cibyl

unknown

Sep 14, 2022

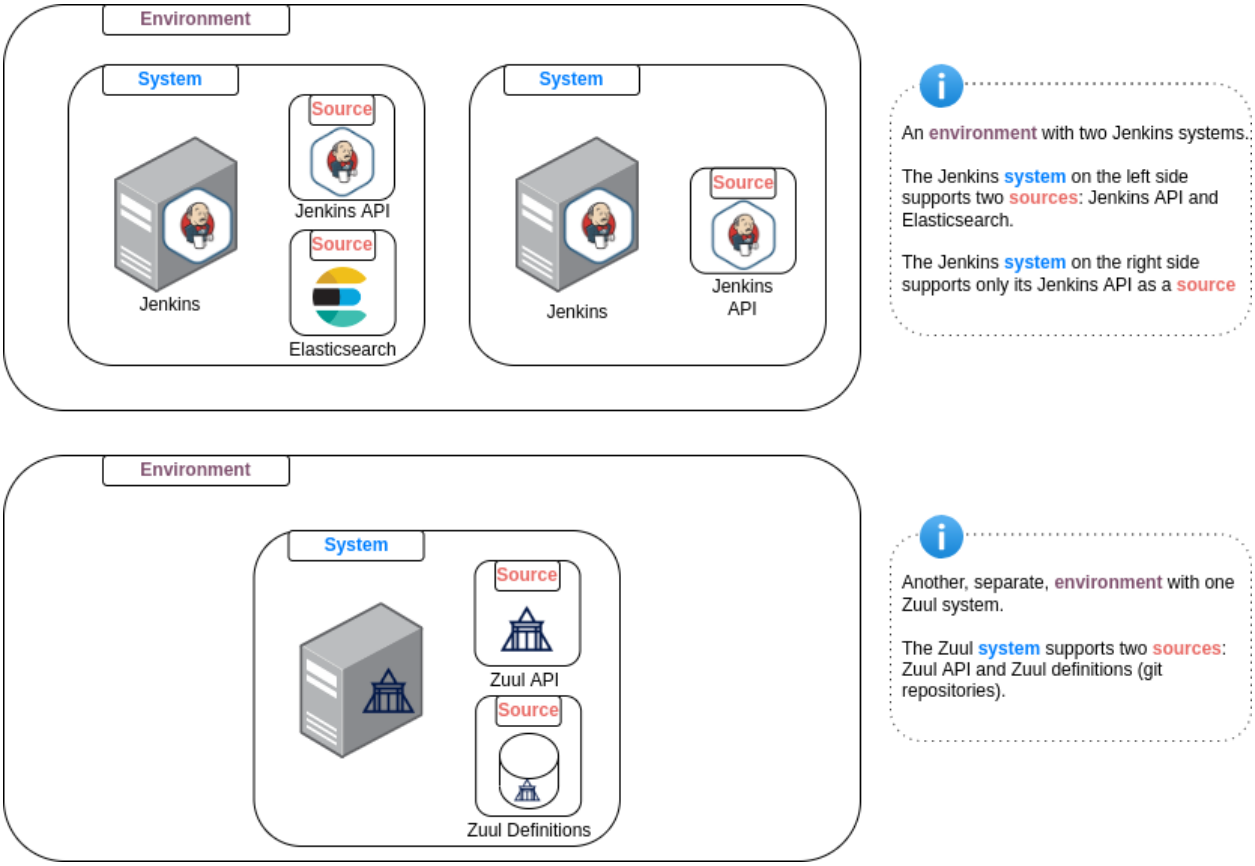
QUICKSTART

Cibyl is a command-line interface and REST API for querying CI environments and systems.

It supports out-of-the-box the following CI systems:

- Jenkins
- Zuul

Cibyl allows you to configure multiple environments, where each environment contains one or more CI/CD system and each CI/CD system can be queried using different types of source



The project originated from Red Hat OpenStack DevOps team that looked for a solution to provide a powerful and flexible way for inspecting multiple different CI environments and systems, with regards to product aspects.

The name Cibyl, a form of Sybil, derived from the Greek sybilla or sibilla. Like a prophetess, Cibyl delves into the depths of CI systems for "hidden" info revelation. CI-byl is also a wordplay that reflects the relation to CI.

CHAPTER

ONE

INDEX

1.1 Bootstrap

1.1.1 Installation

Install *cibyl* from GitHub (Recommended):

```
pip install 'git+https://github.com/rhos-infra/cibyl.git'
```

1.1.2 Configuration

In order to use Cibyl's CLI, you should set up configuration first in `~/.config/cibyl.yaml`.

Configuration is structured as follows

```
# Minimal configuration

environments:                 # List of CI/CD environments
  production:                 # An environment called "production"
    production_jenkins:       # A single system called "production_jenkins"
      system_type: jenkins    # The type of the system (jenkins or zuul)
      sources:                # List of sources belong to "production_jenkins" system
        jenkins_api:          # The name of the source which belongs to "production_
↪jenkins" system
          driver: jenkins     # The driver the source will be using
          url: https://...    # The URL of the system
          username: user      # The username to use for the authentication
          token: xyz          # The token to use for the authentication
          cert: False         # Disable/Enable certificates to use for the authentication

plugins:                      # (Optional) Specify the plugins to enable when running␣
↪Cibyl
  - openstack                 # OpenStack adds its own product related models and␣
↪arguments
```

Note:

Red Hat OpenStack user? use the following command to set up the configuration:
wget https://url.corp.redhat.com/cibyl-config -O ~/.config/cibyl.yaml

For more information on how to set up the configuration, read the configuration section.

1.1.3 Usage - CLI

Once you've installed Cibyl and set up the configuration, you can start running cibyl commands

`cibyl query --jobs` will print all the jobs from each specified system in the configuration

To get an idea of what type of commands you can use with Cibyl, run `cibyl -h`

To get an idea of what type of information you query for with Cibyl, run `cibyl query -h`

For a more in depth guide on how to use Cibyl, read the CLI usage section.

1.2 Usage

1.2.1 CLI

Basic

Running `cibyl` with no arguments, will print the environments and systems as set up in your configuration. Cibyl supports multiple subcommands. The most common one is query, that allows to query many environments for different CI/CD and product specific data. Another example is the `features` subcommand, which allows to query whether certain product-specific features are supported in each of the environments of the configuration (see features section for more details).

Jobs

Running `cibyl query --jobs` will retrieve information on all the jobs for each environment specified in your configuration.

In order to retrieve jobs for a specific environment, use the `--envs Environment` argument. If the environment includes multiple systems, you can also choose a specific system with the `--systems System` argument. The same can be done for sources with the `--sources Source` argument.

This was a simple example of what you could do with cibyl, to get a more depth overview see the CLI usage section.

1.2.2 Python

To Do

1.3 Installation

Install *cibyl* from GitHub (Recommended):

```
pip install 'git+https://github.com/rhos-infra/cibyl.git'
```

To obtain latest stable released version of Cibyl, install it from PyPi:

```
pip install cibyl
```

Warning: Using Cibyl from virtualenv might not work as expected if certifications are required to connect the CI system(s)

Note: For development purposes, it's recommended to use `pip install -e 'git+https://github.com/rhos-infra/cibyl.git'`

1.3.1 Configuration

In order to use Cibyl's CLI, you should set up configuration first. Configuration is structured as follows:

```
environments:
  example_env:
    example_system:
      system_type: jenkins
      sources:
        osp_jenkins:
          driver: jenkins
          url: 'https://some.jenkins.com'
          cert: False
          username: example_username  # Required specifically by Jenkins
          token: example_token        # Required specifically by Jenkins
```

Default location for the configuration file is `~/.config/cibyl.yaml`

Each type of system will require a different set of parameters in order to start using/querying it. For more information on how to set up configuration for CLI usage, read the configuration section.

