```
import concurrent.futures
import os
import uuid
from dataclasses import dataclass
from io import BytesIO
from typing import List

import backoff
import torch
from cachetools import TTLCache
from diffusers import StableDiffusionXLPipeline
from PIL import Image
from pydantic import field_validator
from termcolor import colored


@dataclass
class SSD1B:
    """
    SSD1B model class

    Attributes:
    -----------
    image_url: str
        The image url generated by the SSD1B API
```

```
Methods:
--------
__call__(self, task: str) -> SSD1B:
    Makes a call to the SSD1B API and returns the image url


Example:
--------
    model = SSD1B()
    task = "A painting of a dog"
    neg_prompt = "ugly, blurry, poor quality"
    image_url = model(task, neg_prompt)
    print(image_url)
"""


model: str = "dall-e-3"
img: str = None
size: str = "1024x1024"
max_retries: int = 3
quality: str = "standard"
model_name: str = "segment/SSD-1B"
n: int = 1
save_path: str = "images"
max_time_seconds: int = 60
save_folder: str = "images"
image_format: str = "png"
device: str = "cuda"
```

```
dashboard: bool = False

cache = TTLCache(maxsize=100, ttl=3600)

pipe = StableDiffusionXLPipeline.from_pretrained(

    "segmind/SSD-1B",

    torch_dtype=torch.float16,

    use_safetensors=True,

    variant="fp16",

).to(device)


def __post_init__(self):

    """Post init method"""


    if self.img is not None:

        self.img = self.convert_to_bytesio(self.img)


    os.makedirs(self.save_path, exist_ok=True)


class Config:

    """Config class for the SSD1B model"""


    arbitrary_types_allowed = True


@field_validator("max_retries", "time_seconds")

@classmethod

def must_be_positive(cls, value):

    if value <= 0:
        raise ValueError("Must be positive")
    return value


def read_img(self, img: str):
    """Read the image using pil"""
    img = Image.open(img)
    return img


def set_width_height(self, img: str, width: int, height: int):
    """Set the width and height of the image"""
    img = self.read_img(img)
    img = img.resize((width, height))
    return img


def convert_to_bytesio(self, img: str, format: str = "PNG"):
    """Convert the image to an bytes io object"""
    byte_stream = BytesIO()
    img.save(byte_stream, format=format)
    byte_array = byte_stream.getvalue()
    return byte_array


@backoff.on_exception(
    backoff.expo, Exception, max_time=max_time_seconds
)
def __call__(self, task: str, neg_prompt: str):
    """
```

```
Text to image conversion using the SSD1B API


Parameters:
-----------
task: str
    The task to be converted to an image


Returns:
--------
SSD1B:
    An instance of the SSD1B class with the image url generated by the SSD1B API


Example:
--------
>>> dalle3 = SSD1B()
>>> task = "A painting of a dog"
>>> image_url = dalle3(task)
>>> print(image_url)
https://cdn.openai.com/dall-e/encoded/feats/feats_01J9J5ZKJZJY9.png
"""
if self.dashboard:
    self.print_dashboard()
if task in self.cache:
    return self.cache[task]
try:
    img = self.pipe(
        prompt=task, neg_prompt=neg_prompt
    ).images[0]


    # Generate a unique filename for the image
    img_name = f"{uuid.uuid4()}.{self.image_format}"
    img_path = os.path.join(self.save_path, img_name)


    # Save the image
    img.save(img_path, self.image_format)
    self.cache[task] = img_path


    return img_path


except Exception as error:
    # Handling exceptions and printing the errors details
    print(
        colored(
            (
                f"Error running SSD1B: {error} try optimizing"
                " your api key and or try again"
            ),
            "red",
        )
    )
    raise error
```

```
def _generate_image_name(self, task: str):
    """Generate a sanitized file name based on the task"""
    sanitized_task = "".join(
        char for char in task if char.isalnum() or char in " _ -"
    ).rstrip()
    return f"{sanitized_task}.{self.image_format}"


def _download_image(self, img: Image, filename: str):
    """
    Save the PIL Image object to a file.
    """
    full_path = os.path.join(self.save_path, filename)
    img.save(full_path, self.image_format)


def print_dashboard(self):
    """Print the SSD1B dashboard"""
    print(
        colored(
            f"""SSD1B Dashboard:
                --------------------


                Model: {self.model}
                Image: {self.img}
                Size: {self.size}
                Max Retries: {self.max_retries}
                Quality: {self.quality}
                N: {self.n}

                Save Path: {self.save_path}

                Time Seconds: {self.time_seconds}

                Save Folder: {self.save_folder}

                Image Format: {self.image_format}

                --------------------




                """,

            "green",

        )

    )


def process_batch_concurrently(

    self, tasks: List[str], max_workers: int = 5

):

    """



    Process a batch of tasks concurrently



    Args:

    tasks (List[str]): A list of tasks to be processed

    max_workers (int): The maximum number of workers to use for the concurrent processing



    Returns:

    --------
```

```
results (List[str]): A list of image urls generated by the SSD1B API


Example:

--------

>>> model = SSD1B()

>>> tasks = ["A painting of a dog", "A painting of a cat"]

>>> results = model.process_batch_concurrently(tasks)

>>> print(results)


"""

with concurrent.futures.ThreadPoolExecutor(

    max_workers=max_workers

) as executor:

    future_to_task = {

        executor.submit(self, task): task for task in tasks

    }

    results = []

    for future in concurrent.futures.as_completed(

        future_to_task

    ):

        task = future_to_task[future]

        try:

            img = future.result()

            results.append(img)


            print(f"Task {task} completed: {img}")
```

```
except Exception as error:
    print(
        colored(
            (
                f"Error running SSD1B: {error} try"
                " optimizing your api key and or try"
                " again"
            ),
            "red",
        )
    )
    print(
        colored(
            (
                "Error running SSD1B:"
                f" {error.http_status}"
            ),
            "red",
        )
    )
    print(
        colored(
            f"Error running SSD1B: {error.error}",
            "red",
        )
    )
```

```
            raise error

    def _generate_uuid(self):
        """Generate a uuid"""
        return str(uuid.uuid4())

    def __repr__(self):
        """Repr method for the SSD1B class"""
        return f"SSD1B(image_url={self.image_url})"

    def __str__(self):
        """Str method for the SSD1B class"""
        return f"SSD1B(image_url={self.image_url})"

    @backoff.on_exception(
        backoff.expo, Exception, max_tries=max_retries
    )
    def rate_limited_call(self, task: str):
        """Rate limited call to the SSD1B API"""
        return self.__call__(task)
```