# IT314 - Software Engineering
# LAB 5 - Static Analysis

Name : Sumit Monpara
ID : 202001122
Group no.: 15

**Message Object Table**

| Sr. No | Message Object | Expansion | Explanation |
|---|---|---|---|
| 1. | C | Convention | It is displayed when the program is not following the standard rules. |
| 2. | R | Refactor | It is displayed for bad code smell |
| 3. | W | Warning | It is displayed for python specific problems |
| 4. | E | Error | It is displayed when that particular line execution results some error |
| 5. | F | Fetal | It is displayed when pylint has no access to further process that line. |

**Tool Used** : Pylint
**Language** : Python
**Used Code** : [Git Repository Link]

**File** : base.py

**Code:**

```
"""Define the base Pynecone class."""
from __future__ import annotations

from typing import Any, Dict, TypeVar

import pydantic
from pydantic.fields import ModelField

# Typevar to represent any class subclassing Base.
PcType = TypeVar("PcType")


class Base(pydantic.BaseModel):
    """The base class subclassed by all Pynecone classes.

    This class wraps Pydantic and provides common methods such as
    serialization and setting fields.

    Any data structure that needs to be transferred between the
    frontend and backend should subclass this class.
    """

    class Config:
        """Pydantic config."""

        arbitrary_types_allowed = True
        use_enum_values = True

    def json(self) -> str:
        """Convert the object to a json string.
```

```
        Returns:
            The object as a json string.
        """
        return self.__config__.json_dumps(self.dict())

    def set(self: PcType, **kwargs) -> PcType:
        """Set multiple fields and return the object.

        Args:
            **kwargs: The fields and values to set.

        Returns:
            The object with the fields set.
        """
        for key, value in kwargs.items():
            setattr(self, key, value)
        return self

    @classmethod
    def get_fields(cls) -> Dict[str, Any]:
        """Get the fields of the object.

        Returns:
            The fields of the object.
        """
        return cls.__fields__

    @classmethod
    def add_field(cls, var: Any, default_value: Any):
        """Add a pydantic field after class definition.

        Used by State.add_var() to correctly handle the new variable.

        Args:
            var: The variable to add a pydantic field for.
            default_value: The default value of the field
        """
        new_field = ModelField.infer(
            name=var.name,
            value=default_value,
```

```
            annotation=var.type_,
            class_validators=None,
            config=cls.__config__,
        )
        cls.__fields__.update({var.name: new_field})

    def get_value(self, key: str) -> Any:
        """Get the value of a field.

        Args:
            key: The key of the field.

        Returns:
            The value of the field.
        """
        return self._get_value(
            key,
            to_dict=True,
            by_alias=False,
            include=None,
            exclude=None,
            exclude_unset=False,
            exclude_defaults=False,
            exclude_none=False,
        )
```

**Static Analysis**:

```
PS C:\Users\sumit\pynecone\pynecone> pylint base.py
************* Module base
base.py:6:0: E0401: Unable to import 'pydantic' (import-error)
base.py:7:0: E0401: Unable to import 'pydantic.fields' (import-error)
base.py:23:4: R0903: Too few public methods (0/2) (too-few-public-methods)

-------------------------------------------------------------------
Your code has been rated at 5.00/10 (previous run: 5.00/10, +0.00)
```

**File** : config.py

**Code**:

```python
"""The Pynecone config."""

from typing import List, Optional

from pynecone import constants
from pynecone.base import Base


class Config(Base):
    """A Pynecone config."""

    # The name of the app.
    app_name: str

    # The username.
    username: Optional[str] = None

    # The frontend port.
    port: str = constants.FRONTEND_PORT

    # The frontend port.
    backend_port: str = constants.BACKEND_PORT

    # The backend API url.
    api_url: str = constants.API_URL

    # The deploy url.
    deploy_url: Optional[str] = None

    # The database url.
    db_url: Optional[str] = constants.DB_URL

    # The redis url.
    redis_url: Optional[str] = None

    # Telemetry opt-in.
```

```
    telemetry_enabled: bool = True

    # The pcdeploy url.
    pcdeploy_url: Optional[str] = None

    # The environment mode.
    env: constants.Env = constants.Env.DEV

    # The path to the bun executable.
    bun_path: str = constants.BUN_PATH

    # Additional frontend packages to install.
    frontend_packages: List[str] = []

    # Backend transport methods.
    backend_transports: Optional[
        constants.Transports
    ] = constants.Transports.WEBSOCKET_POLLING

    # List of origins that are allowed to connect to the backend API.
    cors_allowed_origins: Optional[list] =
[constants.CORS_ALLOWED_ORIGINS]

    # Whether credentials (cookies, authentication) are allowed in
requests to the backend API.
    cors_credentials: Optional[bool] = True

    # The maximum size of a message when using the polling backend
transport.
    polling_max_http_buffer_size: Optional[int] =
constants.POLLING_MAX_HTTP_BUFFER_SIZE
```

