DXR Documentation

Release 2.0

various

Nov 16, 2017

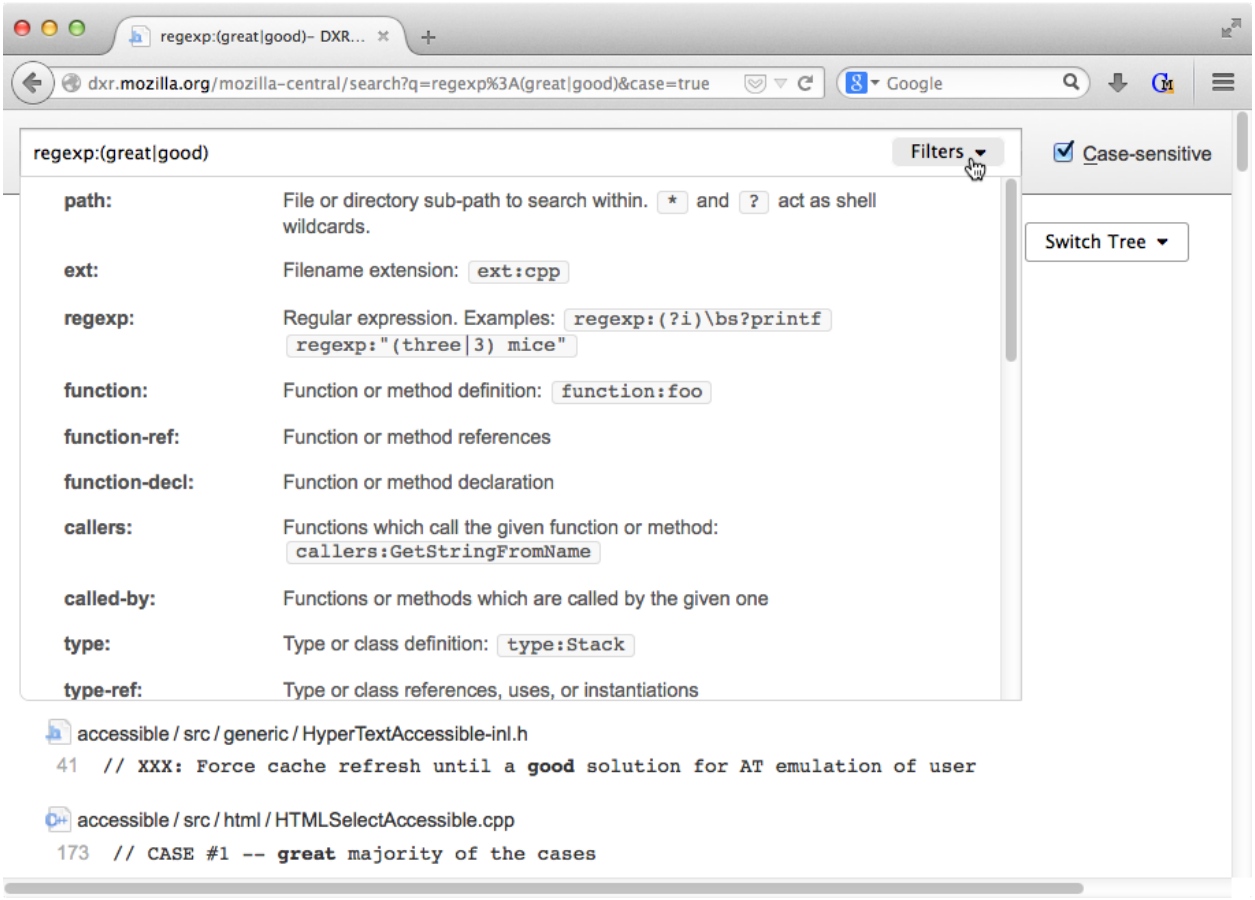
Contents

1 Contents **3**
- 1.1 Welcome To The DXR Community . . . 3
- 1.2 Getting Started . . . 4
- 1.3 Configuration . . . 6
- 1.4 Deployment . . . 10
- 1.5 Use . . . 15
- 1.6 Development . . . 18
- 1.7 Appendix A: Indexing Firefox . . . 33

2 Back Matter **35**
- 2.1 Glossary . . . 35
- 2.2 Icon Credits . . . 35

DXR is a code search and navigation tool aimed at making sense of large projects like Firefox. It supports full-text and regex searches as well as structural queries like "Find all the callers of this function." Behind the scenes, it uses trigram indices, elasticsearch, and static analysis data collected by instrumented compilers to make searches faster and more accurate than is possible with simple tools like grep. DXR also exposes a plugin API through which understanding of more languages can be added.

Here's an example of DXR running against the Firefox codebase. It looks like this:



CHAPTER 1

Contents

1.1 Welcome To The DXR Community

Though DXR got its start at Mozilla, it's seen contributions from a variety of companies and individuals around the world. We welcome contributions of code, bug reports, and helpful feedback.

1.1.1 Bug Reports

Did something explode? Not act as you expected? Let us know.

1.1.2 Submitting Patches

To contribute code, just file a pull request. Include tests to double your chances of getting it merged and qualify for a free Bundt cake. We love tests. Bundt cake isn't bad, either.

1.1.3 IRC

We hang out in the #static channel of irc.mozilla.org. Poke your head in and say hello.

If you have questions, please address them to the public channel; don't /msg someone in particular. That way, more people have a chance at answering your question, and more people can benefit from hearing the answers. We realize that no one likes looking naive, but please be brave and set an example to embolden the less-brave naive people. We're a friendly bunch and will never deride anyone for being a beginner.

1.1.4 Open Bugs

Looking for something to hack on? Here are...

- Our easy bugs

- All our bugs

Before starting work on a bug, hop into the IRC channel and confirm it's still relevant. We try to garden our bugs, but DXR often moves faster than we can weed.

1.1.5 Wiki

The wiki is full of roadmap documents, sketches toward future features, and bikeshedding sessions. Feel free to scribble on it.

1.2 Getting Started

Note: These instructions are for trying out DXR to see if you like it. If you plan to contribute code to DXR itself, please see *Development* instead.

The easiest way to get DXR working on your own machine is...

1. Get the source code you want to index.
2. If it a language analyzed at build time (like C++ or Rust), tell DXR how to build it.
3. Run **`dxr index`** to index your code.
4. Run **`dxr serve`** to present a web-based search interface.

But first, we have some installation to do.

1.2.1 Downloading DXR

Using git, clone the DXR repository:

```
git clone https://github.com/mozilla/dxr.git
```

1.2.2 Booting And Building

DXR runs only on Linux at the moment (and possibly other UNIX-like operating systems). The easiest way to get things set up is to use the included, preconfigured Docker setup. If you're not running Linux on your host machine, you'll need a virtualization provider. We recommend VirtualBox.

After you've installed VirtualBox (or ignored that bit because you're on Linux), grab the three Docker tools you'll need: docker, docker-compose, and, if you're not on Linux, docker-machine. If you're running the homebrew package manager on the Mac, this is as easy as...

```
brew install docker docker-compose docker-machine
```

Otherwise, visit the Docker Engine page for instructions.

Next, unless you're already on Linux, you'll need to spin up a Linux VM to host your Docker containers:

```
docker-machine create --driver virtualbox --virtualbox-disk-size 50000 --virtualbox-
↪cpu-count 2 --virtualbox-memory 512 default
eval "$(docker-machine env default)"
```

Feel free to adjust the resource allocation numbers above as you see fit.

Note: Next time you reboot (or run `make docker_stop`), you'll need to restart the VM:

```
docker-machine start default
```

And each time you use a new shell, you'll need to set the environment variables that tell Docker how to find the VM:

```
eval "$(docker-machine env default)"
```

When you're done with DXR and want to reclaim the RAM taken by the VM, run...

```
make docker_stop
```

Now you're ready to fire up DXR's Docker containers, one to run elasticsearch and the other to interact with you, index code, and serve web requests:

```
make shell
```

This drops you at a shell prompt in the interactive container. Now you can build DXR and run the tests to make sure it works. Type this at the prompt *within the container*:

```
# Within the docker container...
make test
```

1.2.3 Configuration

Before DXR can index your code, it needs to know where it is and, if you want to be able to do structural queries (like find-all-the-callers) for C, C++, or Rust, how to kick off a build. (Analysis of more dynamic languages like Python does not require a build step.) If you have a simple build process powered by **`make`**, a configuration like this might suffice. Place the following in a file called `dxr.config`. The location of the file doesn't matter, but the usual place is adjacent to your source directory.

```
[DXR]
# Some global options here, if you like

[yourproject]
source_folder       = /code/my-checkout
build_command       = make clean; make -j {workers}
```

Note: Be sure to replace the placeholder paths in the above config. You'll need to move your code to be indexed into the VM, either by downloading it from within the VM, or by moving it into your DXR repository folder, where it will be visible from within the VM in the shared `~/dxr` folder. It's possible to index your code from a folder within `~/dxr`, but, if you are using a non-Linux host machine, moving it to `/code` will give you much faster IO by taking VirtualBox's shared-folder machinery out of the mix.

By building your project with clang and under the control of **`dxr index`**, DXR gets a chance to interpose a custom compiler plugin that emits analysis data. It then processes that into an index.

