores between `arbitrary` and `some`. Refer to the full documentation of *exhale.qualifyKind()* for the possible return values. If this is not working, simply generate the API once and look at the top of the file generated for the thing you are trying to link to. Copy the link (ignoring the leading underscore) and use that.

These are reStructuredText links, so in the above example you would write

```
I am linking to :ref:`struct_arbitrary__some_thing`.
```

Alternatively, you can link to a class with `:class:`namespace::ClassName``, as well as link to a method within that class using `:func:`namespace::ClassName::method``.

Customizing Breathe Output

Breathe provides you with many excellent configurations for the various reStructuredText directives it provides. Your preferences will likely be different than mine for what you do / do not want to show up. The default behavior of exhale is to use all default values for all Breathe directives except for classes and structs. Classes and structs will request documentation for `:members:`, `:protected-members:`, and `:undoc-members:`.

To change the behavior of any of the breathe directives, you will need to implement your own function and specify that as the `customSpecificationFunction` for *exhale.generate()*. Please make sure you read the documentation for *exhale.specificationsForKind()* before implementing, the requirements are very specific. An example custom implementation could be included in `conf.py` as follows:

```
def customSpecificationsForKind(kind):
    if kind == "class" or kind == "struct":
        return "   :members:\n   :protected-members:\n   :no-link:\n"
    elif kind == "enum":
        return "   :outline:\n"
    return ""
```

and you would then change the declaration of the dictionary you are passing to *exhale.generate()* to be:

```
exhaleArgs = {
    "doxygenIndexXMLPath"          : "./doxyoutput/xml/index.xml",
    "containmentFolder"            : "./generated_api",
    "rootFileName"                 : "library_root.rst",
    "rootFileTitle"                : "Library API",
    "customSpecificationFunction" : customSpecificationsForKind
 }
```

Note: The value of the key `customSpecificationFunction` is **not** a string, just the name of the function. These are first class objects in Python, which makes the above exceptionally convenient :)

Customizing `File` Pages

File pages are structured something like

File {{ filename of exhale node }}		Heading
1	Definition ({{ path to file with folders }})	Section 1
2	• Program Listing for file (hyperlink)	
... other common information ...		
3	{{ appendBreatheFileDirective }}	

Heading: Uses the file name without a path to it. If the path was `include/File.h`, then the line would be `File File.h`.

Section 1: The following Doxygen variables control what this section looks like, as well as whether or not it is included at all.

1. Set the Doxygen variable `STRIP_FROM_PATH` to change the output inside of parentheses.

   If the file path is `../include/arbitrary/File.h` and `STRIP_FROM_PATH = ..`, the parentheses line will be `Definition ( include/arbitrary/File.h )`. If you change `STRIP_FROM_PATH` to `../include`, then line 1 will be `Definition ( arbitrary/File.h )`.

   The appearance of this line will also be affected by whether or not you are using the Doxygen variable `FULL_PATH_NAMES`. In addition to leaving its default `YES` value, I have had best success with setting the `STRIP_FROM_PATH` variable.

2. If you set `XML_PROGRAMLISTING = YES`, then the code of the program (as Doxygen would display it) will be included as a bulleted hyperlink. It is the full file including whitespace, with documentation strings removed. Programming comments remain in the file.

   Unlike Doxygen, I do not link to anything in the code. Maybe sometime in the future?

3. If the value of `"appendBreatheFileDirective" = True` in the arguments passed to *exhale.generate()*, then the following section will be appended to the bottom of the file being generated:

   This will hopefully be a temporary workaround until I can figure out how to robustly parse the xml for this, or figure out how to manipulate Breathe to give me this information (since it clearly exists...). This workaround is unideal in that any errors you have in any of the documentation of the items in the file will be duplicated by the build, as well as a large number of DUPLICATE id's will be flagged. The generated links inside of the produced output by Breathe will now also link to items on this page first. AKA this is a buggy feature that I hope to fix soon, but if you *really* need the file documentation in your project, this is currently the only way to include it.

Note: If you set `XML_PROGRAMLISTING = NO`, then the file in which an `enum`, `class`, `variable`, etc is declared may **not** be recovered. To my experience, the missing items not recovered are only declared in the program-listing. See the *exhale.ExhaleRoot.fileRefDiscovery()* part of the parsing process.

Fully Automated Building

It is preferable to have everything generated at once, e.g. if you wish to host your documentation on Read the Docs. I make the assumption that you already have a `Makefile` created by `sphinx-quickstart`. Instead of a Doxyfile, though, we're going to take it one step further. Your specific arguments to Doxygen may be more involved than this, but the below should get you started in the right direction.

In `conf.py` we now define at the bottom

```
def generateDoxygenXML(stripPath):
    '''
    Generates the doxygen xml files used by breathe and exhale.
    Approach modified from:

    - https://github.com/fmtlib/fmt/blob/master/doc/build.py

    :param stripPath:
        The value you are sending to exhale.generate via the
        key 'doxygenStripFromPath'.  Usually, should be '..'.
    '''
    from subprocess import PIPE, Popen
    try:
        doxygen_cmd   = ["doxygen", "-"]# "-" tells Doxygen to read configs from stdin
        doxygen_proc  = Popen(doxygen_cmd, stdin=PIPE)
        doxygen_input = r'''
            # Make this the same as what you tell exhale.
            OUTPUT_DIRECTORY       = doxyoutput
            # If you need this to be YES, exhale will probably break.
            CREATE_SUBDIRS         = NO
            # So that only include/ and subdirectories appear.
            FULL_PATH_NAMES        = YES
            STRIP_FROM_PATH        = "%s/"
            # Tell Doxygen where the source code is (yours may be different).
            INPUT                  = ../include
            # Nested folders will be ignored without this.  You may not need it.
            RECURSIVE              = YES
            # Set to YES if you are debugging or want to compare.
            GENERATE_HTML          = NO
            # Unless you want it?
            GENERATE_LATEX         = NO
            # Both breathe and exhale need the xml.
            GENERATE_XML           = YES
            # Set to NO if you do not want the Doxygen program listing included.
            XML_PROGRAMLISTING     = YES
            # Allow for rst directives and advanced functions (e.g. grid tables)
            ALIASES                = "rst=\verbatim embed:rst:leading-asterisk"
            ALIASES               += "endrst=\endverbatim"
        ''' % stripPath)
        # In python 3 strings and bytes are no longer interchangeable
        if sys.version[0] == "3":
            doxygen_input = bytes(doxygen_input, 'ASCII')
        doxygen_proc.communicate(input=doxygen_input)
        doxygen_proc.stdin.close()
        if doxygen_proc.wait() != 0:
            raise RuntimeError("Non-zero return code from 'doxygen'...")
    except Exception as e:
        raise Exception("Unable to execute 'doxygen': {}".format(e))
```

Note: The above code should work for Python 2 and 3, but be careful not to modify the somewhat delicate treatment of strings:

- `doxygen_input = r'''...`: the `r` is required to **prevent** the verbatim rst directives to expand into control sequences (`\v`)
- In Python 3 you need to explicitly construct the bytes for communicating with the process.

