streaming-form-data Documentation

Release 1.5.1

Siddhant Goel

Nov 03, 2019

Contents:

`streaming_form_data` provides a Python parser for parsing `multipart/form-data` input chunks (the most commonly used encoding when submitting data through HTML forms).

Chunk size is determined by the API user, but currently there are no restrictions on what the chunk size should be, since the parser works byte-by-byte (which means that passing the entire input as a single chunk should also work).

CHAPTER 1

Installation

```
$ pip install streaming_form_data
```

The core parser is written in `Cython`, which is a superset of Python that compiles the input down to a C extension which can then be imported in normal Python code.

The compiled C parser code is included in the PyPI package, hence the installation requires a working C compiler.

CHAPTER 2

Usage

```
>>> from streaming_form_data import StreamingFormDataParser
>>> from streaming_form_data.targets import ValueTarget, FileTarget, NullTarget
>>>
>>> headers = {'Content-Type': 'multipart/form-data; boundary=boundary'}
>>>
>>> parser = StreamingFormDataParser(headers=headers)
>>>
>>> parser.register('name', ValueTarget())
>>> parser.register('file', FileTarget('/tmp/file.txt'))
>>> parser.register('discard-me', NullTarget())
>>>
>>> for chunk in request.body:
...     parser.data_received(chunk)
...
>>>
```

Usage can broadly be split into three stages.

2.1 1. Initialization

The `StreamingFormDataParser` class expects a dictionary of HTTP request headers when being instantiated. These headers are used to determine the input `Content-Type` and a few other metadata.

2.2 2. Input Registration

HTML forms typically have multiple fields. For instance, a form could have a text input field called `name` and a file input field called `file`.

This needs to be communicated to the parser using the `parser.register` function. This function expects two arguments - the name of the input field, and the associated `Target` class (which determines how the input should be

handled).

For instance, if you want to store the contents of the `name` field in an in-memory variable, and the `file` field in a file on disk, you can tell this to the parser as follows.

```
>>> name_target = ValueTarget()
>>> file_target = FileTarget('/tmp/file.dat')
>>>
>>> parser.register('name', name_target)
>>> parser.register('file', file_target)
```

2.3 3. Streaming data

At this stage the parser has everything it needs to be able to work. Depending on what web framework you're using, just pass the actual HTTP request body to the parser, either one chunk at a time or the complete thing at once.

```
>> chunk = read_next_chunk() # depends on your web framework of choice
>>
>> parser.data_received(chunk)
```

CHAPTER 3

API

3.1 `StreamingFormDataParser`

This class is the main entry point, and expects a dictionary of HTTP request `headers`. These headers are used to determine the input `Content-Type` and a few other metadata.

3.2 `Target` classes

When registering inputs with the parser, instances of subclasses of the `Target` class should be used. These target classes ultimately determine what to do with the data.

3.2.1 `ValueTarget`

`ValueTarget` objects hold the input in memory.

```
>>> target = ValueTarget()
```

3.2.2 `FileTarget`

`FileTarget` objects stream the contents to a file on-disk.

```
>>> target = FileTarget('/tmp/file.txt')
```

3.2.3 `SHA256Target`

`SHA256Target` objects calculate a `SHA256` hash of the given input, and hold the result in memory.

```
>>> target = SHA256Target()
```

3.2.4 `NullTarget`

`NullTarget` objects discard the input completely.

```
>>> target = NullTarget()
```

3.2.5 Custom `Target` classes

It's possible to define custom targets for your specific use case by inheriting the `streaming_form_data.targets.BaseTarget` class and overriding the `on_data_received` function.

```
>>> from streaming_form_data.targets import BaseTarget
>>>
>>> class CustomTarget(BaseTarget):
...     def on_data_received(self, chunk):
...         do_something_with(chunk)
```

If the `Content-Disposition` header included the `filename` directive, this value will be available as the `self.multipart_filename` attribute in `Target` classes.

Similarly, if the `Content-Type` header is available for the uploaded files, this value will be available as the `self.multipart_content_type` attribute in `Target` classes.

3.3 `Validator` classes

`Target` classes accept a `validator` callable when being instantiated. Every time `data_received` is called with a given `chunk`, the target runs this `chunk` through the given callable.

This is useful for performing certain validation tasks like making sure the input size is not exceeding a certain value. This is shown in the following code snippet.

```
>>> from streaming_form_data.targets import ValueTarget
>>>
>>> target = ValueTarget(validator=MaxSizeValidator(100))
```

CHAPTER 4

Examples

- `Bottle` - https://git.io/vhCUy
- `Flask` - https://git.io/fjPoA
- `Tornado` - https://git.io/vhCUM

If you'd like to document usage with another web framework (which ideally allows chunked HTTP reads), please open an issue or a pull request.

CHAPTER 5

Indices and tables

- search