If you have a non-C++ project and simply want to index it as text, the `build_command` can be set to blank:

```
build_command =
```

Though you shouldn't need any of them yet, further config directives are described in *Configuration*.

1.2.4 Indexing

Now that you've told DXR about your codebase, it's time to build an *index*:

```
dxr index --config dxr.config
```

Note: If you have a large codebase, the VM might run out of RAM. If that happens, wipe out the VM using `docker-machine rm default`, and then go back to the `docker-machine create` instruction and crank up the numbers. For example, this is plenty of space to build Firefox:

```
docker-machine create --driver virtualbox --virtualbox-disk-size 50000 --virtualbox-
↪cpu-count 4 --virtualbox-memory 8000 default

# Reset your shell variables:
eval "$(docker-machine env default)"

# And drop back into the DXR container:
make shell
```

Note: If you have trouble getting your own code to index, step back and see if you can get one of the included test cases to work:

```
cd ~/dxr/tests/test_basic
dxr index
```

If that works, it's just a matter of getting your configuration right. Pop into #static on irc.mozilla.org if you need a hand.

1.2.5 Serving Your Index

Congratulations; your index is built! Now, spin up DXR's development server, and see what you've wrought:

```
dxr serve --all
```

If you're using `docker-machine`, run `docker-machine ip default` to find the address of your VM. Then surf to http://*that IP address*:8000/ from the host machine, and poke around your fancy new searchable codebase.

If you're not using `docker-machine`, your code should be accessible from http://localhost:8000/.

1.3 Configuration

DXR learns how to index and serve your source trees by means of an ini-formatted configuration file:

```
[DXR]
# Some global options here, if you like

[yourproject]
source_folder       = /code/my-checkout
build_command       = make clean; make -j {workers}
```

When you invoke **`dxr index`**, it defaults to reading `dxr.config` in the current directory:

```
dxr index
```

Or you can pass in a config file explicitly:

```
dxr index --config /some/place/dxr.config
```

1.3.1 Sections

The configuration file is divided into sections. The `[DXR]` section holds global options; each other section describes a tree to be indexed.

You can use all the fancy interpolation features of Python's ConfigParser class to save repetition.

[DXR] Section

Here are the options that can live in the `[DXR]` section. For options representing path names, relative paths are relative to the directory containing the config file.

`disabled_plugins` Names of plugins to disable. Default: empty

`enabled_plugins` Names of plugins to enable. Default: `*`

`es_alias` A `format()`-style template for coming up with elasticsearch alias names. These live in the same namespace as indices, so don't pave over any index name you're already using. The variables `{format}` and `{tree}` will be substituted, and their meanings are as in `es_index`. Default: `dxr_{format}_{tree}`.

`es_index` A `format()`-style template for coming up with elasticsearch index names. The variable `{tree}` will be replaced with the tree name, `{format}` will be replaced with the format version, and `{unique}` will be replaced with a unique ID to keep a tree's new index from colliding with the old one. The unique ID includes a random number and the build hosts's MAC address so errant concurrent builds on different hosts at least won't clobber each other. Default: `dxr_{format}_{tree}_{unique}`

`es_catalog_replicas` The number of elasticsearch replicas to make of the *catalog index*. This is read often and written only when an indexing run completes, so crank it up so there's a replica on every node for best performance. But remember that writes will hang if at least half of the attempted copies aren't available. Default: `1`

`es_indexing_timeout` The number of seconds DXR should wait for elasticsearch responses during indexing. Default: 60

`es_indexing_retries` How many other ES nodes to try if a query to one during indexing times out or the connection fails. This is an experimental feature. Default: 0

`es_refresh_interval` The number of seconds between elasticsearch's consolidation passes during indexing. Set to -1 to do no refreshes at all, except directly after an indexing run completes. Default: 60

`generated_date` The "generated on" date stamped at the bottom of every DXR web page, in RFC-822 (also known as RFC 2822) format. Default: the time the indexing run started

log_folder A `format()`-style template for deciding where to store log files written while indexing. The token `{tree}` will be replaced with the name of the tree being indexed. Default: `dxr-logs-{tree}` (in the current working directory).

skip_stages Build/indexing/clean stages to skip, for debugging: `build`, `index`, `clean`, or any combination, whitespace-separated Either of `build` or `index` implies `clean`. Default: none

temp_folder A `format()`-style template for deciding where to store temporary files used while indexing. The token `{tree}` will be replaced with the name of each tree you index. Default: `dxr-temp-{tree}`. It's a good idea to keep this out of `/tmp` if it's on a small partition, since it can grow to tens of gigabytes on a large codebase.

workers Number of concurrent processes to use for building and indexing projects. Default: the number of CPUs on the system. Set to 0 to use no worker processes and do everything in the master process. This is handy for debugging.

Web App Options That Need a Restart

These options are used by the DXR web app (though some are used at index time as well). They are not frozen into the *catalog index* but rather are read when the web app starts up. Thus, the web app must be restarted to see new values of these.

default_tree The tree to redirect to when you visit the root of the site. Default: the first tree in the config file

es_hosts A whitespace-delimited list of elasticsearch nodes to talk to. Be sure to include port numbers. Default: http://127.0.0.1:9200/. Remember that you can split whitespace-containing things across lines in an ini file by leading with spaces.

es_catalog_index The name to use for the *catalog index*. You probably don't need to change this unless you want multiple otherwise-independent DXR deployments, with disjoint Switch Tree menus, sharing the same ES cluster. Default: `dxr_catalog`.

google_analytics_key Google analytics key. If set, the analytics snippet will added automatically to every page.

max_thumbnail_size The file size in bytes at which images will not be used for their icon previews on folder browsing pages. Default: 20000.

www_root URL path prefix to the root of DXR's web app. Example: `/smoo`. Default: empty.

Tree Sections

Any section not named `[DXR]` represents a tree to be indexed. Changes to per-tree options take effect when the tree is next indexed.

build_command Command for building your source code. Default: `make -j {workers}`. This is run within `object_folder`. Note that `{workers}` will be replaced with `workers` from the `[DXR]` section (though 1 if `workers` is set to 0).

clean_command Command for deleting the build products of `build_command`, restoring things to the pre-built state. Default: `make clean`. This is run within `object_folder`.

disabled_plugins Plugins disabled in this tree, in addition to ones already disabled in the `[DXR]` section. Default: `*`

enabled_plugins Plugins enabled in this tree. Default: `*`, which enables the same plugins enabled in the `[DXR]` section.

es_shards The number of shards to break the elasticsearch index into. Default: 5

ignore_patterns Whitespace-separated list of Unix shell-style file names or paths to ignore. Paths start with a slash, and file names don't. Patterns containing whitespace can be expressed by enclosing them in double quotes: `"Lovely readable name.human"`.

object_folder Folder where the `build_command` will be run. This is generally the folder where object files will be stored. Default: same as `source_folder`

source_folder The folder containing the source code to index. **Required.**

source_encoding The Unicode encoding of the tree's source files. Default: `utf-8`

temp_folder A `format()`-style template for deciding where to store temporary files used while indexing. The token `{tree}` will be replaced with the name of each tree you index. Default: `temp_folder` setting from `[DXR]` section. You generally don't need to set this.

p4web_url The URL to the root of a p4web installation. Default: `http://p4web/`

workers Number of concurrent processes to use for building and indexing this tree. Default: `workers` setting from `[DXR]` section. You might want to set this lower for a tree that uses memory-hungry plugins if you're low on RAM.

1.3.2 Plugin Configuration

Plugin-specific options go in `[[double-bracketed]]` sections under trees. For example...

```
[some-tree]

    [[buglink]]
    url = http://www.example.com/
    name = Example bug tracker
```

Currently, changes to plugin configuration take effect at index time or after restarting the web app; none are picked up by the web app in realtime.

See *Writing Plugins* for more details on plugin development.

[[buglink]]

name Name of the tree's bug tracker installation, e.g. `Mozilla's Bugzilla`

regex Regex for finding bug references to link in the source code. Default: `(?i)bug\s+#?([0-9]+)`, which catches things like "bug 123456"

url URL pattern for building links to tickets. `%s` will be replaced with the ticket number. The option should include the URL scheme.

[[python]]

python_path Path to the folder from which the codebase imports Python modules

[[xpidl]]

header_path Path to the folder where generated .h headers will be placed, used for URL construction.

include_folders Whitespace-separated list of paths to search in to resolve include directives. Default: [] (current folder)

1.3.3 Syntax

comments out the remainder of its line in most cases. To express a config value that contains #, place it in triple quotes:

```
regex = '''(?i)bug\s+\#?([0-9]+)'''
regex = """(?i)bug\s+\#?([0-9]+)"""
```

A single surrounding pair of single or double quotes will end up as part of the value at the moment, due to an apparent bug in configobj.

1.4 Deployment

Note: The best deployment story probably involves Docker and our setup scripts in `tooling/docker/dev/`. However, we haven't got that quite figured out yet. Feel free to chip in! In the meantime, enjoy this page about manually installing DXR on bare metal.

