Frontera Documentation

Release 0.8.0

ScrapingHub

Jul 27, 2018

Contents

Frontera is a web crawling tool box, allowing to build crawlers of any scale and purpose. It includes:

- *crawl frontier* framework managing *when* and *what* to crawl and checking for crawling goal* accomplishment,
- workers, Scrapy wrappers, and data bus components to scale and distribute the crawler.

Frontera contain components to allow creation of fully-operational web crawler with Scrapy. Even though it was originally designed for Scrapy, it can also be used with any other crawling framework/system.

CHAPTER 1

Introduction

The purpose of this chapter is to introduce you to the concepts behind Frontera so that you can get an idea of how it works and decide if it is suited to your needs.

1.1 Frontera at a glance

Frontera is an implementation of crawl frontier, a web crawler component used for accumulating URLs/links before downloading them from the web. Main features of Frontera are:

- Online processing oriented,
- distributed spiders and backends architecture,
- customizable crawling policy,
- easy integration with Scrapy,
- relational databases support (MySQL, PostgreSQL, sqlite, and more) with SQLAlchemy and HBase key-value database out of the box,
- ZeroMQ and Kafka message bus implementations for distributed crawlers,
- precise crawling logic tuning with crawling emulation using fake sitemaps with the *Graph Manager*.
- transparent transport layer concept (*message bus*) and communication protocol,
- pure Python implementation.
- Python 3 support.

1.1.1 Use cases

Here are few cases, external crawl frontier can be suitable for:

- URL ordering/queueing isolation from the spider (e.g. distributed cluster of spiders, need of remote management of ordering/queueing),

- URL (meta)data storage is needed (e.g. to demonstrate it's contents somewhere),
- advanced URL ordering logic is needed, when it's hard to maintain code within spider/fetcher.

One-time crawl, few websites

For such use case probably single process mode would be the most appropriate. Frontera can offer these prioritization models out of the box:

- FIFO,
- LIFO,
- Breadth-first (BFS),
- Depth-first (DFS),
- based on provided score, mapped from 0.0 to 1.0.

If website is big, and it's expensive to crawl the whole website, Frontera can be suitable for pointing the crawler to the most important documents.

Distributed load, few websites

If website needs to be crawled faster than single spider one could use distributed spiders mode. In this mode Frontera is distributing spider processes and using one instance of backend worker. Requests are distributed using *message bus* of your choice and distribution logic can be adjusted using custom partitioning. By default requests are distributed to spiders randomly, and desired request rate can be set in spiders.

Consider also using proxy services, such as Crawlera.

Revisiting

There is a set of websites and one need to re-crawl them on timely (or other) manner. Frontera provides simple revisiting backend, scheduling already visited documents for next visit using time interval set by option. This backend is using general relational database for persistence and can be used in single process or distributed spiders modes.

Watchdog use case - when one needs to be notified about document changes, also could be addressed with such a backend and minor customization.

Broad crawling

This use case requires full distribution: spiders and backend. In addition to spiders process one should be running *strategy worker* (s) and *db worker* (s), depending on chosen partitioning scheme.

Frontera can be used for broad set of tasks related to large scale web crawling:

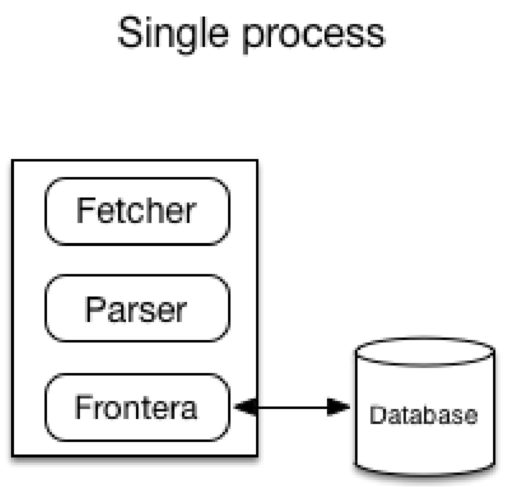
- Broad web crawling, arbitrary number of websites and pages (we tested it on 45M documents volume and 100K websites),
- Host-focused crawls: when you have more than 100 websites,
- Focused crawling:
  - Topical: you search for a pages about some predefined topic,
  - PageRank, HITS or other link graph algorithm guided.

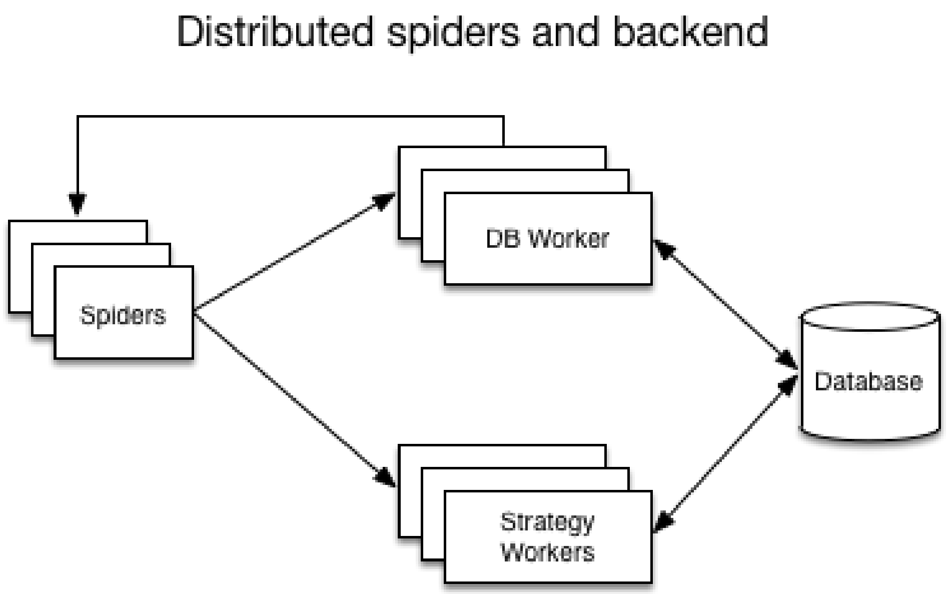
Here are some of the real world problems:

- Building a search engine with content retrieval from the web.
- All kinds of research work on web graph: gathering links, statistics, structure of graph, tracking domain count, etc.
- More general focused crawling tasks: e.g. you search for pages that are big hubs, and frequently changing in time.

1.2 Run modes

A diagram showing architecture of running modes:





Mode	Components needed
Single process	single process running the crawler
Distributed	spiders, *strategy worker* (s) and db worker(s).

1.2.1 Single process

Frontera is instantiated in the same process as fetcher (for example in Scrapy). Read more on how to use that mode *here*.

This mode is suitable for developing the crawling strategy locally and applications where its critical to fetch small number of documents fast.

1.2.2 Distributed

Spiders and backend are distributed. Backend is divided on two parts: *strategy worker* and *db worker*. Strategy worker instances are assigned to their own part of *spider log*.

1. Use `BACKEND` in spider processes set to `MessageBusBackend`
2. In DB and SW workers `BACKEND` should point to `DistributedBackend` subclasses. And selected backend have to be configured.
3. Every spider process should have it's own `SPIDER_PARTITION_ID`, starting from 0 to `SPIDER_FEED_PARTITIONS`. Last must be accessible also to all DB worker instances.
4. Every SW worker process should have it's own `SCORING_PARTITION_ID`, starting from 0 to `SPIDER_LOG_PARTITIONS`. Last must be accessible to all SW worker instances.
5. Both spiders and workers should have it's `MESSAGE_BUS` setting set to the message bus class of your choice and selected message bus have to be configured.

Only Kafka message bus can be used in this mode out of the box.

This mode is designed for crawling of web-scale large amount of domains and pages.

1.3 Quick start single process

The idea is that you develop and debug crawling strategy in single process mode locally and use distributed one when deploying crawling strategy for crawling in production at scale. Single process is also good as a first step to get something running quickly.

> Note, that this tutorial doesn't work for `frontera.contrib.backends.memory.MemoryDistributedBackend`.

1.3.1 1. Create your Scrapy spider

Create your Scrapy project as you usually do. Enter a directory where you'd like to store your code and then run:

```
scrapy startproject tutorial
```

This will create a tutorial directory with the following contents:

```
tutorial/
    scrapy.cfg
    tutorial/
        __init__.py
        items.py
        pipelines.py
        settings.py
        spiders/
            __init__.py
            ...
```

These are basically:

- **scrapy.cfg**: the project configuration file
- **tutorial/**: the project's python module, you'll later import your code from here.
- **tutorial/items.py**: the project's items file.
- **tutorial/pipelines.py**: the project's pipelines file.
- **tutorial/settings.py**: the project's settings file.
- **tutorial/spiders/**: a directory where you'll later put your spiders.

1.3.2 2. Install Frontera

See *Installation Guide*.

1.3.3 3. Integrate your spider with the Frontera

This article about *integration with Scrapy* explains this step in detail.

1.3.4 4. Choose your crawling strategy

Here are the options you would need to redefine when running in single process mode the crawler configured for distributed mode:

```
# these two parameters are pointing Frontera that it will run locally

SPIDER_FEED_PARTITIONS = 1
SPIDER_LOG_PARTITIONS = 1

STRATEGY = "frontera.strategy.basic.BasicCrawlingStrategy"
```

1.3.5 5. Choose your backend

Configure frontier settings to use a built-in backend like:

```
BACKEND = 'frontera.contrib.backends.sqlalchemy.Distributed'
```

1.3.6 6. Inject the seed URLs

This step is required only if your crawling strategy requires seeds injection from external source.:

```
$ python -m frontera.utils.add_seeds --config [your_frontera_config] --seeds-file␣
↪[path to your seeds file]
```

After script is finished succesfully your seeds should be stored in backend's queue and scheduled for crawling.

1.3.7 7. Run the spider

Run your Scrapy spider as usual from the command line:

```
scrapy crawl myspider
```

And that's it! You got your crawler running integrated with Frontera.

1.3.8 What else?

You've seen a simple example of how to use Frontera with Scrapy, but this is just the surface. Frontera provides many powerful features for making frontier management easy and efficient, such as:

- Built-in support for *database storage* for crawled pages.
- Easy *built-in integration with Scrapy* and *any other crawler* through its API.
- *Two distributed crawling modes* with use of ZeroMQ or Kafka and distributed backends.
- Creating different crawling logic/policies *defining your own backend*.
- Plugging your own request/response altering logic using *middlewares*.
- Create fake sitemaps and reproduce crawling without crawler with the *Graph Manager*.
- *Record your Scrapy crawls* and use it later for frontier testing.
- Logging facility that you can hook on to for catching errors and debug your frontiers.

1.4 Quick start distributed mode

Here is a guide how to quickly setup Frontera for single-machine, multiple process, local hacking. We're going to deploy the simpliest possible setup with SQLite and ZeroMQ. Please proceed to *Cluster setup guide* article for a production setup details for broad crawlers.

1.4.1 Prerequisites

Here is what services needs to be installed and configured before running Frontera:

- Python 2.7+ or 3.4+
- Scrapy

Frontera installation

For Ubuntu, type in command line:

```
$ pip install frontera[distributed,zeromq,sql]
```

1.4.2 Get a spider example code

First checkout a GitHub Frontera repository:

```
$ git clone https://github.com/scrapinghub/frontera.git
```

There is a general spider example in `examples/general-spider` folder.

This is a general spider, it does almost nothing except extracting links from downloaded content. It also contains some settings files, please consult *settings reference* to get more information.

1.4.3 Start cluster

First, let's start ZeroMQ broker.

```
$ python -m frontera.contrib.messagebus.zeromq.broker
```

You should see a log output of broker with statistics on messages transmitted.

All further commands have to be made from `general-spider` root directory.

Second, let's start DB worker.

```
$ python -m frontera.worker.db --config frontier.workersettings
```

You should notice that DB is writing messages to the output. It's ok if nothing is written in ZeroMQ sockets, because of absence of seed URLs in the system.

There are Spanish (.es zone) internet URLs from DMOZ directory in general spider repository, let's use them as seeds to bootstrap crawling. Starting the spiders:

```
$ scrapy crawl general -L INFO -s FRONTERA_SETTINGS=frontier.spider_settings -s SEEDS_
↪SOURCE=seeds_es_smp.txt -s SPIDER_PARTITION_ID=0
$ scrapy crawl general -L INFO -s FRONTERA_SETTINGS=frontier.spider_settings -s␣
↪SPIDER_PARTITION_ID=1
```

You should end up with 2 spider processes running. Each should read it's own Frontera config, and first one is using `SEEDS_SOURCE` option to read seeds to bootstrap Frontera cluster.

After some time seeds will pass the streams and will be scheduled for downloading by workers. At this moment crawler is bootstrapped. Now you can periodically check DB worker output or `metadata` table contents to see that there is actual activity.

1.5 Cluster setup guide

This guide is targeting an initial setup of crawling cluster, probably further tuning will be needed. This guide implies you use Kafka message bus for cluster setup (recommended), although it is also possible to use ZeroMQ, which is less reliable option.

1.5.1 Things to decide

- The speed you want to crawl with,
- number of spider processes (assuming that single spider process gives a maximum of 1200 pages/min),
- number of DB and Strategy worker processes.

1.5.2 Things to setup before you start

- Kafka,
- HBase (we recommend 1.0.x and higher),
- *DNS Service* (recommended but not required).

1.5.3 Things to implement before you start

- *Crawling strategy* or *pick one from Frontera package*
- Spider code

1.5.4 Configuring Kafka

Create all topics needed for Kafka message bus

- *spider log* (*frontier-done* (see *SPIDER_LOG_TOPIC*)), set the number of partitions equal to number of strategy worker instances,
- *spider feed* (*frontier-todo* (see *SPIDER_FEED_TOPIC*)), set the number of partitions equal to number of spider instances,
- *scoring log* (*frontier-score* (see *SCORING_LOG_TOPIC*))

1.5.5 Configuring HBase

- create a namespace `crawler` (see *HBASE_NAMESPACE*),
- make sure Snappy compression is supported natively.

1.5.6 Configuring Frontera

Every Frontera component requires it's own configuration module, but some options are shared, so we recommend to create a common modules and import settings from it in component's modules.