1.4 Configuration

Cibyl CLI is fully visible and usable only once you've setup the configuration file. This is because the CLI is dynamically changing based on the type of CI systems you have defined in the configuration file.

1.4.1 Format

The configuration file is written in YAML and is divided in two sections: *environments* and *plugins*. See below for an example of a minimal configuration that shows both sections.

```
# Minimal configuration

environments:                 # List of CI/CD environments
  production:                 # An environment called "production"
    production_jenkins:       # A single system called "production_jenkins"
      system_type: jenkins    # The type of the system (jenkins or zuul)
      sources:                # List of sources belong to "production_jenkins" system
        jenkins_api:          # The name of the source which belongs to "production_
↪jenkins" system
          driver: jenkins     # The driver the source will be using
          url: https://...    # The URL of the system
          username: user      # The username to use for the authentication
          token: xyz          # The token to use for the authentication
          cert: False         # Disable/Enable certificates to use for the authentication

plugins:                      # (Optional) Specify the plugins to enable when running␣
↪Cibyl
  - openstack                 # OpenStack adds its own product related models and␣
↪arguments
```

The *environments* contains a list of environments. Each environment might contain one or more systems, which in turn might contain one or more sources. More details about this hierarchy can be found in the Core Models section. In short, an environment models a group of CI systems that are setup for a common purpose. At the same time, each CI system can have multiple sources of information available. See the *Full Configuration* section for an example of a configuration file with multiple environments, systems and sources.

The *plugins* section contains a list of plugins that should be loaded to provide cibyl with product-specific functionality.

1.4.2 Configuration Path

By default cibyl will look for the configuration file in the following paths:

- `~/.config/cibyl.yaml`
- `/etc/cibyl/cibyl.yaml`

A different path can be used if the argument `--conf path` is used. Additionally, the `--conf` argument also supports passing a URL to configuration file. If a URL is passed, it will be downloaded to `~/.config/cibyl.yaml`.

1.4.3 Sources

Cibyl supports multiple different types of sources. Each source may require some specific configuration. Below we link a page for each source implemented in cibyl. This pages contain a brief description of the source, a configuration sample and which plugins support it.

- Jenkins
- Zuul API
- Elasticsearch

- Zuul Definitions
- Jenkins Job Builder

1.4.4 Validate Configuration

The best way to validate the configuration you've added is correct, is to run the `cibyl` command. This should list the environments and systems specified in the configuration file. If the configuration is correct, then cibyl will print the environments and systems defined in the configuration. Taking the minimal configuration defined in the *Format* section, running cibyl will print:

```
Environment: production
  System: production_jenkins
```

If there is some problem with the configuration file, cibyl will raise one of the following errors:

- `ConfigurationNotFound`: There is no configuration file in any of the default paths or the path specified by the user.
- `EmptyConfiguration`: A configuration file was found, but it's empty.
- `MissingEnvrionments`: The configuration file does not include any environments
- `MissingSystems`: An environment in the configuration file does not include any systems
- `MissingSystemKey`: A system in the configuration is missing a required key
- `MissingSystemType`: The type of one system in the configuration was not specified
- `NonSupportedSystemKey`: A key in the configuration of one system is not supported (e.g. a parameter was added to the wrong system)
- `MissingSystemSources`: A system in the configuration has no sources
- `NonSupportedSourceKey`: A key in the configuration of one source is not supported (e.g. a parameter was added to the wrong source)
- `NonSuppportedSourceType`: A source type in the configuration is not supported
- `MissingSourceKey`: The configuration of one source is incomplete and missing a required key
- `MissingSourceType`: The type of a source in the configuration is not specified

1.4.5 Full Configuration

As mentioned before, the configuration file might contain many environments, systems and sources. In the example below, a configuration consisting of two environments is shown. The first environment *production*, contains three systems: *production_jenkins_1*, *production_jenkins_2* and *production_zuul*. The *production_jenkins_1* system contains two sources, a Jenkins source called *jenkins1_api* and a Jenkins Job Builder source called *job_definitions*. The *production_jenkins_2* and *production_zuul* systems contain one source each, a Jenkins and Zuul source, respectively. Finally, the *staging* environment contains a system *staging_jenkins* with a single Jenkins source.

```
# Full Configuration Example

environments:                 # List of CI/CD environments

  production:                 # An environment called "production"

```

(continues on next page)

(continued from previous page)

```
    production_jenkins_1:     # A single system called "production_jenkins_1" belongs to
↪"production" environment
      system_type: jenkins    # The type of the system (jenkins or zuul)
      sources:                # List of sources belong to "production_jenkins" system

        jenkins1_api:       # The name of the source which belongs to "production_
↪jenkins_1" system
          driver: jenkins   # The driver the source will be using
          url: https://...  # The URL of the system
          username: user    # The username to use for the authentication
          token: xyz        # The token to use for the authentication
          cert: False       # Disable/Enable certificates to use for the authentication

        job_definitions:    # Another source that belongs to the same system called
↪"production_jenkins_1"
          driver: jenkins_job_builder
          repos:
            - url: https://job_definitions_repo.git

    production_jenkins_2:     # Another system belongs to the "production" environment
      system_type: jenkins
      sources:

        jenkins2_api:
          driver: jenkins
          url: https://...
          username: user
          token: xyz
          cert: False

    production_zuul:
      system_type: zuul
      sources:

        zuul_api:
          driver: zuul
          url: https://...

  staging:                    # Another environment called "staging"

    staging_jenkins:
      system_type: jenkins
      sources:

        staging_jenkins_api:
          driver: jenkins
          url: https://...
          username: user
          token: xyz
```

1.4.6 Disabling environments, systems and sources

It's possible to disable each type of entity in Cibyl with the directive `enabled: false`. For example, the following will disable the environment `staging` and the system `production-2`

```
environments:
  production:
    production_jenkins_1:
      system_type: jenkins
      sources:
        jenkins_api_prod:
          driver: jenkins
          url: https://...
          username: user
          token: xyz
    production_jenkins_2:
      enabled: false        # Makes 'production_jenkins_2' system disabled
      system_type: jenkins
      sources:
        jenkins_api_prod:
          driver: jenkins
          url: https://...
          username: user
          token: xyz
  staging:
    enabled: false          # Makes 'staging' environment disabled
    staging_jenkins:
      system_type: jenkins
      sources:
        jenkins_api_staging:
          driver: jenkins
          url: https://...
          username: user
          token: xyz
```

Note: you can't use a disabled environment, even if specifying it directly with one of the following arguments: –envs, –systems and –sources.

1.5 CLI

Cibyl can be used as as a CLI tool to query CI related information from multiple CI systems. Cibyl can provide information from two domains: CI/CD data and product-specific data. The first corresponds to concepts that are pertinent to many CI systems, like builds, jobs, tests or system-specific concepts like pipelines and tenants in Zuul systems (for more details on CI concepts supported by cibyl check the core models section).

Product-specific information is provided by plugins (see the openstack plugin as an example). As the name indicates, these are properties that are not related to the CI system but to the product being tested. In the case of openstack, some examples include the topology of the deployment, or the openstack version being deployed.

Due to this dual source of information, cibyl can be used to query both kinds of properties, as well as combine them in more complex queries. This page will provide several examples of queries that can be done with cibyl.

CLI arguments that accept values follow the assumption that if they are passed without any value, the user is requesting to list the corresponding information, while if passed with a value, the value will be used as a filter. As an example, running `cibyl query --jobs` will list all available jobs, while `cibyl query --jobs abc` will list the jobs that have the string *abc* in their names.

Note: Throughout this page we assume for simplicity that there is only one CI system defined in the user configuration. Nevertheless, every command shown here can be run with a configuration composed of multiple environments, systems and sources. Check the configuration section for more details on how to construct the configuration for such cases.