Now that you have defined this at the bottom of `conf.py`, we'll add a modified `setup(app)` method:

```
# setup is called auto-magically for you by Sphinx
def setup(app):
    stripPath = ".."
    generateDoxygenXML(stripPath)

    # create the dictionary to send to exhale
    exhaleArgs = {
        "doxygenIndexXMLPath"  : "./doxyoutput/xml/index.xml",
        "containmentFolder"    : "./generated_api",
        "rootFileName"         : "library_root.rst",
        "rootFileTitle"        : "Library API",
        "doxygenStripFromPath" : stripPath
    }

    # import the exhale module from the current directory and generate the api
    sys.path.insert(0, os.path.abspath('.')) # exhale.py is in this directory
    from exhale import generate
    generate(exhaleArgs)
```

Now you can build the docs with `make html` and it will re-parse using Doxygen, generate all relevant files, and give you an updated website. While some may argue that this is wasteful, `exhale` is not smart enough and never will be smart enough to provide incremental updates. The full api is regenerated. Every time. So you may as well run Doxygen each time ;)

Note: Where Doxygen is concerned, you will likely need to give special attention to macros and preprocessor definitions. Refer to the linked `fmt` docs in the above code snippet. Of particular concern would be the following Doxygen config variables:

- `ENABLE_PREPROCESSING`
- `MACRO_EXPANSION`
- `EXPAND_ONLY_PREDEF`
- `PREDEFINED` (very useful if the Doxygen preprocessor is choking on your macros)
- `SKIP_FUNCTION_MACROS`

Doxygen Documentation Specifics

If you have not used Doxygen before, the below may be helpful in getting things started. To make sure you have Doxygen working, first try just using Doxygen and viewing the html output by setting `GENERATE_HTML = YES`. This is the default value of the variable, when you get Sphinx / Breathe / exhale going, just set this variable to `NO` to avoid creating unnecessary files.

There is a lot to make sure you do in terms of the documentation you write in a C++ file to make Doxygen work. To get started, though, execute `doxygen -g` from your terminal in a directory where there is no `Doxyfile` present and it will give you a large file called `Doxyfile` with documentation on what all of the variables do. You can leave a large number of them to their default values. To execute doxygen now, just enter `doxygen` in the same directory as the `Doxyfile` and it will generate the html output for you so you can verify it is working. Doxygen builds similarly to `make`.

Later, you can just use `conf.py` and won't need to keep your `Doxyfile`, but you could also just keep the `Doxyfile` you have working for you and execute `doxygen` with no parameters in `conf.py` before calling

exhale.generate().

1. Files you want documented **must** have `\file` somewhere. From the Doxygen documentation reiteration:

   > Let's repeat that, because it is often overlooked: to document global objects (functions, typedefs, enum, macros, etc), you must document the file in which they are defined.

2. Classes, structs, and unions need additional care in order for them to appear in the hierarchy correctly. If you have a file in a directory, the Doxygen FAQ explains that you need to specify this location:

   > You can also document your class as follows:

   ```
   /*! \class MyClassName include.h path/include.h
    *
    *  Docs for MyClassName
    */
   ```

So a minimal working example of the file `directory/file.h` defining `struct thing` might look like:

```
/** \file */
#ifndef _DIRECTORY_THING_H
#define _DIRECTORY_THING_H

/**
 * \struct thing file.h directory/file.h
 *
 * \brief The documentation about the thing.
 */
 struct thing {
    /// The thing that makes the thing.
    thing() {}
 };

#endif // _DIRECTORY_THING_H
```

3. Deviations from the norm. The cool thing about using Sphinx in this context is that you have some flexibility inherent in the fact that we are using reStructuredText. For example, instead of using `\ref`, you can just link to another documented item with `` `item` ``. This works across files as well, so you could link to class **A** in a different file from class **B** with `` `A` `` in the documentation string. You could make a statement **bold** in your documentation with just `**bold**`!

   I believe this includes the full range of reStructuredText syntax, but would not be surprised if there were directives or notation that break something.

Note: I do not support `groups` with Doxygen, as I assume if you have gone through the effort to group everything then you have a desire to manually control the output. Breathe already has an excellent `doxygengroup` directive, and you should use that.

Start to finish for Read the Docs

Assuming you already had the code that you are generating the API for documented, navigate to the top-level folder of your repository. Read the Docs (RTD) will be looking for a folder named either `doc` or `docs` at the root of your repository by default:

```
$ cd ~/my_repo/
$ mkdir docs
```

Now we are ready to begin.

1. Generate your sphinx code by using the `sphinx-quickstart` utility. It may look something like the following:

```
$ ~/my_repo/docs> sphinx-quickstart
Welcome to the Sphinx 1.3.1 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Enter the root path for documentation.
> Root path for the documentation [.]:

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]:

Inside the root directory, two more directories will be created; "_templates"
for custom HTML templates and "_static" for custom stylesheets and other static
files. You can enter another prefix (such as ".") to replace the underscore.
> Name prefix for templates and static dir [_]:

... and a whole lot more ...
```

   Warning: The default value for `> Create Makefile? (y/n) [y]:` must be yes to work on RTD. They are giving you a unix virtual environment.

2. This will create the files `conf.py`, `index.rst`, `Makefile`, and `make.bat` if you are supporting Windows. It will also create the directories `_static` and `_templates` for customizing the sphinx output.

3. Create a `requirements.txt` file with the line `breathe` so RTD will install it:

```
$ ~/my_repo/docs> echo 'breathe' > requirements.txt
```

   Alternatively, you can have RTD install via Git Tags. At the time of writing this, the latest tag for `breathe` is `4.3.1`, so in your `requirements.txt` you would have

```
git+git://github.com/michaeljones/breathe@v4.3.1#egg=breathe
```

4. Clone exhale and steal all of the files you will need:

```
$ ~/my_repo/docs> git clone https://github.com/svenevs/exhale.git
$ ~/my_repo/docs> mv exhale/exhale.py .
$ ~/my_repo/docs> mv exhale/treeView/_static/collapse/ ./_static/
$ ~/my_repo/docs> mv exhale/treeView/_templates/layout.html _templates/
$ ~/my_repo/docs> rm -rf exhale/
```

5. Uncomment the line `sys.path.insert(0, os.path.abspath('.'))` at the top of the generated `conf.py` so that Sphinx will know where to look for `exhale.py`.

6. Two options below (5) in `conf.py`, add `'breathe'` to the `extensions` list so that the directives from Breathe can be used.

7. Just below the `extensions` list, configure breathe. Adding the following should be sufficient:

```
breathe_projects = { "yourProjectName": "./doxyoutput/xml" }
breathe_default_project = "yourProjectName"
```

8. Edit `conf.py` to use the RTD Theme. You are of course able to use a different Sphinx theme, but the RTD Theme is what this will enable. Replace the `html_theme` and `html_theme_path` lines (or comment them out) with:

```
# on_rtd is whether we are on readthedocs.org, this line of code grabbed from
↪docs.readthedocs.org
on_rtd = os.environ.get('READTHEDOCS', None) == 'True'

if not on_rtd:  # only import and set the theme if we're building docs locally
    import sphinx_rtd_theme
    html_theme = 'sphinx_rtd_theme'
    html_theme_path = [sphinx_rtd_theme.get_html_theme_path()]
```

9. Edit `conf.py` to include the `generateDoxygenXML` and `setup` methods provided in *Fully Automated Building* at the bottom of the file.

