LeemanF Update README.md c52ab12 · now History

Preview Code Blame 263 lines (196 loc) · 12.6 KB Raw


Leeman Frank

**Eindproef Data-Scientist1 - 25/26**

# Evolutie productie zonne- en windenergie

Laatste update 06/07/2025

Voor de opleiding Data-Scientist werd gevraagd om een eindproef in Python te maken met de focus op het ETL-proces:

- **Extract**: het binnenhalen van de data
- **Transform**: het bewerken (opkuisen) van de data
- **Load**: het analyseren van de opgekuiste data

De opdracht moest voldoende complex maar haalbaar zijn, en bovendien in staat om nieuwe of bijkomende data te verwerken.

Ik koos voor een analyse van de productie van zonne- en windenergie, gecombineerd met de Belpex-spotmarktprijzen.

## 🛢️ Databronnen

### Elia

Elia stelt via hun website gegevens beschikbaar:

- Zonne-energieproductie
- Windenergieproductie

```
UPDATE 06/07/2025: Elia biedt niet langer de data aan via een gewone download op hun website en verplicht het gebruik van API
```

Omdat Elia ook een API aanbiedt, werd gekozen voor deze stabielere oplossing:

- Dataset zonne-energie (ODS031)
- Dataset windenergie (ODS032)

Meer informatie over de aangeboden data vind je hier:

- Voorbeeld van opgehaalde windgegevens: Wind.json
- Voorbeeld van opgehaalde zonnegegevens: Solar.json

### Elexys (Belpex)

De Belpex-spotmarktprijzen zijn beschikbaar via de Elexys-website:
https://my.elexys.be/MarketInformation/SpotBelpex.aspx

Omdat er geen API beschikbaar is, werd hiervoor gebruik gemaakt van **webscraping**.

## 🧱 Projectopbouw

### Configuratie (`settings.py`)

De standaardlocaties van de data- en outputbestanden zijn configureerbaar via het bestand `settings.py`.

### Extractie van gegevens (`data_import_tools.py`)

Het ophalen van data gebeurt via het script `data_import_tools.py`. De belangrijkste functies zijn:

- `import_wind(year, month)`
- `import_solar(year, month)`
- `import_belpex(year, month)`

Omdat Elia maximaal 100 records per request toelaat, werd ervoor gekozen om de data dag per dag op te halen en afzonderlijk op te slaan.

#### Foutafhandeling

Om fouten tijdens het ophalen van data op te vangen, wordt gebruik gemaakt van:

- `retry_on_failure(...)` : een decorator die herhaalde pogingen toelaat bij falende requests.
- `safe_requests_get(...)` : een robuustere versie van `requests.get()` met retry- en timeout-logica.

Hierdoor worden netwerkproblemen automatisch opgevangen met maximaal twee extra pogingen.

#### Gegevensopslag en compressie

De opgehaalde Elia-data worden opgeslagen als dagelijkse JSON-bestanden.
Om opslagruimte te beperken en versiebeheer te vereenvoudigen, worden deze per maand gecomprimeerd (.zip).
Dit maakt het eenvoudiger om data op GitHub te beheren en voorkomt overbodige duplicatie in versiebeheer.
De zip-bestanden bevatten steeds een volledige maand per type (zonne- of windenergie), wat ook het manueel beheren vergemakkelijkt.

Belangrijke functies:

- `file_needs_zip(...)`
- `zip_forecast_data(...)`
- `unzip_forecast_data(...)`
- `unzip_all_forecast_zips(...)`

Het hele proces kan automatisch uitgevoerd worden met:

- `update_data()` : deze functie automatiseert het ophalen, unzippen, verwerken en opslaan van alle data over een gekozen periode.

## Transformatie van gegevens ( `database_tools.py` )

De module `database_tools.py` bevat de klassen en functies die nodig zijn om de data op te slaan in een SQLite-database.
Volgende klassen vormen het fundament van het SQLAlchemy-model:

- `class SolarData` : model voor het creëren en vullen van de tabel `solar_data`
- `class WindData` : model voor het creëren en vullen van de tabel `wind_data`
- `class BelpexPrice` : model voor het creëren en vullen van de tabel `belpex_prices`

Een overzicht van de beschikbare modellen en hun kolommen is terug te vinden via de functie `alle_modellen_en_kolommen()` in de module
`sqlalchemy_model_utils.py` .

### Verrijking van de data

De verkregen data wordt in alle modellen aangevuld met extra tijdsdimensies in de vorm van onderstaande kolommen:

- `day`
- `month`
- `year`
- `weekday`
- `hour`
- `minute`

Deze extra tijdsdimensies maken het mogelijk om flexibel te groeperen en te visualiseren op dag-, week-, maand-, weekdag- of uurniveau.

**Toevoegen records aan database**

De JSON-bestanden (Elia) en de CSV-bestanden (Belpex-prijzen) worden op een andere manier verwerkt.
Het verrijken van de JSON-bestanden gebeurt via de functie `parse_record()`.
Bij de CSV-bestanden wordt de extra data toegevoegd bij het inlezen.

- `process_directory()` : verwerking van de zonne- en windenergiedata
- `process_belpex_directory()` : opkuisen, verrijken en verwerken van de Belpex-prijzen

De functie `insert_batch()` maakt connectie met de database en verstuurt de records in batches.
Als de verwerking per batch fouten oplevert, worden de records afzonderlijk verwerkt.
Zo gaat er bij een fout in één record geen volledige batch verloren.

Er is bewust gekozen om alle beschikbare data te laten doorstromen naar de database.
Hierdoor blijft alle data beschikbaar voor extra analyses in de toekomst.

Het volledige proces van transformeren en verwerken wordt samengebracht in de functie `to_sql()`.

## Automatische update (`auto_update.py`)

Het script `auto_update.py` automatiseert zowel het ophalen als het verwerken van de data.
De functies `update_data()` en `to_sql()` worden hierbij binnen de contextmanager `DualLogger()` uitgevoerd.
Dit script kan via de Windows Taakplanner automatisch op maandelijkse basis uitgevoerd worden: zie voorbeeld

**Logging (`dual_logger.py`)**

De klasse `DualLogger()` zorgt ervoor dat alle console-uitvoer ook naar een logbestand geschreven wordt: zie voorbeeld.
Hoewel de logging-module de professionele standaard is, biedt DualLogger in het kader van dit eindproject een eenvoudige, robuuste en onderhoudsarme manier om zowel standaarduitvoer als foutmeldingen en tqdm-voortgangsbalken simultaan te loggen — zonder dat ik alle print()-aanroepen moest herschrijven. Voor grotere projecten zou ik uiteraard de logging-module verkiezen.

### Laden en visualiseren van de data

*Under construction*
In een latere fase worden interactieve grafieken en samenvattende visualisaties toegevoegd op basis van de opgeladen data.

## 🔧 Installatie

1. Zorg voor een recente Python-omgeving (3.10+ aanbevolen).
2. Installeer vereiste modules (bij voorkeur manueel, maar dit verloopt ook automatisch via het importeren van de scripts): zie requirements.txt

```
pip install -r requirements.txt
```

3. Zorg dat ChromeDriver of EdgeDriver beschikbaar is (wordt automatisch beheerd via `webdriver_manager`, geen handmatige installatie nodig).

## 🗃️ Database

De SQLite-database bevindt zich standaard in:
`./Database/energie_data.sqlite`

Tabellen:

- tbl_solar_data
- tbl_wind_data
- tbl_belpex_prices

Elke tabel bevat indexen op datetime, jaar, maand, dag, weekdag en uur, en gebruikt unieke constraints om duplicaten te vermijden.

Views:

- `v_solar`
- `v_wind`
- `v_belpex`

## 📚 Documentatie

De volledige code is voorzien van duidelijke docstrings (conform de PEP 257-stijl) en inline commentaar waar nodig.
Dit vergemakkelijkt het onderhoud, hergebruik en uitbreiding van het project.

## 🧩 Herbruikbaarheid code

De code is modulair opgebouwd, met een duidelijke scheiding tussen extractie, transformatie/opslag en — in de toekomst — visualisaties.
Veelgebruikte logica is ondergebracht in herbruikbare hulpfuncties binnen de `utils` -map.
Hierdoor is het eenvoudig om het project uit te breiden met andere databronnen of opslagstructuren.

