r2c Documentation

Release 0.0.18

r2c

July 09, 2019

CONTENTS:

1 Getting Started **3**
- 1.1 Installation . . . 3
- 1.2 Creating an Analyzer . . . 4
- 1.3 Writing Analysis . . . 5
- 1.4 Running Analysis Locally . . . 7
- 1.5 Running on r2c . . . 8
- 1.6 Analyzing Results . . . 11

2 API Reference **15**
- 2.1 `analyzer.json` . . . 15
- 2.2 Database Schemas . . . 16
- 2.3 Docker Containers . . . 17
- 2.4 JSON Results . . . 17

3 Best Practices **19**

4 Troubleshooting **21**
- 4.1 The `/analysis` directory is completely empty . . . 21

Note: This documentation is for our closed beta as of release 0.0.18. Visit app.r2c.dev to join the waitlist!

r2c is a program analysis platform and cli for security researchers, hackers, and developers. Run your analysis on our infrastructure to quicky get results for millions of projects.

The platform offers:

1. scale: run your analysis at large scale across millions of open-source projects and their history
2. re-use: create analysis pipelines with reusable analyzers and shared results
3. determinism: easily define source code targets from GitHub, npm, PyPi, and others
4. collaboration & feedback: collaboratively triage, label, and disclose results

Have questions about the beta, r2c, or our fellowships? We'd love to help! Contact us at hello@r2c.dev.

CHAPTER

ONE

GETTING STARTED

This section will walk you through using `r2c-cli` and the platform by writing an analyzer the computes the percentage of whitespace in the source code of 1000 npm packages. WeâĂŹll use it to find which projects have checked in minified JavaScript files.

For detailed documentation of commands, options, and formats, see API Reference.

Installation

We'll start our journey with `r2c-cli`, which is used to develop and test analyzers locally before pushing them to the platform to run at scale.

Requirements

Note: We support Mac OSX and Ubuntu for local analyzer development.

Please install the following required software:

- docker: a tool for running software in isolated containers
- python3.6+: the programming language needed to run (but not to develop) analyzers
- pip3: the Python package manager used to install and update `r2c-cli`

When installing `docker`:

- Do not use `snap`, it is incompatible with `r2c-cli`
- For Ubuntu users we highly encourage setting up Docker for non-root users. You need to run `docker` as the same user you'll run `r2c-cli` with, and we discourage running `r2c-cli` as a root or system administrator

Run the following to test `docker`, `python3`, and `pip3` installation:

```
$ docker run hello-world
$ python3 --version
$ pip3 --version
```

Getting `r2c-cli`

It's easy! Just run:

```
$ pip3 install r2c-cli
```

To verify the `r2c-cli` installation, run:

```
$ r2c --help
Usage: r2c [OPTIONS] COMMAND [ARGS]...

Options:
  --debug                  Show extra output, error messages, and exception
                           stack traces
  --version                Show current version of r2c cli.
  --no-traverse-manifest   Don't attempt to find an analyzer.json if it doesn't
                           exist in the current or specified directory
  --help                   Show this message and exit.

Commands:
  build     Builds an analyzer without running it.
  init      Creates an example analyzer for analyzing JavaScript/TypeScript.
  login     Log in to the R2C analysis platform.
  logout    Log out of the R2C analysis platform.
  push      Push the analyzer in the current directory to the R2C platform.
  run       Run the analyzer in the current directory over a code directory.
  test      Locally run integration tests for the current analyzer.
  unittest  Locally unit tests for the current analyzer directory.
```

If the help prompt prints, you're ready to move on to Creating an Analyzer.

Creating an Analyzer

Creating the Boilerplate

In a directory where you want to create your new analyzer, run:

```
$ r2c init
```

For the `Analyzer name` prompt, enter `<YOUR-USERNAME>-minifinder`. For the rest of the prompts we'll use the defaults.

```
$ r2c init
Analyzer name (can only contain lowercase letters, numbers or - and _): <YOUR-USERNAME>-minifinder
Author name [Jav A. Script]:
Author email [hello@example.com]:
Will your analyzer produce:
- output for a particular `git` repository
- output for a particular git `commit` in a repo
- the same `constant` output regardless of commit or repo? (git, commit, constant) [commit]:
Does your analyzer output
- a single schema-compliant JSON file
- a full filesystem output? (json, filesystem) [json]:
âIJŤ Done! Your analyzer can be found in the <YOUR-USERNAME>-minifinder directory
```

Check it out by changing to the new folder:

```
$ cd <YOUR-USERNAME>-minifinder/
```

Understanding Analyzer Files

The `init` command created several files in the directory you initialized:

```
.
âŤIJ-- Dockerfile
âŤIJ-- README.md
âŤIJ-- analyzer.json
âŤŤ-- src
    âŤIJ-- analyze.sh
    âŤŤ-- unittest.sh
```

Each of these files is used by the r2c system in a different way.

