Sphinx AutoAPI Documentation

Release 1.8.1

Read the Docs, Inc

Apr 25, 2021

CONTENTS

CHAPTER

ONE

TUTORIALS

1.1 Setting up Automatic API Documentation Generation

This tutorial will assume that you already have a basic Sphinx project set up. If you are not sure how to do this, you can follow the Getting Started guide in the Sphinx documentation.

The recommended way of installing AutoAPI is through a virtualenv. Once you have a virtualenv set up, you can install AutoAPI with the command:

Language	Command
Python	`pip install sphinx-autoapi`
Go	`pip install sphinx-autoapi[go]`
Javascript	`pip install sphinx-autoapi`
.NET	`pip install sphinx-autoapi[dotnet]`

Depending on which language you are trying to document, each language has a different set of steps for finishing the setup of AutoAPI:

1.1.1 Python

To enable the extension, we need to add it to the list of extensions in Sphinx's `conf.py` file:

```
extensions = ['autoapi.extension']
```

For Python, there is only one required configuration option that we need to set. *`autoapi_dirs`* tells AutoAPI which directories contain the source code to document. This can either be absolute, or relative to Sphinx's conf.py file. For example, say we have a package and we have used `sphinx-quickstart` to create a Sphinx project in a `docs/` folder. The directory structure might look like this:

```
mypackage/
├── docs
│   ├── _build
```

(continues on next page)

(continued from previous page)

```
    ├── conf.py
    ├── index.rst
    ├── make.bat
    ├── Makefile
    ├── _static
    └── _templates
├── mypackage
    ├── _client.py
    ├── __init__.py
    └── _server.py
└── README.md
```

`sphinx-quickstart` sets up the `sphinx-build` to run from inside the `docs/` directory, and the source code is one level up. So the value of our *`autoapi_dirs`* option would be:

```
autoapi_dirs = ['../mypackage']
```

If you are documenting many packages, you can point AutoAPI to the directory that contains those packages. For example if our source code was inside a `src/` directory:

```
mypackage/
├── docs
    ├── _build
    ├── conf.py
    ├── index.rst
    ├── make.bat
    ├── Makefile
    ├── _static
    └── _templates
├── README.md
└── src
    └── mypackage
        ├── _client.py
        ├── __init__.py
        └── _server.py
```

We can configure *`autoapi_dirs`* to be:

```
autoapi_dirs = ['../src']
```

Now that everything is configured, AutoAPI will generate documentation when you run Sphinx!

```
cd docs/
sphinx-build -b html . _build
```

1.1.2 Go

Support for Go requires you to have the go environment installed (https://golang.org/dl/), as well as our godocjson tool:

```
go get github.com/readthedocs/godocjson
```

and the Go domain extension for Sphinx:

```
pip install sphinxcontrib-golangdomain
```

To enable the AutoAPI extension, we need to add it to the list of extensions in Sphinx's `conf.py` file with the Go domain extension:

```
extensions = [
    'sphinxcontrib.golangdomain',
    'autoapi.extension',
]
```

For Go, there are two required configuration options that we need to set. *autoapi_type* tells AutoAPI what type of language we are documenting. For Go, this is:

```
autoapi_type = 'go'
```

The second configuration option is *autoapi_dirs*, which tells AutoAPI which directories contain the source code to document. These can either be absolute, or relative to Sphinx's *conf.py* file. So if your documentation was inside a `docs/` directory and your source code is in an `example` directory one level up, you would configure *autoapi_dirs* to be:

```
autoapi_dirs = ['../example']
```

Now that everything is configured, AutoAPI will generate documentation when you run Sphinx!

```
cd docs/
sphinx-build -b html . _build
```

1.1.3 Javascript

Support for Javascript requires you to have jsdoc (http://usejsdoc.org/) installed:

```
npm install jsdoc -g
```

To enable the AutoAPI extension, we need to add it to the list of extensions in Sphinx's `conf.py` file:

```
extensions = ['autoapi.extension']
```

For Javascript, there are two required configuration options that we need to set. *autoapi_type* tells AutoAPI what type of language we are documenting. For Javascript, this is:

```
autoapi_type = 'javascript'
```

