Antenna Documentation

Release 1.0

Mozilla

Jan 14, 2020

CONTENTS

1 Contents **3**

1.1 Symbolication . . . 3
1.2 Download . . . 8
1.3 Upload . . . 12
1.4 Configuration . . . 15
1.5 Celery . . . 22
1.6 Admin/Developer Documentation . . . 23
1.7 Frontend Documentation . . . 35
1.8 Redis Documentation . . . 38
1.9 Docker Tips and Tricks . . . 40
1.10 Authentication . . . 41
1.11 Benchmarking . . . 42
1.12 End-to-End Testing . . . 42

2 Indices and tables **45**

Mozilla Symbol Server, codename “tecken”, is a web service for handling all things symbols for Mozilla platform. In particular three major features:

1. Symbolication
2. Symbol Download
3. Symbol Upload

CHAPTER

ONE

CONTENTS

User docs:

1.1 Symbolication

1.1.1 History

The original implementation was Vladan Djeric's Snappy-Symbolication-Server which was written in Tornado and was never hosted with ops support or monitoring.

This is a re-write of that and baked in as one of multiple features in Mozilla Symbol Server.

1.1.2 What It Is

You make a POST request to https://symbols.mozilla.org/symbolicate/v5 with a JSON body. That JSON body has to contain certain keys and adhere to a specific format. Here is an example:

```
{
    "jobs": [
        {
          "memoryMap": [
            [
              "xul.pdb",
              "44E4EC8C2F41492B9369D6B9A059577C2"
            ],
            [
              "wntdll.pdb",
              "D74F79EB1F8D4A45ABCD2F476CCABACC2"
            ]
          ],
          "stacks": [
            [
              [0, 11723767],
              [1, 65802]
            ]
          ]
        }
    ]
}
```

As a shortcut, you don't have to have a list `jobs`. E.g. this works the same as the example above:

```
{
  "memoryMap": [
    [
      "xul.pdb",
      "44E4EC8C2F41492B9369D6B9A059577C2"
    ],
    [
      "wntdll.pdb",
      "D74F79EB1F8D4A45ABCD2F476CCABACC2"
    ]
  ],
  "stacks": [
    [
      [0, 11723767],
      [1, 65802]
    ]
  ]
}
```

The `memoryMap` is list of symbol filenames and their debug ID. Each 2-D tuple corresponds to a path in our S3 symbol store. The full URL comes from taking the base URL (e.g. `https://s3-us-west-2.amazonaws.com/org.mozilla.crash-stats.symbols-public/v1/`) plus the first first part (e.g. `wntdll.pdb`) and then the debug ID (e.g. `D74F79EB1F8D4A45ABCD2F476CCABACC2`) and then lastly the first part again but instead of `.pdb` it's replaced with `.sym`.

So, as a full example, `["wntdll.pdb", "D74F79EB1F8D4A45ABCD2F476CCABACC2"]` becomes `https://s3-us-west-2.amazonaws.com/org.mozilla.crash-stats.symbols-public/v1/wntdll.pdb/D74F79EB1F8D4A45ABCD2F476CCABACC2/wntdll.sym`.

The `stacks` part is a list of lists, also known as "an array of stack traces". Each stack trace is a list of "frames". Each frame is a 2-D tuple of "module index" and "module offset". That module index is a number that corresponds to an item in the `memoryMap` (see above). And the module offset is the offset of this frame's instruction pointer relative to the base memory address of the module in which the code is contained as an integer (in base 16).

So, in the example above, `[1, 65802]` means the 1th element (starting with 0) in the `memoryMap` which in this example is `["wntdll.pdb", "D74F79EB1F8D4A45ABCD2F476CCABACC2"]`.

What you get back is a JSON output that looks like this:

```
{
  "results": [
    {
      "stacks": [
        [
          {
            "frame": 0,
            "module_offset": "0xb2e3f7",
            "module": "xul.pdb",
            "function": "sctp_send_initiate",
            "function_offset": "0x4ca"
          },
          {
            "frame": 1,
            "module_offset": "0x1010a",
            "module": "wntdll.pdb"
          }
        ]
      ],
```

(continues on next page)

(continued from previous page)

```
      "found_modules": {
        "wntdll.pdb/D74F79EB1F8D4A45ABCD2F476CCABACC2": false,
        "xul.pdb/44E4EC8C2F41492B9369D6B9A059577C2": true
      }
    }
  ]
}
```

The `results` list's order matches the list of `jobs` in the input.

1.1.3 Legacy API, Version 4

Prior to Version 5, was version 4 and the difference are as follows. The input could only be exactly 1 job. And the JSON input had to contain `"version": 4`.

The output had less information and it was always the output of exactly 1 job. And example output:

```
{
  "symbolicatedStacks": [
    [
      "XREMain::XRE_mainRun() (in xul.pdb)",
      "KiUserCallbackDispatcher (in wntdll.pdb)"
    ]
  ],
  "knownModules": [
    true,
    true
  ]
}
```

The order of the `symbolicatedStacks` matches the order of the `stacks` sent in. Each item is a string of the format `{function name} in ({module file name})`.

If the symbol can't find found a particular module, then the format becomes `{module offset in hex} in ({module file name})`.

The `knownModules` list matches the order of the `memoryMap` list. For each module and debug ID tuple, this is either `True` if the symbol file could be found or `False` if it couldn't be found or couldn't be downloaded.

Note! `knownModules` makes no distinction if the symbol file failure to download is permanent or temporary. The symbolication server attempts to retry failed downloads but it uses caching to remember that it failed for a limited amount of time to avoid hitting the same failure over and over.

The module offset (e.g. `65802`) might not correspond to an exact location in the module. If no exact location is found, it uses the nearest offset rounded down.

1.1.4 How It Works

Each module index and module offset pair is iterated over in each stack. The module index is used to load the symbol. Either from cache or from the S3 source.

When it was not available in the cache and had to be downloaded, we parse every line of the symbol file and extract all offsets and their function names from the lines that start with either `FUNC{space}` or `PUBLIC{space}`. Only this mapping is saved in the cache.

Once the symbols have been loaded from that module, we try to look up the offset. First we try to look up the exact offset and if that fails we sort ALL offsets in that module and find the nearest one, rounded down.

If any of the offsets can't be converted to a hex, it gets skipped and ignored. For example if you have a frame tuple that looks like this: `[0, 1.00000]` the resulting symbolication of that will simply be `["1.00000"]` as a string.

1.1.5 Example Symbolication

Here's an example you can copy and paste:

```
curl -d '{"jobs": [{"stacks":[[[0,11723767],[1, 65802]]],"memoryMap":[["xul.pdb",
↪"44E4EC8C2F41492B9369D6B9A059577C2"],["wntdll.pdb",
↪"D74F79EB1F8D4A45ABCD2F476CCABACC2"]]}]}' http://localhost:8000/symbolicate/v5
```

Note, if you only have a single job you can write it as this too:

```
curl -d '{"stacks":[[[0,11723767],[1, 65802]]],"memoryMap":[["xul.pdb",
↪"44E4EC8C2F41492B9369D6B9A059577C2"],["wntdll.pdb",
↪"D74F79EB1F8D4A45ABCD2F476CCABACC2"]]}' http://localhost:8000/symbolicate/v5
```

1.1.6 Symbolication With Debug

To get more debug information in the response output, you can add a header to the request. Simply set header `Debug` to something like `true`. For example:

```
curl -H 'Debug: true' -d '{"stacks":[[[0,11723767],[1, 65802]]],"memoryMap":[["xul.pdb
↪","44E4EC8C2F41492B9369D6B9A059577C2"],["wntdll.pdb",
↪"D74F79EB1F8D4A45ABCD2F476CCABACC2"]],"version":4}' http://localhost:8000/
↪symbolicate/v4
```

This will return an output that can look like this:

```
{
    "debug": {
        "cache_lookups": {
            "count": 2,
            "size": 0,
            "time": 0.006340742111206055
        },
        "downloads": {
            "count": 2,
            "size": 70490521,
            "time": 16.34278154373169
        },
        "modules": {
            "count": 2,
```

(continues on next page)

(continued from previous page)


```
            "stacks_per_module": {
                "wntdll.pdb/D74F79EB1F8D4A45ABCD2F476CCABACC2": 1,
                "xul.pdb/44E4EC8C2F41492B9369D6B9A059577C2": 1
            }
        },
        "stacks": {
            "count": 2,
            "real": 2
        },
        "time": 16.75939154624939
    },
    "knownModules": [
        true,
        true
    ],
    "symbolicatedStacks": [
        [
            "XREMain::XRE_mainRun() (in xul.pdb)",
            "KiUserCallbackDispatcher (in wntdll.pdb)"
        ]
    ]
}
```


The keys inside the `debug` block means as follows:

- `cache_lookups.count` - how many times it tried to do a query on the LRU cache
- `cache_lookups.size` - the total bytes size of data returned by the LRU cache
- `cache_lookups.time` - total time it took to make these queries on the LRU cache
- `downloads.count` - number of successful downloads of symbols over the network
- `downloads.size` - the total bytes size of symbols downloaded when uncompressed
- `downloads.time` - total time it took to make these downloads over the network
- `modules.count` - number of modules that needed to be looked up
- `modules.stacks_per_module` - number of stacks that were referring to each module
- `stacks.count` - total number of frames in all stack traces that were symbolicated
- `stacks.real` - total number of frames in all stack traces that were symbolicated except those offsets that couldn't be converted to hex.

1.1.7 URL shortcut

The ideal URL to POST request to is https://symbols.mozilla.org/symbolicate/v4 but to support legacy usage when the domain was *symbolapi.mozilla.org* you can also do the same POST request to https://symbols.mozilla.org/ too.

1.1.8 Sporadic Network Errors

If, during symbolication, we fail to download a symbol from the S3 store, the symbolication fails and a `503 Service Unavailable` is returned. This only happens if the communication with S3 fails in an **operational way** such as failing to connect, SSL errors, or timeouts.