10. Add `createTreeView = True` to the dictionary arguments sent to *exhale.generate()*.

11. Go to the admin page of your RTD website and select the *Advanced Settings* tab. Make sure the *Install your project inside a virtualenv using* `setup.py install` button is checked. In the *Requirements file* box below, enter `docs/requirements.txt` assuming you followed the steps above.

    I personally prefer to keep the `requirements.txt` hidden in the `docs` folder so that it is implicit that those are only requirements for building the docs, and not the actual project itself.

And you are done. Make sure you `git add` all of the files in your new `docs` directory, RTD will clone your repository / update when you push commits. You can build it locally using `make html` in the current directory, but make sure you do not add the `_build` directory to your git repository.

I hope that the above is successful for you, it looks like a lot but it's not too bad... right?

CHAPTER 3

Developer Reference Documentation

Primary Entry Point

The main entry point to exhale is through the generate function. This method internally calls breathe, reparses / rebuilds the hierarchies, and then generates the API.

`exhale.`**`generate`**(*exhaleArgs*)

The main entry point to exhale, which parses and generates the full API.

Parameters

`exhaleArgs` **(dict)** The dictionary of arguments to configure exhale with. All keys are strings, and most values should also be strings. See below.

Required Entries:

key: `"doxygenIndexXMLPath"` — value type: `str` The absolute or relative path to where the Doxygen index.xml is. A relative path must be relative to the file **calling** exhale.

key: `"containmentFolder"` — value type: `str` The folder the generated API will be created in. If the folder does not exist, exhale will create the folder. The path can be absolute, or relative to the file that is **calling** exhale. For example, `"./generated_api"`.

key: `"rootFileName"` — value type: `str` The name of the file that **you** will be linking to from your reStructuredText documents. Do not include the `containmentFolder` path in this file name, exhale will create the file `"{}/{}".format(containmentFolder, rootFileName)`.

In order for Sphinx to be happy, you should include a `.rst` suffix. All of the generated API uses reStructuredText, and that will not ever change.

For example, if you specify

- `"containmentFolder" = "./generated_api"`, and
- `"rootFileName" = "library_root.rst"`

Then exhale will generate the file `./generated_api/library_root.rst`.

You could include this file in a toctree directive (say in `index.rst`) with:

```
.. toctree:
   :maxdepth: 2

   generated_api/library_root
```

Since Sphinx allows for some flexibility (e.g. your primary domain may be using `.txt` files), **no error checking will be performed**.

key: `"rootFileTitle"` — value type: `str` The title to be written at the top of `rootFileName`, which will appear in your file including it in the `toctree` directive.

key: `"doxygenStripFromPath"` — value type: `str` When building on Read the Docs, there seem to be issues regarding the Doxygen variable `STRIP_FROM_PATH` when built remotely. That is, it isn't stripped at all. Provide me with a string path (e.g. `".."`), and I will strip this for you for the File nodes being generated. I will use the exact value of `os.path.abspath("..")` in the example above, so you can supply either a relative or absolute path. The File view hierarchy **will** break if you do not give me a value for this, and therefore I hesitantly require this argument. The value `".."` assumes that `conf.py` is in a `docs/` or similar folder exactly one level below the repository's root.

Additional Options:

key: `"afterTitleDescription"` — value type: `str` Properly formatted reStructuredText with **no indentation** to be included directly after the title. You can use any rst directives or formatting you wish in this string. I suggest using the `textwrap` module, e.g.:

```
description = textwrap.dedent('''
This is a description of the functionality of the library being documented.

.. warning::

   Please be advised that this library does not do anything.
''')
```

Then you can add `"afterTitleDescription" = description` to your dictionary.

key: `"afterBodySummary"` — value type: `str` Similar to `afterTitleDescription`, this is a string with reStructuredText formatting. This will be inserted after the generated API body. The layout looks something like this:

```
rootFileTitle
================================================================================

afterTitleDescription (if provided)

[[[ GENERATED API BODY ]]]

afterBodySummary (if provided)
```

key: `"createTreeView"` — value type: `bool` For portability, the default value if not specified is `False`, which will generate reStructuredText bulleted lists for the Class View and File View hierarchies. If `True`, raw html unordered lists will be generated. Please refer to the *Clickable Hierarchies* subsection of *Additional Usage and Customization* for more details.

key: `"fullToctreeMaxDepth"` — value type: `int` Beneath the Class View and File View hierarchies a Full API listing is generated as there are items that may not appear in the Class View hierarchy, as well as without this an obscene amount of warnings are generated from Sphinx because neither view actually uses a `toctree`, they link directly.

The default value is 5 if not specified, but you may want to give a smaller value depending on the framework being documented. This value must be greater than or equal to 1 (this is the value of `:maxdepth:`).

key: `"appendBreatheFileDirective"` — value type: `bool` Currently, I do not know how to reliably extract the brief / detailed file descriptions for a given file node. Therefore, if you have file level documentation in your project that has meaning, it would otherwise be omitted. As a temporary patch, if you specify this value as `True` then at the bottom of the file page the full `doxygenfile` directive output from Breathe will be appended to the file documentiation. File level brief and detailed descriptions will be included, followed by a large amount of duplication. I hope to remove this value soon, in place of either parsing the xml more carefully or finding out how to extract this information directly from Breathe.

The default value of this behavior is `False` if it is not specified in the dictionary passed as input for this method. Please refer to the *Customizing File Pages* subsection of *Customizing File Pages* for more information on what the impact of this variable is.

key: `"customSpecificationFunction"` — value type: `function` The custom specification function to override the default behavior of exhale. Please refer to the *exhale.specificationsForKind()* documentation.

Raises

- **`ValueError`** – If the required dictionary arguments are not present, or any of the (key, value) pairs are invalid.
- **`RuntimeError`** – If any **fatal** error is caught during the generation of the API.

Helper Functions

There are a few helper functions used throughout the framework that effectively just reformat the input into a specific kind of output for incorporating into reStructuredText documents, and the directives used in those documents. The last function is largely unrelated to exhale, and just prints something to the console in a way that makes it stick out a little more.

`exhale.`**`qualifyKind`**(*kind*)

Qualifies the breathe `kind` and returns an qualifier string describing this to be used for the text output (e.g. in generated file headings and link names).

The output for a given kind is as follows:

Input Kind	Output Qualifier
“class”	“Class”
“define”	“Define”
“enum”	“Enum”
“enumvalue”	“Enumvalue”
“file”	“File”
“function”	“Function”
“group”	“Group”
“namespace”	“Namespace”
“struct”	“Struct”
“typedef”	“Typedef”
“union”	“Union”
“variable”	“Variable”

The following breathe kinds are ignored:

- “autodoxygenfile”

- "doxygenindex"
- "autodoxygenindex"

Note also that although a return value is generated, neither "enumvalue" nor "group" are actually used.

