Open-Source Technology Use Report

Proof of knowing your stuff in CSE312

Flask

General Information & Licensing

|  |  |
| --- | --- |
| Code Repository | <https://github.com/pallets/flask> |
| License Type | BSD-3 |
| License Description | * Any source code must contain the license and copyright statement * Documentation for binaries must contain the license and copyright statement * Redistribution for commercial purposes is fine as long as Flask is clearly listed as the copyright holder |
| License Restrictions | * We are not allowed to use the name of the copyright holder or any contributors for endorsing products made using Flask without written permission |
| Who worked with this? |  |

*Use as many of the sections below as needed, or create more, to explain every function, method, class, or object type you used from this library/framework.*

Purpose

|  |
| --- |
| Replace this text with some that answers the following questions for the above tech:   * We used this part of Flask's codebase to access and handle headers recieved from the front end and to set headers on responses (like cookies). This was very important in how we served content and allowed us to set up an easy cookie based authentication system. |

*Magic* ⋆★͎۪۫｡˚۰˚☽˚⁀➷｡˚★彡͎۪۫⋆ ༄

|  |
| --- |
| *Dispel the magic of this technology. Replace this text with some that answers the following questions for the above tech:*   * *How does this technology do what it does for you in the* ***Purpose*** *section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.* * *Where is the specific code that does what you use the tech for? You* ***must*** *provide a link to the specific file in the repository for your tech with a line number or number range.* * *If there is more than one step in the chain of calls (hint: there will be), you must provide links for the entire chain of calls from your code, to the library code that actually accomplishes the task for you.* * *Example: If you use an object of type HttpRequest in your code which contains the headers of the request, you must show exactly how that object parsed the original headers from the TCP socket. This will often involve tracing through multiple libraries and you must show the entire trace through all these libraries with links to all the involved code.*   *\*This section may grow beyond the page for many features.*  In the TCP report, we said the web application will treats the socket as a stream and reads data from it. But since it calls “readline”, it will only read the first line, which is the requestline.  <https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L267>  def parse\_request(self):  *"""Parse a request (internal).   The request should be stored in self.raw\_requestline; the results  are in self.command, self.path, self.request\_version and  self.headers.   Return True for success, False for failure; on failure, any relevant  error response has already been sent back.   """* self.command = None # set in case of error on the first line  self.request\_version = version = self.default\_request\_version  self.close\_connection = True  requestline = str(self.raw\_requestline, 'iso-8859-1')  requestline = requestline.rstrip('\r\n')  self.requestline = requestline  words = requestline.split()  if len(words) == 0:  return False   if len(words) >= 3: # Enough to determine protocol version  version = words[-1]  try:  if not version.startswith('HTTP/'):  raise ValueError  base\_version\_number = version.split('/', 1)[1]  version\_number = base\_version\_number.split(".")  # RFC 2145 section 3.1 says there can be only one "." and  # - major and minor numbers MUST be treated as  # separate integers;  # - HTTP/2.4 is a lower version than HTTP/2.13, which in  # turn is lower than HTTP/12.3;  # - Leading zeros MUST be ignored by recipients.  if len(version\_number) != 2:  raise ValueError  version\_number = int(version\_number[0]), int(version\_number[1])  except (ValueError, IndexError):  self.send\_error(  HTTPStatus.BAD\_REQUEST,  "Bad request version (%r)" % version)  return False  if version\_number >= (1, 1) and self.protocol\_version >= "HTTP/1.1":  self.close\_connection = False  if version\_number >= (2, 0):  self.send\_error(  HTTPStatus.HTTP\_VERSION\_NOT\_SUPPORTED,  "Invalid HTTP version (%s)" % base\_version\_number)  return False  self.request\_version = version   if not 2 <= len(words) <= 3:  self.send\_error(  HTTPStatus.BAD\_REQUEST,  "Bad request syntax (%r)" % requestline)  return False  command, path = words[:2]  if len(words) == 2:  self.close\_connection = True  if command != 'GET':  self.send\_error(  HTTPStatus.BAD\_REQUEST,  "Bad HTTP/0.9 request type (%r)" % command)  return False  self.command, self.path = command, path   # Examine the headers and look for a Connection directive.  