If you, as a client receive this error it means you can try again later and it should work then.

If however there are problems with the symbol files downloaded from S3, the client will receive a `200 OK` modules needed will be reported as **not found**. For example, if a certain `xul.sym` file is corrupt, the symbolication will work but it will say the module can't be found.

1.1.9 How Caching Works

The S3 symbol storage is vastly bigger than the Symbol Server Symbolication can have available at short notice so each symbol is looked up on the fly and when looked up once, stored in a Redis server that is configured to work to work as an LRU (Least Recently Used) cache.

It means it's capped and it will keep symbols that are frequently used hot. When the Redis LRU cache saves an entry, it compares if the total memory used is now going to be bigger than the maximum memory amount (configured by config or by the limit of the server it runs on) allowed. If so, it figures out which keys were **least recently** used and deletes them from Redis. The default configuration for how many it deletes is 5 but you can change that in configuration.

The eviction policy of the Redis LRU is `allkeys-lru`. If the eviction policy is not changed to one that evicts, every write would cause an error when you try to save new symbols.

1.1.10 Symbolication Caching and New Uploads

Symbolication relies on using Redis as a LRU cache. Meaning, if the symbolication process has to use the Internet (i.e. AWS S3) to download a symbol file, it prevents this from "happening again" by storing that in the LRU *without* an expiration time. That means that if a new symbol upload comes in that replaces an existing symbol file, in S3, the LRU cache needs to be informed.

The way it works is that for every single file that is uploaded (the individual file, not the `.zip` file), its key is sent to the LRU to be invalidated. It currently does not replace what gets invalidated. Instead that "void" is left untouched until the symbolication process needs it and it will then have to re-download it and store it again.

1.2 Download

1.2.1 History

The original solution that Tecken replaces is symbols.m.o which was a Heroku app that ran an Apache server that used proxy rewrites to draw symbols from `https://s3-us-west-2.amazonaws.com/org.mozilla.crash-stats.symbols-public/v1/`.

Its rewrite rules contained two legacy solutions:

1. Uppercase the debug ID in the filename.
2. Support having specific product names (e.g. `firefox`) prefixing the name of the symbol file.

The old symbol download server was using `symbols.mozilla.org` and was accessible only with `http://`.

1.2.2 What It Is

Tecken Download's **primary use-case** is to redirect requests for symbols to their ultimate source which is S3. For example, with a `GET`, requesting https://symbols.mozilla.org/firefox.pdb/448794C699914DB8A8F9B9F88B98D7412/firefox.sym will return a `302 Found` redirect to (at the time of writing) `https://s3-us-west-2.amazonaws.com/org.mozilla.crash-stats.symbols-public/v1/firefox.pdb/448794C699914DB8A8F9B9F88B98D7412/firefox.sym`.

This way, all configuration of S3 buckets is central to Tecken even if we decide to change to a different bucket or add/remove buckets.

The primary benefit of using Tecken Download instead of hitting S3 public URLs directly is that it's just one URL to remember and Tecken Download can iterate over a **list of S3 buckets**. This makes it possible to upload symbols in multiple places but have them all accessible from one URL.

The other use-case of this is if you're simply curious to see if a symbol file exists. Simply make a `HEAD` request instead of a `GET`.

1.2.3 404s Logged

All `GET` requests are logged and counted within Tecken. There is a basic reporting option to extract ALL symbols that was requested *yesterday* but couldn't be found. But note that the format is quite particular since it doesn't report the third part of the URI. And additionally it reports two extra possible query string parameters called `code_file` and `code_id`. So if you make a query like: https://symbols.mozilla.org/foo.pdb/448794C699914DB8A8F9B9F88B98D7412/foo.sym?code_file=FOO.dll&code_id=BAR ... yesterday, then request the CSV report at: https://symbols.mozilla.org/missingsymbols.csv it will contain a CSV line like this:

```
foo.pdb,448794C699914DB8A8F9B9F88B98D7412,FOO.dll,BAR
```

The CSV report is actually ultimately to help the Socorro Processor which used to manage reporting symbols that can't be found during processing. See https://bugzilla.mozilla.org/show_bug.cgi?id=1361809

It only yields missing symbols whose symbol ended with `.pdb` and filename ended with `.sym` (case insensitively). The purpose of this is to get missing symbols that *could* be fetched from Microsoft.

1.2.4 Microsoft on-the-fly Symbol Lookups

Under certain conditions, if a symbol can not be found in S3, we might try to look it up from Microsoft's download server (`https://msdl.microsoft.com/download/symbols/`) if the symbol file ends in `.pdb` and filename ends in `.sym`.

The HTTP error response code is still `404` but the response body will be `Symbol Not Found Yet` (instead of `Symbol Not Found`).

The lookup is relatively expensive since it depends two a network calls (to Microsoft's server and potentially our S3 upload) and various command line subprocesses (`cabextract` and `dump_syms`) so it's important it runs in the background.

Note that this operation is cached for a limited time so if you ask for the same symbol within a short window of time, it does *not* start another attempt to download from Microsoft.

All symbols that turns out to not be found are cached by an in-memory cache. However, every time the filename is matched to potentially be downloaded from Microsoft the general symbol download cache is invalidated. Meaning you can do this:

```
$ curl https://symbols.mozilla.org/foo.pdb/HEXHEX/foo.sym
...
404 Symbol Not Found Yet
$ curl https://symbols.mozilla.org/foo.pdb/HEXHEX/foo.sym
...
404 Symbol Not Found Yet
$ sleep 3  # roughly assume the download + S3 upload takes less than 3 sec
$ curl https://symbols.mozilla.org/foo.pdb/HEXHEX/foo.sym
...
302
```

Note: This was the original implementation https://gist.github.com/luser/92d5bc88478665554898

1.2.5 Ignore Patterns

We know with confidence users repeatedly query certain files that are never in our symbol stores. We can ignore them to suppress logging that they couldn't be found.

Right now, this is maintained as a configurable blacklist but is hard coded inside the `_ignore_symbol` code in `tecken.download.views`.

This approach might change over time as we're able to confidently identify more and more patterns that we know we can ignore.

1.2.6 File Extension Whitelist

When someone requests to download a symbol, as mentioned above, we have some ways to immediately decide that it's a 404 Symbol Not Found without even bothering to ask the cache or S3.

As part of that, there is also a whitelist of file extensions that are the only ones we should bother with. This list is maintained in `settings.DOWNLOAD_FILE_EXTENSIONS_WHITELIST` (managed by the environment variable `DJANGO_DOWNLOAD_FILE_EXTENSIONS_WHITELIST`) and this list is found in the source code (`settings.py`) and also visible on the home page if you're signed in as a superuser.

1.2.7 Download With Debug

To know how long it took to make a "download", you can simply measure the time it takes to send the request to Tecken for a specific symbol. For example:

```
$ time curl https://symbols.mozilla.org/firefox.pdb/448794C699914DB8A8F9B9F88B98D7412/
↪firefox.sym
```

Note, that will tell you the total time it took your computer to make the request to Tecken **plus** Tecken's time to talk to S3.

If you want to know how long it took Tecken *internally* to talk to S3, you can add a header to your outgoing request. For example:

```
$ curl -v -H 'Debug: true' https://symbols.mozilla.org/firefox.pdb/
↪448794C699914DB8A8F9B9F88B98D7412/firefox.sym
```

Then you'll get a response header called `Debug-Time`. In the `curl` output it will look something like this:

```
< Debug-Time: 0.627500057220459
```

If that value is not present it's because Django was not even able to route your request to the code that talks to S3. It can also come back as exactly `Debug-Time:  0.0` which means the symbol is in a blacklist of symbols that are immediately `404 Not Found` based on filename pattern matching.

1.2.8 Download Without Caching

Generally we can cache our work around S3 downloads quite aggressively since we tightly control the (only) input. Whenever a symbol archive file is uploaded, for every file within that we upload to S3 we also invalidate it from our cache. That means we can cache information about whether certain symbols exist in S3 or not quite long.

However, if you are debugging something or if you manually remove a symbol from S3 that control is "lost". But there is a way to force the cache to be ignored. However, it only ignores looking in the cache. It will always update the cache.

To do this append `?_refresh` to the URL. For example:

```
$ curl https://symbols.mozilla.org/foo.pdb/HEX/foo.sym
...302 Found...

# Now suppose you delete the file manually from S3 in the AWS Console.
# And without any delay do the curl again:
$ curl https://symbols.mozilla.org/foo.pdb/HEX/foo.sym
...302 Found...
# Same old "broken", which is wrong.

# Avoid it by adding ?_refresh
$ curl https://symbols.mozilla.org/foo.pdb/HEX/foo.sym?_refresh
...404 Symbol Not Found...

# Now our cache will be updated.
$ curl https://symbols.mozilla.org/foo.pdb/HEX/foo.sym
...404 Symbol Not Found...
```

1.2.9 Try Builds

By default, when you request to download a symbol, Tecken will iterate through a list of available S3 configurations. By default it's only really one, the main S3 bucket for public symbols.

To download symbols that might be part of a Try build you have to pass an optional query string key: `try`. Or you can prefix the URL with `/try`. For example:

```
$ curl https://symbols.mozilla.org/tried.pdb/HEX/tried.sym
...404 Symbol Not Found...

$ curl https://symbols.mozilla.org/tried.pdb/HEX/tried.sym?try
...302 Found...

$ curl https://symbols.mozilla.org/try/tried.pdb/HEX/tried.sym
...302 Found...
```

What Tecken does is, if you pass `?try` to the URL or use the `/try` prefix, it takes the existing list of S3 configurations and *appends* the S3 configuration for Try builds.

Note; symbols from Try builds is always tried last! So if there's a known symbol called `foo.pdb/HEX/foo.sym` and someone triggers a Try build (which uploads its symbols) with the exact same name (and build ID) and even if you use `https://symbols.mozilla.org/foo.pdb/HEX/foo.sym?try` the existing (non-Try build) symbol will be matched first.