Parameters

kind (str) The return value of a Breathe `compound` object's `get_kind()` method.

Return (str) The qualifying string that will be used to build the reStructuredText titles and other qualifying names. If the empty string is returned then it was not recognized.

`exhale.`**`kindAsBreatheDirective`**(*kind*)

Returns the appropriate breathe restructured text directive for the specified kind. The output for a given kind is as follows:

Input Kind	Output Directive
"class"	"doxygenclass"
"define"	"doxygendefine"
"enum"	"doxygenenum"
"enumvalue"	"doxygenenumvalue"
"file"	"doxygenfile"
"function"	"doxygenfunction"
"group"	"doxygengroup"
"namespace"	"doxygennamespace"
"struct"	"doxygenstruct"
"typedef"	"doxygentypedef"
"union"	"doxygenunion"
"variable"	"doxygenvariable"

The following breathe kinds are ignored:

- "autodoxygenfile"
- "doxygenindex"
- "autodoxygenindex"

Note also that although a return value is generated, neither "enumvalue" nor "group" are actually used.

Parameters

kind (str) The kind of the breathe compound / ExhaleNode object (same values).

Return (str) The directive to be used for the given `kind`. The empty string is returned for both unrecognized and ignored input values.

`exhale.`**`specificationsForKind`**(*kind*)

Returns the relevant modifiers for the restructured text directive associated with the input kind. The only considered values for the default implementation are `class` and `struct`, for which the return value is exactly:

```
"   :members:\n   :protected-members:\n   :undoc-members:\n"
```

Formatting of the return is fundamentally important, it must include both the prior indentation as well as newlines separating any relevant directive modifiers. The way the framework uses this function is very specific; if you do not follow the conventions then sphinx will explode.

Consider a `struct thing` being documented. The file generated for this will be:

```
.. _struct_thing:

Struct thing
```

```
================================================================================

.. doxygenstruct:: thing
   :members:
   :protected-members:
   :undoc-members:
```

Assuming the first two lines will be in a variable called `link_declaration`, and the next three lines are stored in `header`, the following is performed:

```
directive = ".. {}:: {}\n".format(kindAsBreatheDirective(node.kind), node.name)
specifications = "{}\n\n".format(specificationsForKind(node.kind))
gen_file.write("{}{}{}{}".format(link_declaration, header, directive,␣
↪specifications))
```

That is, **no preceding newline** should be returned from your custom function, and **no trailing newline** is needed. Your indentation for each specifier should be **exactly three spaces**, and if you want more than one you need a newline in between every specification you want to include. Whitespace control is handled internally because many of the directives do not need anything added. For a full listing of what your specifier options are, refer to the breathe documentation:

http://breathe.readthedocs.io/en/latest/directives.html

Parameters

kind (str) The kind of the node we are generating the directive specifications for.

Return (str) The correctly formatted specifier(s) for the given `kind`. If no specifier(s) are necessary or desired, the empty string is returned.

`exhale.`**`exclaimError`**(*msg*, *ansi_fmt='34;1m'*)

Prints `msg` to the console in color with `(!)` prepended in color.

Example (uncolorized) output of `exclaimError("No leading space needed.")`:

```
(!) No leading space needed.
```

All messages are written to `sys.stderr`, and are closed with `[0m`. The default color is blue, but can be changed using `ansi_fmt`.

Documentation building has a verbose output process, this just helps distinguish an error message coming from exhale.

Parameters

msg (str) The message you want printed to standard error.

ansi_fmt (str) An ansi color format. `msg` is printed as `"\033[" + ansi_fmt + msg + "\033[0m\n`, so you should specify both the color code and the format code (after the semicolon). The default value is `34;1m` — refer to http://misc.flogisoft.com/bash/tip_colors_and_formatting for alternatives.

Exposed Utility Variables

`exhale.`**`EXHALE_FILE_HEADING`**

The restructured text file heading separator (`"=" * 88`).

exhale.**EXHALE_SECTION_HEADING**
The restructured text section heading separator (`"-" * 88`).

exhale.**EXHALE_SUBSECTION_HEADING**
The restructured text sub-section heading separator (`"*" * 88`).

Ownership Graph Representation

A graph representing what classes belong to what namespaces, what file defines what, etc is built with a single `ExhaleRoot`. This root node contains multiple different lists of `ExhaleNode` objects that it parses from both Breathe and the Doxygen xml output.

If you are reading this, then you are likely trying to make changes. To avoid having such a huge reference page, and enable viewing the reference documentation for the two primary classes at the same time, they are on separate pages.

Primary Class ExhaleRoot Reference

class exhale.**ExhaleRoot**(*breatheRoot*, *rootDirectory*, *rootFileName*, *rootFileTitle*, *rootFileDescription*, *rootFileSummary*, *createTreeView*)

The full representation of the hierarchy graphs. In addition to containing specific lists of ExhaleNodes of interest, the ExhaleRoot class is responsible for comparing the parsed breathe hierarchy and rebuilding lost relationships using the Doxygen xml files. Once the graph parsing has finished, the ExhaleRoot generates all of the relevant reStructuredText documents and links them together.

The ExhaleRoot class is not designed for reuse at this time. If you want to generate a new hierarchy with a different directory or something, changing all of the right fields may be difficult and / or unsuccessful. Refer to the bottom of the source code for *exhale.generate()* for safe usage (just exception handling), but the design of this class is to be used as follows:

```
textRoot = ExhaleRoot(... args ...)
textRoot.parse()
textRoot.generateFullAPI()
```

Zero checks are in place to enforce this usage, and if you are modifying the execution of this class and things are not working make sure you follow the ordering of those methods.

Parameters

breatheRoot (instance) Type unknown, this is the return value of `breathe.breathe_parse`.

rootDirectory (str) The name of the root directory to put everything in. This should be the value of the key `containmentFolder` in the dictionary passed to *exhale.generate()*.

rootFileName (str) The name of the file the root library api will be put into. This should not contain the `rootDirectory` path. This should be the value of the key `rootFileName` in the dictionary passed to *exhale.generate()*.

rootFileTitle (str) The title to be written to the top of `rootFileName`. This should be the value of the key `rootFileTitle` in the dictionary passed to *exhale.generate()*.

rootFileDescription (str) The description of the library api file placed after `rootFileTitle`. This should be the value of the key `afterTitleDescription` in the dictionary passed to *exhale.generate()*.

rootFileSummary (str) The summary of the library api placed after the generated hierarchy views. This should be the value of the key `afterBodySummary` in the dictionary passed to *exhale.generate()*.

createTreeView (bool) Creates the raw html unordered lists for use with `collapsibleList` if True. Otherwise, creates standard reStructuredText bulleted lists. Should be the value of the key `createTreeView` in the dictionary passed to *exhale.generate()*.