The second configuration option is *autoapi_dirs*, which tells AutoAPI which directories contain the source code to document. These can either be absolute, or relative to Sphinx's *conf.py* file. So if your documentation was inside a `docs/` directory and your source code is in an `example` directory one level up, you would configure *autoapi_dirs* to be:

```
autoapi_dirs = ['../example']
```

Now that everything is configured, AutoAPI will generate documentation when you run Sphinx!

```
cd docs/
sphinx-build -b html . _build
```

1.1.4 .NET

Support for .NET requires you to have the docfx (https://dotnet.github.io/docfx/) tool installed, as well as the .NET domain extension for Sphinx:

```
pip install sphinxcontrib-dotnetdomain
```

Firstly, we need to configure docfx to output to a directory known to AutoAPI. By default, `docfx` will output metadata files into the `_api` path. You can configure which path to output files into by setting the path in your docfx configuration file in your project repository. For example, if your `conf.py` file is located inside a `docs/` directory:

```
{
  "metadata": [{
    "dest": "docs/_api"
  }]
}
```

To enable the AutoAPI extension, we need to add it to the list of extensions in Sphinx's `conf.py` file with the .NET domain extension:

```
extensions = [
    'sphinxcontrib.dotnetdomain',
    'autoapi.extension',
]
```

For .NET, there are two required configuration options that we need to set. *`autoapi_type`* tells AutoAPI what type of language we are documenting. For .NET, this is:

```
autoapi_type = 'dotnet'
```

The second configuration option is *`autoapi_dirs`*, which tells AutoAPI which directories contain the source code to document. These can either be absolute, or relative to Sphinx's *conf.py* file. So if your documentation was inside a `docs/` directory and your source code is in an `example` directory one level up, you would configure *`autoapi_dirs`* to be:

```
autoapi_dirs = ['../example']
```

Now that everything is configured, AutoAPI will generate documentation when you run Sphinx!

```
cd docs/
sphinx-build -b html . _build
```

CHAPTER

TWO

HOW-TO GUIDES

2.1 How to Customise Layout Through Templates

You can customise the look of the documentation that AutoAPI generates by changing the Jinja2 templates that it uses. The default templates live in the `autoapi/templates` directory of the AutoAPI package. Simply copy whichever templates you want to customise to a local directory and edit them. To get AutoAPI to use these templates, point the *`autoapi_template_dir`* configuration option to your directory. It can be absolute, or relative to the root of the documentation directory (ie the directory with the `conf.py` file).

```
autoapi_template_dir = '_autoapi_templates'
```

Your template directory must to follow the same layout as the default templates. For example, to override the Python class and module templates:

```
_autoapi_templates
└── python
    ├── class.rst
    └── module.rst
```

2.2 How to Customise the Index Page

The index page that AutoAPI creates is generated using a template. So customising the index page follows the same steps as customising a template. Simply edit the `autoapi/templates/index.rst` template with the same steps as *customising a template*.

2.2.1 How to Remove the Index Page

To remove the index page altogether, turn off the *`autoapi_add_toctree_entry`* configuration option:

```
autoapi_add_toctree_entry = False
```

You will then need to include the generated documentation in the toctree yourself. For example if you were generating documentation for a package called "example", you would add the following toctree entry:

```
.. toctree::

   autoapi/example/index
```

Note that `autoapi/` is the default location of documentation, as configured by *`autoapi_root`*. If you change *`autoapi_root`*, then the entry that you need to add would change also.

2.3 How to Configure Where Documentation Appears in the TOC Tree

The *autoapi_root* configuration option defines where generated documentation is output. To change where documentation is output, simply change this option to another directory relative to the `conf.py` file:

```
autoapi_root = 'technical/api'
```

2.4 How to Transition to Autodoc-Style Documentation

Once you have written some documentation with the *Autodoc-Style Directives*, turning the automatic documentation generation off is as easy as disabling the *autoapi_generate_api_docs* configuration option:

```
autoapi_generate_api_docs = False
```

2.5 How to Transition to Manual Documentation

To start writing API documentation yourself, you can get AutoAPI to keep its generated files around as a base to start from using the *autoapi_keep_files* option:

```
autoapi_keep_files = True
```

Once you have built your documentation with this option turned on, you can disable AutoAPI altogether from your project.

