Open-Source Technology Use Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your report for each of the technologies you use in 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'd like 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.
- **Who worked with this?**: It's not necessary for the entire team to work with every technology used, but we'd like to know who worked with what.

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

General Information & Licensing

Code Repository	git@github.com:weidajiang/CSE312.git
License Type	BSD(flask), MIT(flask_socketIO)
License Description	According to the site, this basically means that as long as the copyright and name of Flask still exists, and the disclaimer exists, we can use it however we want. This means we can actually reuse it any way we give them credit.
License Restrictions	We can actually reuse it anyway we give them credit.
Who worked with this?	Whole team

Flask - app.run (Start TCP Server)

Purpose

- Instead of using what we did in the homework to hold the TCP server, the flask uses WSGIServer to hold the server.

```
host = "0.0.0.0"

port = 8000

with socketserver.ThreadingTCPServer((host,port),MyTCPHandler) as server:

        server.serve_forever()
```

Magic

When we start the Flask project, the main entry is

```
app = Flask(__name__)
if __name__ == '__main__':
    app.run()
```

Inside the run() flask/app.py from lines range (871 - 925), we see the host, and port.

Line 920: `run_simple(t.cast(str, host), port, self, **options)`

run_simple() from werkzeug/serving.py lines (1049 - 1059), a WSGI server, calls make_server() and serve_forever() to start the server.

make_server():

```
    """
    if threaded and processes > 1:
        raise ValueError("Cannot have a multi-thread and multi-process server.")

    if threaded:
        return ThreadedWSGIServer(
            host, port, app, request_handler, passthrough_errors, ssl_context, fd=fd
        )

    if processes > 1:
        return ForkingWSGIServer(
            host,
            port,
            app,
            processes,
            request_handler,
            passthrough_errors,
            ssl_context,
            fd=fd,
        )

    return BaseWSGIServer(
        host, port, app, request_handler, passthrough_errors, ssl_context, fd=fd
    )
```

make_server will depend on the number of thread/processor to return WSGI server. By default, that returns BaseWSGIServer(extend from HTTPServer).

serve_forever():

```
    def serve_forever(self, poll_interval: float = 0.5) -> None:
        try:
            super().serve_forever(poll_interval=poll_interval)
        except KeyboardInterrupt:
            pass
        finally:
            self.server_close()
```

serve_forever call parent's(HTTPServer) serve_forever() function.

Flask - Basic request parameters

Purpose

- The Request has the ability to read the user input and translate the data into usable data for our code. With this ability we can use Request to help us to retrieve more accurate and security data from the user.

- After Request is called the data we get from the users will be translated into datas with MIME types, headers, x-content-type-options and Path management/routing accordingly, and by having those properties it can help us by letting us have a more effect way of accessing the data.

- The function will be activated when users send the request in. We can get those data we want such as MIME types, headers, x-content-type-options and Path management/routing etc by getting into the _SansIORequest to find the base of the function.
- The request function located in app.py in Flask, refers to the RequestBase and inherits from _SansIORequest. Then call the codes from line 260-267 to get the content_type:

```
260     content_type = header_property[str](
261         "Content-Type",
262         doc="""The Content-Type entity-header field indicates the media
263         type of the entity-body sent to the recipient or, in the case of
264         the HEAD method, the media type that would have been sent had
265         the request been a GET.""",
266         read_only=True,
267     )
```

From line 270-286 to get content length:

```
270     def content_length(self) -> t.Optional[int]:
271         """The Content-Length entity-header field indicates the size of the
272         entity-body in bytes or, in the case of the HEAD method, the size of
273         the entity-body that would have been sent had the request been a
274         GET.
275         """
276         if self.headers.get("Transfer-Encoding", "") == "chunked":
277             return None
278
279         content_length = self.headers.get("Content-Length")
280         if content_length is not None:
281             try:
282                 return max(0, int(content_length))
283             except (ValueError, TypeError):
284                 pass
285
286         return None
```

From line 351-358 to get mimetype:

```
351     def mimetype(self) -> str:
352         """Like :attr:`content_type`, but without parameters (eg, without
353         charset, type etc.) and always lowercase.  For example if the con
354         type is ``text/HTML; charset=utf-8`` the mimetype would be
355         ``'text/html'``.
356         """
357         self._parse_content_type()
358         return self._parsed_content_type[0].lower()
359
```