Attributes

breathe_root (instance) The value of the parameter `breatheRoot`.

root_directory (str) The value of the parameter `rootDirectory`.

root_file_name (str) The value of the parameter `rootFileName`.

full_root_file_path (str) The full file path of the root file (`"root_directory/root_file_name"`).

root_file_title (str) The value of the parameter `rootFileTitle`.

root_file_description (str) The value of the parameter `rootFileDescription`.

root_file_summary (str) The value of the parameter `rootFileSummary`.

class_view_file (str) The full file path the class view hierarchy will be written to. This is incorporated into `root_file_name` using an `.. include:` directive.

directory_view_file (str) The full file path the file view hierarchy will be written to. This is incorporated into `root_file_name` using an `.. include:` directive.

unabridged_api_file (str) The full file path the full API will be written to. This is incorporated into `root_file_name` using a `.. toctree:` directive with a `:maxdepth:` according to the value of the key `fullToctreeMaxDepth` in the dictionary passed into *exhale.generate()*.

use_tree_view (bool) The value of the parameter `createTreeView`.

all_compounds (list) A list of all the Breathe compound objects discovered along the way. Populated during *exhale.ExhaleRoot.discoverAllNodes()*.

all_nodes (list) A list of all of the ExhaleNode objects created. Populated during *exhale.ExhaleRoot.discoverAllNodes()*.

node_by_refid (dict) A dictionary with string ExhaleNode `refid` values, and values that are the ExhaleNode it came from. Storing it this way is convenient for when the Doxygen xml file is being parsed.

class_like (list) The full list of ExhaleNodes of kind `struct` or `class`

defines (list) The full list of ExhaleNodes of kind `define`.

enums (list) The full list of ExhaleNodes of kind `enum`.

enum_values (list) The full list of ExhaleNodes of kind `enumvalue`. Populated, not used.

functions (list) The full list of ExhaleNodes of kind `function`.

dirs (list) The full list of ExhaleNodes of kind `dir`.

files (list) The full list of ExhaleNodes of kind `file`.

groups (list) The full list of ExhaleNodes of kind `group`. Pupulated, not used.

namespaces (list) The full list of ExhaleNodes of kind `namespace`.

typedefs (list) The full list of ExhaleNodes of kind `typedef`.

unions (list) The full list of ExhaleNodes of kind `union`.

variables (list) The full list of ExhaleNodes of kind `variable`.

parse()
The first method that should be called after creating an ExhaleRoot object. The Breathe graph is parsed first, followed by the Doxygen xml documents. By the end of this method, all of the `self.<breathe_kind>`, `self.all_compounds`, and `self.all_nodes` lists as well as the `self.node_by_refid` dictionary will be populated. Lastly, this method sorts all of the internal lists. The order of execution is exactly

1. *exhale.ExhaleRoot.discoverAllNodes()*
2. *exhale.ExhaleRoot.reparentAll()*
3. Populate `self.node_by_refid` using `self.all_nodes`.
4. *exhale.ExhaleRoot.fileRefDiscovery()*
5. *exhale.ExhaleRoot.filePostProcess()*
6. *exhale.ExhaleRoot.sortInternals()*

discoverAllNodes()
Stack based traversal of breathe graph, creates some parental relationships between different ExhaleNode objects. Upon termination, this method will have populated the lists `self.all_compounds`, `self.all_nodes`, and the `self.<breathe_kind>` lists for different types of objects.

trackNodeIfUnseen(*node*)
Helper method for *exhale.ExhaleRoot.discoverAllNodes()*. If the node is not in self.all_nodes yet, add it to both self.all_nodes as well as the corresponding `self.<breathe_kind>` list.

Parameters

node (ExhaleNode) The node to begin tracking if not already present.

discoverNeigbors(*nodesRemaining*, *node*)
Helper method for *exhale.ExhaleRoot.discoverAllNodes()*. Some of the compound objects received from Breathe have a member function `get_member()` that returns all of the children. Some do not. This method checks to see if the method is present first, and if so performs the following:

```
For every compound in node.compound.get_member():
    If compound not present in self.all_compounds:
        - Add compound to self.all_compounds
        - Create a child ExhaleNode
        - If it is not a class, struct, or union, add to nodesRemaining
        - If it is not an enumvalue, make it a child of node parameter
```

Parameters

nodesRemaining (list) The list of nodes representing the stack traversal being done by *exhale.ExhaleRoot.discoverAllNodes()*. New neighbors found will be appended to this list.

node (ExhaleNode) The node we are trying to discover potential new neighbors from.

reparentAll()
Fixes some of the parental relationships lost in parsing the Breathe graph. File relationships are recovered in *exhale.ExhaleRoot.fileRefDiscovery()*. This method simply calls in this order:

1. *exhale.ExhaleRoot.reparentUnions()*
2. *exhale.ExhaleRoot.reparentClassLike()*
3. *exhale.ExhaleRoot.reparentDirectories()*
4. *exhale.ExhaleRoot.renameToNamespaceScopes()*
5. *exhale.ExhaleRoot.reparentNamespaces()*

reparentUnions()
Helper method for *exhale.ExhaleRoot.reparentAll()*. Namespaces and classes should have the unions defined in them to be in the child list of itself rather than floating around. Union nodes that are reparented (e.g. a union defined in a class) will be removed from the list `self.unions` since the Breathe directive for its parent (e.g. the class) will include the documentation for the union. The consequence of this is that a union defined in a class will **not** appear in the full api listing of Unions.

reparentClassLike()
Helper method for *exhale.ExhaleRoot.reparentAll()*. Iterates over the `self.class_like` list and adds each object as a child to a namespace if the class, or struct is a member of that namespace. Many classes / structs will be reparented to a namespace node, these will remain in `self.class_like`. However, if a class or struct is reparented to a different class or struct (it is a nested class / struct), it *will* be removed from so that the class view hierarchy is generated correctly.

reparentDirectories()
Helper method for *exhale.ExhaleRoot.reparentAll()*. Adds subdirectories as children to the relevant directory ExhaleNode. If a node in `self.dirs` is added as a child to a different directory node, it is removed from the `self.dirs` list.

renameToNamespaceScopes()
Helper method for *exhale.ExhaleRoot.reparentAll()*. Some compounds in Breathe such as functions and variables do not have the namespace name they are declared in before the name of the actual compound. This method prepends the appropriate (nested) namespace name before the name of any child that does not already have it.

For example, the variable `MAX_DEPTH` declared in namespace `external` would have its ExhaleNode's `name` attribute changed from `MAX_DEPTH` to `external::MAX_DEPTH`.

reparentNamespaces()
Helper method for *exhale.ExhaleRoot.reparentAll()*. Adds nested namespaces as children to the relevant namespace ExhaleNode. If a node in `self.namespaces` is added as a child to a different namespace node, it is removed from the `self.namespaces` list. Because these are removed from `self.namespaces`, it is important that *exhale.ExhaleRoot.renameToNamespaceScopes()* is called before this method.

fileRefDiscovery()
Finds the missing components for file nodes by parsing the Doxygen xml (which is just the `doxygen_output_dir/node.refid`). Additional items parsed include adding items whose `refid` tag are used in this file, the <programlisting> for the file, what it includes and what includes it, as well as the location of the file (with respsect to the *Doxygen* root).