1. Create a common module and add there:

```
from __future__ import absolute_import
from frontera.settings.default_settings import MIDDLEWARES
MAX_NEXT_REQUESTS = 512
SPIDER_FEED_PARTITIONS = 2 # number of spider processes
SPIDER_LOG_PARTITIONS = 2 # worker instances
MIDDLEWARES.extend([
    'frontera.contrib.middlewares.domain.DomainMiddleware',
```

(continues on next page)

(continued from previous page)

```
    'frontera.contrib.middlewares.fingerprint.DomainFingerprintMiddleware'
])

QUEUE_HOSTNAME_PARTITIONING = True
KAFKA_LOCATION = 'localhost:9092' # your Kafka broker host:port
SCORING_TOPIC = 'frontier-scoring'
URL_FINGERPRINT_FUNCTION='frontera.utils.fingerprint.hostname_local_fingerprint'
```

2. Create workers shared module:

```
from __future__ import absolute_import
from .common import *

BACKEND = 'frontera.contrib.backends.hbase.HBaseBackend'

MAX_NEXT_REQUESTS = 2048
NEW_BATCH_DELAY = 3.0

HBASE_THRIFT_HOST = 'localhost' # HBase Thrift server host and port
HBASE_THRIFT_PORT = 9090
```

3. Create DB worker module:

```
from __future__ import absolute_import
from .worker import *

LOGGING_CONFIG='logging-db.conf' # if needed
```

4. Create Strategy worker's module:

```
from __future__ import absolute_import
from .worker import *

CRAWLING_STRATEGY = '' # path to the crawling strategy class
LOGGING_CONFIG='logging-sw.conf' # if needed
```

The logging can be configured according to https://docs.python.org/2/library/logging.config.html see the *list of loggers*.

5. Configure spiders module:

```
from __future__ import absolute_import
from .common import *

BACKEND = 'frontera.contrib.backends.remote.messagebus.MessageBusBackend'
KAFKA_GET_TIMEOUT = 0.5
```

6. Configure Scrapy settings module. It's located in Scrapy project folder and referenced in scrapy.cfg. Let's add there:

```
FRONTERA_SETTINGS = ''  # module path to your Frontera spider config module

SCHEDULER = 'frontera.contrib.scrapy.schedulers.frontier.FronteraScheduler'

SPIDER_MIDDLEWARES = {
    'frontera.contrib.scrapy.middlewares.schedulers.SchedulerSpiderMiddleware': 999,
    'frontera.contrib.scrapy.middlewares.seeds.file.FileSeedLoader': 1,
```

(continues on next page)

(continued from previous page)

```
}
DOWNLOADER_MIDDLEWARES = {
    'frontera.contrib.scrapy.middlewares.schedulers.SchedulerDownloaderMiddleware':␣
↪999,
}
```

1.5.7 Starting the cluster

First, let's start storage worker:

```
# start DB worker only for batch generation
$ python -m frontera.worker.db --config [db worker config module] --no-incoming
...
# Then start next one dedicated to spider log processing
$ python -m frontera.worker.db --no-batches --config [db worker config module]
```

Next, let's start strategy workers, one process per spider log partition:

```
$ python -m frontera.worker.strategy --config [strategy worker config] --partition-id␣
↪0
$ python -m frontera.worker.strategy --config [strategy worker config] --partition-id␣
↪1
...
$ python -m frontera.worker.strategy --config [strategy worker config] --partition-id␣
↪N
```

You should notice that all processes are writing messages to the log. It's ok if nothing is written in streams, because of absence of seed URLs in the system.

Let's put our seeds in text file, one URL per line and start spiders. A single spider per spider feed partition:

```
$ scrapy crawl [spider] -L INFO -s SEEDS_SOURCE = 'seeds.txt' -s SPIDER_PARTITION_ID=0
...
$ scrapy crawl [spider] -L INFO -s SPIDER_PARTITION_ID=1
$ scrapy crawl [spider] -L INFO -s SPIDER_PARTITION_ID=2
...
$ scrapy crawl [spider] -L INFO -s SPIDER_PARTITION_ID=N
```

You should end up with N spider processes running. Usually it's enough for a single instance to read seeds from `SEEDS_SOURCE` variable to pass seeds to Frontera cluster. Seeds are only read if spider queue is empty. :*`SPIDER_PARTITION_ID`* can be read from config file also.

After some time seeds will pass the streams and will be scheduled for downloading by workers. Crawler is bootstrapped.

Frontera at a glance Understand what Frontera is and how it can help you.

Run modes High level architecture and Frontera run modes.

Quick start single process using Scrapy as a container for running Frontera.

Quick start distributed mode with SQLite and ZeroMQ.

Cluster setup guide Setting up clustered version of Frontera on multiple machines with HBase and Kafka.

CHAPTER 2

Using Frontera

2.1 Installation Guide

The installation steps assume that you have the following requirements installed:

- Python 2.7+ or 3.4+
- pip and setuptools Python packages. Nowadays pip requires and installs setuptools if not installed.

You can install Frontera using pip.

To install using pip:

```
pip install frontera[option1,option2,...optionN]
```

2.1.1 Options

Each option installs dependencies needed for particular functionality.

- *sql* - relational database,
- *graphs* - Graph Manager,
- *logging* - color logging,
- *tldextract* - can be used with `TLDEXTRACT_DOMAIN_INFO`
- *hbase* - HBase distributed backend,
- *zeromq* - ZeroMQ message bus,
- *kafka* - Kafka message bus,
- *distributed* - workers dependencies.
- *s3* - dependencies required for seeds addition from S3 share,
- *redis* - RedisBackend dependencies,

- *strategies* - built-in crawling strategy dependencies.

2.2 Crawling strategies

2.2.1 Basic

Location: `frontera.strategy.basic.BasicCrawlingStrategy`

Designed to showcase the minimum amount of code needed to implement working *crawling strategy*. It reads the seed URLs, schedules all of them and crawls indefinitely all links that is discovered during the crawl.

Used for testing purposes too.

2.2.2 Breadth-first

Location: `frontera.strategy.depth.BreadthFirstCrawlingStrategy`

Starts with seed URLs provided and prioritizes links depending on their distance from seed page. The bigger the distance, the lower the priority. This will cause close pages to be crawled first.

2.2.3 Depth-first

Location: `frontera.strategy.depth.DepthFirstCrawlingStrategy`

The same as breadth-first, but prioritization is opposite: the bigger the distance the higher the priority. Thus, crawling deeper links first.

2.2.4 Discovery

Location: `frontera.strategy.discovery.Discovery`

This crawling strategy is used for crawling and discovery of websites in the Web. It respects robots.txt rules, follows sitemap.xml and has a limit on a number of pages to crawl from every website. It will also skip the website in case of fatal errors like connection reset or dns resolution errors. There are two settings used to configure it

- `DISCOVERY_MAX_PAGES`,
- `USER_AGENT`

2.3 Frontier objects

Frontier uses 2 object types: `Request` and `Response`. They are used to represent crawling HTTP requests and responses respectively.

These classes are used by most Frontier API methods either as a parameter or as a return value depending on the method used.

Frontier also uses these objects to internally communicate between different components (middlewares and backend).

2.3.1 Request objects

class `frontera.core.models.`**`Request`**(*url*, *method='GET'*, *headers=None*, *cookies=None*, *meta=None*, *body="*)

A *Request* object represents an HTTP request, which is generated for seeds, extracted page links and next pages to crawl. Each one should be associated to a *Response* object when crawled.

Parameters

- **url** (*string*) – URL to send.
- **method** (*string*) – HTTP method to use.
- **headers** (*dict*) – dictionary of headers to send.
- **cookies** (*dict*) – dictionary of cookies to attach to this request.
- **meta** (*dict*) – dictionary that contains arbitrary metadata for this request, the keys must be bytes and the values must be either bytes or serializable objects such as lists, tuples, dictionaries with byte type items.

body

A string representing the request body.

cookies

Dictionary of cookies to attach to this request.

headers

A dictionary which contains the request headers.

meta

A dict that contains arbitrary metadata for this request. This dict is empty for new Requests, and is usually populated by different Frontera components (middlewares, etc). So the data contained in this dict depends on the components you have enabled. The keys are bytes and the values are either bytes or serializable objects such as lists, tuples, dictionaries with byte type items.

method

A string representing the HTTP method in the request. This is guaranteed to be uppercase. Example: `GET`, `POST`, `PUT`, etc

url

A string containing the URL of this request.

2.3.2 Response objects

class `frontera.core.models.`**`Response`**(*url*, *status_code=200*, *headers=None*, *body="*, *request=None*)

A *Response* object represents an HTTP response, which is usually downloaded (by the crawler) and sent back to the frontier for processing.

Parameters

- **url** (*string*) – URL of this response.
- **status_code** (*int*) – the HTTP status of the response. Defaults to 200.
- **headers** (*dict*) – dictionary of headers to send.
- **body** (*str*) – the response body.
- **request** (Request) – The Request object that generated this response.

body
A str containing the body of this Response.

headers
A dictionary object which contains the response headers.

meta
A shortcut to the *Request.meta* attribute of the *Response.request* object (ie. self.request.meta).

request
The *Request* object that generated this response.

status_code
An integer representing the HTTP status of the response. Example: 200, 404, 500.

url
A string containing the URL of the response.

Fields domain and fingerprint are added by *built-in middlewares*

2.3.3 Identifying unique objects

As frontier objects are shared between the crawler and the frontier, some mechanism to uniquely identify objects is needed. This method may vary depending on the frontier logic (in most cases due to the backend used).

By default, Frontera activates the *fingerprint middleware* to generate a unique fingerprint calculated from the *Request.url* and *Response.url* fields, which is added to the *Request.meta* and *Response.meta* fields respectively. You can use this middleware or implement your own method to manage frontier objects identification.

An example of a generated fingerprint for a *Request* object:

```
>>> request.url
'http://thehackernews.com'

>>> request.meta['fingerprint']
'198d99a8b2284701d6c147174cd69a37a7dea90f'
```

2.3.4 Adding additional data to objects

In most cases frontier objects can be used to represent the information needed to manage the frontier logic/policy.

Also, additional data can be stored by components using the *Request.meta* and *Response.meta* fields.

For instance the frontier *domain middleware* adds a domain info field for every *Request.meta* and *Response.meta* if is activated:

```
>>> request.url
'http://www.scrapinghub.com'

>>> request.meta['domain']
{
    "name": "scrapinghub.com",
    "netloc": "www.scrapinghub.com",
    "scheme": "http",
    "sld": "scrapinghub",
    "subdomain": "www",
    "tld": "com"
}
```

2.4 Middlewares

Frontier *Middleware* sits between `FrontierManager` and *Backend* objects, using hooks for *Request* and *Response* processing according to *frontier data flow*.

It's a light, low-level system for filtering and altering Frontier's requests and responses.

2.4.1 Activating a middleware

To activate a *Middleware* component, add it to the *MIDDLEWARES* setting, which is a list whose values can be class paths or instances of *Middleware* objects.

Here's an example:

```
MIDDLEWARES = [
    'frontera.contrib.middlewares.domain.DomainMiddleware',
]
```

Middlewares are called in the same order they've been defined in the list, to decide which order to assign to your middleware pick a value according to where you want to insert it. The order does matter because each middleware performs a different action and your middleware could depend on some previous (or subsequent) middleware being applied.

Finally, keep in mind that some middlewares may need to be enabled through a particular setting. See *each middleware documentation* for more info.

2.4.2 Writing your own middleware

Writing your own frontier middleware is easy. Each *Middleware* component is a single Python class inherited from *Component*.

`FrontierManager` will communicate with all active middlewares through the methods described below.

class `frontera.core.components.`**`Middleware`**

Interface definition for a Frontier Middlewares

Methods

`frontier_start``()`

Called when the frontier starts, see *starting/stopping the frontier*.

`frontier_stop``()`

Called when the frontier stops, see *starting/stopping the frontier*.

`page_crawled``(`*response*`)`

This method is called every time a page has been crawled.

Parameters **`response`** (`object`) – The *Response* object for the crawled page.

Returns *Response* or `None`

Should either return `None` or a *Response* object.

If it returns `None`, `FrontierManager` won't continue processing any other middleware and *Backend* will never be notified.

If it returns a *Response* object, this will be passed to next middleware. This process will repeat for all active middlewares until result is finally passed to the *Backend*.

If you want to filter a page, just return None.

request_error(*page*, *error*)

This method is called each time an error occurs when crawling a page.

Parameters

- **request** (*object*) – The crawled with error *Request* object.
- **error** (*string*) – A string identifier for the error.

Returns *Request* or None

Should either return None or a *Request* object.

If it returns None, FrontierManager won't continue processing any other middleware and *Backend* will never be notified.

If it returns a *Response* object, this will be passed to next middleware. This process will repeat for all active middlewares until result is finally passed to the *Backend*.

If you want to filter a page error, just return None.

Class Methods

classmethod from_manager(*manager*)

Class method called from FrontierManager passing the manager itself.

Example of usage:

```
def from_manager(cls, manager):
    return cls(settings=manager.settings)
```

2.4.3 Built-in middleware reference

This page describes all *Middleware* components that come with Frontera. For information on how to use them and how to write your own middleware, see the *middleware usage guide.*.

For a list of the components enabled by default (and their orders) see the *MIDDLEWARES* setting.

DomainMiddleware

class frontera.contrib.middlewares.domain.**DomainMiddleware**

This *Middleware* will add a domain info field for every *Request.meta* and *Response.meta* if is activated.

domain object will contain the following fields, with both keys and values as bytes:

- **netloc**: URL netloc according to RFC 1808 syntax specifications
- **name**: Domain name
- **scheme**: URL scheme
- **tld**: Top level domain
- **sld**: Second level domain
- **subdomain**: URL subdomain(s)

An example for a *Request* object:

```
>>> request.url
'http://www.scrapinghub.com:8080/this/is/an/url'

>>> request.meta['domain']
{
    "name": "scrapinghub.com",
    "netloc": "www.scrapinghub.com",
    "scheme": "http",
    "sld": "scrapinghub",
    "subdomain": "www",
    "tld": "com"
}
```

If *TEST_MODE* is active, It will accept testing URLs, parsing letter domains:

```
>>> request.url
'A1'

>>> request.meta['domain']
{
    "name": "A",
    "netloc": "A",
    "scheme": "-",
    "sld": "-",
    "subdomain": "-",
    "tld": "-"
}
```

UrlFingerprintMiddleware

class frontera.contrib.middlewares.fingerprint.**UrlFingerprintMiddleware**

This *Middleware* will add a `fingerprint` field for every *Request.meta* and *Response.meta* if is activated.

