# Frontera Documentation

***Release 0.4.0***

**ScrapingHub**

December 30, 2015

# Contents

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

Frontera provides *Crawl Frontier* framework by managing *when* and *what* to crawl next, and checking for *crawling goal* accomplishment.

Frontera also provides replication, sharding and isolation of all parts of Frontera-based crawler to scale and distribute it.

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 as the framework offers a generic tool box.

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.

### 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:

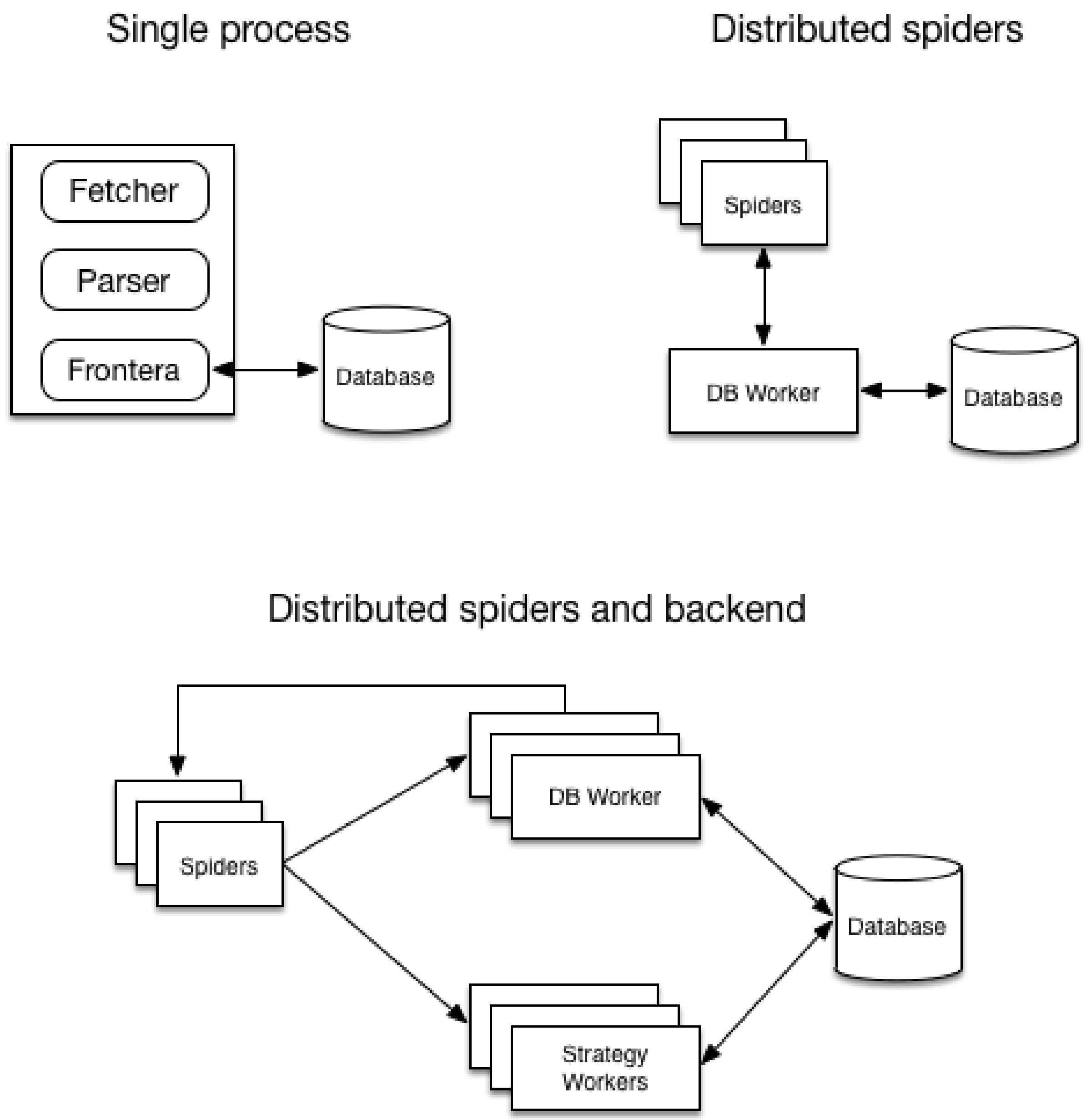
- Broad web crawling, arbitrary number of websites and pages (we tested it on 45M documents volume and 100K websites),
- Host-focused crawls: when you have more than 100 websites,
- Focused crawling:
  - Topical: you search for a pages about some predefined topic,
  - PageRank, HITS or other link graph 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 | Parent class | Components needed | Available backends |
|---|---|---|---|
| Single process | `Backend` | single process running the crawler | Memory, SQLAlchemy |
| Distributed spiders | `Backend` | spiders and single *db worker* | Memory, SQLAlchemy |
| Distributed backends | `DistributedBackend` | spiders, *strategy worker* (s) and db worker(s). | SQLAlchemy, HBase |

### 1.2.1 Single process

Frontera is instantiated in the same process as fetcher (for example in Scrapy). To achieve that use *BACKEND* setting set to storage backend subclass of `Backend`. This run mode is suitable for small number of documents and time non-critical applications.