Care must be taken to only include a refid found with specific tags. The parsing of the xml file was done by just looking at some example outputs. It seems to be working correctly, but there may be some subtle use cases that break it.

> **Warning:** Some enums, classes, variables, etc declared in the file will not have their associated refid in the declaration of the file, but will be present in the <programlisting>. These are added to the files' list of children when they are found, but this parental relationship cannot be formed if you set

`XML_PROGRAMLISTING = NO` with Doxygen. An example of such an enum would be an enum declared inside of a namespace within this file.

filePostProcess()
The real name of this method should be `reparentFiles`, but to avoid confusion with what stage this must happen at it is called this instead. After the *exhale.ExhaleRoot.fileRefDiscovery()* method has been called, each file will have its location parsed. This method reparents files to directories accordingly, so the file view hierarchy can be complete.

sortInternals()
Sort all internal lists (`class_like`, `namespaces`, `variables`, etc) mostly how doxygen would, alphabetical but also hierarchical (e.g. structs appear before classes in listings). Some internal lists are just sorted, and some are deep sorted (*exhale.ExhaleRoot.deepSortList()*).

deepSortList(*lst*)
For hierarchical internal lists such as `namespaces`, we want to sort both the list as well as have each child sort its children by calling *exhale.ExhaleNode.typeSort()*.

Parameters

lst (list) The list of ExhaleNode objects to be deep sorted.

generateFullAPI()
Since we are not going to use some of the breathe directives (e.g. namespace or file), when representing the different views of the generated API we will need:

1. Generate a single file restructured text document for all of the nodes that have either no children, or children that are leaf nodes.
2. When building the view hierarchies (class view and file view), provide a link to the appropriate files generated previously.

If adding onto the framework to say add another view (from future import groups) you would link from a restructured text document to one of the individually generated files using the value of `link_name` for a given ExhaleNode object.

This method calls in this order:

1. *exhale.ExhaleRoot.generateAPIRootHeader()*
2. *exhale.ExhaleRoot.generateNodeDocuments()*
3. *exhale.ExhaleRoot.generateAPIRootBody()*
4. *exhale.ExhaleRoot.generateAPIRootSummary()*

generateAPIRootHeader()
This method creates the root library api file that will include all of the different hierarchy views and full api listing. If `self.root_directory` is not a current directory, it is created first. Afterward, the root API file is created and its title is written, as well as the value of `self.root_file_description`.

generateNodeDocuments()
Creates all of the reStructuredText documents related to types parsed by Doxygen. This includes all leaf-like documents (`class`, `struct`, `enum`, `typedef`, `union`, `variable`, and `define`), as well as namespace, file, and directory pages.

During the reparenting phase of the parsing process, nested items were added as a child to their actual parent. For classes, structs, enums, and unions, if it was reparented to a `namespace` it will *remain* in its respective `self.<breathe_kind>` list. However, if it was an internally declared child of a class or struct (nested classes, structs, enums, and unions), this node will be removed from its `self.<breathe_kind>` list to avoid duplication in the class hierarchy generation.

When generating the full API, though, we will want to include all of these and therefore must call *exhale.ExhaleRoot.generateSingleNodeRST()* with all of the nested items. For nested classes and structs, this is done by just calling `node.findNestedClassLike` for every node in `self.class_like`. The resulting list then has all of `self.class_like`, as well as any nested classes and structs found. With `enum` and `union`, these would have been reparented to a **class** or **struct** if it was removed from the relevant `self.<breathe_kind>` list. Meaning we must make sure that we genererate the single node RST documents for everything by finding the nested enums and unions from `self.class_like`, as well as everything in `self.enums` and `self.unions`.

initializeNodeFilenameAndLink(*node*)

Sets the `file_name` and `link_name` for the specified node. If the kind of this node is "file", then this method will also set the `program_file` as well as the `program_link_name` fields.

Since we are operating inside of a `containmentFolder`, this method **will** include `self.root_directory` in this path so that you can just use:

```
with open(node.file_name, "w") as gen_file:
    ... write the file ...
```

Having the `containmentFolder` is important for when we want to generate the file, but when we want to use it with `include` or `toctree` this will need to change. Refer to *exhale.ExhaleRoot.gerrymanderNodeFilenames()*.

This method also sets the value of `node.title`, which will be used in both the reStructuredText document of the node as well as the links generated in the class view hierarchy (<a href="...">> for the `createTreeView = True` option).

Type exhale.ExhaleNode

Param node The node that we are setting the above information for.

generateSingleNodeRST(*node*)

Creates the reStructuredText document for the leaf like node object. This method should only be used with nodes in the following member lists:

- `self.class_like`
- `self.enums`
- `self.functions`
- `self.typedefs`
- `self.unions`
- `self.variables`
- `self.defines`

File, directory, and namespace nodes are treated separately.

Parameters

node (ExhaleNode) The leaf like node being generated by this method.

generateNamespaceNodeDocuments()

Generates the reStructuredText document for every namespace, including nested namespaces that were removed from `self.namespaces` (but added as children to one of the namespaces in `self.namespaces`).

The documents generated do not use the Breathe namespace directive, but instead link to the relevant documents associated with this namespace.

generateSingleNamespace(*nspace*)
Helper method for *exhale.ExhaleRoot.generateNamespaceNodeDocuments()*. Writes the reStructuredText file for the given namespace.

Parameters

nspace (ExhaleNode) The namespace node to create the reStructuredText document for.

generateNamespaceChildrenString(*nspace*)
Helper method for *exhale.ExhaleRoot.generateSingleNamespace()*, and *exhale.ExhaleRoot.generateFileNodeDocuments()*. Builds the body text for the namespace node document that links to all of the child namespaces, structs, classes, functions, typedefs, unions, and variables associated with this namespace.

Parameters

nspace (ExhaleNode) The namespace node we are generating the body text for.

Return (str) The string to be written to the namespace node's reStructuredText document.

generateSortedChildListString(*sectionTitle*, *previousString*, *lst*)
Helper method for *exhale.ExhaleRoot.generateNamespaceChildrenString()*. Used to build up a continuous string with all of the children separated out into titled sections.

This generates a new titled section with `sectionTitle` and puts a link to every node found in `lst` in this section. The newly created section is appended to `previousString` and then returned.

TODO Change this to use string streams like the other methods instead.

Parameters

sectionTitle (str) The title of the section for this list of children.

previousString (str) The string to append the newly created section to.

lst (list) A list of ExhaleNode objects that are to be linked to from this section. This method sorts `lst` in place.

generateFileNodeDocuments()
Generates the reStructuredText documents for files as well as the file's program listing reStructuredText document if applicable. Refer to *Customizing File Pages* for changing the output of this method. The remainder of the file lists all nodes that have been discovered to be defined (e.g. classes) or referred to (e.g. included files or files that include this file).

generateDirectoryNodeDocuments()
Generates all of the directory reStructuredText documents.

generateDirectoryNodeRST(*node*)
Helper method for *exhale.ExhaleRoot.generateDirectoryNodeDocuments()*. Generates the reStructuredText documents for the given directory node. Directory nodes will only link to files and subdirectories within it.

