Nectar Documentation

Release 1.4.0

Pulp Team

Jun 06, 2017

Contents

1 Request Objects 3

2 Report Objects 5

3 Downloader Configuration Objects 7

4 Event Listeners 9

5 Base Downloader API 11

6 Threads+Requests-Based Downloader 13

7 Indices and tables 15

The Interface:

CHAPTER 1

Request Objects

Request objects represent remote resource, in the form of a URL, and local storage in form of either a path or an open file-like object.

The `DownloadRequest` class has these parameters:

- url (required) the URL of the file to be downloaded as a string
- destination (required) either a local filesystem path as a string or an open file-like object
- data (optional) arbitrary data that will be passed back as part of a corresponding *report object*
- headers (optional) a dictionary of additional headers

Constructor Signature:

```
def __init__(self, url, destination, data=None, headers=None):
```

URL

The URL parameter `url` must be of a scheme (read: protocol) supported by the downloader instance that it will be passed to.

Destination

The `destination` parameter is either an absolute filesystem path as a string or an open file-like object. If the destination is an open file-like object, the downloader will **not** close it upon completion of the download; even if an error occurs.

Example:

```
destination = '/tmp/myfile'

destination = open('/tmp/myfile', wb)
```

Data

The parameter is passed, unadulterated, to the *report object* that corresponds to the request object. This is convenience mechanism to allow developers to pass arbitrary data to an *event listener*.

Headers

The `headers` parameter is an option dictionary that can contain any custom headers for a particular request.

CHAPTER 2

Report Objects

Report objects are created to correspond to a give *request object*. They are not usually instantiated by a developer, but are passed back as the parameter to event methods in an *event listener*.

They contain following fields that directly correspond to its request object:

- `url`
- `destination`
- `data`

They also contain the following informational fields:

- `state`
- `total_bytes`
- `bytes_downloaded`
- `start_time`
- `finish_time`
- `error_report`

State

The `state` field describes the current state of the download request. It is always one of the following five states:

- `waiting` - the download has not yet started
- `downloading` - the download is in progress
- `succeeded` - the download is done and was successful
- `failed` - the download is done and was unsuccessful
- `canceled` - the download is done and was canceled

Total Bytes

The total bytes to be downloaded as an integer. If this could not be determined, this field will be None.

Bytes Downloaded

The bytes downloaded so far as an integer. Initially 0.

Start Time

The date and time the download started as a `datetime.datetime` instance in the UTC timezone.

Finish Time

The date and time the download finished as a `datetime.datetime` instance in the UTC timezone.

Error Report

This field is an arbitrary dictionary that is populated only with the `state` field is `failed`. It's primary purpose is for debugging unsuccessful downloads.

When the state is not failed, this dictionary will be empty.

CHAPTER 3

Downloader Configuration Objects

Configuration objects represent common configuration across a set of *download requests*. They are *arbitrary* objects in that any keyword value passed to the constructor will be a field in the configuration object. However, only a very specific set of fields are honored by the Downloader objects.

Construct Signature:

```
def __init__(self, **kwargs):
```

The currently honored fields (read: keyword arguments) are:

- `basic_auth_username`
- `basic_auth_password`
- `headers`
- `max_concurrent`
- `max_speed`
- `proxy_url`
- `proxy_port`
- `proxy_username`
- `proxy_password`
- `ssl_validation`
- `ssl_ca_cert`
- `ssl_ca_cert_path`
- `ssl_client_cert`
- `ssl_client_cert_path`
- `ssl_client_key`
- `ssl_client_key_path`

This list will continue to grow and evolve as more downloaders are added, especially downloaders that support protocols other than HTTP and HTTPS.

Download Control

`max_concurrent` is an integer that tells the downloader the maximum number of files to download concurrently (read: in parallel). If this number is not provided, each downloader has its own default value that will be used instead.

`max_speed` is an integer that tells the downloader at what speed to throttle the downloads. The units are: bytes/second.