## 📂 Verwachte mappenstructuur

```
Project/
├── .gitignore                      # Bestanden/mappen uitgesloten van versiebeheer
├── auto_update.py                  # Script voor automatische updates van data
├── main.ipynb                      # Hoofdnotebook voor analyse en/of visualisaties
├── README.md                       # Beschrijving van het project
├── requirements.txt                # Vereiste Python-pakketten
├── settings.py                     # Centrale instellingen (paden, parameters)
├── Data/                           # Bevat alle geïmporteerde data
│   ├── Belpex/                     # CSV-bestanden met Belpex-marktprijzen
│   │   ├── Belpex_202001.csv
│   │   ├── Belpex_202002.csv
│   │   └── ...
│   ├── SolarForecast/              # Zonneproductievoorspellingen en - metingen (JSON & ZIP)
│   │   ├── SolarForecast_2020.zip
│   │   ├── ...
│   │   ├── 2025/
│   │   │   ├── SolarForecast_Elia_20250425.json
│   │   │   └── ...
│   └── WindForecast/               # Windproductievoorspellingen en - metingen (JSON & ZIP)
│       ├── WindForecast_2020.zip
│       ├── ...
│       ├── 2025/
```

```
│       │   ├── WindForecast_Elia_20250425.json
│       │   └── ...
├── Database/
│   └── energie_data.sqlite             # SQLite-database met gestructureerde gegevens
├── Documents/                          # Bijkomende documentatie van het project
│   ├── Solar.json
│   ├── Wind.json
│   ├── Images/                         # Afbeeldingen die gebruikt worden in README.md
│   │   ├── Banner.png
│   │   └── ...
├── Log/                                # Logbestanden gegenereerd door scripts
│   └── log_YYYY-MM-DD.txt
├── src/                                # Broncode van het project (modulair opgebouwd)
│   ├── __init__.py
│   ├── data_import_tools.py            # Importtools voor verschillende databronnen
│   ├── database_tools.py               # Tools voor interactie met SQLite
│   └── utils/                          # Algemene hulpfuncties en helpers
│       ├── __init__.py
│       ├── constants_inspector.py      # Inspecteert de datakolommen en types
│       ├── decorators.py               # Decorators zoals retry_on_failure
│       ├── dual_logger.py              # Print + logfile logging in één
│       ├── package_tools.py            # Controle en installatie van dependencies
│       └── safe_requests.py            # Veilige HTTP-requests met retries
```

## 🌐 Gebruikte bronnen en documentatie

Tijdens de ontwikkeling van dit project werden volgende websites geraadpleegd:

### Data- en API-bronnen

- Elia Grid Data — overzichtspagina van de Elia-webinterface
- Elia Open Data — officiële datasets voor zonne- en windenergie

- Elexys - Spotmarktprijzen — bron van Belpex-prijzen

### Python, tools en documentatie

- Python Documentation — officiële Python documentatie
- SQLAlchemy Documentation
- webdriver-manager — automatische driverinstallatie voor Selenium
- retrying package — decorator voor herhaalpogingen
- PEP 257 – Docstring Conventions — richtlijnen voor docstrings
- Stack Overflow — veelgebruikte bron voor specifieke codevragen
- Real Python — heldere tutorials en uitleg over Python-concepten
- How to redirect stdout and stderr to logger in Python

## 🤖 Ondersteuning via ChatGPT

Bij het opzetten van dit project werd ChatGPT gebruikt als aanvulling op eigen onderzoek en ontwikkeling.
De tool diende vooral ter ondersteuning bij:

- Het verfijnen van Python-syntax en foutafhandeling
- Het opzoeken van documentatie en best practices
- Het uitschrijven van bepaalde functies of decoratoren
- Het herformuleren van uitleg of commentaar in de code
- Het structureren van de `README.md` in heldere markdownstijl

# Inhoudsopgave



## Inhoud uit script: `auto_update.py`

In [ ]:
```
r"""
auto_update.py

Automatisch update-script voor periodieke datavernieuwing.

Beschrijving:
- Dit script is bedoeld om automatisch uitgevoerd te worden via Windows Taakplanner.
- Het importeert de laatste energiegegevens (bv. van Elia en Belpex) en schrijft deze weg naar een lokale SQL-database.
- Alle output wordt zowel weergegeven op het scherm als gelogd in een bestand.

Modules:
```

```
- data_import_tools: bevat logica voor ophalen en verwerken van data.
- database_tools: bevat logica voor het wegschrijven naar een SQL-database.
- dual_logger: bevat logica om de printopdrachten ook naar een logbestand te schrijven.
- settings: bevat alle globale variabelen.

Gebruik:
- Inplannen via Windows Task Scheduler (bijv. maandelijks op de 5de dag).


Registratie in Windows Taakplanner:
1. Open de Taakplanner (Task Scheduler) via het startmenu.
2. Klik in het rechterpaneel op "Taak maken" (niet "Basis-taak maken" voor meer controle).
3. Op het tabblad **Algemeen**:
  - Geef de taak een herkenbare naam, bv. `Auto Update Script`.
  - Kies "Uitvoeren ongeacht of gebruiker is aangemeld" (optioneel) en geef indien nodig wachtwoord op.
4. Op het tabblad **Triggers**:
  - Klik op "Nieuw..." en stel in wanneer de taak moet worden uitgevoerd (bv. dagelijks om 06:00).
5. Op het tabblad **Acties**:
  - Klik op "Nieuw..." en kies bij "Actie": `Programma starten`.
  - Vul bij "Programma/script" het pad in naar je Python-interpreter, bv.:
    `C:\Users\<gebruikersnaam>\Anaconda3\python.exe`
  - Vul bij "Parameters toevoegen" het pad in naar dit script (tussen dubbele aanhalingstekens!), bv.:
    `C:\pad\naar\project\auto_update.py`
  - **Beginnen in (optioneel)**: vul hier de map in waarin het script zich bevindt, zonder aanhalingstekens.
    Bijvoorbeeld:
    `C:\pad\naar\project`
    Dit is belangrijk als je script relatieve paden gebruikt (zoals voor logbestanden of instellingen).
  - Of gebruik een `.bat`-bestand dat de juiste omgeving activeert en daarna het script uitvoert (aanbevolen).
6. (Optioneel) Op het tabblad **Instellingen** kun je herstartpogingen inschakelen als het mislukt.
7. Klik op OK om de taak op te slaan.
"""

import os
import traceback
from datetime import datetime

from src.data_import_tools import update_data
from src.database_tools import to_sql
from src.utils.dual_logger import DualLogger
from settings import LOG_DIR
```

```
# -------- Logging Setup --------

# Maak een map aan voor logbestanden (indien die nog niet bestaat)
os.makedirs(LOG_DIR, exist_ok=True)

# Stel het logbestand in met als naam het huidige datumformaat (log_YYYY-MM-DD.txt)
log_filename = datetime.now().strftime("log_%Y-%m-%d.txt")
log_path = os.path.join(LOG_DIR, log_filename)


# -------- Scriptuitvoering --------

with DualLogger(log_path):
    print("================================================================================================")
    print(f"🕒 Start auto-update: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")

    try:
        update_data()
        to_sql()
        print(f"\n✅ Update afgerond: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")
    except Exception as e:
        print(f"\n❌ Fout tijdens update: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')} - {e}")
        print("\n--------------------------------------------------------------------------------\n")
        print(traceback.format_exc())
        print("--------------------------------------------------------------------------------\n")
```

## Inhoud uit notebook: `main.ipynb`

In [ ]:
```
# Installeren van de vereiste modules
# OPGELET: dit moet slechts 1x uitgevoerd worden!
%pip install -r requirements.txt
```

In [ ]:
```
from src.database_tools import Base
from src.utils.sqlalchemy_model_utils import alle_modellen_en_kolommen

# afdrukken van de SQLAlchemy-modellen en hun bijhorende kolommen met de eventule omschrijvingen van Elia
print(alle_modellen_en_kolommen(Base))
```

In [ ]:
```python
from src.utils.constants_inspector import list_module_constants
import settings

# afdrukken van de huidige standaardwaarden van de bestandsnamen en paden
constants = list_module_constants(settings)
```

In [ ]:
```python
from src.data_import_tools import update_data
from src.database_tools import to_sql

# updaten van de data-bestanden (JSON en CSV) en bijwerken van de sqlite database
# bij voorkeur uitvoeren via auto_update.py

if input("Mogen de bestanden en de database bijgewerkt worden? OPGELET: dit duurt lang! (J/N)").upper() == "J":
    #update_data(data_type="belpex")
    update_data()
    #to_sql("belpex")
    to_sql()
```

## Inhoud uit tekstbestand: `requirements.txt`

In [ ]:
```
requests>=2.25.0
selenium>=4.1.0
webdriver_manager>=3.5.0
sqlalchemy>=2.0.0
tqdm>=4.60.0
```

## Inhoud uit script: `settings.py`

In [ ]:
```python
# settings.py

from pathlib import Path

# ————————————————————————————————————————————————————————
# Basisdirectory van de datamappen
BASE_DIR = Path(__file__).resolve().parent  # de map waarin settings.py staat
```

```
#BASE_DIR = Path("D:/Eindwerk")

# Scripts en vereisten in de projectroot
REQUIREMENTS_FILE = BASE_DIR / "requirements.txt"
AUTO_UPDATE_SCRIPT = BASE_DIR / "auto_update.py"
MAIN_SCRIPT = BASE_DIR / "main.ipynb"

# ——————————————————————————————————————————————————————
# Package-map
SRC_DIR = BASE_DIR / "src"

# ——————————————————————————————————————————————————————
# Data-dir met de submappen Belpex, SolarForecast, WindForecast
DATA_DIR            = BASE_DIR / "Data"
BELPEX_DIR          = DATA_DIR / "Belpex"
SOLAR_FORECAST_DIR  = DATA_DIR / "SolarForecast"
WIND_FORECAST_DIR   = DATA_DIR / "WindForecast"

# ——————————————————————————————————————————————————————
# Log-directory
LOG_DIR = BASE_DIR / "Log"

# ——————————————————————————————————————————————————————
# Database-bestanden
DB_DIR  = BASE_DIR / "Database"
DB_FILE = DB_DIR / "energie_data.sqlite"

# ——————————————————————————————————————————————————————
# (Optioneel) Timeouts, retries, etc.
HTTP_TIMEOUT = 10  # default timeout voor API-calls
DEFAULT_ATTEMPTS = 3    # default aantal pogingen bij fouten
RETRY_DELAY   = 5  # default wachttijd bij retry
```