- `analyzer.json` defines how your analyzer will interact with the r2c system and tools. Some important values in this file are:
- `analyzer_name`: The namespaced analyzer name. For the beta, all analyzers must be namespaced with `beta` to be uploaded to the platform.
- `version`: The version of the analyzer. *Versions should follow semantic versioning*. r2c uses the analyzer name, version, and other parameters for caching: when running an analyzer at scale, all of these are used to determine if the analysis has already been run on that piece of code.
- `dependencies`: This analyzer depends on the most basic component, `public/source-code`. It specifies that it depends on any version of the `public/source-code` component by specifying `"*"` as the version. For more complicated analysis, we could depend on components such as `r2c/transpiler` or `r2c/typed-ast`, which are beyond the scope of this tutorial.
- `Dockerfile` is responsible for the container's setup and configuration. In this file you can install dependencies to build and run your analysis. To learn more about Dockerfiles in general, see Docker's tutorial.

Note: Though it can be tempting to use images like `node:latest`, most analyzers should be deterministic and therefore benefit from pinning their base image to a specific version. For more information, see Best Practices.

- `src/analyze.sh` is the main entry point. From this file, we'll run your program that performs analysis!
- `src/unittest.sh` run your analyzer's unit tests, if it has them, inside the container by caling them from this file.

Once you've checked out those files, let's move on to Writing Analysis.

Writing Analysis

For this tutorial, we're writing an analysis that reports **how much of each file in a project is whitespace**. We'll use it to find which projects have checked in minified JavaScript files.

Setting up the Container

We will write this analysis in the Python programming language. Though we are using Python for this tutorial, **you can write analysis in any programming language** as each `docker` container can have different software installed.

Before we can write Python, we'll need to install it in our `docker` container. To do this, add the following line to the project's Dockerfile:

```
FROM ubuntu:17.10

RUN apt update && apt install -y python3 \
    && rm -rf /var/lib/apt/lists/*

RUN groupadd -r analysis && useradd -m --no-log-init --gid analysis analysis

USER analysis
COPY src /analyzer

WORKDIR /
CMD ["/analyzer/analyze.sh"]
```

When we edit our code later, the container will automatically be rebuilt by `r2c run`.

Writing the Code

We need to be able to count both whitespace and non-whitespace characters in a given file. We can do this with a simple regular expression in Python. Create a file `src/whitespace.py` with the following contents:

```
import json
import re
import sys

WHITESPACE_RE = re.compile("\s")


def count_whitespace(path):
    print("Counting whitespace in file {}".format(path))
    with open(path, "r", encoding="utf-8") as file:
        data = file.read()
    result = {}
    result["check_id"] = "whitespace"
    result["path"] = path
    result["extra"] = {}
    result["extra"]["size"] = len(data)
    result["extra"]["num_whitespace"] = len(WHITESPACE_RE.findall(data))
    return result


all_results = []
for path in sys.argv[1:]:
    all_results.append(count_whitespace(path))

with open("/analysis/output/output.json", "w") as output:
    output.write(json.dumps({"results": all_results}, sort_keys=True, indent=4))
```

This file computes the number of whitespace characters and total characters in each file in its input. When we run our analyzer, we want to run this file with all JavaScript input files as arguments.

We write this object to `/analysis/output/output.json` because this is a JSON-type analyzer. r2c also supports filesystem type analyzers, that modify or augment their input but want to preserve a filesystem structure or output large binary data, e.g. neural net training results. Most analysis eventually leads to JSON output, because JSON output is what gets consumed r2c's other tools.

To get just JavaScript files, we'll use the `find` program on our mounted source-code directory. Change `src/analyze.sh` to look like this:

```
1  #!/bin/bash
2
3  set -e
4  CODE_DIR="/analysis/inputs/public/source-code"
5
6  cd ${CODE_DIR}
7
8  find . -type f -name '*.js' -print0 | xargs -0 python3 /analyzer/whitespace.py
```

First, we change to the directory our source code is checked out. That folder is `/analysis/inputs/public/source-code/` inside the `docker` container. This location is a result of minifinder depending on the `source-code` component (configured in `analyzer.json`). For more information about dependencies and locating their output, see API Reference.