Note: In cibyl, all cli arguments that accept a value, like *–jobs* or *–tests* will consider the input as a regular expression. The regex are matched using the syntax defined in the re module (docs).

1.5.1 CLI organization

Cibyl supports the following subcommands:

- query
- features
- spec

This page will cover many uses of the `query` subcommand, for examples of the `features` one see the features section and for examples of the `spec` subcommand see the spec section.

1.5.2 General parameters

Before listing interesting use-cases, cibyl has also a set of application-wide cli arguments that will affect the queries. This arguments must be specified before the subcommand. For example, to run a command to list of all jobs with a verbose output and a configuration file outside the default path, you should run:

```
cibyl -v --config path/to/config.yml query --jobs
```

The application-level arguments supported by cibyl are:

-d, --debug
: Turn on debug-level logging

-v, --verbose
: Verbosity level. This flag is additive, -vv will print more output than -v. In verbose mode, additional fields for the output are printed, such as the url for jobs, or the duration for builds.

-c, --config
: Path to the configuration file, this can be a local path, or a url. If it's a url, the file will be downloaded and stored in your local machine.

--log-mode=[terminal|file|both]
: Where to write the logging output. Options are terminal, file or both, default is both.

--log-file
: Path to store the logging output if the *file* or *both* option for `--log-mode` is selected, default is *cibyl_output.log*.

--output-format=[text|colorized|json]
: Sets the output format. Both text and colorized print to standard output, but the colorized uses color for better visuals. Json support is not complete.

-o, --output
Write output to the file passed as value.

-p, --plugin
Plugins to use in the queries.

1.5.3 CI/CD queries

Environment selection

The user configuration might consist of many environments, systems and sources. However, for any particular query the user might want to only use a subset of the defined environments. There are four arguments that can be used to achive this:

--envs
Environments to use in the query, filtering by name

--systems
Systems to use in the query, filtering by name

--system-type
Systems to use in the query, filtering by type

--sources
Sources to use in the query, filtering by name

The arguments presented in this section can be combined with any of the commands shown anywhere in this page.

Job queries

Cibyl can be used to query the list of all jobs defined in a CI system:

```
cibyl query --jobs
```

or to list the jobs that contain the string *123*:

```
cibyl query --jobs 123
```

or to list the jobs that end with the string *123*:

```
cibyl query --jobs "123$"
```

Build queries

Cibyl can be used to query the list of all builds for all jobs defined in a CI system:

```
cibyl query --jobs --builds
```

or the last build for all jobs:

```
cibyl query --jobs --last-build
```

or the last build for all jobs where that build failed:

```
cibyl query --jobs --last-build --build-status FAILED
```

Note: The value for the –build-status argument in case insensitive, so both *FAILED* and *failed* would produce the same result

or the last build for all jobs that have the string *123* in the name and where that build failed:

```
cibyl query --jobs 123 --last-build --build-status FAILED
```

Test queries

Cibyl can be used to query the list of all tests for all jobs defined in a CI system. To query for tests, the user must specify a build where the tests were run, either through the –last-build or –builds arguments:

```
cibyl query --jobs --last-build --tests
```

listing the tests that run in build number 5:

```
cibyl query --jobs --builds 5 --tests
```

or list the tests that contain the string *123* in their name:

```
cibyl query --jobs --last-build --tests 123
```

or list only the failing tests:

```
cibyl query --jobs --last-build --test-result FAILED
```

or list only the tests that run for more than 5 minutes, but less than 10 minutes (test duration is specified in seconds):

```
cibyl query --jobs --last-build --test-duration ">300" "<600"
```

Note: The –test-duration is a ranged argument. In cibyl, ranged arguments take multiple values in the form "OPERATOR VALUE", without the space in between. Common operators like "<", ">", "!=", "==", "<=", ">=" are supported. Additionally using a single equal sign "=" is also a valid equality operator, and if no operator is specified, the equality one is used by deafault.

Zuul specific queries

In cibyl, there are some argumetns that are only supported when running queries against a Zuul system, and will be ignored otherwise. For example, we can list all jobs in the *default* tenant:

```
cibyl query --tenants default --jobs
```

or list all jobs related to project *example-project* in all tenants:

```
cibyl query --projects example-project --jobs
```

or list all jobs under the *check* pipeline:

```
ciby query --pipelines check --jobs
```

The arguments shown in previous sections can be combined with the Zuul specific ones. For example, we could use cibyl to list the last build of the jobs that have the string *123* in their name, belong to a project named *example*, to a *check* pipeline and under the *default* tenant, but only if the build was successful:

```
cibyl query --tenants default --project example --pipeline check --jobs 123
--last-build --build-statu SUCCESS
```

Jenkins specific queries

As is the case with Zuul systems, Jenkins systems have some specific arguments that can combined with the more general ones. Cibyl can query Jenkins systems to list the stages that were run in a build. For example the following command would show the stages run for the last build of the job called *job_name*:

```
cibyl query --jobs job_name --last-build --stages
```

1.5.4 Product queries

Openstack queries

As part of the functionality provided by the openstack plugin, cibyl can query the CI systems for openstack related information. For example it's quite simple to list the version of the ip protocol used in each job:

```
cibyl query --ip-version
```

or listing the jobs that use ipv6 protocol:

```
cibyl query --ip-version 6
```

Similarly, other openstack properties can be used for queries, and can be combined for more complex queries. Building on the previous example, let's build a cibyl command to show the network backend used in every job that also used ipv6:

```
cibyl query --ip-version 6 --network-backend
```

Other examples of relevant openstack arguments include checking which jobs setup the tests from git, instead of rpm packages:

```
cibyl query --test-setup git
```

or filtering by the number of compute and controller nodes used in a deployment. This can be done via the `--controllers` and `--computes` arguments, which are ranged arguments (see *note above* for more deatils on what that means). Let's see an example of how to query for those jobs that use at least 2 compute nodes and more than 3 controller nodes, but no more than 6 controllers:

```
cibyl query --controllers ">3" "<=6" --computes ">=2"
```

The list shown here is not a comprehensive collection of all the arguments defined in the openstack plugin, check the plugin page in the documentation for the full list.

Combination of openstack and CI/CD queries

In a cibyl query, CI/CD and openstack arguments can be combined to form more complex queries. This section will show some examples of such calls. The following call will list all jobs that contain the string *example*, deploy openstack using *ceph* as the cinder backend and *geneve* as the network backend, and also print the last build for each job:

```
cibyl query --jobs example --cinder-backend ceph --network-backend geneve
--last-build
```

the previous example could be expanded to only list those jobs that had a passing last build:

```
cibyl query --jobs example --cinder-backend ceph --network-backend geneve
--last-build --build-status SUCCESS
```

1.6 API

1.7 Parser

Cibyl provides two sources of user input, the configuration file and the command line arguments. The configuration file details the ci environment that the user wants to query, while the command line arguments tell Cibyl what the user wants to query.

Cibyl's cli is divided in several subcommands. The parser is the component responsible for bringing all the subcommands together and ensuring the corresponding arguments are added. In the case of the `features` subcommands that is simple, since it only has one argument. The case of the `query` sucommand is different, since the cli arguments are extended dynamically depending on the contents of the configuration.

Note: The rest of this page is relevant **only** for the `query` subcommand.

When running `cibyl query -h` only the arguments that are relevant to the user, according to its configuration, will be shown. If there is no configuration file, Cibyl will just print a few general arguments when calling `cibyl query -h`. If the configuration is populated then arguments will be added depending on its contents.