Once you decide to put DXR into production for use by multiple people, it's time to move beyond the *Getting Started* instructions. You likely need a real elasticsearch cluster, and you definitely need a robust web server like Apache. This chapter helps you deploy DXR on the Linux machines[1] of your choice and configure them to handle multi-user traffic volumes.

DXR generates an elasticsearch-dwelling *index* for one or more source trees as a batch process. This is well suited to a dedicated build server. One or more web servers then serve pages based on it.

1.4.1 Dependencies

OS Packages

You'll need to install several packages on both your build and web servers. These are the Ubuntu package names, but they should be clear enough to map to their equivalents on other distributions:

- make
- build-essential
- libclang-dev (clang dev headers). Version 3.5 is recommended, though we theoretically support back to 3.2.
- llvm-dev (LLVM dev headers, version 3.5 recommended)
- pkg-config
- npm; node 6.0.0 or higher
- openjdk-7-jdk
- elasticsearch 1.1 or higher. The elasticsearch corporation maintains its own packages; they aren't often found in distros. Newer is better, though I tend to avoid x.0 releases.

Technically, you could probably do without most of these on the web server, though you'd then need to build DXR itself on a different machine and transfer it over.

[1] DXR might also work with other UNIX-like operating systems, but we make no promises.

Note: On some systems (for example Debian and Ubuntu) the Node.js interpreter is named **nodejs**, but DXR expects it to be named **node**. One simple solution is to add a symlink:

```
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

Note: The list of packages above is maintained by hand and might fall behind, despite our best efforts. If you suspect something is missing, look at `tooling/docker/dev/set_up_ubuntu.sh` in the DXR source tree, which does the actual setup of the included container and is automatically tested.

Additional Installation

You'll need to install the JavaScript plugin for elasticsearch on your elasticsearch server (regardless of what type of code you're indexing). The plugin version you need depends on your version of elasticsearch (see https://github.com/elastic/elasticsearch-lang-javascript). See `tooling/docker/es/Dockerfile` for the command currently being used to install the plugin in our container, something like:

```
sudo /usr/share/elasticsearch/bin/plugin --install elasticsearch/elasticsearch-lang-
↪javascript/<version>
```

where you'll need to insert the appropriate `<version>`.

(The JavaScript plugin can be uninstalled with `sudo /usr/share/elasticsearch/bin/plugin remove lang-javascript`.)

To get all of the DXR tests to pass, or if you're indexing rust code, you'll also need to install rust. Refer to `tooling/docker/dev/set_up_common.sh` for the currently recommended install command, something like:

```
curl -s https://static.rust-lang.org/rustup.sh | sh -s -- --channel=nightly --date=
↪<date> --yes
```

Note: The 2015-06-14 version of rust has a bug on Fedora-based systems - see https://github.com/rust-lang/rust/issues/15684 for a fix if you're seeing shared library errors during rust compiles.

(Rust can be uninstalled with `sudo /usr/local/lib/rustlib/uninstall.sh`.)

Python Packages

You'll also need several third-party Python packages. In order to isolate the specific versions we need from the rest of the system, use Virtualenv:

```
virtualenv dxr_venv  # Create a new virtual environment.
source dxr_venv/bin/activate
```

You'll need to repeat that **activate** command each time you want to use DXR from a new shell.

1.4.2 Configuring Elasticsearch

Elasticsearch is the data store shared between the build and web servers. Obviously, they both need network access to it. ES tuning is a complex art, but these pointers should start you off with reasonable performance:

- Give ES its own server. It loves RAM and IO speed. If you want high availability or need more power than one machine can provide, set up a cluster.
- Configure the following in `/etc/elasticsearch/elasticsearch.yml`:
  - Set `bootstrap.mlockall` to `true`. You don't want any swapping.
  - Set `script.disable_dynamic` to `false`. This enables DXR's use of the JavaScript plugin.
  - Whether you intend to set up a cluster or not, beware that ES makes friends all too easily. Be sure to change the `cluster.name` to something unusual and disable autodiscovery by setting `discovery.zen.ping.multicast.enabled` to `false`, instead specifying your cluster members directly in `discovery.zen.ping.unicast.hosts`.
- And set the following in `/etc/default/elasticsearch` (for debian-based systems) or `/etc/sysconfig/elasticsearch` (for RPM-based distributions):
  - Crank up your kernel's max file descriptors:

```
MAX_OPEN_FILES=65535
MAX_LOCKED_MEMORY=unlimited
```

  - Set `ES_HEAP_SIZE` to half of your system RAM, not exceeding 32GB (because at that point the JVM can no longer use compressed pointers). Giving it one big chunk of RAM up front will avoid heap fragmentation and costly reallocations. The remaining memory will easily be filled by the OS's file cache as it tussles with Lucene indices.
  - If you have storage constraints, you may want to set `DATA_DIR` and `LOG_DIR` to control where elasticsearch puts its data and logs; the defaults are `/var/lib/elasticsearch` and `/var/log/elasticsearch`. Elasticsearch doesn't require much log space...until things go wrong.
- It is often recommended to use Oracle's JVM, but OpenJDK works fine.

DXR will create one index per indexed tree per *format version*. Reindexing a tree automatically replaces the old index with the new one as its last step. This happens atomically. Be sure there's enough space on the cluster to hold both the old and new indices at once during indexing.

1.4.3 Building

First, arrange for the correct versions of **`llvm-config`**, **`clang`**, and **`clang++`** to be available under those names, whether by a mechanism like Debian's alternatives system or with symlinks.

Then, activate the Python virtualenv you made above if you haven't already in your current login session:

```
source dxr_venv/bin/activate
```

Next, build DXR from its top-level directory:

```
make
```

It will build `libclang-index-plugin.so` in `dxr/plugins/clang`, compile the JavaScript-based templates, cache-bust the static assets, and install the Python dependencies.

1.4.4 Installation and Tests

Once you've built it, install DXR in the activated virtualenv:

```
pip install --no-deps .
```

Note: If you intend to develop DXR itself, run `pip install --no-deps -e .` instead. Otherwise, pip will make a copy of the code, severing its relationship with the source checkout.

To ensure everything has built correctly and that elasticsearch and other dependencies are installed and running correctly, you can run the tests. Make sure elasticsearch is started first, of course.

```
make test
```

1.4.5 Indexing

Now that we've got DXR installed on both the build and web machines, let's talk about just the build server for a moment.

As in *Getting Started*, copy your projects' source trees to the build server, and create a config file. (See *Configuration* for details.) Then, kick off the indexing process:

```
dxr index --config dxr.config
```

Note: You can also append one or more tree names to index just those trees. This is useful for parallelization across multiple build servers.

Generally, you use something like cron or Jenkins to repeat indexing on a schedule or in response to source-tree changes.

1.4.6 Serving Your Index

Now let's set up the web server. Here we have some alternatives.

dxr serve

`dxr serve` runs a tiny web server for publishing an index. Though it is underpowered for production use, it can come in handy for testing that the index was built properly and DXR's dependencies are installed:

```
dxr serve
```

Then visit http://localhost:8000/.

Apache and mod_wsgi

DXR is also a WSGI application and can be deployed on Apache with mod_wsgi, on uWSGI, or on any other web server that supports the WSGI protocol.

The main mod_wsgi directive is WSGIScriptAlias, and the DXR WSGI application is defined in `dxr/wsgi.py`, so an example Apache directive might look something like this:

```
WSGIScriptAlias / /path/to/dxr/dxr/wsgi.py
```

You must also specify the path to the config file. This is done with the `DXR_CONFIG` environment variable. For example, add this to your Apache configuration:

```
SetEnv DXR_CONFIG /path/to/dxr.config
```

Because we used virtualenv to install DXR's runtime dependencies, add the path to the virtualenv to your Apache configuration as well:

```
WSGIPythonHome /path/to/dxr_venv
```

Note that the WSGIPythonHome directive is allowed only in the server config context, not in the virtual host context. It's analogous to running virtualenv's **`activate`** command.

Finally, make sure mod_wsgi is installed and enabled. Then, restart Apache:

```
sudo service apache2 stop
sudo service apache2 start
```

Note: Changes to `/etc/apache2/envvars` don't take effect if you run only **`sudo service apache2 restart`**.

Additional configuration might be required, depending on your version of Apache, your other Apache configuration, and where DXR is installed. For example, if you can't access your DXR index and your Apache error log contains lines like `client denied by server configuration: /path/to/dxr/dxr/wsgi.py`, try adding this to your Apache configuration:

```
<Directory /path/to/dxr/dxr>
   Require all granted
</Directory>
```

Here is a complete example config, for reference:

```
WSGIPythonHome /home/dxr/dxr/venv
<VirtualHost *:80>
    # Serve static resources, like CSS and images, with plain Apache:
    Alias /static/ /home/dxr/dxr/dxr/static/

    # Tell DXR where its config file is:
    SetEnv DXR_CONFIG /home/dxr/dxr/tests/test_basic/dxr.config

    WSGIScriptAlias / /usr/local/lib/python2.7/site-packages/dxr/dxr.wsgi
</VirtualHost>
```

uWSGI

uWSGI is the new hotness and well worth considering. The first person to deploy DXR under uWSGI should document it here.

1.4.7 Upgrading

To update to a new version of DXR...

1. Update your DXR clone:

```
git pull origin master
```

2. Delete your old virtual env:

```
rm -rf /path/to/dxr_venv
```

3. Repeat these parts of the installation:
   (a) *Python Packages*
   (b) *Building*
   (c) *Installation and Tests*

1.5 Use

1.5.1 Querying

DXR queries are almost entirely text-based. In addition to being fast to input for experienced users, having an all-text representation invites handy tricks like Firefox keyword bookmarks.

A DXR query is a series of space-delimited *terms*:

- *Filtered terms* are structured as `<filter name>:<argument>`:
  - `callers:frobulate`
  - `var:num_caribou`

  Everything but plain text search is done using filtered terms.
- *Text terms* are just bare text and do simple substring matching:
  - `hello`
  - `three independent words`

All terms, filtered or not, are ANDed together, and lines matching all of them are returned as results.

Quoting

Single and double quotes can be used in filter arguments and in text terms to help express literal spaces and other oddities. Singles can contain doubles, doubles can contain singles, and each kind can contain itself if backslash-escaped:

- A phrase with a space: `"Hello, world"`
- Quotes in a plain text search, taken as literals since they're not leading: `id="whatShouldIDoContent"`
- Double quotes inside single quotes, as a filter argument: `regexp:'"wh(at|y)'`
- Backslash escaping: `"I don't \"believe\" in fairies."`

1.5.2 Highlighting

Source code views support highlighting lines, runs of lines, and even multiple runs of lines at once.

There are four ways to highlight. Each updates the hash portion of the URL so the highlighted regions are maintained when a page is bookmarked or shared via chat, bug reports, etc.

single click Single-click a line to select it. Click it again to deselect it. Single-clicking a line will also deselect all other lines.

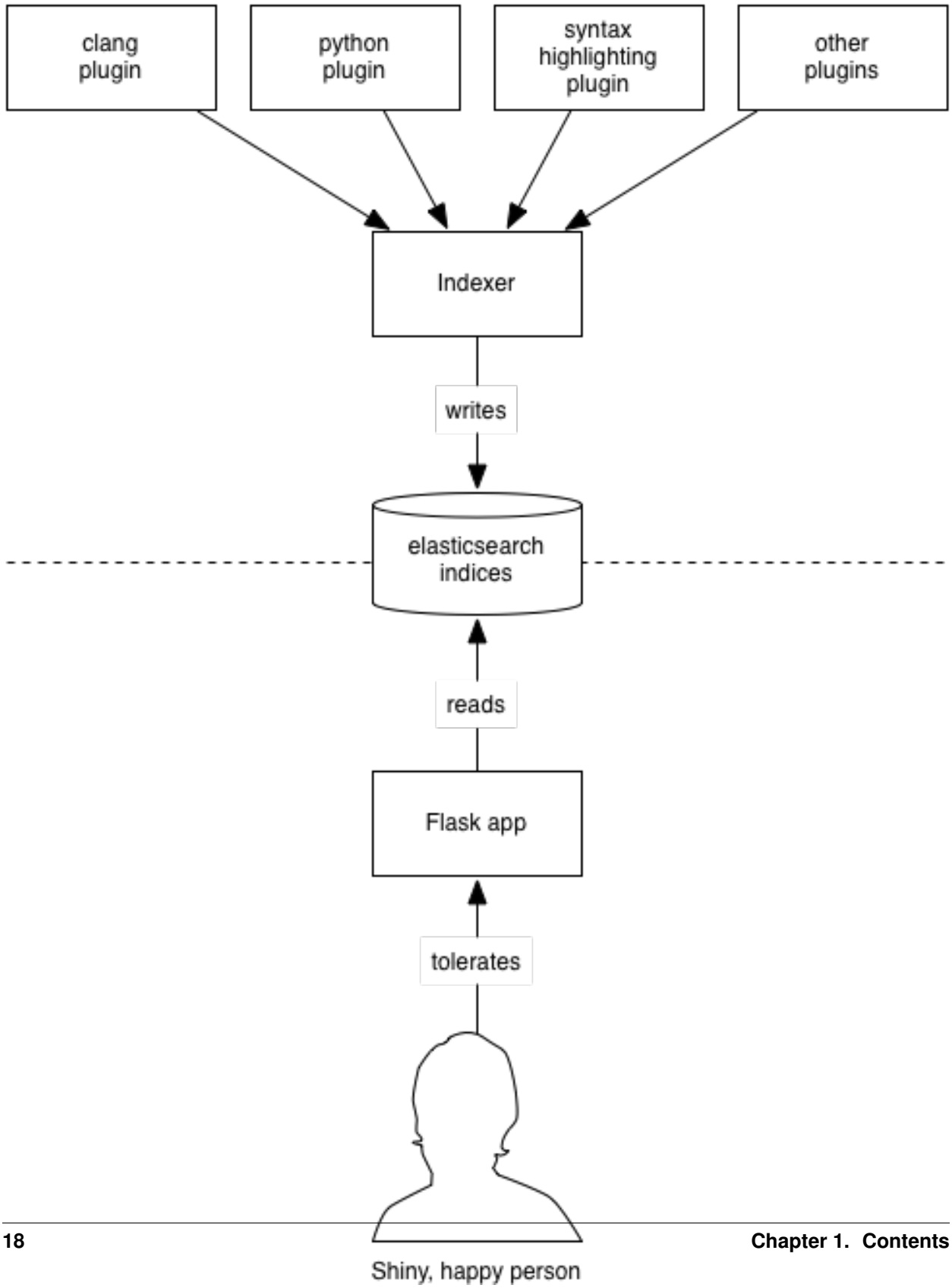
single click then shift-click After selecting a single line, hold Shift, and click a line above or below it to highlight the entire range between.

control- or command-click Hold Control or Command (depending on your OS) while clicking a line to add it to the set of already highlighted lines. Do it again to deselect it.

control- or command-click, then shift-click After selecting one or more lines, use Control- or Command-Click to highlight the first in a new range of lines. Then, Shift-click, and the second range will be added to the existing highlighted set.

1.6 Development

1.6.1 Architecture



DXR divides into 2 halves, with stored indices in the middle:

1. The indexer, run via **`dxr index`**, is a batch job which analyzes code and builds indices in elasticsearch, one per tree, plus a *catalog index* that keeps track of them. The indexer hosts various plugins which handle everything from syntax coloring to static analysis.

   Generally, the indexer is kicked off asynchronously—often even on a separate machine—by cron or a build system. It's up to deployers to come up with strategies that make sense for them.

2. The second half is a Flask web application which lets users run queries. **`dxr serve`** runs a toy instance of the application for development purposes; a more robust method should be used for *Deployment*.

How Indexing Works

We store every line of source code as an elasticsearch document of type `line` (hereafter called a "LINE doc" after the name of the constant used in the code). This lends itself to the per-line search results DXR delivers. In addition to the text of the line, indexed into trigrams for fast substring and regex search, a LINE doc contains some structural data.

- First are *needles*, search targets that structural queries can hunt for. For example, if we indexed the following Python source code, the indicated (simplified) needles might be attached:

  ```
  def frob():      # py-function: frob
      nic(ate())   # py-callers:  [nic, ate]
  ```

  If the user runs the query `function:frob`, we look for LINE docs with "frob" in their "py-function" properties. If the user runs the query `callers:nic`, we look for docs with "py-callers" properties containing "nic".

  These needles are offered up by plugins via the *`needles_by_line()`* API. For the sake of sanity, we've settled on the convention of a language prefix for language-specific needles. However, the names are technically arbitrary, since the plugin emitting the needle is also its consumer, through the implementation of a *`Filter`*.

- Also attached to a LINE doc are offsets/metadata pairs that attach CSS classes and contextual menus to various spans of the line. These also come out of plugins, via *`refs()`* and *`regions()`*. Views of entire source-code files are rendered by stitching multiple LINE docs together.

The other major kind of entity is the FILE doc. These support directory listings and the storage of per-file rendering data like navigation-pane entries (given by *`links()`*) or image contents. FILE docs may also contain needles, supporting searches like `ext:cpp` which return entire files rather than lines. Plugins provide these needles via *`needles()`*.

1.6.2 Setting Up

Here is the fastest way to get hacking on DXR.

Downloading DXR

Using git, clone the DXR repository:

```
git clone https://github.com/mozilla/dxr.git
```

Booting And Building

DXR runs only on Linux at the moment (and possibly other UNIX-like operating systems). The easiest way to get things set up is to use the included, preconfigured Docker setup. If you're not running Linux on your host machine, you'll need a virtualization provider. We recommend VirtualBox.

After you've installed VirtualBox (or ignored that bit because you're on Linux), grab the three Docker tools you'll need: docker, docker-compose, and, if you're not on Linux, docker-machine. If you're running the homebrew package manager on the Mac, this is as easy as...

```
brew install docker docker-compose docker-machine
```

Otherwise, visit the Docker Engine page for instructions.

Next, unless you're already on Linux, you'll need to spin up a Linux VM to host your Docker containers:

```
docker-machine create --driver virtualbox --virtualbox-disk-size 50000 --virtualbox-
↪cpu-count 2 --virtualbox-memory 512 default
eval "$(docker-machine env default)"
```

Feel free to adjust the resource allocation numbers above as you see fit.

Note: Next time you reboot (or run `make docker_stop`), you'll need to restart the VM:

```
docker-machine start default
```

And each time you use a new shell, you'll need to set the environment variables that tell Docker how to find the VM:

```
eval "$(docker-machine env default)"
```

When you're done with DXR and want to reclaim the RAM taken by the VM, run...

