Code Analysis Report

Analyzed Code:

```
import typing
from datetime import datetime, timedelta, timezone
from json.decoder import JSONDecodeError
from secrets import token_urlsafe
from typing import Any, Dict, Optional, Tuple

from flask import Flask, Request, Response
from flask import current_app as app
from flask.sessions import SessionInterface as FlaskSessionInterface
from flask.sessions import SessionMixin, session_json_serializer
from flask_babel import gettext
from itsdangerous import BadSignature, URLSafeTimedSerializer
from models import Journalist
from redis import Redis
from werkzeug.datastructures import CallbackDict

class ServerSideSession(CallbackDict, SessionMixin):
"""Baseclass for server-side based sessions."""

def __init__(self, sid: str, token: str, lifetime: int = 0, initial: Any =
None) -> None:
def on_update(self: ServerSideSession) -> None:
self.modified = True

if initial and "uid" in initial:
self.set_uid(initial["uid"])
self.set_user()
else:
self.uid: Optional[int] = None
self.user = None
CallbackDict.__init__(self, initial, on_update)
self.sid = sid
self.token: str = token
self.lifetime = lifetime
self.is_api = False
self.to_destroy = False
self.to_regenerate = False
self.modified = False
self.flash: Optional[Tuple[str, str]] = None
self.locale: Optional[str] = None

def get_token(self) -> Optional[str]:
return self.token

def get_lifetime(self) -> datetime:
return datetime.now(timezone.utc) + timedelta(seconds=self.lifetime)

def set_user(self) -> None:
if self.uid is not None:
self.user = Journalist.query.get(self.uid)
if self.user is None:
# The uid has no match in the database, and this should really not happen
self.uid = None
self.to_destroy = True

def get_user(self) -> Optional[Journalist]:
return self.user

def get_uid(self) -> Optional[int]:
return self.uid

def set_uid(self, uid: int) -> None:
self.user = None
self.uid = uid

def logged_in(self) -> bool:
return self.uid is not None

def destroy(
self, flash: Optional[Tuple[str, str]] = None, locale: Optional[str] = None
) -> None:
# The parameters are needed to pass the information to the new session
self.locale = locale
self.flash = flash
self.uid = None
self.user = None
self.to_destroy = True

def regenerate(self) -> None:
```

```
self.to_regenerate = True

class SessionInterface(FlaskSessionInterface):
def _generate_sid(self) -> str:
return token_urlsafe(32)

def _get_signer(self, app: Flask) -> URLSafeTimedSerializer:
if not app.secret_key:
raise RuntimeError("No secret key set")
return URLSafeTimedSerializer(app.secret_key, salt=self.salt)

"""Uses the Redis key-value store as a session backend.

:param redis: A ``redis.Redis`` instance.
:param key_prefix: A prefix that is added to all Redis store keys.
:param salt: Allows to set the signer salt from the calling interface
:param header_name: if use_header, set the header name to parse
"""

def __init__(
self,
lifetime: int,
renew_count: int,
redis: Redis,
key_prefix: str,
salt: str,
header_name: str,
) -> None:
self.serializer = session_json_serializer
self.redis = redis
self.lifetime = lifetime
self.renew_count = renew_count
self.key_prefix = key_prefix
self.api_key_prefix = "api_" + key_prefix
self.salt = salt
self.api_salt = "api_" + salt
self.header_name = header_name
self.new = False
self.has_same_site_capability = hasattr(self, "get_cookie_samesite")
```

```
def _new_session(self, is_api: bool = False, initial: Any = None) ->
ServerSideSession:
sid = self._generate_sid()
token: str = self._get_signer(app).dumps(sid) # type: ignore
session = ServerSideSession(sid=sid, token=token, lifetime=self.lifetime,
initial=initial)
session.new = True
session.is_api = is_api
return session

def open_session(self, app: Flask, request: Request) ->
Optional[ServerSideSession]:
"""This function is called by the flask session interface at the
beginning of each request.
"""
is_api = request.path.split("/")[1] == "api"

if is_api:
self.key_prefix = self.api_key_prefix
self.salt = self.api_salt
auth_header = request.headers.get(self.header_name)
if auth_header:
split = auth_header.split(" ")
if len(split) != 2 or split[0] != "Token":
return self._new_session(is_api)
sid: Optional[str] = split[1]
else:
return self._new_session(is_api)
else:
sid = request.cookies.get(app.session_cookie_name)
if sid:
try:
sid = self._get_signer(app).loads(sid)
except BadSignature:
sid = None
if not sid:
return self._new_session(is_api)

val = self.redis.get(self.key_prefix + sid)
if val is not None:
```

```
try:
data = self.serializer.loads(val.decode("utf-8"))
token: str = self._get_signer(app).dumps(sid) # type: ignore
return ServerSideSession(sid=sid, token=token, initial=data)
except (JSONDecodeError, NotImplementedError):
return self._new_session(is_api)
# signed session_id provided in cookie is valid, but the session is not on
the server
# anymore so maybe here is the code path for a meaningful error message
msg = gettext("You have been logged out due to inactivity.")
return self._new_session(is_api, initial={"_flashes": [("error", msg)]})

def save_session( # type: ignore[override]
self, app: Flask, session: ServerSideSession, response: Response
) -> None:
"""This is called at the end of each request, just
before sending the response.
"""
domain = self.get_cookie_domain(app)
path = self.get_cookie_path(app)
if session.to_destroy:
initial: Dict[str, Any] = {"locale": session.locale}
if session.flash:
initial["_flashes"] = [session.flash]
self.redis.delete(self.key_prefix + session.sid)
if not session.is_api:
# Instead of deleting the cookie and send a new sid with the next request
# create the new session already, so we can pass along messages and locale
session = self._new_session(False, initial=initial)
expires = self.redis.ttl(name=self.key_prefix + session.sid)
if session.new:
session["renew_count"] = self.renew_count
expires = self.lifetime
elif expires < (30 * 60) and session["renew_count"] > 0:
session["renew_count"] -= 1
expires += self.lifetime
session.modified = True
conditional_cookie_kwargs = {}
```

```
httponly = self.get_cookie_httponly(app)
secure = self.get_cookie_secure(app)
if self.has_same_site_capability:
conditional_cookie_kwargs["samesite"] = self.get_cookie_samesite(app)
val = self.serializer.dumps(dict(session))
if session.to_regenerate:
self.redis.delete(self.key_prefix + session.sid)
session.sid = self._generate_sid()
session.token = self._get_signer(app).dumps(session.sid) # type: ignore
if session.new or session.to_regenerate:
self.redis.setex(name=self.key_prefix + session.sid, value=val,
time=expires)
elif session.modified:
# To prevent race conditions where session is delete by an admin in the
middle of a req
# accept to save the session object if and only if already exists using the
xx flag
self.redis.set(name=self.key_prefix + session.sid, value=val, ex=expires,
xx=True)
if not session.is_api and (session.new or session.to_regenerate):
response.headers.add("Vary", "Cookie")
response.set_cookie(
app.session_cookie_name,
session.token,
httponly=httponly,
domain=domain,
path=path,
secure=secure,
**conditional_cookie_kwargs, # type: ignore
)

def logout_user(self, uid: int) -> None:
for key in self.redis.keys(self.key_prefix + "*") + self.redis.keys(
"api_" + self.key_prefix + "*"
):
found = self.redis.get(key)
if found:
sess = session_json_serializer.loads(found.decode("utf-8"))
if "uid" in sess and sess["uid"] == uid:
```

```
self.redis.delete(key)

class Session:
def __init__(self, app: Flask) -> None:
self.app = app
if app is not None:
self.init_app(app)

def init_app(self, app: Flask) -> "None":
"""This is used to set up session for your app object.
:param app: the Flask app object with proper configuration.
"""
app.session_interface = self._get_interface(app) # type: ignore

def _get_interface(self, app: Flask) -> SessionInterface:
config = app.config.copy()
config.setdefault("SESSION_REDIS", Redis())
config.setdefault("SESSION_LIFETIME", 2 * 60 * 60)
config.setdefault("SESSION_RENEW_COUNT", 5)
config.setdefault("SESSION_SIGNER_SALT", "session")
config.setdefault("SESSION_KEY_PREFIX", "session:")
config.setdefault("SESSION_HEADER_NAME", "authorization")

return SessionInterface(
config["SESSION_LIFETIME"],
config["SESSION_RENEW_COUNT"],
config["SESSION_REDIS"],
config["SESSION_KEY_PREFIX"],
config["SESSION_SIGNER_SALT"],
config["SESSION_HEADER_NAME"],
)

# Re-export flask.session, but with the correct type information for mypy.
from flask import session # noqa

session = typing.cast(ServerSideSession, session)
```