Then, we use the `find` command to find all files that end in `.js` and use `xargs` to have it pass all of those file paths as arguments to our python program.

Note: Though we wrote our python in a file `src/whitespace.py`, inside of `src/analyze.sh` we invoke it at the path `/analyzer/whitespace.py`. This is because in line 12 of our Dockerfile, we copy the `src` folder to the `/analyzer` folder inside the container.

Now that we've written our code, let's try Running Analysis Locally.

Running Analysis Locally

Let's test our analyzer! First we'll need a target JavaScript project to test it on. Let's use the popular left-pad project. Make a folder to keep our test projects in and clone `left-pad` to the folder:

```
$ mkdir ~/test-repos
$ git clone https://github.com/stevemao/left-pad ~/test-repos/left-pad
```

Now we can run our analyzer:

```
$ r2c run ~/test-repos/left-pad
```

We get the following output:

```
$ r2c run ~/test-repos/left-pad/
ðŞŔČ Running analyzer...
Counting whitespace in file ./test.js
Counting whitespace in file ./perf/es6Repeat.js
Counting whitespace in file ./perf/perf.js
Counting whitespace in file ./perf/O(n).js
Counting whitespace in file ./index.js
{
    "results": [
...
```

Notice how the `print` statements in our Python code appear as if we were running the program outside of r2c. This makes it easy to debug programs normally. The full JSON output of the program should look something like this:

```
1  {
2      "results": [
3          {
4              "check_id": "whitespace",
```

```
            "extra": {
                "num_whitespace": 638,
                "size": 3673
            },
            "path": "./test.js"
        },
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 61,
                "size": 216
            },
            "path": "./perf/es6Repeat.js"
        },
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 213,
                "size": 1442
            },
            "path": "./perf/perf.js"
        },
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 77,
                "size": 241
            },
            "path": "./perf/O(n).js"
        },
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 360,
                "size": 1137
            },
            "path": "./index.js"
        }
    ]
}
```

Cool! Now we can hunt for minified files. All of these files look reasonable; though we don't know exactly what our threshold should be, minified files will probably be less than 5 to 10 percent whitespace. To find minified files in top projects and get a sense of what cut-off point to use, we'll want to run this analyzer at scale against npm packages.

To get started, head on over to Running on r2c.

Running on r2c

Note: The following sections require you to be a member of the r2c beta.

Logging in to r2c

To publish and download analyzers you'll need to log in to the r2c platform.

```
$ r2c login
Please enter your org name, or to use the common r2c platform, press enter [wildwest]:
Opening web browser to get login token. Do you want to continue? [Y/n]:
trying to open https://app.r2c.dev/settings/token in your browser...
Please enter the API token: ...
```

Follow the on-screen instructions, accepting the default `wildwest` org name (which is tied to the beta).

Pushing to r2c

Before uploading our analyzer, let's take a quick look at it to make sure everything is ready. **Once published, analyzer versions can't be unpublished**. For larger analyzers, this would also be the time to run unit and integration tests.

For this tutorial analyzer, we'll just sanity-check the fields in `analyzer.json`:

```
1  {
2      "analyzer_name": "[YOUR GROUP]/minifinder",
3      "author_name": "Jav. A. Script",
4      "author_email": "hello@example.com",
5      "version": "0.0.1",
6      "spec_version": "1.2.0",
7      "dependencies": {
8          "public/source-code": "*"
9      },
10     "type": "commit",
11     "output": {
12         "type": "json"
13     },
14     "deterministic": true
15 }
```

Everything looks mostly good. However, to follow best practices, our analyzer should use Semantic Versioning. As this is our first release, but we're not yet sure if everything is production ready, we should designate this release version `0.1.0`:

```
"version": "0.1.0",
```