1.3 Upload

1.3.1 History

Prior to 2018, Symbol upload was originally done in Socorro as part of Crash Stats. Now it's handled by Tecken in much the same way except Tecken additionally logs every individual file inside the ZIP file in the ORM.

1.3.2 Uploading basics

Uploading requires special permission. The process for requesting access to upload symbols is roughly the following:

1. *Create a bug <https://bugzilla.mozilla.org/enter_bug.cgi?format=__standard__&product=Socorro&component=Tecken>* requesting access to upload symbols.
2. A Tecken admin will process the request.

   If you are a Mozilla employee, your manager will be needinfo'd to verify you need upload access.

   If you are not a Mozilla employee, we'll need to find someone to vouch for you.
3. After that's been worked out, the Tecken admin will give you permission to upload symbols.

Once you have permission to upload symbols, you will additionally need an API token. Once you log in, you can create an API token. It needs to have the "Upload Symbols" permission.

1.3.3 How to upload by HTTP POST

Uploads by HTTP POST must have a `multipart/form-data` payload with a ZIP file containing the symbols files.

Here's a `curl` example:

```
$ curl -X POST -H 'auth-token: xxx' --form myfile.zip=@myfile.zip https://symbols.
↪mozilla.org/upload/
```

Here's a `Python` example using `requests`:

```
>>> import requests
>>> files = {'myfile.zip': open('path/to/myfile.zip', 'rb')}
>>> url = 'https://symbols.mozilla.org/upload/'
>>> response = requests.post(url, files=files, headers={'Auth-token': 'xxx'})
>>> response.status_code
201
```

1.3.4 How to upload by download URL

Instead of uploading the symbols file by HTTP POST, you can POST the url to where the symbols file is and Tecken will download the file from that location and process it.

This is helpful if the symbols file is very big and is already available at a publicly available URL.

An example with `curl`:

```
$ curl -X POST -H 'auth-token: xxx' -d url="https://queue.taskcluster.net/YC0FgOlE/
↪artifacts/symbols.zip" https://symbols.mozilla.org/upload/
```

An example with `Python` and the `requests` library:

```
>>> import requests
>>> url = 'https://symbols.mozilla.org/upload/'
>>> data = {'url': 'https://queue.taskcluster.net/YC0FgOlE/artifacts/symbols.zip'}
>>> response = requests.post(url, data=data, headers={'Auth-token': 'xxx'})
>>> response.status_code
201
```

Domains that Tecken will download from is specified in the `DJANGO_ALLOW_UPLOAD_BY_DOWNLOAD_DOMAINS` environment variable and at the time of this writing is set to:

```
queue.taskcluster.net
public-artifacts.taskcluster.net
```

If you need another domain supported, file a bug.

Note that Tecken will check redirects. At first a HEAD request is made with the URL and Tecken will check both the original URL and the redirected URL against the list of allowed URLs.

1.3.5 Symbols processing

Once the `.zip` file is uploaded, it's processed. The first part of the processing is validation. See section below on "Checks and Validation".

Once validation passes, it proceeds to iterate over the files within. For each file, it queries S3 if the file already exists by the exact same name and exact same size. If indeed it exists (same name and same size) it notes it as "skipped" and just logs that filename. If it does not exist, it proceeds to upload it to S3.

Once the upload processing is complete it creates one `Upload` object and one `FileUpload` object for every file that is uploaded to S3.

1.3.6 Which S3 Bucket

The S3 bucket that gets used for upload is based on a "default" and a map of exceptions for certain users.

The default is configured as `DJANGO_UPLOAD_DEFAULT_URL`. For example: `https://s3-us-west-2.amazonaws.com/org-mozilla-symbols-public`. From the URL the bucket name is deduced and that's the default S3 bucket used.

The overriding is based on the **uploader's email address**. The default configuration is to make no exceptions. But you can set `DJANGO_UPLOAD_URL_EXCEPTIONS` as a Python dict like this:

```
$ export DJANGO_UPLOAD_URL_EXCEPTIONS={'*@adobe.com': 'https://s3.amazonaws.com/
↪private-bucket'}
```

1.3.7 Checks and Validations

When you upload your `.zip` file the first check is to see that it's a valid ZIP file that can be extracted into at least 1 file.

The next check is that it iterates over the files within and checks if any file contains the list of strings in `settings.DISALLOWED_SYMBOLS_SNIPPETS`. This check is a blacklist check and its purpose is to assert, for example, that proprietary files are never uploaded in S3 buckets that might be exposed publicly.

To override this amend the `DJANGO_DISALLOWED_SYMBOLS_SNIPPETS` environment variable as a comma separated list. But be aware to include the existing defaults which can be seen in `settings.py`.

The final check is that each file path in the zip file matches the pattern `<module>/<hex>/<file>` or `<name>-symbols.txt`. All other file paths are rejected.

1.3.8 Gzip

Certain files get gzipped before being uploaded into S3. At the time of writing that list is all `.sym` files. S3, unlike something like Nginx, doesn't do content encoding on the fly based on the client's capabilities. Instead, we manually gzip the file in memory in Tecken and set the additional `ContentEncoding` header to `gzip`. Since these `.sym` files are always text based, it saves a lot of memory in the S3 storage.

Additionally, the `.sym` files get their content type (aka. mime type) set when uploading to S3 to `text/plain`. Because S3 can't know in advance that the files are actually ASCII plain text, if you try to open them in a browser it will set the `Content-Type` to `application/octet-stream` which makes it hard to quickly look at its content in a browser.

Both the gzip and the mimetype overrides can be changed by setting the `DJANGO_COMPRESS_EXTENSIONS` and `DJANGO_MIME_OVERRIDES` environment variables. See `settings.py` for the current defaults.

1.3.9 Metadata and Optimization

For every gzipped file we upload, we attach 2 pieces of metadata to the key:

1. Original size
2. Original MD5 checksum

The reasons for doing this is to be able to quickly skip a file if it's uploaded a second time.

A similar approach is done for files that *don't* need to be compressed. In the case of those files, we skip uploading, again, simply if the file size of an existing file hasn't changed. However, that approach is too expensive for compressed files. If we don't store and retrieve the original size and original MD5 checksum, we have to locally compress the file to be able to make that final size comparison. By instead checking the original size (and hash) we can skip early without having to do the compression again.

1.3.10 Try Builds

A Try build is a build of Firefox that isn't necessarily triggered by landing a patch in `mozilla-central`. The access model for triggering Try builds is much more relaxed. Try builds generate symbols that are useful to have for debugging too. However, because of the difference in access rights, it's important that symbols from Try builds aren't allowed to override symbols from non-Try builds. For this reason, Tecken uploads all symbols from Try builds in a different S3 configuration.

Another important difference between a Try build and a non-Try build is that the symbols are much less likely to be useful for a long time. A developer might be testing something out for a couple of days, do some debugging and then move on to something else. Therefore we don't save the Try build symbols for equally long in AWS S3.

So how do you distinguish between symbols from a Try build and those from a non-Try build?

1. By the API token's permission, or,
2. Explicitly passing the `try` POST key with a non-empty value.

If you upload symbols with the frontend, there's a checkbox to indicate that it's from a Try build. It's unchecked by default.

To upload by API key permission, create a new API Token and when you select permission to associate with it, select `Upload Try Symbols Files`. This is how the backend knows to associate this upload with the files coming from a Try build.

There's an override though. You can manually set the key-value `try`. Like this:

```
$ curl -X POST -H 'auth-token: xxx' --form try=true --form myfile.zip=@myfile.zip␣
↪https://symbols.mozilla.org/upload/
```

See the *Try builds* documentation under **Download**.

1.4 Configuration

Contents

- *Configuration*
  - *High-level things*
  - *General Configuration*
  - *AWS*
  - *Gunicorn*
  - *AWS S3*
    - *Downloading*
    - *Uploading*
    - *Upload By Download*
    - *Symbolication*
    - *Try Builds*
  - *PostgreSQL*
  - *Redis Cache*
  - *Redis Cache Errors*
  - *Redis Store*
  - *Redis Socket Timeouts*
  - *StatsD*

- *Authentication*
  - *oidcprovider*
  - *Auth0 and other OIDC providers*
  - *First Superuser*
- *Microsoft Symbol Download*

1.4.1 High-level things

Tecken requires the following services in production:

1. PostgreSQL 9.5
2. Redis for general "performance" caching
3. Redis for LRU caching

1.4.2 General Configuration

The Django settings depends on there being an environment variable called `DJANGO_CONFIGURATION`. The `Dockerfile` automatically sets this to `Prod` and the `Dockerfile.dev` overrides it to `Dev`.

```
# If production
DJANGO_CONFIGURATION=Prod

# If stage
DJANGO_CONFIGURATION=Stage

# If development server
DJANGO_CONFIGURATION=Dev
```

You need to set a random `DJANGO_SECRET_KEY`. It should be predictably random and a decent length:

```
DJANGO_SECRET_KEY=sSJ19WAj06QtvwunmZKh8yEzDdTxC2IPUXfea5FkrVGNoM4iOp
```

The `ALLOWED_HOSTS` needs to be a list of valid domains that will be used to from the outside to reach the service. If there is only one single domain, it doesn't need to list any others. For example:

```
DJANGO_ALLOWED_HOSTS=symbols.mozilla.org
```

For Sentry the key is `SENTRY_DSN` which is sensitive but for the front-end (which hasn't been built yet at the time of writing) we also need the public key called `SENTRY_PUBLIC_DSN`. For example:

```
SENTRY_DSN=https://bb4e266xxx:d1c1eyyy@sentry.prod.mozaws.net/001
SENTRY_PUBLIC_DSN=https://bb4e266xxx@sentry.prod.mozaws.net/001
```

Note! There are two configurations, related to S3, that needs to be configured (presumably different) on every deployment environment. They are: `DJANGO_SYMBOL_URLS` and `DJANGO_UPLOAD_DEFAULT_URL`. See the section below about **AWS S3**.

