Documento de Estilo de Código

1. Introducción

Propósito

El objetivo de este documento es establecer un conjunto de directrices y convenciones para la escritura del código fuente en el proyecto de implementación de la User Story "Aceptar Cotización". Estas directrices están diseñadas para asegurar que el código sea consistente, legible y mantenible, facilitando la colaboración entre los miembros del equipo y la futura evolución del proyecto.

Alcance

Este documento de estilo de código se aplica a todos los archivos de código fuente del proyecto, incluyendo scripts, módulos y paquetes escritos en Python. Cubre aspectos como el formato del código, la nomenclatura de identificadores, la organización del código y las prácticas recomendadas para la documentación. Las reglas establecidas en este documento deben ser seguidas durante el desarrollo del proyecto y también se aplicarán a cualquier mantenimiento o ampliación futura del código.

Las directrices están basadas en las recomendaciones de PEP 8, que es la guía de estilo oficial para Python, así como en buenas prácticas adicionales que el equipo ha decidido implementar para mejorar la calidad del código y asegurar la coherencia del proyecto.

2. Convenciones de Codificación

Formato General

- **Longitud de Línea:**
    - Las líneas de código no deben exceder los 79 caracteres. En el caso de comentarios y docstrings, el límite es de 72 caracteres.
- **Indentación:**
    - Utilizar 4 espacios por nivel de indentación. No se deben usar tabulaciones para la indentación.

- **Espaciado:**
  - **Antes y después de operadores:** Se debe poner un espacio alrededor de los operadores (por ejemplo, `a + b`, `x == y`).
  - **Después de comas, puntos y dos puntos:** Se debe poner un espacio después de cada uno (por ejemplo, `x, y, z`).
  - **Antes de puntos y comas:** No se debe poner un espacio antes del punto y coma.

Nombres de Identificadores

- **Variables:**
  - Usar nombres descriptivos en `snake_case` (por ejemplo, `metodo_pago`, `importe`).
- **Funciones/Métodos:**
  - Usar nombres descriptivos en `snake_case` (por ejemplo, `procesar_pago(metodo_pago, importe)`).
- **Clases:**
  - Usar `PascalCase` para los nombres de clases (por ejemplo, `Cotizacion`, `Transportista`).
- **Constantes:**
  - Usar `UPPER_CASE` con guiones bajos para los nombres de constantes (por ejemplo, `EMAIL_USER`, `EMAIL_PASS`).

Comentarios

- **Comentarios en Línea:**
  - Utilizar comentarios en línea para explicar partes del código que no son inmediatamente claras. Deben comenzar con un `#` seguido de un espacio y estar alineados con el código al que se refieren.