### 1.2.2 Distributed spiders

Spiders are distributed and backend isn't. Backend is running in *db worker* and it's communicating with spiders using *message bus*.

1. Use *BACKEND* in spider processes set to `MessageBusBackend`
2. In DB worker *BACKEND* should point to `Backend` subclasse.
3. Every spider process should have it's own *SPIDER_PARTITION_ID*, starting from 0 to *SPIDER_FEED_PARTITIONS*.
4. Both spiders and workers should have it's *MESSAGE_BUS* setting set to the message bus class of your choice, and other implementation depending settings.

This mode is suitable for applications where it's critical to fetch documents fast, at the same time amount of them is relatively small.

### 1.2.3 Distributed spiders and backend

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 and SQLAlchemy and HBase distributed backends.

This mode is suitable for broad crawling and large amount of pages.

## 1.3 Quick start single process

### 1.3.1 1. Create your 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 2. Integrate your spider with the Frontera

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

### 1.3.4 3. Choose your backend

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

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

### 1.3.5 4. Run the spider

Run your Scrapy spider as usual from the command line:

```
scrapy crawl myspider
```

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

### 1.3.6 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 Production broad crawling 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+
- 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.spider0 -s SEEDS_SOURCE=seeds_es_smp.txt
$ scrapy crawl general -L INFO -s FRONTERA_SETTINGS=frontier.spider1
```

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.

**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.

CHAPTER 2

# Using Frontera

## 2.1 Installation Guide

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

- Python 2.7
- 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.

## 2.2 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).

### 2.2.1 Request objects

### 2.2.2 Response objects

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

### 2.2.3 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'
```

### 2.2.4 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"
}
```

## 2.3 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.

### 2.3.1 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.

### 2.3.2 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.

### 2.3.3 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

#### UrlFingerprintMiddleware

#### DomainFingerprintMiddleware

## 2.4 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.

### 2.4.1 Built-in canonical URL solvers reference

#### Basic

Used as default.

## 2.5 Backends

Frontier `Backend` is where the crawling logic/policies lies, essentially a brain of your crawler. `Queue`, `Metadata` and `States` are classes where all low level code is meant to be placed, and Backend opposite, operates on a higher levels. Frontera is bundled with database and in-memory implementations of Queue, Metadata and States which can be combined in your custom backends or used standalone by directly instantiating `FrontierManager` and Backend.

Backend 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 Backend can be used per frontier.

### 2.5.1 Activating a backend

To activate the frontier backend component, set it through the *BACKEND* setting.

Here's an example:

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

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

### 2.5.2 Writing your own backend

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

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

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

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

#### Metadata

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

#### Queue

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

### States

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

## 2.5.3 Built-in backend reference

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

To know the default activated `Backend` check the *`BACKEND`* setting.

### Basic algorithms

Some of the built-in `Backend` objects implement basic algorithms as as FIFO/LIFO or DFS/BFS for page visit ordering.

Differences between them will be on storage engine used. For instance, *`memory.FIFO`* and *`sqlalchemy.FIFO`* will use the same logic but with different storage engines.

All these backend variations are using the same `CommonBackend` class implementing one-time visit crawling policy with priority queue.

### Memory backends

This set of `Backend` objects will use an heapq module as queue and native dictionaries as storage for *basic algorithms*.

**class** `frontera.contrib.backends.memory.`**`BASE`**
Base class for in-memory `Backend` objects.

**class** `frontera.contrib.backends.memory.`**`FIFO`**
In-memory `Backend` implementation of FIFO algorithm.

**class** `frontera.contrib.backends.memory.`**`LIFO`**
In-memory `Backend` implementation of LIFO algorithm.

**class** `frontera.contrib.backends.memory.`**`BFS`**
In-memory `Backend` implementation of BFS algorithm.

**class** `frontera.contrib.backends.memory.`**`DFS`**
In-memory `Backend` implementation of DFS algorithm.

**class** `frontera.contrib.backends.memory.`**`RANDOM`**
In-memory `Backend` implementation of a random selection algorithm.

### SQLAlchemy backends

This set of `Backend` objects will use SQLAlchemy as storage for *basic algorithms*.

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.

This setting uses a dictionary where `key` represents the name of the model to define and `value` the model to use.

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

**class** `frontera.contrib.backends.sqlalchemy.`**`BASE`**
Base class for SQLAlchemy `Backend` objects.

**class** `frontera.contrib.backends.sqlalchemy.`**`FIFO`**
SQLAlchemy `Backend` implementation of FIFO algorithm.

**class** `frontera.contrib.backends.sqlalchemy.`**`LIFO`**
SQLAlchemy `Backend` implementation of LIFO algorithm.

**class** `frontera.contrib.backends.sqlalchemy.`**`BFS`**
SQLAlchemy `Backend` implementation of BFS algorithm.

**class** `frontera.contrib.backends.sqlalchemy.`**`DFS`**
SQLAlchemy `Backend` implementation of DFS algorithm.

**class** `frontera.contrib.backends.sqlalchemy.`**`RANDOM`**
SQLAlchemy `Backend` implementation of a random selection algorithm.

#### Revisiting backend

Based on custom SQLAlchemy backend, and queue. Crawling starts with seeds. After seeds are crawled, every new document will be scheduled for immediate crawling. On fetching every new document will be scheduled for recrawling after fixed interval set by *`SQLALCHEMYBACKEND_REVISIT_INTERVAL`*.

Current implementation of revisiting backend has no prioritization. During long term runs spider could go idle, because there are no documents available for crawling, but there are documents waiting for their scheduled revisit time.

#### 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.

## 2.6 Message bus

Is the transport layer abstraction mechanism. It provides interface and several implementations. Only one message bus can be used in crawler at the time, and it's selected with *`MESSAGE_BUS`* vim maksetting.

Spiders process can use

to communicate using message bus.

### 2.6.1 Built-in message bus reference

#### ZeroMQ

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

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

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

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.

### 2.6.2 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 (also JSON is available, but needs to be selected in code manually).

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

## 2.7 Crawling strategy

Use `frontera.worker.strategies.bfs` module for reference. In general, you need to write a `CrawlingStrategy` class implementing the interface:

The class named `CrawlingStrategy` should put in a standalone module and passed to *strategy worker* using command line option on startup.

The strategy class instantiated in strategy worker, and can use it's 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.

## 2.8 Using the Frontier with Scrapy

Using Frontera is quite easy, it includes a set of Scrapy middlewares and Scrapy scheduler that encapsulates Frontera usage and can be easily configured using Scrapy settings.

### 2.8.1 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'
```