Static Code Metrics:

Metric	Value

Lines of Code (LOC)	233
Comment Density	0.04017857142857143
Cyclomatic Complexity	63
Maintainability Index	38.621740547495676
Halstead Metrics	
Unique Operators	13
Unique Operands	60
Total Operators	42
Total Operands	80
Volume	755.1585961833622
Effort	6544.707833589138
Depth of Inheritance Tree (DIT)	0
Coupling Between Object classes (CBO)	51
Lack of Cohesion of Methods (LCOM)	1.3333333333333333
Fan-in	12
Fan-out	12
Number of Methods (NOM)	20

McCall's Intermediate Quality Metrics:

Metric	Value
Modifiability	0.02
Testability	0.06
Reliability	-0.56
Understandability	0.51
Self-Descriptiveness	0.14
Reusability	-0.27
Portability	0.46
Efficiency	0

GPT Analysis Report:

Analysis of the Provided Python Code

The presented code implements a server-side session management system using Flask and Redis, encapsulating functionalities for user sessions. While the code serves its intended purpose, several areas can be improved in various dimensions such as readability, maintainability, testing, and overall design.

1. Code Structure and Organization

- **Class Design**: The `ServerSideSession` class is quite large, encompassing multiple responsibilities. Consider applying the Single Responsibility Principle, breaking the class down into smaller components. For example, session management (handling state) and user-related actions could be separated.

- **Method Length**: Some methods, particularly `open_session`, are long and complex. Aim for smaller methods that perform a single task, which improves readability and makes the code easier to test.

2. Complexity

- **Cyclomatic Complexity**: The cyclomatic complexity of 63 indicates that the code is highly complex and may be challenging to understand. Target methods with high complexity for refactoring. For instance, the `open_session` method could be simplified by extracting parts of its logic into helper methods.

- **Conditional Logic**: Several places use deep nested logic (e.g. condition checks within `open_session`). Extracting these conditionals into well-named methods could clarify their purpose.

3. Comments and Documentation

- **Comment Density**: The comment density of approximately 4% suggests that significant portions of the code lack explanations. Comments should explain why certain decisions were made or provide context about complex logic rather than stating what is evident from the code itself. Increasing comment density may improve understandability.

- **Docstrings**: Some methods lack docstrings. Adding meaningful docstrings that follow PEP 257 conventions would help clarify the purpose and expected behavior of the methods.

4. Type Hints and Static Typing

- **Type Annotations**: The code uses type hints but could be further enhanced. Ensure all public methods are fully annotated to improve readability and enable better static analysis.

- **Consistency in Type Usage**: Ensure consistent usage of types; for example, using `Optional[str]` is appropriate where a None return might be expected, but confidence checks on returned types could prevent bugs (like prematurely converting `sid` to `str` after decoding).

5. Error Handling

- **Error Messaging**: In `open_session`, when a `BadSignature` or decoding error occurs, consider logging these incidents at a suitable log level rather than simply returning a new session. This enhances tracking issues in production environments.

- **Specific Exceptions**: Rather than catching general exceptions (like `JSONDecodeError`), be specific about what you are catching. This will prevent unexpected behaviors and make debugging easier.

6. Performance Considerations

- **Redis Interaction**: Multiple calls to Redis (e.g., in `logout_user`) could be consolidated or batched. This reduces the number of network roundtrips and improves performance significantly, especially in high-traffic situations.

- **Using `setex` Appropriately**: In the `save_session` method, ensure `setex` is used optimally; there's potential for overwriting based on conditions. The use of conditional setting can lead to race conditions—avoiding data loss.

7. Maintainability and Testability

- **Configurability**: Consider moving hardcoded values (like `30 * 60` and `"Token"`) to constants or configuration settings. This not only improves readability but also facilitates easier testing and adjustments in the future.

- **Testing**: Given the complex nature of this code, developing a suite of unit tests is critical. The high value of `Fan-out` and `Coupling Between Object classes (CBO)` suggests high inter-function code dependency. Mocking out dependencies (e.g., `Redis`) in unit tests could help isolate logic and improve test coverage.

8. Performance Metrics

- **Halstead Metrics and Maintainability Index**: The low Maintainability Index indicates that the code may be difficult to maintain over time. Code that balances complexity, modularity, and clarity often results in a better Maintainability Index, indicating a need for refactoring.

Conclusion

The code has a solid foundation but suffers from complexity, length, and possibly unclear responsibilities. By breaking down classes/methods, improving commenting practices, refactoring long methods, enhancing error handling, and promoting configurability, the code can become much more maintainable and understandable. Fostering these improvements will also make the codebase easier to test and adapt in the future.