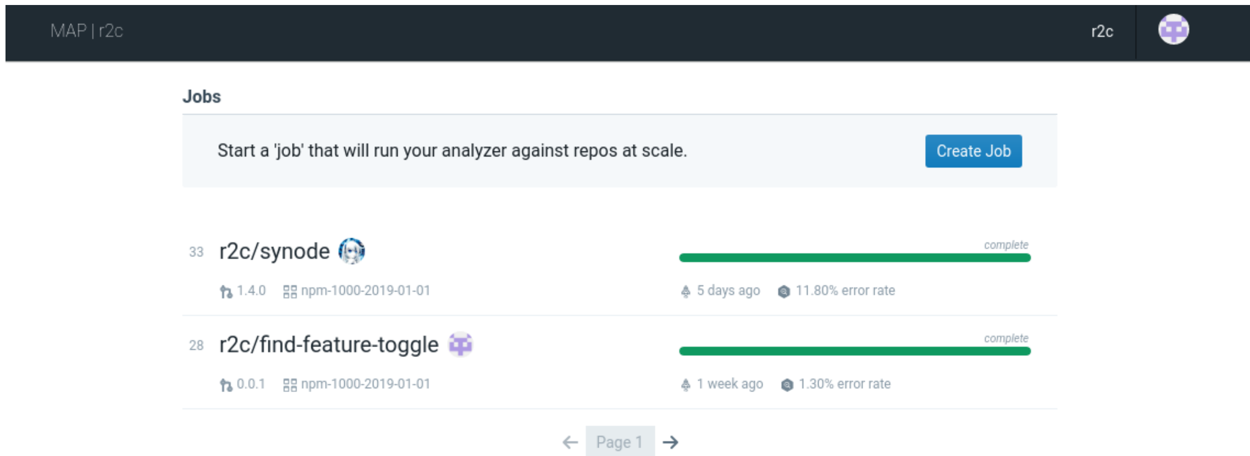
Great! Now we're all set. We can push our analyzer to r2c by running the following command from within the analyzer directory:

```
$ r2c push
```

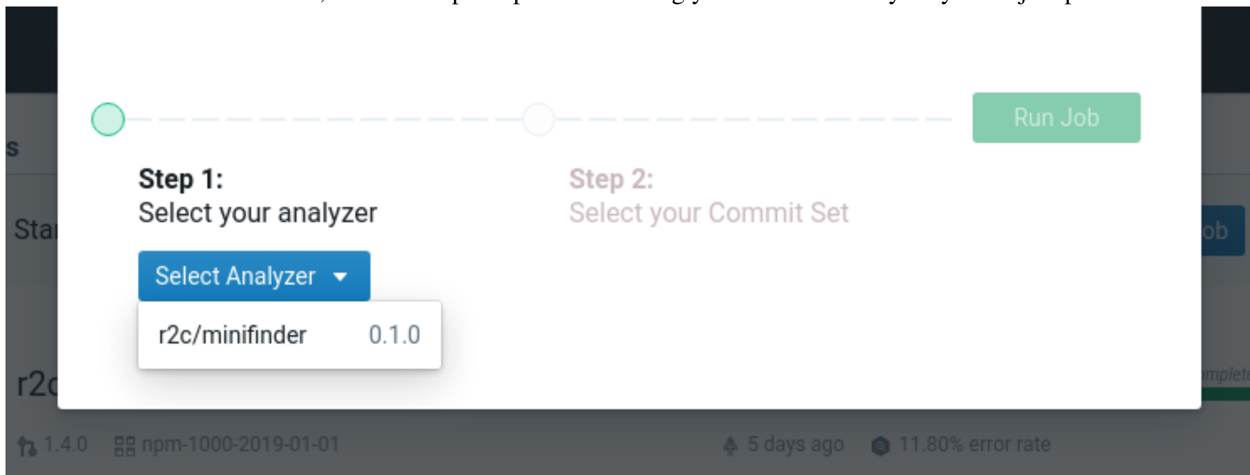
That's it! Head on over to app.r2c.dev to run your analyzer on an npm input set and dive into the results.

