Codocpy Documentation

Codoc

Jul 04, 2021

CONTENTS

Codoc is the first Continuous Documentation tool. It provides a revolutionary graph based interface, that gives a powerful overview of any software system.

For more information about codoc, please visit our website.

Codocpy is used to publish your graphical documentation & architectural views from a command-line or CI solution, by writing simple Python scripts.

> **Warning:** Codoc is still very early beta, and incomplete. We are currently looking for beta testers.

CHAPTER

ONE

A QUICK EXAMPLE

```
@view(
    label="Module View",
)
def view_modules(graph):
    """
    This view contains all the modules that our system contain.
    """
    return filters.exclude_classes(graph)
```

CHAPTER

TWO

FEATURES

- Always-up-to-date architectural views
- A simple framework integrated in your favorite language
- Variety of filters to show only the relevant information
- Historical information about prior views
- **COMING SOON** See graphical representation of test coverage, contributors etc.
- **COMING SOON** Get live monitoring data on your views
- **COMING SOON** Comments and dialogue about views.
- **COMING SOON** A git bot that provide context for pull requests
- **COMING SOON** A variety of export possibilities
- **COMING SOON** Sphinx, Confluence, GitBook (and other) integrations

CHAPTER

THREE

CONTENT

3.1 Getting started

This guide will go through setting up Codoc with Python, creating a config and a few simple architectural views with the supplied framework. Finally you will publish them, and see the diagrams of your system. Very neat!

Don't know what views are? A *view*, or *architectural view* is, according to opengroup.org is:

> Architecture views are representations of the overall architecture that are meaningful to one or more stakeholders in the system. The architect chooses and develops a set of views that will enable the architecture to be communicated to, and understood by, all the stakeholders, and enable them to verify that the system will address their concerns.

We have an indepth motivation and explanation on https://codoc.org - it also has examples!

`codocpy` requires: Python 3.6, 3.7, 3.8, 3.9.

3.1.1 Install `codoc-python`

1. Install the package by running:

```
pip install codoc-python
```

2. Check that it's installed correctly:

```
$ codocpy
```

3.1.2 Create a config

Everything Codoc related should be located inside a folder called `codoc_views` located at the root directory of your project.

Start by creating a configuration file:

You will also need a basic config file in the same folder, called `config.py`. This file mainly needs a function called `setup` to return a graph of the system in question. The function takes `**kwargs`, to pass along any flags. The example below returns a graph containing the `myproject` module, and it's direct dependencies - please replace `myproject` with the module you want to document:

> **Warning:** Using django? Please see *Django* to bootstrap that correctly. Please see *Multiple modules* if your code exposes multiple packages.

```
# codoc_views/config.py
from codoc import new_graph

import myproject

def setup(**kwargs):
    return new_graph(myproject, **kwargs)
```

3.1.3 Your first *view function*

You'll be creating what we call a *view function* now. This is a function that takes, as input, a graph that details the whole python codebase, and as output returns a new graph. This makes it possible for you to

Inside the `codoc_views` folder, create a new python file, the name of which can be anything you choose. This file will include your first *view function*, which generates a view of the modules of your system.

```
# codoc_views/module_views.py
from codoc import filters, view
@view(
    label="Module View",
)
def modules(graph):
    """
    This view contains all the modules that your project contains.
    """
    return filters.include_only_modules(graph)
```

You can verify that codoc can find your views:

```
$ codocpy list_views
- module_views.modules
```

> **Warning:** Please make sure you are in the root directory of the project.

This should be your filename appended with the name of each view function.

3.1.4 Publishing your view

> **Warning:** Codoc will load all your code, and by effect execute all side-effects! Make sure you don't have files that execute critical code on import! see *Dangerous side effects!* for more info.

By now we hope you are already signed up and a registered user.

You'll have to fetch the API key for the project you are currently working on.

Go to your codoc project and scroll to the bottom and fetch your API key of choice.

This has to be set as an environmental variable called `CODOC_API_KEY`. One way of doing is simply by writing:

```
$ export CODOC_API_KEY=f5f9c07f4ce96aeee3aeb32faf35c0e821b8c831
```

You can now publish your views:

```
$ codocpy publish
Publishing Module View...
published at https://codoc.org/app/view/181
```

Note: Did it failed? Codoc is a bit sensitive, sadly. Read *It crashed!* for what to do.