Parameters

node (ExhaleNode) The directory node to generate the reStructuredText document for.

generateAPIRootBody()
Generates the root library api file's body text. The method calls *exhale.ExhaleRoot.gerrymanderNodeFilenames()* first to enable proper internal linkage between reStructuredText documents. Afterward, it calls *exhale.ExhaleRoot.generateViewHierarchies()* followed by *exhale.ExhaleRoot.generateUnabridgedAPI()* to generate both hierarchies as well as the full API listing. As a result, three files will now be ready:

1. `self.class_view_file`

2. `self.directory_view_file`

3. `self.unabridged_api_file`

These three files are then *included* into the root library file. The consequence of using an `include` directive is that Sphinx will complain about these three files never being included in any `toctree` directive. These warnings are expected, and preferred to using a `toctree` because otherwise the user would have to click on the class view link from the `toctree` in order to see it. This behavior has been acceptable for me so far, but if it is causing you problems please raise an issue on GitHub and I may be able to conditionally use a `toctree` if you really need it.

gerrymanderNodeFilenames()
When creating nodes, the filename needs to be relative to `conf.py`, so it will include `self.root_directory`. However, when generating the API, the file we are writing to is in the same directory as the generated node files so we need to remove the directory path from a given ExhaleNode's `file_name` before we can `include` it or use it in a `toctree`.

generateViewHierarchies()
Wrapper method to create the view hierarchies. Currently it just calls *exhale.ExhaleRoot.generateClassView()* and *exhale.ExhaleRoot.generateDirectoryView()* — if you want to implement additional hierarchies, implement the additionaly hierarchy method and call it from here. Then make sure to `include` it in *exhale.ExhaleRoot.generateAPIRootBody()*.

generateClassView(*treeView*)
Generates the class view hierarchy, writing it to `self.class_view_file`.

Parameters

treeView (bool) Whether or not to use the collapsibleList version. See the `createTreeView` description in *exhale.generate()*.

generateDirectoryView(*treeView*)
Generates the file view hierarchy, writing it to `self.directory_view_file`.

Parameters

treeView (bool) Whether or not to use the collapsibleList version. See the `createTreeView` description in *exhale.generate()*.

generateUnabridgedAPI()
Generates the unabridged (full) API listing into `self.unabridged_api_file`. This is necessary as some items may not show up in either hierarchy view, depending on:

1. The item. For example, if a namespace has only one member which is a variable, then neither the namespace nor the variable will be declared in the class view hierarchy. It will be present in the file page it was declared in but not on the main library page.

2. The configurations of Doxygen. For example, see the warning in *exhale.ExhaleRoot.fileRefDiscovery()*. Items whose parents cannot be rediscovered withouth the programlisting will still be documented, their link appearing in the unabridged API listing.

Currently, the API is generated in the following (somewhat arbitrary) order:

- Namespaces
- Classes and Structs
- Enums
- Unions
- Functions
- Variables

- Defines
- Typedefs
- Directories
- Files

If you want to change the ordering, just change the order of the calls to *exhale.ExhaleRoot.enumerateAll()* in this method.

enumerateAll(*subsectionTitle*, *lst*, *openFile*)
Helper function for *exhale.ExhaleRoot.generateUnabridgedAPI()*. Simply writes a subsection to `openFile` (a `toctree` to the `file_name`) of each ExhaleNode in `sorted(lst)` if `len(lst) > 0`. Otherwise, nothing is written to the file.

Parameters

subsectionTitle (str) The title of this subsection, e.g. `"Namespaces"` or `"Files"`.

lst (list) The list of ExhaleNodes to be enumerated in this subsection.

openFile (File) The **already open** file object to write to directly. No safety checks are performed, make sure this is a real file object that has not been closed already.

generateAPIRootSummary()
Writes the library API root summary to the main library file. See the documentation for the key `afterBodySummary` in *exhale.generate()*.

toConsole()
Convenience function for printing out the entire API being generated to the console. Unused in the release, but is helpful for debugging ;)

consoleFormat(*sectionTitle*, *lst*)
Helper method for *exhale.ExhaleRoot.toConsole()*. Prints the given `sectionTitle` and calls *exhale.ExhaleNode.toConsole()* with `0` as the level for every ExhaleNode in `lst`.

Parameters

sectionTitle (str) The title that will be printed with some visual separators around it.

lst (list) The list of ExhaleNodes to print to the console.

Helper Class ExhaleNode Reference

class exhale.**ExhaleNode**(*breatheCompound*)
A wrapper class to track parental relationships, filenames, etc.

Parameters

breatheCompound (breathe.compound) The Breathe compound object we will use to gather the name, chilren, etc.

Attributes

compound (breathe.compound) The compound discovered from breathe that we are going to track.

kind (str) The string returned by the `breatheCompound.get_kind()` method. Used to qualify this node throughout the framework, as well as for hierarchical sorting.

name (str) The string returned by the `breatheCompound.get_name()` method. This name will be fully qualified — `class A` inside of `namespace n` will have a `name` of `n::A`. Files and directories may have `/` characters as well.

refid (str) The reference ID as created by Doxygen. This will be used to scrape files and see if a given reference identification number should be associated with that file or not.

children (list) A potentially empty list of `ExhaleNode` object references that are considered a child of this Node. Please note that a child reference in any `children` list may be stored in **many** other lists. Mutating a given child will mutate the object, and therefore affect other parents of this child. Lastly, a node of kind `enum` will never have its `enumvalue` children as it is impossible to rebuild that relationship without more Doxygen xml parsing.

parent (ExhaleNode) If an ExhaleNode is determined to be a child of another ExhaleNode, this node will be added to its parent's `children` list, and a reference to the parent will be in this field. Initialized to `None`, make sure you check that it is an object first.

> **Warning:** Do not ever set the `parent` of a given node if the would-be parent's kind is `"file"`. Doing so will break many important relationships, such as nested class definitions. Effectively, **every** node will be added as a child to a file node at some point. The file node will track this, but the child should not.

The following three member variables are stored internally, but managed externally by the *exhale.ExhaleRoot* class:

file_name (str) The name of the file to create. Set to `None` on creation, refer to *exhale.ExhaleRoot.initializeNodeFilenameAndLink()*.

link_name (str) The name of the reStructuredText link that will be at the top of the file. Set to `None` on creation, refer to *exhale.ExhaleRoot.initializeNodeFilenameAndLink()*.

title (str) The title that will appear at the top of the reStructuredText file `file_name`. When the reStructuredText document for this node is being written, the root object will set this field.

The following two fields are used for tracking what has or has not already been included in the hierarchy views. Things like classes or structs in the global namespace will not be found by *exhale.ExhaleNode.inClassView()*, and the ExhaleRoot object will need to track which ones were missed.

in_class_view (bool) Whether or not this node has already been incorporated in the class view.

in_file_view (bool) Whether or not this node has already been incorporated in the file view.

