Socorro Documentation

Socorro team

Mar 20, 2018

Contents

Socorro is a set of components for collecting, processing, and analyzing crash data. It is used by Mozilla for analyzing crash data for Mozilla products. Mozilla's crash analysis tool is hosted at https://crash-stats.mozilla.com/.

CHAPTER 1

Project info

Free software Mozilla Public License version 2.0

Code https://github.com/mozilla-services/socorro/ and https://github.com/mozilla-services/antenna

Documentation https://socorro.readthedocs.io/

Mailing list https://lists.mozilla.org/listinfo/tools-socorro

IRC irc://irc.mozilla.org/breakpad

Bugs https://bugzilla.mozilla.org/enter_bug.cgi?format=__standard__&product=Socorro

CHAPTER 2

Contents

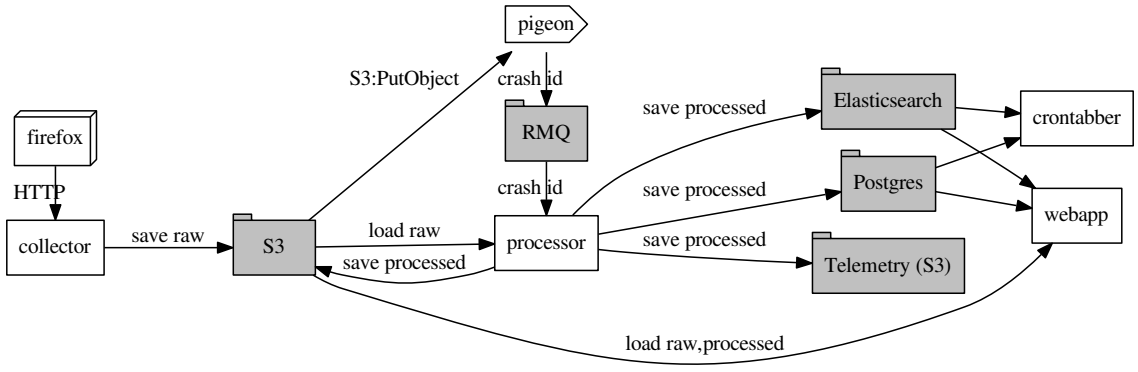
2.1 Overview

2.1.1 What is Socorro?

Socorro is software that implements a crash ingestion pipeline.

The Socorro code is hosted in a GitHub repository at https://github.com/mozilla-services/socorro and released and distributed under the Mozilla Public License v2.

The crash ingestion pipeline that we have at Mozilla looks like this:



Arrows direction represents the flow of interesting information like saving crash information, loading crash information, pulling crash ids from queues, and so on.

Important services in the diagram:

- Collector: Collects incoming crash reports via HTTP POST. The collector we currently use is Antenna now collects crashes for Socorro.
- Processor: Turns breakpad minidump crashes into stack traces and other info.
- Webapp/Crash Stats: Web user interface for analyzing crash data.
- Crontabber: Runs hourly/daily/weekly tasks for analyzing and processing data.

The processor stores crash data in several crash storage destinations:

- S3
- Postgres
- Elasticsearch
- Telemetry (S3)

2.1.2 Repository structure

Top-level folders

If you clone our git repository, you will find the following folders.

Here is what each of them contains:

alembic/ Alembic-managed database migrations.

bin/ Some scripts that should get moved to a more sensible place but haven't, yet.

config/ Configuration for an old way of running Socorro that you can completely ignore.

docker/ Docker environment related scripts, configuration, and other bits.

docs/ Documentation of the Socorro project (you're reading it right now).

e2e-tests/ The Selenium tests for the webapp.

minidump-stackwalk/ The minidump stackwalker program that the processor runs for pulling out information from crash report dumps.

requirements/ Files that hold Python library requirements information.

scripts/ Arbitrary scripts.

socorro/ The bulk of the Socorro source code.

tools/ Some files that should get moved, but haven't, yet.

webapp-django/ The webapp source code.

wsgi/ Another part of the webapp.

2.2 Getting started

This chapter covers getting started with Socorro using Docker for a local development environment.

If you're interested in running Socorro in a server environment, then check out *Deploying Socorro*.

2.2.1 Quickstart

1. Install required software: Docker, docker-compose (1.10+), make, and git.

   Linux:

   Use your package manager.

   OSX:

   Install Docker for Mac which will install Docker and docker-compose.

   Use homebrew to install make and git:

   ```
   $ brew install make git
   ```

   Other:

   Install Docker.

   Install docker-compose. You need 1.10 or higher.

   Install make.

   Install git.

2. Clone the repository so you have a copy on your host machine. Instructions are on GitHub.

3. From the root of this repository, run:

   ```
   $ make dockerbuild
   ```

   That will build the Docker images required for development: processor, webapp, and crontabber.

   Each of these images covers a single Socorro component: processor, webapp, and crontabber.

4. Then you need to set up the Postgres database and Elasticssearch. To do that, run:

   ```
   $ make dockersetup
   ```

   This creates the Postgres database and sets up tables, stored procedures, integrity rules, types, and a bunch of other things. It also adds a bunch of static data to lookup tables.

   For Elasticsearch, it sets up Supersearch fields and the index for raw and processed crash data.

   You can run `make dockersetup` any time you want to wipe the Postgres database and Elasticsearch to pick up changes to those storage systems or reset your environment.

5. Then you need to pull in product release and some other data that makes Socorro go.

   To do that, run:

   ```
   $ make dockerupdatedata
   ```

   This adds data that changes day-to-day. Things like product builds and normalization data.

   Depending on what you're working on, you might want to run this weekly or maybe even daily.

At this point, you should have a basic functional Socorro development environment that has no crash data in it.

See also:

Run the processor and get some crash data! If you need crash data, see *Service: Processor* for additional setup, fetching crash data, and running the processor.

Update your local development environment! See *Updating data in a dev environment* for how to maintain and update your local development environment.

Learn about configuration! See *Configuration* for how configuration works and about `my.env`.

Run the webapp! See *Service: Crash Stats Webapp* for additional setup and running the webapp.

Run crontabber! See *Service: Crontabber* for additional setup and running crontabber.

2.2.2 Updating data in a dev environment

Updating the code

Any time you want to update the code in the repostory, run something like this from the master branch:

```
$ git pull
```

It depends on what you're working on and the state of things.

After you do that, you'll need to update other things.

If there were changes to the requirements files or setup scripts, you'll need to build new images:

```
$ make dockerbuild
```

If there were changes to the database tables, stored procedures, types, migrations, or anything like that, you'll need to wipe the Postgres database and Elasticsearch:

```
$ make dockersetup
```

After doing that, you'll definitely want to update data:

```
$ make dockerupdatedata
```

Wiping crash storage and state

Any time you want to wipe all the crash storage destinations, remove all the data, and reset the state of the system, run:

```
$ make dockersetup
```

Updating release data

Release data and comes from running ftpscraper. After you run ftpscraper, you have to run featured-versions-automatic which will update the featured versions list. Additionally, there's other data that changes day-to-day that you need to pick up in order for some views in the webapp to work well.

Updating that data is done with a single make rule.

Run:

```
$ make dockerupdatedata
```

Note: This can take a long while to run and if you're running it against an existing database, then ftpscraper will "catch up" since the last time it ran which can take a long time, maybe hours.

If you don't have anything in the database that you need, then it might be better to wipe the database and start over.

2.2.3 Configuration

Configuration is pulled from three sources:

1. Envronment variables
2. ENV files located in `/app/docker/config/`. See `docker-compose.yml` for which ENV files are used in which containers, and their precedence.
3. The `config_defaults` attribute for each `SocorroApp` subclass.

The sources above are ordered by precedence, i.e. configuration values defined by environment variables will override values from ENV files or `config_defaults`.

The following ENV files can be found in `/app/docker/config/`:

`local_dev.env` This holds *secrets* and *environment-specific configuration* required to get services to work in a Docker-based local development environment.

This should **NOT** be used for server environments, but you could base configuration for a server environment on this file.

`never_on_a_server.env` This holds a few environment variables that override secure defaults and are explicitly for a local development environment.

These should never show up in a server environment.

`test.env` This holds configuration specific to running the tests. It has some configuration value overrides because the tests are "interesting".

`my.env` This file lets you override any environment variables set in other ENV files as well as set variables that are specific to your instance.

It is your personal file for your specific development environment–it doesn't get checked into version control.

The template for this is in `docker/config/my.env.dist`.

In this way:

1. environmental configuration which covers secrets, hosts, ports, and infrastructure-specific things can be set up for every environment
2. behavioral configuration which covers how the code behaves and which classes it uses is versioned alongside the code making it easy to deploy and revert behavioral changes with the code depending on them
3. `my.env` lets you set configuration specific to your development environment as well as override any configuration and is not checked into version control

Setting configuration specific to your local dev environment

There are some variables you need to set that are specific to your local dev environment. Put them in `my.env`.

Overriding configuration

If you want to override configuration temporarily for your local development environment, put it in `my.env`.