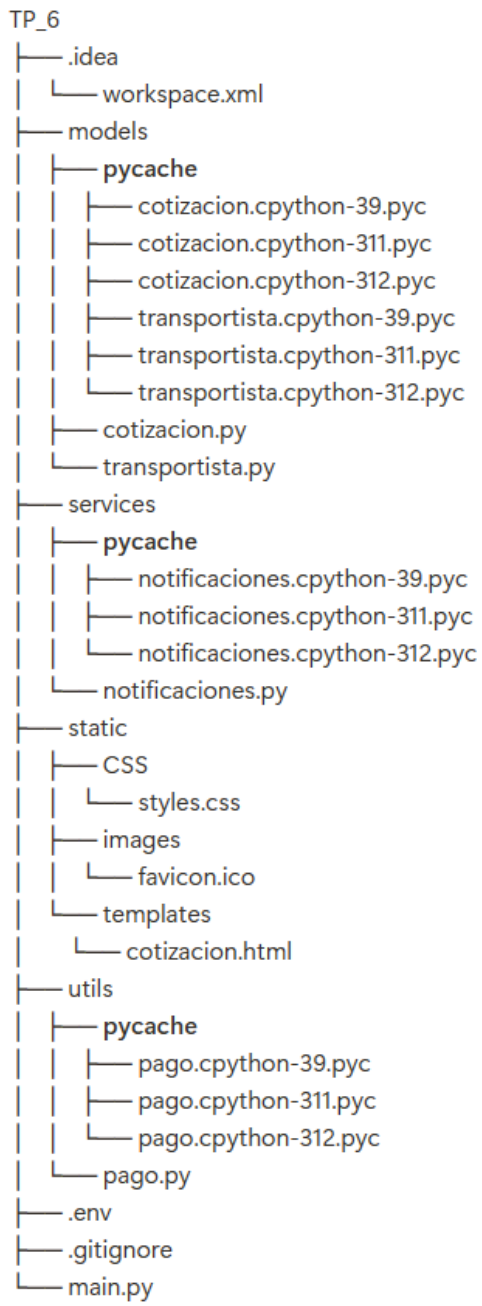
3. Estructura del código

Módulos y Paquetes

- **Organización de Módulos:**
  - Los módulos deben ser organizados de manera lógica y coherente según su funcionalidad.
  - Utilizar nombres descriptivos para los módulos que reflejen claramente su propósito (por ejemplo, `pago.py`, `notificaciones.py`).

- **Estructura de Paquetes:**
    - Los paquetes deben organizarse en carpetas que agrupen módulos relacionados. Cada paquete debe contener un archivo `__init__.py` para ser reconocido como un paquete por Python.

Estructura del trabajo:



Importaciones

- **Orden de Importaciones:**
  - Las importaciones deben estar ordenadas en tres secciones:
    1. Importaciones de módulos estándar de Python (por ejemplo, `import os`).
    2. Importaciones de terceros (por ejemplo, `import flask`, `import dotenv`).
    3. Importaciones locales (por ejemplo, `from models.cotizacion import Cotizacion`).
- **Agrupación y Espaciado:**
  - Las importaciones deben estar separadas por una línea en blanco entre cada grupo para mejorar la legibilidad.

Funciones y Métodos

- **Definición y Diseño:**
  - Las funciones y métodos deben ser cortos y realizar una sola tarea específica.
  - Usar nombres descriptivos para las funciones y métodos que reflejen claramente su propósito (por ejemplo, `enviar_notificacion_push`, `enviar_mail_confirmacion`).
- **Parámetros y Valores de Retorno:**
  - Utilizar nombres de parámetros descriptivos para que el propósito de cada uno sea claro.

Clases

- **Definición y Diseño:**
  - Las clases deben ser diseñadas para representar entidades y comportamientos claros en el dominio del problema. Cada clase debe tener una única responsabilidad.
  - Usar métodos y atributos que sean relevantes para la funcionalidad de la clase.

4. Seguridad y Manejo de Errores

Seguridad

Manejo de Errores

- **Captura de Errores:**
  - Implementar un manejo de errores robusto utilizando bloques `try` y `except` para capturar excepciones y evitar que el programa se detenga inesperadamente. (`notificaciones.py` y `pago.py`)
- **Mensajes de Error para el Usuario:**
  - Proporcionar mensajes de error claros y amigables para el usuario final que no revelen información técnica innecesaria. Los mensajes deben ser lo suficientemente descriptivos para que el usuario entienda qué hacer a continuación.
- **Validación y Manejo de Excepciones:**
  - Validar las excepciones conocidas y manejar cada tipo de error de manera específica si es necesario. Evitar capturar excepciones genéricas que pueden ocultar errores inesperados.
  - Implementar estrategias de recuperación y manejo adecuado para errores esperados, como errores de red o errores de entrada del usuario.

5. Explicación de las Funciones

Notificaciones.py

`enviar_notificacion_push(transportista, forma_pago)`

Descripción: Esta función envía una notificación PUSH al transportista con un mensaje personalizado según la forma de pago seleccionada. El mensaje varía dependiendo de si el pago se realizó con tarjeta, en efectivo al retirar o en efectivo contra entrega.

Parámetros:

- `transportista` (str): El nombre o identificador del transportista al que se envía la notificación.
- `forma_pago` (str): La forma de pago seleccionada (`'tarjeta'`, `'contado_retiro'`, `'contado_entrega'`).

Retorno:

- Devuelve una cadena que confirma el envío de la notificación PUSH.

Excepciones:

- No lanza excepciones explícitas, pero la función maneja internamente casos en los que la forma de pago no está en el diccionario de mensajes.

