Parsing HTTP Headers Open-Source Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your reports for your project.

Here are some things to note when working on your report, specifically about the **General Information & Licensing** section for each technology.

- **Code Repository**: Please link the code and not the documentation. If you'd like to refer to the documentation in the **Magic** section, you're more than welcome to, but we need to see the code you're referring to as well.
- **License Type**: Three letter acronym is fine.
- **License Description**: No need for the entire license here, just what separates it from the rest.
- **License Restrictions**: What can you *not* do as a result of using this technology in your project? Some licenses prevent you from using the project for commercial use, for example.

Also, feel free to extend the cell of any section if you feel you need more room.

If there's anything we can clarify, please don't hesitate to reach out! You can reach us using the methods outlined on the course website or see us during our office hours.

Flask and werkzeug

General Information & Licensing

Code Repository	https://github.com/pallets/werkzeug
License Type	BSD-3-Clause License
License Description	• Copyright 2007 Pallets • Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: • Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. • Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

	• Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. • THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
License Restrictions	Following from the above, this license allows for a wide range of permitted use. Restrictions, however, do exist surrounding liability and warranty. Copyright holders have explicitly distanced themselves from association with the products derived from this framework without explicit written consent.

Magic ★★ ° ˙ ˚ ☽ ˚ ⌒ ˚ ★ ≡ ✧ ★ ⌯

(I do not know why this document is formatted like this, it would not let me change it. The full report with tracebacks are on the next page.

```
request.method == 'POST'
```

- This is the beginning of the traceback for header parsing code. This is in ***our*** code, which calls the “method” function on the current request to get what type of HTTP request the server is receiving.

```
def __init__(
    self,
    method: str,
    scheme: str,
    server: t.Optional[t.Tuple[str, t.Optional[int]]],
    root_path: str,
    path: str,
    query_string: bytes,
    headers: Headers,
    remote_addr: t.Optional[str],
) -> None:
    #: The method the request was made with, such as ``GET``.
    self.method = method.upper()
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/sansio/request.py#L131
- This is the next traceback from the previous “request.method”. We go to a library named werkzeug. In the library, we go to request.py, which is the image above. This is where the function “method” is initialized for the request.

```
class Request:
    """Represents the non-IO parts of a HTTP request, including the
    method, URL info, and headers.

    This class is not meant for general use. It should only be used when
    implementing WSGI, ASGI, or another HTTP application spec. Werkzeug
    provides a WSGI implementation at :cls:`werkzeug.wrappers.Request`.

    :param method: The method the request was made with, such as
        ``GET``.
    :param scheme: The URL scheme of the protocol the request used, such
        as ``https`` or ``wss``.
    :param server: The address of the server. ``(host, port)``,
        ``(path, None)`` for unix sockets, or ``None`` if not known.
    :param root_path: The prefix that the application is mounted under.
        This is prepended to generated URLs, but is not part of route
        matching.
    :param path: The path part of the URL after ``root_path``.
    :param query_string: The part of the URL after the "?".
    :param headers: The headers received with the request.
    :param remote_addr: The address of the client sending the request.
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/sansio/request.py#L38
- In this same request.py file, we have a class called Request. This class is created, and contains all the functions to use to get information from the HTTP request, such as the “method” function that we saw earlier.

```
def __init__(
    self,
    method: str,
    scheme: str,
    server: t.Optional[t.Tuple[str, t.Optional[int]]],
    root_path: str,
    path: str,
    query_string: bytes,
    headers: Headers,
    remote_addr: t.Optional[str],
) -> None:
    #: The method the request was made with, such as ``GET``.
    self.method = method.upper()
    #: The URL scheme of the protocol the request used, such as
    #: ``https`` or ``wss``.
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/sansio/request.py#L131
- In the Request class, we now see the “method” function that was called in our first line of code. “method” is one of initialized variables of this Request class which is why we can call it on a request.
- In this class, we also see a initialized variable named “headers”, which is a Headers class.

```
class Headers:
    """An object that stores some headers. It has a dict-like interface,
    but is ordered, can store the same key multiple times, and iterating
    yields ``(key, value)`` pairs instead of only keys.

    This data structure is useful if you want a nicer way to handle WSGI
    headers which are stored as tuples in a list.

    From Werkzeug 0.3 onwards, the :exc:`KeyError` raised by this class is
    also a subclass of the :class:`~exceptions.BadRequest` HTTP exception
    and will render a page for a ``400 BAD REQUEST`` if caught in a
    catch-all for HTTP exceptions.