Your view is now published, and you can view it at the URL shown in your console (in our example https://codoc.org/app/graph/181) which offers a public example from our sample project

3.1.5 Your second *view function*

This prior view might be very verbose, depending on the system you have. It also shows all external dependencies too, which might not be ideal.

If you feel confident and want to play around, you can look at either *Examples of view functions* for examples of views we created or *Filters* for a complete lists of possible views.

Otherwise read on! We will go into how you can use these filters for more complex needs.

As mentioned, filters are simply functions that remove nodes from your graph, however by combining them one can express rather complex needs.

For instance by chaining them (i.e using one on the result of another) one can use the possibilities of both. The following examples uses a `depth_based_filter` to only get the top modules and any direct content of those.

Any important thing to note is that the function has a different name. Otherwise one would override the other.

```
# codoc_views/module_views.py
from codoc import filters, view

@view(
    label="Top level Module View",
)
def top_level_modules(graph):
    """
    This view contains all the modules that your project contains.
    """
    graph = filters.include_only_modules(graph)
    # we only want the outer most modules and their direct content
    depth_based_filter = filters.get_depth_based_filter(2)
    return depth_based_filter(graph)
```

If you run `codocpy publish` again, you'll see two views being generated, and if you click on the new one, you'll see a simpler graph.

Another great filter is the `get_children_of`, which makes the graph "zoom in" on a subsection (subgraph) of the graph/system. So if you are analyzing a project called `myproject` but only want to view the content of a submodule, i.e `myproject.submodule` the following view would help:

```
# codoc_views/module_views.py
from codoc import filters, view
import myproject.submodule

@view(
    label="Content of Submodule",
)
```

(continues on next page)

(continued from previous page)

```
def content_of_submodule(graph):
    return filters.get_children_of(myproject.submodule)(graph)
```

You could also use the | (OR) operator to get the union of two graphs, i.e both modules AND classes. We increase depth here, to make sure we get more content.

```
# codoc_views/module_views.py
from codoc import filters, view
import myproject.submodule

@view(
    label="Classes & Module View",
)
def modules_and_classes(graph):
    graph = (
            filters.include_only_modules(graph)
            | filters.include_only_classes(graph)
    )
    return filters.get_children_of(myproject.submodule)(graph)
```

Want more? There are a bunch of examples and reference documentation etc that you can consult. I hope it made sense - otherwise please contact us.

See also:

- *Examples of view functions*
- *How does codocpy work?*
- *Filters*
- *Views*
- *Configuration*

3.2 Examples of view functions

This file contains examples of different views, explain what they do and why you might want them. They are merely examples and might need tweaking to work within your project

3.2.1 Top level modules view

This view shows all the top level modules and their direct descendants (content).

```
# codoc_views/module_views.py
from codoc import filters, view

@view(
    label="Top level Module View",
)
def top_level_modules(graph):
    """
    This view contains all the modules that your project contains.
    """
    graph = filters.include_only_modules(graph)
```

(continues on next page)

(continued from previous page)

```
    # we only want the outer most modules and their direct content
    depth_based_filter = filters.get_depth_based_filter(2)
    return depth_based_filter(graph)
```

3.2.2 Domain Model

Model diagram, class diagram or something else. People call it different things.

To quote Cosmic Python, a highly recommended book:

> The domain is a fancy way of saying the problem you're trying to solve. Your authors currently work for an online retailer of furniture. Depending on which system you're talking about, the domain might be purchasing and procurement, or product design, or logistics and delivery. Most programmers spend their days trying to improve or automate business processes; the domain is the set of activities that those processes support.

The following view function will show all the classes of your domain model, given that it's defined in `myproject.domain`.

```
from codoc import filters, view

import myproject


@view(
    label="Domain model (Classes)",
)
def domain_model(graph):
    graph = filters.get_children_of(myproject.domain, keep_external_
↪nodes=False)(graph)

    return filters.include_only_classes(graph)
```

3.2.3 Django models

In django you define your dataclasses by inheriting the *Model* class. Our filters currently do not support filtering based on inheritance, however by concatinating

```
from codoc import filters, view

import accounts.models
import billing.models

@view(
    label="Domain Model (API)",
)
def all_models(graph):
    graph = filters.include_only_classes(graph) | filters.include_only_modules(graph)

    return (
        filters.get_children_of(accounts.models)(graph)
        | filters.get_children_of(billing.models)(graph)
    )
```