**Static Analysis**:

```
PS C:\Users\sumit\pynecone\pynecone> pylint config.py

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

**File** : constants.py

**Code**:

```
"""Constants used throughout the package."""

import os
import re
from enum import Enum
from types import SimpleNamespace

import pkg_resources

# App names and versions.
# The name of the Pynecone module.
MODULE_NAME = "pynecone"
# The name of the pip install package.
PACKAGE_NAME = "pynecone"
# The current version of Pynecone.
VERSION = pkg_resources.get_distribution(PACKAGE_NAME).version
# Minimum version of Node.js required to run Pynecone.
MIN_NODE_VERSION = "12.22.0"

# Files and directories used to init a new project.
# The root directory of the pynecone library.
ROOT_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
# The name of the file used for pc init.
APP_TEMPLATE_FILE = "tutorial.py"
# The name of the assets directory.
APP_ASSETS_DIR = "assets"
# The template directory used during pc init.
TEMPLATE_DIR = os.path.join(ROOT_DIR, MODULE_NAME, ".templates")
# The web subdirectory of the template directory.
WEB_TEMPLATE_DIR = os.path.join(TEMPLATE_DIR, "web")
# The app subdirectory of the template directory.
APP_TEMPLATE_DIR = os.path.join(TEMPLATE_DIR, "app")
# The assets subdirectory of the template directory.
ASSETS_TEMPLATE_DIR = os.path.join(TEMPLATE_DIR, APP_ASSETS_DIR)

# The frontend directories in a project.
```

```
# The web folder where the NextJS app is compiled to.
WEB_DIR = ".web"
# The name of the utils file.
UTILS_DIR = "utils"
# The name of the state file.
STATE_PATH = "/".join([UTILS_DIR, "state"])
# The name of the components file.
COMPONENTS_PATH = "/".join([UTILS_DIR, "components"])
# The directory where the app pages are compiled to.
WEB_PAGES_DIR = os.path.join(WEB_DIR, "pages")
# The directory where the static build is located.
WEB_STATIC_DIR = os.path.join(WEB_DIR, "_static")
# The directory where the utils file is located.
WEB_UTILS_DIR = os.path.join(WEB_DIR, UTILS_DIR)
# The directory where the assets are located.
WEB_ASSETS_DIR = os.path.join(WEB_DIR, "public")
# The sitemap config file.
SITEMAP_CONFIG_FILE = os.path.join(WEB_DIR, "next-sitemap.config.js")
# The node modules directory.
NODE_MODULES = "node_modules"
# The package lock file.
PACKAGE_LOCK = "package-lock.json"
# The pcversion template file.
PCVERSION_TEMPLATE_FILE = os.path.join(WEB_TEMPLATE_DIR, "pynecone.json")
# The pcversion app file.
PCVERSION_APP_FILE = os.path.join(WEB_DIR, "pynecone.json")


# Commands to run the app.
# The frontend default port.
FRONTEND_PORT = "3000"
# The backend default port.
BACKEND_PORT = "8000"
# The backend api url.
API_URL = "http://localhost:8000"
# The default path where bun is installed.
BUN_PATH = "$HOME/.bun/bin/bun"
# Command to install bun.
INSTALL_BUN = "curl -fsSL https://bun.sh/install | bash -s -- bun-v0.5.5"
# Default host in dev mode.
```

```
BACKEND_HOST = "0.0.0.0"
# The default timeout when launching the gunicorn server.
TIMEOUT = 120
# The command to run the backend in production mode.
RUN_BACKEND_PROD = f"gunicorn --worker-class
uvicorn.workers.UvicornH11Worker --preload --timeout {TIMEOUT} --log-level
critical".split()
RUN_BACKEND_PROD_WINDOWS = f"uvicorn --timeout-keep-alive
{TIMEOUT}".split()
# Socket.IO web server
PING_INTERVAL = 25
PING_TIMEOUT = 5