    Headers is mostly compatible with the Python :class:`wsgiref.headers.Headers`
    class, with the exception of `__getitem__`.  :mod:`wsgiref` will return
    `None` for ``headers['missing']``, whereas :class:`Headers` will raise
    a :class:`KeyError`.
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/datastructures.py#L848
- When we trace back the class from the previous code block we saw, we are now in a new python file called “datastructures.py”. This file contains a class named

“Headers”.

```
def __str__(self):
    """Returns formatted headers suitable for HTTP transmission."""
    strs = []
    for key, value in self.to_wsgi_list():
        strs.append(f"{key}: {value}")
    strs.append("\r\n")
    return "\r\n".join(strs)
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/datastructures.py#L1289
- In this Headers class, we have a function named str. This seems to heavily resemble code we see in our Homeworks, with the CRLF, “\r\n”. This function takes in headers, and parses is so that it can be read by the client in a HTTP response.

```
def _set_property(name, doc=None):
    def fget(self):
        def on_update(header_set):
            if not header_set and name in self:
                del self[name]
            elif header_set:
                self[name] = header_set.to_header()

        return http.parse_set_header(self.get(name), on_update)

    return property(fget, doc=doc)
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/datastructures.py#L2757
- In the same Headers class, we have a function called “_set_property) which calls a function called “parse_set_header).

```
def parse_set_header(
    value: t.Optional[str],
    on_update: t.Optional[t.Callable[["ds.HeaderSet"], None]] = None,
) -> "ds.HeaderSet":
    """Parse a set-like header and return a
    :class:`~werkzeug.datastructures.HeaderSet` object:

    >>> hs = parse_set_header('token, "quoted value"')

    The return value is an object that treats the items case-insensitively
    and keeps the order of the items:

    >>> 'TOKEN' in hs
    True
    >>> hs.index('quoted value')
    1
    >>> hs
    HeaderSet(['token', 'quoted value'])

    To create a header from the :class:`HeaderSet` again, use the
    :func:`dump_header` function.

    :param value: a set header to be parsed.
    :param on_update: an optional callable that is called every time a
                      value on the :class:`~werkzeug.datastructures.HeaderSet`
                      object is changed.
    :return: a :class:`~werkzeug.datastructures.HeaderSet`
    """
    if not value:
        return ds.HeaderSet(None, on_update)
    return ds.HeaderSet(parse_list_header(value), on_update)
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/http.py#L608
- When we trace back the function we saw in the previous code block, we get the “parse_set_header” function. This function is in a new python file called “http.py”. This seems to be just what we are looking for because we are parsing HTTP requests throughout our homeworks and this project.

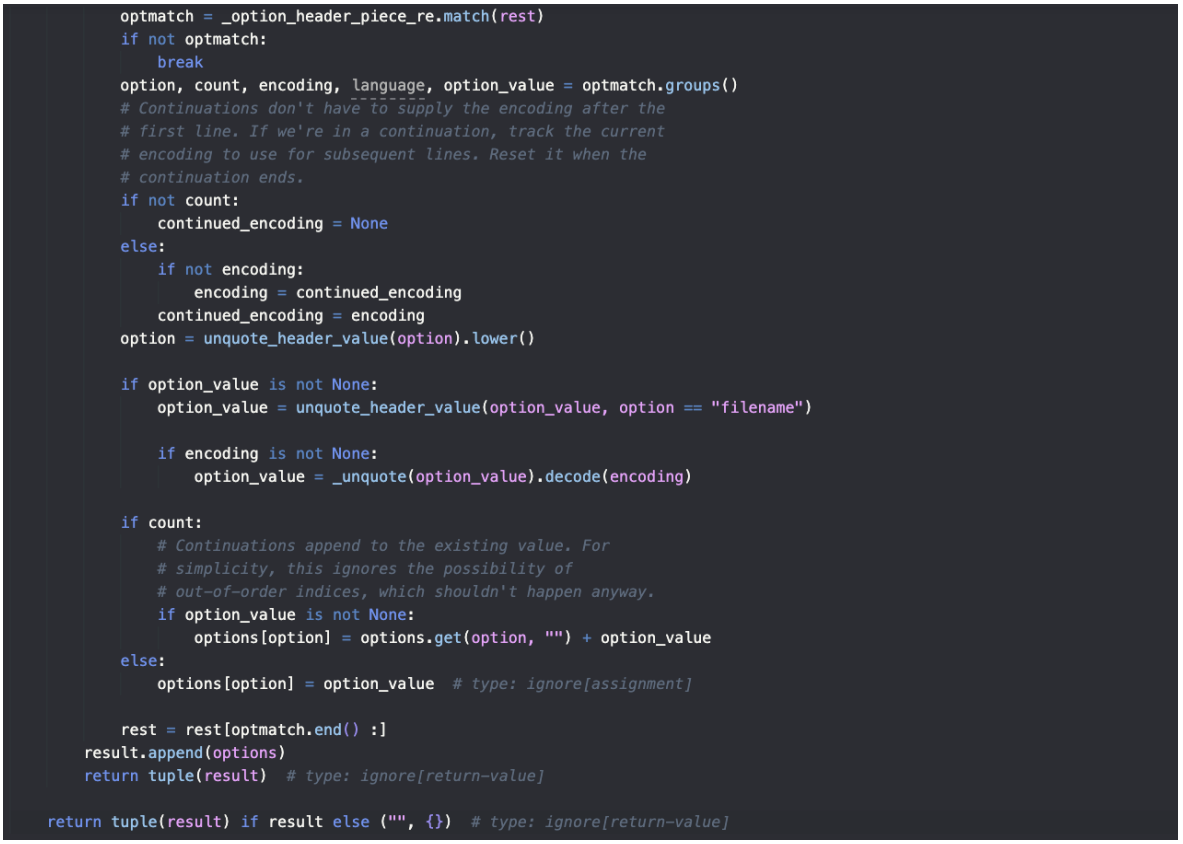
```python
def parse_list_header(value: str) -> t.List[str]:
    """Parse lists as described by RFC 2068 Section 2.

