Frontera Documentation

Release 0.3.0

ScrapingHub

April 15, 2015

Contents

This documentation contains everything you need to know about Frontera.

CHAPTER 1

First steps

1.1 Frontera at a glance

Frontera is an application framework that is meant to be used as part of a Crawling System, allowing you to easily manage and define tasks related to a *crawl frontier*.

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 frontier functionality.

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

1.1.1 1. Create your crawler

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.

1.1.2 2. Integrate your crawler with the frontier

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

1.1.3 3. Choose your backend

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

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

1.1.4 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.

1.1.5 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:

- Easy *built-in integration with Scrapy* and *any other crawler* through its API.
- Creating different crawling logic/policies *defining your own backend*.
- Built-in support for *database storage* for crawled pages.
- Support for extending Frontera by plugging your own functionality using *middlewares*.
- Built-in middlewares for:
  - Extracting *domain info* from page URLs.
  - Create *unique fingerprints for page URLs* and *domain names*.
- Create fake sitemaps and reproduce crawling without crawler with the *graph Manager*.
- Tools for *easy frontier testing*.
- *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.

1.1.6 What's next?

The next obvious steps are for you to *install Frontera*, and read the *architecture overview* and *API docs*. Thanks for your interest!

1.2 Installation Guide

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

- Python 2.7

- pip and setuptools Python packages. Nowadays pip requires and installs setuptools if not installed.

You can install Frontera using pip (which is the canonical way to install Python packages).

To install using pip:

```
pip install frontera
```

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

Installation Guide Get Frontera installed on your computer.

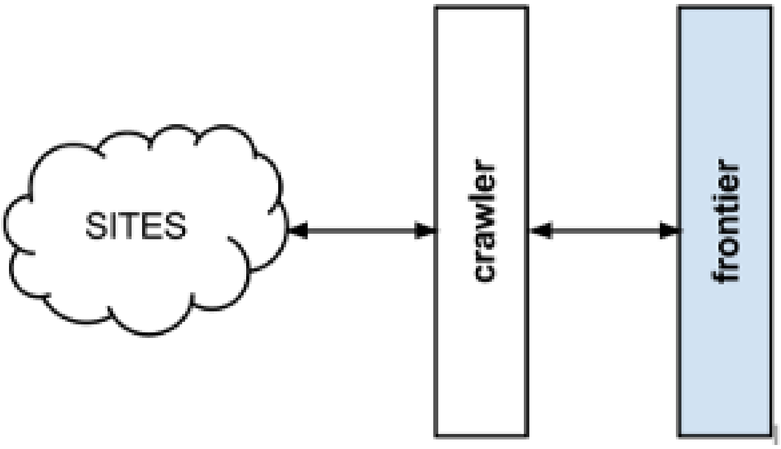
CHAPTER 2

Basic concepts

2.1 What is a Frontera?

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

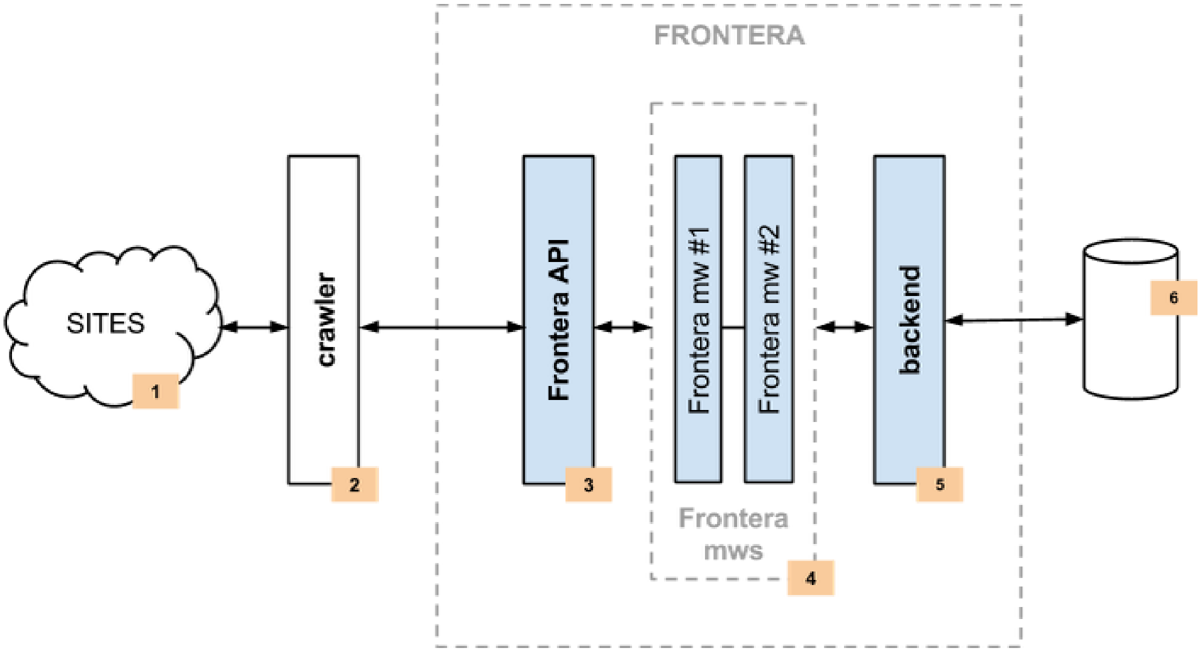
Depending on frontier logic, a persistent storage system may be needed to store, update or query information about the pages. Other systems can be 100% volatile and not share any information at all between different crawls.