3.2.4 Clean Django module diagram

Django includes a few files that you might not be that interested in, like the `migrations` file or `tests` of each app. We personally like to get a bit cleaner diagrams. We also don't always need external dependencies. The following custom made filter can help you here, and is utilized in the below example:

```
from codoc import view, filters

import accounts, billing

@view(
    label="Internal modules",
)
def internal_modules(graph):
    return remove_django_bloat(
        remove_external_nodes(
            filters.include_only_modules(graph)
        )
    )

def remove_external_nodes(graph):
    return (
        filters.get_children_of(accounts)(graph)
        filters.get_children_of(accounts)(graph)
        | filters.get_children_of(billing)(graph)
    )

def remove_django_bloat(graph):
    graph = filters.exclude_by_regex(r".migration")(graph)
    graph = filters.exclude_by_regex(r".test")(graph)
    graph = filters.exclude_by_regex(r".apps")(graph)
    graph = filters.exclude_by_regex(r".snapshots")(graph)

    return graph
```

See also:

- *Filters*
- *How does codocpy work?*

3.3 Reference guides

Here we provide a complete list of the API features that can be used when writing views for Codoc.

3.3.1 Filters

list of all filters

Filters can be used to filter a graph.

This is used to create a more special and specific view, making it more viewer friendly.

exclude_classes(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*
Returns a graph that doesn't have any classes

exclude_functions(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*
Returns a graph that doesn't have any functions

exclude_modules(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*
Returns a graph that doesn't have any modules

exclude_exceptions(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*
Returns a graph that doesn't have any exceptions

include_only_classes(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*
Returns a graph that only has classes

include_only_functions(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*
Returns a graph that only has functions

include_only_modules(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*
Returns a graph that only has modules

include_only_exceptions(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*
Returns a graph that only has exceptions

content_of(*node: Union[str, object,* codoc.domain.model.Node*]*, *keep_external_nodes: bool = False*, *keep_parents: bool = False*) → Callable[[*codoc.domain.model.Graph*], *codoc.domain.model.Graph*]

Parameters

- **node** – The node, object or string identifier of what to filter based on
- **keep_external_nodes** – Whether to keep external dependencies of children

Returns A filter function that excludes non-children

Return type GraphFilter

Returns a filter that only returns children of the node given via the identifier.

The returned filter (function) can then be called with a given graph.

Important: Children are **NOT** dependencies, they are things defined inside the current node. I.e if a class, Foo, defined in FooModule, then FooModule is the parent of Foo.

Example that returns all modules/classes/exceptions/functions defined inside *myproject.subproject*. .. code-block:: python

filter_function = filters.content_of(myproject.subproject)

filtered_graph = filter_function(graph)

get_depth_based_filter(*depth: int*) → Callable[[*codoc.domain.model.Graph*], *codoc.domain.model.Graph*]

class_diagram_filter(*graph:* codoc.domain.model.Graph) → *codoc.domain.model.Graph*

Parameters **graph** – Graph to filter

Returns A graph that only contains classes and methods (functions inside classes)

The Class Diagram filter is useful if you want a traditional class diagram, as it will remove all functions (which aren't inside classes - functions inside classes, are often called methods).

filter_by_regex(*pattern: str*, *flags=0*) → Callable[[*codoc.domain.model.Graph*], *codoc.domain.model.Graph*]
This allows you to filter nodes based on whether they fulfill some regex query. This is ideal if you, for instance, want to remove all test related things.

The regex is done solely on the *name* attribute.

The following example removes all instances that don't include "test". Example:

```
graph = filters.filter_by_regex("test", flags=re.IGNORECASE)(graph)
```

To understand how to use regex, please consult the python documentation:

https://docs.python.org/3/library/re.html

exclude_by_regex(*pattern: str, flags=0*) → Callable[[*codoc.domain.model.Graph*], *codoc.domain.model.Graph*]

This allows you to filter nodes based on whether they fulfill some regex query. This is ideal if you, for instance, want to remove all test related things.

The regex is done solely on the *name* attribute.

The following example removes all instances with a name that contains "test". Example:

```
graph = filters.exclude_by_regex("test", flags=re.IGNORECASE)(graph)
```

To understand how to use regex, please consult the python documentation:

https://docs.python.org/3/library/re.html

Customization

Filters are, from a implementation perspective, simply a function that takes a graph and returns a new graph, making it very easy to implement custom filters.