## Inhoud uit script: `src/__init__.py`

In [ ]:

## Inhoud uit script: `src/data_import_tools.py`

In [ ]:

```
"""
data_import_tools.py

Data Import Tools for Elia Open Data & Belpex Market Data.

Functies:
- Automatische installatie van vereiste Python-modules.
- Ophalen en lokaal opslaan van wind- en zonne-energievoorspellingen en -metingen (per dag) (JSON).
    ° windenergie: https://opendata.elia.be/explore/dataset/ods031/information/
    ° zonne-energie: https://opendata.elia.be/explore/dataset/ods032/information/
- Ophalen van Belpex-spotmarktprijzen via webscraping (CSV).
- Zippen en unzippen van de json-bestanden (wind- en zonne-energie) per jaar.
- Opvangen van netwerkfouten en browserproblemen via retry-mechanismen.
"""

# ----------- Imports -----------

import os
import calendar
import json
import time
import shutil
from datetime import datetime
import zipfile

from src.utils.package_tools import update_or_install_if_missing
from src.utils.decorators import retry_on_failure
from settings import HTTP_TIMEOUT, DEFAULT_ATTEMPTS, RETRY_DELAY, BELPEX_DIR, SOLAR_FORECAST_DIR, WIND_FORECAST_DIR, BASE_DIR

# Controleer en installeer indien nodig de vereiste modules
# Dit is een vangnet als de gebruiker geen rekening houdt met requirements.txt.
update_or_install_if_missing("requests","2.25.0")
update_or_install_if_missing("selenium","4.1.0")
update_or_install_if_missing("webdriver_manager","3.5.0")

# Pas na installatie importeren
```

```python
from src.utils.safe_requests import safe_requests_get
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# ----------- Data Import Functies -----------

@retry_on_failure(tries=DEFAULT_ATTEMPTS, delay=RETRY_DELAY)
def import_wind(year,month):
    """
    Download en sla windforecast-data op voor een opgegeven maand uit de Elia Open Data API.

    Deze functie haalt voor elke dag van de opgegeven maand de windvoorspellingsgegevens en metingen op
    via de Elia API en slaat deze lokaal op in afzonderlijke JSON-bestanden per dag, per jaar gestructureerd.

    Indien er een netwerkprobleem of andere tijdelijke fout optreedt tijdens het ophalen,
    wordt de volledige functie automatisch herhaald tot 3 keer dankzij de retry-decorator.

    Parameters:
    - year (int): Het jaar waarvoor data opgehaald moet worden.
    - month (int): De maand waarvoor data opgehaald moet worden (1 t.e.m. 12).

    Opmerkingen:
    - Bestanden worden opgeslagen in: Data\WindForecast\<jaar>\WindForecast_Elia_YYYYMMDD.json
    - Bestanden die al bestaan worden niet opnieuw gedownload.
    """

    # Bepaal het aantal dagen in de gevraagde maand en jaar
    _, num_days = calendar.monthrange(year, month)

    # Stel pad samen voor jaarspecifieke map en maak deze aan indien nodig
    year_folder = os.path.join(WIND_FORECAST_DIR, str(year))
    os.makedirs(year_folder, exist_ok=True)

    # Loop over elke dag van de maand
    for day in range(1, num_days + 1):
        # Datum omzetten naar formaat YYYY-MM-DD (vereist door Elia API)
        date_str = f"{year}-{month:02d}-{day:02d}"
```

```
        # Bestandsnaam en volledig pad genereren voor output
        output_filename = f"WindForecast_Elia_{year}{month:02d}{day:02d}.json"
        output_path = os.path.join(year_folder, output_filename)

        # Indien het bestand reeds bestaat, sla deze dag over
        if os.path.exists(output_path):
            #print(f"✅ Bestand bestaat al: {output_filename}")
            continue

        print(f"      ⬇️ Ophalen: {output_filename}")

        # Basis-URL van de Elia API voor winddata
        url = "https://opendata.elia.be/api/explore/v2.1/catalog/datasets/ods031/records"
        all_records = []
        limit = 100  # Elia legt een beperking op van 100 records per call
        offset = 0

        while True:
            # API-parameters inclusief filter op specifieke dag
            params = {
                "order_by": "datetime",          # Sorteer op tijd
                "limit": limit,                  # Aantal records per batch
                "offset": offset,                # Startpunt voor batch
                "refine": [
                    f'datetime:"{date_str}"'     # Filter op specifieke dag
                ]
            }

            # Voer het verzoek uit via de veilige request-functie met retry
            response = safe_requests_get(url, params=params,tries=DEFAULT_ATTEMPTS,delay=RETRY_DELAY,timeout=HTTP_TIMEOUT)

            # Extra controle op HTTP-status (niet echt nodig door raise_for_status(), maar extra informatief)
            if response.status_code != 200:
                print(f"      ❌ Fout bij {date_str} (offset {offset}): {response.status_code}")
                break

            # Haal JSON-gegevens op, neem alleen 'results' (records)
            data = response.json().get("results", [])
            if not data:
                break  # Geen data meer, stop loop
```

```
            all_records.extend(data)
            if offset != 0:
                print(f'       ⏳ De eerste {offset} records werden binnengehaald.', end='\r')
            offset += limit

        # Als er data gevonden werd, sla deze op in JSON-bestand
        if all_records:
            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(all_records, f, ensure_ascii=False, indent=2)
            print(f"       ✅ Opgeslagen ({len(all_records)} records): {output_filename}")
        else:
            print(f"       ❌ Geen data voor {date_str}")

@retry_on_failure(tries=DEFAULT_ATTEMPTS, delay=RETRY_DELAY)
def import_solar(year,month):
    """
    Download en sla zonneforecast-data op voor een opgegeven maand uit de Elia Open Data API.

    Deze functie haalt voor elke dag van de opgegeven maand de zonvoorspellingsgegevens en metingen op
    via de Elia API en slaat deze lokaal op in afzonderlijke JSON-bestanden per dag, per jaar gestructureerd.

    Parameters:
    - year (int): Het jaar waarvoor data opgehaald moet worden.
    - month (int): De maand waarvoor data opgehaald moet worden (1 t.e.m. 12).

    Opmerkingen:
    - Bestanden worden opgeslagen in: Data\SolarForecast\<jaar>\SolarForecast_Elia_YYYYMMDD.json
    - Bestanden die al bestaan worden niet opnieuw opgehaald.
    - Enkel records voor de regio "Belgium" worden opgehaald om te vermijden dat er dubbele data is.
    """

    # Bepaal het aantal dagen in de gevraagde maand en jaar
    _, num_days = calendar.monthrange(year, month)

    # Stel pad samen voor jaarspecifieke map en maak deze aan indien nodig
    year_folder = os.path.join(SOLAR_FORECAST_DIR, str(year))
    os.makedirs(year_folder, exist_ok=True)

    # Loop over elke dag van de maand
    for day in range(1, num_days + 1):
        # Datum omzetten naar formaat YYYY-MM-DD (vereist door Elia API)
```

```
        date_str = f"{year}-{month:02d}-{day:02d}"

        # Bestandsnaam en volledig pad genereren voor output
        output_filename = f"SolarForecast_Elia_{year}{month:02d}{day:02d}.json"
        output_path = os.path.join(year_folder, output_filename)

        # Indien het bestand reeds bestaat, sla deze dag over
        if os.path.exists(output_path):
            #print(f"✅ Bestand bestaat al: {output_filename}")
            continue

        print(f"      ⬇️ Ophalen: {output_filename}")

        # Basis-URL van de Elia API voor zonnedata
        url = "https://opendata.elia.be/api/explore/v2.1/catalog/datasets/ods032/records"
        all_records = []
        limit = 100  # Elia legt een beperking op van 100 records per call
        offset = 0

        while True:
            # API-parameters inclusief filter op specifieke dag en regio
            params = {
                "order_by": "datetime",          # Sorteer op tijd
                "limit": limit,                  # Aantal records per batch
                "offset": offset,                # Startpunt voor batch
                "refine": [
                    f'datetime:"{date_str}"',    # Filter op specifieke dag
                    'region:"Belgium"'           # Filter op Belgische regio
                ]
            }

             # Voer het verzoek uit via de veilige request-functie met retry
            response = safe_requests_get(url, params=params,tries=DEFAULT_ATTEMPTS,delay=RETRY_DELAY,timeout=HTTP_TIMEOUT)

            # Extra controle op HTTP-status (niet echt nodig door raise_for_status(), maar extra informatief)
            if response.status_code != 200:
                print(f"      ❌ Fout bij {date_str} (offset {offset}): {response.status_code}")
                break

            # Haal JSON-gegevens op, neem alleen 'results' (records)
            data = response.json().get("results", [])
```

```
            if not data:
                break  # Geen data meer, stop loop

            all_records.extend(data)
            if offset != 0:
                print(f'      ⏳ De eerste {offset} records werden binnengehaald.', end='\r')
            offset += limit

        # Als er data gevonden werd, sla deze op in JSON-bestand
        if all_records:
            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(all_records, f, ensure_ascii=False, indent=2)
            print(f"      ✅ Opgeslagen ({len(all_records)} records): {output_filename}")
        else:
            print(f"      ❌ Geen data voor {date_str}")

@retry_on_failure(tries=DEFAULT_ATTEMPTS, delay=RETRY_DELAY)
def import_belpex(year, month):
    """
    Download Belpex-spotmarktprijzen via browserautomatisering (Selenium).

    Deze functie automatiseert het downloaden van maandelijkse Belpex-spotmarktprijzen
    van de Elexys-website met behulp van een headless (onzichtbare) Chrome-browser.
    De resultaten worden gedownload als CSV en opgeslagen met bestandsnaam 'Belpex_YYYYMM.csv'.

    Parameters:
    - year (int): Het jaar waarvoor data opgehaald moet worden.
    - month (int): De maand waarvoor data opgehaald moet worden (1 t.e.m. 12).

    Opmerkingen:
    - Gebruikt een headless Chrome-browser (geen visueel venster).
    - Downloadlocatie: ./Data/Belpex/Belpex_YYYYMM.csv
    - Indien het bestand reeds bestaat, wordt het niet opnieuw gedownload.
    """
    # Stel de 'from'-datum in op de eerste dag van de maand (dd/mm/yyyy)
    from_date = f"01/{month:02d}/{year}"

    # Bepaal de eerste dag van de volgende maand voor de 'until_date'
    if month == 12:
        next_month = 1
        next_year = year + 1
```

```
else:
    next_month = month + 1
    next_year = year

# De eerste dag van de volgende maand
next_month_first_day = datetime(next_year, next_month, 1)
until_date = next_month_first_day.strftime("%d/%m/%Y")

# Downloadpad instellen en aanmaken indien nodig
download_dir = str(BELPEX_DIR)
os.makedirs(download_dir, exist_ok=True)

# Setup voor Chrome
options = Options()
prefs = {
    "download.default_directory": download_dir,
    "download.prompt_for_download": False,
    "directory_upgrade": True,
    "safebrowsing.enabled": True
}
options.add_argument("--headless")  # Chrome wordt onzichtbaar geopend
options.add_experimental_option("prefs", prefs)
driver = webdriver.Chrome(options=options)


# Niet hernoemde bestanden opkuisen
if "BelpexFilter.csv" in os.listdir(download_dir):
    os.remove(os.path.join(download_dir, "BelpexFilter.csv"))