This class wields duck typing. If `self.kind == "file"`, then the additional member variables below exist:

namespaces_used (list) A list of namespace nodes that are either defined or used in this file.

includes (list) A list of strings that are parsed from the Doxygen xml for this file as include directives.

included_by (list) A list of (refid, name) string tuples that are parsed from the Doxygen xml for this file presenting all of the other files that include this file. They are stored this way so that the root class can later link to that file by its refid.

location (str) A string parsed from the Doxygen xml for this file stating where this file is physically in relation to the *Doxygen* root.

program_listing (list) A list of strings that is the Doxygen xml <programlisting>, without the opening or closing <programlisting> tags.

program_file (list) Managed externally by the root similar to `file_name` etc, this is the name of the file that will be created to display the program listing if it exists. Set to `None` on creation, refer to *exhale.ExhaleRoot.initializeNodeFilenameAndLink()*.

program_link_name (str) Managed externally by the root similar to `file_name` etc, this is the reStructuredText link that will be declared at the top of the `program_file`. Set to `None` on creation, refer to *exhale.ExhaleRoot.initializeNodeFilenameAndLink()*.

__lt__(*other*)

The `ExhaleRoot` class stores a bunch of lists of `ExhaleNode` objects. When these lists are sorted, this method will be called to perform the sorting.

Parameters

other (ExhaleNode) The node we are comparing whether `self` is less than or not.

Return (bool) True if `self` is less than `other`, False otherwise.

findNestedNamespaces(*lst*)

Recursive helper function for finding nested namespaces. If this node is a namespace node, it is appended to `lst`. Each node also calls each of its child `findNestedNamespaces` with the same list.

Parameters

lst (list) The list each namespace node is to be appended to.

findNestedDirectories(*lst*)

Recursive helper function for finding nested directories. If this node is a directory node, it is appended to `lst`. Each node also calls each of its child `findNestedDirectories` with the same list.

Parameters

lst (list) The list each directory node is to be appended to.

findNestedClassLike(*lst*)

Recursive helper function for finding nested classes and structs. If this node is a class or struct, it is appended to `lst`. Each node also calls each of its child `findNestedClassLike` with the same list.

Parameters

lst (list) The list each class or struct node is to be appended to.

findNestedEnums(*lst*)

Recursive helper function for finding nested enums. If this node is a class or struct it may have had an enum added to its child list. When this occurred, the enum was removed from `self.enums` in the *exhale.ExhaleRoot* class and needs to be rediscovered by calling this method on all of its children. If this node is an enum, it is because a parent class or struct called this method, in which case it is added to `lst`.

Note: this is used slightly differently than nested directories, namespaces, and classes will be. Refer to *exhale.ExhaleRoot.generateNodeDocuments()* function for details.

Parameters

lst (list) The list each enum is to be appended to.

findNestedUnions(*lst*)

Recursive helper function for finding nested unions. If this node is a class or struct it may have had a

union added to its child list. When this occurred, the union was removed from `self.unions` in the *exhale.ExhaleRoot* class and needs to be rediscovered by calling this method on all of its children. If this node is a union, it is because a parent class or struct called this method, in which case it is added to `lst`.

Note: this is used slightly differently than nested directories, namespaces, and classes will be. Refer to *exhale.ExhaleRoot.generateNodeDocuments()* function for details.

Parameters

lst (list) The list each union is to be appended to.

toConsole(*level*, *printChildren=True*)

Debugging tool for printing hierarchies / ownership to the console. Recursively calls children `toConsole` if this node is not a directory or a file, and `printChildren == True`.

Parameters

level (int) The indentation level to be used, should be greater than or equal to 0.

printChildren (bool) Whether or not the `toConsole` method for the children found in `self.children` should be called with `level+1`. Default is True, set to False for directories and files.

typeSort()

Sorts `self.children` in place, and has each child sort its own children. Refer to *exhale.ExhaleRoot.deepSortList()* for more information on when this is necessary.

inClassView()

Whether or not this node should be included in the class view hierarchy. Helper method for *exhale.ExhaleNode.toClassView()*. Sets the member variable `self.in_class_view` to True if appropriate.

Return (bool) True if this node should be included in the class view — either it is a node of kind `struct`, `class`, `enum`, `union`, or it is a `namespace` that one or more if its descendants was one of the previous four kinds. Returns False otherwise.

toClassView(*level*, *stream*, *treeView*, *lastChild=False*)

Recursively generates the class view hierarchy using this node and its children, if it is determined by *exhale.ExhaleNode.inClassView()* that this node should be included.

Parameters

level (int) An integer greater than or equal to 0 representing the indentation level for this node.

stream (StringIO) The stream that is being written to by all of the nodes (created and destroyed by the ExhaleRoot object).

treeView (bool) If False, standard reStructuredText bulleted lists will be written to the `stream`. If True, then raw html unordered lists will be written to the `stream`.

lastChild (bool) When `treeView == True`, the unordered lists generated need to have an <li class="lastChild"> tag on the last child for the `collapsibleList` to work correctly. The default value of this parameter is False, and should only ever be set to True internally by recursive calls to this method.

inDirectoryView()

Whether or not this node should be included in the file view hierarchy. Helper method for *exhale.ExhaleNode.toDirectoryView()*. Sets the member variable `self.in_directory_view` to True if appropriate.

Return (bool) True if this node should be included in the file view — either it is a node of kind `file`, or it is a `dir` that one or more if its descendants was a `file`. Returns False otherwise.

toDirectoryView(*level*, *stream*, *treeView*, *lastChild=False*)

Recursively generates the file view hierarchy using this node and its children, if it is determined by *exhale.ExhaleNode.inDirectoryView()* that this node should be included.

Parameters

level (int) An integer greater than or equal to 0 representing the indentation level for this node.

stream (StringIO) The stream that is being written to by all of the nodes (created and destroyed by the ExhaleRoot object).

treeView (bool) If False, standard reStructuredText bulleted lists will be written to the `stream`. If True, then raw html unordered lists will be written to the `stream`.

lastChild (bool) When `treeView == True`, the unordered lists generated need to have an <li class="lastChild"> tag on the last child for the `collapsibleList` to work correctly. The default value of this parameter is False, and should only ever be set to True internally by recursive calls to this method.

__weakref__

list of weak references to the object (if defined)

CHAPTER 4

FAQ

Nothing is working, where did I go wrong?

Sorry to hear that. Please try comparing how your documentation is set up with the companion website.

If things look similar enough, or something isn't clear, raise an issue on GitHub. I'll do my best to support what I can, and if similar questions come up then I can add them to this FAQ.

Why does it build locally, but not on Read the Docs?

Most likely Exhale is failing to build if you are getting this.

Make sure you have the *virtualenv* functionality available on the Admin page of your website enabled, and provide a `requirements.txt` that has at the very least the line `breathe` (lower case, RTD will `pip install` every line in `requirements.txt`). Refer to the RTD docs here.

Metaprogramming and full template specialization?

Nope. Partial template specialization at best is supported by Breathe; not full template specialization. Furthermore, Doxygen can barely handle metaprogramming...YMMV.

For partial templates, see the breathe templates section for how you would specialize. My understanding is the spacing is sensitive. I have yet to be able to include any form of template specialization in breathe, though, including their example code.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

Symbols

C

D

E

F

G

I

K

P

Q

R

S

T