The parser is extended using a hierarchy of CI models. This hierarchy is Cibyl's internal representation of the CI environments. The models are created after reading the configuration and the hierarchy is implicitely defined in the API attribute of said models. For example, one environment might include a Jenkins instance as CI system, and have it also as source for information, in addition to an ElasticSearch instance as a second source. With this environment, if the user runs `cibyl query -h`, it will show arguments that are relevant to a Jenkins system, like `--jobs`, `--builds` or `--build-status`. In such a case it will not show arguments like `--pipelines` which would be useful if the CI system was a Zuul instance.

The API of a CI model is a dictionary with the following structure (extracted from the System API):

```
API = {
    'name': {
        'attr_type': str,
        'arguments': []
    },
    'sources': {
        'attr_type': Source,
        'attribute_value_class': AttributeListValue,
```

(continues on next page)

(continued from previous page)

```
        'arguments': [Argument(name='--sources', arg_type=str,
                               nargs="*",
                               description="Source name")]
    },
    'jobs': {'attr_type': Job,
             'attribute_value_class': AttributeDictValue,
             'arguments': [Argument(name='--jobs', arg_type=str,
                                    nargs='*',
                                    description="System jobs",
                                    func='get_jobs')]}
}
```

each key corresponds to the name of an attribute, and the value is another dictionary with attribute-related information. At this point we need to distinguish between arguments and attributes. In Cibyl an `Argument` is the object that is obtained from parsing the user input. The values passed to each option like `--debug` or `--jobs` are stored in an `Argument`. Attributes correspond to the actual key-value pairs in the API. An attribute has an `attribute_value_class` which by default is `AttributeValue`, but can also be `AttributeDictValue` and `AttributeListValue`. The difference between the three is the how they store the arguments. The first is intended to hold a single option (things like name, type, etc.). While the other two hold a collection of values either in a dictionary or a list (hence the name). The information provided by the user is accessible throgh the `value` field of any `Attribute` class.

Each API element has also an *attr_type*, which describes what kind of object will it hold. In the example above *name* will hold a string, while *jobs* will hold a dictonary of Job objects. This allows us to establish the hierarchy mentioned previously, by checking if the *attr_type* field is not a builtin type. Finally, there is an *arguments* field, which associates the actual options that will be shown in the cli with an attribute. An attribute may have no arguments, one argument or multiple arguments associated with it.

`Argument` objects have a set of options to configure the behavior of the cli. The *name* determines the option that will be shown, *arg_type* specifies the type used to store the user input (str, int, etc.), *nargs* and *description* have the same meaning as they do in the arparse module. The *level* argument, measures how deep in the hierarchy a given model is. Finally, we see the *func* argument, which points to the method a source must implement in order to provide information about a certain model. In the example shown here, only jobs has an argument with *func* defined, as it is the only CI model present. If the user runs a query like:

```
cibyl query --jobs
```

then Cibyl will look at the sources defined and check whether any has a method `get_jobs`, and if it finds one it will use it to get all the jobs available in that source.

Arguments are added to the application parser in the `extend_parser` method of the `Orchestrator` class. This method loops through the API of a model (in the first call it will be an `Environment` model) and adds its arguments. If any of the API elements is a CI model, the element's API is recursively used to augment the parser. As the `extend_parser` method iterates through the model hierarchy, it creates a graph of the relationships between query methods (the sources' methods that are added to the arguments' *func* attribute). The edges of the graph are created when a new recursive call is made. As an example, when exploring the API for the Job model, we know that the arguments will call `get_jobs`, so when a new call is made for the Build API, a new edge wil be created from `get_jobs` to all the new query methods that are found, in this case it will be `get_builds`.

For each recursive call, the **level** is increased. The level parameter is key to identify the source of information for the query that the user sends. In the Jenkins environment example mentioned before, we may have a hierarchy like:

```
Environment => System => Job => Build
```

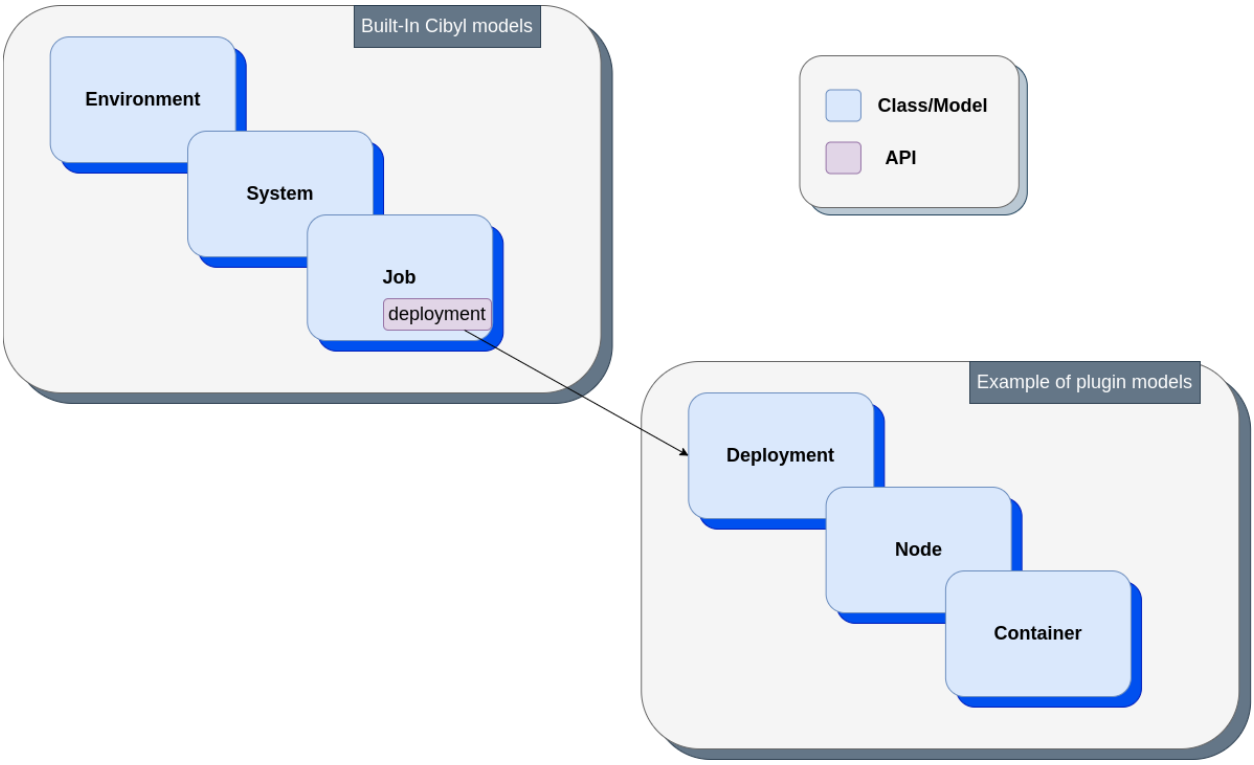
where each at each step we increase the level by 1. We can then parse the cli arguments and sort by decreasing level. To select which query method should be called, cibyl relies on the graph constructed during the call to `extend_parser`. It

iterates over the sorted list of arguments and for each of them constructs a path to the root of the graph. The intermediate nodes in this path are removed from the list of arguments to query, since by the hierarchical nature of the relationship between the models, calling an argument's *func* makes the call to the argument's parent *func* redundant.

In the example above, *Build* is the model with the largest level. If we assume that user has made a call like `cibyl --jobs --builds`, we want to query the sources for builds, but we known that each build will be associated with a job, and each job will be associated with a system, etc. We also know that after calling `get_builds`, we will not need to call `get_jobs`. Thus we get a sorted list of arguments, which is [*builds*, *jobs*]. We create a path from *builds* to the root of the graph, which in the case of a Jenkins systems is *jobs* (for a zuul system this would be more complex). After iterating over the path, we remove *jobs* from the list of arguments to query, since *builds* already will provide the *jobs* information.