    print("      ❌ Niet hernoemde bestand BelpexFilter.csv werd verwijderd.")

new_filename = f"Belpex_{year}{month:02d}.csv"


# Indien het bestand reeds bestaat, sla deze maand over
if new_filename in os.listdir(download_dir):
    #print(f"✅ Bestand bestaat al: {new_filename}")
    pass
else:
    # Ga naar de website
    driver.get("https://my.elexys.be/MarketInformation/SpotBelpex.aspx")
```

```
        print(f"      ⬇️ Starten met het opvragen Belpex-gegevens periode {month}/{year}")

        # Wacht op beschikbaarheid van datumvelden
        wait = WebDriverWait(driver, 20)
        wait.until(EC.presence_of_element_located((By.ID, "contentPlaceHolder_fromASPxDateEdit_I")))

        # Vul de datums in
        from_input = driver.find_element(By.ID, "contentPlaceHolder_fromASPxDateEdit_I")
        until_input = driver.find_element(By.ID, "contentPlaceHolder_untilASPxDateEdit_I")

        print(f"      🗓️ Vul 'From' datum in: {from_date}")
        from_input.clear()
        from_input.send_keys(from_date)

        print(f"      🗓️ Vul 'Until' datum in: {until_date}")
        until_input.clear()
        until_input.send_keys(until_date)

        # Klik op "Show data"
        show_data_button = driver.find_element(By.ID, "contentPlaceHolder_refreshBelpexCustomButton_I")
        print("      🚀 Klik op 'Show data'")
        driver.execute_script("arguments[0].click();", show_data_button)

        # Wacht tot de resultaten zichtbaar zijn in de tabel
        print("      ⌛ Wacht op zoekresultaten...")
        wait.until(EC.presence_of_element_located((By.ID, "contentPlaceHolder_belpexFilterGrid_DXMainTable")))
        time.sleep(5)  # Extra wachttijd voor stabiliteit

        # Klik op de juiste export-div
        print("      🚀 Klik op 'Exporteer naar CSV'")
        export_button_div = wait.until(EC.element_to_be_clickable((By.ID, "ctl00_contentPlaceHolder_GridViewExportUserControl1
        driver.execute_script("arguments[0].click();", export_button_div)

        # Wacht op de download
        print("      ⌛ Wacht op download...")
        time.sleep(5)

        # Als het bestand bestaat, hernoem het bestand naar 'Belpex_JJJJMM.csv' (bijv. Belpex_202401.csv)
        if "BelpexFilter.csv" in os.listdir(download_dir):
            new_filename = f"Belpex_{year}{month:02d}.csv"
            os.rename(os.path.join(download_dir, "BelpexFilter.csv"), os.path.join(download_dir, new_filename))
```

```
            print(f"      ✅ Gedownload en hernoemd naar: {new_filename}")
        else:
            print("      ❌ Download mislukt.")

        # Sluit de browser
        driver.quit()

# ---------- Zip Functies ----------

def file_needs_zip(zip_path, folder_path):
    """
    Controleer of een ZIP-bestand ouder is dan de JSON-bestanden in een opgegeven map.

    Parameters:
    - zip_path (str): Pad naar het te controleren ZIP-bestand.
    - folder_path (str): Map waarin .json-bestanden zich bevinden.

    Returns:
    - bool:
        - True als:
            - het ZIP-bestand niet bestaat, of
            - één of meer JSON-bestanden in de map nieuwer zijn dan het ZIP-bestand.
        - False als:
            - alle JSON-bestanden ouder zijn dan het ZIP-bestand (zip is up-to-date).
    """
    if not os.path.exists(zip_path):
        return True  # Zip bestaat niet → zeker zippen

    zip_mtime = os.path.getmtime(zip_path)

    for root, _, files in os.walk(folder_path):
        for file in files:
            if file.endswith(".json"):
                file_path = os.path.join(root, file)
                if os.path.getmtime(file_path) > zip_mtime:
                    return True  # Bestand is recenter dan de zip → zip nodig
    return False  # Alles is ouder → zip is up-to-date

def zip_forecast_data(forecast_types=["SolarForecast", "WindForecast"]):
    """
    Maak ZIP-bestanden aan voor zonne- en windbestanden (JSON) per jaar en per type.
```

```
Parameters:
- forecast_types (list[str]): Lijst met types, zoals "SolarForecast" of "WindForecast".

Werking:
- Voor elk type en elk jaar wordt gecontroleerd of er een zip moet worden aangemaakt.
- Bestanden met extensie `.json` worden gebundeld in één zip per jaar.
- Bestandsstructuur binnen de zip wordt behouden relatief aan het forecasttypepad.
- Bestaat een zip reeds en is deze up-to-date, dan wordt deze overgeslagen.
"""
for forecast_type in forecast_types:
    if forecast_type == "WindForecast":
        type_folder = WIND_FORECAST_DIR
    elif forecast_type == "SolarForecast":
        type_folder = SOLAR_FORECAST_DIR
    else:
        type_folder = os.path.join(BASE_DIR, forecast_type)

    if not os.path.isdir(type_folder):
        print(f"    ⚠️ Map bestaat niet: {type_folder}")
        continue

    for year in os.listdir(type_folder):
        year_path = os.path.join(type_folder, year)

        if not os.path.isdir(year_path):
            continue  # Sla bestanden of vreemde dingen over

        zip_filename = f"{forecast_type}_{year}.zip"
        zip_path = os.path.join(type_folder, zip_filename)

        # Check of zip nodig is
        if not file_needs_zip(zip_path, year_path):
            print(f"    ⏭️ Up-to-date: {zip_filename}")
            continue

        print(f"    📦 Zippen van {year_path} → {zip_filename}")

        with zipfile.ZipFile(zip_path, 'w', zipfile.ZIP_DEFLATED) as zipf:  # 'w" overschrijft vorig bestand als dit besta
            for root, _, files in os.walk(year_path):
                for file in files:
```

```
                        if file.endswith(".json"):
                            file_path = os.path.join(root, file)
                            arcname = os.path.relpath(file_path, type_folder)
                            zipf.write(file_path, arcname)

            print(f"    ✅ Klaar: {zip_filename}")

def unzip_forecast_data(zip_path, extract_to=None):
    """
    Pak een zipbestand met forecastgegevens uit, met behoud van tijdstempels.

    Parameters:
    - zip_path (str): Pad naar het te unzippen bestand.
    - extract_to (str of None): Doelmap. Indien None, gebruik de map waarin het zipbestand zit.

    Werking:
    - Alleen nieuwe bestanden worden uitgepakt (bestaande worden overgeslagen).
    - Herstelt originele modificatietijd (modification time) per bestand.
    """
    if extract_to is None:
        extract_to = os.path.dirname(zip_path)

    with zipfile.ZipFile(zip_path, 'r') as zipf:
        for member in zipf.infolist():
            extracted_path = os.path.join(extract_to, member.filename)
            if os.path.exists(extracted_path):
                # Sla bestaande bestanden over
                continue
            os.makedirs(os.path.dirname(extracted_path), exist_ok=True)
            with zipf.open(member) as source, open(extracted_path, 'wb') as target:
                shutil.copyfileobj(source, target)
            # Zet de oorspronkelijke modificatie-tijd terug
            date_time = time.mktime(member.date_time + (0, 0, -1))
            os.utime(extracted_path, (date_time, date_time))
            print(f"       ✅ Uitgepakt: {member.filename}")

def unzip_all_forecast_zips(forecast_types=["SolarForecast", "WindForecast"]):
    """
    Pak alle zipbestanden uit in opgegeven forecastfolders.

    Parameters:
```

```
    - forecast_types (list[str]): Lijst van types met forecasts, bijvoorbeeld ["SolarForecast", "WindForecast"].

    Werking:
    - Zoekt in elk type-folder naar alle `.zip`-bestanden en roept `unzip_forecast_data()` op.
    - Alleen nieuwe bestanden worden uitgepakt.
    """
    for forecast_type in forecast_types:
        if forecast_type == "WindForecast":
            type_folder = WIND_FORECAST_DIR
        elif forecast_type == "SolarForecast":
            type_folder = SOLAR_FORECAST_DIR
        else:
            type_folder = os.path.join(BASE_DIR, forecast_type)

        if not os.path.isdir(type_folder):
            print(f"    ❌ Map niet gevonden: {type_folder}")
            continue

        for file in os.listdir(type_folder):
            if file.endswith(".zip"):
                zip_path = os.path.join(type_folder, file)
                print(f"    📦 Bezig met uitpakken: {file}")
                unzip_forecast_data(zip_path)

    print('')

# ----------- Bijwerken van de data-bestanden -----------

def update_data(from_year=None, to_year=None, data_type='all'):
    """
    Update wind-, zonne- en/of Belpex-data tussen opgegeven jaartallen.
    Hierbij worden eerste de bestaande zip-bestanden uitgepakt.
    Vervolgens wordt de recenste data opgehaald bij Elia en Elexys.
    Ten slotte wordt nieuwe data toegevoegd aan de zip-bestanden

    Parameters:
    - from_year (int, optional): Startjaar. Default = vorig jaar.
    - to_year (int, optional): Eindjaar. Default = huidig jaar.
    - data_type (str, optional): 'wind', 'solar', 'belpex' of 'all'. Default = 'all'.
    """
```

```
today = datetime.today()

# Defaults
if from_year is None:
    from_year = today.year - 1
if to_year is None:
    to_year = today.year

# Bepalen tot welke maand er data beschikbaar is. Pas vanaf de 4de dag is de info van de vorige maand beschikbaar.
if today.day <= 4:
    latest_available_month = today.month - 2
else:
    latest_available_month = today.month - 1

latest_available_year = today.year
if latest_available_month <= 0:
    latest_available_month += 12
    latest_available_year -= 1

allowed_types = {'wind', 'solar', 'belpex', 'all'}
if data_type not in allowed_types:
    raise ValueError(f"❌ Ongeldig data_type '{data_type}'. Kies uit {allowed_types}.")

# Altijd eerst de huidige bestanden unzippen als er gekozen werd voor 'wind' of 'solar'
if data_type in ('wind', 'solar', 'all'):
    print("\n📦 Unzippen van de forecast-data...")
    unzip_all_forecast_zips()


print(f"🗓️ Start met ophalen data voor periode {from_year}-{to_year}")
counter = 0
# Loop over jaren en maanden
for year in range(from_year, to_year + 1):
    for month in range(1, 13):
        if (year == latest_available_year and month > latest_available_month) or (year > latest_available_year):
            continue

        print(f"   🗓️ Ophalen data voor {year}-{month:02d} ({data_type})")

        if data_type in ('wind', 'all'):
            try:
```

```
                    import_wind(year, month)
                    counter += 1
                except Exception as e:
                    print(f"❌ Fout bij ophalen winddata {year}-{month:02d}: {e}")

            if data_type in ('solar', 'all'):
                try:
                    import_solar(year, month)
                    counter += 1
                except Exception as e:
                    print(f"❌ Fout bij ophalen zonndata {year}-{month:02d}: {e}")

            if data_type in ('belpex', 'all'):
                try:
                    import_belpex(year, month)
                    counter += 1
                except Exception as e:
                    print(f"❌ Fout bij ophalen Belpex-data {year}-{month:02d}: {e}")
    if counter == 0:
        print("    ❌ Geen data beschikbaar.")

    # Alleen als 'wind' of 'solar' werd geüpdatet: zip de forecast-data
    if data_type in ('wind', 'solar', 'all'):
        print("\n📦 Zippen van de forecast-data...")
        zip_forecast_data()

    print("\n✅ Data-import afgerond.\n")
```