2.6 How to Include Type Annotations as Types in Rendered Docstrings

Warning: This feature is experimental and may change or be removed in future versions.

Since v3.0, `sphinx` has included an `sphinx.ext.autodoc.typehints` extension that is capable of rendering type annotations as parameter types and return types.

For example the following function:

```
def _func(a: int, b: Optional[str]) -> bool
    """My function.

    :param a: The first arg.
    :param b: The second arg.

    :returns: Something.
    """
```

would be rendered as:

_func(*a*, *b*)

Parameters

- **a** (*int*) – The first arg.
- **b** (*Optional[str]*) – The second arg.

Returns Something.

Return type bool

AutoAPI is capable of the same thing. To enable this behaviour, load the `sphinx.ext.autodoc.typehints` (or `sphinx.ext.autodoc`) extension in Sphinx's `conf.py` file and set `autodoc_typehints` to `description` as normal:

```
extensions = ['sphinx.ext.autodoc', 'autoapi.extension']
autodoc_typehints = 'description'
```

Note: Unless `autodoc_typehints` is set to `none`, the type annotations of overloads will always be output in the signature and never merged into the description because it is impossible to represent all overloads as a list of parameters.

CHAPTER

THREE

CONFIGURATION OPTIONS

autoapi_dirs
Required

Paths (relative or absolute) to the source code that you wish to generate your API documentation from. The paths are searched recursively for files matching *autoapi_file_patterns*. Relative paths should be relative to the root of the documentation directory (ie the directory with the `conf.py` file).

For Python, if a package directory is specified, the package directory itself will be included in the relative path of the children. If an ordinary directory is specified, that directory will not be included in the relative path.

autoapi_type
Default: `python`

Set the type of files you are documenting. This depends on the programming language that you are using:

Language	`autoapi_type`
Python	`'python'`
Go	`'go'`
Javascript	`'javascript'`
.NET	`'dotnet'`

autoapi_template_dir
Default: `''`

A directory that has user-defined templates to override our default templates. The path can either be absolute, or relative to the root of the documentation directory (ie the directory with the `conf.py` file). An path relative to where *sphinx-build* is run is allowed for backwards compatibility only and will be removed in a future version.

You can view the default templates in the autoapi/templates directory of the package.

autoapi_file_patterns
Default: Varies by Language

A list containing the file patterns to look for when generating documentation. Patterns should be listed in order of preference. For example, if `autoapi_file_patterns` is set to the default value and a *.py* file and a *.pyi* file are found, then the *.py* will be read.

The defaults by language are:

Language	`autoapi_file_patterns`
Python	`['*.py', '*.pyi']`
Go	`['*.go']`
Javascript	`['*.js']`
.NET	`['project.json', '*.csproj', '*.vbproj']`

autoapi_generate_api_docs
Default: `True`

Whether to generate API documentation. If this is `False`, documentation should be generated though the *Directives*.

3.1 Customisation Options

autoapi_options
Default: `[ 'members', 'undoc-members', 'private-members', 'show-inheritance', 'show-module-summary', 'special-members', 'imported-members', ]`

Options for display of the generated documentation.

- `members`: Display children of an object
- `inherited-members`: Display children of an object that have been inherited from a base class.
- `undoc-members`: Display objects that have no docstring
- `private-members`: Display private objects (eg. `_foo` in Python)
- `special-members`: Display special objects (eg. `__foo__` in Python)
- `show-inheritance`: Display a list of base classes below the class signature.
- `show-inheritance-diagram`: Display an inheritance diagram in generated class documentation. It makes use of the `sphinx.ext.inheritance_diagram` extension, and requires Graphviz to be installed.
- `show-module-summary`: Whether to include autosummary directives in generated module documentation.
- `imported-members`: Display objects imported from the same top level package or module. The default module template does not include imported objects, even with this option enabled. The default package template does.

autoapi_ignore
Default: Varies By Language

A list of patterns to ignore when finding files. The defaults by language are:

Language	`autoapi_file_patterns`
Python	`['*migrations*']`
Go	`[]`
Javascript	`[]`
.NET	`['*toc.yml', '*index.yml']`

autoapi_root
Default: `autoapi`

Path to output the generated AutoAPI files into, including the generated index page. This path must be relative to the root of the documentation directory (ie the directory with the `conf.py` file). This can be used to place the generated documentation anywhere in your documentation hierarchy.

autoapi_add_toctree_entry
Default: `True`

Whether to insert the generated documentation into the TOC tree. If this is `False`, the default AutoAPI index page is not generated and you will need to include the generated documentation in a TOC tree entry yourself.

autoapi_python_class_content
Default: `class`

Which docstring to insert into the content of a class.