# Compiler variables.
# The extension for compiled Javascript files.
JS_EXT = ".js"
# The extension for python files.
PY_EXT = ".py"
# The expected variable name where the pc.App is stored.
APP_VAR = "app"
# The expected variable name where the API object is stored for
deployment.
API_VAR = "api"
# The name of the router variable.
ROUTER = "router"
# The name of the socket variable.
SOCKET = "socket"
# The name of the variable to hold API results.
RESULT = "result"
# The name of the process variable.
PROCESSING = "processing"
# The name of the state variable.
STATE = "state"
# The name of the events variable.
EVENTS = "events"
# The name of the initial hydrate event.
HYDRATE = "hydrate"
# The name of the index page.
INDEX_ROUTE = "index"
# The name of the document root page.
```

```
DOCUMENT_ROOT = "_document"
# The name of the theme page.
THEME = "theme"
# The prefix used to create setters for state vars.
SETTER_PREFIX = "set_"
# The name of the frontend zip during deployment.
FRONTEND_ZIP = "frontend.zip"
# The name of the backend zip during deployment.
BACKEND_ZIP = "backend.zip"
# The name of the sqlite database.
DB_NAME = "pynecone.db"
# The sqlite url.
DB_URL = f"sqlite:///{DB_NAME}"
# The default title to show for Pynecone apps.
DEFAULT_TITLE = "Pynecone App"
# The default description to show for Pynecone apps.
DEFAULT_DESCRIPTION = "A Pynecone app."
# The default image to show for Pynecone apps.
DEFAULT_IMAGE = "favicon.ico"
# The default meta list to show for Pynecone apps.
DEFAULT_META_LIST = []


# The gitignore file.
GITIGNORE_FILE = ".gitignore"
# Files to gitignore.
DEFAULT_GITIGNORE = {WEB_DIR, DB_NAME}
# The name of the pynecone config module.
CONFIG_MODULE = "pcconfig"
# The python config file.
CONFIG_FILE = f"{CONFIG_MODULE}{PY_EXT}"
# The deployment URL.
PRODUCTION_BACKEND_URL = "https://{username}-{app_name}.api.pynecone.app"
# Token expiration time in seconds.
TOKEN_EXPIRATION = 60 * 60


# Env modes
class Env(str, Enum):
    """The environment modes."""
```

```
    DEV = "dev"
    PROD = "prod"


# Log levels
class LogLevel(str, Enum):
    """The log levels."""

    DEBUG = "debug"
    INFO = "info"
    WARNING = "warning"
    ERROR = "error"
    CRITICAL = "critical"


class Endpoint(Enum):
    """Endpoints for the pynecone backend API."""

    PING = "ping"
    EVENT = "event"
    UPLOAD = "upload"

    def __str__(self) -> str:
        """Get the string representation of the endpoint.

        Returns:
            The path for the endpoint.
        """
        return f"/{self.value}"

    def get_url(self) -> str:
        """Get the URL for the endpoint.

        Returns:
            The full URL for the endpoint.
        """
        # Import here to avoid circular imports.
        from pynecone import utils
```

```
        # Get the API URL from the config.
        config = utils.get_config()
        url = "".join([config.api_url, str(self)])

        # The event endpoint is a websocket.
        if self == Endpoint.EVENT:
            # Replace the protocol with ws.
            url = url.replace("https://", "wss://").replace("http://",
"ws://")

        # Return the url.
        return url


class SocketEvent(Enum):
    """Socket events sent by the pynecone backend API."""

    PING = "ping"
    EVENT = "event"

    def __str__(self) -> str:
        """Get the string representation of the event name.

        Returns:
            The event name string.
        """
        return str(self.value)


class Transports(Enum):
    """Socket transports used by the pynecone backend API."""

    POLLING_WEBSOCKET = "['polling', 'websocket']"
    WEBSOCKET_POLLING = "['websocket', 'polling']"
    WEBSOCKET_ONLY = "['websocket']"
    POLLING_ONLY = "['polling']"

    def __str__(self) -> str:
        """Get the string representation of the transports.
```

```
        Returns:
            The transports string.
        """
        return str(self.value)

    def get_transports(self) -> str:
        """Get the transports config for the backend.

        Returns:
            The transports config for the backend.
        """
        # Import here to avoid circular imports.
        from pynecone import utils

        # Get the transports from the config.
        config = utils.get_config()
        return str(config.backend_transports)


class RouteArgType(SimpleNamespace):
    """Type of dynamic route arg extracted from URI route."""