1.4.3 AWS

Parts of Tecken does use `boto3` to talk directly to S3. For that to work the following environment variables needs to be set:

```
AWS_ACCESS_KEY_ID=AKI....H6A
AWS_SECRET_ACCESS_KEY=....
```

This S3 access needs to be able to talk to the `org.mozilla.crash-stats.symbols-public` bucket which is in `us-west-2`.

Note: This default is likely to change in mid-2017.

1.4.4 Gunicorn

At the moment, the only configuration for `Gunicorn` is that you can set the number of workers. The default is 4 and it can be overwritten by setting the environment variable `GUNICORN_WORKERS`.

The number should ideally be a function of the web head's number of cores according to this formula: `(2 x $num_cores) + 1` as documented here.

1.4.5 AWS S3

First of all, Tecken will never *create* S3 buckets for you. They are expected to already exist. There is one exception to this; if you do local development with Docker and `minio`, those configured buckets are automatically created when the server starts. This is a convenience just for local development to avoid needing any complicated instructions to get up and running.

S3 buckets needs to be specified in two distinct places. One for where Tecken can **read** symbols from and one for where Tecken can **write**.

Downloading

The *reading configuration* (used for downloading) is called `DJANGO_SYMBOL_URLS`. It's a comma separated string. Each value, comma separated, is expected to be a URL. The URL is deconstructed to extract out things like AWS region, bucket name, prefix and whether the bucket should be reached by HTTP (i.e. public) or by `boto3` (i.e. private).

What determines if a symbol URL is private or public is if it has `access=public` inside the query string.

The bucket name is always expected to the be first part of the URL path. For example, in `http://example.com/bucket-name-here/rest/is/prefix` the bucket name is `bucket-name-here` and the prefix `rest/is/prefix`.

Uploading

The *write configuration* (used for uploading) is called potentially by two different environment variables:

1. `DJANGO_UPLOAD_DEFAULT_URL` - a URL to indicate the bucket where, by default, all uploads goes into unless it matches an exception based on the uploader's email address.

2. `DJANGO_UPLOAD_URL_EXCEPTIONS` - a Python dictionary that maps an email address or a email address glob pattern to a different URL.

As an example, imagine:

```
DJANGO_UPLOAD_DEFAULT_URL=https://s3-us-west-2.amazonaws.com/mozilla-symbols-public/
↪myprefix
DJANGO_UPLOAD_BUCKET_EXCEPTIONS={'*example.com': 'https://s3-us-west-2.amazonaws.com/
↪mozilla-symbols-private/', 'foo@bar.com': 'https://s3-us-west-2.amazonaws.com/
↪mozilla-symbols-special'}
```

In this case, if someone, who does the upload, has email `me@example.com` all files within the uploaded `.zip` gets uploaded to a bucket called `mozilla-symbols-private`.

Note: This functionality with `DJANGO_UPLOAD_BUCKET_EXCEPTIONS` is a bit clunky to say the least. It exists to get parity with symbol upload when it was done in Socorro. In the future, this kind of configuration is best moved to user land. That way superusers can decided about these kinds of exceptions.

Upload By Download

To upload symbols, clients can either HTTP POST a .zip file, or the client can HTTP POST a form field called `url`. Tecken will then download the file from there and proceed as normal (as if the same file had been part of the upload).

The environment variable to control this is `DJANGO_ALLOW_UPLOAD_BY_DOWNLOAD_DOMAINS`. It's default is:

```
queue.taskcluster.net, public-artifacts.taskcluster.net
```

Note that, if you decide to add another domain, if requests to that domain trigger redirects to *another* domain you have to add that domain too. For example, if you have a `mybigsymbolzips.example.com` that redirects to `cloudfront.amazonaws.net` you need to add both.

Symbolication

Symbolication uses the same configuration as Download does, namely `DJANGO_SYMBOL_URLS`.

The value of the `DJANGO_SYMBOL_URLS` is encoded (as a short hash) into every key Redis uses to store previous downloads as structured data. Meaning, if you change `DJANGO_SYMBOL_URLS` on an already running, all existing Redis store caching will be reset. And the old keys, that are now no longer accessible, will slowly be recycled as the Redis store uses a LRU eviction policy.

Try Builds

Try build symbols are symbols that come from builds with a much more relaxed access policy. That's why it's important that these kinds of symbols don't override the non-Try build symbols. Also, the nature of them is much more short-lived and when stored in S3 they should have a much shorter expiration time than all other symbols.

The configuration key to set is `DJANGO_UPLOAD_TRY_SYMBOLS_URL` and it works very similar to `DJANGO_UPLOAD_DEFAULT_URL`.

It's blank (aka. unset) by default, and if not explicitly set it becomes the same as `DJANGO_UPLOAD_DEFAULT_URL` but with the prefix `try` after the bucket name and before anything else.

So if `DJANGO_UPLOAD_TRY_SYMBOLS_URL` isn't set and `DJANGO_UPLOAD_DEFAULT_URL` is `http://s3.example.com/bucket/version0` then `DJANGO_UPLOAD_TRY_SYMBOLS_URL` "becomes" `http://s3.example.com/bucket/try/version0`.

If the URL points to a S3 bucket that doesn't already exist, you have to manually create the S3 bucket first.

1.4.6 PostgreSQL

The environment variable that needs to be set is: `DATABASE_URL` and it can look like this:

```
DATABASE_URL="postgres://username:password@hostname/databasename"
```

The connection needs to be able connect in SSL mode. The database server is expected to have a very small footprint. So, as long as it can scale up in the future it doesn't need to be big.

Note: Authors note; I don't actually know the best practice for setting the credentials or if that's automatically "implied" the VPC groups.

1.4.7 Redis Cache

The environment variable that needs to be set is: `REDIS_URL` and it can look like this:

```
REDIS_URL="redis://test.v8jvds.0001.usw1.cache.amazonaws.com:6379/0"
```

The amount of space needed is minimal. No backups are necessary.

In future versions of Tecken this Redis will most likely be used as a broker for message queues by Celery.

Expected version is **3.2** or higher.

1.4.8 Redis Cache Errors

By default, all exceptions that might happen when `django-redis` uses the default Redis cache are swallowed. This is done to alleviate potential disruption when AWS Elasticache is unresponsive, such as when it's upgraded. The Redis Cache is supposed to be for the sake of optimization in that it makes some slow computation unnecessary if repeated. But if the cache is not working at all (operational errors for example) it's better that the service continue to work even if it's slower than normal.

If you want to disable this and have all Redis Cache exceptions bubbled up, which ultimately yields a 500 server error, change the environment variable to:

```
DJANGO_REDIS_IGNORE_EXCEPTIONS=False
```

Note: If exceptions *do* happen, they are swallowed and logged and not entirely disregarded.

1.4.9 Redis Store

Aka. Redis Store. This is the cache used for downloaded symbol files. The environment value key is called `REDIS_STORE_URL` and it can look like this:

```
REDIS_STORE_URL="redis://store.deef34.0001.usw1.cache.amazonaws.com:6379/0"
```

This Redis will steadily grow large so it needs to not fail when it reaches max memory capacity. For this to work, it needs to be configured to have a `maxmemory-policy` config set to the value `allkeys-lru`.

In Docker (development) this is automatically set at start-up time but in AWS ElastiCache config is not a valid command. So this needs to configured once in AWS by setting up an ElastiCache Redis Parameter Group. In particular the expected config is: `maxmemory-policy=allkeys-lru`.

Expected version is **3.2** or higher.

1.4.10 Redis Socket Timeouts

There are two Redis connections. The "Redis Cache" and the "Redis Store". These have both have the same defaults for `SOCKET_CONNECT_TIMEOUT` (1 second) and `SOCKET_TIMEOUT` (2 seconds).

The environment variables and their defaults are listed below:

```
DJANGO_REDIS_SOCKET_CONNECT_TIMEOUT=1
DJANGO_REDIS_SOCKET_TIMEOUT=2
DJANGO_REDIS_STORE_SOCKET_CONNECT_TIMEOUT=1
DJANGO_REDIS_STORE_SOCKET_TIMEOUT=2
```

1.4.11 StatsD

The three environment variables to control the statsd are as follows (with their defaults):

1. `DJANGO_STATSD_HOST` (*localhost*)
2. `DJANGO_STATSD_PORT` (*8125*)
3. `DJANGO_STATSD_NAMESPACE` ('' (empty string))

1.4.12 Authentication

In the production, stage, and development deployments, Tecken uses Mozilla SSO, a self-hosted Auth0 instance that integrates with Mozilla's LDAP system.

For local development, Tecken uses a test OpenID Connect (OIDC) provider. This can be overridden to use an Auth0 account or other OIDC provider.

oidcprovider

Local developement is configured to use `oidcprovider`, a containerized OpenID Connect provider that allows self-created accounts. The default configuration is:

```
DJANGO_OIDC_RP_CLIENT_ID=1
DJANGO_OIDC_RP_CLIENT_SECRET=bd01adf93cfb
DJANGO_OIDC_OP_AUTHORIZATION_ENDPOINT=http://oidc.127.0.0.1.nip.io:8081/openid/
↪authorize
DJANGO_OIDC_OP_TOKEN_ENDPOINT=http://oidcprovider:8080/openid/token
DJANGO_OIDC_OP_USER_ENDPOINT=http://oidcprovider:8080/openid/userinfo
DJANGO_OIDC_VERIFY_SSL=False
DJANGO_ENABLE_AUTH0_BLOCKED_CHECK=False
```

To use the provider:

1. Load http://localhost:3000
2. Click "Sign In" to start an OpenID Connect session on `oidcprovider`
3. **Click "Sign up" to create an `oidcprovider` account:**
    - Username: A non-email username, like `username`
    - Email: Your email address
    - Password: Any password, like `password`
4. Click "Authorize" to authorize Tecken to use your `oidcprovider` account
5. You are returned to http://localhost:3000. If needed, a parallel Tecken User will be created, with default permissions and identified by email address.

You'll remain logged in to `oidcprovider`, and the account will persist until the `oidcprovider` container is stopped. You can visit http://oidc.127.0.0.1.nip.io:8081/account/logout to manually log out.