2.3 Contributing

2.3.1 Bugs

All bugs are tracked in https://bugzilla.mozilla.org/.

Write up a new bug:

https://bugzilla.mozilla.org/enter_bug.cgi?product=Socorro

There are multiple components to choose. If in doubt use `General`.

2.3.2 Docker

Everything runs in a Docker container. Thus Socorro requires fewer things to get started and you're guaranteed to have the same setup as everyone else and it solves some other problems, too.

If you're not familiar with Docker and docker-compose, it's worth reading up on.

2.3.3 Python code conventions

All Python code files should have an MPL v2 header at the top:

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
```

Python code should follow PEP-8.

Max line length is 100 characters.

4-space indentation.

To run the linter, do:

```
$ make lint
```

If you hit issues, use `# noqa`.

2.3.4 Javascript code conventions

4-space indentation.

If in doubt, see https://github.com/mozilla-services/socorro/blob/master/.editorconfig

2.3.5 Git conventions

First line is a summary of the commit. It should start with one of the following:

```
Fixes bug nnnnnnn
```

or:

```
Bug nnnnnnn
```

The first, when it lands, will cause the bug to be closed. The second one does not.

After that, the commit should explain *why* the changes are being made and any notes that future readers should know for context or be aware of.

2.3.6 Pull requests

Pull request summary should indicate the bug the pull request addresses.

Pull request descriptions should cover at least some of the following:

1. what is the issue the pull request is addressing?
2. why does this pull request fix the issue?
3. how should a reviewer review the pull request?
4. what did you do to test the changes?
5. any steps-to-reproduce for the reviewer to use to test the changes

2.3.7 Code reviews

Pull requests should be reviewed before merging.

Style nits should be covered by linting as much as possible.

Code reviews should review the changes in the context of the rest of the system.

2.3.8 Python Dependencies

Python dependencies for all parts of Socorro are split between two files:

1. `requirements/default.txt`, containing dependencies that Socorro uses directly.
2. `requirements/constraints.txt`, containing dependencies required by the dependencies in `default.txt` that Socorro does not use directly.

Dependencies in both files must be pinned and hashed. Use hashin.

For example, to add `foobar` version 5:

```
hashin -r requirements/default.txt foobar==5
```

If `foobar` has any dependencies that would also be installed, you must add them to the constraints file:

```
hashin -r requirements/constraints.txt bazzbiff==4.0
```

Then rebuild your docker environment:

```
make dockerbuild
```

If there are problems, it'll tell you.

Note: If you're unsure what dependencies to add to the constraints file, the error from running `make dockerbuild` should include a list of dependencies that were missing, including their version numbers and hashes.

2.3.9 JavaScript Dependencies

Frontend dependencies for the webapp are in `webapp-django/package.json`. They must be pinned and included in package-lock.json.

You can add new dependencies using `npm` (you must use version 5 or higher):

npm install –save-exact foobar@1.0.0

Then rebuild your docker environment:

```
make dockerbuild
```

If there are problems, it'll tell you.

2.3.10 Documentation

Documentation for Socorro is build with Sphinx and is available on ReadTheDocs. API is automatically extracted from docstrings in the code.

To build the docs, run this:

```
$ make docs
```

2.3.11 Running tests

The tests in `socorro/unittests/` use pytest.

The tests in `webapp-django/` use pytest.

To run the tests, do:

```
$ make dockertest
```

That runs the `/app/docker/run_test.sh` script in the webapp container using test configuration.

To run specific tests or specify arguments, you'll want to start a shell in the test container:

```
$ make dockertestshell
```

Then you can run pytest or the webapp tests as you like.

Running all the unittests:

```
app@...:/app$ pytest
```

Running a directory of unittests:

```
app@...:/app$ pytest socorro/unittest/processor/
```

Running a file of unittests:

```
app@...:/app$ pytest socorro/unittest/processor/test_processor_app.py
```

Running webapp tests (make sure you run `./manage.py collectstatic` first):

```
app@...:/app/webapp-django$ ./manage.py test
```

Running a directory of webapp tests:

```
app@...:/app/webapp-django$ ./manage.py test crashstats/home/tests/
```

Running a file of tests:

```
app@...:/app/webapp-django$ ./manage.py test crashstats/home/tests/test_views.py
```

2.3.12 Writing tests

For webapp tests, put them in the `tests/` directory of the appropriate app in `webapp-django/`.

For other tests, put them in `socorro/unittest/`.

Mock usage

Mock is a python library for mocks objects. This allows us to write isolated tests by simulating services beside using the real ones. Best examples is existing tests which admittedly do mocking different depending on the context.

Tip! Try to mock in limited context so that individual tests don't affect other tests. Use context managers and instead of monkey patching imported modules.

2.4 Signature Generation

Contents

2.4.1 Introduction

During processing of a crash, Socorro creates a signature using the signature generation module. Signature generation typically starts with a string based on the stack of the crashing thread. Various rules are applied and after everything is done, we have a Socorro crash signature.

The signature generation code is here:

https://github.com/mozilla-services/socorro/tree/master/socorro/signature

The lists for configuring the C signature generation class are here:

https://github.com/mozilla-services/socorro/tree/master/socorro/siglists

2.4.2 How to request a change to signature generation

To request a change to signature generation:

Write up a bug in the Socorro product and please include the following:

1. explanation of what the problem you want to solve is
2. urls of examples of crashes that have the problem you're trying to solve
3. expected signatures for those crashes

We need this to make sure we can help you make the right changes.

Examples of bugs:

- https://bugzilla.mozilla.org/show_bug.cgi?id=1397926
- https://bugzilla.mozilla.org/show_bug.cgi?id=1402037

2.4.3 How to review a signature generation changes

1. Make sure the PR has a corresponding bug in Bugzilla and references the bug in the commit summary.

   This is important because signature generation is tricky and we need the historical data for what changes we made, for whom, why, and how it affected signature generation.

2. Verify there are no typos in the change.

   We have a unit test that verifies there are no syntax errors in those files, but that (obviously) doesn't cover typos.

3. Run the pull request changes through signature generation using the command line interface in your local dev environment.
4. Verify with the author that the changes occur as intended.
5. Merge the PR and verify the example crashes on -stage.

The easiest way to do that is to use Super Search and search for a signature. The most common change is an addition to the prefix list, in which case you want to search for the frame signature that was added, and verify that in recent signatures there is something following it.

If you don't want to wait for new crash reports to arrive, you can find an existing one and send it to reprocessing. That can be done on the report/index page directly, or via the admin panel.

Note that after a signature change has been pushed to production, you might want to reprocess the affected signatures.

2.4.4 Signature generation module

This Python module covers crash signature generation.

command line interface

This module defines a command line interface for signature generation. Given a crash id, it pulls the raw and processed data from Socorro -prod, generates a signature using the code in this module, and then tells you the original signature and the newly generated one.

This can be used for testing signature generation changes, regression testing, and astounding your friends at parties.

To use:

```
$ python -m socorro.signature CRASHID [CRASHID ...]
```

Pulling crash ids from the file `crashids.txt`:

```
$ cat crashids.txt | python -m socorro.signature
```

Pulling crash ids from another script:

```
$ ./scripts/fetch_crashids.py --num=10 | python -m socorro.signature
```

Spitting output in CSV format to more easily analyze results for generating signatures for multiple crashes:

```
$ cat crashids.txt | python -m socorro.signature --format=csv
```

For more argument help, see:

```
$ python -m socorro.signature --help
```

Note: You need to run this inside a Socorro environment. For example, you could do this:

```
$ docker-compose run processor bash
app@.../app$ python -m socorro.signature --help
```

---

library

This code can sort of be used as a library. It's been decoupled from many of Socorro's bits, but still has some requirements. Roughtly, it requires:

- requests
- socorro.siglists
- socorro.lib.treelib
- ujson

The main class is `socorro.signature.SignatureGenerator`. It takes a pipeline of rules to use to generate signatures.

Rough usage:

```
from socorro.signature import SignatureGenerator

generator = SignatureGenerator()

raw_crash = get_raw_crash_from_somewhere()
processed_crash = get_processed_crash_from_somewhere()

ret = generator.generate(raw_crash, processed_crash)
print(ret['signature'])
```

Note: If you're interested in using this library, write up a bug and let us know the use case and we'll work with you to make it more library-friendly to meet your needs.

2.4.5 Signatures Utilities Lists

This folder contains lists that are used to configure the C signature generation process. Each `.txt` file contains a list of signatures or regex matching signatures, that are used at various steps of our algorithm. Regular expressions use the Python syntax.

Signature Generation Algorithm

When generating a C signature, 5 steps are involved.

1. We walk the crashing thread's stack, looking for things that would match the *Signature Sentinels*. The first matching element, if any, becomes the top of the sub-stack we'll consider going forward.
2. We walk the stack, ignoring everything that matches the *Irrelevant Signatures*. We consider the first non-matching element the top of the new sub-stack.
3. We rewrite every signature missing symbols that matches the *Trim DLL Signatures* to be the module only (the part before the first `@` sign). We also merge them so only one of those frames makes it to the final signature.
4. We accumulate signatures that match the *Prefix Signatures*, until something doesn't match.
5. We normalize each signature we accumulated. Signatures that match the *Signatures With Line Numbers* have their associated code line number added to them, like this: `signature:42`.

The generated signature is a concatenation of all the accumulated signatures, separated with a pipe sign (`|`), and converted to a regular expression.

Signature generation then uses `.match()` to match frames.

Because of that, when changing these lists, make sure you keep the following things in mind:

1. Make sure you're using valid regular expression syntax and escape special characters like `(`, `)`, `.`, and `$`.
2. There's no need to add a trailing `.*` since signature generation uses `.match()` which will match from the beginning of the string.
3. Try to keep it roughly in alphabetical order so as to make it easier to skim through later.

Signature Sentinels

File: `signature_sentinels.txt`

Signature Sentinels are signatures (not regular expression) that should be used as the top of the stack if present. Everything before the first matching signature will be ignored.

The code iterates through the stack frame, throwing away everything it finds until it encounters a match to this regular expression or the end of the stack. If it finds a match, it passes all the frames after the match to the next step. If it finds no match, it passes the whole list of frames to the next step.

A typical line might be `_purecall`.

Irrelevant Signatures

File: `irrelevant_signature_re.txt`

Irrelevant Signatures are regular expressions of signatures that will be ignored while going through the stack. Anything that matches this list will not be added to the overall signature.

A typical rule might be `(Nt|Zw)?WaitForSingleObject(Ex)?`.

Prefix Signatures

File: `prefix_signature_re.txt`

Prefix Signatures are regular expressions of signatures that will be combined with the following frame's signature. Signature generation stops at the first non-matching signature it finds.

A typical rule might be `JSAutoCompartment::JSAutoCompartment.*`.

Note: These are regular expressions. Dollar signs and other regexp characters need to be escaped with a `\`.

Trim DLL Signatures

File: `trim_dll_signature_re.txt`

Trim DLL Signatures are regular expressions of signatures that will be trimmed down to only their module name. For example, if the list contains `foo32\.dll.*` and a stack trace looks like this:

The generated signature will be: `0x0 | foo32.dll | myFavoriteSig()`.

Signatures With Line Numbers

File: `signatures_with_line_numbers_re.txt`

Signatures with line number are regular expressions of signatures that will be combined with their associated source code line numbers.

How to edit these lists

The first thing we will ask you to do is to file a bug. We keep track of every change in Socorro via bugs, so it's important that each commit has one associated to it. File a bug in the Socorro::General component, describe the changes you want to make, and assign it to you.

Then proceed to making those changes...

Using the command line

If you are a git power user, you probably don't need us to explain how to do this! :)

If you are not, you're probably better off using GitHub's interface. Read on!

Using GitHub's interface

First, you need to be logged in to GitHub. Open the file you want to edit, and then click the little pen in the top right corner of the page, the one that says *Fork this project and edit the file*, or *Edit the file in your fork of this project* if you already have a fork of it.

That will take you to an editor, where you can write any changes you want. Once you are done editting the file, enter a commit description. We have some conventions, and a bot that will automatically close bugs, so please make your commit message following this pattern: *Fixes bug XYZ - Desciption of the change*. Once you are ready, click *Propose file change*.

That will create a branch in your fork of the socorro project, and take you to the commit you just created. You can verify that the changes you made are correct, and then click *Create pull request*, and then *Create pull request* again. Once the pull request is opened, Circle CI will automatically start running our test suite, which includes sanity checks for those signature lists. You can see the status of those tests in the pull request, and click the *Details* link to see logs in case of a failure.

That's it! You have proposed a change, we have been notified about it. Someone from the Socorro team will review your changes and merge them if they are appropriate. Thank you for contributing to Socorro!

Watching only the siglists folder

If you are interested in watching what's changing in the `siglists` directory in the repository, but don't care much about what happens in the rest of the Socorro repo, you can easily set a filter in your email client to do that. Here's an example filter you can use today:

2.5 Top Crashers By Signature

2.5.1 Introduction

Topcrashers By Signature compiles the 14 days' worth of crash reports (organized by signature) for a given version. This report is useful for finding new topcrashes, determining if topcrashes have been filed, and seeing trending of topcrashes over time (for a specific version).

2.5.2 Details

For the ideal topcrashers by signature report, we want to gather the following data:

- crashes by version (e.g., Firefox 3.0.9)
- date a crash occurred (to know if it's within our window)
- stack signature
- average uptime (since last browser start) averaged over window
- bug numbers related to crash signature

Additionally, we need the ability to either a) go back in time or b) "freeze" the topcrashers by signature report on a specific day. This allows us to compare, say, the last day of a release to the newest release (e.g., Firefox 3.0.8 to Firefox 3.0.9). Without the ability to go back to a specific day of topcrash reports or freeze topcrash reports, we have no easy ability to compare releases (as new crashes come in for old releases, the topcrash list changes substantially).

Ideal Outputs

(to be filled)

See [[SocorroUIInstallation]] for additional details.

2.5.3 Operations

- Need a recalculation every 4 to 6 hours
- Need top 500 signatures, ranked over last 14 days
- Note that this implies for the database that each slice is aggregated from the full window (which slides forward each time)

2.6 Service: Collector

The collector's job is to collect incoming crash reports, generate crash ids, and save that data to S3 as soon as possible.

"Antenna" is the name of the collector that we're using now.

For more information on Antenna, see the Antenna docs.

2.7 Service: Crontabber

Crontabber is a project that manages scheduled tasks. Unlike traditional UNIX crontab, all execution is done via the crontabber script and the configuration about frequency and exact time to run is part of the configuration files.

The configuration is done using `configman` and can be specified in a `.ini` file or in the process environment. An example looks like this:

```
# name: jobs
# doc: List of jobs and their frequency separated by `|`
# converter: configman.converters.class_list_converter
jobs=socorro.cron.jobs.foo.FooCronApp|12h
     socorro.cron.jobs.bar.BarCronApp|1d
     socorro.cron.jobs.pgjob.PGCronApp|1d|03:00
```

The default jobs specification lives in `socorro/cron/crontabber_app.py` as `DEFAULT_JOBS`.

Different server environments use different jobs specifications based on `DEFAULT_JOBS`.

2.7.1 What runs crontabber?

In our current infrastructure, `crontabber` is run by `crontab` with a spec like this:

```
*/5 * * * * PYTHONPATH="..." crontabber
```

Every 5 minutes, `crontabber` runs, updates crontabber jobs bookkeeping, checks which jobs need to run, and runs those jobs.

In our new infrastructure, `crontabber` runs, then sleeps for 5 minutes, then runs again. This is different than running every 5 minutes. Amongst other things, we're guaranteed to only have one `crontabber` process running on a node.

2.7.2 Crontabber theory

Crontabber runs a set of jobs.

A job specification includes the class to run, a frequency, and optionally a specific time to run at. In this way, we can specify jobs to run weekly, daily, hourly, daily at a specific time, and so on.

Crontabber maintains some bookkeeping for each job including when the job was first run, most recently run, the time of the last success, the time of the last failure, and the next run. If the job failed, it logs some error information.

Jobs can have zero or more dependencies on other jobs. Crontabber makes sure that dependencies are filled before running a job. For example, if `FooCronApp` *depends* on `BarCronApp` it just won't run if `BarCronApp` last resulted in an error or simply hasn't been run the last time it should.

Crontabber has several command line arguments that let you override the job spec to run things manually. For example, you can override dependencies for a job with the `--force` parameter like this:

```
./crontabber --job=BarCronApp --force
```

Dependencies inside the cron apps are defined by settings a class attribute on the cron app. The attribute is called `depends_on` and its value can be a string, a tuple or a list. In this example, since `BarCronApp` depends on `FooCronApp` it's class would look something like this:

```
from crontabber.base import BaseCronApp

class BarCronApp(BaseCronApp):
    app_name = 'BarCronApp'
    app_description = 'Does some bar things'
    depends_on = ('FooCronApp',)
```

(continues on next page)

(continued from previous page)

```
    def run(self):
        ...
```

Raising an error inside a cron app **will not stop the other jobs** from running other than the those that depend on it.

2.7.3 App names and class names

Every cron app in `crontabber` must have a class attribute called `app_name`. This value must be unique. If you like, it can be the same as the class it's in. When you list jobs you **list the full path to the class** but it's the `app_name` within the found class that gets remembered.

If you change the `app_name` all previously know information about it being run is lost. If you change the name and path of the class, the only other thing you need to change is the configuration that refers to it.

Best practice recommendation is this:

- Name the class like a typical Python class, i.e. capitalize and optionally camel case the rest. For example: `UpdateADUCronApp`
- Optional but good practice is to keep the suffix `CronApp` to the class name.
- Make the `app_name` value lower case and replace spaces with `-`.

2.7.4 Automatic backfilling

`crontabber` supports automatic backfilling for cron apps that need a date (it's a python `datetime.datetime` instance) parameter which, if all is well, defaults to the date right now.

To use backfilling your cron app needs to subclass another class. Basic example:

```
from socorro.cron.base import BaseBackfillCronApp

class ThumbnailMoverCronApp(BaseBackfillCronApp):
    app_name = 'thumbnail-mover'
    app_version = 1.0
    app_description = 'moves thumbnails into /dev/null'

    def run(self, date):
        dir_ = '/some/path/' + date.strftime('%Y%m%d-%H%M%S')
        shutil.rmtree(dir_)
```

There's also a specific subclass for use with Postgres that uses backfill:

```
from socorro.cron.base import PostgresBackfillCronApp

class ThumbnailUpdaterCronApp(PostgresBackfillCronApp):
    app_name = 'thumbnail-updater'
    app_version = 1.0
    app_description = 'marks thumbnails as moved'

    def run(self, connection, date):
        sql = """UPDATE thumbnails
        SET removed=true
        WHERE upload_date=%s
        """
```

(continues on next page)

(continued from previous page)

```
        cursor = connection.cursor()
        cursor.execute(sql, date)
```

These cron apps are automatically backfilled because whenever they wake up to run, they compare when it was last run with when it was last successful. By also knowing the frequency it's easy to work out how many times it's "behind". So, for example, if a job has a frequency of 1 day; today is Friday and the last successful run was Monday four days ago. That means, it needs to re-run the `run(connection, date)` method four times. One for Tuesday, one for Wednesday, one for Thursday and one for today Friday. If, it fails still the same thing will be repeated and re-tried the next day but with one more date to re-run.

When backfilling across, say, three failed attempts. If the first of those three fail, the `last_success` date is moved forward accordingly.

2.7.5 Troubleshooting

Examining the last error

All errors that happen are reported to the standard python `logging` module. Also, the latest error (type, value and traceback) is stored in the JSON database too. If any of your cron apps have an error you can see it with:

```
python socorro/cron/crontabber_app.py --list-jobs
```

Here's a sample output:

```
=== JOB ===========================================================================
Class:       socorro.cron.jobs.foo.FooCronApp
App name:    foo
Frequency:   12h
Last run:    2012-04-05 14:49:56  (1 minute ago)
Next run:    2012-04-06 02:49:56  (in 11 hours, 58 minutes)

=== JOB ===========================================================================
Class:       socorro.cron.jobs.bar.BarCronApp
App name:    bar
Frequency:   1d
Last run:    2012-04-05 14:49:56  (1 minute ago)
Next run:    2012-04-06 14:49:56  (in 23 hours, 58 minutes)
Error!!      (1 times)
  File "socorro/cron/crontabber_app.py", line 316, in run_one
    self._run_job(job_class)
  File "socorro/cron/crontabber_app.py", line 369, in _run_job
    instance.main()
  File "/Use[snip]orro/socorro/cron/crontabber_app.py", line 47, in main
    self.run()
  File "/Use[snip]orro/socorro/cron/jobs/bar.py", line 10, in run
    raise NameError('doesnotexist')
```

It will only keep the latest error but it will include an error count that tells you how many times it has tried and failed. The error count increments every time **any** error happens and is reset once no error happens. So, only the latest error is kept and to find out about past error you have to inspect the log files.

Note: If a cron app that is configured to run every 2 days runs into an error, it will try to run again in 2 days.

Running a job manually

Suppose you inspect the error and write a fix. If you're impatient and don't want to wait till it's time to run again, you can start it again like this:

```
python socorro/cron/crontabber_app.py --job=my-app-name
```

This will attempt it again and no matter if it works or errors it will pick up the frequency from the configuration and update what time it will run next.

Resetting a job

If you want to pretend that a job has never run before you can use the `--reset` switch. It expects the name of the app. Like this:

```
python socorro/cron/crontabber_app.py --reset=my-app-name
```

That's going to wipe that job out of the state database rendering basically as if it's never run before. That can make this tool useful for bootstrapping new apps that don't work on the first run or you know what you're doing and you just want it to start afresh.

Figuring out configuration parameters

Best way to figure out the keys for configuration parameters is by running crontabber and telling it to list the jobs. It'll spit out all the configuration keys at startup.

2.7.6 Scheduling jobs

The format for configuring jobs looks like this:

```
socorro.cron.jobs.bar.BarCronApp|30m
```

or like this:

```
socorro.cron.jobs.pgjob.PGCronApp|2d|03:00
```

Hopefully the format is self-explanatory. The first number is required and it must be a number followed by "y" (years), "d" (days), "h" (hours), or "m" (minutes).

For jobs that have a frequency longer than 24 hours you can specify exactly when it should run. This format has to be in the 24-hour format of `HH:MM`.

If you're ever uncertain that your recent changes to the configuration file is correct or not, instead of waiting around you can check it with:

```
python socorro/cron/crontabber_app.py --configtest
```

which will do nothing if all is OK.

2.7.7 Timezone and UTC

All dates and times are in UTC. All Python `datetime.datetime` instances as non-native meaning they have a `tzinfo` value which is set to `UTC`.

This means that if you're an IT or ops person configuring a job to run at 01:00 it's actually at 7pm pacific time.

2.7.8 Writing cron apps (aka. jobs)

First off, if you can implement whatever you're implementing as something other than a crontabber job, do that. If not, proceed.

Code for crontabber jobs goes in `socorro/cron/jobs/`.

Make sure to write tests for them if you can.

2.7.9 Testing crontabber jobs manually

We have unit tests for crontabber jobs (located in: socorro/cron/jobs), but sometimes it is helpful to test these jobs locally before deploying changes.

For "backfill-based" jobs, you will need to reset them to run them immediately rather than waiting for the next available time period for running them.

Example:

```
$ python socorro/cron/crontabber_app.py --reset-job=ftpscraper
```

Then you can run them:

```
$ python socorro/cron/crontabber_app.py --job=ftpscraper
```

2.8 Service: Processor

Note: We're in the process of extracting the processor out of Socorro as a separate project. There's no ETA for that work, yet.

2.8.1 Running the processor

To run the processor using the processor configuration, do:

```
$ docker-compose up processor
```

That will bring up all the services the processor requires to run and start the processor using the `/app/docker/run_processor.sh` script and the processor configuration.

To ease debugging in the container, you can run a shell:

```
$ docker-compose run processor bash
```

Then you can start and stop the processor and tweak files and all that jazz.

Note: The stackwalk binaries are in `/stackwalk` in the container.

2.8.2 Processing crashes

Running the processor is pretty uninteresting since it'll just sit there until you give it something to process.

In order to process something, you first need to acquire raw crash data, put the data in the S3 container in the appropriate place, then you need to add the crash id to the "socorro.normal" RabbitMQ queue.

We have helper scripts for these steps.

fetch_crashids

This will generate a list of crash ids from crash-stats.mozilla.com that meet specified criteria. Crash ids are printed to stdout, so you can use this in conjunction with other scripts or redirect to a file.

This pulls 100 crash ids from yesterday for Firefox product:

```
$ docker/as_me.sh ./socorro-cmd fetch_crashids
```

This pulls 5 crash ids from 2017-09-01:

```
$ docker/as_me.sh ./socorro-cmd fetch_crashids --num=5 --date=2017-09-01
```

This pulls 100 crash ids for criteria specified with a Super Search url that we copy and pasted:

```
$ docker/as_me.sh ./socorro-cmd fetch_crashids "--url=https://crash-stats.mozilla.com/
↪search/?product=Firefox&date=%3E%3D2017-09-05T15%3A09%3A00.000Z&date=%3C2017-09-
↪12T15%3A09%3A00.000Z&_sort=-date&_facets=signature&_columns=date&_columns=signature&
↪_columns=product&_columns=version&_columns=build_id&_columns=platform"
```

You can get command help:

```
$ docker/as_me.sh ./socorro-cmd fetch_crash_data --help
```

fetch_crash_data

This will fetch raw crash data from crash-stats.mozilla.com and save it in the appropriate directory structure rooted at outputdir.

Usage from host:

```
$ docker/as_me.sh ./socorro-cmd fetch_crash_data <outputdir> <crashid> [<crashid> ...]
```

For example (assumes this crash exists):

```
$ docker/as_me.sh ./socorro-cmd fetch_crash_data ./testdata 5c9cecba-75dc-435f-b9d0-
↪289a50170818
```

Use with `fetch_crashids` to fetch crash data from 100 crashes from yesterday for Firefox:

```
$ docker/as_me.sh bash
app@processor:/app$ ./socorro-cmd fetch_crashids | socorro-cmd fetch_crash_data ./
↪testdata
```

You can get command help:

```
$ docker/as_me.sh ./socorro-cmd fetch_crash_data --help
```

You should run this with `docker/as_me.sh` so that the files that get saved to the file system are owned by the user/group of the account you're using on your host.

Note: This script requires that you have a valid API token from the crash-stats.mozilla.com environment that has the "View Raw Dumps" permission in order to download personally identifiable information and dumps.

You can generate API tokens at https://crash-stats.mozilla.com/api/tokens/.

Add the API token value to your `my.env` file:

```
SOCORRO_API_TOKEN=apitokenhere
```

If you don't have an API token, this will download some raw crash information, but it will be redacted.

Note: Make sure you treat any data you pull from production in accordance with our data policies that you agreed to when granted access to it.

---

scripts/socorro_aws_s3.sh

This script is a convenience wrapper around the aws cli s3 subcommand that uses Socorro environment variables to set the credentials and endpoint.

Usage from host:

```
$ docker/as_me.sh ./scripts/socorro_aws_s3.sh <s3cmd> ...
```

For example, this creates an S3 bucket named `dev_bucket`:

```
$ docker/as_me.sh ./scripts/socorro_aws_s3.sh mb s3://dev_bucket/
```

This copies the contents of `./testdata` into the `dev_bucket`:

```
$ docker/as_me.sh ./scripts/socorro_aws_s3.sh sync ./testdata s3://dev_bucket/
```

This lists the contents of the bucket:

```
$ docker/as_me.sh ./scripts/socorro_aws_s3.sh ls s3://dev_bucket/
```

Since this is just a wrapper, you can get help:

```
$ docker/as_me.sh ./scripts/socorro_aws_s3.sh help
```

add_crashid_to_queue

This script adds crash ids to the specified queue. Typically, you want to add crash ids to the `socorro.normal` queue, but if you're testing priority processing you'd use `socorro.priority`.

Usage from host:

```
$ ./docker/as_me.sh ./socorro-cmd add_crashid_to_queue <queue> <crashid> [<crashid> ..
↪.]
```

For example:

```
$ ./docker/as_me.sh ./socorro-cmd add_crashid_to_queue socorro.normal 5c9cecba-75dc-
↪435f-b9d0-289a50170818
```

Note: Processing will fail unless the crash data is in the S3 container first!

Example using all the scripts

Let's process crashes for Firefox from yesterday. We'd do this:

```
# Start bash in the processor container as me
$ docker/as_me.sh bash

# Generate a file of crashids--one per line
you@processor:/app$ socorro-cmd fetch_crashids > crashids.txt

# Pull raw crash data from -prod for each crash id and put it in the
# "crashdata" directory on the host
you@processor:/app$ cat crashids.txt | socorro-cmd fetch_crash_data ./crashdata

# Create a dev_bucket in localstack-s3
you@processor:/app$ ./scripts/socorro_aws_s3.sh mb s3://dev_bucket/

# Copy that data from the host into the localstack-s3 container
you@processor:/app$ scripts/socorro_aws_s3.sh sync ./crashdata s3://dev_bucket/

# Add all the crash ids to the queue
you@processor:/app$ cat crashids.txt | socorro-cmd add_crashid_to_queue socorro.normal

# Then exit the container
you@processor:/app$ exit

# Run the processor to process all those crashes
$ docker-compose up processor
```

Note: That's a lot of commands. Definitely worth writing shell scripts to automate this for your specific needs.

2.8.3 Processing crashes from Antenna

Antenna is the collector of the Socorro crash ingestion pipeline. It was originally part of the Socorro repository, but we extracted and rewrote it and now it lives in its own repository and infrastructure.

Antenna deployments are based on images pushed to Docker Hub.

To run Antenna in the Socorro local dev environment, do:

```
$ docker-compose up antenna
```

It will listen on `http://localhost:8888/` for incoming crashes from a breakpad crash reporter. It will save crash data to the `dev_bucket` in the local S3 which is where the processor looks for it.

FIXME(willkg): How to get crash ids into the processing queue?

2.9 Service: Crash Stats Webapp

2.9.1 Running the webapp

To run the webapp using the webapp configuration, do:

```
$ docker-compose up webapp
```

That will bring up all the services the webapp requires to run and start the webapp using the `/app/docker/run_webapp.sh` script using the webapp configuration.

To ease debugging, you can run a shell in the container:

```
$ docker-compose run webapp /bin/bash
```

Then you can start and stop the webapp, adjust files, and debug.

Note: The `STATIC_ROOT` is set to `/tmp/crashstats-static/` rather than `/app/webapp-django/static`. This alleviates permissions-related problems because the process in the container runs as uid 10001 which is not the uid of the user you're using on your host computer.

The problem this creates is that `/tmp/crashstats-static/` is ephemeral and any changes there disappear when you stop the container.

If you want it persisted, you should mount that directory using `volumes` in a `docker-compose.override.yml` file.

https://docs.docker.com/compose/extends/

2.9.2 Setting up authentication and a superuser

Creating a superuser

If you want to do anything in the webapp admin, you'll need to create a superuser.

Run this:

```
$ docker-compose run webapp python webapp-django/manage.py makesuperuser␣
↪email@example.com
```

You can do this as many times as you like.

Setting up the webapp for Google Sign-In

In order to authenticate in the webapp using the "Google signin" button, you need to get a set of oauth credentials.

1. Go to https://console.developers.google.com/apis/credentials
2. Create a project
3. Set up credentials:
   - authorized js origin: `http://localhost:8000`
   - callback url: `http://localhost:8000/oauth2/signin/`

Note: There is no trailing slash on the JS origin, but there is a trailing slash on the callback url.

4. Open `my.env` in an editor and change these lines:

```
OAUTH2_CLIENT_ID=<client id>
OAUTH2_CLIENT_SECRET=<secret>
```

   where `<client id>` is the oauth client id and `<secret>` is the oauth client secret that you got from step 3

5. If you see a "Signed in to Google, but unable to sign in on the server." with a 403 CSRF error, mark the callback url as csrf_exempt e.g. with this patch

After that, Google Sign-In should work.

2.9.3 About Permissions, User and Groups

Accessing certain parts of the webapp requires that the user is signed-in and also in a group that contains the required permissions.

A permission consists of a code name, a verbose name and a content type (aka Django model) that it belongs to. For business logic specific permissions we use in Socorro to guard certain data that is not a Django model we use permissions that belong to a blank content type. These are the permissions that appear in the list when you visit the Your Permissions page.

2.9.4 Administering Users and Groups

To see the admin user interface, you need to be a superuser. This is not dependent on any permissions. Being a superuser means you have root-like access to everything. Any permission check against a superuser user will always return true.

You can create groups freely in the Admin section and you can attach any permissions to the groups.

You can not give a specific user a specific permission, or combination of permissions. Instead you have to solve this by creating, potentially multiple, groups and attach those accordingly to the user you want to affect.

2.9.5 Extending permissions

All current custom permissions we use are defined in the constants at the top of `webapp-django/crashstats/crashstats/management/__init__.py`. That file also defines some default groups.

This file is executed when you run:

```
cd webapp-django
export SECRET_KEY="..."
./manage.py migrate auth
./manage.py migrate
```

This is run on every deploy because it's idempotent it can be run repeatedly without creating any duplicates.

Note: Removing a permission for this file (assuming you know it's never referenced anywhere else) will **not** delete it from the database. This will require special database manipulation.

To add a new permission for something else, extend that above mentioned file as per how it's currently layed out. You'll need to come up with a code name and a verbose name. For example, a permission for being allowed to save searches could be:

code name save_search

verbose name Save User Searches

Then, once that's added to the file, run `./manage.py migrate` and it will be ready to depend on in the code.

Here's how you might use this permission in a view:

```
def save_search(request):
    if not request.user.has_perm('crashstats.save_search'):
        return http.HttpResponseForbidden('Not allowed!')
```

Note the added `crashstats.` prefix added to the code name when using the `user.has_perm()` function.

Here's an example in a template:

```
{% if request.user.has_perm('crashstats.save_search') %}
  <form action="{{ url('crashstats:save_search') }}" method="post">
    <button>Save this search</button>
  </form>
{% endif %}
```

When you add a new permission here they will automatically appear on the Your Permissions page.

2.9.6 Troubleshooting

If you have set up your webapp but you can't sign in, it could very well be because some configuration is wrong compared to how you're running the webapp.

If this is the problem go to `http://localhost:8000/_debug_login`.

This works for both production and development. If you're running in production you might not be using `localhost:8000` so all you need to remember is to go to `/_debug_login` on whichever domain you will use in production.

If web services are not starting up, `/var/log/nginx/` is a good place to look.

If you are not able to log in to the crash-stats UI, try hitting `http://crash-stats/_debug_login`

If you are having problems with crontabber jobs, this page shows you the state of the dependencies: `http://crash-stats/crontabber-state/`

If you're seeing "Internal Server Error", you can get Django to send you email with stack traces by adding this to `/data/socorro/webapp-django/crashstats/settings/base.py`:

```
# Recipients of traceback emails and other notifications.
ADMINS = (
    ('Your Name', 'your_email@domain.com'),
)
MANAGERS = ADMINS
```

2.9.7 Running Web App in a Prod-like Way

When you run `docker-compose up webapp` in the local development environment, it starts the web app using Django's `runserver` command. `DEBUG=True` is set in the `docker/config/never_on_a_server.env` file, so static assets are automatically served from within the individual Django apps rather than serving the minified and concatenated static assets you'd get in a production-like environment.

If you want to run the web app in a more "prod-like manner", you want to run the webapp using `uwsgi` and with `DEBUG=False`. Here's how you do that.

First start a `bash` shell with service ports:

```
$ docker-compose run --service-ports webapp bash
```

Then compile the static assets:

```
app@...:/app$ cd webapp-django/
app@...:/app/webapp-django$ ./manage.py collectstatic --noinput
app@...:/app/webapp-django$ cd ..
```

Now run the webapp with `uwsgi` and `DEBUG=False`:

```
app@...:/app$ DEBUG=False bash docker/run_webapp.sh
```

You will now be able to open `http://localhost:8000` on the host and if you view the source you see that the minified and concatenated static assets are served instead.

Because static assets are compiled, if you change JS or CSS files, you'll need to re-run `./manage.py collectstatic`.

2.10 Crash storage: API and Implementations

Documentation of our CrashStorage API. This attempts to provide a complete picture of all the crash storage classes that are provided by Socorro.

Base class implemented in `socorro/external/crashstorage_base.py`

These are our base classes for all crash storage for Socorro.

Warning: August 17th, 2017: These docs are outdated.

2.10.1 socorro.external.crashstorage_base

base class that defines the crash storage API. You implement this when you want to plug into any of the Socorro backend components

Base class:

- *CrashStorageBase*: Defines `save_raw_and_processed()`, `get_raw()`, etc.

Concrete implementation:

- *NullCrashStorage*: Silently ignores everything it is told to do.

Examples of other concrete implementations are: *PostgreSQLCrashStorage*, *BotoCrashStorage*.

CrashStorage containers for aggregating multiple crash storage implementations:

- *PolyCrashStorage*: Container for other crash storage systems.
- *FallbackCrashStorage*: Container for two other crash storage systems, a primary and a secondary. Attempts on the primary, if it fails it will fallback to the secondary. In use when we have cutover between data stores. Can be heterogeneous, example: S3 + filesystem and use crashmovers to move from filesystem into S3 when S3 comes back.

- *PrimaryDeferredStorage*: Container for two different storage systems and a predicate function. If predicate is false, store in primary, otherwise store in secondary. Usecase: situation where we want crashes to be put somewhere else and not be processed.
- *PrimaryDeferredProcessedStorage*: Container for a PrimaryDeferredStorage, but there's a third separate storage for Processed crashes. Example: could fork on Product.

Helper for PolyCrashStore:

- *PolyCrashStorageError*: Exception for *PolyCrashStorage*.

How we use these:

We use *CrashStorageBase* in our `socorro/external` crash storage implementations. We use *PolyCrashStorage* (and related containers) as a way to fork "streams of crashes" into different storage engines. Also, the *CrashStorage* containers can contain each other!

TODO: Add an attribute to or rename the CrashStorage containers.

2.10.2 socorro.database.transaction_executor

- *TransactionExecutor*: A functor; a default version of a transaction function that contains a commit, rollback depending on whether a transaction succeeds or fails.
- *TransactionExecutorWithInfiniteBackoff* - will retry a transaction forever as long as the failure is a retriable failure. The failures which are retriable are defined in 'operational_exceptions' - in the implementation of *ConnectionContext* for any *CrashStorageBase* class. Also have 'conditional_exceptions', where some exceptions are retriable and others are not and we have a function `is_operational_exception` to test the contents of the exception string passed back to determine whether or not we really want to retry. `wait_log_interval` is the configured value for notifying the logger that the backoff system is sleeping, rather than just silently waiting.
- *TransactionExecutorWithLimitedBackoff* - Redefines the `backoff_generator()` to stop after the last emitted `backoff_delays` list item.

TODO: Move this to socorro.external

2.10.3 connection_context

This is just duck-typed, so we don't have a base class, currently.

2.10.4 About crash storage implementations

In each of our crash storage implementations, we create: (Found in: `socorro/external` directory tree.)

- *crash_data*: implementation of middleware service.
- *crash*: implementation of middleware service.
- *crashstorage*: a fully abstracted method of saving and retrieving crashes. An implementation within a external resource directory.
- *connection_context*: a connection Factory in the form of a Functor that returns thinly wrapped connections to the resource.

Reasons we have `connection_context`:

- wrapper for use with `with`
- pooled connection context - connections held on to, doesn't log out

- to make threading easier to manage

Below we describe the various implementations used by Socorro to store crashes.

This section should help answer these questions:

- What is this class implementing?
- What was the intended use case for the class?
- Which classes may be used together with which Socorro backend apps?

2.10.5 socorro.external.fs

socorro.external.fs.crashstorage

Implements Radix Tree storage of crashes in a filesystem. Regular and Legacy classes need to be used like-with-like, but Dated and non-dated classes should be compatible.

If you need on-disk queue for crashmovers or processors, use the Dated variety of the classes. If you are starting fresh, use the non-Legacy modules.

Use cases:

- For Mozilla use by the collectors.
- For other users, you can use this class as your primary storage instead of S3. Be sure to implement this in collectors, crashmovers, processors and middleware (depending on which components you use in your configuration).

Note: Because of the slowness of deleting directories created by on-disk, non-SSD storage, the collectors do not unlink directories over time. For many environments, you will need to periodically unlink directories, possibly by entirely wiping out partitions, rather than using *find* or some other UNIX utility to delete.

Classes:

- *FSRadixTreeStorage* - Doesn't have a queueing mechanism. Processors can use these for local storage that doesn't require any knowledge of queueing.
- *FSDatedRadixTreeStorage* - Use in-filesystem queueing techniques so that we know which crashes are new.
- *FSLegacyRadixTreeStorage* - Doesn't have a queueing mechanism. Processors can use these for local storage that doesn't require any knowledge of queueing. Backwards compatible with *socorro.external.filesystem* (aka the 2009 system).
- *FSLegacyDatedRadixTreeStorage* - In production use on collectors. Use in-filesystem queueing techniques so that we know which crashes are new. Backwards compatible with *socorro.external.filesystem* (aka the 2009 system).

2.10.6 socorro.external.postgresql

socorro.external.postgresql.crashstorage

- *PostgreSQLCrashStorage*: In Production. *reports* table mapping is a member of the class. Needs to be kept in sync with reports schema. For use with a processed crash

socorro.external.postgresql.connection_context

- *ConnectionContext*: In Production.
- *ConnectionContextPooled*: not in use because we use pgbouncer. Is threadsafe.

psycopg2 implements all the "connection" semantics we need, so we do not implement the thin wrapper that `socorro.external.hb` and `socorro.external.rabbitmq` have.

socorro.external.postgresql.dbapi2_util

A set of utilities for wrapping psycopg2 and designed to be handed to Transactions.

- *single_value_sql*: Give an SQL statement and receive a single value from a single column.
- *single_row_sql*: Give an SQL statement and receive a single row.
- *execute_query_iter*: Wraps a cursor in an interator.
- *execute_query_fetchall*: Returns a list of tuples.
- *execute_no_results*: Executes something you know won't return results.

socorro.external.postgresql.setupdb_app

This is used by the *Makefile* and `build.sh` to create a test database from scratch.

socorro.external.postgresql.models

These contain our canonical schema definitions. This is used by alembic to create migrations.

socorro.external.postgresql.raw_sql

This directory contains all of the stored procedures used by PostgreSQL.

2.10.7 socorro.external.rabbitmq

socorro.external.rabbitmq.crashstorage

- *RabbitMQCrashStorage*: In Production. Only is capable of storing the crash_id of a raw_crash. It *could* implement storage of dumps etc, but it is not suitable to actually store crashes at this time.

socorro.external.rabbitmq.connection_context

- *Connection*: In Production. A thin wrapper around *pika*. Also defines a channel and our declared queues (*socorro.normal* and *socorro.priority*). For commit/rollback, we just pass.
- *ConnectionContext*: Our factory implemented as a functor that we never use, but is a base class for our Pooled connections. If we use this directly, it will fail because the connections will close before the processors have a chance to have a look and ack.
- *ConnectionContextPooled*: In production. This is implemented as a per-thread pool.

socorro.external.rabbitmq.rmq_new_crash_source

A pluggable Functor/generator for feeding new crashes to the processor, implemented as a wrapper around new_crashes().

2.10.8 Which classes are used with which _app

- *socorro.collector.collector_app*: We currently only use *socorro.external.fs* in production. In testing we use *socorro.external.fs* and *socorro.external.rabbitmq*.
- *socorro.collector.crashmover_app*: In production: reads from *socorro.external.fs*, write to *socorro.external.hb*. In testing we use *socorro.external.fs*.
- *socorro.processor.processor_app*: In production: reads from *socorro.external.hb*, writes to *socorro.external.es*, *socorro.external.hb* and *socorro.external.postgresql* using *PolyCrashStore*. In testing we use *socorro.external.fs*, *socorro.external.rabbitmq*, and *socorro.external.postgresql*.

2.10.9 Which classes can be used together

Cannot mix *LegacyRadix* and *Radix* in one system which runs more than one app and shares a filesystem.

2.10.10 Potential Edicts

- Every container has an attribute that describes it as a container!

2.11 Crash storage: Elasticsearch

2.11.1 Features

Socorro uses Elasticsearch as a back-end for several intensive features in the web app. Here is a list of those features:

- **Super Search**

  Probably the most important one, Super Search is an improved search page that allows users to search through any field they like in the database. It exposes convenient and powerful operators and allows to build complex queries. It is also accessible via the public API, allowing users to build their own tools.

  Example: https://crash-stats.mozilla.com/search/

- **Custom Queries**

  Based on Super Search, this feature allows users to write JSON queries that are executed directly against Elasticsearch. This is a very sensitive feature that gives unrestricted access to your data. Specific permissions are needed for users to have access to it.

  Example: nope, this is not publicly accessible :)

- **Signature Report**

  A replacement for the old `/report/list/` page. It is heavily based on Super Search, and provides useful information about crashes that share a signature. Features include listing crash reports, listing user comments and showing aggregation on any field of the database.

  Example: https://crash-stats.mozilla.com/signature/?signature=nsTimeout::~nsTimeout%28%29

- **Profile page**

  On the profile page, Super Search is used to show the recent crash reports that contain the user's email address.

  Example: https://crash-stats.mozilla.com/profile/

2.11.2 Supported versions of Elasticsearch

Socorro currently requires Elasticsearch version 1.4.

2.11.3 Configuration

You can see Elasticsearch common options by passing `--help` to the processor app and looking at the `resource.elasticsearch` options like this:

```
$ docker-compose run processor bash
app@processor:/app$ python ./socorro/processor/processor_app \
    --destination.crashstorage_class=socorro.external.es.crashstorage.ESCrashStorage \
    --help
```

As of this writing, it looks like this:

```
--resource.elasticsearch.elasticsearch_class
  (default: socorro.external.es.connection_context.ConnectionContext)

--resource.elasticsearch.elasticsearch_connection_wrapper_class
  a classname for the type of wrapper for ES connections
  (default: socorro.external.es.connection_context.Connection)

--resource.elasticsearch.elasticsearch_doctype
  the default doctype to use in elasticsearch
  (default: crash_reports)

--resource.elasticsearch.elasticsearch_index
  an index format to pull crashes from elasticsearch (use datetime's strftime format␣
↪to have daily, weekly or monthly indexes)
  (default: socorro%Y%W)

--resource.elasticsearch.elasticsearch_timeout
  the time in seconds before a query to elasticsearch fails
  (default: 30)

--resource.elasticsearch.elasticsearch_timeout_extended
  the time in seconds before a query to elasticsearch fails in restricted sections
  (default: 120)

--resource.elasticsearch.elasticsearch_urls
  the urls to the elasticsearch instances
  (default: http://elasticsearch:9200)
```

Validate your configuration

The best way to verify you have correctly configured your application for Elasticsearch is to send it a crash report and verify it is indexed. Follow the steps in *Service: Processor* to process a crash in your system. Once it is processed, verify that your Elasticsearch instance has the data:

```
$ curl -XGET localhost:9200/socorroYYYYWW/crash_reports/_count
```

By default, the indices used by Socorro are `socorroYYYYWW`, so make sure you get this part right depending on your configuration and the current date.

If you want to use the Web app the check your data, the best way is to go to the Super Search page (you need to switch it on) and hit the Search button with no parameter. That should return all the crash reports that were indexed in the passed week.

2.11.4 Master list of fields

Super Search, and thus all the features based on it, is powered by a master list of fields that tells it what data to expose and how to expose it. That list contains data about each field from Elasticsearch that can be manipulated.

The list is managed in code in `socorro/external/es/super_search_fields.py` as a dict of `name` -> `properties`.

The name of a field is exposed in the Super Search API. This must be unique.

Here is an explanation of each properties of a field:

Parameter	Description
in_database_name	Name of the field in the database and in Elasticsearch.
namespace	Namespace of the field. Separated with dots.
description	Description of the field, for admins only.
query_type	Defines operators that can be used in Super Search. See details below.
data_validation_type	Defines the validation done on values passed to filers of this field in Super Search.
permissions_needed	Permissions needed for a user to access this field.
form_field_choices	Choices offered for filters of that field in the Super Search form.
is_exposed	Is this field exposed as a filter?
is_returned	Is this field returned in results?
has_full_version	Does this field have a full version in Elasticsearch? Enable only if you use a multitype field in the storage mapping.
storage_mapping	Mapping that is used in Elasticsearch for this field. See Elasticsearch documentation for more info.

Here are the operators that will be available for each `query_type`. Note that each operator automatically has an opposite version (for example, each field that has access to the `contains` operator also has `does not contain`).

Query type value	Operators
enum	has terms
string	contains, is, starts with, ends with, exists
number	has terms, >, >=, <, <=
date	has terms, >, >=, <, <=
bool	is true

2.12 Crash storage: Postgres

The processor saves raw crash data to Postgres. That data is then moved around to a bunch of other tables for analysis.

Current status: This is going away 2018h1.

2.13 Crash storage: AWS S3

The collector saves raw crash data to Amazon S3.

The processor loads raw crash data from Amazon S3, processes it, and then saves the processed crash data back to Amazon S3.

All of this is done in a single S3 bucket.

The "directory" hierarchy of that bucket looks like this:

- `{prefix}/v2/{name_of_thing}/{entropy}/{date}/{id}`: Raw crash data.
- `{prefix}/v1/{name_of_thing}/{id}`: Processed crash data, dumps, dump_names, and other things.

2.14 Socorro Test Checklist

This is a high-level system-wide checklist for making sure Socorro is working correctly in a specific environment. It's a helpful template for figuring out what you need to change if you're pushing out a significant change.

Note: This is used infrequently, so if you're about to make a significant change, you should go through the checklist to make sure the checklist is correct and that everything is working as expected and fix anything that's wrong, THEN make your change, then go through the checklist again.

Lonnen the bear says, "Only you can prevent production fires!"

Last updated: February 6th, 2018

2.14.1 How to use

"Significant change" can mean any number of things, so this is just a template. You should do the following:

1. Copy and paste the contents of this into a Google Doc, Etherpad, or whatever system you plan to use to keep track of status and outstanding issues.
2. Go through what you copy-and-pasted, remove things that don't make sense, and add additional things that are important. (Please uplift any changes via PR to this document that are interesting.)

2.14.2 Checklist

```
Verify version
==============

Before doing anything, verify the environment(s) that you're testing
are running the version you expect.

* local dev: http://localhost:8000/api/Status
* -stage: https://crash-stats.allizom.org/api/Status
* -prod: https://crash-stats.mozilla.com/api/Status


Migrations
==========

Make sure we can run migrations

* Django migrations

  Local dev environment:

  1. "docker-compose run webapp bash"
  2. "cd webapp-django/"
  3. "./manage.py showmigrations"

  -stage/-prod:

  1. See Mana

* Alembic migrations
```

(continues on next page)

(continued from previous page)

```
  Local dev environment:

  1. "docker-compose run processor bash"
  2. "alembic -c docker/config/alembic.ini current"

  -stage/-prod:

  1. See Mana


Collector (Antenna)
===================

Is the collector handling incoming crashes?

* Check datadog Antenna dashboard for the appropriate environment.

  localdev: Check the logging in the console
  stage: https://app.datadoghq.com/dash/272676/antenna--stage
  prod: https://app.datadoghq.com/dash/274773/antenna--prod

* Log into Sentry and check for errors.

* Submit a crash to the collector. Verify raw crash made it to S3.


Processor
=========

Is the processor process running?

* Log into a processor node and watch the processor logs for errors.

  Do: "journalctl -u socorro-processor -f"

  To check for errors grep for "ERRORS".

* Check Datadog "processor.save_raw_and_processed" for appropriate
  environment.

  localdev: Check the logging in the console
  stage: https://app.datadoghq.com/dash/187676/socorro-stage-perf
  prod: https://app.datadoghq.com/dash/65215/socorro-prod

Is the processor saving to ES? Postgres? S3?

* Check Datadog
  "processor.es.ESCrashStorageRedactedJsonDump.save_raw_and_processed.avg"

  stage: https://app.datadoghq.com/dash/187676/socorro-stage-perf
  prod: https://app.datadoghq.com/dash/65215/socorro-prod

* Check Datadog
  "processor.s3.BotoS3CrashStorage.save_raw_and_processed" for
  appropriate environment.

  stage: https://app.datadoghq.com/dash/187676/socorro-stage-perf
```

(continues on next page)

(continued from previous page)

```
  prod: https://app.datadoghq.com/dash/65215/socorro-prod

* Check Datadog
  "processor.postgres.PostgreSQLCrashStorage.save_raw_and_processed"

  stage: https://app.datadoghq.com/dash/187676/socorro-stage-perf
  prod: https://app.datadoghq.com/dash/65215/socorro-prod


Submit a crash or reprocess a crash. Wait a few minutes. Verify the crash was
processed and made it to S3, ES and Postgres.

**FIXME:** We should write a script that uses envconsul to provide vars and takes
a uuid via the command line and then checks all the things to make sure it's
there. This assumes we don't already have one--we might!