From line 119-150 to get headers, methods, path, etc:

```python
    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``.
        self.scheme = scheme
        #: The address of the server. ``(host, port)``, ``(path, None)``
        #: for unix sockets, or ``None`` if not known.
        self.server = server
        #: The prefix that the application is mounted under, without a
        #: trailing slash. :attr:`path` comes after this.
```

```python
        #: trailing slash. :attr:`path` comes after this.
        self.root_path = root_path.rstrip("/")
        #: The path part of the URL after :attr:`root_path`. This is the
        #: path used for routing within the application.
        self.path = "/" + path.lstrip("/")
        #: The part of the URL after the "?". This is the raw value, use
        #: :attr:`args` for the parsed values.
        self.query_string = query_string
        #: The headers received with the request.
        self.headers = headers
        #: The address of the client sending the request.
        self.remote_addr = remote_addr
```

Flask - Route

Purpose

- To map URL and processing function.
- When the browser client sends a URL request to the web server, the routing in the server will immediately find the corresponding function to process.
- Example : (http://127.0.0.1:5000/), this url will take us the hello_word function

```
@app.route('/')
def hello_world():
  return 'Hello World!'
```

Magic ★★。°・° ☽ °⌒✦。°★≡☆

Here is the sample code for registering a path.

```
6   @app.route('/')
7   def hello_world():  # put application's code here
8       print(request.args.get('data',type=str))
9       return 'Hello World!'
10
```

In Flask, they use decorator to add a url rule to the requestContext (flask.scaffold.py)

```
414     def route(self, rule: str, **options: t.Any) -> t.Callable[[F], F]:
415         """Decorate a view function to register it with the given URL
416         rule and options. Calls :meth:`add_url_rule`, which has more
417         details about the implementation.
418
419         .. code-block:: python
420
421             @app.route("/")
422             def index():
423                 return "Hello, World!"
424
425         See :ref:`url-route-registrations`.
426
427         The endpoint name for the route defaults to the name of the view
428         function if the ``endpoint`` parameter isn't passed.
429
430         The ``methods`` parameter defaults to ``["GET"]``. ``HEAD`` and
431         ``OPTIONS`` are added automatically.
432
433         :param rule: The URL rule string.
434         :param options: Extra options passed to the
435             :class:`~werkzeug.routing.Rule` object.
436         """
437
438         def decorator(f: F) -> F:
439             endpoint = options.pop("endpoint", None)
440             self.add_url_rule(rule, endpoint, f, **options)
441             return f
442
443         return decorator
```

Inside the add_url_rule (flask/app.py 1038 - 1094). In line 1086,

```python
        # Add the required methods now.
        methods |= required_methods

        rule = self.url_rule_class(rule, methods=methods, **options)
        rule.provide_automatic_options = provide_automatic_options  # type: ignore

        self.url_map.add(rule)
```

flask will set the route rule(Werkzeug.routing.Rule) to the url_map(werkzeug.routing:Map) endpoint, and view_function(python dict)

In the flask.app (1755 - 1790), this creates a url adapter based on how many endpoints we create in the project.

```python
    def create_url_adapter(
        self, request: t.Optional[Request]
    ) -> t.Optional[MapAdapter]:
        """Creates a URL adapter for the given request. The URL adapter
        is created at a point where the request context is not yet set
        up so the request is passed explicitly.

        .. versionadded:: 0.6

        .. versionchanged:: 0.9
           This can now also be called without a request object when the
           URL adapter is created for the application context.

        .. versionchanged:: 1.0
            :data:`SERVER_NAME` no longer implicitly enables subdomain
            matching. Use :attr:`subdomain_matching` instead.
        """
        if request is not None:
            # If subdomain matching is disabled (the default), use the
            # default subdomain in all cases. This should be the default
            # in Werkzeug but it currently does not have that feature.
            if not self.subdomain_matching:
                subdomain = self.url_map.default_subdomain or None
            else:
                subdomain = None

            return self.url_map.bind_to_environ(
                request.environ,
                server_name=self.config["SERVER_NAME"],
                subdomain=subdomain,
            )
        # We need at the very least the server name to be set for this
```