Starting the Job

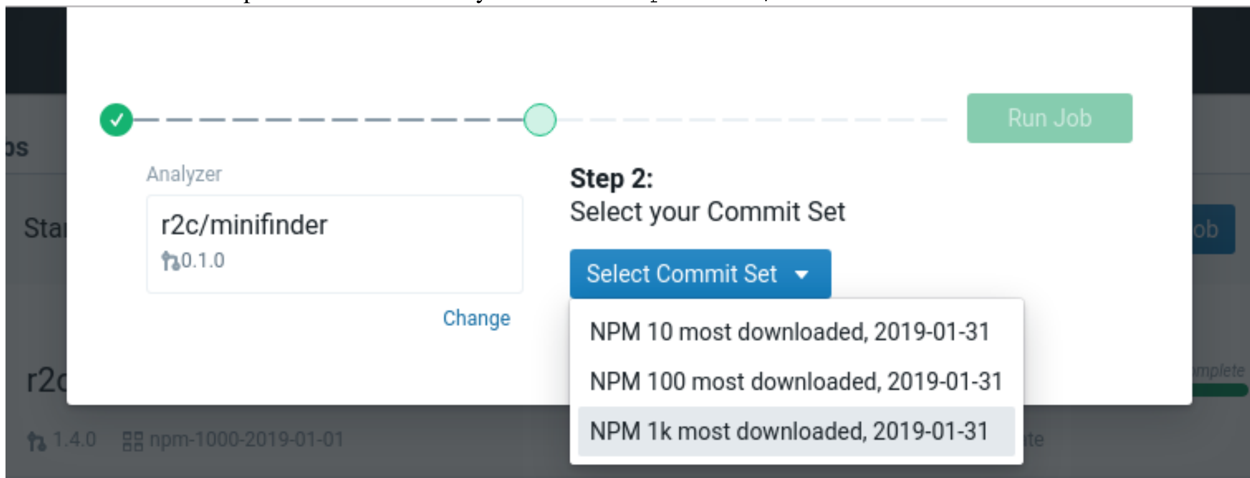
Once you've logged in to the platform you'll be taken to the Jobs page, which will look similar to this:



Click the "Create Job" button, which will pull up a menu letting you select the analyzer you've just pushed!



Now we'll select an input set to run our analyzer on: `r2c npm 1000, 2019-04-01`.



Click "Run Job" to start the analysis! Your job will be added to the jobs list, where you can click on it to see output, console logs, and errors coming in in real time!

See Analyzing Results for next steps.

Note: When a job is first kicked off, the infrastructure may need to "warm up"; idle machines will start processing your job and new machines will be brought online to handle the demand. Once warmed up a job should proceed quickly and you'll get results within just a few minutes for quick analyzers like the tutorial.

Note: We currently limit your container to running for 5 minutes with 1.5 GB of memory. Most analyzers don't hit this limit, however, if you believe you're running into them please let us know and we can adjust them for your research.

Analyzing Results

Note: The following instructions are for analyzers with output type `json`. See analyzer.json to learn more about this and other output types.

Now that your analyzer has been run in the web UI (and completed successfully, we hope!) we'll write some Python to load and graph the results.

When you run an analyzer on the r2c platform we store the results in a PostgreSQL database. To interact with this database, we'll use a mix of sqlalchemy, psycopg2, pandas, and matplotlib Python libraries. Let's install these in your local environment:

```
$ pip3 install sqlalchemy psycopg2-binary pandas matplotlib
```

Next, create a file in a scratch directory called `minifinder-results.py`:

```
$ touch minifinder-results.py
```