```
make docker_stop
```

Now you're ready to fire up DXR's Docker containers, one to run elasticsearch and the other to interact with you, index code, and serve web requests:

```
make shell
```

This drops you at a shell prompt in the interactive container. Now you can build DXR and run the tests to make sure it works. Type this at the prompt *within the container*:

```
# Within the docker container...
make test
```

Running A Test Index

The folder-based test cases make decent workspaces for development, suitable for manually trying out your changes. `test_basic` is a good one to start with. To get it running...

```
cd ~/dxr/tests/test_basic
dxr index
dxr serve -a
```

If you're using `docker-machine`, run `docker-machine ip default` to find the address of your VM. Then surf to http://*that IP address*:8000/ from the host machine, and explore the index. If you're not using `docker-machine`, the index should be accessible from http://localhost:8000/.

When you're done, stop the server with `Control-C`.

1.6.3 Workflow

The repository on your host machine is mirrored over to the interactive container via Docker volume mounting. Changes you make in the DXR repository on your host machine will be instantly available within `/home/dxr/dxr` on the container and vice versa, so you can edit using your usual tools on the host and still use the container to run DXR.

After making changes to DXR, a build step is sometimes needed to see the effects of your work:

Changes to C++ code or to HTML templates in the nunjucks folder: `make` (at the root of the project)

Changes to the format of the elasticsearch index: Re-run `dxr index` inside your test folder (e.g., `tests/test_basic`). Before committing, you should increment the *format version*.

Stop **`dxr serve`**, run any applicable build steps, and then fire up the server again. If you're changing Python code that runs only at request time, you shouldn't need to do anything; **`dxr serve`** will notice and restart itself a few seconds after you save.

1.6.4 Coding Conventions

Follow PEP 8 for Python code, but don't sweat the line length too much. Follow PEP 257 for docstrings, and use Sphinx-style argument documentation. Single quotes are preferred for strings; use 3 double quotes for docstrings and multiline strings or if the string contains a single quote.

1.6.5 Testing

DXR has a fairly mature automated testing framework, and all server-side patches should come with tests. (Tests for client-side contributions are welcome as well, but we haven't got the harness set up yet.)

Writing Tests for DXR

DXR supports two kinds of integration tests:

1. A lightweight sort with a single file worth of analyzed code. This kind stores the code as a Python string within a subclass of `SingleFileTestCase`. At test time, it instantiates the file on disk in a temp folder, builds it, and makes assertions about it. If the `stop_for_interaction` class variable is falsy (the default), it then deletes the index. If you want to browse the instance manually for troubleshooting, set this to `True`.
2. A heavier sort of test: a folder containing one or more source trees and a DXR config file. These are useful for tests that require a multi-file tree to analyze or more than one tree. `test_ignores` is an example. Within these folders are also one or more Python files containing subclasses of `DxrInstanceTestCase` which express the actual tests. These trees can be built like any other using `dxr index`, in case you want to do manual exploration.

Running the Tests

To run all the tests, run this from the root of the DXR repository (in the container):

```
make test
```

To run just the tests in `tests/test_functions.py`...

```
nosetests tests/test_functions.py
```

To run just the tests from a single class...

```
nosetests tests/test_functions.py:ReferenceTests
```

To run a single test...

```
nosetests tests/test_functions.py:ReferenceTests.test_functions
```

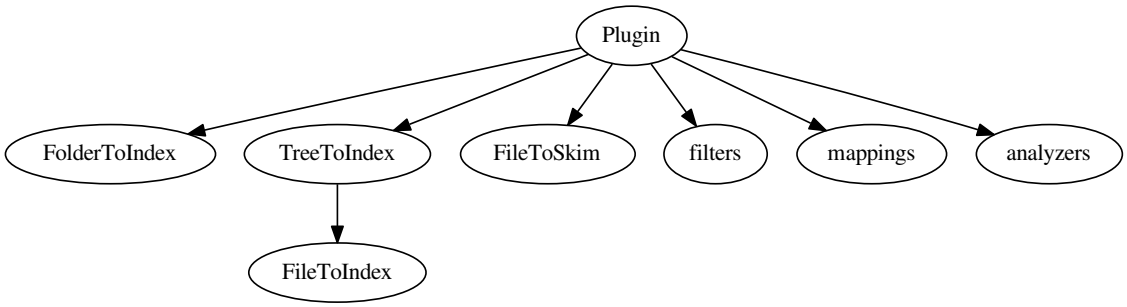
If you have trouble, make sure you didn't mistranscribe any colons or periods.

To omit the often distracting elasticsearch logs that nose typically presents when a test fails, add the `--nologcapture` flag.

1.6.6 Writing Plugins

Plugins are the way to add new types of analysis, indexing, searching, or display to DXR. In fact, even DXR's basic capabilities, such as text search and syntax coloring, are implemented as plugins. Want to add support for a new language? A new kind of search to an existing language? A new kind of contextual menu cross-reference? You're in the right place.

At the top level, a *`Plugin`* class binds together a collection of subcomponents which do the actual work:



Registration

A Plugin class is registered via a setuptools entry point called `dxr.plugins`. For example, here are the registrations for the built-in plugins, from DXR's own `setup.py`:

```
entry_points={'dxr.plugins': ['urllink = dxr.plugins.urllink',
                              'buglink = dxr.plugins.buglink',
                              'clang = dxr.plugins.clang',
```