2.2 Architecture overview

This document describes the architecture of Frontera and how its components interact.

2.2.1 Overview

The following diagram shows an overview of the Frontera architecture 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.



2.2.2 Components

Crawler

The Crawler (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.

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

Frontera API / Manager

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

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 in future versions.

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.

For more information see *Middlewares*.

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.

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

For more information see *Backends*.

2.2.3 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 crawler asks for a list of requests to crawl.
3. Each url is crawled and the frontier is notified back of the crawl result as well of the extracted links 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*.

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*)

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.

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.

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** (*dict*) – the response body.
- **request** (*dict*) – 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 Frontera API

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

2.4.1 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.

2.4.2 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 Settings 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()
```

2.4.3 Frontier Manager

class frontera.core.manager.**FrontierManager**(*request_model*, *response_model*, *backend*, *logger*, *event_log_manager*, *middlewares=None*, *test_mode=False*, *max_requests=0*, *max_next_requests=0*, *auto_start=True*, *settings=None*)

The FrontierManager object encapsulates the whole frontier, providing an API to interact with. It's also responsible of loading and communicating all different frontier components.

Parameters

- **request_model** (*object/string*) – The Request object to be used by the frontier.
- **response_model** (*object/string*) – The Response object to be used by the frontier.
- **backend** (*object/string*) – The Backend object to be used by the frontier.
- **logger** (*object/string*) – The Logger object to be used by the frontier.
- **event_log_manager** (*object/string*) – The EventLogger object to be used by the frontier.
- **middlewares** (*list*) – A list of Middleware objects to be used by the frontier.
- **test_mode** (*bool*) – Activate/deactivate *frontier test mode*.
- **max_requests** (*int*) – Number of pages after which the frontier would stop (See *Finish conditions*).
- **max_next_requests** (*int*) – Maximum number of requests returned by get_next_requests method.
- **auto_start** (*bool*) – Activate/deactivate automatic frontier start (See *starting/stopping the frontier*).
- **settings** (*object/string*) – The Settings object used by the frontier.

Attributes

request_model
The Request object to be used by the frontier. Can be defined with REQUEST_MODEL setting.

response_model
The Response object to be used by the frontier. Can be defined with RESPONSE_MODEL setting.

backend
The Backend object to be used by the frontier. Can be defined with BACKEND setting.

logger
The Logger object to be used by the frontier. Can be defined with LOGGER setting.

event_log_manager
The EventLogger object to be used by the frontier. Can be defined with EVENT_LOGGER setting.

middlewares
A list of Middleware objects to be used by the frontier. Can be defined with MIDDLEWARES setting.

test_mode
Boolean value indicating if the frontier is using *frontier test mode*. Can be defined with TEST_MODE setting.

max_requests
Number of pages after which the frontier would stop (See *Finish conditions*). Can be defined with MAX_REQUESTS setting.

max_next_requests
Maximum number of requests returned by get_next_requests method. Can be defined with MAX_NEXT_REQUESTS setting.

auto_start
Boolean value indicating if automatic frontier start is activated. See *starting/stopping the frontier*. Can be defined with AUTO_START setting.

settings
The Settings object used by the frontier.

iteration
Current *frontier iteration*.

n_requests
Number of accumulated requests returned by the frontier.

finished
Boolean value indicating if the frontier has finished. See *Finish conditions*.

API Methods

start()
Notifies all the components of the frontier start. Typically used for initializations (See *starting/stopping the frontier*).