Fingerprint will be calculated from object `URL`, using the function defined in *URL_FINGERPRINT_FUNCTION* setting. You can write your own fingerprint calculation function and use by changing this setting. The fingerprint must be bytes.

An example for a *Request* object:

```
>>> request.url
'http//www.scrapinghub.com:8080'

>>> request.meta['fingerprint']
'60d846bc2969e9706829d5f1690f11dafb70ed18'
```

frontera.utils.fingerprint.**hostname_local_fingerprint**(*key*)

This function is used for URL fingerprinting, which serves to uniquely identify the document in storage. `hostname_local_fingerprint` is constructing fingerprint getting first 4 bytes as Crc32 from host, and rest is MD5 from rest of the URL. Default option is set to make use of HBase block cache. It is expected to fit all the documents of average website within one cache block, which can be efficiently read from disk once.

Parameters **key** – str URL

Returns str 20 bytes hex string

DomainFingerprintMiddleware

class frontera.contrib.middlewares.fingerprint.**DomainFingerprintMiddleware**

This *Middleware* will add a `fingerprint` field for every *Request.meta* and *Response.meta* `domain` fields if is activated.

Fingerprint will be calculated from object `URL`, using the function defined in *DOMAIN_FINGERPRINT_FUNCTION* setting. You can write your own fingerprint calculation function and use by changing this setting. The fingerprint must be bytes

An example for a *Request* object:

```
>>> request.url
'http//www.scrapinghub.com:8080'

>>> request.meta['domain']
{
    "fingerprint": "5bab61eb53176449e25c2c82f172b82cb13ffb9d",
    "name": "scrapinghub.com",
    "netloc": "www.scrapinghub.com",
    "scheme": "http",
    "sld": "scrapinghub",
    "subdomain": "www",
    "tld": "com"
}
```

2.5 Canonical URL Solver

Is a special *middleware* object responsible for identifying canonical URL address of the document and modifying request or response metadata accordingly. Canonical URL solver always executes last in the middleware chain, before calling Backend methods.

The main purpose of this component is preventing metadata records duplication and confusing crawler behavior connected with it. The causes of this are: - Different redirect chains could lead to the same document. - The same document can be accessible by more than one different URL.

Well designed system has it's own, stable algorithm of choosing the right URL for each document. Also see Canonical link element.

Canonical URL solver is instantiated during Frontera Manager initialization using class from *CANONICAL_SOLVER* setting.

2.5.1 Built-in canonical URL solvers reference

Basic

Used as default.

class frontera.contrib.canonicalsolvers.basic.**BasicCanonicalSolver**

Implements a simple CanonicalSolver taking always first URL from redirect chain, if there were redirects. It allows easily to avoid leaking of requests in Frontera (e.g. when request issued by `get_next_requests()` never matched in `page_crawled()`) at the price of duplicating records in Frontera for pages having more than one URL or complex redirects chains.

2.6 Backends

A `DistributedBackend` is used to separate higher level code of *crawling strategy* from low level storage API. `Queue`, `Metadata`, `States` and

> `DomainMetadata` are inner components of the DistributedBackend.

The latter is meant to instantiate and hold the references to the objects of above mentioned classes. Frontera is bundled with database and in-memory implementations of Queue, Metadata, States and DomainMetadata which can be combined in your custom backends or used standalone by directly instantiating specific variant of `FrontierManager`.

DistributedBackend methods are called by the FrontierManager after `Middleware`, using hooks for `Request` and `Response` processing according to *frontier data flow*.

Unlike Middleware, that can have many different instances activated, only one DistributedBackend can be used per frontier.

2.6.1 Activating a backend

To activate the specific backend, set it through the `BACKEND` setting.

Here's an example:

```
BACKEND = 'frontera.contrib.backends.memory.MemoryDistributedBackend'
```

Keep in mind that some backends may need to be additionally configured through a particular setting. See *backends documentation* for more info.

2.6.2 Writing your own backend

Each backend component is a single Python class inherited from `DistributedBackend` and using one or all of `Queue`, `Metadata`, `States` and `DomainMetadata`.

`FrontierManager` will communicate with active backend through the methods described below.

class `frontera.core.components.`**`Backend`**

Interface definition for frontier backend.

Methods

`frontier_start``()`

Called when the frontier starts, see *starting/stopping the frontier*.

Returns None.

`frontier_stop``()`

Called when the frontier stops, see *starting/stopping the frontier*.

Returns None.

`finished``()`

Quick check if crawling is finished. Called pretty often, please make sure calls are lightweight.

Returns boolean

`page_crawled``(`*response*`)`

This method is called every time a page has been crawled.

Parameters **`response`** (`object`) – The `Response` object for the crawled page.

Returns None.

request_error(*page*, *error*)
This method is called each time an error occurs when crawling a page.

Parameters

- **request** (*object*) – The crawled with error `Request` object.
- **error** (*string*) – A string identifier for the error.

Returns None.

get_next_requests(*max_n_requests*, ***kwargs*)
Returns a list of next requests to be crawled.

Parameters

- **max_next_requests** (*int*) – Maximum number of requests to be returned by this method.
- **kwargs** (*dict*) – A parameters from downloader component.

Returns list of `Request` objects.

Class Methods

classmethod from_manager(*manager*)
Class method called from `FrontierManager` passing the manager itself.

Example of usage:

```
def from_manager(cls, manager):
    return cls(settings=manager.settings)
```

Properties

queue

Returns associated `Queue` object

states

Returns associated `States` object

metadata

Returns associated `Metadata` object

class frontera.core.components.**DistributedBackend**
Interface definition for distributed frontier backend. Implies using in strategy worker and DB worker.

Inherits all methods of Backend, and has two more class methods, which are called during strategy and db worker instantiation.

classmethod DistributedBackend.**strategy_worker**(*manager*)

classmethod DistributedBackend.**db_worker**(*manager*)

Backend should communicate with low-level storage by means of these classes:

Metadata

Is used to store the contents of the crawl.

class `frontera.core.components.`**`Metadata`**

Interface definition for a frontier metadata class. This class is responsible for storing documents metadata, including content and optimized for write-only data flow.

Methods

`request_error`(*page*, *error*)

This method is called each time an error occurs when crawling a page.

Parameters

- **`request`** (*`object`*) – The crawled with error *`Request`* object.
- **`error`** (*`string`*) – A string identifier for the error.

`page_crawled`(*response*)

This method is called every time a page has been crawled.

Parameters **`response`** (*`object`*) – The *`Response`* object for the crawled page.

Known implementations are: `MemoryMetadata` and `sqlalchemy.components.Metadata`.

Queue

Is a priority queue and used to persist requests scheduled for crawling.

class `frontera.core.components.`**`Queue`**

Interface definition for a frontier queue class. The queue has priorities and partitions.

Methods

`get_next_requests`(*max_n_requests*, *partition_id*, ***kwargs*)

Returns a list of next requests to be crawled, and excludes them from internal storage.

Parameters

- **`max_next_requests`** (*`int`*) – Maximum number of requests to be returned by this method.
- **`kwargs`** (*`dict`*) – A parameters from downloader component.

Returns list of *`Request`* objects.

`schedule`(*batch*)

Schedules a new documents for download from batch, and updates score in metadata.

Parameters **`batch`** – list of tuples(fingerprint, score, request, schedule), if `schedule` is True, then document needs to be scheduled for download, False - only update score in metadata.

`count`()

Returns count of documents in the queue.

Returns int

Known implementations are: `MemoryQueue` and `sqlalchemy.components.Queue`.

States

Is a storage used for checking and storing the link states. Where state is a short integer of one of states descibed in *`frontera.core.components.States`*.

class `frontera.core.components.`**`States`**

Interface definition for a link states management class. This class is responsible for providing actual link state, and persist the state changes in batch-oriented manner.

Methods

`update_cache`(*objs*)

Reads states from meta['state'] field of request in objs and stores states in internal cache.

Parameters **`objs`** – list or tuple of *`Request`* objects.

`set_states`(*objs*)

Sets meta['state'] field from cache for every request in objs.

Parameters **`objs`** – list or tuple of *`Request`* objects.

`flush`()

Flushes internal cache to storage.

`fetch`(*fingerprints*)

Get states from the persistent storage to internal cache.

Parameters **`fingerprints`** – list document fingerprints, which state to read

Known implementations are: `MemoryStates` and `sqlalchemy.components.States`.

DomainMetadata

Is used to store per-domain flags, counters or even robots.txt contents to help *crawling strategy* maintain features like per-domain number of crawled pages limit or automatic banning.

class `frontera.core.components.`**`DomainMetadata`**

Interface definition for a domain metadata storage. It's main purpose is to store the per-domain metadata using Python-friendly structures. Meant to be used by crawling strategy to store counters and flags in low level facilities provided by Backend.

Methods

`__setitem__`(*key*, *value*)

Puts key, value tuple in storage.

Parameters

- **`key`** – str
- **`value`** – Any

`__getitem__`(*key*)

Retrieves the value associated with the storage. Raises KeyError if key is absent.

Parameters **`key`** – str

Return value Any

`__delitem__`(*key*)

Removes the tuple associated with key from storage. Raises KeyError if key is absent.

Parameters **`key`** – str

`__contains__`(*key*)

Checks if key is present in the storage.

Parameters **`key`** – str

Returns boolean

Known implementations are: native dict and `sqlalchemy.components.DomainMetadata`.

2.6.3 Built-in backend reference

This article describes all backend components that come bundled with Frontera.

Memory backend

This implementation is using heapq module to store the requests queue and native dicts for other purposes and is meant to be used for educational or testing purposes only.

class `frontera.contrib.backends.memory.`**`MemoryDistributedBackend`**(*manager*)

SQLAlchemy backends

This implementations is using RDBMS storage with SQLAlchemy library.

By default it uses an in-memory SQLite database as a storage engine, but any databases supported by SQLAlchemy can be used.

If you need to use your own declarative sqlalchemy models, you can do it by using the *`SQLALCHEMYBACKEND_MODELS`* setting.

For a complete list of all settings used for SQLAlchemy backends check the *settings* section.

HBase backend

Is more suitable for large scale web crawlers. Settings reference can be found here *HBase backend*. Consider tuning a block cache to fit states within one block for average size website. To achieve this it's recommended to use *`hostname_local_fingerprint`* to achieve documents closeness within the same host. This function can be selected with *`URL_FINGERPRINT_FUNCTION`* setting.

Redis backend

This is similar to the HBase backend. It is suitable for large scale crawlers that still has a limited scope. It is recommended to ensure Redis is allowed to use enough memory to store all data the crawler needs. In case of Redis running out of memory, the crawler will log this and continue. When the crawler is unable to write metadata or queue items to the database; that metadata or queue items are lost.

In case of connection errors; the crawler will attempt to reconnect three times. If the third attempt at connecting to Redis fails, the worker will skip that Redis operation and continue operating.

2.7 Message bus

Message bus is the transport layer abstraction mechanism. Frontera provides interface and several implementations. Only one message bus can be used in crawler at the time, and it's selected with *`MESSAGE_BUS`* setting.

Spiders process can use

class `frontera.contrib.backends.remote.messagebus.`**`MessageBusBackend`**(*manager*)

to communicate using message bus.

2.7.1 Built-in message bus reference

ZeroMQ

It's the default option, implemented using lightweight ZeroMQ library in

class `frontera.contrib.messagebus.zeromq.`**`MessageBus`**(*settings*)

and can be configured using *ZeroMQ message bus settings*.

ZeroMQ message bus requires installed ZeroMQ library and running broker process, see *Start cluster*.

Overall ZeroMQ message bus is designed to get a working PoC quickly and smaller deployments. Mainly because it's prone to message loss when data flow of components isn't properly adjusted or during startup. Here's the recommended order of components startup to avoid message loss:

1. *db worker*
2. *strategy worker*
3. :term:'spider's

Unfortunately, it's not possible to avoid message loss when stopping running crawler with unfinished crawl. We recommend
to use Kafka message bus if your crawler application is sensitive to small message loss.

WARNING! ZeroMQ message bus doesn't support yet multiple SW and DB workers, only one instance of each worker type is allowed.

Kafka

Can be selected with

and configured using *Kafka message bus settings*.

Requires running Kafka service and more suitable for large-scale web crawling.

2.7.2 Protocol

Depending on stream Frontera is using several message types to code it's messages. Every message is a python native object serialized using msgpack or JSON. The codec module can be selected using *`MESSAGE_BUS_CODEC`*, and it's required to export `Encoder` and `Decoder` classes.

Here are the classes needed to subclass to implement own codec:

class `frontera.core.codec.`**`BaseEncoder`**

`encode_page_crawled`(*response*)
Encodes a page_crawled message

Parameters **`response`** (*`object`*) – A frontier Response object

Returns bytes encoded message

`encode_request_error`(*request, error*)
Encodes a request_error message

Parameters

- **`request`** (*`object`*) – A frontier Request object
- **`error`** (*`string`*) – Error description

Returns bytes encoded message

`encode_request`(*request*)
Encodes requests for spider feed stream.

Parameters **`request`** (*`object`*) – Frontera Request object

Returns bytes encoded message

`encode_update_score`(*request*, *score*, *schedule*)
Encodes update_score messages for scoring log stream.

Parameters

- **`request`** (*`object`*) – Frontera Request object
- **`score`** (*`float`*) – score
- **`schedule`** (*`bool`*) – True if document needs to be scheduled for download

Returns bytes encoded message

`encode_new_job_id`(*job_id*)
Encodes changing of job_id parameter.

Parameters **`job_id`** (*`int`*) –

Returns bytes encoded message

`encode_offset`(*partition_id*, *offset*)
Encodes current spider offset in spider feed.