```
                                'omniglot = dxr.plugins.omniglot',
                                'pygmentize = dxr.plugins.pygmentize']},
```

The keys in the key/value pairs, like "urllink" and "buglink", are the strings the deployer can use in the `enabled_plugins` config directive to turn them on or off. The values, like "dxr.plugins.urllink", can point to either...

1. A *`Plugin`* class which itself points to filters, skimmers, indexers, and such. This is the explicit approach—more lines of code, more opportunities to buck convention—and thus not recommended in most cases. The *`Plugin`* class itself is just a dumb bag of attributes whose only purpose is to bind together a collection of subcomponents that should be used together.
2. Alternatively, an entry point value can point to a module which contains the subcomponents of the plugin, each conforming to a naming convention by which it can be automatically found. This method saves boilerplate and should be used unless there is a compelling need otherwise. Behind the scenes, an actual Plugin object is constructed implicitly: see *`from_namespace()`* for details of the naming convention.

Here is the Plugin object's API, in case you do decide to construct one manually:

class `dxr.plugins.`**`Plugin`**(*filters=None*, *folder_to_index=None*, *tree_to_index=None*, *file_to_skim=None*, *mappings=None*, *analyzers=None*, *direct_searchers=None*, *refs=None*, *badge_colors=None*, *config_schema=None*)

Top-level entrypoint for DXR plugins

A Plugin is an indexer, skimmer, filter set, and other miscellany meant to be used together; it is the deployer-visible unit of pluggability. In other words, there is no way to subdivide a plugin via configuration; there would be no sense running a plugin's filters if the indexer that was supposed to extract the requisite data never ran.

If the deployer should be able to independently enable parts of your plugin, consider exposing those as separate plugins.

Note that Plugins may be instantiated multiple times; don't assume otherwise.

Parameters

- **`filters`** – A list of filter classes
- **`folder_to_index`** – A `FolderToIndex` subclass
- **`tree_to_index`** – A `TreeToIndex` subclass
- **`file_to_skim`** – A `FileToSkim` subclass
- **`mappings`** – Additional Elasticsearch mapping definitions for all the plugin's elasticsearch-destined data. A dict with keys for each doctype and values reflecting the structure described at http://www.elastic.co/guide/en/elasticsearch/reference/current/indices-put-mapping.html. Since a FILE-domain query will be promoted to a LINE query if any other query term triggers a line-based query, it's important to keep field names and semantics the same between lines and files. In other words, a LINE mapping should generally be a superset of a FILE mapping.
- **`analyzers`** – Analyzer, tokenizer, and token and char filter definitions for the elasticsearch mappings. A dict with keys "analyzer", "tokenizer", etc., following the structure outlined at http://www.elastic.co/guide/en/elasticsearch/reference/current/analysis.html.
- **`direct_searchers`** – Functions that provide direct search capability. Each must take a single query term of type 'text', return an elasticsearch filter clause to run

against LINEs, and have a `direct_search_priority` attribute. Filters are tried in order of increasing priority. Return None from a direct searcher to skip it.

Note: A more general approach may replace direct search in the future.

- **refs** – An iterable of *Ref* subclasses supported by this plugin. This is used at request time, to turn abreviated ES index data back into HTML.
- **badge_colors** – Mapping of Filter.lang -> color for menu badges.
- **config_schema** – A validation schema for this plugin's configuration. See https://pypi.python.org/pypi/schema/ for docs.

`mappings` and `analyzers` are recursively merged into other plugins' mappings and analyzers using the algorithm described at `deep_update()`. This is mostly intended so you can add additional kinds of indexing to fields defined in the core plugin using multifields. Don't go too crazy monkeypatching the world.

classmethod from_namespace(*namespace*)

Construct a Plugin whose attrs are populated by naming conventions.

Parameters **namespace** – A namespace from which to pick components

Filters are taken to be any class whose name ends in "Filter" and doesn't start with "_".

Refs are taken to be any class whose name ends in "Ref" and doesn't start with "_".

The **tree indexer** is assumed to be called "TreeToIndex". If there isn't one, one will be constructed which does nothing but delegate to the class called `FileToIndex` (if there is one) when `file_to_index()` is called on it.

The **file skimmer** is assumed to be called "FileToSkim".

Mappings are pulled from `mappings` attribute and **analyzers** from `analyzers`.

If these rules don't suit you, you can always instantiate a Plugin yourself.

Actual plugin functionality is implemented within file indexers, tree indexers, folder indexers, filters, and skimmers.

Folder Indexers

class dxr.indexers.**FolderToIndex**(*plugin_name*, *tree*, *path*)

The FolderToIndex generates needles for folders and provides an optional list of headers to display in browse view as *browse_headers*.

Tree Indexers

class dxr.indexers.**TreeToIndex**(*plugin_name*, *tree*, *vcs_cache*)

A TreeToIndex performs build environment setup and teardown and serves as a repository for scratch data that should persist across an entire indexing run.

Instances must be pickleable so as to make the journey to worker processes. You might also want to keep the size down. It takes on the order of 2s for a 150MB pickle to make its way across process boundaries, including pickling and unpickling time. For this reason, we send the TreeToIndex once and then have it index several files before sending it again.

Parameters

- **tree** – The configuration of the tree to index: a TreeConfig

- **vcs_cache** – A `VcsCache` that describes any VCSes used by this tree. May be None if tree does not contain any VCS repositories.

environment(*vars*)
Return environment variables to add to the build environment.

This is where the environment is commonly twiddled to activate and parametrize compiler plugins which dump analysis data.

Parameters **vars** – A dict of the already-set variables. You can make decisions based on these.

You may return a new dict or scribble on `vars` and return it. In either case, the returned dict is merged into those from other plugins, with later plugins taking precedence in case of conflicting keys.

file_to_index(*path*, *contents*)
Return an object that provides data about a given file.

Return an object conforming to the interface of `FileToIndex`, generally a subclass of it.

Parameters

- **path** – A path to the file to index, relative to the tree's source folder
- **contents** – What's in the file: unicode if we managed to guess an encoding and decode it, None otherwise

Return None if there is no indexing to do on the file.

Being a method on TreeToIndex, this can easily pass along the location of a temp directory or other shared setup artifacts. However, beware of passing mutable things; while the FileToIndex can mutate them, visibility of those changes will be limited to objects in the same worker process. Thus, a TreeToIndex-dwelling dict might be a suitable place for a cache but unsuitable for data that can't evaporate.

If a plugin omits a TreeToIndex class, *`from_namespace()`* constructs one dynamically. The method implementations of that class are inherited from this class, with one exception: a `file_to_index()` method is dynamically constructed which returns a new instance of the `FileToIndex` class the plugin defines, if any.

post_build()
Hook called after the tree's build command completes

This is a good place to do any whole-program analysis, storing it on me or on disk.

pre_build()
Hook called before the tree's build command is run

This is a good place to make a temp folder to dump said data in. You can stash away a reference to it on me so later methods can find it.

File Indexers

class `dxr.indexers.`**FileToIndex**(*path*, *contents*, *plugin_name*, *tree*)
A source of search and rendering data about one source file

Analyze a file or digest an analysis that happened at compile time.

Parameters

- **path** – The (bytestring) path to the file to index, relative to the tree's source folder
- **contents** – What's in the file: unicode if we managed to guess at an encoding and decode it, None otherwise. Don't return any by-line data for None; the framework won't have succeeded in breaking up the file by line for display, so there will be no useful UI for

those data to support. Think more along the lines of returning EXIF data to search by for a JPEG. For unicode, split the file into lines using universal newlines (`dxr.utils.split_content_lines()`); that's what the rest of the framework expects.

- **tree** – The `TreeConfig` of the tree to which the file belongs

Initialization-time analysis results may be socked away on an instance var. You can think of this constructor as a per-file post-build step. You could do this in a different method, using memoization, but doing it here makes for less code and less opportunity for error.

FileToIndex classes of plugins may take whatever constructor args they like; it is the responsibility of their TreeToIndex objects' *file_to_index()* methods to supply them. However, the `path` and `contents` instance vars should be initialized and have the above semantics, or a lot of the provided convenience methods and default implementations will break.

needles()
Return an iterable of key-value pairs of search data about the file as a whole: for example, modification date or file size.

Each pair becomes an elasticsearch property and its value. If the framework encounters multiple needles of the same key (whether coming from the same plugin or different ones), all unique values will be retained using an elasticsearch array.

needles_by_line()
Return per-line search data for one file: for example, markers that indicate a function called "foo" is defined on a certain line.

Yield an iterable of key-value pairs for each of a file's lines, one iterable per line, in order. The data might be data to search on or data stowed away for a later realtime thing to generate refs or regions from. In any case, each pair becomes an elasticsearch property and its value.

If the framework encounters multiple needles of the same key on the same line (whether coming from the same plugin or different ones), all unique values will be retained using an elasticsearch array. Values may be dicts, in which case common keys get merged by `append_update()`.

This method is not called on symlink files, to maintain the illusion that they do not have contents, seeing as they cannot be viewed in file browsing.

FileToIndex also has all the methods of its superclass, *FileToSkim*.

Looking Inside Elasticsearch

While debugging a file indexer, it can help to see what is actually getting into elasticsearch. For example, if you are debugging *needles_by_line()*, you can see all the data attached to each line of code (up to 1000) with this curl command:

```
curl -s -XGET "http://localhost:9200/dxr_10_code/line/_search?pretty&size=1000"
```

Be sure to replace "dxr_10_code" with the name of your DXR index. You can see which indexes exist by running...

```
curl -s -XGET "http://localhost:9200/_status?pretty"
```

Similarly, when debugging *needles()*, you can see all the data attached to files-as-a-whole with...

```
curl -s -XGET "http://localhost:9200/dxr_10_code/file/_search?pretty&size=1000"
```

File Skimmers

class dxr.indexers.**FileToSkim**(*path*, *contents*, *plugin_name*, *tree*, *file_properties=None*, *line_properties=None*)

A source of rendering data about a file, generated at request time

This is appropriate for unindexed files (such as old revisions pulled out of a VCS) or for data so large or cheap to produce that it's a bad tradeoff to store it in the index. An instance of me is mostly an opportunity for a shared cache among my methods.

Parameters

- **path** – The (bytestring) conceptual path to the file, relative to the tree's source folder. Such a file might not exist on disk. This is useful mostly as a hint for syntax coloring.
- **contents** – What's in the file: unicode if we knew or successfully guessed an encoding, None otherwise. Don't return any by-line data for None; the framework won't have succeeded in breaking up the file by line for display, so there will be no useful UI for those data to support. In fact, most skimmers won't be be able to do anything useful with None at all. For unicode, split the file into lines using universal newlines (`dxr.utils.split_content_lines()`); that's what the rest of the framework expects.
- **tree** – The `TreeConfig` of the tree to which the file belongs

If the file is indexed, there will also be...

Parameters

- **file_properties** – Dict of file-wide needles emitted by the indexer
- **line_properties** – List of per-line needle dicts emitted by the indexer

absolute_path()

Return the (bytestring) absolute path of the file to skim.

Note: in skimmers, the returned path may not exist if the source folder moved between index and serve time.

annotations_by_line()

Yield extra user-readable information about each line, hidden by default: compiler warnings that occurred there, for example.

Yield a list of annotation maps for each line:

```
{'title': ..., 'class': ..., 'style': ...}
```

char_offset(*row*, *col*)

Return the from-BOF unicode char offset for the char at the given row and column of the file we're indexing.

This is handy for translating row- and column-oriented input to the format *refs()* and *regions()* want.

Parameters

- **row** – The 1-based line number, according to splitting in universal newline mode
- **col** – The 0-based column number

contains_text()

Return whether this file can be decoded and divided into lines as text. Empty files contain text.

This may come in handy as a component of your own *is_interesting()* methods.

is_interesting()
Return whether it's worthwhile to examine this file.

For example, if this class knows about how to analyze JS files, return True only if `self.path.endswith('.js')`. If something falsy is returned, the framework won't call data-producing methods like *links()*, *refs()*, etc.

The default implementation selects only text files that are not symlinks. Note: even if a plugin decides that symlinks are interesting, it should remember that links, refs, regions and by-line annotations will not be called because views of symlinks redirect to the original file.

is_link()
Return whether the file is a symlink.

Note: symlinks are never displayed in file browsing; a request for a symlink redirects to its target.

links()
Return an iterable of links for the navigation pane:

```
(sort order, heading, [(icon, title, href), ...])
```

File views will replace any {{line}} within the href with the last-selected line number.

refs()
Provide cross references for various spans of text, accessed through a context menu.

Yield an ordered list of extents and menu items:

```
(start, end, ref)
```

`start` and `end` are the bounds of a slice of a Unicode string holding the contents of the file. (`refs()` will not be called for binary files.)

`ref` is a *Ref*.

regions()
Yield instructions for syntax coloring and other inline formatting of code.

Yield an ordered list of extents and CSS classes (encapsulated in *Region* instances):

```
(start, end, Region)
```

`start` and `end` are the bounds of a slice of a Unicode string holding the contents of the file. (`regions()` will not be called for binary files.)

class dxr.lines.**Ref**(*tree*, *menu_data*, *hover=None*, *qualname=None*, *qualname_hash=None*)
Abstract superclass for a cross-reference attached to a run of text

Carries enough data to construct a context menu, highlight instances of the same symbol, and show something informative on hover.

Parameters

- **menu_data** – Arbitrary JSON-serializable data from which we can construct a context menu
- **hover** – The contents of the <a> tag's title attribute. (The first one wins.)
- **qualname** – A hashable unique identifier for the symbol surrounded by this ref, for highlighting
- **qualname_hash** – The hashed version of `qualname`, which you can pass instead of `qualname` if you have access to the already-hashed version

es()
Return a serialization of myself to store in elasticsearch.

static es_to_triple(*es_data*, *tree*)
Convert ES-dwelling ref representation to a (start, end, *Ref* subclass) triple.

Return a subclass of Ref, chosen according to the ES data. Into its attributes "menu_data", "hover" and "qualname_hash", copy the ES properties of the same names, JSON-decoding "menu_data" first.

Parameters

- **es_data** – An item from the array under the 'refs' key of an ES LINE document
- **tree** – The `TreeConfig` representing the tree from which the `es_data` was pulled

menu_items()
Return an iterable of menu items to be attached to a ref.

Return an iterable of dicts of this form:

```
{
    html: the HTML to be used as the menu item itself
    href: the URL to visit when the menu item is chosen
    title: the tooltip text given on hovering over the menu item
    icon: the icon to show next to the menu item: the name of a PNG
        from the ``icons`` folder, without the .png extension
}
```

Typically, this pulls data out of `self.menu_data`.

opener()
Emit the opening anchor tag for a cross reference.

Menu item text, links, and metadata are JSON-encoded and dumped into a data attr on the tag. JS finds them there and creates a menu on click.

class dxr.lines.**Region**(*css_class*)
A tag with a CSS class, wrapped around a run of text

classmethod es_to_triple(*es_region*)
Convert ES-dwelling region representation to a (start, end, *Region*) triple.

Filters

class dxr.filters.**Filter**(*term*, *enabled_plugins*)
A provider of search strategy and highlighting

Filter classes, which roughly correspond to the items in the Filters dropdown menu, tell DXR how to query the data stored in elasticsearch by `needles()` and `needles_by_line()`. An instance is created for each query term whose `name` matches and persists through the querying and highlighting phases.

This is an optional base class that saves code on many filters. It also serves to document the filter API.

Variables

- **name** – The string prefix used in a query term to activate this filter. For example, if this were "path", this filter would be activated for the query term "path:foo". Multiple filters can be registered against a single name; they are ORed together. For example, it is good practice for a language plugin to query against a language specific needle (like "js-function") but register against the more generic "function" here. (This allows us to do language-specific queries.)

- **domain** – Either LINE or FILE. LINE means this filter returns results that point to specific lines of files; FILE means they point to files as a whole. Default: LINE.
- **description** – A description of this filter for the Filters menu: unicode or Markup (in case you want to wrap examples in `<code>` tags). Of filters having the same name, the description of the first one encountered will be used. An empty description will hide a filter from the menu. This should probably be used only internally, by the TextFilter.
- **union_only** – Whether this filter will always be ORed with others of the same name, useful for filters where the intersection would always be empty, such as extensions
- **is_reference** – Whether to include this filter in the "ref:" aggregate filter
- **is_identifier** – Whether to include this filter in the "id:" aggregate filter

This is a good place to parse the term's arg (if it requires further parsing) and stash it away on the instance.

Parameters

- **term** – a query term as constructed by a `QueryVisitor`
- **enabled_plugins** – an iterable of the enabled *`Plugin`* instances, for use by filters that build upon the filters provided by plugins

Raise `BadTerm` to complain to the user: for instance, about an unparseable term.

filter()

Return the ES filter clause that applies my restrictions to the found set of lines (or files and folders, if `domain` is FILES).

To quietly do no filtration, return None. This would be suitable for `path:*`, for example.

To do no filtration and complain to the user about it, raise `BadTerm`.

We might even make this return a list of filter clauses, for things like the RegexFilter which want a bunch of match_phrases and a script.

highlight_content(*result*)

Return an unsorted iterable of extents that should be highlighted in the `content` field of a search result.

Parameters **result** – A mapping representing properties from a search result, whether a file or a line. With access to all the data, you can, for example, use the extents from a 'c-function' needle to inform the highlighting of the 'content' field.

highlight_path(*result*)

Return an unsorted iterable of extents that should be highlighted in the `path` field of a search result.

Parameters **result** – A mapping representing properties from a search result, whether a file or a line. With access to all the data, you can, for example, use the extents from a 'c-function' needle to inform the highlighting of the 'content' field.

Mappings

When you're laying down data to search upon, it's generally not enough just to write *`needles()`* or *`needles_by_line()`* implementations. If you want to search case-insensitively, for example, you'll need elasticsearch to fold your data to lowercase. (Don't fall into the trap of doing this in Python; the Lucene machinery behind ES is better at the complexities of Unicode.) The way you express these instructions to ES is through mappings and analyzers.

ES *mappings* are schemas which specify type of data (string, int, datetime, etc.) and how to index it. For example, here is an excerpt of DXR's core mapping, defined in the `core` plugin:

```
mappings = {
    # Following the typical ES mapping format, `mappings` is a hash keyed
    # by doctype. So far, the choices are ``LINE`` and ``FILE``.
    LINE: {
        'properties': {
            # Line number gets mapped as an integer. Default indexing is fine
            # for numbers, so we don't say anything explicitly.
            'number': {
                'type': 'integer'
            },

            # The content of the line itself gets mapped 3 different ways.
            'content': {
                # First, we store it as a string without actually putting it
                # into any ordered index structure. This is for retrieval and
                # display in search results, not for searching on:
                'type': 'string',
                'index': 'no',

                # Then, we index it in two different ways: broken into
                # trigrams (3-letter chunks) and either folded to lowercase or
                # not. This cleverness takes care of substring matching and
                # accelerates our regular expression search:
                'fields': {
                    'trigrams_lower': {
                        'type': 'string',
                        'analyzer': 'trigramalyzer_lower'
                    },
                    'trigrams': {
                        'type': 'string',
                        'analyzer': 'trigramalyzer'
                    }
                }
            }
        }
    },
    FILE: ...
}
```

Mappings follow exactly the same structure as required by ES's "put mapping" API. The choice of mapping types is also outlined in the ES documentation.

Warning: Since a FILE-domain query will be promoted to a LINE query if any other query term triggers a line-based query, it's important to keep field names and semantics the same between lines and files. In other words, a LINE mapping should generally be a superset of a FILE mapping. Otherwise, ES will guess mappings for the undeclared fields, and surprising search results will likely ensue. Worse, the bad guesses will likely happen intermittently.

The Format Version

In the top level of the `dxr` package (not the top of the source checkout, mind you) lurks a file called `format`. Its role is to facilitate the automatic deployment of new versions of DXR using **`dxr deploy`**. The format file contains an integer which represents the index format expected by **`dxr serve`**. If a change in the code requires a mapping or semantics change in the index, the format version must be incremented. In response, the deployment script will wait

until new indices, of the new format, have been built before deploying the change.

If you aren't sure whether to bump the format version, you can always build an index using the old code, then check out the new code and try to serve the old index with it. If it works, you're probably safe not bumping the version.

Analyzers

In Mappings, we alluded to custom indexing strategies, like breaking strings into lowercase trigrams. These strategies are called *analyzers* and are the final component of a plugin. ES has strong documentation on defining analyzers. Declare your analyzers (and building blocks of them, like tokenizers) in the same format the ES documentation prescribes. For example, the analyzers used above are defined in the core plugin as follows:

```
analyzers = {
    'analyzer': {
        # A lowercase trigram analyzer:
        'trigramalyzer_lower': {
            'filter': ['lowercase'],
            'tokenizer': 'trigram_tokenizer'
        },
        # And one for case-sensitive things:
        'trigramalyzer': {
            'tokenizer': 'trigram_tokenizer'
        }
    },
    'tokenizer': {
        'trigram_tokenizer': {
            'type': 'nGram',
            'min_gram': 3,
            'max_gram': 3
            # Keeps all kinds of chars by default.
        }
    }
}
```

1.6.7 Contributing Documentation

We use Read the Docs for building and hosting the documentation, which uses sphinx to generate HTML documentation from reStructuredText markup.

To edit documentation:

- Edit `*.rst` files in `docs/source/` in your local checkout. See reStructuredText primer for help with syntax.
- Use `cd ~/dxr/docs && make html` in the VM to preview the docs.
- When you're satisfied, submit the pull request as usual.

1.6.8 Troubleshooting

Why is my copy of DXR acting erratic, failing at searches, making requests for JS templates that shouldn't exist, and just gener
Did you run `python setup.py install` for DXR at some point? Never, ever do that in development; use `python setup.py develop` instead. Otherwise, you will end up with various files copied into your virtualenv, and your edits to the originals will have no effect.

How can I use pdb to debug indexing? In the DXR config file for the tree you're building, add `workers = 0` to the `[DXR]` section. That will keep DXR from spawning multiple worker processes, something pdb doesn't tolerate well.

I pulled a new version of the code that's supposed to have a new plugin (or I added one myself), but it's acting like it doesn't exis Re-run `python setup.py develop` to register the new setuptools entry point.

1.7 Appendix A: Indexing Firefox

As both a practical example and a specific reference, here is how to tweak the included container to build a DXR index of mozilla-central, the repository from which Firefox is built.

1.7.1 Increase Your RAM

Stop your containers, and increase the RAM and disk on your docker-machine VM (if using docker-machine). The compilation needs around 7GB. The temp files are 15GB, and the ES index and generated HTML are also on that order. It's also a good idea to add more virtual CPUs, up to the limit of your physical ones. On your host machine...