- `class`: Use only the class docstring.
- `both`: Use the concatentation of the class docstring and the `__init__` docstring.
- `init`: Use only the `__init__` docstring.

If the class does not have an `__init__` or the `__init__` docstring is empty and the class defines a `__new__` with a docstring, the `__new__` docstring is used instead of the `__init__` docstring.

autoapi_member_order
Default: `bysource`

The order to document members. This option can have the following values:

- `alphabetical`: Order members by their name, case sensitively.
- `bysource`: Order members by the order that they were defined in the source code.
- `groupwise`: Order members by their type then alphabetically, ordering the types as follows:
  - Submodules and subpackages
  - Attributes
  - Exceptions
  - Classes
  - Functions
  - Methods

autoapi_python_use_implicit_namespaces
Default: `False`

This changes the package detection behaviour to be compatible with **PEP 420**, but directories in *autoapi_dirs* are no longer searched recursively for packages. Instead, when this is `True`, *autoapi_dirs* should point directly to the directories of implicit namespaces and the directories of packages.

If searching is still required, this should be done manually in the `conf.py`.

autoapi_prepare_jinja_env
Default: `None`

A callback that is called shortly after the Jinja environment is created. It passed the Jinja environment for editing before template rendering begins.

The callback should have the following signature:

prepare_jinja_env (*jinja_env: jinja2.Environment*) → None

3.1.1 Events

The following events allow you to control the behaviour of AutoAPI.

autoapi-skip-member (*app*, *what*, *name*, *obj*, *skip*, *options*)
(Python only) Emitted when a template has to decide whether a member should be included in the documentation. Usually the member is skipped if a handler returns `True`, and included otherwise. Handlers should return `None` to fall back to the default skipping behaviour of AutoAPI or another attached handler.

Listing 1: Example conf.py

```
def skip_util_classes(app, what, name, obj, skip, options):
    if what == "class" and "util" in name:
        skip = True
    return skip

def setup(sphinx):
    sphinx.connect("autoapi-skip-member", skip_util_classes)
```

Parameters

- **app** – The Sphinx application object.
- **what** (*str*) – The type of the object which the docstring belongs to. This can be one of: `"attribute"`, `"class"`, `"data"`, `"exception"`, `"function"`, `"method"`, `"module"`, `"package"`.
- **name** (*str*) – The fully qualified name of the object.
- **obj** (*PythonPythonMapper*) – The object itself.
- **skip** (*bool*) – Whether AutoAPI will skip this member if the handler does not override the decision.
- **options** – The options given to the directive.

3.2 Advanced Options

autoapi_keep_files
Default: `False`

Keep the AutoAPI generated files on the filesystem after the run. Useful for debugging or transitioning to manual documentation.

Keeping files will also allow AutoAPI to use incremental builds. Providing none of the source files have changed, AutoAPI will skip parsing the source code and regenerating the API documentation.

3.3 Suppressing Warnings

suppress_warnings

This is a sphinx builtin option that enables the granular filtering of AutoAPI generated warnings.

Items in the `suppress_warnings` list are of the format `"type.subtype"` where `".subtype"` can be left out to cover all subtypes. To suppress all AutoAPI warnings add the type `"autoapi"` to the list:

```
suppress_warnings = ["autoapi"]
```

If narrower suppression is wanted, the available subtypes for AutoAPI are:

- python_import_resolution Used if resolving references to objects in an imported module failed. Potential reasons include cyclical imports and missing (parent) modules.
- not_readable Emitted if processing (opening, parsing, ...) an input file failed.
- toc_reference Used if a reference to an entry in a table of content cannot be resolved.