    # Typecast to str is needed for Enum to work.
    SINGLE = str("arg_single")
    LIST = str("arg_list")


class RouteVar(SimpleNamespace):
    """Names of variables used in the router_data dict stored in State."""

    CLIENT_IP = "ip"
    CLIENT_TOKEN = "token"
    HEADERS = "headers"
    PATH = "pathname"
    SESSION_ID = "sid"
    QUERY = "query"


class RouteRegex(SimpleNamespace):
    """Regex used for extracting route args in route."""
```

```
    ARG = re.compile(r"\[(?!\.)([^\[\]]+)\]")
    # group return the catchall pattern (i.e. "[[..slug]]")
    CATCHALL = re.compile(r"(\[?\[\.{3}(?![0-9]).*\]?\])")
    # group return the arg name (i.e. "slug")
    STRICT_CATCHALL = re.compile(r"\[\.{3}([a-zA-Z_][\w]*)\]")
    # group return the arg name (i.e. "slug")
    OPT_CATCHALL = re.compile(r"\[\[\.{3}([a-zA-Z_][\w]*)\]\]")


# 404 variables
ROOT_404 = ""
SLUG_404 = "[..._]"
TITLE_404 = "404 - Not Found"
FAVICON_404 = "favicon.ico"
DESCRIPTION_404 = "The page was not found"

# Color mode variables
USE_COLOR_MODE = "useColorMode"
COLOR_MODE = "colorMode"
TOGGLE_COLOR_MODE = "toggleColorMode"

# Server socket configuration variables
CORS_ALLOWED_ORIGINS = "*"
POLLING_MAX_HTTP_BUFFER_SIZE = 1000 * 1000
```

## Static Analysis:

```
PS C:\Users\sumit\pynecone\pynecone> pylint constants.py
************ Module constants
constants.py:81:0: C0301: Line too long (137/100) (line-too-long)
constants.py:191:8: C0415: Import outside toplevel (pynecone.utils) (import-outside-toplevel)
constants.py:244:8: C0415: Import outside toplevel (pynecone.utils) (import-outside-toplevel)
constants.py:237:4: R0201: Method could be a function (no-self-use)
constants.py:251:0: R0903: Too few public methods (0/2) (too-few-public-methods)
constants.py:259:0: R0903: Too few public methods (0/2) (too-few-public-methods)
constants.py:270:0: R0903: Too few public methods (0/2) (too-few-public-methods)

---------------------------------
Your code has been rated at 9.47/10
```

**File** : event.py

**Code**:

```python
"""Define event classes to connect the frontend and backend."""
from __future__ import annotations

import inspect
from typing import Any, Callable, Dict, List, Set, Tuple

from pynecone import utils
from pynecone.base import Base
from pynecone.var import BaseVar, Var


class Event(Base):
    """An event that describes any state change in the app."""

    # The token to specify the client that the event is for.
    token: str

    # The event name.
    name: str

    # The routing data where event occurred
    router_data: Dict[str, Any] = {}

    # The event payload.
    payload: Dict[str, Any] = {}


class EventHandler(Base):
    """An event handler responds to an event to update the state."""

    # The function to call in response to the event.
    fn: Callable

    class Config:
        """The Pydantic config."""
```

```
        # Needed to allow serialization of Callable.
        frozen = True

    def __call__(self, *args: Var) -> EventSpec:
        """Pass arguments to the handler to get an event spec.

        This method configures event handlers that take in arguments.

        Args:
            *args: The arguments to pass to the handler.

        Returns:
            The event spec, containing both the function and args.

        Raises:
            TypeError: If the arguments are invalid.
        """
        # Get the function args.
        fn_args = inspect.getfullargspec(self.fn).args[1:]

        # Construct the payload.
        values = []
        for arg in args:
            # If it is a Var, add the full name.
            if isinstance(arg, Var):
                values.append(arg.full_name)
                continue

            if isinstance(arg, FileUpload):
                return EventSpec(handler=self, upload=True)

            # Otherwise, convert to JSON.
            try:
                values.append(utils.json_dumps(arg))
            except TypeError as e:
                raise TypeError(
                    f"Arguments to event handlers must be Vars or
JSON-serializable. Got {arg} of type {type(arg)}."
                ) from e
        payload = tuple(zip(fn_args, values))
```

```
        # Return the event spec.
        return EventSpec(handler=self, args=payload)


class EventSpec(Base):
    """An event specification.