The main loinc of the routing is implemented in the werkzeug module. In

werkzeug.routing.py line (2211 - 2334), build function binds the routing path to a specific endpoint through m.bind and matches the URL through urls.match. Under normal circumstances, the corresponding endpoint name and parameter dictionary are returned, and redirection or 404 exceptions may be reported.

```
2211    def build(
2212        self,
2213        endpoint: str,
2214        values: t.Optional[t.Mapping[str, t.Any]] = None,
2215        method: t.Optional[str] = None,
2216        force_external: bool = False,
2217        append_unknown: bool = True,
2218        url_scheme: t.Optional[str] = None,
2219    ) -> str:
2220        """Building URLs works pretty much the other way round.  Instead of
2221        `match` you call `build` and pass it the endpoint and a dict of
2222        arguments for the placeholders.
2223
2224        The `build` function also accepts an argument called `force_external`
2225        which, if you set it to `True` will force external URLs. Per default
2226        external URLs (include the server name) will only be used if the
2227        target URL is on a different subdomain.
2228
2229        >>> m = Map([
2230        ...     Rule('/', endpoint='index'),
2231        ...     Rule('/downloads/', endpoint='downloads/index'),
2232        ...     Rule('/downloads/<int:id>', endpoint='downloads/show')
2233        ... ])
2234        >>> urls = m.bind("example.com", "/")
2235        >>> urls.build("index", {})
2236        '/'
2237        >>> urls.build("downloads/show", {'id': 42})
2238        '/downloads/42'
2239        >>> urls.build("downloads/show", {'id': 42}, force_external=True)
2240        'http://example.com/downloads/42'
```

Then, in the match function, werkzeug use regular expression th match the path.

```python
        # a \n by WSGI.
        regex = rf"^{''.join(regex_parts)}{tail}$\Z"
        self._regex = re.compile(regex)

    def match(
        self, path: str, method: t.Optional[str] = None
    ) -> t.Optional[t.MutableMapping[str, t.Any]]:
        """Check if the rule matches a given path. Path is a string in the
        form ``"subdomain|/path"`` and is assembled by the map.  If
        the map is doing host matching the subdomain part will be the host
        instead.

        If the rule matches a dict with the converted values is returned,
        otherwise the return value is `None`.

        :internal:
        """
        if not self.build_only:
            require_redirect = False

            m = self._regex.search(path)
            if m is not None:
                groups = m.groupdict()
                # we have a folder like part of the url without a trailing
                # slash and strict slashes enabled. raise an exception that
                # tells the map to redirect to the same url but with a
                # trailing slash
                if (
                    self.strict_slashes
                    and not self.is_leaf
                    and not groups.pop("__suffix__")
                    and (
                        method is None or self.methods is None or method in self.methods
                    )
                ):
                    path += "/"
                    require_redirect = True
                # if we are not in strict slashes mode we have to remove
                # a __suffix__
                elif not self.strict_slashes:
                    del groups["__suffix__"]
```

Rule › match()

Flask - make_response(content)

Purpose

- It will return a response object, with default status code and header(if not given). The argument can be considered as a body.

Magic ★★ ° ˙ ° ☽ ° ⌒ ° ★ ≡ ★

- The function will generate the response we usually send after we receive a request from a client, it contains all the elements such as, header, status code, body ect., just as we did in previous homework. The content you provide will be the body of the response, you can also provide information about status code, content-type, ect, if you are not given this information, the function will use the default value.

- The make_response function is mainly implemented in app.py located in the flask folder. In the function, the input will be cast into Reponse(a class) then return that class in line 1737:

```
rv = t.cast(Response, rv)
```

- When you get into the Response class, where you can find the file in werkzeug/wrappers/response.py, the class extend_SansI0Response. In the class of _SansI0Response you will realize in lines 88-108, there are some default values that have been set for you, such as charset, status code, MIME types, cookie size, and the class header(represent the response header). You can also provide information to the default value, which you can find in the init method of the response class. Back the Response class, the information of the body will be generated using the set_data function.

Flask - render_template

Purpose

- In order to send the html file back to the frontend and use the template engine(we learn in the class), we decided to use the render_template function in the flask framework.