So if all AutoAPI warnings concerning unreadable sources and failing Python imports should be filtered, but all other warnings should not, the option would be

```
suppress_warnings = ["autoapi.python_import_resolution", "autoapi.not_readable"]
```

CHAPTER

FOUR

DIRECTIVES

4.1 Autodoc-Style Directives

You can opt to write API documentation yourself using autodoc style directives. These directives work similarly to autodoc, but docstrings are retrieved through static analysis instead of through imports.

See also:

When transitioning to autodoc-style documentation, you may want to turn the *autoapi_generate_api_docs* option off so that automatic API documentation is no longer generated.

To use these directives you will need to enable the autodoc extension in your Sphinx project's `conf.py`:

```
extensions = ['sphinx.ext.autodoc', 'autoapi.extension']
```

For Python, all directives have an autodoc equivalent and accept the same options. The following directives are available:

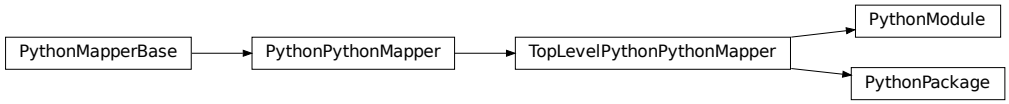
.. autoapimodule::
.. autoapiclass::
.. autoapiexception::
Equivalent to `automodule`, `autoclass`, and `autoexception` respectively. `autodoc_inherit_docstrings` does not currently work.

.. autoapifunction::
.. autoapidata::
.. autoapimethod::
.. autoapiattribute::
Equivalent to `autofunction`, `autodata`, `automethod`, and `autoattribute` respectively.

4.2 Inheritance Diagrams

.. autoapi-inheritance-diagram::
This directive uses the `sphinx.ext.inheritance_diagram` extension to create inheritance diagrams for classes.

For example:



`sphinx.ext.inheritance_diagram` makes use of the `sphinx.ext.graphviz` extension, and therefore it requires Graphviz to be installed.

The directive can be configured using the same options as `sphinx.ext.inheritance_diagram`.

CHAPTER

FIVE

TEMPLATES

A lot of the power from AutoAPI comes from templates. We are basically building a mapping from code to docs, and templates let you highly customise the display of said docs.

5.1 Structure

Every type of data structure has its own template. It uses the form *language/type*.rst to find the template to render. The full search path is:

- *language/type*.rst

So for a .NET Class, this would resolve to:

- dotnet/class.rst

We provide base/base.rst as an incredibly basic output of every object:

```
.. {language}:{type}:: {name}
```

5.2 Custom Filters, Tests, and Globals

The *autoapi_prepare_jinja_env* configuration option allows you to pass a callback that can edit the jinja2.Environment object before rendering begins. This callback, among other things, can be used to add custom filters, tests, and/or globals to the Jinja environment. For example:

```
def autoapi_prepare_jinja_env(jinja_env):
        jinja_env.filters["my_custom_filter"] = lambda value: value.upper()
```

5.3 Context

Every template is given a set context that can be accessed in the templates. This contains:

- autoapi_options: The value of the *autoapi_options* configuration option.
- include_summaries: The value of the autoapi_include_summaries configuration option.
- obj: A Python object derived from PythonMapperBase.
- sphinx_version: The contents of sphinx.version_info.

The object in `obj` has a number of standard attributes that you can reliably access per language.

Warning: These classes should not be constructed manually. They can be reliably accessed through templates only.

5.3.1 Python

class autoapi.mappers.python.objects.**PythonPythonMapper**(*obj*, *class_content='class'*, ***kwargs*)

A base class for all types of representations of Python objects.

Variables

- **name** (*str*) – The name given to this object.
- **id** (*str*) – A unique identifier for this object.
- **children** (*list (PythonPythonMapper)*) – The members of this object.

property display(*self*)

Whether this object should be displayed in documentation.

This attribute depends on the configuration options given in *autoapi_options* and the result of *autoapi-skip-member*.

Type bool

property docstring(*self*)

The docstring for this object.

If a docstring did not exist on the object, this will be the empty string.

For classes this will also depend on the *autoapi_python_class_content* option.

Type str

inherited

Whether this was inherited from an ancestor of the parent class.

Type bool

property is_private_member(*self*)

Whether this object is private (True) or not (False).

Type bool

property is_special_member(*self*)

Whether this object is a special member (True) or not (False).

Type bool

property is_undoc_member(*self*)

Whether this object has a docstring (False) or not (True).