Auth0 and other OIDC providers

Mozilla SSO, a self-hosted instance of Auth0, is used in the production, stage, and development deployments, and Tecken has additional functionality that uses SSO / Auth0 features. See *Authentication* for details.

To use Auth0 in local development, customize your environment:

```
DJANGO_OIDC_RP_CLIENT_ID=clientidhereclientidhere
DJANGO_OIDC_RP_CLIENT_SECRET=clientsecretthereclientsecrethere
DJANGO_OIDC_OP_AUTHORIZATION_ENDPOINT=https://auth.mozilla.auth0.com/authorize
DJANGO_OIDC_OP_TOKEN_ENDPOINT=https://auth.mozilla.auth0.com/oauth/token
DJANGO_OIDC_OP_USER_ENDPOINT=https://auth.mozilla.auth0.com/userinfo
DJANGO_OIDC_VERIFY_SSL=True
DJANGO_ENABLE_AUTH0_BLOCKED_CHECK=True
```

Any OpenID Connect (OIDC) provider can be used. Many OIDC providers publish their endpoints, for example https://auth.mozilla.auth0.com/.well-known/openid-configuration.

First Superuser

Users need to create their own API tokens but before they can do that they need to be promoted to have that permission at all. The only person/people who can give other users permissions is the superuser. To bootstrap the user administration you need to create at least one superuser. That superuser can promote other users to superusers too.

This action does NOT require that the user signs in at least once. If the user does not exist, it gets created.

The easiest way to create your first superuser is to use `docker-compose`:

```
docker-compose run web superuser peterbe@example.com
```

1.4.13 Microsoft Symbol Download

We have, in the Symbol Download, a feature that can attempt to download missing symbols from Microsoft's server "on-the-fly". This is a new and quite untested feature so it's disabled by default. To enable it set the following environment variable:

```
DJANGO_ENABLE_DOWNLOAD_FROM_MICROSOFT=True
```

1.5 Celery

Contents

- *Celery*
  - *Primary Use Case*
  - *Testing Celery*
  - *Unit Testing with Celery*

1.5.1 Primary Use Case

Celery is used for the following tasks:

1. Every time an upload is made, a piece of code is triggered that counts how many uploads have been done in that UTC 24 day. This populates the `UploadsCreated` model. By querying that you can see how much was uploaded by date rather than having to do heavy aggregates on the main `Upload` model.
2. Every time a symbol URL is attempted to be retrieved but we find out it's not in our storage backend, we need to write this down in the database. We do that by calling `store_missing_symbol` which is synchronous. However, if any operational error happens, instead of giving up we send the same parameters to a wrapped function that runs as a Celery task. That task is wrapped with a patient decorator that retries repeatedly. Note that all of this is "guarded" by a memoization wrapper that tries to make sure we only all of this only happens once per 24 per symbol signature.
3. (NOT enabled as of Dec 2018) If a symbol is missing and its signature looks like we *might* be able to download it from Microsoft, we attempt to do all this work in a Celery task.

1.5.2 Testing Celery

For more information about how to end-to-end test the Celery tasks see the *End-to-end testing Celery* section.

1.5.3 Unit Testing with Celery

The ideal pattern is to write tests that test individual tasks directly rather than testing the views/functions that depend on the task. However, since in `conftest.py` there's a default `celery_config` fixture that enables `task_always_eager=True` which means the tasks run immediately in the same process as the test.

Project docs:

1.6 Admin/Developer Documentation

Contents

- *Admin/Developer Documentation*
  - *Setting up a development environment*
  - *Development services*
    - *What's Deployed*
    - *Datadog*
    - *Sentry*
  - *Code Conventions*
  - *Documentation*
  - *Hyperactive Document Writing*
  - *Testing*
  - *Hyperactive Test Running*
  - *Managing Python Requirements*
  - *Running `gunicorn` locally*
  - *Integration Testing*
    - *tecken-loader*
    - *webapp*
  - *Prod-like running locally*
  - *Frontend and prod-like running locally*
  - *Running things in background vs foreground*
  - *All metrics keys*
  - *Celery in local development mode*
  - *Minio (S3 mock server)*
  - *How to Memory Profile Python*

- *How to do local Upload by Download URL*
- *Frontend linting - Prettier files*
- *Python warnings*
- *How to `psql`*
- *Backup and Restore PostgreSQL*
- *Enable full logging of SQL used*
- *Auth not working*
- *How to make a Zip file*
- *How to make a Zip file from downloading*
- *`black`*
- *Debugging `minio` container*
- *Debugging a "broken" Redis*
- *Giving users upload permission*
- *Deployment*
  - *Dev*
  - *Stage*
  - *Prod*

1.6.1 Setting up a development environment

You can set up a development environment with:

```
# Builds Docker containers
$ make build

# Initializes service state (db)
$ make setup
```

Tecken has a webapp.

To run the webapp, do:

```
# Runs web and ui and required services
$ make run
```

Now a development server should be available at `http://localhost:3000`.

To test the symbolication run:

```
$ curl -d '{"stacks":[[[0,11723767],[1, 65802]]],"memoryMap":[["xul.pdb",
↪"44E4EC8C2F41492B9369D6B9A059577C2"],["wntdll.pdb",
↪"D74F79EB1F8D4A45ABCD2F476CCABACC2"]],"version":4}' http://localhost:3000/
↪symbolicate/v5
```

1.6.2 Development services

What's Deployed

Check out https://whatsdeployed.io/s-5HY

Datadog

If you have access to a Mozilla Cloud Ops Datadog account, use this to consume the metrics Tecken sends via `statsd`. One is for staying health, the other is for keeping track how it does things.

Tecken dashboards

Sentry

Prod: https://sentry.prod.mozaws.net/operations/symbols-prod/

Stage: https://sentry.prod.mozaws.net/operations/symbols-stage/

Dev: https://sentry.prod.mozaws.net/operations/symbols-dev/

Frontend: https://sentry.prod.mozaws.net/operations/symbols-frontend/

1.6.3 Code Conventions

All code files need to start with the MPLv2 header:

```
# 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/.
```

To check if any file is lacking a license preamble, run:

```
$ ./bin/sanspreamble.sh
```

It will exit non-zero if there are files lacking the preamble. It only checks git checked in files.

PEP8 is nice. All files are expected to be PEP8 and pyflakes compliant and the PEP8 rules (and exceptions) are defined in `.flake8` under the `[flake8]` heading.

If you hit issues, instead of re-writing the rules consider appending a comment on the end of the line that says `# noqa`.

All Python code is and should be formatted with black.

1.6.4 Documentation

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

To build the docs, run this:

```
$ make docs
```

This is the same as running:

```
$ ./bin/build-docs-locally.sh
```

To iterate on writing docs and testing that what you type compiles correctly, run the above mentioned command on every save and then open the file `docs/_build/html/index.html`. E.g.

```
# the 'open' command is for OSX
$ open docs/_build/html/index.html
```

1.6.5 Hyperactive Document Writing

If you write a lot and want to see the changes much sooner after having written them, you can temporarily enter a shell and run exactly the minimum needed. First start a shell and install the Python requirements:

```
$ make test
> pip install -r requirements/docs.txt
```

Now, you can run the command manually with just. . . :

```
> make -C docs html
```

And keep an browser open to the file `docs/_build/html/index.html` in the host environment.

If you're really eager to have docs built as soon as you save the `.rst` file in your editor, run this command:

```
> watchmedo shell-command -W -c 'make -C docs html' -R .
```

Note that if you make a change/save *during* the build, it will ignore that. So wait until it has finished before you save again. Note, that the `.rst` file you're working on doesn't actually need to change. A save-file is enough.

Also note that it won't build the docs until there has been at least one file save.

1.6.6 Testing

To run the tests, run this:

```
$ make test
```

Tests go in `tests/`. Data required by tests goes in `tests/data/`.

If you need to run specific tests or pass in different arguments, you can run bash in the base container and then run `py.test` with whatever args you want. For example:

```
$ make shell
> py.test

<pytest output>

> py.test tests/test_symbolicate.py
```

We're using py.test for a test harness and test discovery.

1.6.7 Hyperactive Test Running

If you want to make tests run as soon as you save a file you have to enter a shell and run `ptw` which is a Python package that is automatically installed when you enter the shell. For example:

```
$ make shell
> ptw
```

That will re-run `py.test` as soon as any of the files change. If you want to pass any other regular options to `py.test` you can after `--` like this:

```
$ make shell
> ptw -- -x --other-option
```

1.6.8 Managing Python Requirements

All Python requirements needed for development and production needs to be listed in `requirements.txt` with sha256 hashes.

The most convenient way to modify this is to run `hashin` in a shell. For example:

```
$ make shell
> pip install hashin
> hashin Django==1.10.99
> hashin other-new-package
```

This will automatically update your `requirements.txt` but it won't install the new packages. To do that, you need to exit the shell and run:

```
$ make build
```

To check which Python packages are outdated, use piprot in a shell:

```
$ make shell
> pip install piprot
> piprot -o
```

The `-o` flag means it only lists requirements that are *out of date*.

Note: A good idea is to install `hashin` and `piprot` globally on your computer instead. It doesn't require a virtual environment if you use pipsi.

1.6.9 Running `gunicorn` locally

To run `gunicorn` locally, which has concurrency, run:

```
$ make gunicorn
```

You might want to temporarily edit `.env` and set `DJANGO_DEBUG=False` to run it in a more production realistic way.

1.6.10 Integration Testing

tecken-loader

Use *tecken-loader* for integration testing uploading, downloading, and symbolication APIs. It contains a series of scripts and molotov loadtests.

Details at mozilla-services/tecken-loader.

This is useful for sending somewhat realistic symbolication requests that reference symbols that are often slightly different.

webapp

Here's a rough webapp test script:

1. go to website
2. wait for front page to load
3. log in
4. click on "downloads" in navbar
5. click on "user management" in navbar
6. click on "API tokens" in navbar
7. click on "uploads" in navbar
8. click on "symbolication" in navbar
9. click on "help" in navbar

1.6.11 Prod-like running locally

First you need to start Tecken with a set of configurations that mimics what's required in prod, except we're doing this in docker.