Returns None.

stop()
Notifies all the components of the frontier stop. Typically used for finalizations (See *starting/stopping the frontier*).

Returns None.

add_seeds(*seeds*)
Adds a list of seed requests (seed URLs) as entry point for the crawl.

Parameters **seeds** (*list*) – A list of Request objects.

Returns None.

get_next_requests(*max_next_requests=0*, ***kwargs*)

Returns a list of next requests to be crawled. Optionally a maximum number of pages can be passed. If no value is passed, `FrontierManager.max_next_requests` will be used instead. (`MAX_NEXT_REQUESTS` setting).

Parameters

- **max_next_requests** (*int*) – Maximum number of requests to be returned by this method.
- **kwargs** (*dict*) – Arbitrary arguments that will be passed to backend.

Returns list of `Request` objects.

page_crawled(*response*, *links=None*)

Informs the frontier about the crawl result and extracted links for the current page.

Parameters

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

Returns None.

request_error(*request*, *error*)

Informs the frontier about a page crawl error. An error identifier must be provided.

Parameters

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

Returns None.

Class Methods

classmethod from_settings(*settings=None*)

Returns a `FrontierManager` instance initialized with the passed settings argument. Argument value can either be a string path pointing to settings file or a `Settings` object instance. If no settings is given, *frontier default settings* are used.

2.4.4 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.

2.4.5 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.

2.4.6 Finishing the frontier

Crawl can be finished either by the Crawler or by the Frontera. Frontera will finish when a maximum number of pages are 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.

2.4.7 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*.

`add_seeds``(`*seeds*`)`

This method is called when new seeds are are added to the frontier.

Parameters **seeds** (*list*) – A list of `Request` objects.

`page_crawled``(`*response*, *links*`)`

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

Parameters

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

2.4.8 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.

2.4.9 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.

2.5 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*.

2.5.1 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.

2.5.2 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.

2.5.3 Settings class

class `frontera.settings.`**`Settings`**(*module=None*, *attributes=None*)
An object that holds frontier settings values.

Parameters

- **module** (*object/string*) – A `Settings` object or a path string.
- **attributes** (*dict*) – A dict object containing the settings values.

2.5.4 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*.

EVENT_LOGGER

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

The EventLoggerManager class to be used by the Frontier.

LOGGER

Default: `'frontera.logger.FrontierLogger'`

The Logger class to be used by the Frontier.

MAX_NEXT_REQUESTS

Default: `0`

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*.

MIDDLEWARES

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

Default:

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

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.

TEST_MODE

Default: `False`

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

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."

DELAY_ON_EMPTY

Default: `30.0`

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 a lot of time, and decrease if you need a fast spider bootstrap from seeds.

2.5.5 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.

2.5.6 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.

Frontier default settings

Values:

```
PAGE_MODEL = 'frontera.core.models.Page'
LINK_MODEL = 'frontera.core.models.Link'
FRONTIER = 'frontera.core.frontier.Frontier'
MIDDLEWARES = [
    'frontera.contrib.middlewares.fingerprint.UrlFingerprintMiddleware',
]
BACKEND = 'frontera.contrib.backends.memory.FIFO'
TEST_MODE = False
MAX_PAGES = 0
MAX_NEXT_PAGES = 0
AUTO_START = True
```

Fingerprints middleware default settings

Values:

```
URL_FINGERPRINT_FUNCTION = 'frontera.utils.fingerprint.sha1'
DOMAIN_FINGERPRINT_FUNCTION = 'frontera.utils.fingerprint.sha1'
```

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'
```

What is a Frontera? Learn what Frontera is and how to use it.

Architecture overview See how Frontera works and its different components.