```
make docker_stop
docker-machine rm default
docker-machine create --driver virtualbox --virtualbox-disk-size 80000 --virtualbox-
↪cpu-count 4 --virtualbox-memory 8000 default

# Reset your shell variables:
eval "$(docker-machine env default)"

# And drop back into the DXR container:
make shell
```

1.7.2 Configure The Source Tree

1. Put a mozilla-central checkout in `/code` on the VM. This is a special, blessed folder that will not evaporate when the docker container exits. (If you decide to put it somewhere else, be sure your choice is reflected in `dxr.config` in Step 4.) You can use `hg clone` as documented at https://developer.mozilla.org/en-US/docs/Simple_Firefox_build.

Note: If using docker-machine and VirtualBox, keep your source code out of `/home/dxr/dxr`; VirtualBox's sharing of that folder between host and guest will kill your performance.

---

2. Have the compiler include the debug code so it can be analyzed. Put this in `/code/mozilla-central/mozconfig`:

```
ac_add_options --enable-debug
ac_add_options --disable-optimize
```

3. Get it ready to build:

```
cd /code/mozilla-central
./mach bootstrap
./mach mercurial-setup
```

4. Put this into a new `dxr.config` file. It doesn't matter where it is, but it's a good idea to keep it outside the checkout.

```
[DXR]
enabled_plugins=clang pygmentize

[mozilla-central]
source_folder=/code/mozilla-central
object_folder=/code/mozilla-central/obj-x86_64-unknown-linux-gnu
build_command=cd $source_folder && ./mach clobber && make -f client.mk build MOZ_
↪OBJDIR=$object_folder MOZ_MAKE_FLAGS="-s -j$jobs"
```

1.7.3 Bump Up Elasticsearch's RAM

1. In `tooling/docker/docker-compose.yml`, add an `environment` stanza like this:

```
es:
  build: ./es
  environment:
    ES_HEAP_SIZE: 2g
  ...
```

2. Run `make docker_es`.

1.7.4 Kick Off The Build

Within the Docker container (remember, `make shell`), in the folder where you put `dxr.config`, run this:

```
dxr index
```

This builds your source tree and indexes it into elasticsearch. You can then run `dxr serve -a` to spin up the web interface against it.

CHAPTER 2

Back Matter

2.1 Glossary

analyzer An elasticsearch indexing strategy. The design of these should be determined by how you plan to query the fields that use them.

catalog index The elasticsearch index which holds metadata about the other elasticsearch indices, which in turn represent source trees. The metadata includes *format version* and other frozen-at-index-time information.

filtered term A query term consisting of an explicit filter name and an argument, like `regexp:hi|hello` or `callers:frob`

format version A string (though usually looking like an int) signifying the index format. It is used to control deployments: **`dxr deploy`** never switches to a new version of the web-serving code until all indices have been brought up to the format version it requires. The format version is declared in `dxr/format`.

index The collected data used to answer queries about a tree and render the web-based UI. These are stored in elasticsearch and created by **`dxr index`**.

mapping An elasticsearch schema, declaring the type and indexing strategy for each field

needle A piece of arbitrary data attached to either an elasticsearch `line` or `file` doc. These are searched for when doing structural queries. Think of these as shining nuggets of information buried in the haystack of a codebase.

term A space-delimited part of a query

text term A query term without an explicit filter name, interpreted as raw text for a substring search

2.2 Icon Credits

DXR uses third-party icons from a variety of sources.

If you add an icon, please document its origin in this document. Feel free to use existing icons, but keep in mind that they use semantic naming. So don't use the `search` icon for zoom, as we may later change the search icon from a magnifying glass to, for example, binoculars.

2.2.1 From Silk

Following icons originates from Silk by Mark James, licensed under Creative Commons Attribution 2.5 License.

- `folder`
- `path_search`
- `exclude_path`
- `goto_folder`
- `page_white_find`
- `page_white_code`
- `page_white`
- `page_white_wrench`
- `buglink`
- `external_link`
- `mimetypes/php`
- `mimetypes/c`
- `mimetypes/build`
- `mimetypes/sh`
- `mimetypes/cs`
- `mimetypes/h`
- `mimetypes/css`
- `mimetypes/js`
- `mimetypes/rb`
- `mimetypes/txt`
- `mimetypes/cpp`
- `mimetypes/xml`
- `mimetypes/unknown`
- `mimetypes/ui`
- `mimetypes/conf`
- `mimetypes/java`
- `mimetypes/svg`
- `mimetypes/html`
- `mimetypes/iso`
- `mimetypes/vs`
- `mimetypes/image`
- `mimetypes/py` (mixed with official python logo)
- `mimetypes/mm` (Remixed by DXR developers)

2.2.2 From FatCow Hosting

Following icons originates from FatCow by FatCow hosting, licensed under Creative Commons Attribution 3.0 License.

2.2.3 From Fugue

Following icons originates from Fugue by Yusuke Kamiyamane, licensed under Creative Commons Attribution 3.0 License.

- `raw`
- `warning`
- `log`
- `blame`
- `diff`
- `search_warning`
- `regexp-search`
- `mimetypes/diff`
- `mimetypes/tex`

2.2.4 From SharpDevelop

Following icons originates from SharpDevelop a mix of (partially) derivative works of Yusuke Kamiyamane, modified by the SharpDevelop team and independent works by the SharpDevelop team all licensed under GNU LGPL.

- `jump`
- `method`
- `reference`
- `type`
- `field`
- `macro`
- `members`
- `struct`
- `union`
- `class`
- `enum`

2.2.5 From Tango Project

Following icons originates from Tango by the Tango desktop project, released into public domain.

- `search`
- *Glossary*

- genindex
- modindex
- search
- *Icon Credits*

Index

A

absolute_path() (dxr.indexers.FileToSkim method), 27
analyzer, **35**
annotations_by_line() (dxr.indexers.FileToSkim method), 27

C

catalog index, **35**
char_offset() (dxr.indexers.FileToSkim method), 27
contains_text() (dxr.indexers.FileToSkim method), 27

D

DATA_DIR, 12
DXR_CONFIG, 14

E

environment variable
    DATA_DIR, 12
    DXR_CONFIG, 14
    ES_HEAP_SIZE, 12
    LOG_DIR, 12
environment() (dxr.indexers.TreeToIndex method), 25
es() (dxr.lines.Ref method), 28
ES_HEAP_SIZE, 12
es_to_triple() (dxr.lines.Ref static method), 29
es_to_triple() (dxr.lines.Region class method), 29

F

file_to_index() (dxr.indexers.TreeToIndex method), 25
FileToIndex (class in dxr.indexers), 25
FileToSkim (class in dxr.indexers), 27
Filter (class in dxr.filters), 29
filter() (dxr.filters.Filter method), 30
filtered term, **35**
FolderToIndex (class in dxr.indexers), 24
format version, **35**
from_namespace() (dxr.plugins.Plugin class method), 24

H

highlight_content() (dxr.filters.Filter method), 30
highlight_path() (dxr.filters.Filter method), 30

I

index, **35**
is_interesting() (dxr.indexers.FileToSkim method), 27
is_link() (dxr.indexers.FileToSkim method), 28

L

links() (dxr.indexers.FileToSkim method), 28
LOG_DIR, 12

M

mapping, **35**
menu_items() (dxr.lines.Ref method), 29

N

needle, **35**
needles() (dxr.indexers.FileToIndex method), 26
needles_by_line() (dxr.indexers.FileToIndex method), 26

O

opener() (dxr.lines.Ref method), 29

P

Plugin (class in dxr.plugins), 23
post_build() (dxr.indexers.TreeToIndex method), 25
pre_build() (dxr.indexers.TreeToIndex method), 25

R

Ref (class in dxr.lines), 28
refs() (dxr.indexers.FileToSkim method), 28
Region (class in dxr.lines), 29
regions() (dxr.indexers.FileToSkim method), 28

T

term, **35**
text term, **35**
TreeToIndex (class in dxr.indexers), 24