Magic ★★ ° ˚ ☽ ˚ ⌒ ° ★ ⚞ ★

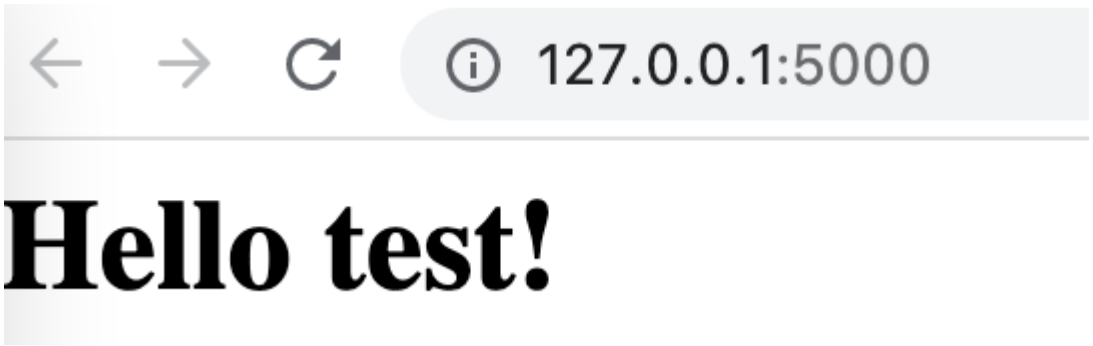
Usage:

```
@app.route('/')
def hello_world():
    return render_template('index.html',name='test')
```

lib/.../app.py templating.py index.html ap

```
<html>
{% if name %}
  <h4>Hello {{ name }}!</h4>
{% else %}
  <h1>Hello</h1>
{% endif %}
</html>
```

In the hello_world function (path:/), this will load the index.html under the template folder, and assign name to a value “test”. In the index.html, based on the template engine, the home page will replace the name with actual value.



Magic behind the scene:
Flask uses Jinja2 as its template engine. In flask/templating.py, the render_template function provides a very detailed comment, which will render a template from the template folder with given context.

```python
def render_template(
    template_name_or_list: t.Union[str, Template, t.List[t.Union[str, Template]]],
    **context: t.Any
) -> str:
    """Renders a template from the template folder with the given
    context.

    :param template_name_or_list: the name of the template to be
                                  rendered, or an iterable with template names
                                  the first one existing will be rendered
    :param context: the variables that should be available in the
                    context of the template.
    """
    ctx = _app_ctx_stack.top
    ctx.app.update_template_context(context)
    return _render(
        ctx.app.jinja_env.get_or_select_template(template_name_or_list),
        context,
        ctx.app,
    )
```

In the Flask (flask/app.py lines 98 - 520), line 392 specifies the source folder.

```python
    def __init__(
        self,
        import_name: str,
        static_url_path: t.Optional[str] = None,
        static_folder: t.Optional[t.Union[str, os.PathLike]] = "static",
        static_host: t.Optional[str] = None,
        host_matching: bool = False,
        subdomain_matching: bool = False,
        template_folder: t.Optional[str] = "templates",
        instance_path: t.Optional[str] = None,
        instance_relative_config: bool = False,
        root_path: t.Optional[str] = None,
    ):
```

we aslo see that Flask object assign a jinja environment. (line 206)

```python
    #: The class that is used for the Jinja environment.
    #:
    #: .. versionadded:: 0.11
    jinja_environment = Environment
```

In Jinja2/Environment.py, line 1074 cals select_template.

```python
    @internalcode
    def get_or_select_template(
        self,
        template_name_or_list: t.Union[
            str, "Template", t.List[t.Union[str, "Template"]]
        ],
        parent: t.Optional[str] = None,
        globals: t.Optional[t.MutableMapping[str, t.Any]] = None,
    ) -> "Template":
        """Use :meth:`select_template` if an iterable of template names
        is given, or :meth:`get_template` if one name is given.

        .. versionadded:: 2.3
        """
        if isinstance(template_name_or_list, (str, Undefined)):
            return self.get_template(template_name_or_list, parent, globals)
        elif isinstance(template_name_or_list, Template):
            return template_name_or_list
        return self.select_template(template_name_or_list, parent, globals)
```