## Inhoud uit script: `src/database_tools.py`

In [ ]:

```
"""
database_tools.py

Database Tools voor Elia Open Data & Belpex Market Data.

Functies:
- Automatische installatie van vereiste Python-modules.
- Definitie van SQLAlchemy-modellen voor zonne-energie, windenergie en Belpex-prijzen.
```

```
- Batchgewijs importeren van JSON- en CSV-data naar een SQLite-database.
- Automatische parsing en verrijking van datetime-informatie.
- Selectief verwerken van datasets via het `to_sql()`-commando.
"""

# ----------- Imports -----------

import os
import json
import csv
from datetime import datetime
import re
from src.utils.package_tools import update_or_install_if_missing

# Controleer en installeer indien nodig de vereiste modules
# Dit is een vangnet als de gebruiker geen rekening houdt met requirements.txt.
update_or_install_if_missing("sqlalchemy","2.0.0")
update_or_install_if_missing("tqdm","4.60.0")

# Pas na installatie importeren
from tqdm import tqdm
from sqlalchemy import create_engine, Column, Integer, String, Float, DateTime, UniqueConstraint, Index, text
from sqlalchemy.orm import declarative_base, sessionmaker
from sqlalchemy.dialects.sqlite import insert as sqlite_insert
from settings import DB_FILE, SOLAR_FORECAST_DIR, WIND_FORECAST_DIR, BELPEX_DIR

# Database setup:
# Initialisatie van de SQLite-engine en sessie, met automatische creatie van tabellen op basis van gedefinieerde modellen.
Base = declarative_base()
os.makedirs(os.path.dirname(DB_FILE), exist_ok=True)
engine = create_engine(f"sqlite:///{DB_FILE}")
Session = sessionmaker(bind=engine)
session = Session()

# Models
class SolarData(Base):
    """
    SQLAlchemy-model voor het opslaan van zonne-energiegegevens in de database.

    Unieke combinatie: datetime + region
    Index op de kolommen datetime, year, month, day, weekday en hour
```

```
    """
    __tablename__ = "tbl_solar_data"
    id = Column(Integer, primary_key=True)
    datetime = Column(DateTime, nullable=False)
    year = Column(Integer)
    month = Column(Integer)
    day = Column(Integer)
    weekday = Column(Integer)
    hour = Column(Integer)
    minute = Column(Integer)
    resolutioncode = Column(String, info={"beschrijving": "Length of the time interval expressed in compliance with ISO 8601."
    region = Column(String, info={"beschrijving": "Location of the production unit."})
    measured = Column(Float, info={"beschrijving": "The value running average measured for the reported time interval."})
    monitoredcapacity = Column(Float, info={"beschrijving": "Total available production capacity."})
    mostrecentforecast = Column(Float, info={"beschrijving": "Most recently forecasted volume."})
    mostrecentconfidence10 = Column(Float,
                                    info={"beschrijving": "Most recently forecasted volume with a probability of less than 10%
    mostrecentconfidence90 = Column(Float,
                                    info={"beschrijving": "Most recently forecasted volume with a probability of less than 10%
    dayahead11hforecast = Column(Float, info={"beschrijving": "Day-ahead forecasted volume published at 11AM. "})
    dayahead11hconfidence10 = Column(Float,
                                     info={"beschrijving": "Day-ahead forecasted volume with a probability of less than 10% th
                                     "published at 11AM."})
    dayahead11hconfidence90 = Column(Float,
                                     info={"beschrijving": "Day-ahead forecasted volume with a probability of less than 10% th
                                     "published at 11AM."})
    dayaheadforecast = Column(Float, info={"beschrijving": "Day-ahead forecasted volume to be produced."})
    dayaheadconfidence10 = Column(Float,
                                  info={"beschrijving": "Day-ahead forecasted volume with a probability of less than 10% that
    dayaheadconfidence90 = Column(Float, info={"beschrijving": "Forecasted volume with a probability of less than 10% that a h
    weekaheadforecast = Column(Float, info={"beschrijving": "Week-ahead forecasted volume."})
    weekaheadconfidence10 = Column(Float,
                                   info={"beschrijving": "Week-ahead forecasted volume with a probability of less than 10% tha
    weekaheadconfidence90 = Column(Float,
                                   info={"beschrijving": "Week-ahead forecasted volume with a probability of less than 10% tha
    loadfactor = Column(Float, info={"beschrijving": "The percentage ratio between measured power generation and the total mon
    __table_args__ = (
        UniqueConstraint('datetime', 'region', name='_datetime_region_uc'),
        Index('idx_solar_datetime', 'datetime'),
        Index('idx_solar_year', 'year'),
        Index('idx_solar_month', 'month'),
```

```
        Index('idx_solar_day', 'day'),
        Index('idx_solar_weekday', 'weekday'),
        Index('idx_solar_hour', 'hour'),
    )

class WindData(Base):
    """
    SQLAlchemy-model voor het opslaan van windenergiegegevens in de database.

    Unieke combinatie: datetime + region + offshoreonshore + gridconnectiontype
    Index op de kolommen datetime, year, month, day, weekday en hour
    """
    __tablename__ = "tbl_wind_data"
    id = Column(Integer, primary_key=True)
    datetime = Column(DateTime, nullable=False)
    year = Column(Integer)
    month = Column(Integer)
    day = Column(Integer)
    weekday = Column(Integer)
    hour = Column(Integer)
    minute = Column(Integer)
    resolutioncode = Column(String, info={"beschrijving": "Length of the time interval expressed in compliance with ISO 8601."
    offshoreonshore = Column(String, info={"beschrijving": "Indicates whether the wind farm is offshore or onshore."})
    region = Column(String, info={"beschrijving": "Location of the production unit."})
    gridconnectiontype = Column(String,
                                info={"beschrijving": "Indicates whether the production unit is connected to the Elia grid or
    measured = Column(Float, info={"beschrijving": "The value running average measured for the reported time interval."})
    monitoredcapacity = Column(Float, info={"beschrijving": "Total available production capacity."})
    mostrecentforecast = Column(Float, info={"beschrijving": "Most recently forecasted volume."})
    mostrecentconfidence10 = Column(Float,
                                    info={"beschrijving": "Most recently forecasted volume with a probability of less than 10%
    mostrecentconfidence90 = Column(Float,
                                    info={"beschrijving": "Most recently forecasted volume with a probability of less than 10%
    dayahead11hforecast = Column(Float,
                                 info={"beschrijving": "Day-ahead forecasted volume published at 11AM. "})
    dayahead11hconfidence10 = Column(Float,
                                     info={"beschrijving": "Day-ahead forecasted volume with a probability of less than 10% th
                                     "published at 11AM. "})
    dayahead11hconfidence90 = Column(Float,
                                     info={"beschrijving": "Day-ahead forecasted volume with a probability of less than 10% th
                                     "published at 11AM."})
```

```
    dayaheadforecast = Column(Float,
                              info={"beschrijving": "Day-ahead forecasted volume to be produced."})
    dayaheadconfidence10 = Column(Float,
                                  info={"beschrijving": "Day-ahead forecasted volume with a probability of less than 10% that
    dayaheadconfidence90 = Column(Float, info={"beschrijving": "Forecasted volume with a probability of less than 10% that a h
    weekaheadforecast = Column(Float, info={"beschrijving": "Week-ahead forecasted volume."})
    weekaheadconfidence10 = Column(Float,
                                   info={"beschrijving": "Week-ahead forecasted volume with a probability of less than 10% tha
    weekaheadconfidence90 = Column(Float,
                                   info={"beschrijving": "Week-ahead forecasted volume with a probability of less than 10% tha
    loadfactor = Column(Float, info={"beschrijving": "The percentage ratio between measured power generation and the total mon
    decrementalbidid = Column(String,
                              info={"beschrijving": "Elia has requested the wind park to reduce production below its maximum c
                              "This is defined as the amount of Megawatt for a given quarter-hour (QH).Empty: No decremental b
                              "and the wind park is not required to lower its production during this QH..Note: Elia does not p
                              "decremental bids on request of the parks owners themselves."})
    __table_args__ = (
        UniqueConstraint('datetime', 'region', 'offshoreonshore', 'gridconnectiontype', name='_datetime_region_offshore_connec
        Index('idx_wind_datetime', 'datetime'),
        Index('idx_wind_year', 'year'),
        Index('idx_wind_month', 'month'),
        Index('idx_wind_day', 'day'),
        Index('idx_wind_weekday', 'weekday'),
        Index('idx_wind_hour', 'hour'),
    )

class BelpexPrice(Base):
    """
    SQLAlchemy-model voor het opslaan van Belpex-elektriciteitsprijzen in de database.

    Uniek veld: datetime
    Index op de kolommen year, month, day, weekday en hour
    """
    __tablename__ = "tbl_belpex_prices"
    id = Column(Integer, primary_key=True)
    datetime = Column(DateTime, nullable=False, unique=True)
    year = Column(Integer)
    month = Column(Integer)