To do that, you need to set `DJANGO_CONFIGURATION=Prodlike` and run the gunicorn workers:

```
$ docker-compose run --service-ports --user 0  web bash
```

This will start 4 `gunicorn` workers exposed on `0.0.0.0:8000` and exposed outside of docker onto your host.

Note: If this fails to start, some exceptions might be hidden. If so, start a shell `docker-compose run --user 0 web bash` and run: `DJANGO_UPLOAD_DEFAULT_URL=http://minio:9000/testbucket DJANGO_SYMBOL_URLS=http://minio:9000/testbucket DJANGO_CONFIGURATION=Prodlike gunicorn tecken.wsgi:application -b 0.0.0.0:8000 --workers 4 --access-logfile -`

That configuration **forces** you to run with `DEBUG=False` independent of what value you have set in `.env` for `DEBUG`. Thus making it easy to switch from regular debug-mode development to prod-like serving.

The second step for this to be testable is to reach the server with `HTTPS` or else the app will forcibly redirect you to the `https://` equivalent of whatever URL you attempt to use (e.g. `http://localhost:8000/` redirects to `https://localhost:8000/`)

To test this, run a local Nginx server. But first, create a suitable hostname. For example, `prod.tecken.dev`. Edit `/etc/hosts` and enter a line like this:

```
127.0.0.1       prod.tecken.dev
```

To generate an nginx config file, run `./test-with-nginx/generate.py`. That will be print out a Nginx configuration file you can put where you normally put Nginx configuration files. For example:

```
$ ./test-with-nginx/generate.py --help
$ ./test-with-nginx/generate.py > /etc/nginx/sites-enabled/tecken.conf
$ # however you reload nginx
```

1.6.12 Frontend and prod-like running locally

When Tecken is deployed with continuous integration, it builds the static assets files for production use. These files are served by Django using Whitenoise. Basically, anything that isn't a matched Django URL-to-view gets served as a static file, if matched.

Suppose you want to run the prod-like frontend locally. For example, you might be hunting a frontend bug that only happens when the assets are minified and compiled. To do that you have to manually build the static assets:

```
$ cd frontend
$ yarn
$ yarn run build
```

This should create `frontend/build/*` files. For example `static/js/main.6d3b4de8.js`. This should now be available *thru* Django at `http://localhost:8000/static/js.main.6d3b4de8.js`.

When you're done you can delete `frontend/build` and `frontend/node_modules`.

1.6.13 Running things in background vs foreground

By default `make run` is wired to start three things in the foreground:

- Django (aka. `web`)
- Celery (aka. `worker`)
- React dev server (aka. `frontend`)

This is done by running `docker-compose up web worker frontend`. These services' output is streamed together to stdout in the foreground that this `docker-compose up ...` runs.

All other things that these depend on are run in the background. Meaning you don't see, for example, what the `minio` service does. It knows to *start* because in `docker-compose.yml` `web` is **linked** to `minio`.

If you instead want to run, for example, `minio` in the foreground here's how:

1. Comment out `minio` from the `links` part of `web` in `docker-compose.yml`
2. In a terminal run `docker-compose up minio`.
3. In another terminal run `make run`

Alternatively, just do step 1, from the list above, and then run: `docker-compose up minio web worker frontend`.

1.6.14 All metrics keys

To get insight into all metrics keys that are used, a special Markus backend is enabled called `tecken.markus_extra.LogAllMetricsKeys`. It's enabled by default in local development. And to inspect its content you can either open `all-metrics-keys.json` directly (it's git ignored) or you can run:

```
$ make shell
$ ./bin/list-all-metrics-keys.py
```

Now you can see a list of all keys that are used. Take this and, for example, make sure you make a graph in Datadog of each and everyone. If there's a key in there that you know you don't need or care about in Datadog, then delete it from the code.

The file `all-metrics-keys.json` can be deleted any time and it will be recreated again.

1.6.15 Celery in local development mode

When you do something like `make run` it starts Django, the frontend and the Celery worker. But it's important to note that it starts Celery with `--purge`. That means that every time you start up the worker, all jobs that have been previously added to the Celery query are purged.

This is to prevent foot-shooting. Perhaps a rogue unit test that didn't mock the broker and accidentally added hundreds of jobs that all fail. Or perhaps you're working on a git branch that changes how the worker job works and as you're jumping between git branches you start and stop the worker so that the wrong jobs are sent using the wrong branch.

Another real thing that can happen is that when you're doing loadtesting of the web app, and only run that in docker, but since the web app writes to the same Redis (the broker) thousands of jobs might be written that never get a chance to be consumed by the worker.

This is why `docker-compose` starts `worker-purge` instead of `worker` which is the same thing except it's started with `--purge` and this should only ever be done on local docker development.

1.6.16 Minio (S3 mock server)

When doing local development we, by default, mock AWS S3 and instead use minio. It's API compatible so it should reflect how AWS S3 works but with the advantage that you don't need an Internet connection and real S3 credentials just to test symbol uploads for example.

When started with docker, it starts a web server on `:9000` that you can use to browse uploaded files. Go to `http://localhost:9000`.

1.6.17 How to Memory Profile Python

The trick is to install https://pypi.python.org/pypi/memory_profiler (and `psutil`) and then start Gunicorn with it. First start a shell and install it there:

```
$ docker-compose run --service-ports --user 0  web bash
# pip install memory_profiler psutil
```

Now, to see memory reports of running functions, add some code to the relevant functions you want to memory profile:

```
from memory_profiler import profile

@profile
```

(continues on next page)

(continued from previous page)

```
def some_view(request):
    ...
```

Now run Gunicorn:

```
$ python -m memory_profiler  `which gunicorn` tecken.wsgi:application -b 0.0.0.0:8000␣
↪--timeout 60 --workers 1 --access-logfile -
```

1.6.18 How to do local Upload by Download URL

When doing local development and you want to work on doing Symbol Upload by HTTP posting the URL, you have a choice. Either put files somewhere on a public network, or serve the locally.

Before we start doing local Upload By Download URL, you need to make your instance less secure since you'll be using URLs like `http://localhost:9090`. Add `DJANGO_ALLOW_UPLOAD_BY_ANY_DOMAIN=True` to your `.env` file.

To serve them locally, first start the dev server (`make run`). Then you need to start a bash shell in the current running web container:

```
$ make currentshell
```

Now, you need some `.zip` files in the root of the project since it's mounted and can be seen by the containers. Once they're there, start a simple Python server:

```
$ ls -lh *.zip
$ python -m http.server --bind 0.0.0.0 9090
```

Now, you can send these in with `tecken-loadtest` like this:

```
$ export AUTH_TOKEN=xxxxxxxxxxxxxxxxxxxxxxxx
$ python upload-symbol-zips.py http://localhost:8000 -t 160 --download-url=http://
↪localhost:9090/symbols.zip
```

This way you'll have 3 terminals. 2 bash terminals inside the container and one outside in the `tecke-loadtests` directory on your host.

1.6.19 Frontend linting - Prettier files

All `.js` files in the frontend code is expected to be formatted with Prettier. Ideally your editor should be configured to automatically apply `Prettier` on save. Or by a git hook.

If you forget to format any files in a Pull Request, a linting check in CircleCI will fail if any file hasn't been formatted. To test this locally, use:

```
$ docker-compose run frontend lint
```

If you get any output, it means it found files that should/could have been formatted. The error message will explain what files need attention and how to just format them all right now.

If you don't really care all that much about what the difference is and just want to fix it automatically run:

```
$ docker-compose run frontend lintfix
```

1.6.20 Python warnings

The best way to get **all** Python warnings out on `stdout` is to run Django with the `PYTHONWARNINGS` environment variable.

```
$ docker-compose run --service-ports --user 0  web bash
```

Then when you're in bash of the web container:

```
# PYTHONWARNINGS=d ./manage.py runserver 0.0.0.0:8000
```

1.6.21 How to `psql`

The simplest way is to use the shortcut in the `Makefile`

```
$ make psql
```

If you have a `.sql` file you want to send into `psql` you can do that too with:

```
$ docker-compose run db psql -h db -U postgres < stats-queries.sql
```

. . . for example.

1.6.22 Backup and Restore PostgreSQL

To make a backup of the whole database use `pg_dump` like this:

```
$ docker-compose run db pg_dump -h db -U postgres > tecken.sql
```

If you import it with:

```
$ docker-compose run db psql -h db -U postgres < tecken.sql
```

1.6.23 Enable full logging of SQL used

To see all the SQL the ORM uses, change the `LOGGING` configuration in `settings.py`.

First, change the level for `django.db.backends` from `INFO` to `DEBUG`. Second, change `LOGGING_DEFAULT_LEVEL` from `INFO` to `DEBUG`.

Now, when you run `make run` you should see all SQL from Django into the terminal stdout.

1.6.24 Auth not working

There are many reasons for why authentication might not work. Most of the pit falls lies with the the configuration and credentials around OpenID Connect. I.e. Auth0 in our current case.

Another important thing is that on the Django side, caching and cookies work.

If you have trouble authenticating you can start the server and go to: `http://localhost:8000/__auth_debug__`. It will check that the cache can work between requests and that session cookies can be set and read.