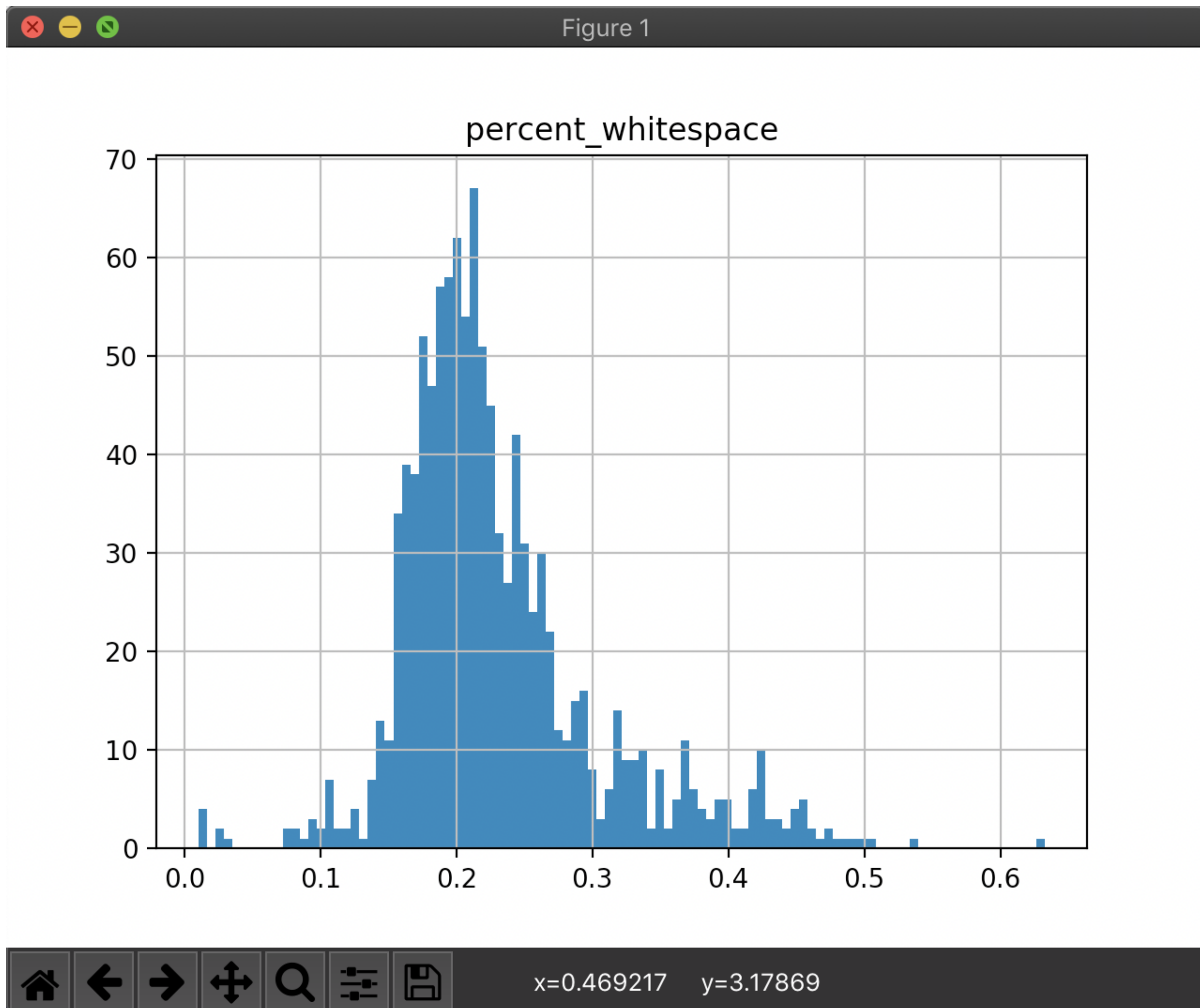
Copy the following code into `minifinder-results.py` and update the constants with your org information:

```
1  import os
2  import pandas as pd
3  import psycopg2
4  from sqlalchemy import create_engine
5  import matplotlib.pyplot as plt
6
7  ###################################
8  # EDIT THESE CONSTANTS
9  ###################################
10
11 DB_PASSWORD = "DB-PASSWORD-FROM-EMAIL"
12 AUTHOR_NAME = "YOUR-NAME"
13
14 ANALYZER_NAME = f"beta/{AUTHOR_NAME}-minifinder"
15 ANALYZER_VERSION = "0.1.0"
16 CORPUS_NAME = "r2c-1000"
17
18 ###################################
19 # END EDIT SECTION
```

```
###################################

# Canonical SQL query to get job-specific results back.
JOB_QUERY = """
SELECT *
FROM   result,
       commit_corpus
WHERE  result.commit_hash = commit_corpus.commit_hash
       AND analyzer_name = %(analyzer_name)s
       AND analyzer_version = %(analyzer_version)s
       AND corpus_name = %(corpus_name)s
"""

QUERY_PARAMS = {
    "corpus_name": CORPUS_NAME,
    "analyzer_name": ANALYZER_NAME,
    "analyzer_version": ANALYZER_VERSION,
}

# Connect to PostgreSQL host and query for job-specific results
engine = create_engine(f"postgresql://notebook_user:{DB_PASSWORD}@db.r2c.dev/postgres")
job_df = pd.read_sql(JOB_QUERY, engine, params=QUERY_PARAMS)

# Print pandas dataframe to stdout for debugging
print("Raw job dataframe:")
print(job_df[1:10])

# Helper method to compute % whitespace from the num_whitespace and size fields in our 'extra' colum
def get_percent_whitespace(row):
    size = row.extra["size"]
    # Avoid 'division by zero' exceptions.
    return row.extra["num_whitespace"] / size if size else 0


# Add 'percent_whitespace' column
job_df["percent_whitespace"] = job_df.apply(get_percent_whitespace, axis=1)

# Create a histogram of our data using the `percent_whitespace` column
job_df.hist(column="percent_whitespace", bins=100)
plt.show()
```

Finally, run `minifinder-results.py`, which will produce a graph of results:

```
$ python3 minifinder-results.py
```



Success! You've gone from a question, "how much of each file in a project is whitespace", to analyzing 1000 npm projects, to graphing the results!