Type bool

property summary(*self*)

The summary line of the docstring.

The summary line is the first non-empty line, as-per **PEP 257**. This will be the empty string if the object does not have a docstring.

Type str

class autoapi.mappers.python.objects.**PythonFunction**(*obj*, ***kwargs*)
Bases: *PythonPythonMapper*

The representation of a function.

property args(*self*)
The arguments to this object, formatted as a string.

Type str

overloads
The list of overloaded signatures `[(args, return_annotation), ...]` of this function.

Type list(tuple(str, str))

properties
The properties that describe what type of function this is.

Can be only be: async

Type list(str)

return_annotation
The type annotation for the return type of this function.

This will be `None` if an annotation or annotation comment was not given.

Type str or None

class autoapi.mappers.python.objects.**PythonMethod**(*obj*, ***kwargs*)
Bases: *PythonFunction*

The representation of a method.

method_type
The type of method that this object represents.

This can be one of: method, staticmethod, or classmethod.

Type str

properties
The properties that describe what type of method this is.

Can be any of: abstractmethod, async, classmethod, property, staticmethod

Type list(str)

class autoapi.mappers.python.objects.**PythonData**(*obj*, ***kwargs*)
Bases: *PythonPythonMapper*

Global, module level data.

annotation
The type annotation of this attribute.

This will be `None` if an annotation or annotation comment was not given.

Type str or None

value
The value of this attribute.

This will be `None` if the value is not constant.

Type str or None

class autoapi.mappers.python.objects.**PythonAttribute**(*obj*, ***kwargs*)
Bases: *PythonData*

An object/class level attribute.

class autoapi.mappers.python.objects.**TopLevelPythonPythonMapper**(*obj*, ***kwargs*)
Bases: *PythonPythonMapper*

A common base class for modules and packages.

all
The contents of __all__ if assigned to.

Only constants are included. This will be None if no __all__ was set.

Type list(str) or None

property classes(*self*)
All of the member classes.

Type list(*PythonClass*)

property functions(*self*)
All of the member functions.

Type list(*PythonFunction*)

top_level_object
Whether this object is at the very top level (True) or not (False).

This will be False for subpackages and submodules.

Type bool

class autoapi.mappers.python.objects.**PythonModule**(*obj*, ***kwargs*)
Bases: *TopLevelPythonPythonMapper*

The representation of a module.

class autoapi.mappers.python.objects.**PythonPackage**(*obj*, ***kwargs*)
Bases: *TopLevelPythonPythonMapper*

The representation of a package.

class autoapi.mappers.python.objects.**PythonClass**(*obj*, ***kwargs*)
Bases: *PythonPythonMapper*

The representation of a class.

property args(*self*)
The arguments to this object, formatted as a string.

Type str

bases
The fully qualified names of all base classes.

Type list(str)

property docstring(*self*)
The docstring for this object.

If a docstring did not exist on the object, this will be the empty string.

For classes this will also depend on the *autoapi_python_class_content* option.

Type str

class autoapi.mappers.python.objects.**PythonException**(*obj*, ***kwargs*)
Bases: *PythonClass*

The representation of an exception class.

5.3.2 Go

class autoapi.mappers.go.**GoPythonMapper**(*obj*, ***kwargs*)
Base object for JSON -> Python object mapping.

Subclasses of this object will handle their language specific JSON input, and map that onto this standard Python object. Subclasses may also include language-specific attributes on this object.

Arguments:

Parameters

- **obj** – JSON object representing this object
- **jinja_env** – A template environment for rendering this object

Required attributes:

Variables

- **id** (*str*) – A globally unique indentifier for this object. Generally a fully qualified name, including namespace.
- **name** (*str*) – A short "display friendly" name for this object.

Optional attributes:

Variables

- ***docstring*** (*str*) – The documentation for this object
- **imports** (*list*) – Imports in this object
- **children** (*list*) – Children of this object
- **parameters** (*list*) – Parameters to this object
- **methods** (*list*) – Methods on this object

property short_name(*self*)
Shorten name property

5.3.3 Javascript

class autoapi.mappers.javascript.**JavaScriptPythonMapper**(*obj*, ***kwargs*)
Base object for JSON -> Python object mapping.