## 2.8.2 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
            middlewares.py
            backends.py
        spiders/
            ...
        __init__.py
        settings.py
     scrapy.cfg
```

These are basically:

- `my_scrapy_project/frontera/settings.py`: the Frontera settings file.
- `my_scrapy_project/frontera/middlewares.py`: the middlewares used by the Frontera.
- `my_scrapy_project/frontera/backends.py`: the backend(s) used by the Frontera.
- `my_scrapy_project/spiders`: the Scrapy spiders folder
- `my_scrapy_project/settings.py`: the Scrapy settings file
- `scrapy.cfg`: the Scrapy config file

## 2.8.3 Running the rawl

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

```
scrapy crawl myspider
```

## 2.8.4 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

## 2.8.5 Writing Scrapy spider

### Spider logic

Creation of basic Scrapy spider is described at Frontier at a glance 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.

Adding one of seed loaders for bootstrapping of crawling process:

```
SPIDER_MIDDLEWARES.update({
    'frontera.contrib.scrapy.middlewares.seeds.file.FileSeedLoader': 1,
})
```

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
                                    # with response latencies.
RANDOMIZE_DOWNLOAD_DELAY = False

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

Check also Scrapy broad crawling recommendations.

### 2.8.6 Scrapy Seed Loaders

Frontera has some built-in Scrapy middlewares for seed loading.

Seed loaders use the `process_start_requests` method to generate requests from a source that are added later to the `FrontierManager`.

#### Activating a Seed loader

Just add the Seed Loader middleware to the `SPIDER_MIDDLEWARES` scrapy settings:

```
SPIDER_MIDDLEWARES.update({
    'frontera.contrib.scrapy.middlewares.seeds.FileSeedLoader': 650
})
```

#### FileSeedLoader

Load seed URLs from a file. The file must be formatted contain one URL per line:

```
http://www.asite.com
http://www.anothersite.com
...
```

Yo can disable URLs using the `#` character:

```
...
#http://www.acommentedsite.com
...
```

**Settings**:

- `SEEDS_SOURCE`: Path to the seeds file

#### S3SeedLoader

Load seeds from a file stored in an Amazon S3 bucket

File format should the same one used in *FileSeedLoader*.

Settings:

- `SEEDS_SOURCE`: Path to S3 bucket file. eg: `s3://some-project/seed-urls/`
- `SEEDS_AWS_ACCESS_KEY`: S3 credentials Access Key
- `SEEDS_AWS_SECRET_ACCESS_KEY`: S3 credentials Secret Access Key

## 2.9 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*.

### 2.9.1 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.

### 2.9.2 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.

### 2.9.3 Settings class

### 2.9.4 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*.

### 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*.

### CONSUMER_BATCH_SIZE

Default: `512`

This is a batch size used by strategy and db workers for consuming of spider log and scoring log streams. 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 withing the same time interval, but with less overall efficiency. Use it when your consumers too slow, or too fast.

### 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.

### EVENT_LOGGER

Default: `'frontera.logger.events.EventLogManager'`

The EventLoggerManager class to be used by the Frontier.

### KAFKA_GET_TIMEOUT

Default: `5.0`

Time process should block until requested amount of data will be received from message bus.

### LOGGER

Default: `'frontera.logger.FrontierLogger'`

The Logger class to be used by the Frontier.

### MAX_NEXT_REQUESTS

Default: `64`

The maximum number of requests returned by `get_next_requests` API method. If value is 0 (default), no maximum value will be used.

### 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.

### 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_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.

### SCORING_PARTITION_ID

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: `2`

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

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.

#### TEST_MODE

Default: `False`

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

### 2.9.5 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*).

### 2.9.6 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.

## 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_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_CACHE_SIZE_LIMIT

Default: `3000000`

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

#### HBASE_THRIFT_HOST

Default: `localhost`

HBase Thrift server host.

#### HBASE_THRIFT_PORT

Default: `9090`

HBase Thrift server port

#### HBASE_USE_COMPACT_PROTOCOL

Default: `False`

Whatever workers should use Thrift compact protocol. Dramatically reduces transmission overhead, but needs to be turned on on server too.

#### 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.

### 2.9.7 ZeroMQ message bus settings

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

#### ZMQ_HOSTNAME

Default: `127.0.0.1`

Hostname, where ZeroMQ socket should bind or connect.

#### 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.

### 2.9.8 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(,).

#### FRONTIER_GROUP

Default: `general`

Kafka consumer group name, used for almost everything.

#### INCOMING_TOPIC

Default: `frontier-done`

Spider log stream topic name.

#### OUTGOING_TOPIC

Default: `frontier-todo`

Spider feed stream topic name.

#### SCORING_GROUP

Default: `strategy-workers`

A group used by strategy workers for spider log reading. Needs to be different than `FRONTIER_GROUP`.

#### SCORING_TOPIC

Kafka topic used for *scoring log* stream.

## 2.9.9 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.

### Logging default settings

Values:

```
LOGGER = 'frontera.logger.FrontierLogger'
LOGGING_ENABLED = True

LOGGING_EVENTS_ENABLED = False
LOGGING_EVENTS_INCLUDE_METADATA = True
LOGGING_EVENTS_INCLUDE_DOMAIN = True
LOGGING_EVENTS_INCLUDE_DOMAIN_FIELDS = ['name', 'netloc', 'scheme', 'sld', 'tld', 'subdomain']
LOGGING_EVENTS_HANDLERS = [
    "frontera.logger.handlers.COLOR_EVENTS",
]
```

```
LOGGING_MANAGER_ENABLED = False
LOGGING_MANAGER_LOGLEVEL = logging.DEBUG
LOGGING_MANAGER_HANDLERS = [
    "frontera.logger.handlers.COLOR_CONSOLE_MANAGER",
]

LOGGING_BACKEND_ENABLED = False
LOGGING_BACKEND_LOGLEVEL = logging.DEBUG
LOGGING_BACKEND_HANDLERS = [
    "frontera.logger.handlers.COLOR_CONSOLE_BACKEND",
]

LOGGING_DEBUGGING_ENABLED = False
LOGGING_DEBUGGING_LOGLEVEL = logging.DEBUG
LOGGING_DEBUGGING_HANDLERS = [
    "frontera.logger.handlers.COLOR_CONSOLE_DEBUGGING",
]