Next up is writing your own analyzer, interrogating your data, and making the JavaScript ecosystem a better place. If you have any questions or concerns, please don't hesitate to reach out to us at .

CHAPTER

TWO

API REFERENCE

The r2c analysis API has three main parts - the `analyzer.json` manifest file which specifies how the analyzer interacts with the rest of the r2c system, the `docker` container in which the analyzer runs and how r2c expects programs in the container to behave, and the final output formats of the analysis that go on to be used by other parts of the r2c toolchain.

`analyzer.json`

The current version of the `analyzer.json` specification is `1.0.0`. This version supports the following fields in the top-level JSON object:

analyzer_name [*string*] The name of the analyzer. All analyzers must belong to an org, e.g. `r2c/transpiler`.

author_name [*string*] The name of the author.

author_email [*string*] A contact address for the author.

version [*string*] The semantic version string representing the current analyzer version.

Note: Analyzer versions are used for caching and result correctness. A given version of a deterministic analyzer run on the same input is expected to always produce the same result. One consequence of this is that you cannot overwrite or push multiple of the same version for a given analyzer.

spec_version [*string*] The version of the analyzer specification this analyzer is compatible with. To match the version of the documented on this page, this should always be `1.0.0`.

dependencies [*object*] This is the main mechanism by which analyzers interact with other analysis components in the r2c system. Each key in this object is the name of another analyzer, such as `r2c/transpiler`. Each value is the version of the analyzer you want to use. An analyzer's output for the same commit or repository appears under its name in `/analysis/inputs` when your container runs, e.g. `/analysis/inputs/r2c/transpiler`.

type [*"constant", "commit", or "git"*] The type of input for which this analyzer produces a unique output.

- A `constant` analyzer always produces the same output. This can be useful for the trained data for a machine learning based analysis, by having the analyzer depend on its trained weights separately from the actual analysis.
- A `commit` analyzer produces output for a JavaScript project at a specific point in its history, a single instance of code. This analyzer should not expect to make use of any version control features. Most analyzers are `commit` anlayzers.
- A `git` analyzer understands the evolution and history of a repository. It produces unique outputs for an entire commit graph or bare git checkout. These analyzers can directly compare trends in a project.

output [*"json", "filesystem", or "both"*] The type of output this analyzer produces.

- `json` output is the most common output format. For the structure of the resulting JSON, see JSON Results.
- `filesystem` output allows the analyzer to output an arbitrary directory and file structure. Hard links are not supported. These analyzers may do such things as transpile code, install dependencies, or produce binary or other data from an analysis that would not be serializable as JSON. They lose the benefit of parts of the r2c toolchain that understand the JSON format, so most analysis should eventually end up as JSON.
- The `both` value indicates that both `json` and `filesystem` output appear together. This is useful for describing properties of the filesystem output to query the results.

deterministic [*boolean*] The current version of the r2c platform does not support non-deterministic analyzers. However, this feature is a part of the analyzer specifciation version `1.0.0` and will be implemneted soon.

Database Schemas

Note: We recommend psql for raw querying and exploration of data or the Python library sqlalchemy for data analysis in environments like Jupyter Notebook. See Analyzing Results for more details and connection strings.

Analyzers with output type `json` (discussed in analyzer.json) are stored in our PostgreSQL database. There are two primary tables:

1. `result`
2. `commit_corpus`

`result`

The schema for the `result` table closely follows the format of JSON Results.

```
      Column       |  Type
-------------------+---------
 id                | integer
 analyzer_name     | text
 analyzer_version  | text
 commit_hash       | text
 check_id          | text
 path              | text
 start_line        | integer
 start_col         | integer
 end_line          | integer
 end_col           | integer
 extra             | jsonb
```

`commit_corpus`

```
   Column    | Type
-------------+------
 commit_hash | text
 repo_url    | text
 corpus_name | text
```

Docker Containers

The Dockerfile for r2c analyzers allows you to customize your operating system, software, and filesystem as much as you like. However, r2c expects a few things of these containers to be able to run analysis, and a few more things for a smooth experience when developing analysis on your local machine.