HTTP Basic Auth Support

The fields `basic_auth_username` and `basic_auth_password` are used for the HTTP basic authorization header. The username and password fields must be provided in plain text. The downloaders will Base64 encode them.

SSL Support

`ssl_validation` is a boolean that tells the downloader to verify the identity of the remote server by checking its SSL certificate. If this parameter is not provided, validation is assumed to be set to True.

`ssl_ca_cert` and `ssl_ca_cert_path` parameters are used to provide an alternative CA cert to the downloader. The `ssl_ca_cert` parameter should point the CA pem data and the `ssl_ca_cert_path` is a file system path to the CA cert file. Both are strings. However, these parameters are mutually exclusive, and the behavior of the downloader is undefined if both are provided.

`ssl_client_cert`, `ssl_client_cert_path`, `ssl_client_key`, and `ssl_client_key_path` are used to provide two-way authentication via the SSL protocol. Just like the ssl_ca_cert params, these point to either the data or to a file path; and correlated parameters are mutually exclusive.

Proxy Support

`proxy_url` is string in the form of scheme://host, where scheme is either *http* or *https*.

`proxy_port` is an integer port number.

`proxy_username` and `proxy_password` are used for authentication and must be provided in plain text.

Headers

`headers` is a dictionary that can contain any additional headers that should be used for every request.

CHAPTER 4

Event Listeners

A DownloadEventListener object is passed into a downloader's constructor. On certain events, methods on the event listener are used as callbacks to inform on headers availability, a download starting, a download's progress, and a download's success or failure.

This gives the developer an opportunity to develop event-driven code by overriding the this base class.

The event listener's interface is as follows:

```
def download_started(self, report):
def download_progress(self, report):
def download_succeeded(self, report):
def download_failed(self, report):
def download_headers(self, report):
```

All methods are passed a *report object* that corresponds to the download request that has triggered the event.

Download Started

This event is handled by the `download_started` method. It is called once per download request when the download starts.

Download Progress

This event is handled by the `download_progress` method. It may be called multiple times per download request. It is guaranteed to be called once.

Download Succeeded

This event is handled by the `download_succeeded` method. It is called if the download completed successfullly.

Download Failed

This event is handled by the `download_failed` method. It is called if the download encountered an error. Additional information about the error will be in the report's `error_report` dictionary.

Download Headers

This event is handled by the `download_headers` method. It is called at the moment when headers from the response are available.

The Downloaders:

CHAPTER 5

Base Downloader API

The Downloader base class defines the general downloader API. It has a number of simple methods and behaviors that are common across any derived classes. This provides the *pluggable* aspect of the Nectar library.

Instantiation

A downloader constructor takes two parameters, one required and one optional:

- a *configuration object*, required
- an *event listener*, optional

Configuration

The *configuration object* provides options that are common across all download requests. Their documentation have be found *here*.

Events

As the downloader downloads files, it fires off events by calling methods on the provided *event listener*.

If no event listener is passed to a downloader's constructor, a no-op event listener is automatically used.

Event listener's methods are described *here*.

Downloading Requests

The downloaders do one thing: they download files. The `download` method on a downloader takes a list of *request objects* and downloads them using information from it's *configuration*.

The `download` signature:

```
def download(self, request_list):
```

The `request_list` parameter doesn't necessarily need to be a list, but it does need to be an **iterator** of request objects.

Canceling Downloads

Downloaders support the canceling of the a call to `download` via the `cancel` method. Since downloading is synchronous and does not return until all the download requests have been either successfully downloaded or have failed in their attempt, the `cancel` method must be called by another thread.

CHAPTER 6

Threads+Requests-Based Downloader

The threaded downloader leverages both python threads and the requests library. It is optimized for speed when use an already threaded applications. It provides the *downloader API*.

Its major use case is downloading lots of files quickly.

Warning: The proxy support for this downloader is incomplete. Due to limitations in the urllib3 library, HTTPS requests via an HTTPS proxy is not supported. However, all other permutations are.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search