Frontier objects Understand the classes used to represent links and pages.

Frontera API Learn how to use the frontier.

Settings See how to configure Frontera.

CHAPTER 3

Extending Frontera

3.1 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.

3.1.1 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.

3.1.2 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*.

add_seeds(*seeds*)

This method is called when new seeds are are added to the frontier.

Parameters **seeds** (*list*) – A list of `Request` objects.

Returns `Request` object list or `None`

Should either return `None` or a list of `Request` objects.

If it returns `None`, `FrontierManager` won't continue processing any other middleware and seed will never reach the `Backend`.

If it returns a list of `Request` objects, 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 any seed, just don't include it in the returned object list.

page_crawled(*response*, *links*)

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

Parameters

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

3.1.3 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 contains the following fields:

- **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.

An example for a Request object:

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

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

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.

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"
}
```

3.2 Backends

Frontier Backend is where the crawling logic/policies lies. It’s responsible for receiving all the crawl info and selecting the next pages to be crawled. It’s 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.

Some backends require, depending on the logic implemented, a persistent storage to manage `Request` and `Response` objects info.

3.2.1 Activating a backend

To activate the frontier middleware 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 enabled through a particular setting. See *each backend documentation* for more info.

3.2.2 Writing your own backend

Writing your own frontier backend is easy. Each `Backend` component is a single Python class inherited from `Component`.

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

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

Interface definition for a 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.

`add_seeds``(`*seeds*`)`

This method is called when new seeds are are added to the frontier.

Parameters **seeds** (*list*) – A list of `Request` objects.

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.

`page_crawled``(`*response*, *links*`)`

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

Parameters

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

- **links** (*list*) – A list of Request objects generated from the links extracted 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.

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)
```

3.2.3 Built-in backend reference

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

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.

Memory backends

This set of Backend objects will use an heapq object as storage for *basic algorithms*.

class frontera.contrib.backends.memory.**BASE**

Base class for in-memory heapq Backend objects.

class frontera.contrib.backends.memory.**FIFO**

In-memory heapq Backend implementation of FIFO algorithm.

class frontera.contrib.backends.memory.**LIFO**

In-memory heapq Backend implementation of LIFO algorithm.

class frontera.contrib.backends.memory.**BFS**

In-memory heapq Backend implementation of BFS algorithm.

class frontera.contrib.backends.memory.**DFS**

In-memory heapq Backend implementation of DFS algorithm.

class `frontera.contrib.backends.memory.`**`RANDOM`**

In-memory heapq `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.

`Request` and `Response` are represented by a declarative sqlalchemy model:

```
class Page(Base):
    __tablename__ = 'pages'
    __table_args__ = (
        UniqueConstraint('url'),
    )
    class State:
        NOT_CRAWLED = 'NOT CRAWLED'
        QUEUED = 'QUEUED'
        CRAWLED = 'CRAWLED'
        ERROR = 'ERROR'

    url = Column(String(1000), nullable=False)
    fingerprint = Column(String(40), primary_key=True, nullable=False, index=True, unique=True)
    depth = Column(Integer, nullable=False)
    created_at = Column(TIMESTAMP, nullable=False)
    status_code = Column(String(20))
    state = Column(String(10))
    error = Column(String(20))
```

If you need to create your own models, you can do it by using the `DEFAULT_MODELS` setting:

```
DEFAULT_MODELS = {
    'Page': 'frontera.contrib.backends.sqlalchemy.models.Page',
}
```

This setting uses a dictionary where `key` represents the name of the model to define and `value` the model to use. If you want for instance to create a model to represent domains:

```
DEFAULT_MODELS = {
    'Page': 'frontera.contrib.backends.sqlalchemy.models.Page',
    'Domain': 'myproject.backends.sqlalchemy.models.Domain',
}
```

Models can be accessed from the Backend dictionary attribute `models`.

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.

Middlewares Filter or alter information for links and pages.

Backends Define your own crawling logic.

CHAPTER 4

Built-in services and tools

4.1 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.