    day = Column(Integer)
    weekday = Column(Integer)
    hour = Column(Integer)
```

```python
#    minute = Column(Integer)
    price_eur_per_mwh = Column(Float)
    __table_args__ = (
        Index('idx_belpex_year', 'year'),
        Index('idx_belpex_month', 'month'),
        Index('idx_belpex_day', 'day'),
        Index('idx_belpex_weekday', 'weekday'),
        Index('idx_belpex_hour', 'hour'),
    )
# Creëer tabellen op basis van de klassen die afstammen van de klasse Base
Base.metadata.create_all(engine)

def create_views(engine):
    """
    Maakt SQL-views aan voor wind-, zonne-energie- en Belpex-gegevens.

    Deze functie creëert drie views in de SQLite-database:
    - qry_wind: aggregatie van gemeten en beschikbare windcapaciteit per datetime.
    - qry_solar: aggregatie van gemeten en beschikbare zonnecapaciteit per datetime.
    - qry_belpex: elektriciteitsprijs per datetime uit de Belpex-markt.

    Views worden enkel aangemaakt als ze nog niet bestaan.

    Parameters:
        engine (sqlalchemy.engine.Engine): De SQLAlchemy-engine die met de database verbonden is.
    """
    with engine.connect() as conn:
        conn.execute(text("""
            CREATE VIEW IF NOT EXISTS v_wind AS
            SELECT datetime, year, month, day, weekday, hour, minute,
                   SUM(measured) AS measured_wind,
                   SUM(monitoredcapacity) AS monitored_wind
            FROM tbl_wind_data
            GROUP BY datetime
        """))

        conn.execute(text("""
            CREATE VIEW IF NOT EXISTS v_solar AS
            SELECT datetime,
                   SUM(measured) AS measured_solar,
                   SUM(monitoredcapacity) AS monitored_solar
```

```
            FROM tbl_solar_data
            GROUP BY datetime
        """))

        conn.execute(text("""
            CREATE VIEW IF NOT EXISTS v_belpex AS
            SELECT datetime, year, month, day, hour,
                   price_eur_per_mwh AS price_belpex
            FROM tbl_belpex_prices
            GROUP BY datetime
        """))

create_views(engine)

def parse_record(record):
    """
    Zet een record om naar het juiste datetime-formaat en voegt datumcomponenten toe.

    Parameters:
    - record (dict): Een dictionary met ten minste een 'datetime'-sleutel (ISO-formaat).

    Returns:
    - dict | None: Het verrijkte record of None bij een parsing-fout.
    """
    try:
        dt = datetime.fromisoformat(record["datetime"].replace("Z", "+00:00"))
        record["datetime"] = dt
        record["day"] = dt.day
        record["month"] = dt.month
        record["year"] = dt.year
        record["hour"] = dt.hour
        record["minute"] = dt.minute
        record["weekday"] = dt.isoweekday()
        return record
    except Exception as e:
        return None

def insert_batch(batch, model):
    """
    Voegt een batch records toe aan de database via een `INSERT OR IGNORE` statement.
```

```
    Parameters:
    - batch (list[dict]): Een lijst met dictionaries die overeenkomen met de databasekolommen.
    - model (Base): SQLAlchemy-modelklasse waarin de data wordt opgeslagen.

    Returns:
    - int: Aantal succesvol toegevoegde records.
    """
    try:
        stmt = sqlite_insert(model).prefix_with("OR IGNORE").values(batch)
        result = session.execute(stmt)
        session.commit()
        return result.rowcount
    except Exception as e:
        print(f"⚠️ Fout bij batch-insert: {e} – probeer individuele inserts...")
        inserted = 0
        for record in batch:
            try:
                stmt = sqlite_insert(model).prefix_with("OR IGNORE").values(**record)
                result = session.execute(stmt)
                if result.rowcount:
                    inserted += 1
            except Exception as e:
                print(f"⚠️ Individuele insert mislukt: {e}")
        session.commit()
        return inserted

def process_directory(path, model, batch_size=1000):
    """
    Verwerkt alle JSON-bestanden in submappen (per jaar) van een opgegeven map en slaat ze batchgewijs op in de database.

    Parameters:
    - path (str): Pad naar de hoofdmap met submappen per jaar.
    - model (Base): SQLAlchemy-model waarin de records moeten worden opgeslagen (bv. SolarData of WindData).
    - batch_size (int): Aantal records per batch-insert (default = 1000).
    """

    for year_dir in sorted(os.listdir(path)):
        inserted_records = 0
        total_records = 0

        year_path = os.path.join(path, year_dir)
```

```
        if not os.path.isdir(year_path) or not year_dir.isdigit():
            continue

        all_files = [
            os.path.join(root, f)
            for root, _, files in os.walk(year_path)
            for f in files if f.endswith(".json")
        ]

        batch = []

        for filepath in tqdm(all_files, desc=f"Verwerken van {model.__name__} van het jaar {year_dir}"):
            try:
                with open(filepath, 'r', encoding='utf-8') as f:
                    records = json.load(f)
                    if isinstance(records, dict):
                        records = [records]
            except Exception as e:
                print(f"⚠️ Fout bij laden van bestand {filepath}: {e}")
                continue

            for record in records:
                total_records += 1
                parsed = parse_record(record)
                if parsed is None:
                    continue
                batch.append(parsed)

                if len(batch) >= batch_size:
                    inserted_records += insert_batch(batch, model)
                    batch.clear()

        if batch:
            inserted_records += insert_batch(batch, model)

        if inserted_records > 0:
            print(f"✅ {inserted_records} van {total_records} records van het jaar {year_dir} succesvol toegevoegd aan {model
        else:
            print(f"✅ Jaar {year_dir} van {model.__name__} is bijgewerkt in de database.")

def process_belpex_directory(path, batch_size=1000):
```

```
    """
    Doorloopt een directory met CSV-bestanden met Belpex-data en voegt records toe aan de database.

    Parameters:
    - path (str): Pad naar de directory met .csv-bestanden.
    - batch_size (int): Aantal records per batch-insert (default = 1000).
    """
    all_files = [os.path.join(path, f) for f in os.listdir(path) if f.endswith(".csv")]
    inserted_records = 0
    total_records = 0
    batch = []

    for filepath in tqdm(all_files, desc="Verwerken van Belpex-data"):
        with open(filepath, encoding='iso-8859-1') as csvfile:
            reader = csv.DictReader(csvfile, delimiter=';')
            for row in reader:
                total_records += 1
                try:
                    dt = datetime.strptime(row["Date"], "%d/%m/%Y %H:%M:%S")
                    euro_raw = row["Euro"]
                    # Verwijder alles behalve cijfers, komma, punt en minteken
                    euro_cleaned = re.sub(r"[^\d,.\-]", "", euro_raw)
                    euro = float(euro_cleaned.replace(",", "."))
                    record = {
                        "datetime": dt,
                        "price_eur_per_mwh": euro,
                        "day": dt.day,
                        "month": dt.month,
                        "year": dt.year,
                        "hour": dt.hour,
                        #"minute": dt.minute,
                        "weekday": dt.isoweekday()
                    }
                    batch.append(record)
                except Exception as e:
                    print(f"⚠️ Fout bij record in {filepath}: {e}")

                if len(batch) >= batch_size:
                    inserted_records += insert_batch(batch, BelpexPrice)
                    batch.clear()
```

```
    if batch:
        inserted_records += insert_batch(batch, BelpexPrice)

    if inserted_records > 0:
        print(f"✅ {inserted_records} van {total_records} Belpex-records toegevoegd (duplicaten genegeerd).")
    else:
        print(f"✅ Belpexprijzen zijn bijgewerkt in de database.")

def to_sql(data_type="all"):
    """
    Laadt gegevens vanuit vaste mappenstructuur en schrijft deze naar de SQLite-database,
    afhankelijk van het opgegeven type data.
    Je kunt zelf kiezen welke datasets verwerkt worden:
        - solar: Verwerk zonne-energie (JSON-bestanden)
        - wind: Verwerk windenergie (JSON-bestanden)
        - belpex: Verwerk Belpex-prijzen (CSV-bestanden)
        - all: Verwerk alle types (JSON- en CSV-bestanden)

    Parameters:
    - data_type (str): 'solar', 'wind', 'belpex' of 'all' – bepaalt welke gegevens worden verwerkt.
    """

    try:
        if data_type in ("solar", "all"):
            try:
                process_directory(SOLAR_FORECAST_DIR, SolarData)
            except Exception as e:
                print(f"❌ Fout bij verwerken data zonne-energie: {e}")

        if data_type in ("wind", "all"):
            try:
                process_directory(WIND_FORECAST_DIR, WindData)
            except Exception as e:
                print(f"❌ Fout bij verwerken data windenergie: {e}")

        if data_type in ("belpex", "all"):
            try:
                process_belpex_directory(BELPEX_DIR)
            except Exception as e:
                print(f"❌ Fout bij verwerken data Belpexprijzen: {e}")
```

```
    except KeyboardInterrupt:
        print("\n🛑 Script onderbroken door gebruiker.")
    except Exception as e:
        print(f"❌ Onverwachte fout: {e}")
    finally:
        session.close()
        engine.dispose()
        print("\n🔒 Databaseverbinding correct afgesloten.\n")
```