1.8 Plugins

Plugins allow you to extend built-in models with your own models. This can be useful in case you would like to associate product related data with your CI models for example, as can be seen in the image below. In this case, the Job model is being associated with the Deployment model through the deployment key in Job's API.



A supported plugin in Cibyl has to adhere following structure:

```
cibyl
├── plugins
│   └── example           # Arbitrary plugin name
│       └── __init__.py   # Should include Plugin class with _extend method
```

1.8.1 Plugin Class

```
class ExamplePlugin:

    def _extend(self, model_api: dict):
        model_api['new_attribute'] = {
            'attr_type': str,
            arguments: []
        }
```

1.9 Sources

Sources in Cibyl are responsible for performing the queries and getting the data the user is interested in. A source can be anything: a CI system, repository, database, etc. Cibyl supports the following sources out-of-the-box:

- Jenkins
- Jenkins Job Builder
- Zuul
- Elasticsearch
- Zuul Job Definitions

1.9.1 Configuring Sources

The following is an example of Jenkins source configuration:

```
environments:
  example_environment:
    jenkins_system:
      system_type: jenkins
      sources:
        jenkins_source:
          driver: jenkins
          username: some_username
          token: some_token
          url: https://jenkins.example.com
```

See configuration to understand how to properly configure Cibyl for CLI usage.

1.9.2 Source Interface

Each source can support one or more of the arguments specified by the different models of Cibyl. The only constraint regarding sources is that each source must inherit from the Source class.

1.9.3 Arguments Matrix

Table 1: The supported arguments in the different built-in sources

Argument / Source	Description	Jenkins	Zuul	ES	JJB	Zuul.d
–jobs	Jobs names or pattern Default: all jobs					
–builds	Build numbers Default: all builds					
–last-build	The last build of a job					
–build-status	Build status (default: all) failure, success, abandoned, unstable					
–tests	Test names or pattern Default: all tests					
–test-result	Test result (default: all) success, failed, skipped					
–test-duration	Test duration (in seconds, default: all) (Can be also range: “>=3”)					

1.10 Output

The output of a cibyl command is the result of the query made by the user. Cibyl provides a great amount of control on the format of this output. By default, cibyl will print the output to the terminal, using colored text.

The user can choose to print to a file using the `-o` or `--output` flag. This flag takes a file path as its value and will write there the query result.

Note: If the file specified exists, it will be overwritten.

The user can choose the format of the output. Currently three formats are supported:

- `colorized`, colored text, is the default mode, well suited for printing to a terminal, but not very useful if printing to a file
- `text`, plain text, ideal to use when writing to a file
- `json`, output in json format, useful if the output of cibyl has to be passed to another piece of software

The user can also control the level of detail of the output, using the `-v` or `--verbose` flag. This flag is cumulative, so `-vv` will produce more output than `-v`. As an example, *Job* models will have a url field, but it will only be printed in verbose mode. Similarly, *Test* models have a duration field that is only shown in verbose mode.

Additionally, cibyl also has a stream of logging output. Normally, cibyl will log the duration of the query, the system queried and where is the output written. If debug mode is used with `-d` or `--debug`, then additional information will be printed.

1.11 Jenkins

The Jenkins source pulls data from the Jenkins API.

1.11.1 Usage

To following is a configuration sample of how to configure the Jenkins source

```
# Minimal configuration

environments:                 # List of CI/CD environments
  production:                 # An environment called "production"
    production_jenkins:       # A single system called "production_jenkins"
      system_type: jenkins    # The type of the system (jenkins or zuul)
      sources:                # List of sources belong to "production_jenkins" system
        jenkins_api:          # The name of the source which belongs to "production_
↪jenkins" system
          driver: jenkins     # The driver the source will be using
          url: https://...    # The URL of the system
          username: user      # The username to use for the authentication
          token: xyz          # The token to use for the authentication
          cert: False         # Disable/Enable certificates to use for the authentication

plugins:                      # (Optional) Specify the plugins to enable when running␣
↪Cibyl
  - openstack                 # OpenStack adds its own product related models and␣
↪arguments
```

1.11.2 Plugin Support

The Jenkins source is supported by the following built-in plugins:

- OpenStack

1.12 Zuul API

The Zuul API source pulls data from the Zuul CI/CD system.

1.12.1 Usage

The following is a configuration sample of how to configure the Zuul source

```
environments:                   # List of CI/CD environments
  production:                 # An environment called "production"
    production_zuul:          # A single system called "production_jenkins"
      system_type: zuul       # The type of the system
      sources:                # List of sources belong to "production_jenkins" system
        zuul_api:             # The name of the source which belongs to "production_zuul
↪" system
          driver: zuul        # The driver the source will be using
          url: https://...    # The URL of the system
          tenants:            # List of tenants to use. This section is optional
              - default       # and allows the user to restrict which zuul
              - local         # tenants will be queried can be useful
```

1.12.2 Plugin Support

The Zuul source is supported by the following built-in plugins:

- OpenStack

1.13 Jenkins Job Builder

"Jenkins Job Builder" is the source for obtaining information from jenkins job definitions repositories. It's supported only with Jenkins CI/CD system.

1.13.1 Usage

To following is a configuration sample of how to configure the 'Jenkins Job Builder' source

```
environments:                       # List of CI/CD environments
  production:                       # An environment called "production"
    production_jenkins:             # A single system called "production_jenkins"
      system_type: jenkins          # The type of the system (jenkins or zuul)
      sources:                      # List of sources belong to "production_jenkins"␣
↪system
```

(continues on next page)

(continued from previous page)

```
        jjb:                            # The name of the source which belongs to
↪"production_jenkins" system
          driver: jenkins_job_builder   # The driver the source will be using
          repos:                        # List of repositories where the job definitions␣
↪are located
              - url: 'https://jjb_repo_example.git'
```

1.13.2 Plugin Support

The 'Jenkins Job Builder' source is supported by the following built-in plugins:

- OpenStack

1.14 Elasticsearch

The Elasticsearch source pulls data from the different indexes of the Elasticsearch database.

1.14.1 Usage

To following is a configuration sample of how to configure the Elasticsearch source

```
environments:                     # List of CI/CD environments
  production:                     # An environment called "production"
    production_jenkins:           # A single system called "production_jenkins"
      system_type: jenkins        # The type of the system (jenkins or zuul)
      sources:                    # List of sources belong to "production_jenkins" system
        es:                       # The name of the source which belongs to "production_
↪jenkins" system
          driver: elasticsearch   # The driver the source will be using
          url: https://...        # The URL of the source
```

1.14.2 Fields

Elasticsearch should include the following fields in order to be fully operational:

- job_name
- build_number
- build_result
- current_build_result

1.14.3 Plugin Support

The Elasticsearch source is supported by the following built-in plugins:

- OpenStack

1.15 Zuul Definitions

Zuul Definitions is the source for pulling data out of Zuul job definition repositories (usually repos with zuul.d directory).

Zuul definitions support two ways to gather data from Zuul job definition repositories:

1. By Clonning and parsing files in zuul.d dir
2. By Quering GitHub API and parsing files in zuul.d dir

Cibyl will clone repos to ~/.cibyl directory.

When `remote: True` option is set it will query using GitHub API instead of cloning the repositories and query them locally.

> **Warning:** To prevent rate limiting on GitHub you might need to add username and token options in the config.

