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 TCP Connections]

General Information & Licensing

Code Repository	https://github.com/pallets/flask
License Type	BSD-3-Clause
License Description	• This license allows unlimited freedom with the software as long as you include the copyright and license notice
License Restrictions	• Redistributions of source code must retain the copyright notice, the list of conditions and disclaimer. • Redistributions in binary form must reproduce the copyright notice, list of conditions, and the 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.

Magic

Dispel the magic of this technology. Replace this text with some that answer the following questions for the above tech:

- How does this technology do what it does? Please explain this in detail, starting from after the TCP socket is created
- 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 will likely grow beyond the page

Flask is built on top of the Python standard library *socket* module which provides ways to interact with TCP sockets. Specifically, Flask actually uses another framework(by the same creators: 'pallets') werkzeug, which creates a WSGI server to create the connection. After creating a flask app, calling run() starts this server automatically. Now, let's look at the specific code.

Let's start at the beginning, when we call the run() method to start the app.

In the flask/cli.py module where the WSGI server is created we can see the imports on lines 16-18 from the werkzeug framework.

```
16    from werkzeug import run_simple
17    from werkzeug.serving import is_running_from_reloader
18    from werkzeug.utils import import_string
```

We see later on in flask/cli.py that *run_simple* is called to start the WSGI server on lines 923-933.

```
923         run_simple(
924             host,
925             port,
926             app,
927             use_reloader=reload,
928             use_debugger=debugger,
929             threaded=with_threads,
930             ssl_context=cert,
931             extra_files=extra_files,
932             exclude_patterns=exclude_patterns,
933         )
```

These are flags that can be set but otherwise default to certain values, for example host defaults to http://127.0.0.1/ and port defaults to 5000.

Now to see what happens after run_simple we must jump to the repository for werkzeug, specifically the serving.py module. This is where we will see the built-in Python *socket* module utilized.

In serving.py, let's go to lines 938-1100 to see the how *run_simple* is defined*:*

```
938 def run_simple(
939     hostname: str,
940     port: int,
941     application: "WSGIApplication",
942     use_reloader: bool = False,
943     use_debugger: bool = False,
944     use_evalex: bool = True,
945     extra_files: t.Optional[t.Iterable[str]] = None,
946     exclude_patterns: t.Optional[t.Iterable[str]] = None,
947     reloader_interval: int = 1,
948     reloader_type: str = "auto",
949     threaded: bool = False,
950     processes: int = 1,
951     request_handler: t.Optional[t.Type[WSGIRequestHandler]] = None,
952     static_files: t.Optional[t.Dict[str, t.Union[str, t.Tuple[str, str]]]] = None,
953     passthrough_errors: bool = False,
954     ssl_context: t.Optional[_TSSLContextArg] = None,
955 ) -> None:
```

Again we see hostname and port as parameters which are carried along from the beginning when we first started the app with run().

Next we skip past some comments explaining version history to line 1068:

```
1068        srv = make_server(
1069            hostname,
1070            port,
1071            application,
1072            threaded,
1073            processes,
1074            request_handler,
1075            passthrough_errors,
1076            ssl_context,
1077            fd=fd,
1078        )
1079        srv.socket.set_inheritable(True)
1080        os.environ["WERKZEUG_SERVER_FD"] = str(srv.fileno())
1081
1082        if not is_running_from_reloader():
1083            srv.log_startup()
1084            _log("info", _ansi_style("Press CTRL+C to quit", "yellow"))
1085
1086        if use_reloader:
1087            from ._reloader import run_with_reloader
1088
1089            try:
1090                run_with_reloader(
1091                    srv.serve_forever,
1092                    extra_files=extra_files,
1093                    exclude_patterns=exclude_patterns,
1094                    interval=reloader_interval,
1095                    reloader_type=reloader_type,
1096                )
1097            finally:
1098                srv.server_close()
1099        else:
1100            srv.serve_forever()
```

Here we see srv is created using *make_serve*r which is actually a *BaseWSGIServer* that serves forever. Also note that *reloader* is primarily used in development and is turned on when the apps debug=True. The *reloader* periodically checks if the code has been modified and restarts the server if changes have occurred.

Here we see the definition of make_sever which does indeed return a *BaseWSGIServer* on line 884:

```
884  def make_server(
885      host: str,
886      port: int,
887      app: "WSGIApplication",
888      threaded: bool = False,
889      processes: int = 1,
890      request_handler: t.Optional[t.Type[WSGIRequestHandler]] = None,
891      passthrough_errors: bool = False,
892      ssl_context: t.Optional[_TSSLContextArg] = None,
893      fd: t.Optional[int] = None,
894  ) -> BaseWSGIServer:
895      """Create an appropriate WSGI server instance based on the value of
896      ``threaded`` and ``processes``.
897
898      This is called from :func:`run_simple`, but can be used separately
899      to have access to the server object, such as to run it in a separate
900      thread.
901
902      See :func:`run_simple` for parameter docs.
903      """
```

Again we see the host and port passed along, as well as additional parameters.

Now let's look at what a *BaseWSGIServer* is on like 682:

```
682  class BaseWSGIServer(HTTPServer):
683      """A WSGI server that that handles one request at a time.
684
685      Use :func:`make_server` to create a server instance.
686      """
687
688      multithread = False
689      multiprocess = False
690      request_queue_size = LISTEN_QUEUE
691      allow_reuse_address = True
692
693      def __init__(
694          self,
695          host: str,
696          port: int,
697          app: "WSGIApplication",
698          handler: t.Optional[t.Type[WSGIRequestHandler]] = None,
699          passthrough_errors: bool = False,
700          ssl_context: t.Optional[_TSSLContextArg] = None,
701          fd: t.Optional[int] = None,
702      ) -> None:
```

This is a long class that spans lines 682-845, but first again we see the host and port passed along from the very beginning in the constructor. Let's look at that *serve_forever()* method we saw called at the end of *run_simple()* on like 795.

```
795        def serve_forever(self, poll_interval: float = 0.5) -> None:
796            try:
797                super().serve_forever(poll_interval=poll_interval)
798            except KeyboardInterrupt:
799                pass
800            finally:
801                self.server_close()
```

Here we see an infinite loop(only interrupted by KeyboardInterrupt) that constantly listens on the port.

But we see super() called here so let's go back to that class constructor and look at what a *HTTPserver* is to see what *serve_forever* really does.

To do that we must go to the http.server module in the Python standard library and look at the class definition of *HTTPserver*.

```
130    class HTTPServer(socketserver.TCPServer):
131
132        allow_reuse_address = 1    # Seems to make sense in testing environment
133
134        def server_bind(self):
135            """Override server_bind to store the server name."""
136            socketserver.TCPServer.server_bind(self)
137            host, port = self.server_address[:2]
138            self.server_name = socket.getfqdn(host)
139            self.server_port = port
```

On line 130 we see the class definition,

Finally, we have reach the socketserver library as seen in our cse312 hws.