## Inhoud uit script: `src/utils/__init__.py`

In [ ]:

## Inhoud uit script: `src/utils/constants_inspector.py`

In [ ]:
```
"""
constants_inspector.py

Deze module biedt een hulpfunctie om alle constante configuratievariabelen (in hoofdletters)
uit een opgegeven Python-module te inspecteren. De functie `list_module_constants` detecteert
alle publieke constante attributen (d.w.z. variabelen met hoofdletters die niet beginnen met een
underscore), drukt ze af en retourneert ze als een dictionary.

Typisch gebruik is het overzichtelijk weergeven van instellingen of configuratieparameters die
in aparte modules zijn gedefinieerd.

Voorbeeld:
    import settings
    from constants_inspector import list_module_constants

    constants = list_module_constants(settings, sort=True)
"""

def list_module_constants(module, sort=False):
    """
    Drukt alle configuratievariabelen (in hoofdletters) van een gegeven module af
```

```
    en retourneert ze ook als dictionary.

    Parameters:
        module (module): De module waarvan de constanten moeten worden weergegeven.
        sort (bool): Of de constante variabelen gesorteerd moeten worden afgedrukt (standaard True).

    Returns:
        dict: Een dictionary met de namen en waarden van de constante variabelen.
    """
    consts = {
        name: value
        for name, value in module.__dict__.items()
        if name.isupper() and not name.startswith("_")
    }

    items = sorted(consts.items()) if sort else consts.items()

    for name, value in items:
        print(f"{name:20} = {value}")

    return consts
```

## Inhoud uit script: `src/utils/decorators.py`

In [ ]:

```
"""
decorators.py

Verzameling van algemene decorators voor hergebruik in verschillende projecten.

Deze module bevat momenteel:

- retry_on_failure: een decorator die functies automatisch opnieuw probeert uit te voeren
  bij tijdelijke fouten, met configureerbare parameters voor aantal pogingen, wachttijd,
  exponentiële backoff en toegestane uitzonderingen.

In de toekomst kunnen hier meer decorators toegevoegd worden.
"""
```

```python
import functools
import time

def retry_on_failure(tries=3, delay=2, backoff=1, allowed_exceptions=(Exception,)):
    """
    Decorator om een functie meerdere keren opnieuw uit te voeren wanneer er een fout optreedt.

    Deze decorator is nuttig bij tijdelijke fouten, zoals netwerkproblemen of onstabiele API-responses.
    Als de gedecoreerde functie een uitzondering genereert die voorkomt in `allowed_exceptions`,
    zal ze automatisch opnieuw uitgevoerd worden tot het maximum aantal `tries` is bereikt.
    Tussen elke poging wacht de functie `delay` seconden. Na elke fout wordt de wachttijd vermenigvuldigd
    met `backoff` (exponentiële backoff).

    Parameters:
    - tries (int): Het maximaal aantal pogingen voor de functie wordt opgegeven. Standaard: 3.
    - delay (float): De initiële wachttijd (in seconden) tussen pogingen. Standaard: 2.
    - backoff (float): De vermenigvuldigingsfactor voor de wachttijd bij elke fout. Standaard: 1 (geen toename).
                       Een waarde >1 verhoogt de wachttijd exponentieel (bijv. 2 voor verdubbeling).
    - allowed_exceptions (tuple): Een tuple van uitzonderingen waarvoor een retry toegestaan is.
                                  Standaard: (Exception,), wat alle standaardfouten omvat.

    Intern maakt de wrapper gebruik van lokale kopieën van de parameters `_tries` en `_delay`
    om te voorkomen dat de oorspronkelijke decoratorwaarden (die gedeeld worden door alle oproepen)
    overschreven of beïnvloed worden tijdens het uitvoeren van retries.

    Gebruik:
    @retry_on_failure(tries=5, delay=1, backoff=2, allowed_exceptions=(ConnectionError,))
    def fetch_data():
        ...

    """
    def decorator(func):
        # Zorgt ervoor dat de metadata (naam, docstring, enz.) van de originele functie
        # behouden blijft in de gegenereerde wrapperfunctie.
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # Lokale kopieën maken om te vermijden dat de originele decorator-argumenten
            # gewijzigd worden tijdens herhaalde pogingen
            _tries, _delay = tries, delay
            while _tries > 1:
                try:
```

```
                    return func(*args, **kwargs)
                except allowed_exceptions as e:
                    _tries -= 1
                    print(f"⚠️ Fout '{e}' in {func.__name__}(). Nog {_tries} pogingen over... Wacht {_delay:.1f}s.")
                    # Wacht voor het opgegeven aantal seconden
                    time.sleep(_delay)
                    _delay *= backoff
            # Laatste poging buiten de while-loop: als deze ook faalt, wordt de uitzondering doorgegeven
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

## Inhoud uit script: `src/utils/dual_logger.py`

In [ ]:

```
"""
dual_logger.py

Bevat de klasse DualLogger, die zowel sys.stdout als sys.stderr tijdelijk omleidt zodat alle uitvoer (print-statements en fout
"""
import sys

class DualLogger:
    """
    Vervangt sys.stdout en sys.stderr zodat alle output (zowel print als foutmeldingen)
    tegelijkertijd naar de console én naar een logbestand geschreven wordt.