In the same file, line 1003 - 1054, calls load_template.

```python
        .. versionadded:: 2.3
        """
        if isinstance(names, Undefined):
            names._fail_with_undefined_error()

        if not names:
            raise TemplatesNotFound(
                message="Tried to select from an empty list of templates."
            )

        for name in names:
            if isinstance(name, Template):
                return name
            if parent is not None:
                name = self.join_path(name, parent)
            try:
                return self._load_template(name, globals)
            except (TemplateNotFound, UndefinedError):
                pass
        raise TemplatesNotFound(names)  # type: ignore
```

Here is the load_template function

```python
    @internalcode
    def _load_template(
        self, name: str, globals: t.Optional[t.MutableMapping[str, t.Any]]
    ) -> "Template":
        if self.loader is None:
            raise TypeError("no loader for this environment specified")
        cache_key = (weakref.ref(self.loader), name)
        if self.cache is not None:
            template = self.cache.get(cache_key)
            if template is not None and (
                not self.auto_reload or template.is_up_to_date
            ):
                # template.globals is a ChainMap, modifying it will only
                # affect the template, not the environment globals.
                if globals:
                    template.globals.update(globals)

                return template

        template = self.loader.load(self, name, self.make_globals(globals))

        if self.cache is not None:
            self.cache[cache_key] = template
        return template
```

The load_template() method will first check whether there is a cache, if the cache is available, use the cache; if the cache is not available, use the loader to load the template.

Also, a lot of the functions are implemented in the jinja2/parser.py function, such as

parse_if, parse_tuple, parse_list, parse_statement and so on. We can use these keywords on the html page.

Flask - socketIO

Purpose

- The purpose of using the library is to build up the websocket handshake, after the handshake the clients and server can send messages to each other through the flask_socketio.
- Live chat interaction.

Magic ★★ ° ˚ ☽ ° ⌒ ° ★ ≡ ★

- Go through the init file in the flask_socketio, you will realize that the flask_soketio was based on the socketIO library(the socketio encapsulates Websocket, Ajax into a unified communication interface). The send method of the flask_socketio takes a message as a second input(kwargs) which will send the WebSocket message to all connecting clients. Before it sends, it will check whether the message is a JSON blob or not. Then extract the information such as callback(the callback function in the front-end), broadcast(whether to send the message to all the clients or not), to(send the specified clients), etc. And finally use the send all these information with socketio.send method. Also, there is an emit function which works same as send function, but it can work for events.

```
    def on(self, message, namespace=None):
        """Decorator to register a SocketIO event handler.

        This decorator must be applied to SocketIO event handlers. Example::

            @socketio.on('my event', namespace='/chat')
            def handle_my_custom_event(json):
                print('received json: ' + str(json))

        :param message: The name of the event. This is normally a user defined
                        string, but a few event names are already defined. Use
                        ``'message'`` to define a handler that takes a string
                        payload, ``'json'`` to define a handler that takes a
                        JSON blob payload, ``'connect'`` or ``'disconnect'``
                        to create handlers for connection and disconnection
                        events.
        :param namespace: The namespace on which the handler is to be
                          registered. Defaults to the global namespace.
        """
        namespace = namespace or '/'