Filesystem Structure

Dependencies for analyzers are mounted into the container at `/analysis/inputs`. For more on specifying dependencies, see manifest.

All dependency names must also be valid Linux filesystem paths; this enables r2c to mount each dependency in a subfolder matching the org and analyzer name. For example, for the `transpiler` analyzer published by `r2c-cli`, specified in the dependency object as `"r2c/transpiler": "1.0.0"`, its output will appear under `/analysis/inputs/r2c/transpiler`.

This behavior is slightly different for filesystem type analyzers and JSON type analyzers. For filesystem type analyzers, their input appears at that path. If `r2c/transpiler` is of type filesystem and writes a file named `foo.bin` to `/analysis/output/output/`, then when depending on `r2c/tranpsiler`, that file would exist at `/analysis/inputs/r2c/transpiler/foo.bin`. If on the other hand, `r2c/transpiler` is of type JSON and writes its output to `/analysis/output/output.json`, that JSON file would exist at `/analysis/inputs/r2c/transpiler.json`.

Above we allude to `/analysis/output/`; this is the folder where analyzers are expected to write their output. For filesystem type analyzers, the contents of the folder `/analysis/output/output/` will be saved. For JSON type analyzers, the contents of `/analysis/output/output.json` will be saved.

Entry Point Behavior

Analyzers may specify any entry point (default Docker CMD) they wish. By default, the r2c init template sets this to `/analyzer/analyze.sh`. However, this command could instead be a `java -jar` command, a python program, or any other valid command. The only expectation r2c has of this command is that if it exits **with a non-zero return code**, the analysis is considered to have failed and output will not be saved.

User Permissions

It is considered bad practice to run inside of Docker containers as the root user. It is also considered bad practice when running locally to develop r2c analyzers as root. This leads to one complication: files created in the container as the user inside the container may have different user permissions that prevent the local user running r2c from cleaning up the temporary files.

To avoid this, the default r2c template takes arguments that create a user with the same ID as the current user on the host machine. This happens once at build time, and does not affect the ability of the analyzer to run at scale on r2c infrastructure, but avoids files being created that require extra permission to clean up when running locally. If you need to, you may have analysis inside the Docker container run as any user you like, though it is often not good practice to do so.

JSON Results

The JSON results produced by your analysis must conform to the `output.json` specification. As of `1.0.0`, there are two top-level fields: `results` (required) and `errors` (optional):

```
{
    "results": [],
    "errors": []
}
```

results

The `results` key is a required top-level field containing an array of 0 or more `result` objects that specify the type of result and its location (if applicable).

A sample `results` response from our whitepace finding analyzer in *Running Analysis Locally* is:

```
{
    "results":[
        {
            "check_id":"whitespace",
            "path":"perf/O(n).js",
            "extra":{
                "whitespace":77,
                "total":241
            }
        }
    ]
}
```

Each `result` object supports the following fields:

Each `point` object supports the following fields:

errors

Optional field containing an array of 0 or more `error` objects with error messages and related data.

A sample `errors` response looks like:

```
{
    "results":[],
    "errors": [
        {
            "message": "Cyclomatic complexity limit reached.",
            "data": {
                "path": "foobar.js"
            }
        }
    ]
}
```

Each `error` object supports the following fields:

CHAPTER

THREE

BEST PRACTICES

Coming soon!

CHAPTER

FOUR

TROUBLESHOOTING

The /analysis directory is completely empty

If the /analysis directory doesn't even have inputs and output subdirectories, then Docker is probably having issues mounting those folders. To verify, run

```
$ tempdir=$(mktemp -d)
$ touch $tempdir/data
$ docker run --volume $tempdir:/dummy alpine:latest ls -l /dummy
```

You should see a single line of output:

```
total 0
```

If this is the case, then Docker can't mount folders inside /tmp. This can happen if you installed Docker via the snap package manager (i.e., `which docker` shows a path starting with /snap). If you did that, you'll need to run `sudo snap remove docker` and then reinstall Docker via your package manager or a similar method.

If you see two lines of output like

```
total 0
-rw-r--r--    1 root     root             0 Apr 15 21:47 data
```

then the issue is something else. In that case, please contact us and we'll be happy to assist.