Subclasses of this object will handle their language specific JSON input, and map that onto this standard Python object. Subclasses may also include language-specific attributes on this object.

Arguments:

Parameters

- **obj** – JSON object representing this object
- **jinja_env** – A template environment for rendering this object

Required attributes:

Variables

- **id** (*str*) – A globally unique indentifier for this object. Generally a fully qualified name, including namespace.
- **name** (*str*) – A short "display friendly" name for this object.

Optional attributes:

Variables

- ***docstring*** (*str*) – The documentation for this object
- **imports** (*list*) – Imports in this object
- **children** (*list*) – Children of this object
- **parameters** (*list*) – Parameters to this object
- **methods** (*list*) – Methods on this object

5.3.4 .NET

class autoapi.mappers.dotnet.**DotNetPythonMapper**(*obj*, ***kwargs*)

Base .NET object representation

Parameters **references** (*list of dict objects*) – object reference list from docfx

property pathname(*self*)

Sluggified path for filenames

Slugs to a filename using the follow steps

- Decode unicode to approximate ascii
- Remove existing hypens
- Substitute hyphens for non-word characters
- Break up the string as paths

property ref_name(*self*)

Return object name suitable for use in references

Escapes several known strings that cause problems, including the following reference syntax:

```
:dotnet:cls:`Foo.Bar<T>`
```

As the *<T>* notation is also special syntax in references, indicating the reference to Foo.Bar should be named T.

See: http://sphinx-doc.org/domains.html#role-cpp:any

property ref_short_name(*self*)

Same as above, return the truncated name instead

resolve_spec_identifier(*self*, *obj_name*)

Find reference name based on spec identifier

Spec identifiers are used in parameter and return type definitions, but should be a user-friendly version instead. Use docfx `references` lookup mapping for resolution.

If the spec identifier reference has a `spec.csharp` key, this implies a compound reference that should be linked in a special way. Resolve to a nested reference, with the corrected nodes.

Note: This uses a special format that is interpreted by the domain for parameter type and return type fields.

Parameters **obj_name** – spec identifier to resolve to a correct reference

Returns resolved string with one or more references

Return type str

property short_name(*self*)
Shorten name property

static transform_doc_comments(*text*)
Parse XML content for references and other syntax.

This avoids an LXML dependency, we only need to parse out a small subset of elements here. Iterate over string to reduce regex pattern complexity and make substitutions easier

See also:

Doc comment reference <https://msdn.microsoft.com/en-us/library/5ast78ax.aspx> Reference on XML documentation comment syntax

CHAPTER

SIX

DESIGN REFERENCE

6.1 Python

When choosing what to document, AutoAPI aims to document anything that is publicly accessible through the actual package when loaded in Python. For example if a function is imported from a submodule into a package, that function is documented in both the submodule and the package. There are some exceptions to this rule:

- Anything that is imported into a module is not documented. Usually a module is where implementations exist. Therefore an import of something is usually for the usage of the implementation, and not as something to be accessed publicly.
- When the module or package defines an `__all__`, only the members named in `__all__` are documented.
- When a configuration option indicates that private or special members should also be documented.

Furthermore, AutoAPI follows the same docstring inheritance rules as `inspect.getdoc()`, with some exceptions:

- The docstrings of the following methods are not inherited because they are usually redundant:
  - `object.__init__()`
  - `object.__new__()`
  - `type.__init__()`
  - `type.__new__()`

6.2 .NET

This document talks about the design of a .NET Sphinx integration. This will include a mechanism for generating Javadoc style API references automatically. We will describe decisions that lead to specific implementation details.

6.3 Goals

The main goal of this project is to be able to generate a MSDN or Javadoc style API reference from a .Net project in Sphinx.

6.3.1 Primary Goals

- Build MSDN/Javadoc style HTML output for arbitrary .Net code.
- Have specific pages for each Package, Class, and all class-level structures.
  - */api/System/*
  - */api/System/String/*
  - */api/System/String/Constructors/*

6.3.2 Secondary Goals

- Allow for definition of .Net classes inside of normal Sphinx prose docs (classic Sphinx style definition).
- Allow generation of Javadoc style docs from other languages.