    Whereas an Event object is passed during runtime, a spec is used
    during compile time to outline the structure of an event.
    """

    # The event handler.
    handler: EventHandler

    # The local arguments on the frontend.
    local_args: Tuple[str, ...] = ()

    # The arguments to pass to the function.
    args: Tuple[Any, ...] = ()

    # Whether to upload files.
    upload: bool = False

    class Config:
        """The Pydantic config."""

        # Required to allow tuple fields.
        frozen = True


class EventChain(Base):
    """Container for a chain of events that will be executed in order."""

    events: List[EventSpec]


class Target(Base):
    """A Javascript event target."""
```

```
    checked: bool = False
    value: Any = None


class FrontendEvent(Base):
    """A Javascript event."""

    target: Target = Target()
    key: str = ""


# The default event argument.
EVENT_ARG = BaseVar(name="_e", type_=FrontendEvent, is_local=True)


class FileUpload(Base):
    """Class to represent a file upload."""

    pass


# Special server-side events.
def redirect(path: str) -> EventSpec:
    """Redirect to a new path.

    Args:
        path: The path to redirect to.

    Returns:
        An event to redirect to the path.
    """

    def fn():
        return None

    fn.__qualname__ = "_redirect"
    return EventSpec(
        handler=EventHandler(fn=fn),
        args=(("path", path),),
    )
```

```python
def console_log(message: str) -> EventSpec:
    """Do a console.log on the browser.

    Args:
        message: The message to log.

    Returns:
        An event to log the message.
    """

    def fn():
        return None

    fn.__qualname__ = "_console"
    return EventSpec(
        handler=EventHandler(fn=fn),
        args=(("message", message),),
    )


def window_alert(message: str) -> EventSpec:
    """Create a window alert on the browser.

    Args:
        message: The message to alert.

    Returns:
        An event to alert the message.
    """

    def fn():
        return None

    fn.__qualname__ = "_alert"
    return EventSpec(
        handler=EventHandler(fn=fn),
        args=(("message", message),),
    )
```

```python
# A set of common event triggers.
EVENT_TRIGGERS: Set[str] = {
    "on_focus",
    "on_blur",
    "on_click",
    "on_context_menu",
    "on_double_click",
    "on_mouse_down",
    "on_mouse_enter",
    "on_mouse_leave",
    "on_mouse_move",
    "on_mouse_out",
    "on_mouse_over",
    "on_mouse_up",
    "on_scroll",
}
```

Static Analysis:

```
PS C:\Users\sumit\pynecone\pynecone> pylint event.py
************* Module event
event.py:73:0: C0301: Line too long (116/100) (line-too-long)
event.py:34:4: R0903: Too few public methods (0/2) (too-few-public-methods)
event.py:71:12: C0103: Variable name "e" doesn't conform to snake_case naming style (invalid-name)
event.py:100:4: R0903: Too few public methods (0/2) (too-few-public-methods)
event.py:134:4: W0107: Unnecessary pass statement (unnecessary-pass)
event.py:148:4: C0103: Function name "fn" doesn't conform to snake_case naming style (invalid-name)
event.py:168:4: C0103: Function name "fn" doesn't conform to snake_case naming style (invalid-name)
event.py:188:4: C0103: Function name "fn" doesn't conform to snake_case naming style (invalid-name)

--------------------------------
Your code has been rated at 8.75/10
```

**File** : model.py

**Code**:

```
"""Database built into Pynecone."""

import sqlmodel

from pynecone import utils
from pynecone.base import Base


def get_engine():
    """Get the database engine.

    Returns:
        The database engine.

    Raises:
        ValueError: If the database url is None.
    """
    url = utils.get_config().db_url
    if url is None:
        raise ValueError("No database url in config")
    return sqlmodel.create_engine(url, echo=False)


class Model(Base, sqlmodel.SQLModel):
    """Base class to define a table in the database."""

    # The primary key for the table.
    id: int = sqlmodel.Field(primary_key=True)

    def dict(self, **kwargs):
        """Convert the object to a dictionary.

        Args:
            kwargs: Ignored but needed for compatibility.