An example where one creates a filter that removes all edges:

```
def remove_edges(graph):
     return Graph(
             nodes=graph.nodes,
             edges=set()
     )
```

We recommend you put all your custom filters in your *codoc_views* folder, in a file called *custom_filters* or similar, however this is completly optional.

3.3.2 Views

A view is a graph with a label and a description, which can be viewed in our web app

They can compared to Architectural views, however we aim to make them interactive and contain more and better information.

The cool thing about having architectural views in your CI pipeline, is that you have historical information, and that they are constantly up to date.

This is what you can use the codoc app for. You can see all prior versions, and how your system evolves from a structural perspective. It's super cool in our humble opinion.

In our framework, we define a view as consisting of:

- A graph
- A label
- A unique id
- A project that (which *owns* the view)

- An optional commit hash

View Functions

Views functions are functions that are used to generate views.

They should reside in files prefixed with *views_* and reside inside a *codoc_views* folder in your root directory.

A simple example can be found in *Your first view function*.

A view is simply a function that takes a graph as input and returns the graph that you want to view. Here you add a label and a description and any related data that is relevant to you. The label is supplied in the *view* decorator, and the description is simply the docstring of the view function.

The label you supply to the decorator is the one that will be visualized on the webapp.

Advanced Usage

By default the view

Dynamic Descriptions

The *view* function can also take a string as input if you want a dynamic description, i.e based on the docstring of something else. This can make documentation more DRY:

```
from codoc.service import filters
from codoc.service.export import view
from codoc.service.parsing.node import get_description

import myproject

@view(
    label="Module View",
    description=get_description(myproject)
)
def view_modules(graph):
    return filters.exclude_functions(filters.exclude_classes(graph)
```

Custom ID

The way we identify whether two views are the same, but different versions, is by using a unique graph_id. We generate this based on the function and file name, which is a combination we hope is unique. You can, however, set this yourself, if you have concrete reasons for it. Don't do this if you don't know what you are doing, because it my create collisions, and make your documentation unusable.

```
from codoc.service import filters
from codoc.service.export import view

import myproject

@view(
    label="Module View",
    graph_id="my_very_long_and_unique_id"
```

(continues on next page)

(continued from previous page)

```
)
def view_modules(graph):
    return filters.exclude_functions(filters.exclude_classes(graph)
```

3.3.3 Configuration

Some times you want more control of your view generation - maybe you want to apply a filter on *all* views for what ever reason, or maybe you want to add annotations to each graph.

This is all possible in the configuration file.

file

The file needs to be located in your *codoc_views* folder and be called *codoconf.py*. The cool thing is that it is an executable python file, making it easy to write custom setup functions based on your environment.

Setup

The graph given to each view function is generated with the `setup` function.

One can utilize the simple version in *Publishing your view*, however more advanced versions could be by utilizing *Filters*, i.e:

```
# codoc_views/config.py
from codoc import new_graph, filters

import myproject

def setup(**kwargs):
    graph = new_graph(myproject)
    return filters.exclude_functions(graph, **kwargs)
```

However the function exposes a variety of other possibilities too.

Prepping your environment

One neat reason to use the *Setup* function, is that you can use it to prepare your environment. If you are using a framework of sorts, there might be a need to bootstrap your code before it can run.

Python dotenv

We personally like python-dotenv, and it can easily be used for, for instance, your CODOC API key. Simply add it like so:

```
# codoc_views/config.py
from codoc import new_graph
from dotenv import load_dotenv

import myproject

```

(continues on next page)

(continued from previous page)

```
def setup(**kwargs):
    load_dotenv()
    return new_graph(myproject, **kwargs)
```

Multiple modules

Some Python codebases exposes multiple packages. If this is the case, then you need to generate graphs for all of these too. Luckily you can use a variety of binary operators to group graphs together. Using the OR (|) operation you can get nodes that exist in either of two graphs. The following configuration file does precisely this to include tests as well as dependencies of the views themselves:

```
# codoc_views/config.py
from codoc import new_graph

import myproject, tests, codoc_views

def setup(**kwargs):
    return (
        new_graph(sample, **kwargs)
        | new_graph(tests, **kwargs)
        | new_graph(codoc_views, **kwargs)
    )
```

Django

Django needs you to bootstrap and import settings prior to importing any modules.