```

```python
843 def send(message, **kwargs):
844     """Send a SocketIO message.
845
846     This function sends a simple SocketIO message to one or more connected
847     clients. The message can be a string or a JSON blob. This is a simpler
848     version of ``emit()``, which should be preferred. This is a function that
849     can only be called from a SocketIO event handler.
850
851     :param message: The message to send, either a string or a JSON blob.
852     :param json: ``True`` if ``message`` is a JSON blob, ``False``
853                  otherwise.
854     :param namespace: The namespace under which the message is to be sent.
855                       Defaults to the namespace used by the originating event.
856                       An empty string can be used to use the global namespace.
857     :param callback: Callback function to invoke with the client's
858                  acknowledgement.
859     :param broadcast: ``True`` to send the message to all connected clients, or
860                       ``False`` to only reply to the sender of the originating
861                       event.
862     :param to: Send the message to all the users in the given room. If this
```

```python
862     :param to: Send the message to all the users in the given room. If this
863                argument is not set and ``broadcast`` is ``False``, then the
864                message is sent only to the originating user.
865     :param include_self: ``True`` to include the sender when broadcasting or
866                          addressing a room, or ``False`` to send to everyone
867                          but the sender.
868     :param skip_sid: The session id of a client to ignore when broadcasting
869                      or addressing a room. This is typically set to the
870                      originator of the message, so that everyone except
871                      that client receive the message. To skip multiple sids
872                      pass a list.
873     :param ignore_queue: Only used when a message queue is configured. If
874                          set to ``True``, the event is emitted to the
875                          clients directly, without going through the queue.
876                          This is more efficient, but only works when a
877                          single server process is used, or when there is a
878                          single addressee. It is recommended to always leave
879                          this parameter with its default value of ``False``.
880     """
881     json = kwargs.get('json', False)
882     if 'namespace' in kwargs:
883         namespace = kwargs['namespace']
```

```python
    if 'namespace' in kwargs:
        namespace = kwargs['namespace']
    else:
        namespace = flask.request.namespace
    callback = kwargs.get('callback')
    broadcast = kwargs.get('broadcast')
    to = kwargs.pop('to', kwargs.pop('room', None))
    if to is None and not broadcast:
        to = flask.request.sid
    include_self = kwargs.get('include_self', True)
    skip_sid = kwargs.get('skip_sid')
    ignore_queue = kwargs.get('ignore_queue', False)

    socketio = flask.current_app.extensions['socketio']
    return socketio.send(message, json=json, namespace=namespace, to=to,
                         include_self=include_self, skip_sid=skip_sid,
                         callback=callback, ignore_queue=ignore_queue)

```

```python
def disconnect(sid=None, namespace=None, silent=False):
    """Disconnect the client.

    This function terminates the connection with the client. As a result of
    this call the client will receive a disconnect event. Example::

        @socketio.on('message')
        def receive_message(msg):
            if is_banned(session['username']):
                disconnect()
            else:
                # ...

    :param sid: The session id of the client. If not provided, the client is
                obtained from the request context.
    :param namespace: The namespace for the room. If not provided, the
                      namespace is obtained from the request context.
    :param silent: this option is deprecated.
    """
    socketio = flask.current_app.extensions['socketio']
    sid = sid or flask.request.sid
    namespace = namespace or flask.request.namespace
    return socketio.server.disconnect(sid, namespace=namespace)
```

```python
    def emit(self, event, *args, **kwargs):
        """Emit a server generated SocketIO event.

        This function emits a SocketIO event to one or more connected clients.
        A JSON blob can be attached to the event as payload. This function can
        be used outside of a SocketIO event context, so it is appropriate to
        use when the server is the originator of an event, outside of any
        client context, such as in a regular HTTP request handler or a
        background task. Example::

            @app.route('/ping')
            def ping():
                socketio.emit('ping event', {'data': 42}, namespace='/chat')

        :param event: The name of the user event to emit.
        :param args: A dictionary with the JSON data to send as payload.
        :param namespace: The namespace under which the message is to be sent.
                          Defaults to the global namespace.
        :param to: Send the message to all the users in the given room. If
                   this parameter is not included, the event is sent to all
                   connected users.
        :param include_self: ``True`` to include the sender when broadcasting
```

```python
        :param include_self: ``True`` to include the sender when broadcasting
                             or addressing a room, or ``False`` to send to
                             everyone but the sender.
        :param skip_sid: The session id of a client to ignore when broadcasting
                         or addressing a room. This is typically set to the
                         originator of the message, so that everyone except
                         that client receive the message. To skip multiple sids
                         pass a list.
        :param callback: If given, this function will be called to acknowledge
                         that the client has received the message. The
                         arguments that will be passed to the function are
                         those provided by the client. Callback functions can
                         only be used when addressing an individual client.
        """
        namespace = kwargs.pop('namespace', '/')
        to = kwargs.pop('to', kwargs.pop('room', None))
        include_self = kwargs.pop('include_self', True)
        skip_sid = kwargs.pop('skip_sid', None)
        if not include_self and not skip_sid:
            skip_sid = flask.request.sid
        callback = kwargs.pop('callback', None)
```