6.3.3 Requirements

6.4 Introduction

We are working with Sphinx, which has an existing way of doing this. Generally, you define a *Domain* which describes the various language structure, a *Class* or *Method*, for example. Then the user will write RST that uses these definitions, and Sphinx will create output from that markup.

```
.. py:function:: spam(eggs)

   Spam the foo.
```

The author of the doucmentation will have now told Sphinx that the *spam* function exists in the Python project that is being documented.

6.4.1 Autogenerated Output

Sphinx then built a series of tools to make the generation of this markup easier and more automatic:

- Autodoc
- Autosummary

Autodoc is a Python-only solution that imports the author's code into memory, and then allows the author to more automatically document full objects. For example, you can document a whole class on a page.

```
.. autoclass:: Noodle
```

This will generate output that looks like:

class Noodle
Noodle's docstring.

There are also options for it to include a full listing of the classes attributes, methods, and other things, automatically.

Warning: Remember, this depends on `Noodle` being importable by the Python interpreter running Sphinx.

6.5 Proposed Architecture

The proposed architecture for this project is as follows:

- A program that will generate a YAML (or JSON) file from a .Net project, representing it's full API information.
- Read the YAML and generate an appropriate tree structure that will the outputted HTML will look like (YAML-Tree)
  - If time allows, we will allow a merging of these objects with multiple YAML files to allow for prose content to be injected into the output
- Take the YAML structure and generate in-memory rst that corresponds to the Sphinx dotnet Domain objects
- dotnet Domain will output HTML based on the doctree generated from the in-memory RST

In diagram form:

```
Code -> YAML -> YAMLTree -> RST (Dotnet Domain) -> Sphinx -> HTML
```

6.5.1 YAMLTree

One of the main problems is how to actually structure the outputted HTML pages. The YAML file will likely be ordered, but we need to have a place to define the page structure in the HTML.

This can be done before or after the loading of the content into RST. We decided to do it before loading into RST because that matches standard Sphinx convention. Generally the markup being fed in as RST is considered to be in a file that maps to it's output loation. If we tried to maniuplate this structure after loading into the Domain, that could lead to unexpected consequences like wrong indexes and missing references.

6.5.2 File Structure vs. Hierarchy

Specific ID's should have one specific detail representation. This means that every YAML docid object should only have one place that it is rendered with a `.. dn:<method>` canonical identifier. All other places it is referenced should be in either:

- A reference
- A toctree (listing)

6.5.3 Sphinx Implementation

The user will run a normal *make html* as part of the experience. The generation and loading will be done as an extension that can be configured.

There will be Sphinx configuration for how things get built:

```
autoapi_root = 'api' # Where HTML is generated
autoapi_dirs = ['yaml'] # Directory of YAML sources
```

We will then loop over all YAML files in the `autoapi_dir` and parse them. They will then be output into `autoapi_root` inside the documentation.

6.6 Examples

A nice example of Sphinx Python output similar to what we want:

- http://dta.googlecode.com/git/doc/_build/html/index.html
- Src: https://raw.githubusercontent.com/sfcta/dta/master/doc/index.rst

An example domain for Spec:

- https://subversion.xray.aps.anl.gov/bcdaext/specdomain/trunk/src/specdomain/sphinxcontrib/specdomain.py

6.7 Other Ideas

> **Warning:** Things in this section might not get implemented.

The .Net domain will not be able to depend on importing code from the users code base. We might be able to implement similar authoring tools with the YAML file. We might be able to output the YAML subtree of an object with autodoc style tools:

```
.. autodnclass:: System.String
   :members:
```

CHAPTER

SEVEN

RELEASE PROCESS

This page documents the steps to be taken to release a new version of Sphinx AutoAPI.

7.1 Pre-Checks

1. Check that the dependencies of the package are correct.
2. Clean the `.tox` directory and run the tests.
3. Commit and push any changes needed to make the tests pass.
4. Check that the tests passed on github.

7.2 Preparation

1. Update the version numbers in `autoapi/__init__.py`.
2. Add any missing changelog entries.
3. Put the version number and release date into the changelog.
4. Commit and push the changes.
5. Check that the tests passed on github.

7.3 Release

```
git clean -fdx
python setup.py sdist bdist_wheel
twine upload dist/*
git tag vX.X.X
git push --tags
```

INDEX