Webapp
======

Is the webapp up?

* Use a browser and check the healthcheck (/monitoring/healthcheck)

  It should say "ok: true".

Is the webapp throwing errors?

* Check Sentry for errors
* Log into webapp node and check logs for errors.

  Do: "journalctl -u socorro-webapp -f"

  To check for errors, grep that for "ERROR".

* Run QA Selenium tests.

  localdev: ?
  stage: In IRC: "webqatestbot build socorro.stage.saucelabs"
  prod: In IRC: "webqatestbot build socorro.prod.saucelabs"

Do webapp errors make it to sentry?

* Log into the webapp, go to the Admin, and use the Crash Me Now tool

Are there JavaScript errors in the webapp?

* While checking individual pages below, open the DevTools console and watch
  for JavaScript errors.

Can we log into the webapp?

* Log in and check the profile page.

Is the product home page working?

* Check the Firefox product home page (/ redirects to /home/product/Firefox)
```

(continues on next page)

(continued from previous page)

```
Is super search working?

* Click "Super Search" and make a search that is not likely to be cached.
  For example, filter on a specific date.

Top Crashers Signature report and Report index

1. Browse to Top Crashers
2. Click on a crash signature to browse to Signature report
3. Click on a crash id to browse to report index


Crontabber
==========

Is crontabber working?

* Check healthcheck endpoint (/monitoring/crontabber/)

  It should say ALLGOOD.

* Check the webapp crontabber-state page (/crontabber-state/)

Is crontabber throwing errors?

* Check Sentry for errors
* Log into admin node and check logs for errors

  Do: "tail -f /var/log/socorro/crontabber"

  To check for errors, grep for "ERROR".


Stage submitter
===============

Is the stage submitter running and sending crashes?

* Check Datadog dashboard for Antenna on -stage to see if it's
  receiving crashes
```

2.15 How app and an example works using configman

2.15.1 The minimum app

To illustrate the example, let's look at an example of an app that uses `socorro_app` to leverage `configman` to run. Let's look at weeklyReportsPartitions.py

As you can see, it's a subclass of the socorro.app.socorro_app.App class which is a the-least-you-need wrapper for a minimal app. As you can see, it takes care of logging and executing your `main` function.

2.15.2 Connecting and handling transactions

Let's go back to the `weeklyReportsPartitions.py` cron script and take a look at what it does.

It only really has one `configman` option and that's the `transaction_executor_class`. The default value is TransactionExecutorWithBackoff which is the class that's going to take care of two things:

1. execute a callable that accepts an opened database connection as first and only parameter
2. committing the transaction if there are no errors and rolling back the transaction if an exception is raised
3. NB: if an `OperationalError` or `InterfaceError` exception is raised, `TransactionExecutorWithBackoff` will log that and retry after configurable delay

Note that `TransactionExecutorWithBackoff` is the default `transaction_executor_class` but if you override it, for example by the command line, with `TransactionExecutor` no exceptions are swallowed and it doesn't retry.

Now, connections are created and closed by the ConnectionContext class. As you might have noticed, the default `database_class` defined in the `TransactionExecutor` is `socorro.external.postgresql.connection_context.ConnectionContext` as you can see here

The idea is that any external module (e.g. Boto, PostgreSQL, etc) can define a `ConnectionContext` class as per this model. What its job is is to create and close connections and it has to do so in a contextmanager. What that means is that you can do this:

```
connector = ConnectionContext()
with connector() as connection:  # opens a connection
    do_something(connection)
# closes the connection
```

And if errors are raised within the `do_something` function it doesn't matter. The connection will be closed.

2.15.3 What was the point of that?!

For one thing, this app being a `configman` derived app means that all configuration settings are as flexible as `configman` is. You can supply different values for any of the options either by the command line (try running `--help` on the `./weeklyReportsPartitions.py` script) and you can control them with various configuration files as per your liking.

The other thing to notice is that when writing another similar cron script, all you need to do is to worry about exactly what to execute and let the framework take care of transactions and opening and closing connections. Each class is supposed to do one job and one job only.

`configman` uses not only basic options such as `database_password` but also more complex options such as aggregators. These are basically invariant options that depend on each other and uses functions in there to get its stuff together.

2.15.4 How to override where config files are read

`socorro_app` supports multiple ways of picking up config files. The most basic option is the *–admin.conf*= option. E.g.:

```
python myapp.py --admin.conf=/path/to/my.ini
```

The default if you don't specify a `--admin.conf` is that it will look for a `.ini` file with the same name as the app. So if `app_name='myapp'` and you start it like this:

```
python myapp.py
```

it will automatically try to read `config/myapp.ini` and if you want to override the directory it searches in you have to set an environment variable called `DEFAULT_SOCORRO_CONFIG_PATH` like this:

```
export DEFAULT_SOCORRO_CONFIG_PATH=/etc/socorro
python myapp.py
```

Which means it will try to read `/etc/socorro/myapp.ini`.

NOTE: If you specify a `DEFAULT_SOCORRO_CONFIG_PATH` that directory must exist and be readable or else you will get an `IOError` when you try to start your app.

2.16 Deploying Socorro

This chapter covers running Socorro in a server environment.

FIXME(willkg): Write this.

2.16.1 A note about support

This is a very Mozilla-specific product. We do not currently have the capacity to support external users. If you are looking to use Socorro for your product, maybe you want to consider alternatives like electron/mini-breakpad-server.

Where'd the collector go? (April 2017)

In April 2017, we spun off the collector as a separate project called Antenna. Antenna has a reduced project scope and works differently than the collector did in some ways. You can find it at https://github.com/mozilla-services/antenna.

After getting that working, we removed the collector code from the Socorro repository. We're removing other code, too.

If you rely on that collector, the last good release is 270.

You can get it with something like this:

```
   git clone https://github.com/mozilla-services/socorro
   git checkout 270


Or get the tarball::

   wget https://github.com/mozilla-services/socorro/archive/270.tar.gz
```

2.17 How to

2.17.1 Run security checks for dependencies

You can run the crontabber job that checks for security vulnerabilities locally:

```
make dockerdependencycheck
```

2.17.2 Connect to PostgreSQL Database

The local development environment's PostgreSQL database exposes itself on a non-standard port when run with docker-compose. You can connect to it with the client of your choice using the following connection settings:

- Username: `postgres`
- Password: `aPassword`
- Port: `8574`

2.17.3 Reprocess crashes

Reprocessing individual crashes

If you have appropriate permissions, you can reprocess an individual crash by viewing the crash report on the Crash Stats site, clicking on the "Reprocess" tab, and clicking on the "Reprocess this crash" button.

Reprocessing lots of crashes if you're not an admin

If you need to reprocess a lot of crashes, please write up a bug. In the bug description, include a Super Search url with the crashes you want reprocessed.

Reprocessing crashes if you're an admin

If you're an admin, you can create an API token with the "Reprocess Crashes" permission. You can use this token in conjunction with the `scripts/reprocess.py` script to set crashes up for reprocessing.

For example, this reprocesses a single crash:

```
$ docker-compose run processor bash
app@processor:app$ ./scripts/reprocess.py c2815fd1-e87b-45e9-9630-765060180110
```

This reprocesses crashes all crashes with a specified signature:

```
$ docker-compose run processor bash
app@processor:app$ ./scripts/fetch_crashids.py --signature="some | signature" | ./
↪scripts/reprocess.py
```

Warning: If you're reprocessing more than 10,000 crashes, make sure to add a sleep argument of 10 seconds (`--sleep 10`). This will slow down adding items to the reprocessing queue such that the rate of crashes being added is roughly the rate of crashes being processed. Otherwise, you'll exceed our alert triggers for queue sizes and it'll page people.

Warning: August 17th, 2017: Everything below this point is outdated.

2.17.4 Populate PostgreSQL Database

Load Socorro schema plus test products:

```
socorro setupdb --database_name=breakpad --createdb
```

2.17.5 Create partitioned tables

Normally this is handled automatically by the cronjob scheduler *Service: Crontabber* but can be run as a one-off:

```
python socorro/cron/crontabber_app.py --job=weekly-reports-partitions --force
```

2.17.6 Populate Elasticsearch database

Note: See the chapter about *Crash storage: Elasticsearch* for more information.

Once you have populated your PostgreSQL database with "fake data", you can migrate that data into Elasticsearch:

```
python socorro/external/postgresql/crash_migration_app.py
```

2.17.7 Sync Django database

Django needs to write its ORM tables:

```
export SECRET_KEY="..."
cd webapp-django
./manage.py migrate auth
./manage.py migrate
```

2.17.8 Adding new products and releases

Each product you wish to have reports on must be added via the Socorro admin UI:

http://crash-stats/admin/products/

All products must have one or more releases:

http://crash-stats/admin/releases/

Make sure to restart memcached so you see your changes right away:

```
sudo systemctl restart memcached
```

Now go to the front page for your application. For example, if your application was named "KillerApp" then it will appear at:

http://crash-stats/home/products/KillerApp