```
"""
天気情報を提供するMCPサーバー (MCP標準プロンプト形式対応版)

このサーバーは以下の機能を提供します:
1. 都市名から天気情報を取得(wttr.in APIを使用)
2. 天気情報のキャッシュ管理(日次更新)
3. キャッシュの統計情報取得とクリア
4. MCP標準形式のプロンプト提供(getPrompts)

主要コンポーネント:
- WeatherCache: 天気情報のキャッシュを管理するクラス
- get_weather: 天気情報を取得するツール
- clear_cache: キャッシュをクリアするツール
- get_cache_stats: キャッシュ統計を取得するツール
- weather_forecaster: 天気予報用のシステムプロンプト(MCP標準形式)
- detailed_weather_report: 詳細レポート用プロンプト(MCP標準形式)

キャッシュ戦略:
- 同じ都市の情報は1日1回のみAPIから取得
- 前日以前のデータは自動的に削除
- JSONファイルに永続化
"""
import requests
from fastmcp import FastMCP
from weather_cache import WeatherCache
from logger_handler import LoggerHandler
from config_handler import ConfigHandler

PROMPT="""You are a professional weather forecaster specializing in Japanese weather reports.
Your role:
- Provide accurate and easy-to-understand weather forecasts in Japanese
- Focus on practical information that helps people plan their day
- Use clear, concise language suitable for general audiences

Output format for {city}:
今日の{city}は[天気]です。気温は[温度]度で、[体感の説明]でしょう。

Guidelines:
1. 都市名を明確に記載すること
2. 気温を摂氏(° C)で表示すること
3. 天気の状態を簡潔に説明すること(晴れ、曇り、雨、雪など)
4. 体感温度について簡単なコメントを追加すること(暑い、涼しい、寒いなど)
5. 必要に応じて服装や外出時のアドバイスを1文で追加すること

Example output:
今日の{city}は晴れです。気温は22度で、過ごしやすい陽気でしょう。軽めの上着があると安心です。
"""


class MikeWeatherMCPServer:

    def __init__(self, config):
        # 設定の読み込み
        self.config = config
        self.logger = LoggerHandler(__name__, self.config["LOG_LEVEL"]).get_logger()
        srv = self.config.get("WEATHER_MCP", {})

        # サーバー用パラメタ設定
        self.SERVER_NAME = str(srv.get("SERVER_NAME", "mike_mcp"))
        self.SERVER_HOST = srv.get("SERVER_HOST", "0.0.0.0")
        self.SERVER_PORT = int(srv.get("SERVER_PORT", 5000))
        #トランスポート stdio ot streamable-http or sse
        self.TRANSPORT = str(srv.get("TRANSPORT", "stdio"))

        # 天気APIのベースURL
        self.WEATHER_API_BASE_URL = str(srv.get("WEATHER_API_BASE_URL", "https://wttr.in"))
        # APIリクエストのタイムアウト(秒)
        self.API_REQUEST_TIMEOUT = int(srv.get("API_REQUEST_TIMEOUT", 10))
        # キャッシュファイルのパス
        self.CACHE_FILE_PATH = str(srv.get("CACHE_FILE_PATH", "cache/weather_cache.json"))
        self.cache = WeatherCache(self.CACHE_FILE_PATH, self.logger)
        self.mcp = FastMCP(self.SERVER_NAME)

          # ツールをMCPサーバに登録
        self._register_tools()

    def _register_tools(self):
        # MCPツールを登録
        self.mcp.tool()(self.get_weather)
        self.mcp.tool()(self.clear_cache)
        self.mcp.tool()(self.get_cache_stats)

    def get_weather(self, city: str) -> dict:
        """定された都市の現在の天気情報を取得します。気温と天候状態を含みます。
    この関数はMCPツールとして使用されます。

    MCP Tool Definition:
        name: get_weather
```

```
        description: 指定された都市の現在の天気情報(気温、天候状態)を取得します。
                     日本語・英語の都市名に対応しています。

        inputSchema:
            city:
                type: string
                description: 天気情報を取得する都市名(例: Tokyo, London, New York, 東京)
                minLength: 1

    Args:
        city (str): 天気情報を取得したい都市名
                    英語表記 (例: "Tokyo", "London", "New York")
                    または日本語表記 (例: "東京", "大阪")

    Returns:
        dict: 天気情報を含む辞書
            成功時:
                - city (str): 都市名
                - temp_C (str): 摂氏温度
                - desc (str): 天気の説明 (英語)
                - cached (bool): キャッシュから取得したかどうか

            エラー時:
                - city (str): 都市名
                - error (str): エラーメッセージ
                - cached (bool): False
    Note:
        - APIタイムアウト: 10秒