Parameters

- **`partition_id`** (*`int`*) –
- **`offset`** (*`int`*) –

Returns bytes encoded message

class `frontera.core.codec.`**`BaseDecoder`**

`decode`(*buffer*)
Decodes the message.

Parameters **`buffer`** (*`bytes`*) – encoded message

Returns tuple of message type and related objects

`decode_request`(*buffer*)
Decodes Request objects.

Parameters **`buffer`** (*`bytes`*) – serialized string

Returns object Request

2.7.3 Available codecs

MsgPack

A MsgPack codec for Frontera. Implemented using native msgpack-python library.

Module: frontera.contrib.backends.remote.codecs.msgpack

JSON

A JSON codec for Frontera. Implemented using native json library.

Module: frontera.contrib.backends.remote.codecs.json

2.8 Writing custom crawling strategy

Crawling strategy is an essential part of Frontera-based crawler and it's guiding the crawler by instructing it which pages to crawl, when and with what priority.

2.8.1 Crawler workflow

Frontera-based crawler consist of multiple processes, which are running indefinitely. The state in these processes are persisted to a permanent storage. When processes are stopped the state is flushed and will be loaded next time when access to certain data item is needed. Therefore it's easy to pause the crawl by stopping the processes, do the maintenance or modify the code and start again without restarting the crawl from the beginning.

> IMPORTANT DETAIL Spider log (see http://frontera.readthedocs.io/en/latest/topics/glossary.html) is using hostname-based partitioning. The content generated from particular host will always land to the same partition (and therefore strategy worker instance). That guarantees the crawling strategy you design will be always dealing with same subset of hostnames on every SW instance. It also means the same domain cannot be operated from multiple strategy worker instances. To get the hostname the 2-nd level domain name is used with public suffix resolved.

To restart the crawl the

- queue contents
- link states
- domain metadata

needs to be cleaned up. This is usually done by means of truncation of tables.

2.8.2 Crawling strategy class

It has to be inherited from BaseCrawlingStrategy and implement it's API.

class `frontera.strategy.`**`BaseCrawlingStrategy`**(*manager*, *args*, *scheduled_stream*, *states_context*)

Interface definition for a crawling strategy.

Before calling these methods strategy worker is adding 'state' key to meta field in every `Request` with state of the URL. Pleases refer for the states to HBaseBackend implementation.

After exiting from all of these methods states from meta field are passed back and stored in the backend.

Constructor of the crawling strategy.

Args: manager: is an instance of :class: *Backend <frontera.core.manager.FrontierManager>* instance args: is a dict with command line arguments from *strategy worker* scheduled_stream: is a helper class for sending scheduled requests states_context: a helper to operate with states for requests created in crawling strategy class

Methods

classmethod from_worker(*manager*, *args*, *scheduled_stream*, *states_context*)
Called on instantiation in strategy worker.

see params for constructor :return: new instance

read_seeds(*stream*)
Called when *strategy worker* is run using add-seeds mode.

Parameters **stream** (`file`) – A file-like object containing seed content

page_crawled(*response*)
Called every time document was successfully crawled, and receiving page_crawled event from spider log.

Parameters **response** (`object`) – The `Response` object for the crawled page.

filter_extracted_links(*request*, *links*)
Called every time on receiving links_extracted event by strategy worker. This call is preceding the call to links_extracted handler and is aiming to filter unused links and return only those where states information is needed.

The motivation for having the filtration separated before the actual handler is to save on HBase state retrieval. Every non-cached link is requested from HBase and it may slow down the cluster significantly on discovery-intensive crawls. Please make sure you use this class to filter out all the links you're not going ot use in :method:'links_extracted <frontera.worker.strategies.BaseCrawlingStrategy.links_extracted> handler.

Parameters

- **request** (`object`) – The `Request` object for the crawled page.
- **links** (`list`) – A list of `Request` objects generated from the links extracted for the crawled page.

Returns A subset of `Request` input objects.

links_extracted(*request*, *links*)
Called every time document was successfully crawled, and receiving links_extracted event from spider log, after the link states are fetched from backend. Should be used to schedule links according to some rules.

Parameters

- **request** (`object`) – The `Request` object for the crawled page.
- **links** (`list`) – A list of `Request` objects generated from the links extracted for the crawled page.

finished()
Called by Strategy worker, after finishing processing each cycle of spider log. If this method returns true, then Strategy worker reports that crawling goal is achieved, stops and exits.

Returns bool

close()
Called when strategy worker is about to close crawling strategy.

schedule(*request*, *score=1.0*, *dont_queue=False*)
Schedule document for crawling with specified score.

Parameters

- **request** – A `Request` object.
- **score** – float from 0.0 to 1.0

- **dont_queue** – bool, True - if no need to schedule, only update the score

create_request (*url*, *method='GET'*, *headers=None*, *cookies=None*, *meta=None*, *body="*)
Creates request with specified fields. This method only creates request, but isn't getting it's state from storage. Use self.refresh_states on a batch of requests to get their states from storage.

Parameters

- **url** – str
- **method** – str
- **headers** – dict
- **cookies** – dict
- **meta** – dict
- **body** – str

Returns `Request`

refresh_states (*requests*)
Retrieves states for all requests from storage.

Parameters **requests** – list(`Request`)

The class can be put in any module and passed to *strategy worker* or local Scrapy process using command line option or `CRAWLING_STRATEGY` setting on startup.

The strategy class can use its own storage or any other kind of resources. All items from *spider log* will be passed through these methods. Scores returned doesn't have to be the same as in method arguments. Periodically `finished()` method is called to check if crawling goal is achieved.

Workflow

There essentially two workflows: seeds addition (or injection) and main workflow. When crawl starts from scratch it has to run the seed injection first and then proceed with main workflow. When paused/resumed crawler is running main workflow.

Seeds addition

The purpose of this step is to inject the seeds into the crawler pipeline. The framework allows to process the seeds stream (which is read from file placed locally or in S3), create requests needed, get their link states, and schedule them. Once requests are scheduled they will get to the queue and propagate to spiders.

To enter this workflow user is running strategy worker in add seeds mode providing arguments to crawling strategy from command line. In particular –seeds-url is used with s3 or local file URL containing seeds to inject.

1. from_worker() → init() 1. read_seeds(stream from file, None if file isn't present) 1. exit

It's very convenient to run seeds addition using helper app in Frontera:

```
$ python -m frontera.utils.add_seeds --config ... --seeds-file ...
```

Main

This is the main cycle used when crawl is in progress. In a nutshell on every spider event the specific handler is called, depending on the type of event. When strategy worker is getting the SIGTERM signal it's trying to stop politely by calling close(). In its normal state it listens for a spider log and executes the event handlers.

1. from_worker() → init() 1. page_crawled(response) OR page_error(request, error) OR filter_extracted_links(request, links) and subsequent links_extracted(request, links) 1. close() 1. exit

Scheduling and creating requests

The ultimate goal of crawling strategy is scheduling of requests. To schedule request there is a method schedule(request, score). The request is an instance of `Request` class and is often available from arguments of event handlers: _page_crawled_, _page_error_ and _links_extracted_, or can be created

on-demand using _create_request()_ method.

IMPORTANT NOTICE

The request created with create_request() has no state (meta[b'state']) after creation. To get the states strategy worker needs to access the backend, and this is not happenning when you call create_request(). Instead it is expected you will create a batch of requests and call refresh_states(iterable) on the whole batch of requests. After refresh_states is done, you will have a states available for your newly created requests.

The Request objects created by strategy worker for event handlers are always having the states assigned.

State operations

Every link has a state. The purpose of this states is to allow the developer to persist the state of the link in the system (allow restart of SW components without data loss) and use it for decision making. The states are cached in strategy worker, flushed to backend and will be loaded when needed. States are defined in `frontera.core.components.States` and can have following values:

- NOT_CRAWLED,
- QUEUED,
- CRAWLED,
- ERROR

NOT_CRAWLED is assigned when link is new, and wasn't seen previously, the rest of the state values must be assigned in the crawling strategy code.

States allow to check that link was visited or discovered, and perform analysis of the states database to collect the state statistics using MapReduce style jobs.

2.8.3 Components

There are certain building blocks and successful solutions exist for the common problems.

DomainMetadata

It's often needed to persist per-host metadata in the permanent storage. To solve this there is a `frontera.core.components.DomainMetadata` instance in backend. It's has an interface of Python mapping types (https://docs.python.org/3/library/stdtypes.html?highlight=mapping#mapping-types-dict). It's expected that one will be using domain names as keys and dicts as values. It's convenient to store there per-domin statistics, ban states, the count of links found, etc.

PublicSuffix

When crawling multiple domains (especially unknown ones) it's important to resolve the 2-nd level domain name properly using publicsuffix.

Is a library from publicsuffix module provided by https://publicsuffix.org/. The purpose is to maintain a publicsuffix of ccTLDs and name resolution routines for them in a single library. For us it's convenient to use these library everywhere where domain name resolution is needed. Here are few examples:

- www.london.co.uk → london.co.uk
- images.yandex.ru → yandex.ru
- t.co → t.co

  As you may see the number of dots of reverted domain name cannot be used for domain name resolution.

2.8.4 Useful details

Debugging crawling strategy

The best approach I found is to log all the events and outcomes using Python native logging. I.e. to setup the logger for crawling strategy class and use it. When debug output is needed you will be able to set the logger to output to a file, with a specific format and log level. After you have logging output set up you should start the crawl of problematic website locally, collect and analyse the log output.

Other approaches include analysis of links database, inspecting of domain metadata and states tables, collecting the log output of link states changes (experimental SW feature).

Meta fields

1 b"slot" Queue partitioning key in bytes, highest priority. Use it if your app requires partitioning other than default 2-nd level domain-based partitioning Optional 2 b"domain" Dict generated by Frontera DomainMiddleware, and containing parsed domain name Always 3 b"state" Integer representing the link state, set by strategy worker. Link states are defined in frontera.core.components.States Always 4 b"encoding" In response, for HTML, encoding detected by Scrapy Optional 5 b"scrapy_meta" When scheduling can be used to set meta field for Scrapy Optional

Keys and string types in nested structures are always bytes.

2.9 Using the Frontier with Scrapy

To use Frontera with Scrapy, you will need to add Scrapy middlewares and redefine the default Scrapy scheduler with custom Frontera scheduler. Both can be done by modifying Scrapy settings.

2.9.1 Activating the frontier

The Frontera uses 2 different middlewares: `SchedulerSpiderMiddleware` and `SchedulerDownloaderMiddleware`, and it's own scheduler `FronteraScheduler`.

To activate the Frontera in your Scrapy project, just add them to the SPIDER_MIDDLEWARES, DOWNLOADER_MIDDLEWARES and SCHEDULER settings:

```
SPIDER_MIDDLEWARES.update({
    'frontera.contrib.scrapy.middlewares.schedulers.SchedulerSpiderMiddleware': 1000,
})

DOWNLOADER_MIDDLEWARES.update({
    'frontera.contrib.scrapy.middlewares.schedulers.SchedulerDownloaderMiddleware':␣
↪1000,
})

SCHEDULER = 'frontera.contrib.scrapy.schedulers.frontier.FronteraScheduler'
```

Create a Frontera `settings.py` file and add it to your Scrapy settings:

```
FRONTERA_SETTINGS = 'tutorial.frontera.settings'
```

Another option is to put these settings right into Scrapy settings module.

2.9.2 Organizing files

When using frontier with a Scrapy project, we propose the following directory structure:

```
my_scrapy_project/
    my_scrapy_project/
        frontera/
            __init__.py
            settings.py
        spiders/
            ...
        __init__.py
        settings.py
     scrapy.cfg
```

These are basically:

- `my_scrapy_project/frontera/settings.py`: the Frontera settings file.
- `my_scrapy_project/spiders`: the Scrapy spiders folder
- `my_scrapy_project/settings.py`: the Scrapy settings file
- `scrapy.cfg`: the Scrapy config file

2.9.3 Running the rawl

Just run your Scrapy spider as usual from the command line:

```
scrapy crawl myspider
```

2.9.4 Frontier Scrapy settings

You can configure your frontier two ways:

- Using `FRONTERA_SETTINGS` parameter, which is a module path pointing to Frontera settings in Scrapy settings file. Defaults to `None`
- Define frontier settings right into Scrapy settings file.

Defining frontier settings via Scrapy settings

Frontier settings can also be defined via Scrapy settings. In this case, the order of precedence will be the following:

1. Settings defined in the file pointed by *FRONTERA_SETTINGS* (higher precedence)
2. Settings defined in the Scrapy settings
3. Default frontier settings

2.9.5 Writing Scrapy spider

Spider logic

Creation of basic Scrapy spider is described at Quick start single process page.

It's also a good practice to prevent spider from closing because of insufficiency of queued requests transport::

```
@classmethod
def from_crawler(cls, crawler, *args, **kwargs):
    spider = cls(*args, **kwargs)
    spider._set_crawler(crawler)
    spider.crawler.signals.connect(spider.spider_idle, signal=signals.spider_idle)
    return spider

def spider_idle(self):
    self.log("Spider idle signal caught.")
    raise DontCloseSpider
```

Configuration guidelines

There several tunings you can make for efficient broad crawling.

Various settings suitable for broad crawling:

```
HTTPCACHE_ENABLED = False   # Turns off disk cache, which has low hit ratio during␣
↪broad crawls
REDIRECT_ENABLED = True
COOKIES_ENABLED = False
DOWNLOAD_TIMEOUT = 120
RETRY_ENABLED = False   # Retries can be handled by Frontera itself, depending on␣
↪crawling strategy
DOWNLOAD_MAXSIZE = 10 * 1024 * 1024  # Maximum document size, causes OOM kills if not␣
↪set
LOGSTATS_INTERVAL = 10  # Print stats every 10 secs to console
```

Auto throttling and concurrency settings for polite and responsible crawling::

```
# auto throttling
AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_DEBUG = False
AUTOTHROTTLE_MAX_DELAY = 3.0
AUTOTHROTTLE_START_DELAY = 0.25     # Any small enough value, it will be adjusted␣
↪during operation by averaging
                                    # with response latencies.
RANDOMIZE_DOWNLOAD_DELAY = False
```

(continues on next page)

(continued from previous page)

```
# concurrency
CONCURRENT_REQUESTS = 256          # Depends on many factors, and should be␣
↪determined experimentally
CONCURRENT_REQUESTS_PER_DOMAIN = 10
DOWNLOAD_DELAY = 0.0
```

Check also Scrapy broad crawling recommendations.