    In particular, parse comma-separated lists where the elements of
    the list may include quoted-strings.  A quoted-string could
    contain a comma.  A non-quoted string could have quotes in the
    middle.  Quotes are removed automatically after parsing.

    It basically works like :func:`parse_set_header` just that items
    may appear multiple times and case sensitivity is preserved.

    The return value is a standard :class:`list`:

    >>> parse_list_header('token, "quoted value"')
    ['token', 'quoted value']

    To create a header from the :class:`list` again, use the
    :func:`dump_header` function.

    :param value: a string with a list header.
    :return: :class:`list`
    """
    result = []
    for item in _parse_list_header(value):
        if item[:1] == item[-1:] == '"':
            item = unquote_header_value(item[1:-1])
        result.append(item)
    return result
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/http.py#L307
- Tracing back the function that was called in “parse_set_header”, we arrive at a “parse_list_header” function in the same “http.py” file. This function also resembles the homework because it is parsing a list of headers from a request, which we do in the homework.

```python
def unquote_header_value(value: str, is_filename: bool = False) -> str:
    r"""Unquotes a header value.  (Reversal of :func:`quote_header_value`).
    This does not use the real unquoting but what browsers are actually
    using for quoting.

    .. versionadded:: 0.5

    :param value: the header value to unquote.
    :param is_filename: The value represents a filename or path.
    """
    if value and value[0] == value[-1] == '"':
        # this is not the real unquoting, but fixing this so that the
        # RFC is met will result in bugs with internet explorer and
        # probably some other browsers as well.  IE for example is
        # uploading files with "C:\foo\bar.txt" as filename
        value = value[1:-1]

        # if this is a filename and the starting characters look like
        # a UNC path, then just return the value without quotes.  Using the
        # replace sequence below on a UNC path has the effect of turning
        # the leading double slash into a single slash and then
        # _fix_ie_filename() doesn't work correctly.  See #458.
        if not is_filename or value[:2] != "\\\\":
            return value.replace("\\\\", "\\").replace('\\"', '"')
    return value
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/http.py#L216
- Following the function from right before which called this function, we are at the function “unquote_header_value”, in the same “http.py” file. This function removes the quotes from the headers of the request, which we have also done in the homework.

```
def parse_options_header(value: t.Optional[str]) -> t.Tuple[str, t.Dict[str, str]]:
    """Parse a ``Content-Type``-like header into a tuple with the
    value and any options:

    >>> parse_options_header('text/html; charset=utf8')
    ('text/html', {'charset': 'utf8'})

    This should is not for ``Cache-Control``-like headers, which use a
    different format. For those, use :func:`parse_dict_header`.

    :param value: The header value to parse.

    .. versionchanged:: 2.2
        Option names are always converted to lowercase.

    .. versionchanged:: 2.1
        The ``multiple`` parameter is deprecated and will be removed in
        Werkzeug 2.2.

    .. versionchanged:: 0.15
        :rfc:`2231` parameter continuations are handled.

