Flask Microframework Report:

What this framework accomplishes for us:

- Handles HTTP requests and responses.
- Provides methods to render HTML templates (however it is not an HTML template itself. Flask is built as a wrapper to the Jinja template engine)
- Capture form responses and store the data so that they may be called upon with functions.

flask.Flask:

- This class is the central object of our application.
- It allows us to configure the app with important attributes such as the project's secret key, environment, and debug mode settings. We set routes and url rules to a Flask object and are able to use it to create socket requests (talked about in another report).
- A Flask object takes a Scaffold object as a parameter, which contain the methods that define how we're able to set routes for our project.
    - From the flask/app.py:

```
#: If a secret key is set, cryptographic components can use this to
#: sign cookies and other things. Set this to a complex random value
#: when you want to use the secure cookie for instance.
#:
#: This attribute can also be configured from the config with the
#: :data:`SECRET_KEY` configuration key. Defaults to ``None``.
secret_key = ConfigAttribute("SECRET_KEY")

#: The secure cookie uses this for the name of the session cookie.
#:
#: This attribute can also be configured from the config with the
#: ``SESSION_COOKIE_NAME`` configuration key.  Defaults to ```'session'```
session_cookie_name = ConfigAttribute("SESSION_COOKIE_NAME")
```

```
class ConfigAttribute:
    """Makes an attribute forward to the config"""

    def __init__(self, name, get_converter=None):
        self.__name__ = name
        self.get_converter = get_converter

    def __get__(self, obj, type=None):
        if obj is None:
            return self
        rv = obj.config[self.__name__]
        if self.get_converter is not None:
            rv = self.get_converter(rv)
        return rv

    def __set__(self, obj, value):
        obj.config[self.__name__] = value
```

[src/flask/scaffold.py]

add_url_rule (line 439):

- This function is used to register a rule for routing incoming requests. The route function accomplishes the same thing, but can be used as a decorator to a view function.

```
@setupmethod
def add_url_rule(
    self,
    rule: str,
    endpoint: t.Optional[str] = None,
    view_func: t.Optional[t.Callable] = None,
    provide_automatic_options: t.Optional[bool] = None,
    **options: t.Any,
) -> t.Callable:
```

route (line 407):

- This route function (commonly seen as *@app.route(rule)*) is used to register a view function with the passed url rule (and optional options parameter).

```
def route(self, rule: str, **options: t.Any) -> t.Callable:
    """Decorate a view function to register it with the given URL
    rule and options. Calls :meth:`add_url_rule`, which has more
    details about the implementation.

    .. code-block:: python

        @app.route("/")
        def index():
            return "Hello, World!"

    See :ref:`url-route-registrations`.

    The endpoint name for the route defaults to the name of the view
    function if the ``endpoint`` parameter isn't passed.

    The ``methods`` parameter defaults to ``["GET"]``. ``HEAD`` and
    ``OPTIONS`` are added automatically.

    :param rule: The URL rule string.
    :param options: Extra options passed to the
        :class:`~werkzeug.routing.Rule` object.
    """

    def decorator(f: t.Callable) -> t.Callable:
        endpoint = options.pop("endpoint", None)
        self.add_url_rule(rule, endpoint, f, **options)
        return f

    return decorator
```

flask.request:

- This is what allows us to take in data entered in forms and turns it into python variables that we can store & interact with. This is useful throughout the app, whenever the user wants to make a post, send a message, edit a post, etc.
- This is accomplished through the _defualt_template_ctx_processor, where we can see requests & sessions being processed in flask/templating.py (line 11)

```
11  def _default_template_ctx_processor():
12      """Default template context processor.  Injects `request`,
13      `session` and `g`.
14      """
15      reqctx = _request_ctx_stack.top
16      appctx = _app_ctx_stack.top
17      rv = {}
18      if appctx is not None:
19          rv["g"] = appctx.g
20      if reqctx is not None:
21          rv["request"] = reqctx.request
22          rv["session"] = reqctx.session
23      return rv
```

```
def _default_template_ctx_processor():
    """Default template context processor.  Injects `request`,
    `session` and `g`.
    """
    reqctx = _request_ctx_stack.top
    appctx = _app_ctx_stack.top
    rv = {}
    if appctx is not None:
        rv["g"] = appctx.g
    if reqctx is not None:
        rv["request"] = reqctx.request
        rv["session"] = reqctx.session
    return rv
```

flask.render_template:

- Render_template is a powerful tool that allows us to inject logic & js commands into our html templates, as well as pass it variables for easier creation of dynamic html files.
- One great use of this in our app is for the edit post button, which only appears when you are the one that created the post. So it is sent the struct that holds the post data and compares the user of that post to the current user stored in session storage.
- In flask/template.py, we can see the render_template function (115) which takes in the html template, and then updates it with actual html code to replace the templating that is used in update_template_context in flask/app.py (712)

```
def update_template_context(self, context):
    """Update the template context with some commonly used variables.
    This injects request, session, config and g into the template
    context as well as everything template context processors want
    to inject.  Note that the as of Flask 0.6, the original values
    in the context will not be overridden if a context processor
    decides to return a value with the same key.

    :param context: the context as a dictionary that is updated in place
                    to add extra variables.
    """
    funcs = self.template_context_processors[None]
    reqctx = _request_ctx_stack.top
    if reqctx is not None:
        for bp in self._request_blueprints():
            if bp in self.template_context_processors:
                funcs = chain(funcs, self.template_context_processors[bp])
    orig_ctx = context.copy()
    for func in funcs:
        context.update(func())
    # make sure the original values win.  This makes it possible to
    # easier add new variables in context processors without breaking
    # existing views.
    context.update(orig_ctx)

def make_shell_context(self):
    """Returns the shell context for an interactive shell for this
    application.  This runs all the registered shell context
    processors.

    .. versionadded:: 0.11
    """
    rv = {"app": self, "g": g}
    for processor in self.shell_context_processors:
        rv.update(processor())
    return rv
```

flask.redirect:

- Redirect allows us to easily redirect the user from one route defined from flask.Flask to another. This changes what would be a complex redirect into a single-line return statement
- This method relies entirely on the redirect method defined in the Werkzeug WSGI web application library, which is defined in src/werkzeug/utils.py (line 525) in their public github repository.
- The method takes a url location as a string, a valid response code (as documented in the function), and a Response object.
- redirect returns a Werkzeug Response object, which is composed of HTML to display should the redirect not automatically work, the response code passed to the function, a mimetype of "text/html", and a location header set to the location passed to the function.

```
def redirect(
    location: str, code: int = 302, Response: t.Optional[t.Type["Response"]] = None
) -> "Response":
    """Returns a response object (a WSGI application) that, if called,
    redirects the client to the target location. Supported codes are
    301, 302, 303, 305, 307, and 308. 300 is not supported because
    it's not a real redirect and 304 because it's the answer for a
    request with a request with defined If-Modified-Since headers.

    .. versionadded:: 0.6
       The location can now be a unicode string that is encoded using
       the :func:`iri_to_uri` function.

    .. versionadded:: 0.10
        The class used for the Response object can now be passed in.

    :param location: the location the response should redirect to.
    :param code: the redirect status code. defaults to 302.
    :param class Response: a Response class to use when instantiating a
        response. The default is :class:`werkzeug.wrappers.Response` if
        unspecified.
    """
    import html

    if Response is None:
        from .wrappers import Response  # type: ignore

    display_location = html.escape(location)
    if isinstance(location, str):
        # Safe conversion is necessary here as we might redirect
        # to a broken URI scheme (for instance itms-services).
        from .urls import iri_to_uri

        location = iri_to_uri(location, safe_conversion=True)
    response = Response(  # type: ignore
        '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">\n'
        "<title>Redirecting...</title>\n"
        "<h1>Redirecting...</h1>\n"
        "<p>You should be redirected automatically to target URL: "
        f'<a href="{html.escape(location)}">{display_location}</a>. If'
        " not click the link.",
        code,
        mimetype="text/html",
    )
    response.headers["Location"] = location
    return response
```

Flask.current_app / session:

- This is another way flask allows us to store variables & data within flask. In our app we use this to store the upload_folder in a way where we can access it from anywhere
- WIthin the flask code we can see that current app/session is created as a variable that we can then access through it in flask/globals.py (line 48)

```
46  _request_ctx_stack = LocalStack()
47  _app_ctx_stack = LocalStack()
48  current_app = LocalProxy(_find_app)
49  request = LocalProxy(partial(_lookup_req_object, "request"))
50  session = LocalProxy(partial(_lookup_req_object, "session"))
51  g = LocalProxy(partial(_lookup_app_object, "g"))
```

Licensing:

Flask is licensed under a modified version of the BSD 2-Clause license (with the addition of a third clause). The license allows redistribution and use of Flask whether with or without any modifications as long as three clauses are met. The first clause requires that any redistributions of source code must retain the licence's copyright notice, the three conditions, and the disclaimer that follows them. The second clause requires that redistributions in binary form must also reproduce the licence's copyright notice, the three conditions, and the disclaimer that follows them. The last clause is that neither the names of the copyright holder or its contributors may be used to endorse or promote products derived from Flask without prior written permission. There is a disclaimer that follows these clauses which essentially explains that Flask comes "as is" and that the copyright holder or any contributors are not responsible for any harm or damages caused by using it.