2.10 Settings

The Frontera settings allows you to customize the behaviour of all components, including the `FrontierManager`, *Middleware* and *Backend* themselves.

The infrastructure of the settings provides a global namespace of key-value mappings that can be used to pull configuration values from. The settings can be populated through different mechanisms, which are described below.

For a list of available built-in settings see: *Built-in settings reference*.

2.10.1 Designating the settings

When you use Frontera, you have to tell it which settings you're using. As `FrontierManager` is the main entry point to Frontier usage, you can do this by using the method described in the *Loading from settings* section.

When using a string path pointing to a settings file for the frontier we propose the following directory structure:

```
my_project/
    frontier/
        __init__.py
        settings.py
        middlewares.py
        backends.py
    ...
```

These are basically:

- `frontier/settings.py`: the frontier settings file.
- `frontier/middlewares.py`: the middlewares used by the frontier.
- `frontier/backends.py`: the backend(s) used by the frontier.

2.10.2 How to access settings

Settings can be accessed through the `FrontierManager.settings` attribute, that is passed to *Middleware.from_manager* and *Backend.from_manager* class methods:

```
class MyMiddleware(Component):

    @classmethod
    def from_manager(cls, manager):
        manager = crawler.settings
        if settings.TEST_MODE:
            print "test mode is enabled!"
```

In other words, settings can be accessed as attributes of the `Settings` object.

2.10.3 Settings class

class `frontera.settings.`**`Settings`**(*module=None*, *attributes=None*)

2.10.4 Built-in frontier settings

Here's a list of all available Frontera settings, in alphabetical order, along with their default values and the scope where they apply.

AUTO_START

Default: `True`

Whether to enable frontier automatic start. See *Starting/Stopping the frontier*

BACKEND

Default: `'frontera.contrib.backends.memory.FIFO'`

The `Backend` to be used by the frontier. For more info see *Activating a backend*.

BC_MIN_REQUESTS

Default: `64`

Broad crawling queue get operation will keep retrying until specified number of requests is collected. Maximum number of retries is hard-coded to 3.

BC_MIN_HOSTS

Default: `24`

Keep retyring when getting requests from queue, until there are requests for specified minimum number of hosts collected. Maximum number of retries is hard-coded and equals 3.

BC_MAX_REQUESTS_PER_HOST

Default:: `128`

Don't include (if possible) batches of requests containing requests for specific host if there are already more then specified count of maximum requests per host. This is a suggestion for broad crawling queue get algorithm.

CANONICAL_SOLVER

Default: `frontera.contrib.canonicalsolvers.Basic`

The `CanonicalSolver` to be used by the frontier for resolving canonical URLs. For more info see *Canonical URL Solver*.

DELAY_ON_EMPTY

Default: `5.0`

Delay between calls to backend for new batches in Scrapy scheduler, when queue size is getting below `CONCURRENT_REQUESTS`. When backend has no requests to fetch, this delay helps to exhaust the rest of the buffer without hitting backend on every request. Increase it if calls to your backend is taking too long, and decrease if you need a fast spider bootstrap from seeds.

DISCOVERY_MAX_PAGES

Default: `100`

The maximum number of pages to schedule by Discovery crawling strategy.

DOMAIN_STATS_LOG_INTERVAL

Default: `300`

Time interval in seconds to rotate the domain statistics in *db worker* batch generator. Enabled only when logging set to DEBUG.

KAFKA_GET_TIMEOUT

Default: `5.0`

Time process should block until requested amount of data will be received from message bus. This is a general message bus setting with obsolete Kafka-related name.

LOCAL_MODE

Default: `True`

Sets single process run mode. Crawling strategy together with backend are used from the same spider process.

LOGGING_CONFIG

Default: `logging.conf`

The path to a file with logging module configuration. See https://docs.python.org/2/library/logging.config.html#logging-config-fileformat If file is absent, the logging system will be initialized with `logging.basicConfig()` and CONSOLE handler will be used. This option is used only in *db worker* and *strategy worker*.

MAX_NEXT_REQUESTS

Default: `64`

The maximum number of requests returned by `get_next_requests` API method. In distributed context it could be amount of requests produced per spider by *db worker* or count of requests read from message bus per attempt to fill the spider queue. In single process it's the count of requests to get from backend per one call to `get_next_requests` method.

MAX_REQUESTS

Default: `0`

Maximum number of returned requests after which Frontera is finished. If value is 0 (default), the frontier will continue indefinitely. See *Finishing the frontier*.

MESSAGE_BUS

Default: `frontera.contrib.messagebus.zeromq.MessageBus`

Points Frontera to *message bus* implementation. Defaults to ZeroMQ.

MESSAGE_BUS_CODEC

Default: `frontera.contrib.backends.remote.codecs.msgpack`

Points Frontera to *message bus* codec implementation. Here is the *codec interface description*. Defaults to MsgPack.

MIDDLEWARES

A list containing the middlewares enabled in the frontier. For more info see *Activating a middleware*.

Default:

```
[
    'frontera.contrib.middlewares.fingerprint.UrlFingerprintMiddleware',
]
```

NEW_BATCH_DELAY

Default: `30.0`

Used in DB worker, and it's a time interval between production of new batches for all partitions. If partition is busy, it will be skipped.

OVERUSED_KEEP_PER_KEY

Default: `1000`

After the purging this number of requests will be left in the queue.

OVERUSED_KEEP_KEYS

Default: `100`

The number of keys for purging to leave.

OVERUSED_MAX_KEYS

Default: `None`

A threshold triggering the keys purging in OverusedBuffer. The purging will end up leaving OVERUSED_KEEP_KEYS. `None` disables purging.

OVERUSED_MAX_PER_KEY

Default: `None`

Purging will start when reaching this number of requests per key and leave OVERUSED_KEEP_PER_KEY requests. `None` disables purging.

OVERUSED_SLOT_FACTOR

Default: `5.0`

(in progress + queued requests in that slot) / max allowed concurrent downloads per slot before slot is considered overused. This affects only Scrapy scheduler."

REQUEST_MODEL

Default: `'frontera.core.models.Request'`

The *`Request`* model to be used by the frontier.

RESPONSE_MODEL

Default: `'frontera.core.models.Response'`

The *`Response`* model to be used by the frontier.

SPIDER_LOG_CONSUMER_BATCH_SIZE

Default: `512`

This is a batch size used by strategy and db workers for consuming of spider log stream. Increasing it will cause worker to spend more time on every task, but processing more items per task, therefore leaving less time for other tasks during some fixed time interval. Reducing it will result to running several tasks within the same time interval, but with less overall efficiency. Use it when your consumers too slow, or too fast.

SCORING_LOG_CONSUMER_BATCH_SIZE

Default: `512`

This is a batch size used by db worker for consuming of scoring log stream. Use it when you need to adjust scoring log consumption speed.

SCORING_PARTITION_ID

Default: `0`

Used by strategy worker, and represents partition startegy worker assigned to.

SPIDER_LOG_PARTITIONS

Default: `1`

Number of *spider log* stream partitions. This affects number of required *strategy worker* (s), each strategy worker assigned to it's own partition.

SPIDER_FEED_PARTITIONS

Default: `1`

Number of *spider feed* partitions. This directly affects number of spider processes running. Every spider is assigned to it's own partition.

SPIDER_PARTITION_ID

Default: `0`

Per-spider setting, pointing spider to it's assigned partition.

STATE_CACHE_SIZE

Default: `1000000`

Maximum count of elements in state cache before it gets clear.

STORE_CONTENT

Default: `False`

Determines if content should be sent over the message bus and stored in the backend: a serious performance killer.

STRATEGY

Default: `frontera.worker.strategies.basic.BasicCrawlingStrategy`

The path to crawling strategy class.

STRATEGY_ARGS

Default: `{}`

Dict with default arguments for crawling strategy. Can be overridien with command line option in *strategy worker*.

SW_FLUSH_INTERVAL

Default: `300`

Interval between flushing of states in *strategy worker*. Also used to set initial random delay to flush states periodically, using formula `RANDINT(SW_FLUSH_INTERVAL)`.

TEST_MODE

Default: `False`

Whether to enable frontier test mode. See *Frontier test mode*

USER_AGENT

Default: `FronteraDiscoveryBot`

User agent string in use by Discovery crawling strategy.

2.10.5 Built-in fingerprint middleware settings

Settings used by the *UrlFingerprintMiddleware* and *DomainFingerprintMiddleware*.

URL_FINGERPRINT_FUNCTION

Default: `frontera.utils.fingerprint.sha1`

The function used to calculate the `url` fingerprint.

DOMAIN_FINGERPRINT_FUNCTION

Default: `frontera.utils.fingerprint.sha1`

The function used to calculate the `domain` fingerprint.

TLDEXTRACT_DOMAIN_INFO

Default: `False`

If set to `True`, will use tldextract to attach extra domain information (second-level, top-level and subdomain) to meta field (see *Adding additional data to objects*).

2.10.6 Built-in backends settings

SQLAlchemy

SQLALCHEMYBACKEND_CACHE_SIZE

Default: `10000`

SQLAlchemy Metadata LRU Cache size. It's used for caching objects, which are requested from DB every time already known, documents are crawled. This is mainly saves DB throughput, increase it if you're experiencing problems with too high volume of SELECT's to Metadata table, or decrease if you need to save memory.

SQLALCHEMYBACKEND_CLEAR_CONTENT

Default: `True`

Set to `False` if you need to disable table content clean up on backend instantiation (e.g. every Scrapy spider run).

SQLALCHEMYBACKEND_DROP_ALL_TABLES

Default: `True`

Set to `False` if you need to disable dropping of DB tables on backend instantiation (e.g. every Scrapy spider run).

SQLALCHEMYBACKEND_ENGINE

Default:: `sqlite:///:memory:`

SQLAlchemy database URL. Default is set to memory.

SQLALCHEMYBACKEND_ENGINE_ECHO

Default: `False`

Turn on/off SQLAlchemy verbose output. Useful for debugging SQL queries.

SQLALCHEMYBACKEND_MODELS

Default:

```
{
    'MetadataModel': 'frontera.contrib.backends.sqlalchemy.models.MetadataModel',
    'StateModel': 'frontera.contrib.backends.sqlalchemy.models.StateModel',
    'QueueModel': 'frontera.contrib.backends.sqlalchemy.models.QueueModel'
}
```

This is mapping with SQLAlchemy models used by backends. It is mainly used for customization. This setting uses a dictionary where `key` represents the name of the model to define and `value` the model to use.

Revisiting backend

SQLALCHEMYBACKEND_REVISIT_INTERVAL

Default: `timedelta(days=1)`

Time between document visits, expressed in `datetime.timedelta` objects. Changing of this setting will only affect documents scheduled after the change. All previously queued documents will be crawled with old periodicity.

HBase backend

HBASE_BATCH_SIZE

Default: `9216`

Count of accumulated PUT operations before they sent to HBase.

HBASE_DROP_ALL_TABLES

Default: `False`

Enables dropping and creation of new HBase tables on worker start.

HBASE_DOMAIN_METADATA_TABLE

Default: `domain_metadata`

Name of the domain metadata table in HBase.

HBASE_DOMAIN_METADATA_CACHE_SIZE

Default: 1000

The count of domain-value pairs cached in memory in *strategy worker*. Pairs are evicted from cache using LRU policy.

HBASE_DOMAIN_METADATA_BATCH_SIZE

Default: 100

Maximum count of domain-value pairs kept in write buffer before actual write happens.

HBASE_METADATA_TABLE

Default: `metadata`

Name of the documents metadata table.

HBASE_NAMESPACE

Default: `crawler`

Name of HBase namespace where all crawler related tables will reside.

HBASE_QUEUE_TABLE

Default: `queue`

Name of HBase priority queue table.

HBASE_STATE_WRITE_LOG_SIZE

Default: `15000`

Number of state changes in the *state cache* of *strategy worker*, before it get's flushed to HBase and cleared.

HBASE_STATE_CACHE_SIZE_LIMIT

Default: `3000000`

Number of cached state changes in the *state cache* of *strategy worker*. Internally there is `cachetools.LRUCache` storing all the recent state changes, discarding least recently used when the cache gets over its capacity.

HBASE_STATES_TABLE

Default: `states`

Name of the table used by *strategy worker* to store link states.

HBASE_THRIFT_HOST

Default: `localhost`

HBase Thrift server host.

HBASE_THRIFT_PORT

Default: `9090`

HBase Thrift server port

HBASE_USE_FRAMED_COMPACT

Default: `False`

Enabling this option dramatically reduces transmission overhead, but the server needs to be properly configured to use Thrifts framed transport and compact protocol.

HBASE_USE_SNAPPY

Default: `False`

Whatever to compress content and metadata in HBase using Snappy. Decreases amount of disk and network IO within HBase, lowering response times. HBase have to be properly configured to support Snappy compression.

2.10.7 ZeroMQ message bus settings

The message bus class is `distributed_frontera.messagebus.zeromq.MessageBus`

ZMQ_ADDRESS

Default: `127.0.0.1`

Defines where the ZeroMQ socket should bind or connect. Can be a hostname or an IP address. Right now ZMQ has only been properly tested with IPv4. Proper IPv6 support will be added in the near future.

ZMQ_BASE_PORT

Default: `5550`

The base port for all ZeroMQ sockets. It uses 6 sockets overall and port starting from base with step 1. Be sure that interval [base:base+5] is available.

2.10.8 Kafka message bus settings

The message bus class is `frontera.contrib.messagebus.kafkabus.MessageBus`

KAFKA_LOCATION

Hostname and port of kafka broker, separated with :. Can be a string with hostname:port pair separated with commas(,).

KAFKA_CODEC

Default: `KAFKA_CODEC`

Kafka-python 1.0.x version compression codec to use, is a string and could be one of `none`, `snappy`, `gzip` or `lz4`.

KAFKA_CERT_PATH

OS path to the folder with three certificate files: ca-cert.pem, client-cert.pem, client-key.pem.