        - キャッシュは日次で管理 (前日のデータは使用されない)
        - APIエラー時はエラー辞書を返す (例外は発生させない)
    Examples:
        >>> get_weather("Tokyo")
        {
            "city": "Tokyo",
            "temp_C": "15",
            "desc": "Partly cloudy",
            "cached": False
        }
        >>> get_weather("存在しない都市")
        {
            "city": "存在しない都市",
            "error": "City not found",
            "cached": False
        }
"""
        self.logger.info(f"get_weather called: {city}")

        # ---- 入力チェック ----
        if not city or not isinstance(city, str):
            return {"city": city, "error": "都市名が不正です", "cached": False}

        # ---- キャッシュ確認 ----
        cached = self.cache.get(city)
        if cached:
            return {"city": city, "cached": True, **self._format_weather(cached)}

        # ---- 天気APIリクエスト ----
        url = f"{self.WEATHER_API_BASE_URL}/{city}?format=j2"

        try:
            res = requests.get(url, timeout=self.API_REQUEST_TIMEOUT)
            res.raise_for_status()
            data = res.json()
        except Exception as e:
            return {"city": city, "error": str(e), "cached": False}

        # ---- キャッシュ保存 ----
        self.cache.set(city, data)
        return {"city": city, "cached": False, **self._format_weather(data)}

    def _format_weather(self, data: dict) -> dict:
        """
        APIまたはキャッシュから取得した天気データを統一フォーマットへ変換
        """
        try:
            current = data.get("current_condition", [{}])[0]
            return {
                "temp_C": current.get("temp_C"),
                "desc": current.get("weatherDesc", [{"value": "不明"}])[0].get("value", "不明"),
            }
        except Exception as e:
            self.logger.error(f"天気データ解析エラー: {e}")
            return {"temp_C": None, "desc": "不明"}


    def clear_cache(self) -> dict:
        """
        全てのキャッシュをクリア
```

```
177:
178:         Returns:
179:             int: クリアされたエントリ数
180:
181:         Raises:
182:             CacheError: キャッシュファイルの保存に失敗した場合
183:         """
184:         try:
185:             count = self.cache.clear_all()
186:             return {"status": "success", "cleared_count": count}
187:         except Exception as e:
188:             self.logger.error(f"キャッシュクリア失敗: {e}")
189:             return {"status": "error", "message": str(e)}
190:
191:     def get_cache_stats(self) -> dict:
192:         """
193:         キャッシュの統計情報を取得
194:
195:         Returns:
196:             dict: 統計情報
197:                 - total_entries: 総エントリ数
198:                 - today_entries: 本日のエントリ数
199:                 - cities: キャッシュされている都市のリスト
200:                 - date: 現在の日付
201:         """
202:         try:
203:             stats = self.cache.get_stats()
204:             return {"status": "success", **stats}
205:         except Exception as e:
206:             self.logger.error(f"統計情報取得失敗: {e}")
207:             return {"status": "error", "message": str(e)}
208:
209:     def run(self):
210:         """MCPサーバ起動"""
211:         if (self.TRANSPORT == "stdio"):
212:             self.mcp.run()
213:         else:
214:             self.mcp.run(
215:                 transport=self.TRANSPORT,
216:                 host=self.SERVER_HOST,
217:                 port=self.SERVER_PORT
218:             )
219:
220: if __name__ == "__main__":
221:     print("MikeWeatherMCPServer...")
222:     server = MikeWeatherMCPServer('config/config_weather.json')
223:     print("Starting the server...")
224:     server.run()
```