1.6.25 How to make a Zip file

Suppose you have a file like `libxul.so.sym`. Suppose also that you have multiple files you want to put into the zip, but for now we'll just make a zip of one file but use the `-r` flag to demonstrate how to do it if there were multiple files:

```
$ mkdir zipthis
$ mkdir zipthis/libxul.so
$ mkdir zipthis/libxul.so/13E87871A778CDBAF11B298FD05E2DBA0
$ cp libxul.so.sym zipthis/libxul.so/13E87871A778CDBAF11B298FD05E2DBA0/
$ cd zipthis
$ zip mysymbols -r *
$ ls -l mysymbols.zip
-rw-r--r--  1 peterbe  staff  40945250 Aug 10 14:54 mysymbols.zip
```

1.6.26 How to make a Zip file from downloading

The above section was about how to create a valid symbol zip file using basic UNIX/bash tools. Another, more convenient, way is to use the script `bin/make-a-zip.py`. You specify the names of symbols from `https://symbols.mozilla.org`, by default, and it downloads them and packages it up in the right way. This is useful if you want to reproduce a problem with a symbol file locally for example. E.g.

```
$ ./bin/make-a-zip.py --help
$ ./bin/make-a-zip.py qipcap.pdb/54EB115B9E735A17A87BCA540732CE171 fake.dll/
↪5C34D92C63000/fake.dl_
Downloaded 3670 bytes (3.6KB, 10.6KB on disk) into /tmp/tmpti3fksrr/qipcap.pdb/
↪54EB115B9E735A17A87BCA540732CE171
Downloaded 199535 bytes (194.9KB, 194.9KB on disk) into /tmp/tmpti3fksrr/fake.dll/
↪5C34D92C63000
Wrote /Users/peterbe/Desktop/symbols.zip
```

Note that if you omit the filename of the symbol file, it assumes `modulename + .sym`. E.g. `qipcap.pdb/54EB115B9E735A17A87BCA540732CE171/qipcap.sym`.

1.6.27 `black`

black. is the Python code formatting tool we use to format all non-generated Python code. In CI, we test that all code passes `black --diff --check .`. When doing local development, consider setting up either some sort of "format on save" in your editor or a git pre-commit hook.

To check that all code is formatted correctly, run:

```
$ docker-compose run linting lintcheck
```

If you have a bunch of formatting complaints you can automatically fix them all with:

```
$ docker-compose run linting blackfix
```

1.6.28 Debugging `minio` container

`minio` is used in `docker-compose` as a local substitute for AWS S3. If it fails to start, it could be because of an upgrade of the image on Dockerhub. If it fails to start, try first to run:

```
$ docker-compose build minio
$ docker-compose up minio
```

If you get an error that looks like this:

> You are running an older version of Minio released 7 months ago

The simplest solution is to delete the `miniodata` directory. E.g:

```
$ rm -fr miniodata
```

1.6.29 Debugging a "broken" Redis

By default, we have our Redis Cache configured to swallow all exceptions (...and just log them). This is useful because the Redis Cache is only supposed to make things faster. It shouldn't block things from working even if that comes at a price of working slower.

To simulate that Redis is "struggling" you can use the CLIENT PAUSE command. For example:

```
$ make redis-cache-cli
redis-cache:6379> client pause 30000
OK
```

Now, for 30 seconds (30,000 milliseconds) all attempts to talk to Redis Cache is going to cause a `redis.exceptions.TimeoutError: Timeout reading from socket` exception which gets swallowed and logged. But you *should* be able to use the service fully.

For example, all things related to authentication, such as your session cookie should continue to work because we use the `cached_db` backend in `settings.SESSION_ENGINE`. It just means we have to rely on PostgreSQL to verify the session cookie value on each and every request.

1.6.30 Giving users upload permission

The user should write up a bug. See *Uploading basics*.

If the user is a Mozilla employee, needinfo the user's manager and verify the user needs upload permission.

If the user is not a Mozilla employee, find someone to vouch for the user.

Once vouched:

1. Log in to https://symbols.mozilla.org/users
2. Use the search filter at the bottom of the page to find the user
3. Click to edit and make give them the "Uploaders" group (only).
4. Respond and say that they now have permission and should be able to either upload via the web or create an API Token with the "Upload Symbol Files" permission.
5. Resolve the bug.

1.6.31 Deployment

Dev

Dev is at: https://symbols.dev.mozaws.net

Dev deploys every time someone lands something in the master branch.

Stage

Stage is at: https://symbols.stage.mozaws.net

To deploy to stage, create a tag:

```
$ ./bin/release.py make-tag
```

Prod

Prod is at: https://symbols.mozilla.org

To deploy to prod, ask an ops person to deploy the tag you pushed to stage.

1.7 Frontend Documentation

1.7.1 Overview

The frontend code tries to be as separate from the web server code as possible. The frontend is a static app (written in React with `react-router`) that communicates with the web server by making AJAX calls for JSON/REST and rendering in run-time.

The goal is for the web server (i.e. Django) to only return pure responses in JSON (or plain text or specific to some files) and never generate HTML templates.

1.7.2 The Code

All source code is in the `./frontend` directory. More specifically the `./frontend/src` which are the files you're most likely going to edit to change the front-end.

All `CSS` is loaded with `yarn` by either drawing from `.css` files installed in the `node_modules` directory or from imported `.css` files inside the `./frontend/src` directory.

The project is based on create-react-app so the main rendering engine is React. There is no server-side rendering. The idea is that all (unless explicitly routed in Nginx) requests that don't immediately find a static file should fall back on `./frontend/build/index.html`. For example, loading https://symbols.mozilla.org/uploads/browse will actually load `./frontend/build/index.html` which renders the `.js` bundle which loads `react-router` which, in turn, figures out which component to render and display based on the path ("/uploads/browse" for example).

1.7.3 Upgrading/Adding Dependencies

A "primitive" way of changing dependencies is to edit the list of dependencies in `frontend/package.json` and running `docker-compose build frontend`. **This is not recommended**.

A much better way to change dependencies is to use `yarn`. Use the `yarn` installed in the Docker frontend container. For example:

```
$ docker-compose run frontend bash
> yarn outdated                    # will display which packages can be upgraded today
> yarn upgrade date-fns --latest   # example of upgrading an existing package
> yarn add new-hotness             # adds a new package
```

When you're done, you have to rebuild the frontend Docker container:

```
$ docker-compose build frontend
```

Your change should result in changes to `frontend/package.json` *and* `frontend/yarn.lock` which needs to both be checked in and committed.

1.7.4 Production Build

(At the moment. . .)

Ultimately, the command `cd frontend && yarn run build` will output all the files you need in the `build` directory. These files are purely static and do *not* depend on NodeJS to run in production.

The contents of the directory changes names every time and `.css` and `.js` files are not only minified and bundled, they also have a hash in the filename so the files can be very aggressively cached.

The command to generate the build artifact is done by CircleCI. See the `.circleci/config.yml` file which kicks off a build.

You never need the production build when doing local development, on your laptop, with Docker.

1.7.5 Dev Server

For local development, when you run `docker-compose up web worker frontend` it starts the NodeJS dev server in the foreground, mixing its output with that of Django and Celery. Normally in `create-react-app` apps, the `yarn start` command is highly interactive, clears the screen, runs in full screen in the terminal, color coded and able to spit out any warnings or compilation errors. When run in docker, with non-TTY terminal, all output from the dev server is sent to `stdout` one line at a time.

When you start Docker for development (again `make run` or `docker-compose up web worker frontend`) it starts the dev server on port `:3000` and it also exposes a WebSocket on port `:35729`.

The WebSocket is there to notice if you change any of the source files, it then triggers a "hot reload" which tells the browser to reload `http://localhost:3000`.

1.7.6 Proxying

The dev server is able to proxy any requests that would otherwise be a `404 Not Found` over to the the same URL but with a different host. See the `frontend/package.json` (the "proxy" section). Instead, it rewrites the request to `http://web:8000/$uri` which is the Django server. So, if in `http://localhost:3000` you try to load something like `http://localhost:3000/api/users/search` it knows to actually forward that to `http://localhost:8000/api/users/search`.

When you run in production, this is entirely disabled. To route requests between the Django server and the static files (with its `react-router` implementation) that has to be configured in Nginx.

1.7.7 Authentication and Auth0

The frontend app does **not** handle authentication. Instead it relies on the browser to be able to maintain a cookie from the web server in consequent AJAX requests. This is done by doing fetches with "same-origin" credentials; meaning the frontend trusts that the client will pass its current cookies when it makes the AJAX request if and only if the origin is the same.

There is a REST endpoint the frontend talks to under `/api/_auth` which will tell the frontend if the client has a valid cookie, and/or the URL needed to go to to make the client authenticate herself with Auth0 and the Django web server.

No credentials are ever passed between the frontend and the Django web server. Only the user's email. This presence helps the frontend decide whether to render the "Sign In" or the "Sign Out" button.

1.7.8 Django API Endpoints

All AJAX requests from the frontend to the Django server should go via the `/api/` prefix which is the `tecken.api` Django app. This Django app will be for all frontend apps such as user management, API tokens or browsing the uploads history.

1.7.9 Watch out for `node_modules`!

If you ever run and build the frontend outside of Docker you end up with a directory `frontend/node_modules` which is ignored by git but is still part of the current working directory that Docker serves up and will cause things like `make build` be excessively slow since the directory can end up north of 100MB.

If you have a `frontend/node_modules` directory, feel free to delete it.

The dev server runs in a separate Docker container which builds its `node_modules` outside the files mounted to the host.

1.7.10 Working on `Docerfile.frontend`

If you make changes to `Dockerfile.frontend` you have to rebuild that container. A trick, to avoid having to rebuild everything is to just run:

```
docker-compose build frontend
```

1.7.11 Testing

There are no unit, integration or functional tests of the frontend.

A nice-to-have but considering the current expected amount of traffic/users it's not worth the effort.

1.7.12 State Management in React

The frontend app uses `react-router` to render different React components depending on the `pushState` URL. If a piece of state is needed, and it's contained to one component, use regular `this.setState()`. If a piece of state is needed across all (or most) components add it to the `Mobx` store. See the file `frontend/src/Store.js`. Changes to that object will trigger re-render of all active components that are observing the store.

1.8 Redis Documentation

1.8.1 Usage

Redis is used for two distinct purposes and the different configurations shouldn't be mixed.

One is used as an LRU cache for the Symbolication service. It's basically an alternative to a disk cache where eviction is automatically taken care of. The LRU functionality is dependent on two things:

- A `maxmemory` configuration being something other than 0 otherwise it will just continue to fill up until the server/computer runs out of RAM.
- A `maxmemory-policy` setting being someting other than `noeviction`.