KAFKA_ENABLE_SSL

Boolean. Set to True to enable SSL connection in Kafka client.

SPIDER_LOG_DBW_GROUP

Default: `dbw-spider-log`

Kafka consumer group name, used for *spider log* by *db worker* s.

SPIDER_LOG_SW_GROUP

Default: `sw-spider-log`

Kafka consumer group name, used for *spider log* by *strategy worker* (s).

SCORING_LOG_DBW_GROUP

Default: `dbw-scoring-log`

Kafka consumer group name, used for *scoring log* by *db worker* (s).

SPIDER_FEED_GROUP

Default: `fetchers-spider-feed`

Kafka consumer group name, used for *spider feed* by *spider* (s).

SPIDER_LOG_TOPIC

Default: `frontier-done`

spider log stream topic name.

SPIDER_FEED_TOPIC

Default: `frontier-todo`

spider feed stream topic name.

SCORING_LOG_TOPIC

Kafka topic used for *scoring log* stream.

2.10.9 Default settings

If no settings are specified, frontier will use the built-in default ones. For a complete list of default values see: *Built-in settings reference*. All default settings can be overridden.

Installation Guide HOWTO and Dependencies options.

Crawling strategies A list of built-in crawling strategies.

Frontier objects Understand the classes used to represent requests and responses.

Middlewares Filter or alter information for links and documents.

Canonical URL Solver Identify and make use of canonical url of document.

Backends Built-in backends, and tips on implementing your own.

Message bus Built-in message bus reference.

Writing custom crawling strategy Implementing your own crawling strategy.

Using the Frontier with Scrapy Learn how to use Frontera with Scrapy.

Settings Settings reference.

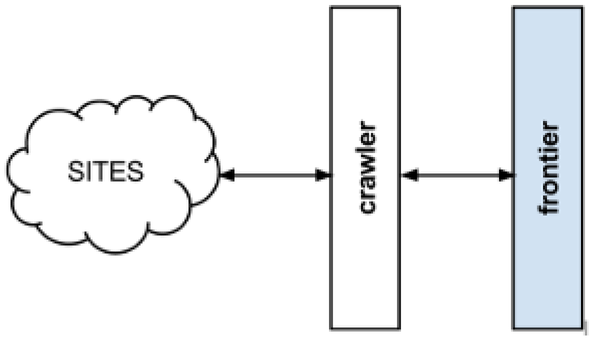
CHAPTER 3

Advanced usage

3.1 What is a Crawl Frontier?

Frontera is a crawl frontier framework, the part of a crawling system that decides the logic and policies to follow when a crawler is visiting websites (what pages should be crawled next, priorities and ordering, how often pages are revisited, etc).

A usual crawl frontier scheme is:



The frontier is initialized with a list of start URLs, that we call the seeds. Once the frontier is initialized the crawler asks it what pages should be visited next. As the crawler starts to visit the pages and obtains results, it will inform the frontier of each page response and also of the extracted hyperlinks contained within the page. These links are added by the frontier as new requests to visit according to the frontier policies.

This process (ask for new requests/notify results) is repeated until the end condition for the crawl is reached. Some crawlers may never stop, that's what we call continuous crawls.

Frontier policies can be based on almost any logic. Common use cases are usually based on scores/priorities, computed from one or many page attributes (freshness, update times, content relevance for certain terms, etc). They can also be based in really simple logic as FIFO/LIFO or DFS/BFS page visit ordering.

Depending on frontier logic, a persistent storage system may be needed to store, update or query information about the pages. Other systems can be 100% volatile and not share any information at all between different crawls.

Please refer for further crawl frontier theory at URL frontier article of Introduction to Information Retrieval book by Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze.

3.2 Graph Manager

The Graph Manager is a tool to represent web sitemaps as a graph.

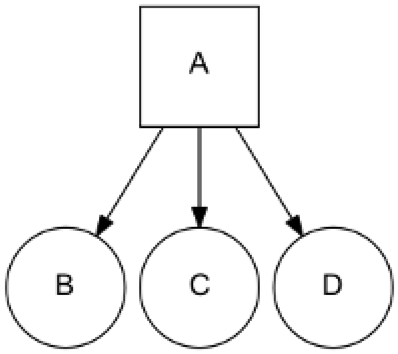
It can easily be used to test frontiers. We can "fake" crawler request/responses by querying pages to the graph manager, and also know the links extracted for each one without using a crawler at all. You can make your own fake tests or use the *Frontier Tester tool*.

You can use it by defining your own sites for testing or use the *Scrapy Recorder* to record crawlings that can be reproduced later.

3.2.1 Defining a Site Graph

Pages from a web site and its links can be easily defined as a directed graph, where each node represents a page and the edges the links between them.

Let's use a really simple site representation with a starting page *A* that have links inside to tree pages *B, C, D*. We can represent the site with this graph:



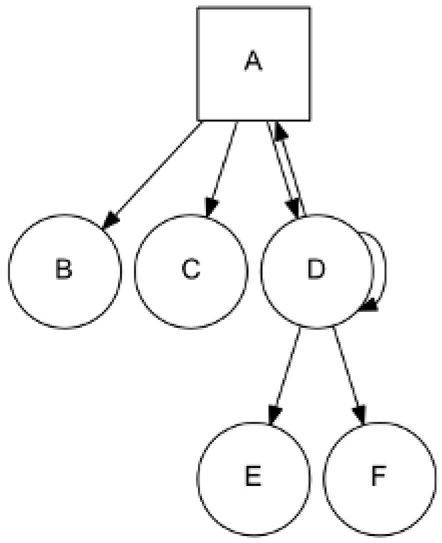
We use a list to represent the different site pages and one tuple to define the page and its links, for the previous example:

```
site = [
    ('A', ['B', 'C', 'D']),
]
```

Note that we don't need to define pages without links, but we can also use it as a valid representation:

```
site = [
    ('A', ['B', 'C', 'D']),
    ('B', []),
    ('C', []),
    ('D', []),
]
```

A more complex site:

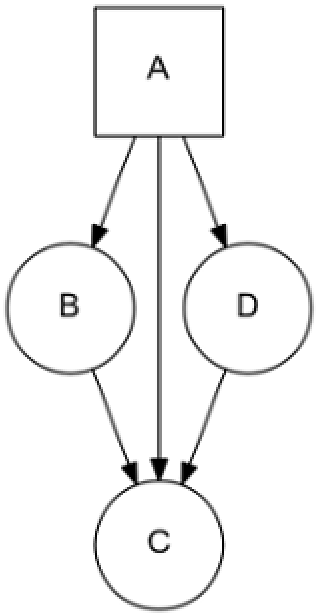


Can be represented as:

```
site = [
    ('A', ['B', 'C', 'D']),
    ('D', ['A', 'D', 'E', 'F']),
]
```

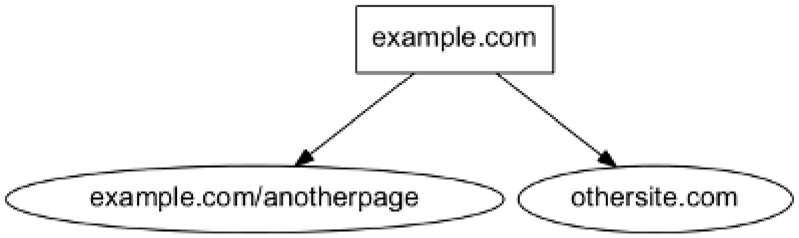
Note that *D* is linking to itself and to his parent *A*.

In the same way, a page can have several parents:



```
site = [
    ('A', ['B', 'C', 'D']),
    ('B', ['C']),
    ('D', ['C']),
]
```

In order to simplify examples we're not using urls for page representation, but of course urls are the intended use for site graphs:



```
site = [
    ('http://example.com', ['http://example.com/anotherpage', 'http://othersite.com
↪']),
]
```

3.2.2 Using the Graph Manager

Once we have defined our site represented as a graph, we can start using it with the Graph Manager.

We must first create our graph manager:

```
>>> from frontera.utils import graphs
>>> g = graphs.Manager()
```

And add the site using the *add_site* method:

```
>>> site = [('A', ['B', 'C', 'D'])]
>>> g.add_site(site)
```

The manager is now initialized and ready to be used.

We can get all the pages in the graph:

```
>>> g.pages
[<1:A*>, <2:B>, <3:C>, <4:D>]
```

Asterisk represents that the page is a seed, if we want to get just the seeds of the site graph:

```
>>> g.seeds
[<1:A*>]
```

We can get individual pages using *get_page*, if a page does not exists None is returned

```
>>> g.get_page('A')
<1:A*>
```

```
>>> g.get_page('F')
None
```

3.2.3 CrawlPage objects

Pages are represented as a `CrawlPage` object:

class CrawlPage
A `CrawlPage` object represents an Graph Manager page, which is usually generated in the Graph Manager.

id
Autonumeric page id.

url
The url of the page.

status
Represents the HTTP code status of the page.

is_seed
Boolean value indicating if the page is seed or not.

links
List of pages the current page links to.

referers
List of pages that link to the current page.

In our example:

```
>>> p = g.get_page('A')
>>> p.id
1

>>> p.url
u'A'

>>> p.status  # defaults to 200
u'200'

>>> p.is_seed
True

>>> p.links
[<2:B>, <3:C>, <4:D>]

>>> p.referers  # No referers for A
[]

>>> g.get_page('B').referers  # referers for B
[<1:A*>]
```

3.2.4 Adding pages and Links

Site graphs can be also defined adding pages and links individually, the same graph from our example can be defined this way:

```
>>> g = graphs.Manager()
>>> a = g.add_page(url='A', is_seed=True)
>>> b = g.add_link(page=a, url='B')
>>> c = g.add_link(page=a, url='C')
>>> d = g.add_link(page=a, url='D')
```

add_page and *add_link* can be combined with *add_site* and used anytime:

```
>>> site = [('A', ['B', 'C', 'D'])]
>>> g = graphs.Manager()
>>> g.add_site(site)
```

(continues on next page)

(continued from previous page)

```
>>> d = g.get_page('D')
>>> g.add_link(d, 'E')
```

3.2.5 Adding multiple sites

Multiple sites can be added to the manager:

```
>>> site1 = [('A1', ['B1', 'C1', 'D1'])]
>>> site2 = [('A2', ['B2', 'C2', 'D2'])]

>>> g = graphs.Manager()
>>> g.add_site(site1)
>>> g.add_site(site2)

>>> g.pages
[<1:A1*>, <2:B1>, <3:C1>, <4:D1>, <5:A2*>, <6:B2>, <7:C2>, <8:D2>]

>>> g.seeds
[<1:A1*>, <5:A2*>]
```

Or as a list of sites with *add_site_list* method:

```
>>> site_list = [
    [('A1', ['B1', 'C1', 'D1'])],
    [('A2', ['B2', 'C2', 'D2'])],
]
>>> g = graphs.Manager()
>>> g.add_site_list(site_list)
```

3.2.6 Graphs Database

Graph Manager uses SQLAlchemy to store and represent graphs.

By default it uses an in-memory SQLite database as a storage engine, but any databases supported by SQLAlchemy can be used.

An example using SQLite:

```
>>> g = graphs.Manager(engine='sqlite:///graph.db')
```

Changes are committed with every new add by default, graphs can be loaded later:

```
>>> graph = graphs.Manager(engine='sqlite:///graph.db')
>>> graph.add_site(('A', []))

>>> another_graph = graphs.Manager(engine='sqlite:///graph.db')
>>> another_graph.pages
[<1:A1*>]
```

A database content reset can be done using *clear_content* parameter:

```
>>> g = graphs.Manager(engine='sqlite:///graph.db', clear_content=True)
```

3.2.7 Using graphs with status codes

In order to recreate/simulate crawling using graphs, HTTP response codes can be defined for each page.

Example for a 404 error:

```
>>> g = graphs.Manager()
>>> g.add_page(url='A', status=404)
```

Status codes can be defined for sites in the following way using a list of tuples:

```
>>> site_with_status_codes = [
    ((200, "A"), ["B", "C"]),
    ((404, "B"), ["D", "E"]),
    ((500, "C"), ["F", "G"]),
]
>>> g = graphs.Manager()
>>> g.add_site(site_with_status_codes)
```

Default status code value is 200 for new pages.

3.2.8 A simple crawl faking example

Frontier tests can better be done using the *Frontier Tester tool*, but here's an example of how fake a crawl with a frontier:

```
from frontera import FrontierManager, Request, Response
from frontera.utils import graphs

if __name__ == '__main__':
    # Load graph from existing database
    graph = graphs.Manager('sqlite:///graph.db')

    # Create frontier from default settings
    frontier = FrontierManager.from_settings()

    # Create and add seeds
    seeds = [Request(seed.url) for seed in graph.seeds]
    frontier.add_seeds(seeds)

    # Get next requests
    next_requets = frontier.get_next_requests()

    # Crawl pages
    while (next_requests):
        for request in next_requests:

            # Fake page crawling
            crawled_page = graph.get_page(request.url)

            # Create response
            response = Response(url=crawled_page.url, status_code=crawled_page.status)

            # Update Page
            page = frontier.page_crawled(response=response
                                         links=[link.url for link in crawled_page.
↪links])
```

(continues on next page)

(continued from previous page)

```
            # Get next requests
            next_requets = frontier.get_next_requests()
```

3.2.9 Rendering graphs

Graphs can be rendered to png files:

```
>>> g.render(filename='graph.png', label='A simple Graph')
```

Rendering graphs uses pydot, a Python interface to Graphviz's Dot language.

3.2.10 How to use it

Graph Manager can be used to test frontiers in conjunction with *Frontier Tester* and also with *Scrapy Recordings*.

3.3 Recording a Scrapy crawl

Scrapy Recorder is a set of Scrapy middlewares that will allow you to record a scrapy crawl and store it into a *Graph Manager*.

This can be useful to perform frontier tests without having to crawl the entire site again or even using Scrapy.

3.3.1 Activating the recorder

The recorder uses 2 different middlewares: `CrawlRecorderSpiderMiddleware` and `CrawlRecorderDownloaderMiddleware`.