The following configuration does this, and creates a graph for two different django apps (Which is what they name their modules). Replace `app_one` and `app_two` with the modules of your system, and add more if applicable.

```
# codoc_views/config.py
from codoc import new_graph
import os

def setup(**kwargs):
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "codoc_api.settings")
    import django
    django.setup()

    import app_one, app_two
    return (
        new_graph(app_one, **kwargs) |
        new_graph(app_two, **kwargs)
    )
```

3.3.4 Domain

Domain Model

The domain model is a conceptual model of the domain that incorporates both behavior and data.

We use it to define the core data classes that are used throughout the project.

Graph

class Graph(*edges: Set[*codoc.domain.model.Dependency*]*, *nodes: Set[*codoc.domain.model.Node*]*)
A Graph is the base element of the system. It contains both edges (Dependencies) as well as nodes (classes, functions, etc).

It supports a variety of operators.

edges: **Set[*codoc.domain.model.Dependency*]**

nodes: **Set[*codoc.domain.model.Node*]**

Node

class Node(*identifier: str*, *name: str*, *of_type:* codoc.domain.model.NodeType, *path: Optional[str]*, *args: Optional[Tuple[str, ...]]*, *lines: Optional[Tuple[int, int]]*, *external: bool = True*, *description: Optional[str] = None*, *parent_identifier: Optional[str] = None*)
Nodes represents a given source code item, i.e a class, function or module.

It contains all the meta data as well as the code that defined the node in question.

Node Type

class NodeType(*value*)
An enumeration.

Dependency

class Dependency(*from_node: str*, *to_node: str*)
A Dependency shows that one node depends on another. Currently it doesn't specify the type of dependency.

3.4 Frequently Asked Questions

3.4.1 How does codocpy work?

You might be wondering how it all works.

codocpy utilizes Dynamic Analysis to examine your code.

This is done because we get a greater insight into exactly how and what your system does in the current environment with the external dependencies you have. This provides a few cool features, one of which making it easy to fully understand all dependencies, but also understand code with unexpected side effects. In the future, it will also make it possible to include information regarding the path your code takes when running tests.

3.4.2 Dangerous side effects!

Codocpy relies on dynamic analysis (see *How does codocpy work?*), which is both good and bad. We strongly advise that you don't have any production api keys or anything set up, in the environment you run codoc in. Codoc is much like automated tests. If your automated tests execute code that sends emails, then codoc might do it too. It's a bit different, but codoc will import all your files into memory, and if your code is written improperly, then that means side effects.

This can happen if codocpy imports a file that doesn't define a __main__ function correctly, and then either exits or something similar. Essentially, if you have a python file that executes python code on import, then don't run codoc. Rewrite your code. It's bad practice.

TL;DR Do this ~~~~~~~~~~~~~~~~~~~~~~~~

```
# scripts/myfile.py
if __name__ == "__main__":
    users = get_users()
    for user in users
        send_spam_email(user)
```

Not this

Warning: DONT DO THE FOLLOWING

```
# scripts/myfile.py
users = get_users()
for user in users
    send_spam_email(user)
```

3.4.3 It crashed!

This might be due to the quality of your code, and I mean that in the nicest way possible.

Codocpy relies on dynamic analysis (see *How does codocpy work?*), which means that if your code crashes, then codoc crashes. There can be a bunch of different reasons. We recommend you read *Prepping your environment* and make sure it is set up correctly.

You can run codocpy with the `raise_errors` for more information if the error message isn't helpful. (`codocpy publish --raise_errors`).

Another possible problem is side-effects in your codebase. See *Dangerous side effects!*.

If you have circular dependencies, that will make codocpy crash some times, due to python crashing.

If you are using Django, then it might be due to a known bug with `admin.py`.

We try our best at providing meaningful messages, where possible, however, it might be difficult at times. Codoc is a sensitive framework, but it will help you forever if you treat it right.

3.4.4 Is it secure?

You might fear losing your data. You shouldn't! Codoc doesn't access data and only reads your source code. It also only exports what you want, and you can always delete your data again. We, however, do not currently offer any self-hosted solutions, so if you want total control over your data, you are out of luck. Please contact us, if this is a big issue for you, and we might be able to help, and/or prioritize it higher.

CHAPTER

FOUR

BUGS/REQUESTS

Please use the GitHub issue tracker to submit bugs or request features.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX