File: api\alt_text.py

In [ ]:

```python
from fastapi import APIRouter, Depends, HTTPException
from pydantic import BaseModel
from models.blip_base import BLIPBase
from utils.image_processing import decode_base64_image
from .auth import get_api_key
import logging
from fastapi import Request
from pydantic import BaseModel
from typing import List, Union
from fastapi import APIRouter, Depends, HTTPException, Request
from models.blip_base import BLIPBase
from utils.image_processing import decode_base64_image
import logging
import base64
from typing import Dict
import asyncio

logger = logging.getLogger(__name__)

router = APIRouter()
model = BLIPBase()



class ImageUrl(BaseModel):
    url: str
    detail: str = None

class Content(BaseModel):
    type: str
    text: str = None
    image_url: ImageUrl = None

class Message(BaseModel):
    role: str
    content: List[Content]

class AltTextRequest(BaseModel):
    model: str
    messages: List[Message]
    max_tokens: int



class AltTextResponse(BaseModel):
    choices: List[dict]

class AltTextResponse(BaseModel):
    choices: List[Dict[str, Dict[str, str]]]

async def generate_alt_text_with_timeout(image, prompt, timeout=30):
    try:
        return await asyncio.wait_for(
            asyncio.to_thread(model.generate_alt_text, image, prompt),
            timeout=timeout
        )
    except asyncio.TimeoutError:
        raise HTTPException(status_code=504, detail="Alt text generation timed out")

@router.post("/generate_alt_text", response_model=AltTextResponse)
async def generate_alt_text(request: AltTextRequest, api_key: str = Depends(get_api_key)):
    try:
        logger.info(f"Received request: {request}")

        # Extract image and prompt from the request
        content = request.messages[0].content
        prompt = next(item.text for item in content if item.type == 'text')
        image_url = next(item.image_url.url for item in content if item.type == 'image_url')

        logger.info(f"Extracted prompt: {prompt}")
        logger.info(f"Extracted image URL (first 100 chars): {image_url[:100]}")

        # Decode base64 image
        image = decode_base64_image(image_url)

        # Generate alt text
        alt_text = await generate_alt_text_with_timeout(image, prompt)

        # Format response to match OpenAI's format
        response = {
            "choices": [
                {
                    "message": {
                        "content": alt_text
                    }
                }
```

```
            ]
        }

        return response
    except Exception as e:
        logger.error(f"Error generating alt text: {str(e)}", exc_info=True)
        raise HTTPException(status_code=500, detail=f"Error generating alt text: {str(e)}")
```

File: api\auth.py

In [ ]:
```
from fastapi import Security, HTTPException, status
from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer
from utils.config import settings

security = HTTPBearer()

async def get_api_key(credentials: HTTPAuthorizationCredentials = Security(security)):
    if credentials.credentials == settings.API_KEY:
        return credentials.credentials
    raise HTTPException(
        status_code=status.HTTP_403_FORBIDDEN, detail="Could not validate API key"
    )
```

File: main.py

In [ ]:
```
import logging
from fastapi import FastAPI
from api import auth, alt_text
from utils.config import settings
from fastapi.middleware.cors import CORSMiddleware
import logging
from fastapi import Request

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = FastAPI(title=settings.PROJECT_NAME)



app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Allows all origins
    allow_credentials=True,
    allow_methods=["*"],  # Allows all methods
    allow_headers=["*"],  # Allows all headers
)

from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc: RequestValidationError):
    logger.error(f"Validation error: {exc.errors()}")
    return JSONResponse(
        status_code=422,
        content={"detail": exc.errors(), "body": exc.body},
    )

app.include_router(alt_text.router, prefix=settings.API_V1_STR, tags=["alt_text"])

@app.on_event("startup")
async def startup_event():
    logger.info("Starting up the application")

@app.on_event("shutdown")
async def shutdown_event():
    logger.info("Shutting down the application")
```

File: models\blip_base.py

In [ ]:
```
import torch
from PIL import Image
from transformers import BlipForConditionalGeneration, BlipProcessor
from models.model_interface import ModelInterface
from utils.config import settings
import logging
import os

logger = logging.getLogger(__name__)

class BLIPBase(ModelInterface):
    def __init__(self):
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
```

```
        self.model_name = "Salesforce/blip-image-captioning-base"

        # Check if the model is already downloaded
        if not os.path.exists(settings.BLIP_MODEL_PATH):
            logger.info(f"BLIP model not found at {settings.BLIP_MODEL_PATH}. Downloading...")
            self.download_model()

        logger.info(f"Loading BLIP model from {settings.BLIP_MODEL_PATH}")
        self.processor = BlipProcessor.from_pretrained(settings.BLIP_MODEL_PATH)
        self.model = BlipForConditionalGeneration.from_pretrained(settings.BLIP_MODEL_PATH).to(self.device)
        logger.info("BLIP model loaded successfully")

    def download_model(self):
        try:
            # This will download and cache the model
            processor = BlipProcessor.from_pretrained(self.model_name)
            model = BlipForConditionalGeneration.from_pretrained(self.model_name)

            # Save the model to the specified path
            os.makedirs(settings.BLIP_MODEL_PATH, exist_ok=True)
            processor.save_pretrained(settings.BLIP_MODEL_PATH)
            model.save_pretrained(settings.BLIP_MODEL_PATH)

            logger.info(f"BLIP model downloaded and saved to {settings.BLIP_MODEL_PATH}")
        except Exception as e:
            logger.error(f"Error downloading BLIP model: {str(e)}")
            raise

    def generate_alt_text(self, image: Image.Image, prompt: str) -> str:
        prompt = ""
        try:
            logger.info(f"Generating alt text for image size: {image.size}")
            inputs = self.processor(image, prompt, return_tensors="pt").to(self.device)
            logger.info("Processed image with BLIP processor")
            output = self.model.generate(**inputs)
            logger.info("Generated output from BLIP model")
            return self.processor.decode(output[0], skip_special_tokens=True)
        except Exception as e:
            logger.error(f"Error in generate_alt_text: {str(e)}", exc_info=True)
            raise
```

File: models\model_interface.py

In [ ]:
```
from abc import ABC, abstractmethod
from PIL import Image

class ModelInterface(ABC):
    @abstractmethod
    def generate_alt_text(self, image: Image.Image, prompt: str) -> str:
        pass
```

File: utils\config.py

In [ ]:
```
import os
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    API_V1_STR: str = "/api/v1"
    PROJECT_NAME: str = "Moodle Alt Text API"
    ALGORITHM: str = "HS256"
    API_KEY: str = os.getenv("API_KEY", "your-default-api-key")
    BLIP_MODEL_PATH: str = os.getenv("BLIP_MODEL_PATH", os.path.expanduser("~/.cache/huggingface/blip-base"))

settings = Settings()
```

File: utils\image_processing.py

In [ ]:
```
import base64
from io import BytesIO
from PIL import Image
import logging

logger = logging.getLogger(__name__)

def decode_base64_image(base64_string):
    try:
        # Remove the "" part if it exists
        if 'base64,' in base64_string:
            base64_string = base64_string.split('base64,')[1]

        image_data = base64.b64decode(base64_string)
        logger.info(f"Decoded image data length: {len(image_data)}")

        image = Image.open(BytesIO(image_data))
```

```
        logger.info(f"Image opened successfully. Format: {image.format}, Size: {image.size}, Mode: {image.mode}")
        return image
    except Exception as e:
        logger.error(f"Error decoding base64 image: {str(e)}")
        logger.error(f"First 100 characters of base64 string: {base64_string[:100]}")
        raise

def encode_image_to_base64(image: Image.Image) -> str:
    buffered = BytesIO()
    image.save(buffered, format="PNG")
    return base64.b64encode(buffered.getvalue()).decode("utf-8")
```