        Returns:
```

```
            The object as a dictionary.
        """
        return {name: getattr(self, name) for name in self.__fields__}

    @staticmethod
    def create_all():
        """Create all the tables."""
        engine = get_engine()
        sqlmodel.SQLModel.metadata.create_all(engine)

    @classmethod
    @property
    def select(cls):
        """Select rows from the table.

        Returns:
            The select statement.
        """
        return sqlmodel.select(cls)


def session(url=None):
    """Get a session to interact with the database.

    Args:
        url: The database url.

    Returns:
        A database session.
    """
    if url is not None:
        return sqlmodel.Session(sqlmodel.create_engine(url))
    engine = get_engine()
    return sqlmodel.Session(engine)
```

**Static Analysis**:

```
PS C:\Users\sumit\pynecone\pynecone> pylint model.py
************* Module model
model.py:3:0: E0401: Unable to import 'sqlmodel' (import-error)
model.py:30:0: W0613: Unused argument 'kwargs' (unused-argument)

-----------------------------------
Your code has been rated at 7.27/10
```

**File** : pc.py

**Code**:

```python
"""Pynecone CLI to create, run, and deploy apps."""

import os
import platform
from pathlib import Path

import httpx
import typer

from pynecone import constants, utils
from pynecone.telemetry import pynecone_telemetry

# Create the app.
cli = typer.Typer()


@cli.command()
def version():
    """Get the Pynecone version."""
    utils.console.print(constants.VERSION)


@cli.command()
def init():
    """Initialize a new Pynecone app in the current directory."""
    app_name = utils.get_default_app_name()
```

```
    # Make sure they don't name the app "pynecone".
    if app_name == constants.MODULE_NAME:
        utils.console.print(
            f"[red]The app directory cannot be named
[bold]{constants.MODULE_NAME}."
        )
        raise typer.Exit()

    with utils.console.status(f"[bold]Initializing {app_name}"):
        # Set up the web directory.
        utils.install_bun()
        utils.initialize_web_directory()

        # Set up the app directory, only if the config doesn't exist.
        if not os.path.exists(constants.CONFIG_FILE):
            utils.create_config(app_name)
            utils.initialize_app_directory(app_name)
            utils.set_pynecone_project_hash()
            pynecone_telemetry("init",
utils.get_config().telemetry_enabled)
        else:
            utils.set_pynecone_project_hash()
            pynecone_telemetry("reinit",
utils.get_config().telemetry_enabled)

        # Initialize the .gitignore.
        utils.initialize_gitignore()
        # Finish initializing the app.
        utils.console.log(f"[bold green]Finished Initializing:
{app_name}")


@cli.command()
def run(
    env: constants.Env = typer.Option(
        constants.Env.DEV, help="The environment to run the app in."
    ),
    frontend: bool = typer.Option(
        True, "--no-frontend", help="Disable frontend execution."
    ),
```

```
    backend: bool = typer.Option(
        True, "--no-backend", help="Disable backend execution."
    ),
    loglevel: constants.LogLevel = typer.Option(
        constants.LogLevel.ERROR, help="The log level to use."
    ),
    port: str = typer.Option(None, help="Specify a different port."),
):
    """Run the app in the current directory."""
    if platform.system() == "Windows":
        utils.console.print(
            "[yellow][WARNING] We strongly advise you to use Windows Subsystem for Linux (WSL) for optimal performance when using Pynecone. Due to compatibility issues with one of our dependencies, Bun, you may experience slower performance on Windows. By using WSL, you can expect to see a significant speed increase."
        )

    frontend_port = utils.get_config().port if port is None else port
    backend_port = utils.get_config().backend_port

    # If something is running on the ports, ask the user if they want to kill or change it.
    if utils.is_process_on_port(frontend_port):
        frontend_port = utils.change_or_terminate_port(frontend_port, "frontend")

    if utils.is_process_on_port(backend_port):
        backend_port = utils.change_or_terminate_port(backend_port, "backend")

    # Check that the app is initialized.
    if frontend and not utils.is_initialized():
        utils.console.print(
            "[red]The app is not initialized. Run [bold]pc init[/bold] first."
        )
        raise typer.Exit()

    # Check that the template is up to date.
```

```
    if frontend and not utils.is_latest_template():
        utils.console.print(
            "[red]The base app template has updated. Run [bold]pc
init[/bold] again."
        )
        raise typer.Exit()