1.15.1 Usage

To following is a configuration sample of how to configure the Zuul definitions source to work with local repos

```
environments:                 # List of CI/CD environments
  production:                 # An environment called "production"
    production_zuul:          # A single system called "production_jenkins"
      system_type: zuul       # The type of the system
      sources:                # List of sources belong to "production_jenkins" system
        zuul_api:             # The name of the source which belongs to "production_zuul
↪" system
          driver: zuul.d      # The driver the source will be using
          remote: False       # Optional as this is the default
          repos:              # The repos to clone and query when running Cibyl query␣
↪commands
            - url: 'http://zuul_defitions_repo.git'
            - url: 'http://zuul_defitions_repo1.git'
```

To following is a configuration sample of how to configure the Zuul definitions source to work with GitHub API

```
environments:                   # List of CI/CD environments
  production:                 # An environment called "production"
    production_zuul:          # A single system called "production_jenkins"
      system_type: zuul       # The type of the system
      sources:                # List of sources belong to "production_jenkins" system
        zuul_api:             # The name of the source which belongs to "production_zuul
↪" system
          driver: zuul.d      # The driver the source will be using
```

(continues on next page)

(continued from previous page)

```
          remote: True        # Query GitHub API instead of querying local repos
          username: user      # Required only when 'remote: True'
          token: xyz          # Required only when 'remote: True'
          repos:              # The repos to query using GitHub API
            - url: 'http://localhost/zuul_defitions_repo.git'
            - url: 'http://localhost/zuul_defitions_repo1.git'
```

1.15.2 Plugin Support

The "Zuul Definitions" source is supported by the following built-in plugins:

- OpenStack

1.16 Core Models

Core models (aka CI/CD models) are built-in CI/CD Cibyl models:

- Environment: A CI/CD environment with one or more CI/CD systems. This is mostly a logical separation, rather than a physical one.
- System: A CI/CD system such as Jenkins, Zuul ,etc.
- Pipeline: A specific Zuul concept which used for describing a workflow
- Job: A particular task/automation in the CI/CD system
- Build: An execution instance of a job
- Test: A test execution that is part of a build

The way they are organized and associated one with each other, mainly depends on the type of the CI/CD system being used. For a Jenkins system for example, the hierarchy includes Job and Build models, while for Zuul system, the hierarchy includes Pipeline, Job and Build models.

```
Environment
├── System
│   └── Job       # Jenkins
│       └── Build
│           └── Test
│   └── Pipeline  # Zuul
│       └── Job
│           └── Build
│               └── Test
```

1.17 Plugin Models

Plugin models are provided by different plugins. They are not associated by default with the core models of Cibyl, but only when the plugin is being used. In addition, the way the plugin models are associated with core models, is very much depends on the implementation of the plugin.

An example of plugin models can be seen in openstack plugin page

1.18 OpenStack Plugin

OpenStack is an open source cloud software. The OpenStack plugin associates CI job model with OpenStack deployment model.

1.18.1 Models

- Deployment: An entire OpenStack cluster
- Node: A single node in an OpenStack deployment/cluster associated with a single deployment
- Container: A container associated with a single node
- Package: An RPM associated with either a single node or a single container
- Service: A service associated with a single deployment

```
Deployment
├── Node
│   └── Container
│       └── Package
│   └── Package
├── Service
```

1.18.2 Usage

To use the OpenStack plugin with Cibyl, specify *–plugin openstack* or include it in the configuration file.

1.18.3 Spec

Note:

This feature is only fully implemented with the Jenkins automation system.
It is partially supported with Zuul (The option will work but will not provide the complete specification)

---

cibyl spec JOB_NAME allows you to easily get the full OpenStack specification of a single job.

The idea behind it is to allow the user to quickly get information on which OpenStack services and features are covered by a single job so the user doesn't have to go and deep dive into the job configuration and build artifacts to figure it out by himself.

An example of an output from running *cibyl spec JOB_NAME*:

```
Openstack deployment:
  Release: 17.0
  Infra type: virt
  Topology: compute:2,controller:3,ironic:2
  Network:
    IP version: 4
    Network backend: geneve
    ML2 driver: ovn
    Security group mechanism: native ovn
    DVR: True
    TLS everywhere: False
  Storage:
    Cinder backend: lvm
  Ironic:
    Ironic inspector: True
    Cleaning network: False
```

Arguments Matrix

Table 2: The supported arguments in the different built-in sources

Argument / Source	Description	Jenkins	Zuul	ES	JJB	Zuul.d
–ip-version	The IP version used by the deployment (4 or 6)					
–release	OpenStack Release (OSP and RDO supported)					
–infra-type	The infrastructure on which OS is deployed (e.g. ovb, baremetal, virthost)					
–topology	The combination of node types deployed					
–nodes	List of nodes on the topology.					
–controllers	Number of controllers (Can be also range: ">=3")					
–computes	Number of computes (Can be also range: ">=3")					
–ml2-driver	Which ml2 driver does the deployment use					
–network-backend	What network protocol is used (e.g. vxlan, vlan, ...)					
–cinder-backend	What cinder backend is used (vlan, Ceph, Netapp, nfs)					
–containers	List of containers running on the hosts					
–packages	Package installed by the					

1.19 Features

Cibyl allows users to define their own product related data in form of what is known as "features". Features are basically blocks of code with the purpose of querying for specific product features in one or more environments.

Out of the box Cibyl supports multiple features for existing plugins and users can easily list them with `cibyl features`

Allowing users to define their own sort of product arguments has multiple advantages:

- Use internal project functions and mechanisms to define complex custom queries
- Consistent approach towards querying for product data, in different environments and sources
- Sharing product related data with other users without extending endlessly the number of product arguments supported by Cibyl

1.19.1 Usage

To list all the existing features: `cibyl features`

Query IPv4 feature: `cibyl features ipv6`

Query two features: `cibyl features ipv6 ha`

Query for a feature in specific set of jobs: `cibyl features ha --jobs production`

1.19.2 Development

Would like to add a new feature? Read the features development section.

1.20 Tests

Cibyl tests cover the following:

- unit: testing each component of the application
- coverage: verify unit testing coverage is above 90%
- e2e: testing as a user would experience it
- linters: code analysis
- docs: documentation testing

Each of the above can be executed with `tox -e <type>` or `tox` to run them all

1.21 Sources

To add/develop a new type of source, follow the following guidelines:

- A source should be added to cibyl/sources/<SOURCE_NAME>
- The source class you develop should inherit from the Source class (`cibyl/sources/source.py`)
- For a source to support an argument, it should implement the function name associated with that argument

- Each source method that implements a method of an argument, should be returning an AttributeDict value of the top level entity associated with the CI systems (e.g. `AttributeDictValue("jobs", attr_type=Job, value=job_objects)`)
- A source should handle only CI/CD related data. If you would like a certain source to pull a product related data, you should add a source class (with the same name as the CI/CD source) to corresponding plugin (`cibyl/plugin/<PLUGIN_NAME>/sources/<SOURCE_DIR/FILE>`)

1.22 Features

In cibyl we define *features*, which are classes containing a query method that can run a custom query using python code. A feature is defined as a class that inherits from the *FeatureDefinition* class, defined in cibyl/features/__init__.py.

There is a FeatureTemplate class that can be used to quickly define simple features. The query method of this class will select the most appropiate source considering the speed_index and the method to query in the source. To define a new feature using this template, one only needs to define a class that inherits from FeatureTemplate, set the attribute *method_to_query* to the method of choice for the source and include in the *args* attribute the arguments that should be passed to the source's method to perform the query (see for example the HA, IPV4, IPV6 features as a sample).