try:  self.headers = http.client.parse\_headers(self.rfile,  \_class=self.MessageClass)  except http.client.LineTooLong as err:  self.send\_error(  HTTPStatus.REQUEST\_HEADER\_FIELDS\_TOO\_LARGE,  "Line too long",  str(err))  return False  except http.client.HTTPException as err:  self.send\_error(  HTTPStatus.REQUEST\_HEADER\_FIELDS\_TOO\_LARGE,  "Too many headers",  str(err)  )  return False   conntype = self.headers.get('Connection', "")  if conntype.lower() == 'close':  self.close\_connection = True  elif (conntype.lower() == 'keep-alive' and  self.protocol\_version >= "HTTP/1.1"):  self.close\_connection = False  # Examine the headers and look for an Expect directive  expect = self.headers.get('Expect', "")  if (expect.lower() == "100-continue" and  self.protocol\_version >= "HTTP/1.1" and  self.request\_version >= "HTTP/1.1"):  if not self.handle\_expect\_100():  return False  return True  Now it starts to parse the HTTP requestline. First, it use “rstrip” to trim the ending “\r\n” .  <https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L282>  Then it splits the request line by whitespace into 3 fields, which is the method, url and the HTTP version. It splits the version field by “/” because the format is “HTTP/{version number}”.  Then it takes the method and path from the “words” by  command, path = words[:2]  Then it uses “http.client.parse\_headers” to parse the headers.  <https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L342>  self.headers = http.client.parse\_headers(self.rfile,  \_class=self.MessageClass)    def parse\_headers(fp, \_class=HTTPMessage):  *"""Parses only RFC2822 headers from a file pointer.   email Parser wants to see strings rather than bytes.  But a TextIOWrapper around self.rfile would buffer too many bytes  from the stream, bytes which we later need to read as bytes.  So we read the correct bytes here, as bytes, for email Parser  to parse.   """* headers = []  while True:  line = fp.readline(\_MAXLINE + 1)  if len(line) > \_MAXLINE:  raise LineTooLong("header line")  headers.append(line)  if len(headers) > \_MAXHEADERS:  raise HTTPException("got more than %d headers" % \_MAXHEADERS)  if line in (b'\r\n', b'\n', b''):  break  hstring = b''.join(headers).decode('iso-8859-1')  return email.parser.Parser(\_class=\_class).parsestr(hstring)  First, it assign a empty list to headers and then use a while loop to keep reading lines from the socket until it hits an empty line, which is (b'\r\n' or b'\n' or b'').  Then it concat the string again and let “email.parser.Parser.parsestr” to parse the header.  <https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/client.py#L236>  def parsestr(self, text, headersonly=False):  *"""Create a message structure from a string.   Returns the root of the message structure. Optional headersonly is a  flag specifying whether to stop parsing after reading the headers or  not. The default is False, meaning it parses the entire contents of  the file.  """* return self.parse(StringIO(text), headersonly=headersonly)  Then in the “parsestr” method, it calls “self.parse” method.  <https://github.com/python/cpython/blob/a29a7b9b786d6b928c4bb4e6e683a3788e3ab1c1/Lib/email/parser.py#L64>  def parse(self, fp, headersonly=False):  *"""Create a message structure from the data in a file.   Reads all the data from the file and returns the root of the message  structure. Optional headersonly is a flag specifying whether to stop  parsing after reading the headers or not. The default is False,  meaning it parses the entire contents of the file.  """* feedparser = FeedParser(self.\_class, policy=self.policy)  if headersonly:  feedparser.\_set\_headersonly()  while True:  data = fp.read(8192)  if not data:  break  feedparser.feed(data)  return feedparser.close()  It uses “FeedParser” for parsing  <https://github.com/python/cpython/blob/a29a7b9b786d6b928c4bb4e6e683a3788e3ab1c1/Lib/email/parser.py#L49>  The “\_parse\_header” in “FeedParser” will parse the header.  <https://github.com/python/cpython/blob/a29a7b9b786d6b928c4bb4e6e683a3788e3ab1c1/Lib/email/feedparser.py#L471>  def \_parse\_headers(self, lines):  # Passed a list of lines that make up the headers for the current msg  lastheader = ''  lastvalue = []  for lineno, line in enumerate(lines):  # Check for continuation  if line[0] in ' \t':  if not lastheader:  # The first line of the headers was a continuation. This  # is illegal, so let's note the defect, store the illegal  # line, and ignore it for purposes of headers.  defect = errors.FirstHeaderLineIsContinuationDefect(line)  self.policy.handle\_defect(self.\_cur, defect)  continue  lastvalue.append(line)  continue  if lastheader:  self.\_cur.set\_raw(\*self.policy.header\_source\_parse(lastvalue))  lastheader, lastvalue = '', []  # Check for envelope header, i.e. unix-from  if line.startswith('From '):  if lineno == 0:  # Strip off the trailing newline  mo = NLCRE\_eol.search(line)  if mo:  line = line[:-len(mo.group(0))]  self.\_cur.set\_unixfrom(line)  continue  elif lineno == len(lines) - 1:  # Something looking like a unix-from at the end - it's  # probably the first line of the body, so push back the  # line and stop.  self.\_input.unreadline(line)  return  else:  # Weirdly placed unix-from line. Note this as a defect  # and ignore it.  defect = errors.MisplacedEnvelopeHeaderDefect(line)  self.\_cur.defects.append(defect)  continue  # Split the line on the colon separating field name from value.  # There will always be a colon, because if there wasn't the part of  # the parser that calls us would have started parsing the body.  i = line.find(':')   # If the colon is on the start of the line the header is clearly  # malformed, but we might be able to salvage the rest of the  # message. Track the error but keep going.  