4.1.1 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.py’
```

4.1.2 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

4.1.3 Running the Crawl

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

```
scrapy crawl myspider
```

4.1.4 Frontier Scrapy settings

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

FRONTERA_SETTINGS

Default: `None`

A file path pointing to Frontera settings.

4.2 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',
]

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)
```

4.3 Graph Manager

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

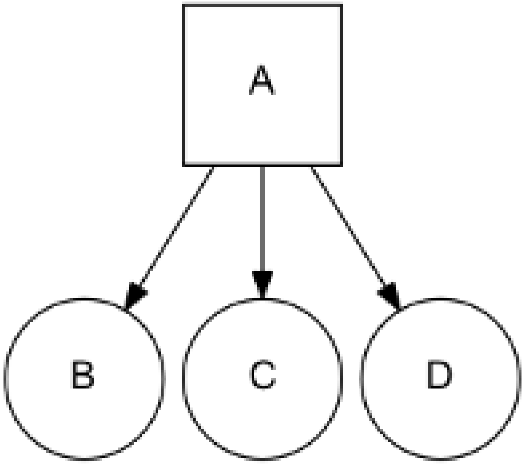
It can easily be used to test frontiers. We can "fake" crawler request/responses by querying pages to the graph manager, and also know the links extracted for each one without using a crawler at all. You can make your own fake tests or use the *Frontier Tester tool*.

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

4.3.1 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:



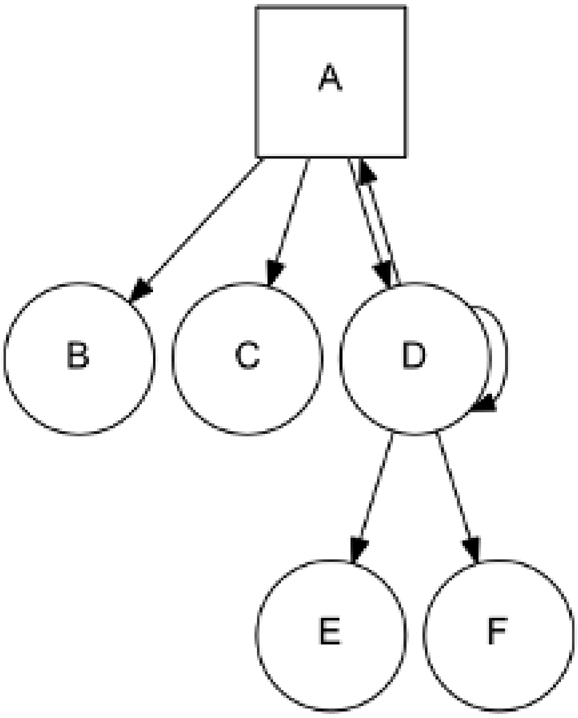
We use a list to represent the different site pages and one tuple to define the page and its links, for the previous example:

```
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:

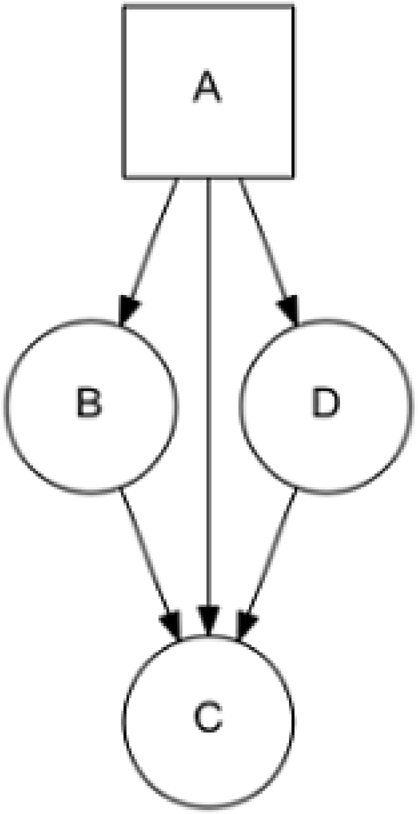


Can be represented as:

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

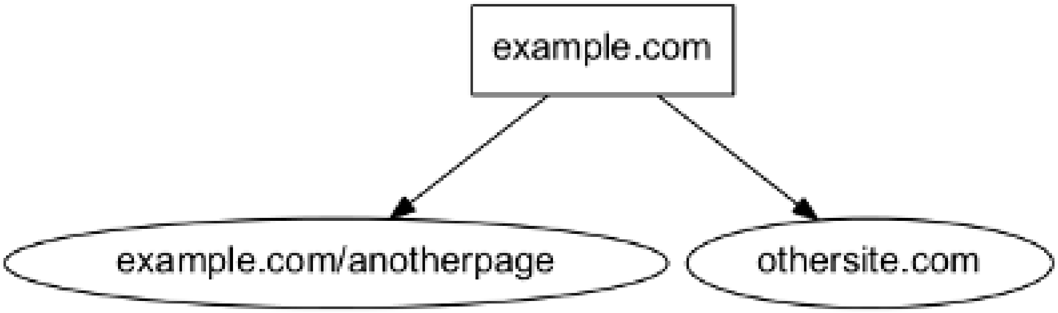
Note that *D* is linking to itself and to his parent *A*.

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']),
]
```