EVENT_LOG_MANAGER = 'frontera.logger.events.EventLogManager'
```

**Installation Guide** HOWTO and Dependencies options.

**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** Define your own crawling policy and custom storage.

**Message bus** Built-in message bus reference.

**Crawling strategy** Implementing own crawling strategy for distributed backend.

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

**Settings** Settings reference.

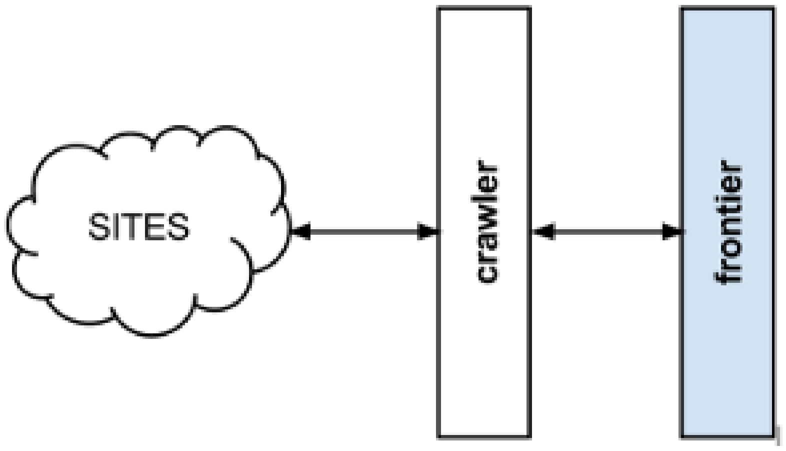
CHAPTER 3

# 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.

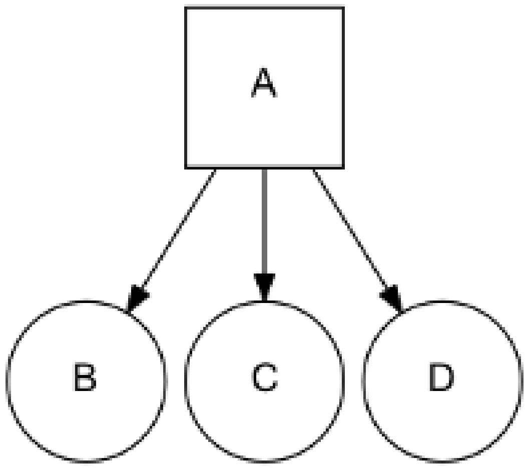
It can easily be used to test frontiers. We can "fake" crawler request/responses by querying pages to the graph manager, and also know the links extracted for each one without using a crawler at all. You can make your own fake tests or use the Frontier Tester tool.

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:



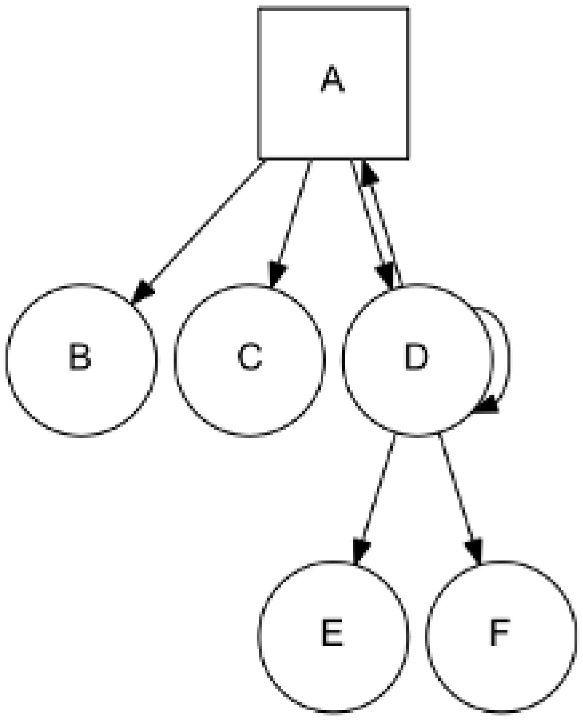
We use a list to represent the different site pages and one tuple to define the page and its links, for the previous example:

```
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:

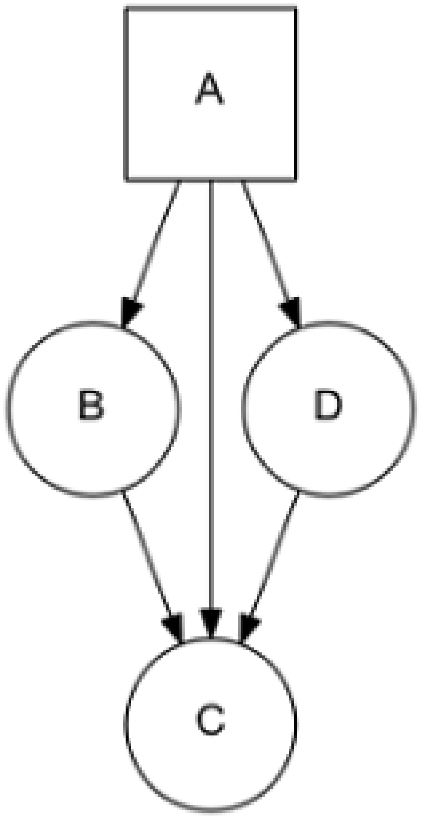


Can be represented as:

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

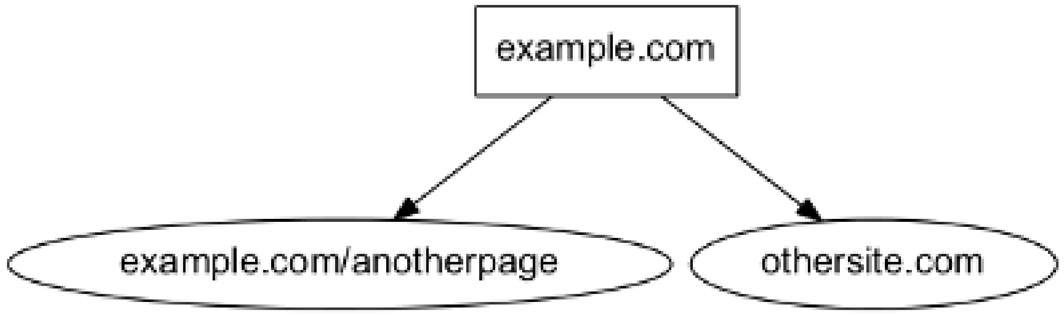
Note that *D* is linking to itself and to his parent *A*.