    # Get the app module.
    utils.console.rule("[bold]Starting Pynecone App")
    app = utils.get_app()

    # Get the frontend and backend commands, based on the environment.
    frontend_cmd = backend_cmd = None
    if env == constants.Env.DEV:
        frontend_cmd, backend_cmd = utils.run_frontend, utils.run_backend
    if env == constants.Env.PROD:
        frontend_cmd, backend_cmd = utils.run_frontend_prod,
utils.run_backend_prod
    assert frontend_cmd and backend_cmd, "Invalid env"

    # Post a telemetry event.
    pynecone_telemetry(f"run-{env.value}",
utils.get_config().telemetry_enabled)

    # Run the frontend and backend.
    try:
        if frontend:
            frontend_cmd(app.app, Path.cwd(), frontend_port)
        if backend:
            backend_cmd(app.__name__, port=int(backend_port),
loglevel=loglevel)
    finally:
        utils.kill_process_on_port(frontend_port)
        utils.kill_process_on_port(backend_port)


@cli.command()
def deploy(dry_run: bool = typer.Option(False, help="Whether to run a dry
run.")):
    """Deploy the app to the Pynecone hosting service."""
```

```
    # Get the app config.
    config = utils.get_config()
    config.api_url = utils.get_production_backend_url()

    # Check if the deploy url is set.
    if config.pcdeploy_url is None:
        typer.echo("This feature is coming soon!")
        return

    # Compile the app in production mode.
    typer.echo("Compiling production app")
    app = utils.get_app().app
    utils.export_app(app, zip=True, deploy_url=config.deploy_url)

    # Exit early if this is a dry run.
    if dry_run:
        return

    # Deploy the app.
    data = {"userId": config.username, "projectId": config.app_name}
    original_response = httpx.get(config.pcdeploy_url, params=data)
    response = original_response.json()
    frontend = response["frontend_resources_url"]
    backend = response["backend_resources_url"]

    # Upload the frontend and backend.
    with open(constants.FRONTEND_ZIP, "rb") as f:
        httpx.put(frontend, data=f)  # type: ignore

    with open(constants.BACKEND_ZIP, "rb") as f:
        httpx.put(backend, data=f)  # type: ignore


@cli.command()
def export(
    zipping: bool = typer.Option(
        True, "--no-zip", help="Disable zip for backend and frontend
exports."
    ),
    frontend: bool = typer.Option(
```

```
        True, "--backend-only", help="Export only backend.",
show_default=False
    ),
    backend: bool = typer.Option(
        True, "--frontend-only", help="Export only frontend.",
show_default=False
    ),
    for_pc_deploy: bool = typer.Option(
        False,
        "--for-pc-deploy",
        help="Whether export the app for Pynecone Deploy Service.",
    ),
):
    """Export the app to a zip file."""
    config = utils.get_config()

    if for_pc_deploy:
        # Get the app config and modify the api_url base on username and
app_name.
        config.api_url = utils.get_production_backend_url()

    # Compile the app in production mode and export it.
    utils.console.rule("[bold]Compiling production app and preparing for
export.")
    app = utils.get_app().app
    utils.export_app(
        app,
        backend=backend,
        frontend=frontend,
        zip=zipping,
        deploy_url=config.deploy_url,
    )

    # Post a telemetry event.
    pynecone_telemetry("export", utils.get_config().telemetry_enabled)

    if zipping:
        utils.console.rule(
            """Backend & Frontend compiled. See [green
bold]backend.zip[/green bold]
```

```
            and [green bold]frontend.zip[/green bold]."""
        )
    else:
        utils.console.rule(
            """Backend & Frontend compiled. See [green bold]app[/green
bold]
            and [green bold].web/_static[/green bold] directories."""
        )


main = cli

if __name__ == "__main__":
    main()
```

## Static Analysis:

```
PS C:\Users\sumit\pynecone\pynecone> pylint pc.py
************* Module pc
pc.py:75:0: C0301: Line too long (319/100) (line-too-long)
pc.py:204:84: C0303: Trailing whitespace (trailing-whitespace)
pc.py:209:76: C0303: Trailing whitespace (trailing-whitespace)
pc.py:7:0: E0401: Unable to import 'httpx' (import-error)
pc.py:8:0: E0401: Unable to import 'typer' (import-error)
pc.py:157:47: C0103: Variable name "f" doesn't conform to snake_case naming style (invalid-name)
pc.py:160:46: C0103: Variable name "f" doesn't conform to snake_case naming style (invalid-name)

-----------------------------------
Your code has been rated at 8.37/10
```