    .. versionadded:: 0.5
    """
    if not value:
        return "", {}

    result: t.List[t.Any] = []

    value = "," + value.replace("\n", ",")
    while value:
        match = _option_header_start_mime_type.match(value)
        if not match:
            break
        result.append(match.group(1))  # mimetype
        options: t.Dict[str, str] = {}
        # Parse options
        rest = match.group(2)
        encoding: t.Optional[str]
        continued_encoding: t.Optional[str] = None
        while rest:
            optmatch = _option_header_piece_re.match(rest)
```



```
            optmatch = _option_header_piece_re.match(rest)
            if not optmatch:
                break
            option, count, encoding, language, option_value = optmatch.groups()
            # Continuations don't have to supply the encoding after the
            # first line. If we're in a continuation, track the current
            # encoding to use for subsequent lines. Reset it when the
            # continuation ends.
            if not count:
                continued_encoding = None
            else:
                if not encoding:
                    encoding = continued_encoding
                continued_encoding = encoding
            option = unquote_header_value(option).lower()

            if option_value is not None:
                option_value = unquote_header_value(option_value, option == "filename")

                if encoding is not None:
                    option_value = _unquote(option_value).decode(encoding)

            if count:
                # Continuations append to the existing value. For
                # simplicity, this ignores the possibility of
                # out-of-order indices, which shouldn't happen anyway.
                if option_value is not None:
                    options[option] = options.get(option, "") + option_value
            else:
                options[option] = option_value  # type: ignore[assignment]

            rest = rest[optmatch.end() :]
        result.append(options)
        return tuple(result)  # type: ignore[return-value]

    return tuple(result) if result else ("", {})  # type: ignore[return-value]
```

- https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/http.py#L377
- This is a function inside the same “http.py” file that we have examined for so long. This function, “parse_options_header” takes in a header key-value pair and parses it. This function is called all throughout our project code; it is the basis of all the HTTP request parsing. It takes in a key-value pair of the header, and parses it. It replaces “\n” with an empty string in the value of the key-value pair. It then does a lot of edge cases and checking to see if it is valid. In the end, it returns a tuple with the key as the first element, and the value as the second element. This is the basis of all our HTTP request headers parsing.

```
def parse_dict_header(value: str, cls: t.Type[dict] = dict) -> t.Dict[str, str]:
    """Parse lists of key, value pairs as described by RFC 2068 Section 2 and
    convert them into a python dict (or any other mapping object created from
    the type with a dict like interface provided by the `cls` argument):

    >>> d = parse_dict_header('foo="is a fish", bar="as well"')
    >>> type(d) is dict
    True
    >>> sorted(d.items())
    [('bar', 'as well'), ('foo', 'is a fish')]

    If there is no value for a key it will be `None`:

    >>> parse_dict_header('key_without_value')
    {'key_without_value': None}

    To create a header from the :class:`dict` again, use the
    :func:`dump_header` function.

    .. versionchanged:: 0.9
       Added support for `cls` argument.

    :param value: a string with a dict header.
    :param cls: callable to use for storage of parsed results.
    :return: an instance of `cls`
    """
    result = cls()
    if isinstance(value, bytes):
        value = value.decode("latin1")
    for item in _parse_list_header(value):
        if "=" not in item:
            result[item] = None
            continue
        name, value = item.split("=", 1)
        if value[:1] == value[-1:] == '"':
            value = unquote_header_value(value[1:-1])
        result[name] = value
    return result
```

```
def quote_header_value(
    value: t.Union[str, int], extra_chars: str = "", allow_token: bool = True
) -> str:
    """Quote a header value if necessary.

    .. versionadded:: 0.5

    :param value: the value to quote.
    :param extra_chars: a list of extra characters to skip quoting.
    :param allow_token: if this is enabled token values are returned
                        unchanged.
    """
    if isinstance(value, bytes):
        value = value.decode("latin1")
    value = str(value)
    if allow_token:
        token_chars = _token_chars | set(extra_chars)
        if set(value).issubset(token_chars):
            return value
    value = value.replace("\\", "\\\\").replace('"', '\\"')
    return f'"{value}"'
```

```
def parse_list_header(value: str) -> t.List[str]:
    """Parse lists as described by RFC 2068 Section 2.

    In particular, parse comma-separated lists where the elements of
    the list may include quoted-strings.  A quoted-string could
    contain a comma.  A non-quoted string could have quotes in the
    middle.  Quotes are removed automatically after parsing.

    It basically works like :func:`parse_set_header` just that items
    may appear multiple times and case sensitivity is preserved.

    The return value is a standard :class:`list`:

    >>> parse_list_header('token, "quoted value"')
    ['token', 'quoted value']

    To create a header from the :class:`list` again, use the
    :func:`dump_header` function.

    :param value: a string with a list header.
    :return: :class:`list`
    """
    result = []
    for item in _parse_list_header(value):
        if item[:1] == item[-1:] == '"':
            item = unquote_header_value(item[1:-1])
        result.append(item)
    return result
```