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 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)
>>> 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, graphs, Request, Response

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])
            # 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 Production broad crawling

These are the topics you need to consider when deploying Frontera-based broad crawler in production system.

### 3.4.1 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. We recommend to setup a dedicated DNS instance locally on every spider machine with upstream using massive DNS caches like OpenDNS or Verizon.

### 3.4.2 Choose message bus

There are two options available:

- Kafka, requires properly configured partitions,
- ZeroMQ (default), requires broker process.

#### Configuring Kafka

The main thing to do here is to set the number of partitions for *`OUTGOING_TOPIC`* equal to the number of spider instances and for *`INCOMING_TOPIC`* equal to number of strategy worker instances. For other topics it makes sense to set more than one partition to better distribute the load across Kafka cluster.

Kafka throughput is key performance issue, make sure that Kafka brokers have enough IOPS, and monitor the network load.

#### Configuring ZeroMQ

ZeroMQ requires almost no configuration except hostname and base port where to bind it's sockets. Please see *ZMQ_HOSTNAME* and *ZMQ_BASE_PORT* settings for more detail. ZeroMQ also requires distributed frontera broker process running and accessible to connect. See *Start cluster*.

### 3.4.3 Configure Frontera workers

There are two type of workers: DB and Strategy.

*DB worker* is doing three tasks in particular:

- Reading *spider log* stream and updates metadata in DB,
- Consult lags in message bus, gets new batches and pushes them to *spider feed*,
- Reads *scoring log* stream and updates DB with new score and schedule URLs to download if needed.

Strategy worker is reading *spider log*, calculating score, deciding if URL needs to be crawled and pushes update_score events to *scoring log*.

Before setting it up you have to decide how many spider instances you need. One spider is able to download and parse about 700 pages/minute in average. Therefore if you want to fetch 1K per second you probably need about 10 spiders. For each 4 spiders you would need one pair of workers (strategy and DB). If your strategy worker is lightweight (not processing content for example) then 1 strategy worker per 15 spider instances could be enough.

Your spider log stream should have as much partitions as *strategy workers* you need. Each strategy worker is assigned to specific partition using option *SCORING_PARTITION_ID*.

Your spider feed stream, containing new batches should have as much partitions as *spiders* you will have in your cluster.

Now, let's create a Frontera workers settings file under `frontera` subfolder and name it `worker_settings.py`.

```
from frontera.settings.default_settings import MIDDLEWARES

MAX_NEXT_REQUESTS = 128     # Size of batch to generate per partition, should be consistent with
                            # CONCURRENT_REQUESTS in spider. General recommendation is 5-7x CONCURREN

#--------------------------------------------------------
# Url storage
#--------------------------------------------------------
BACKEND = 'distributed_frontera.contrib.backends.hbase.HBaseBackend'
HBASE_DROP_ALL_TABLES = False
HBASE_THRIFT_PORT = 9090
HBASE_THRIFT_HOST = 'localhost'

MIDDLEWARES.extend([
    'frontera.contrib.middlewares.domain.DomainMiddleware',
    'frontera.contrib.middlewares.fingerprint.DomainFingerprintMiddleware'
])

#--------------------------------------------------------
# Logging
#--------------------------------------------------------
LOGGING_EVENTS_ENABLED = False
LOGGING_MANAGER_ENABLED = True
LOGGING_BACKEND_ENABLED = True
LOGGING_DEBUGGING_ENABLED = False
```

You should add there settings related to message bus you have chosen. Default is ZeroMQ, running on local host.

### 3.4.4 Configure Frontera spiders

Next step is to create own Frontera settings file for every spider instance. Often it's a good idea to name settings file according to partition ids assigned. E.g. `settingsN.py`.

```
from distributed_frontera.settings.default_settings import MIDDLEWARES

MAX_NEXT_REQUESTS = 256     # Should be consistent with MAX_NEXT_REQUESTS set for Frontera worker

MIDDLEWARES.extend([
    'frontera.contrib.middlewares.domain.DomainMiddleware',
    'frontera.contrib.middlewares.fingerprint.DomainFingerprintMiddleware'
])

#--------------------------------------------------------
# Crawl frontier backend
#--------------------------------------------------------
BACKEND = 'distributed_frontera.backends.remote.KafkaOverusedBackend'
SPIDER_PARTITION_ID = 0                 # Partition ID assigned

#--------------------------------------------------------
# Logging
#--------------------------------------------------------
LOGGING_ENABLED = True
LOGGING_EVENTS_ENABLED = False
LOGGING_MANAGER_ENABLED = False
LOGGING_BACKEND_ENABLED = False
LOGGING_DEBUGGING_ENABLED = False
```

Again, add message bus related options.