if i == 0:  defect = errors.InvalidHeaderDefect("Missing header name.")  self.\_cur.defects.append(defect)  continue   assert i>0, "\_parse\_headers fed line with no : and no leading WS"  lastheader = line[:i]  lastvalue = [line]  # Done with all the lines, so handle the last header.  if lastheader:  self.\_cur.set\_raw(\*self.policy.header\_source\_parse(lastvalue))  It uses “line.find” to find the index of “:” in the header line to split the raw header.  Then it uses “line[:i]” to get the stirng that’s from 0 to i-1 of the line string, which is exactly the header name.  <https://github.com/python/cpython/blob/a29a7b9b786d6b928c4bb4e6e683a3788e3ab1c1/Lib/email/feedparser.py#L514>  Then it uses “self.\_cur.set\_raw(\*self.policy.header\_source\_parse(lastvalue))” to parse the header value.  <https://github.com/python/cpython/blob/a29a7b9b786d6b928c4bb4e6e683a3788e3ab1c1/Lib/email/feedparser.py#L529>  According to the Python 3 Documentation  (<https://docs.python.org/3/library/email.policy.html#email.policy.Policy.header_source_parse>)  **header\_source\_parse**(*sourcelines*)  The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the (name, value) tuple that is to be stored in the Message to represent the parsed header.  If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the ‘:’ separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.  *sourcelines* may contain surrogateescaped binary data.  The parsed header will be returned and will be stored to the “self.\_cur”.  Then, back in the “\_parsegen” method, it will use “self.\_cur.get\_content\_type()” function to get the type of content by from the header “content-type”  def get\_content\_type(self):  *"""Return the message's content type.   The returned string is coerced to lower case of the form  `maintype/subtype'. If there was no Content-Type header in the  message, the default type as given by get\_default\_type() will be  returned. Since according to RFC 2045, messages always have a default  type this will always return a value.   RFC 2045 defines a message's default type to be text/plain unless it  appears inside a multipart/digest container, in which case it would be  message/rfc822.  """* missing = object()  value = self.get('content-type', missing)  if value is missing:  # This should have no parameters  return self.get\_default\_type()  ctype = \_splitparam(value)[0].lower()  # RFC 2045, section 5.2 says if its invalid, use text/plain  if ctype.count('/') != 1:  return 'text/plain'  return ctype  def \_splitparam(param):  # Split header parameters. BAW: this may be too simple. It isn't  # strictly RFC 2045 (section 5.1) compliant, but it catches most headers  # found in the wild. We may eventually need a full fledged parser.  # RDM: we might have a Header here; for now just stringify it.  a, sep, b = str(param).partition(';')  if not sep:  return a.strip(), None  return a.strip(), b.strip()  It uses the “\_splitparam” to get the parts before “;”, which is how the type appears in the “content-type” header. The part after “;” will be some detail information such as the “boundary” when it’s a “multipart-form”.  For now, the headers are fully processed.  response.set\_cookie('auth\_token', value=auth\_token, httponly=True,max\_age=604800)  response.set\_cookie is a method used to set a cookie on an HTTP response. This method takes the name of the cookie and its value as arguments, and stores them in the HTTP response header. It can also take other optional arguments to specify things like the cookie's expiration date, domain, and security settings.  In ‘set\_cookies’, it calls ‘headers.add’ to add two elements into header, one is “"Set-Cookie" ,the other one is dump\_cookie, it will return a string base on the parameters from set \_cookies . <https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/sansio/response.py#L228>  <https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/sansio/response.py#L230>  When browser receive response from server, it will take the corresponding value from ‘set\_cookie’, and add them to cookies, these content will be send to server when next time it send request. When the ‘auth\_token’ is verified, server can recognized it.  **make\_response()** – converts the output of view functions like redirect() into a response\_class; response\_class objects are used to set cookies and perform other modifications on the HTTP response before sending it to the client. <https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L2055> The code starts by unpacking the input tuple into various variables and performing error checking on the body and tuple itself. It then instantiates a Response (<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L2149>) using the body and the unpacked tuple values. Then, it calls the cast function (<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L2181>) to cast the Response object as type Response. After that, it associates any status or headers with the response\_class and returns it. Response class- <https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/wrappers.py#L136> Basic response wrapper for Flask. It contains code for determining max cookie size based on either the application configuration or the Werkzeug defeaults (<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/wrappers.py#L168>). |