To activate the recording in your Scrapy project, just add them to the SPIDER_MIDDLEWARES and DOWNLOADER_MIDDLEWARES settings:

```
SPIDER_MIDDLEWARES.update({
    'frontera.contrib.scrapy.middlewares.recording.CrawlRecorderSpiderMiddleware':␣
↪1000,
})

DOWNLOADER_MIDDLEWARES.update({
    'frontera.contrib.scrapy.middlewares.recording.CrawlRecorderDownloaderMiddleware
↪': 1000,
})
```

3.3.2 Choosing your storage engine

As *Graph Manager* is internally used by the recorder to store crawled pages, you can choose between *different storage engines*.

We can set the storage engine with the *RECORDER_STORAGE_ENGINE* setting:

```
RECORDER_STORAGE_ENGINE = 'sqlite:///my_record.db'
```

You can also choose to reset database tables or just reset data with this settings:

```
RECORDER_STORAGE_DROP_ALL_TABLES = True
RECORDER_STORAGE_CLEAR_CONTENT = True
```

3.3.3 Running the Crawl

Just run your Scrapy spider as usual from the command line:

```
scrapy crawl myspider
```

Once it's finished you should have the recording available and ready for use.

In case you need to disable recording, you can do it by overriding the *RECORDER_ENABLED* setting:

```
scrapy crawl myspider -s RECORDER_ENABLED=False
```

3.3.4 Recorder settings

Here's a list of all available Scrapy Recorder settings, in alphabetical order, along with their default values and the scope where they apply.

RECORDER_ENABLED

Default: `True`

Activate or deactivate recording middlewares.

RECORDER_STORAGE_CLEAR_CONTENT

Default: `True`

Deletes table content from *storage database* in Graph Manager.

RECORDER_STORAGE_DROP_ALL_TABLES

Default: `True`

Drop *storage database* tables in Graph Manager.

RECORDER_STORAGE_ENGINE

Default: `None`

Sets *Graph Manager storage engine* used to store the recording.

3.4 Fine tuning of Frontera cluster

3.4.1 Why crawling speed is so low?

Search for a bottleneck.

- All requests are targeted towards a few websites.
- DNS resolution (see *DNS Service* article),
- *strategy worker* performance,
- *db worker* batch generation insufficiency.
- HBase response times are too high,
- Network within cluster is overloaded.

3.4.2 Tuning HBase

- Increase block cache in HBase.
- Put Thrift server on each HBase region server and spread load from SW to Thrift.
- Enable Snappy compression (see *HBASE_USE_SNAPPY*).

3.4.3 Tuning Kafka

- Decrease the log size to minimum and optimize the system to avoid storing in Kafka huge volumes of data. Once data was written it should be consumed as fast as possible.
- Use SSD or even RAM storage for Kafka logs,
- Enable Snappy compression for Kafka.

3.4.4 Flow control between various components

The *MAX_NEXT_REQUESTS* is used for controlling the batch size. In spiders config it controls how much items will be consumed per one `get_next_requests` call. At the same time in DB worker config it sets count of items to generate per partition. When setting these parameters keep in mind:

- DB worker and spider values have to be consistent to avoid overloading of message bus and loosing messages. In other words, DB worker have to produce slightly more than consumed by spiders, because the spider should still be able to fetch new pages even though the DB worker has not pushed a new batch yet.
- Spider consumption rate depends on many factors: internet connection latency, amount of spider parsing/scraping work, delays and auto throttling settings, usage of proxies, etc.
- Keep spider queue always full to prevent spider idling.
- General recommendation is to set DB worker value 2-4 times bigger than spiders.
- Batch size shouldn't be big to not generate too much load on backend, and allow system quickly react on queue changes.
- Watch out warnings about lost messages.

3.5 DNS Service

Along with what was mentioned in *Prerequisites* you may need also a dedicated DNS Service with caching. Especially, if your crawler is expected to generate substantial number of DNS queries. It is true for breadth-first crawling, or any other strategies, implying accessing large number of websites, within short period of time.

Because of huge load DNS service may get blocked by your network provider eventually.

There are two options for DNS strategy:

- Recursive DNS resolution,
- using upstream servers (massive DNS caches like OpenDNS or Verizon).

The second is still prone to blocking.

There is good DNS server software https://www.unbound.net/ released by NLnet Labs. It allows to choose one of above mentioned strategies and maintain your local DNS cache.

Have a look at Scrapy options `REACTOR_THREADPOOL_MAXSIZE` and `DNS_TIMEOUT`.

What is a Crawl Frontier? Learn Crawl Frontier theory.

Graph Manager Define fake crawlings for websites to test your frontier.

Recording a Scrapy crawl Create Scrapy crawl recordings and reproduce them later.

Fine tuning of Frontera cluster Cluster deployment and fine tuning information.

DNS Service Few words about DNS service setup.

CHAPTER 4

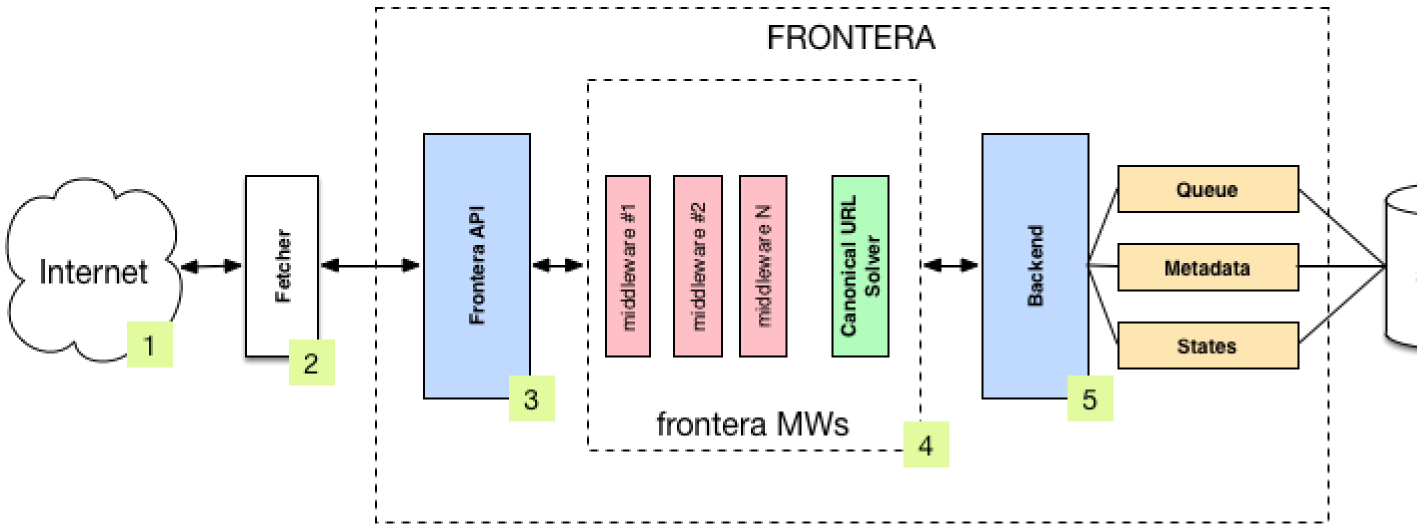
Developer documentation

4.1 Architecture overview

This document describes the Frontera Manager pipeline, distributed components and how they interact.

4.1.1 Single process

The following diagram shows an architecture of the Frontera pipeline with its components (referenced by numbers) and an outline of the data flow that takes place inside the system. A brief description of the components is included below with links for more detailed information about them. The data flow is also described below.



Components

Fetcher

The Fetcher (2) is responsible for fetching web pages from the sites (1) and feeding them to the frontier which manages what pages should be crawled next.

Fetcher can be implemented using Scrapy or any other crawling framework/system as the framework offers a generic frontier functionality.

In distributed run mode Fetcher is replaced with message bus producer from Frontera Manager side and consumer from Fetcher side.

Frontera API / Manager

The main entry point to Frontera API (3) is the `FrontierManager` object. Frontier users, in our case the Fetcher (2), will communicate with the frontier through it.

For more information see *Frontera API*.

Middlewares

Frontier middlewares (4) are specific hooks that sit between the Manager (3) and the Backend (5). These middlewares process `Request` and `Response` objects when they pass to and from the Frontier and the Backend. They provide a convenient mechanism for extending functionality by plugging custom code. Canonical URL solver is a specific case of middleware responsible for substituting non-canonical document URLs wiht canonical ones.

For more information see *Middlewares* and *Canonical URL Solver*

Backend

The frontier Backend (5) is where the crawling logic/policies lies. It's responsible for receiving all the crawl info and selecting the next pages to be crawled. Backend is meant to be operating on higher level, and `Queue`, `Metadata` and `States` objects are responsible for low-level storage communication code.

May require, depending on the logic implemented, a persistent storage (6) to manage `Request` and `Response` objects info.

For more information see *Backends*.

Data Flow

The data flow in Frontera is controlled by the Frontier Manager, all data passes through the manager-middlewares-backend scheme and goes like this:

1. The frontier is initialized with a list of seed requests (seed URLs) as entry point for the crawl.
2. The fetcher asks for a list of requests to crawl.
3. Each url is fetched and the frontier is notified back of the crawl result as well of the extracted data the page contains. If anything went wrong during the crawl, the frontier is also informed of it.

Once all urls have been crawled, steps 2-3 are repeated until crawl of frontier end condition is reached. Each loop (steps 2-3) repetition is called a *frontier iteration*.

4.1.2 Distributed

The same Frontera Manager pipeline is used in all Frontera processes when running in distributed mode.

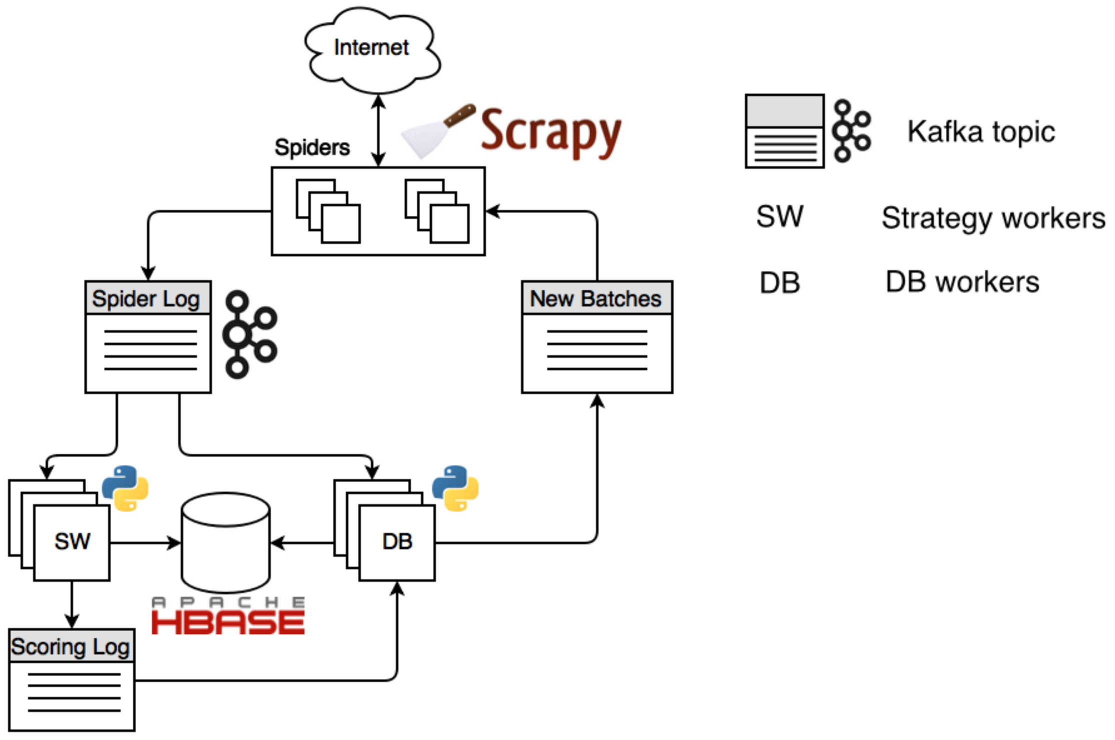
Overall system forms a closed circle and all the components are working as daemons in infinite cycles. There is a *message bus* responsible for transmitting messages between components, persistent storage and fetchers (when combined with extraction these processes called spiders). There is a transport and storage layer abstractions, so one can plug it's own transport. Distributed backend run mode has instances of three types:

- **Spiders or fetchers, implemented using Scrapy (sharded).** Responsible for resolving DNS queries, getting content from the Internet and doing link (or other data) extraction from content.
- **Strategy workers (sharded).** Run the crawling strategy code: scoring the links, deciding if link needs to be scheduled and when to stop crawling.
- **DB workers (sharded).** Store all the metadata, including scores and content, and generating new batches for downloading by spiders.

Where *sharded* means component consumes messages of assigned partition only, e.g. processes certain share of the stream, and *replicated* is when components consume stream regardless of partitioning.

Such design allows to operate online. Crawling strategy can be changed without having to stop the crawl. Also *crawling strategy* can be implemented as a separate module; containing logic for checking the crawling stopping condition, URL ordering, and scoring model.

Frontera is polite to web hosts by design and each host is downloaded by no more than one spider process. This is achieved by stream partitioning.



Data flow

Let's start with spiders. The seed URLs defined by the user inside spiders are propagated to strategy workers and DB workers by means of *spider log* stream. Strategy workers decide which pages to crawl using state cache, assigns a score to each page and sends the results to the *scoring log* stream.

DB Worker stores all kinds of metadata, including content and scores. Also DB worker checks for the spider's consumers offsets and generates new batches if needed and sends them to *spider feed* stream. Spiders consume these batches, downloading each page and extracting links from them. The links are then sent to the 'Spider Log' stream where they are stored and scored. That way the flow repeats indefinitely.

4.2 Frontera API

This section documents the Frontera core API, and is intended for developers of middlewares and backends.

4.2.1 Frontera API / Manager

The main entry point to Frontera API is the `FrontierManager` object, passed to middlewares and backend through the from_manager class method. This object provides access to all Frontera core components, and is the only way for middlewares and backend to access them and hook their functionality into Frontera.