`enviar_email_confirmacion(transportista, email, forma_pago)`

Descripción: Esta función envía un correo electrónico al transportista confirmando que su cotización ha sido aceptada, especificando la forma de pago. Utiliza el servidor SMTP de Gmail para enviar el correo.

Parámetros:

- `transportista` (str): El nombre del transportista al que se envía el correo.
- `email` (str): La dirección de correo electrónico del transportista.
- `forma_pago` (str): La forma de pago seleccionada (`'tarjeta'`, `'contado_retiro'`, `'contado_entrega'`).

Retorno:

- Devuelve una cadena que indica el resultado del envío del correo, ya sea exitoso o con un error.

Excepciones:

- Captura y retorna errores relacionados con la conexión al servidor SMTP y el proceso de envío del correo.

Pago.py

`validar_tarjeta(numero_tarjeta, pin, nombre_completo, tipo_documento, numero_documento, fecha_vencimiento)`

Descripción: Esta función valida los datos de una tarjeta de crédito, incluyendo el número de tarjeta, el PIN, el nombre completo del titular, el tipo y número de documento, y la fecha de vencimiento. La validación asegura que todos los campos estén en el formato correcto y que la tarjeta no esté vencida.

Parámetros:

- `numero_tarjeta` (str): El número de la tarjeta de crédito (debe tener 16 dígitos).
- `pin` (str): El PIN de la tarjeta (debe tener 4 dígitos).
- `nombre_completo` (str): El nombre completo del titular de la tarjeta.
- `tipo_documento` (str): El tipo de documento de identidad del titular.
- `numero_documento` (str): El número de documento de identidad del titular.
- `fecha_vencimiento` (str): La fecha de vencimiento de la tarjeta en formato `MM/YY`.

Retorno:

- Devuelve una tupla `(bool, str)` donde el primer elemento es `True` si los datos son válidos y `False` si hay errores, y el segundo elemento es un mensaje que indica el resultado de la validación.

Excepciones:

- Captura `ValueError` si la fecha de vencimiento no puede ser convertida a una fecha válida.

`procesar_pago(metodo_pago, importe)`

Descripción: Esta función simula el procesamiento de un pago basado en el método de pago seleccionado y el importe a pagar. Verifica si el saldo disponible es suficiente para cubrir el importe si el método de pago es "tarjeta".

Parámetros:

- `metodo_pago` (str): El método de pago seleccionado (`'tarjeta'`, `'contado_retiro'`, `'contado_entrega'`).
- `importe` (float): El importe total a pagar.

Retorno:

- Devuelve una tupla `(bool, str)` donde el primer elemento es `True` si el pago fue procesado exitosamente y `False` si hubo un error, y el segundo elemento es un mensaje que indica el resultado del procesamiento del pago (por ejemplo, "Saldo insuficiente" o el número de transacción).

Excepciones:

- No lanza excepciones explícitas, pero asume que el saldo disponible es una constante y no se actualiza.

Main.py

`index()`

Descripción: Esta función maneja la solicitud a la ruta raíz (`"/"`) de la aplicación web. Renderiza la plantilla `cotizacion.html` con los detalles de la cotización actual, incluyendo el nombre y calificación del transportista, la fecha de retiro y entrega, y el importe.

Retorno:

- Devuelve el contenido renderizado de la plantilla `cotizacion.html` con los datos de la cotización.

`procesar_pago_view()`

Descripción: Esta función maneja las solicitudes a la ruta `"/procesar_pago"` con el método `POST`. Procesa el pago de la cotización según el método de pago seleccionado. Valida los datos de la tarjeta si el método de pago es "tarjeta" y simula el procesamiento del pago. Actualiza el estado de la cotización a "Confirmado" y envía notificaciones al transportista por push y email.

Parámetros:

- `forma_pago` (str): El método de pago seleccionado (`'tarjeta'`, `'contado al retirar'`, `'contado contra entrega'`).

Retorno:

- Devuelve un mensaje que indica el resultado del procesamiento del pago, el número de transacción (si el pago es exitoso), y las notificaciones enviadas.