4.3.2 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
```

4.3.3 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*>]
```

4.3.4 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')
```

4.3.5 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)
```

4.3.6 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)
```

4.3.7 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.

4.3.8 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()
```

4.3.9 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.

4.3.10 How to use it

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

4.4 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.

4.4.1 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)
```

4.4.2 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.

4.4.3 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)
```

4.4.4 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')
```

4.5 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.

4.5.1 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,
})
```

4.5.2 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
```

4.5.3 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
```

4.5.4 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.

4.6 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`.

4.6.1 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
})
```

4.6.2 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

4.6.3 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

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

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

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

Testing a Frontier Test your frontier in an easy way.

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

Scrapy Seed Loaders Scrapy middlewares for seed loading

CHAPTER 5

All the rest

5.1 Examples

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

```
examples/
    requests/
    scrapy_frontier/
    scrapy_recording/
    scripts/
```

- **requests**: Example script with Requests library.
- **scrapy_frontier**: Scrapy Frontier 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
```

5.1.1 requests

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

How to run it:

```
python links_follower.py
```

5.1.2 scrapy_frontier

A simple script with a spider that follows all the links for the sites defined in a `seeds.txt` file.

How to run it:

```
scrapy crawl example
```

5.1.3 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
```

5.1.4 scripts

Some sample scripts on how to use different frontier components.

5.2 Best practices

5.2.1 Efficient parallel downloading

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.

5.3 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
```

5.3.1 Running tests

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

```
py.test
```

5.3.2 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.

5.3.3 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()
        ...

    ...
```

5.3.4 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

    ...
```

5.3.5 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'
```

5.4 Release Notes

5.4.1 0.2.0 (released 2015-01-12)

- Added documentation (Scrapy Seed Loaders+Tests+Examples) (8e5f60d)
- Refactored backend tests (00910bf, 5702bef, 9567566)
- Added requests library example (8796011)
- Added requests library manager and object converters (d6590b6)
- Added FrontierManagerWrapper (4f04a48)
- Added frontier object converters (7da51a4)
- Fixed script examples for new changes (101ea27)
- Optional Color logging (only if available) (c0ba0ba)
- Changed Scrapy frontier and recorder integration to scheduler+middlewares (cbe5f4f / 2fcdc06 / f7bf02b / 0d15dc1)
- Changed default frontier backend (03cd307)
- Added comment support to seeds (7d48973)
- Added doc requirements for RTD build (27daea4)
- Removed optional dependencies for setup.py and requirements (c6099f3 / 79a4e4d / e6910e3)
- Changed tests to pytest (848d2bf / edc9c01 / c318d14)
- Updated docstrings and documentation (fdccd92 / 9dec38c / 71d626f / 0977bbf)
- Changed frontier componets (Backend and Middleware) to abc (1e74467)
- Modified Scrapy frontier example to use seed loaders (0ad905d)
- Refactored Scrapy Seed loaders (a0eac84)
- Added new fields to `Request` and `Response` frontier objects (bb64afb)
- Added `ScrapyFrontierManager` (Scrapy wrapper for Frontier Manager) (8e50dc0)
- Changed frontier core objects (`Page`/`Link` to `Request`/`Response`) (74b54c8)

5.4.2 0.1

First release of Crawl Frontier.

Examples Some example projects and scripts using Frontera.

Best practices The best practices of Frontera usage.

Tests How to run and write Frontera tests.

Release Notes See what has changed in recent Frontera versions.

Index