The `FrontierManager` is responsible for loading the installed middlewares and backend, as well as for managing the data flow around the whole frontier.

4.2.2 Loading from settings

Although `FrontierManager` can be initialized using parameters the most common way of doing this is using *Frontera Settings*.

This can be done through the `from_settings` class method, using either a string path:

```
>>> from frontera import FrontierManager
>>> frontier = FrontierManager.from_settings('my_project.frontier.settings')
```

or a `BaseSettings` object instance:

```
>>> from frontera import FrontierManager, Settings
>>> settings = Settings()
>>> settings.MAX_PAGES = 0
>>> frontier = FrontierManager.from_settings(settings)
```

It can also be initialized without parameters, in this case the frontier will use the *default settings*:

```
>>> from frontera import FrontierManager, Settings
>>> frontier = FrontierManager.from_settings()
```

4.2.3 Frontier Manager

4.2.4 Starting/Stopping the frontier

Sometimes, frontier components need to perform initialization and finalization operations. The frontier mechanism to notify the different components of the frontier start and stop is done by the `start()` and `stop()` methods respectively.

By default `auto_start` frontier value is activated, this means that components will be notified once the `FrontierManager` object is created. If you need to have more fine control of when different components are initialized, deactivate `auto_start` and manually call frontier API `start()` and `stop()` methods.

Note: Frontier `stop()` method is not automatically called when `auto_start` is active (because frontier is not aware of the crawling state). If you need to notify components of frontier end you should call the method manually.

4.2.5 Frontier iterations

Once frontier is running, the usual process is the one described in the *data flow* section.

Crawler asks the frontier for next pages using the `get_next_requests()` method. Each time the frontier returns a non empty list of pages (data available), is what we call a frontier iteration.

Current frontier iteration can be accessed using the `iteration` attribute.

4.2.6 Finishing the frontier

Crawl can be finished either by the Crawler or by the Frontera. Frontera will finish when a maximum number of pages is returned. This limit is controlled by the `max_requests` attribute (*`MAX_REQUESTS`* setting).

If `max_requests` has a value of 0 (default value) the frontier will continue indefinitely.

Once the frontier is finished, no more pages will be returned by the `get_next_requests` method and `finished` attribute will be True.

4.2.7 Component objects

class `frontera.core.components.`**`Component`**

Interface definition for a frontier component The `Component` object is the base class for frontier `Middleware` and `Backend` objects.

`FrontierManager` communicates with the active components using the hook methods listed below.

Implementations are different for `Middleware` and `Backend` objects, therefore methods are not fully described here but in their corresponding section.

Attributes

`name`

The component name

Abstract methods

`frontier_start``()`

Called when the frontier starts, see *starting/stopping the frontier*.

`frontier_stop``()`

Called when the frontier stops, see *starting/stopping the frontier*.

`page_crawled``(`*response*`)`

This method is called every time a page has been crawled.

Parameters **`response`** (*`object`*) – The `Response` object for the crawled page.

`request_error``(`*page*, *error*`)`

This method is called each time an error occurs when crawling a page.

Parameters

- **request** (*object*) – The crawled with error *Request* object.
- **error** (*string*) – A string identifier for the error.

Class Methods

classmethod from_manager(*manager*)

Class method called from FrontierManager passing the manager itself.

Example of usage:

```
def from_manager(cls, manager):
    return cls(settings=manager.settings)
```

4.2.8 Test mode

In some cases while testing, frontier components need to act in a different way than they usually do (for instance *domain middleware* accepts non valid URLs like 'A1' or 'B1' when parsing domain urls in test mode).

Components can know if the frontier is in test mode via the boolean test_mode attribute.

4.2.9 Other ways of using the frontier

Communication with the frontier can also be done through other mechanisms such as an HTTP API or a queue system. These functionalities are not available for the time being, but hopefully will be included in future versions.

4.3 Using the Frontier with Requests

To integrate frontier with Requests library, there is a RequestsFrontierManager class available.

This class is just a simple FrontierManager wrapper that uses Requests objects (Request/Response) and converts them from and to frontier ones for you.

Use it in the same way that FrontierManager, initialize it with your settings and use Requests Request and Response objects. get_next_requests method will return a Requests Request object.

An example:

```
import re

import requests

from urlparse import urljoin

from frontera.contrib.requests.manager import RequestsFrontierManager
from frontera import Settings

SETTINGS = Settings()
SETTINGS.BACKEND = 'frontera.contrib.backends.memory.FIFO'
SETTINGS.LOGGING_MANAGER_ENABLED = True
SETTINGS.LOGGING_BACKEND_ENABLED = True
SETTINGS.MAX_REQUESTS = 100
SETTINGS.MAX_NEXT_REQUESTS = 10
```

(continues on next page)

(continued from previous page)

```
SEEDS = [
    'http://www.imdb.com',
]

LINK_RE = re.compile(r'href="(.*?)"')


def extract_page_links(response):
    return [urljoin(response.url, link) for link in LINK_RE.findall(response.text)]

if __name__ == '__main__':

    frontier = RequestsFrontierManager(SETTINGS)
    frontier.add_seeds([requests.Request(url=url) for url in SEEDS])
    while True:
        next_requests = frontier.get_next_requests()
        if not next_requests:
            break
        for request in next_requests:
                try:
                    response = requests.get(request.url)
                    links = [requests.Request(url=url) for url in extract_page_
↪links(response)]
                    frontier.page_crawled(response=response)
                    frontier.links_extracted(request=request, links=links)
                except requests.RequestException, e:
                    error_code = type(e).__name__
                    frontier.request_error(request, error_code)
```

4.4 Examples

The project repo includes an `examples` folder with some scripts and projects using Frontera:

```
examples/
    requests/
    general-spider/
    scrapy_recording/
    scripts/
```

- **requests**: Example script with Requests library.
- **general-spider**: Scrapy integration example project.
- **scrapy_recording**: Scrapy Recording example project.
- **scripts**: Some simple scripts.

Note: This examples may need to install additional libraries in order to work.

You can install them using pip:

```
pip install -r requirements/examples.txt
```

4.4.1 requests

A simple script that follow all the links from a site using Requests library.

How to run it:

```
python links_follower.py
```

4.4.2 general-spider

A simple Scrapy spider that follows all the links from the seeds. Contains configuration files for single process, distributed spider and backends run modes.

See *Quick start distributed mode* for how to run it.

4.4.3 cluster

Is a large scale crawling application for performing broad crawls with number of pages per host limit. It preserves each host state in HBase and uses it when schedule new requests for downloading. Designed for running in distributed backend run mode using HBase.

4.4.4 scrapy_recording

A simple script with a spider that follows all the links for a site, recording crawling results.

How to run it:

```
scrapy crawl recorder
```

4.4.5 scripts

Some sample scripts on how to use different frontier components.

4.5 Tests

Frontera tests are implemented using the pytest tool.

You can install pytest and the additional required libraries used in the tests using pip:

```
pip install -r requirements/tests.txt
```

4.5.1 Running tests

To run all tests go to the root directory of source code and run:

```
py.test
```

4.5.2 Writing tests

All functionality (including new features and bug fixes) must include a test case to check that it works as expected, so please include tests for your patches if you want them to get accepted sooner.

4.5.3 Backend testing

A base pytest class for *Backend* testing is provided: `BackendTest`

Let's say for instance that you want to test to your backend `MyBackend` and create a new frontier instance for each test method call, you can define a test class like this:

```
class TestMyBackend(backends.BackendTest):

    backend_class = 'frontera.contrib.backend.abackend.MyBackend'

    def test_one(self):
        frontier = self.get_frontier()
        ...

    def test_two(self):
        frontier = self.get_frontier()
        ...

    ...
```

And let's say too that it uses a database file and you need to clean it before and after each test:

```
class TestMyBackend(backends.BackendTest):

    backend_class = 'frontera.contrib.backend.abackend.MyBackend'

    def setup_backend(self, method):
        self._delete_test_db()

    def teardown_backend(self, method):
        self._delete_test_db()

    def _delete_test_db(self):
        try:
            os.remove('mytestdb.db')
        except OSError:
            pass

    def test_one(self):
        frontier = self.get_frontier()
        ...

    def test_two(self):
        frontier = self.get_frontier()
        ...

    ...
```

4.5.4 Testing backend sequences

To test *Backend* crawling sequences you can use the `BackendSequenceTest` class.

`BackendSequenceTest` class will run a complete crawl of the passed site graphs and return the sequence used by the backend for visiting the different pages.

Let's say you want to test to a backend that sort pages using alphabetic order. You can define the following test:

```python
class TestAlphabeticSortBackend(backends.BackendSequenceTest):

    backend_class = 'frontera.contrib.backend.abackend.AlphabeticSortBackend'

    SITE_LIST = [
        [
            ('C', []),
            ('B', []),
            ('A', []),
        ],
    ]

    def test_one(self):
        # Check sequence is the expected one
        self.assert_sequence(site_list=self.SITE_LIST,
                             expected_sequence=['A', 'B', 'C'],
                             max_next_requests=0)

    def test_two(self):
        # Get sequence and work with it
        sequence = self.get_sequence(site_list=SITE_LIST,
                            max_next_requests=0)
        assert len(sequence) > 2

    ...
```

4.6 Logging

Frontera is using Python native logging system. This allows a user to manage logged messages by writing a logger configuration file (see *LOGGING_CONFIG*) or configuring logging system during runtime.

Logger configuration syntax is here https://docs.python.org/2/library/logging.config.html

4.6.1 Loggers used

- kafka
- hbase.backend
- hbase.states
- hbase.queue
- sqlalchemy.revisiting.queue
- sqlalchemy.metadata
- sqlalchemy.states

- sqlalchemy.queue
- offset-fetcher
- overusedbuffer
- messagebus-backend
- cf-server
- db-worker
- strategy-worker
- messagebus.kafka
- memory.queue
- memory.dequequeue
- memory.states
- manager.components
- manager
- frontera.contrib.scrapy.schedulers.FronteraScheduler

4.7 Testing a Frontier

Frontier Tester is a helper class for easy frontier testing.

Basically it runs a fake crawl against a Frontier, crawl info is faked using a *Graph Manager* instance.

4.7.1 Creating a Frontier Tester

FrontierTester needs a *Graph Manager* and a `FrontierManager` instances:

```
>>> from frontera import FrontierManager, FrontierTester
>>> from frontera.utils import graphs
>>> graph = graphs.Manager('sqlite:///graph.db')  # Crawl fake data loading
>>> frontier = FrontierManager.from_settings()  # Create frontier from default␣
↪settings
>>> tester = FrontierTester(frontier, graph)
```

4.7.2 Running a Test

The tester is now initialized, to run the test just call the method *run*:

```
>>> tester.run()
```

When run method is called the tester will:

1. Add all the seeds from the graph.
2. Ask the frontier about next pages.
3. Fake page response and inform the frontier about page crawl and its links.

Steps 1 and 2 are repeated until crawl or frontier ends.

Once the test is finished, the crawling page `sequence` is available as a list of frontier *`Request`* objects.

4.7.3 Test Parameters

In some test cases you may want to add all graph pages as seeds, this can be done with the parameter `add_all_pages`:

```
>>> tester.run(add_all_pages=True)
```

Maximum number of returned pages per `get_next_requests` call can be set using frontier settings, but also can be modified when creating the FrontierTester with the `max_next_pages` argument:

```
>>> tester = FrontierTester(frontier, graph, max_next_pages=10)
```

4.7.4 An example of use

A working example using test data from graphs and *backends*:

```
from frontera import FrontierManager, Settings, FrontierTester, graphs


def test_backend(backend):
    # Graph
    graph = graphs.Manager()
    graph.add_site_list(graphs.data.SITE_LIST_02)

    # Frontier
    settings = Settings()
    settings.BACKEND = backend
    settings.TEST_MODE = True
    frontier = FrontierManager.from_settings(settings)

    # Tester
    tester = FrontierTester(frontier, graph)
    tester.run(add_all_pages=True)

    # Show crawling sequence
    print '-'*40
    print frontier.backend.name
    print '-'*40
    for page in tester.sequence:
        print page.url

if __name__ == '__main__':
    test_backend('frontera.contrib.backends.memory.heapq.FIFO')
    test_backend('frontera.contrib.backends.memory.heapq.LIFO')
    test_backend('frontera.contrib.backends.memory.heapq.BFS')
    test_backend('frontera.contrib.backends.memory.heapq.DFS')
```

4.8 Contribution guidelines

- Use Frontera google group for all questions and discussions.

- Use Github repo pull request for submitting patches.
- Use Github repo issues for issues Frontera could benefit from in the future. Please don't put your own problems running Frontera there, instead use a google group.

We're always happy to accept well-thought solution with documentation and tests.

4.9 Glossary

spider log A stream of encoded messages from spiders. Each message is product of extraction from document content. Most of the time it is links, scores, classification results.

scoring log Contains score updating events and scheduling flag (if link needs to be scheduled for download) going from strategy worker to db worker.

spider feed A stream of messages from *db worker* to spiders containing new batches of documents to crawl.

strategy worker Special type of worker, running the *crawling strategy* code: scoring the links, deciding if link needs to be scheduled (consults *state cache*) and when to stop crawling. That type of worker is sharded.

db worker Is responsible for communicating with storage DB, and mainly saving metadata and content along with retrieving new batches to download.

state cache In-memory data structure containing information about state of documents, whatever they were scheduled or not. Periodically synchronized with persistent storage.

message bus Transport layer abstraction mechanism. Provides interface for transport layer abstraction and several implementations.

spider A process retrieving and extracting content from the Web, using *spider feed* as incoming queue and storing results to *spider log*. In this documentation fetcher is used as synonym.

crawling strategy A class containing crawling logic covering seeds addition, processing of downloaded content and scheduling of new requests to crawl.

Architecture overview See how Frontera works and its different components.

Frontera API Learn how to use the frontier.

Using the Frontier with Requests Learn how to use Frontera with Requests.

Examples Some example projects and scripts using Frontera.

Tests How to run and write Frontera tests.

Logging A list of loggers for use with python native logging system.

Testing a Frontier Test your frontier in an easy way.

Contribution guidelines HOWTO contribute.

Glossary Glossary of terms.

Python Module Index

Index