You should end up having as much settings files as spider instances your system will have. You can also store permanent options in common module, and import it's contents from each instance-specific config file.

It is recommended to run spiders on a dedicated machines, they quite likely to consume lots of CPU and network bandwidth.

The same thing have to be done for strategy workers, each strategy worker should have it's own partition id (see *SCORING_PARTITION_ID*) assigned in config files named `strategyN.py`.

### 3.4.5 Starting the cluster

First, let's start storage worker. It's recommended to dedicate one worker instance for new batches generation and others for the rest. Batch generation instance isn't much dependent on the count of spider instances, but saving to storage is. Here is how to run all in the same process:

```
# start DB worker, enabling batch generation, DB saving and scoring log consumption
$ python -m frontera.worker.db --config frontera.worker_settings
```

Next, let's start strategy worker with sample strategy for crawling the internet in Breadth-first manner.:

```
$ python -m frontera.worker.strategy --config frontera.strategy0 --strategy frontera.worker.strategie
$ python -m frontera.worker.strategy --config frontera.strategy1 --strategy frontera.worker.strategie
...
$ python -m frontera.worker.strategy --config frontera.strategyN --strategy frontera.worker.strategie
```

You should notice that all processes are writing messages to the output. 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. Starting the spiders::

```
$ scrapy crawl tutorial -L INFO -s FRONTERA_SETTINGS=frontera.settings0 -s SEEDS_SOURCE = 'seeds.txt
...
$ scrapy crawl tutorial -L INFO -s FRONTERA_SETTINGS=frontera.settings1
$ scrapy crawl tutorial -L INFO -s FRONTERA_SETTINGS=frontera.settings2
$ scrapy crawl tutorial -L INFO -s FRONTERA_SETTINGS=frontera.settings3
...
$ scrapy crawl tutorial -L INFO -s FRONTERA_SETTINGS=frontera.settingsN
```