One could define a feature without using the FeatureTemplate code at all, the only requirements would be that the class should provide a query method that accepts a system and returns an AttributeDictValue object, and should define a name attribute for the feature. This way of implementing a feature gives the developer total freedom, but does not provide some functionality like selecting the best sources given the input arguments. There could be a mixed implementation, that relies on the FeatureTemplate query method but provided a bit more flexibility. Let's say for example that one wanted a feature called Example that wants to check whether a system has any job called 'example' with at least 3 passing builds and runs a test called 'test_example'. Such a feature could be implemented for example like:

```
class Example(FeatureTemplate, FeatureDefinition):
    def __init__(self):
        self.name = "Example"

    def get_template_args(self):
        """Get the arguments necessary to obtain the information that defines
        the feature."""
        args = {}
        args['jobs'] = Argument("jobs",  arg_type=str,
                                description="jobs",
                                value=["example"])
        args['builds'] = Argument("build", arg_type=str,
                                  description="build",
                                  value=[])
        args['jobs'] = Argument("tests",  arg_type=str,
                                description="tests",
                                value=["test_example"])
        return args

    def query(self, system, **kwargs):

        def get_method_to_query(self):
            return "get_builds"
         self.get_method_to_query = get_method_to_query
        return_builds = super().query(system, **self.args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        def get_method_to_query(self):
            return "get_tests"
        self.get_method_to_query = get_method_to_query
        return_tests = super().query(system, **self.args, **kwargs)
        # more code to combine the the returns and apply the desired
        # conditions
```

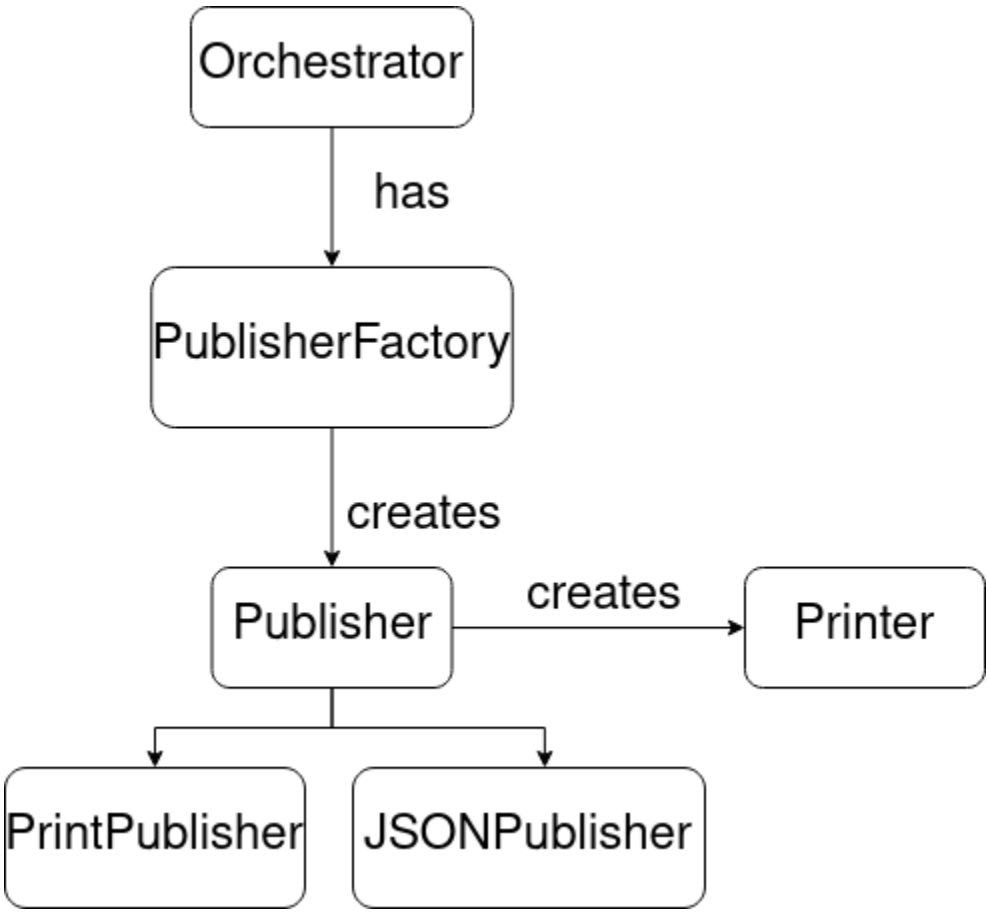
Features are classes that inherit from the FeatureDefinition class. The name of the module where the feature is defined is used as a category for the features it contains. Features are loaded in the orchestrator, in the load_features method. There, cibyl will go through the paths that are registered by the plugins and the default location to look for features. If features are found and requested by the user, the the run_features method is executed. If not, a normal query is executed.

As mentioned before, the query method should return an AttributeDictValue with the appropiate CI model according to the system and source used. These models are added to the system in the run_features method. If more than one feature is run, the output is combined to filter the returned models to add only those that satisfy all features. In addition, for each feature that runs, a Feature model is added to the system. This model (which is a CI model, akin to a System or Job) has only two attributes, the feature name and a boolean marking whether the feature is present in the system or not.

After all features run, the publisher is used to print all the output. The same publisher is used for both normal queries and feature queries. The printers for all systems will print the Feature models added to each system, and after that it will continue printing other information found in the system if the user ran cibyl with other arguments like –jobs. In order to handle the different cases, there are two kind of queries added to the QueryType class: *FEATURES* and *FEATURES_JOBS*. The first will signal the case when the user has called the features subcommand, while the second will mark the case where the user has called the features subcommand with the `--jobs` argument.

1.23 Output

To see an overview of cibyl output from a user's perspective see the output page. From a developer's perspective there are three objects that are involved in printing the output, which are summarized in the diagram below.

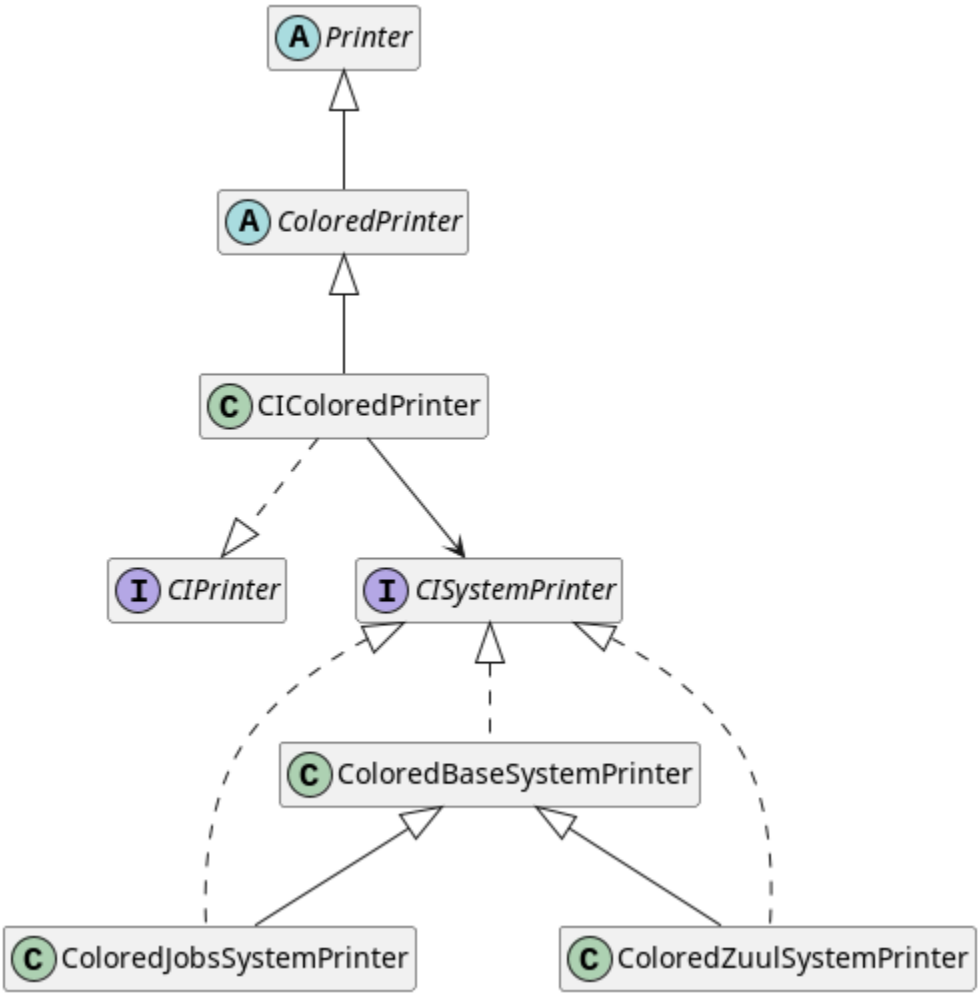


First, the *Orchestrator* processes the query and creates a *Publisher* to handle the output creation. There are two kinds of publishers: the *PrintPublisher*, which prints human-readeable text and the *JSONPublisher* that prints JSON output.