The other Redis server is used for miscellaneous caching and as a broker for message queue workers (TO BE UPDATED).

1.8.2 Predicted Production Use

In 2014, Ted calculated that we need approximately 35GB storage to have a 99% cache hit ratio of all symbols that Socorro needs when symbolicating. In Tecken we don't store downloaded symbol files, but instead we store a minor subset of the downloaded files so by estimates we only store 20% of that weight. In conclusion, to maintain a 99% cache hit ratio we need 6GB and 2GB for a 95% cache hit ratio.

1.8.3 Usage In Django

Within the Django code, these two are accessible in this way:

```
from django.core.cache import caches

regular_cache = caches['default']
lru_cache = caches['store']
```

The first ("default") cache is also available in this form:

```
from django.core.cache import cache

regular_cache = cache
```

Because it uses the django-redis and the Django Cache framework API you can use it for all other sorts of caching such as caching sessions or cache control for HTTP responses. Another feature of this is that you can bypass the default expiration by explicitly setting a `None` timeout which means it's persistent. For example:

```
from django.core.cache import cache

cache.set('key', 'value', timeout=None)
```

1.8.4 CLIs

To go into CLI of each two Redis database use these shortcuts:

```
$ make redis-cache-cli
(...or...)
$ make redis-store-cli
```

From there you can see exactly what's stored. For example, to see the list of all symbols stored in the LRU cache:

```
$ make redis-store-cli

redis-store:6379> keys *
1) ":1:symbol:xul.pdb/44E4EC8C2F41492B9369D6B9A059577C2"
2) ":1:symbol:wntdll.pdb/D74F79EB1F8D4A45ABCD2F476CCABACC2"
```

1.8.5 Configuration

The default configuration, in Docker, for the Redis service used as a LRU cache is defined in the `docker/images/redis/redis.conf` file and it sets a `maxmemory` number that is appropriate for local development. When deployed in production this should be better tuned to fit the server it's on. This configuration also sets the right `maxmemory-policy` to the value `allkeys-lru` which is also ideal for production usage.

To see the configuration, use the `redis-store` service in the shell:

```
$ make redis-store-cli

redis-store:6379> config get maxmemory
1) "maxmemory"
2) "524288000"
redis-store:6379> config get maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

To override this, simply use `config set` instead of `config get`. For example:

```
$ make redis-store-cli

redis-store:6379> config set maxmemory 100mb
OK
redis-store:6379> config get maxmemory
1) "maxmemory"
2) "104857600"
```

To get an insight into the state of the Redis service use the `INFO` command:

```
$ make redis-store-cli

redis-store:6379> info
# Server
redis_version:3.2.8
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:9c531c9c1d171a62
redis_mode:standalone
os:Linux 4.9.13-moby x86_64
arch_bits:64
multiplexing_api:epoll
<redacted>
```

If you stop the Docker service and start it again it will revert to the configuration in `docker/images/redis/redis.conf`.

1.8.6 Unit Testing in Docker

Since Redis is the actual cache backend used even in unit tests, its data is persistent between tests. To avoid confusion between unit tests use the `clear_redis_store` pytest fixture. For example:

```
from django.core.cache import cache

def test_storage1(clear_redis_store):
    assert not cache.get('foo')
    cache.set('foo', 'bar')

def test_storage2(clear_redis_store):
    assert not cache.get('foo')
    cache.set('foo', 'different')
```

1.9 Docker Tips and Tricks

Docker is used to do development and continuous integration on Tecken. Below is a list of miscellaneous techniques to make development easier.

1.9.1 Bashing whilst running

If you have two terminals open, and run `make run` in one and `make bash` in another they are entirely separate containers. Meaning, if you, in the second terminal where you started bash try editing a file like this:

```
> jed /usr/local/lib/python3.5/site-packages/django/http/request.py
```

Any changes are ignored by the first container. The solution is to run this command:

```
$ docker-compose exec --user 0 web bash
```

When you run that, you'll first notice that it opens almost instantly since there's no booting-up time. Also, if you, in a *third* terminal now run `docker-compose ps` you'll see that there is no new container started. Now, any changes you make to files in this bash gets used by the Django runserver in the first terminal.

All the changes you make (e.g. print statements inside some Python dependency) is wiped any and forgotten when you rebuild the containers.

This is aliased in the `Makefile` as `make currentshell`.

1.9.2 Running from bash

When you run `docker-compose up` it automatically takes care of exposing the port instructions mentioned in `docker-compose.yml`. That means when you run `docker-compose up web` you'll be able to reach port `8000` inside the container from *outside* the container. However, if you for some reason want to go into the shell, do some magic, and then start Django's runserver from within, it won't be reachable unless you start the shell in this way:

```
$ docker-compose run --service-ports -u 0 web bash
```

Note the extra `--service-ports`.

1.9.3 What's Booted?

The `docker ps` list will all containers that are running. But `docker-compose ps` will list the same containers but with different information. The former gives nice stats on up when it was created and how long it's been up. The latter will give information about containers also have have been stopped.

1.10 Authentication

In the production, stage, and development deployments, Tecken uses Mozilla SSO, a self-hosted Auth0 instance that integrates with Mozilla's LDAP system.

For local development, Tecken uses a test OpenID Connect (OIDC) provider. This can be overridden to use an Auth0 or other OIDC account. See *Authentication configuration*.

Authentication **will let anybody** become a signed in user. But note, the user will **not have any useful permissions** to do anything more than anonymous users can do. That is, until someone uses the user administration to elevate this user's permissions.

1.10.1 Auth0 Blocked

A potential pattern is that a user logs in with their work email (e.g. `example@mozilla.com`), gets permissions to create API tokens, the uses the API tokens in a script and later *leaves* the company whose email she *used* she can no longer sign in to again. If this happens her API token should cease to work, because it was created based on the understanding that she was an employee and has access to the email address.

This is why there's a piece of middleware that periodically checks that users who once authenticated with Auth0 still is there and **not blocked**.

Being "blocked" in Auth0 is what happens, "internally", if a user is removed from LDAP/Workday and Auth0 is informed. There could be other reasons why a user is blocked in Auth0. Whatever the reasons, users who are blocked immediately become inactive and logged out if they're logged in.

If it was an error, the user can try to log in again and if that works, the user becomes active again.

This check is done (at the time of writing) max. every 24 hours. Meaning, if you managed to sign or use an API token, you have 24 hours to use this cookie/API token till your user account is checked again in Auth0. To override this interval change the environment variable `DJANGO_NOT_BLOCKED_IN_AUTH0_INTERVAL_SECONDS`.

1.10.2 Testing Blocked

To check if a user is blocked, use the `is-blocked-in-auth0` which is development tool shortcut for what the middleware does:

```
$ docker-compose run web python manage.py is-blocked-in-auth0 me@example.com
```

1.11 Benchmarking

1.11.1 Motivation

This documentation is about the benchmarking that is explicit. I.e. code that is written purely for the benefit of running benchmarks. You can of course do your own benchmarking of the "real functionality" your own way but we have built-in benchmarking code that is ideal for stress testing certain contained features.

1.11.2 Configuration

The most important configuration is to **enable or disable *all* benchmarking**. To do this set the `DJANGO_BENCHMARKING_ENABLED` environment variable.

By default all benchmarking is disabled. That's because we don't want to risk a bad benchmark that could disrupt production systems. It's best to enable, per environment, one at a time explicitly and disable benchmarking again when no more testing is necessary.

Then `DJANGO_BENCHMARKING_ENABLED` is ignored if the current user hitting the benchmark URL is a superuser.

1.11.3 Usage

The best approach is to read the source code to find out what benchmarks are available and what kind of options they accept or require.

To do that look at the code of `tecken/benchmarking/urls.py` and `tecken/benchmarking/views.py`

But basically the idea is that every benchmark is started by querying a key in S3, priming the cache, and then querying the key and the cache a bunch of times and summarizing the results.

1.12 End-to-End Testing

1.12.1 Overview

There is no automated end-to-end testing that triggers on post-deploy steps or anything like that. Mozilla Infrasec will continually check that our server responds with the right security headers. However, there are some noted techniques for testing.

1.12.2 Uploads

To run this, you need to have access and permissions to Prod (`symbols.mozilla.org`) **and** Stage (`https://symbols.stage.mozaws.net`). Go to Prod and grab (or create) an API token with permission: `View All Symbols Uploads`. For Stage, grab (or create) an API token with permission: `Upload Symbols Files`. Then run the following script like this:

```
$ ./bin/end-to-end-test-symbol-upload.py --help
$ ./bin/end-to-end-test-symbol-upload.py PRODTOKEN STAGETOKEN
```

It all goes well, it will find the most recent upload on Prod that uses the "Upload by Download URL" and send that URL to Stage. If all goes well it will output something like this:

```
Stage user: pbengtsson@example.com
Prod user: pbengtsson@example.com
About to upload 916.6MB as URL to Stage.

Took 3.5 minutes
Files skipped: 0
Files uploaded: 72
Files uploaded, completed: 72

To see it, go to: https://symbols.stage.mozaws.net/uploads/upload/1186

It worked!
```

1.12.3 Celery

To test that the relationship between the web app and the Celery worker is worker you can use a special, and public, endpoint called `/__task_tester__`. When you send a HTTP POST request to it, it starts a Celery job that writes to the main cache (Redis). Then, if you do a HTTP GET request afterwards, it will either respond with 200 OK if the cache got updated or 500 Internal Server Error if the cache did not get updated.

To run the test, first HTTP POST as per this example. . . :

```
 curl -v -XPOST localhost:8000/__task_tester__
> POST /__task_tester__ HTTP/1.1
>
< HTTP/1.1 201 Created
<
Now make a GET request to this URL
```

Then, the HTTP GET:

```
 curl -v localhost:8000/__task_tester__
> GET /__task_tester__ HTTP/1.1
>
< HTTP/1.1 200 OK
<
It works!
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search