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

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

**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.

**Production broad crawling** Deployment and tuning information.

CHAPTER 4

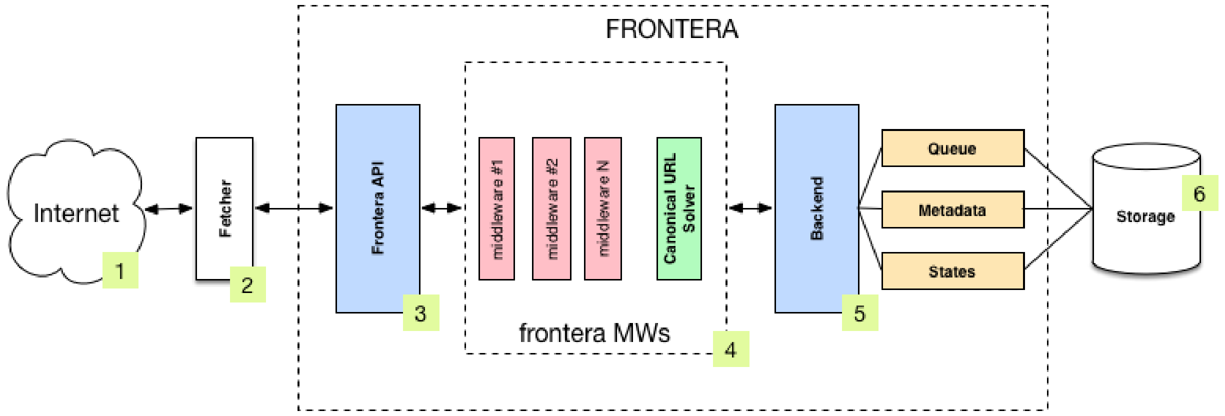
# Developer documentation

## 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:

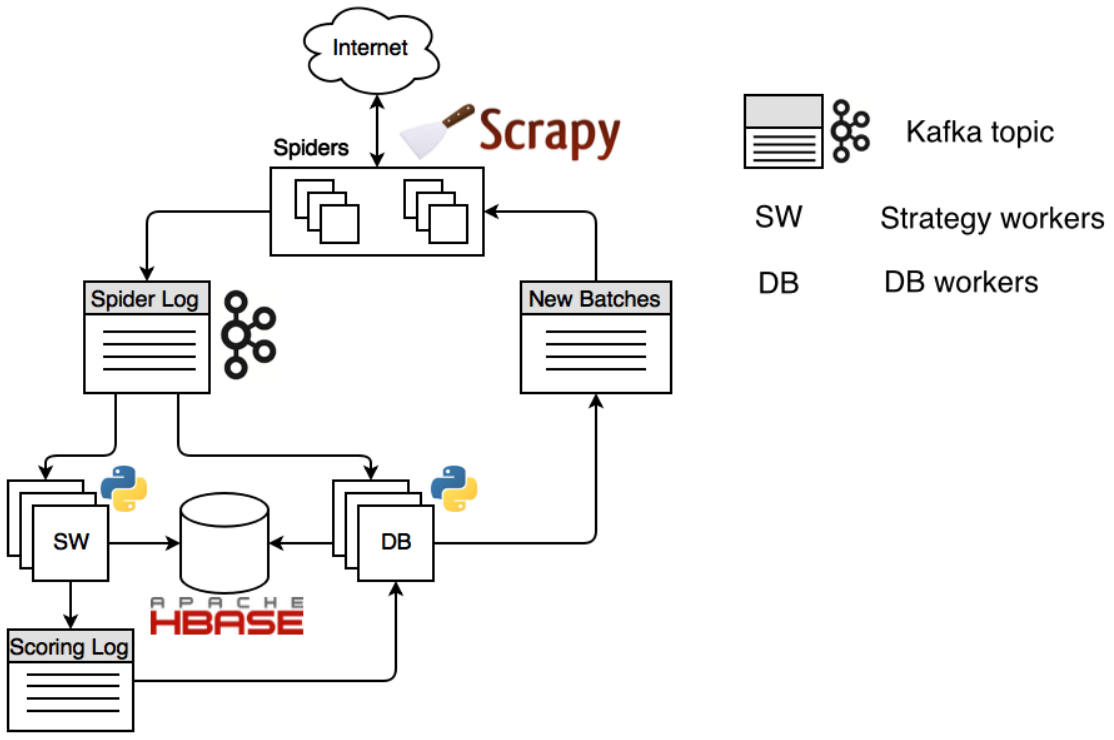
- **Spiders or fetchers, implemented using Scrapy (sharded).** Responsible for resolving DNS queries, getting content from the Internet and doing link (or other data) extraction from content.
- **Strategy workers (sharded).** Run the crawling strategy code: scoring the links, deciding if link needs to be scheduled and when to stop crawling.

- **DB workers (replicated).** 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

### 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 Another 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

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

```python
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, 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 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
```

### 4.4.4 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:

```
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

    ...
```

### 4.5.5 Testing basic algorithms

If your backend uses any of the *basic algorithms logics*, you can just inherit the correponding test base class for each logic and sequences will be automatically tested for it:

```
from frontera.tests import backends


class TestMyBackendFIFO(backends.FIFOBackendTest):
    backend_class = 'frontera.contrib.backends.abackend.MyBackendFIFO'


class TestMyBackendLIFO(backends.LIFOBackendTest):
    backend_class = 'frontera.contrib.backends.abackend.MyBackendLIFO'


class TestMyBackendDFS(backends.DFSBackendTest):
    backend_class = 'frontera.contrib.backends.abackend.MyBackendDFS'


class TestMyBackendBFS(backends.BFSBackendTest):
    backend_class = 'frontera.contrib.backends.abackend.MyBackendBFS'


class TestMyBackendRANDOM(backends.RANDOMBackendTest):
    backend_class = 'frontera.contrib.backends.abackend.MyBackendRANDOM'
```

## 4.6 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.

### 4.6.1 Creating a Frontier Tester

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

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

### 4.6.2 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.

### 4.6.3 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)
```

### 4.6.4 An example of use

A working example using test data from graphs and *basic 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')
```

## 4.7 F.A.Q.

### 4.7.1 How to download efficiently in parallel?

Typically the design of URL ordering implies fetching many URLs from the same domain. If crawling process needs to be polite it has to preserve some delay and rate of requests. From the other side, there are downloaders which can afford downloading many URLs (say 100) at once, in parallel. So, flooding of the URLs from the same domain leads to inefficient waste of downloader connection pool resources.

Here is a short example. Imagine, we have a queue of 10K URLs from many different domains. Our task is to fetch it as fast as possible. During downloading we want to be polite and limit per host RPS. At the same time we have a prioritization which tends to group URLs from the same domain. When crawler will be requesting for batches of URLs to fetch, it will be getting hundreds of URLs from the same host. The downloader will not be able to fetch them quickly because of RPS limit and delay. Therefore, picking top URLs from the queue leeds us to the time waste, because connection pool of downloader most of the time underused.

The solution is to supply Frontera backend with hostname/ip (usually, but not necessary) usage in downloader. We have a keyword arguments in method `get_next_requests` for passing these stats, to the Frontera backend. Information of any kind can be passed there. This arguments are usually set outside of Frontera, and then passed to CF via `FrontierManagerWrapper` subclass to backend.

## 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.