The main difference is that the PrintPublisher prints the output for each system after each environment is queried, while the JSONPublisher prints the output after all environment are queried. To produce valid json, all the output needs to be aggregated into a single object, but when printing human-readeable text, producing output after each environment is queried gives faster feedback to the user.

To produce the output, the Publisher creates a *Printer* object. Cibyl has a *Printer* abstract class that is specialized. The used printer is typically called *CI*Printer*. The *CI* prefix is used because the class implements the interface defined by the `CIPrinter` class. The interface mandates the implementation of a *print_environment* method. This method takes an environment object and produces a string representation of its contents.

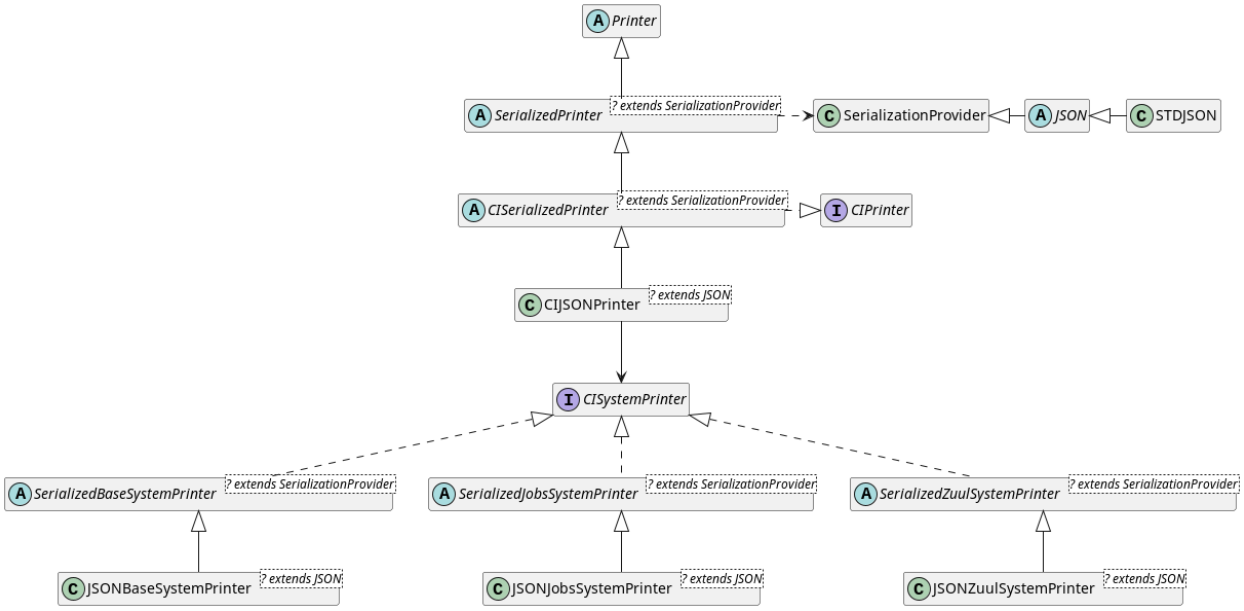
There are several Printer classes in cibyl, specialized depending on the output format and the contents. For example, for printing colored output, the hierarchy shown in the diagram below is established.



The class *CIColoredPrinter* is the Printer that is used for colored text and it will produce a string representation for all core models. While producing the output, the printer creates a *CISystemPrinter* object, which is specialized depending on which kind of system (zuul or jenkins) is being processed. The system printer is the object that will go through the whole model hierarchy, starting at the system level, and complete the output string.

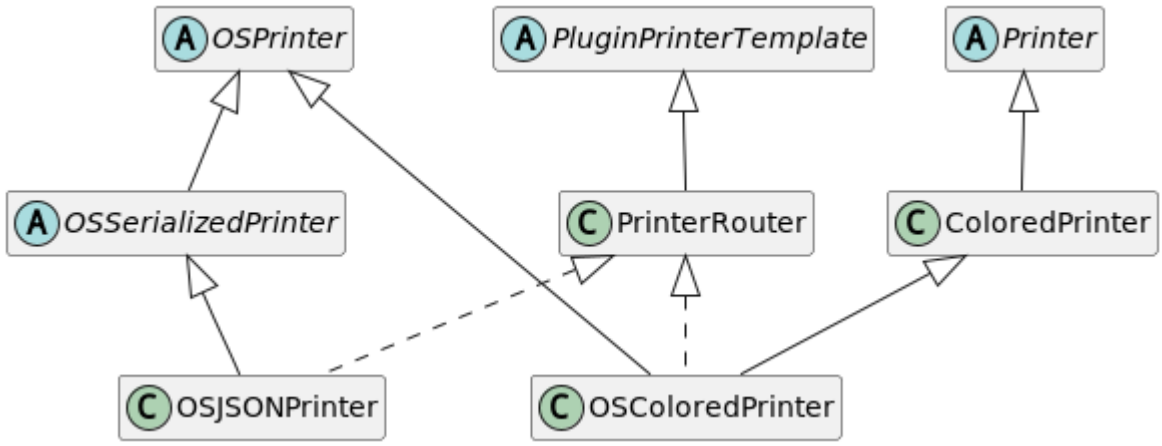
As an aside, the *ColoredPrinter* class takes as argument a *Palette* object. Using a *DefaultPalette* will produce colored output, while using a *ClearText* palette will produce plain text (which is the result of passing the flag *-f text* to cibyl).

For serialized text (json being the main example), there is another set of classes that provide the funcionality, as shown in the diagram below:



In this case, we have a generic *CISerializedPrinter* that can be specialized depending on the output format. Currently only a JSON implementation is available, but through the use of a different *SerializationProvider*, a YAML or XML implementation could be easily created. For json output, the printer would be the *CIJSONPrinter*, which would again have some type of *CISystemPrinter*. In this case it would be either a *JSONBaseSystemPrinter*, a *JSONJobsSystemPrinter* or a *JSONZuulSystemPrinter*. As can be seen in the diagram, these three classes are extensions of the *SerializedBaseSystemPrinter*, *SerializedJobsSystemPrinter* and *SerializedZuulSystemPrinter*, respectively.

The printers explained above deal with the core models. If the query involved any funcionality or models provided by a plugin, then the plugin own printer must be also called. Plugins must create their own printers by inheriting from the *PluginPrinterTemplate* abstract class. We will ilustrate this relationship using the openstack plugin as an example:



The openstack plugin introduces a *PrinterRouter* class which implements the *PluginPrinterTemplate* requirements (an *as_text* and an *as_json* method). Then, the plugin introduces two printers: *OSJSONPrinter* and *OSColoredPrinter* for json and human-readeable output. When producing the output, the system printers explained above will call the *as_text* or *as_json* method from the appropiate openstack printer and will get the correct string representation for the plugin-specific models found in the query.

1.24 Contribute

Please submit a pull requests to the cibyl project on GitHub.

1.24.1 Style

Cibyl CI enforces code linting according to the Google Python Style Guide

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search