    Deze klasse werkt zowel als:
    - Contextmanager: gebruik `with DualLogger(path):` om automatisch stdout/stderr te vervangen
                      en het logbestand na afloop veilig te sluiten.
    - Losse instantie: roep `logger = DualLogger(path)` aan, en vergeet `logger.close()` niet.

    Parameters:
    - logfile_path (str): Volledig pad naar het logbestand (zal geopend worden in append-modus).

    Gebruik als contextmanager:
    ---------------------------
    with DualLogger("pad/naar/log.txt"):
        print("Dit gaat naar console én naar logbestand.")
```

```
    raise Exception("Fouten ook!")

Gebruik als losse instantie:
---------------------------
logger = DualLogger("pad/naar/log.txt")
sys.stdout = sys.stderr = logger
print("Loggen zonder contextmanager.")
logger.close()  # Belangrijk!
"""

def __init__(self, logfile_path):
    # Sla pad op en open het logbestand (append-modus, UTF-8)
    self.logfile_path = str(logfile_path)
    self.log = open(self.logfile_path, "a", encoding="utf-8", errors="replace")

    # Bewaar originele standaard streams om later te kunnen herstellen
    self.original_stdout = sys.stdout
    self.original_stderr = sys.stderr

def write(self, message):
    """
    Wordt automatisch aangeroepen door print() of foutmeldingen.
    Schrijft het bericht zowel naar het scherm als naar het logbestand.
    """
    self.original_stdout.write(message)   # Toon op het scherm
    self.log.write(message)               # Schrijf naar het logbestand

def flush(self):
    """
    Wordt automatisch aangeroepen om de buffer te legen.
    Noodzakelijk voor realtime logging of bij gebruik van print(..., flush=True).
    """
    self.original_stdout.flush()
    self.log.flush()

def close(self):
    # Herstel standaard streams
    sys.stdout = self.original_stdout
    sys.stderr = self.original_stderr

    # Sluit expliciet het logbestand bij manueel gebruik
```

```
        self.log.close()

    def __enter__(self):
        # Contextmanager start: vervang stdout en stderr door deze logger
        sys.stdout = sys.stderr = self
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        # Herstel oorspronkelijke streams
        sys.stdout = self.original_stdout
        sys.stderr = self.original_stderr
        # Sluit het logbestand
        self.log.close()

    def __del__(self):
        # Herstel standaard streams indien nog actief
        try:
            if sys.stdout is self:
                sys.stdout = self.original_stdout
            if sys.stderr is self:
                sys.stderr = self.original_stderr
        except Exception:
            pass  # Tijdens interpreter shutdown kunnen globals verdwijnen

        # Probeer logbestand te sluiten indien nog open
        try:
            if hasattr(self, "log") and not self.log.closed:
                self.log.close()
        except Exception:
            pass  # Stilletjes falen indien iets misloopt bij garbage collection
```

## Inhoud uit script: `src/utils/package_tools.py`

In [ ]:

```
"""
package_tools.py

Hulpfuncties voor het beheren van Python-packages binnen een project.
```

```
Deze module bevat functies om te controleren of een package aanwezig is, indien nodig automatisch te installeren of te upgrade

Momenteel bevat deze module:

- update_or_install_if_missing(package_name, min_version=None):
    Installeert of upgrade een package indien het ontbreekt of de versie
    niet aan de minimumvereiste voldoet, en importeert het daarna.
"""

import importlib
import importlib.util
import subprocess
import sys

def update_or_install_if_missing(package_name, min_version=None):
    """
    Zorgt ervoor dat een Python-package geïnstalleerd is, en indien gewenst,
    dat het voldoet aan een minimale versie.

    - Installeert het package als het nog niet aanwezig is.
    - Voert een upgrade uit als de aanwezige versie te laag is of niet numeriek vergelijkbaar is.
    - Herlaadt het package na installatie of upgrade, zodat het meteen bruikbaar is.

    Parameters:
    - package_name (str): Naam van het package zoals op PyPI (bv. 'requests').
    - min_version (str, optional): Minimale vereiste versie (bv. '2.25.0'). Indien None, wordt geen versiecontrole uitgevoerd.

    Returns:
    - module: Het geïmporteerde package-object (na installatie of upgrade indien nodig).
    """

    def parse_version(v, width=None):
        """
        Zet een versie-string (bv. '2.10.3' of '2.25.0.1') om naar een lijst van gehele getallen.
        Indien 'width' is opgegeven, wordt de lijst opgevuld tot die lengte met nullen.
        Als 'width' None is, wordt de ruwe lijst teruggegeven.

        Parameters:
        - v (str): Versie als string (bv. '2.10.3').
        - width (int or None): Minimale lengte van de resulterende lijst (aangevuld met nullen indien nodig).
```

```
        Returns:
        - list[int]: Lijst van gehele getallen.

        Raises:
        - ValueError: Als een deel van de versie geen geheel getal is (bv. '2.10a1').
        """
        parts = v.split('.')
        int_parts = []
        for p in parts:
            if not p.isdigit():
                raise ValueError(f"Niet-numeriek versieonderdeel: '{p}'")
            int_parts.append(int(p))
        if width:
            return int_parts + [0] * (width - len(int_parts))
        return int_parts

    def is_version_at_least(current, minimum):
        """
        Vergelijkt twee versie-strings op basis van numerieke onderdelen.
        De kortere lijst wordt opgevuld met nullen zodat beide even lang zijn.

        Parameters:
        - current (str): Huidige geïnstalleerde versie.
        - minimum (str): Vereiste minimumversie.

        Returns:
        - bool: True als current >= minimum, anders False.
                Geeft ook False terug als de versie niet numeriek vergeleken kan worden.
        """
        try:
            current_parts = parse_version(current)
            minimum_parts = parse_version(minimum)
            max_len = max(len(current_parts), len(minimum_parts))
            current_parts += [0] * (max_len - len(current_parts))
            minimum_parts += [0] * (max_len - len(minimum_parts))
            return current_parts >= minimum_parts
        except ValueError as e:
            print(f"⚠️  Versie '{current}' is niet numeriek vergelijkbaar ({e}) → installeren/upgrade vereist")
            return False

    needs_reload = False  # vlag om te bepalen of we herladen na installatie/upgrade
```

```python
    # Controleer of het package al aanwezig is
    spec = importlib.util.find_spec(package_name)
    if spec is None:
        # Package is nog niet geïnstalleerd → installeer het
        print(f"📦 Module '{package_name}' niet gevonden. Bezig met installeren...")
        pip_target = f"{package_name}>={min_version}" if min_version else package_name
        subprocess.check_call([sys.executable, "-m", "pip", "install", pip_target])
        needs_reload = True
    else:
        # Package is reeds aanwezig → importeer het
        module = importlib.import_module(package_name)

        if min_version:
            # Controleer de huidige versie (default = '0.0.0' als niet beschikbaar)
            current_version = getattr(module, "__version__", "0.0.0")

            # Vergelijk versies; upgrade indien nodig
            if not is_version_at_least(current_version, min_version):
                print(f"🔁 Upgrade nodig: {package_name} ({current_version} < {min_version})")
                subprocess.check_call([sys.executable, "-m", "pip", "install", f"{package_name}>={min_version}"])

                # Verwijder het package en submodules uit sys.modules om herladen mogelijk te maken
                to_delete = [mod for mod in sys.modules if mod == package_name or mod.startswith(package_name + ".")]
                for mod in to_delete:
                    del sys.modules[mod]

                needs_reload = True

    # Herlaad het package indien nodig
    module = importlib.import_module(package_name)

    # Toon geïnstalleerde of geüpgradede versie indien herladen
    if needs_reload:
        version = getattr(module, "__version__", "onbekend")
        print(f"✅ Module '{package_name}' geïnstalleerd (versie {version}).")

    return module
```

## Inhoud uit script: `src/utils/safe_requests.py`

In [ ]:

```
"""
safe_requests.py

Bevat de functie `safe_requests_get` - een uitgebreide en veilige wrapper rond `requests.get()` met ingebouwde retry-logica.

Deze module biedt een eenvoudige manier om HTTP GET-verzoeken uit te voeren met foutafhandeling en herhaalde pogingen bij misl
Ideaal voor scripts die robuust moeten omgaan met tijdelijke netwerk- of serverproblemen (bijv. bij het ophalen van data van e

Voorbeeldgebruik:
    from safe_requests import safe_requests_get

    response = safe_requests_get("https://api.example.com/data", tries=5, delay=1)
"""

import time
import requests

def safe_requests_get(url, params=None, headers=None, tries=3, delay=2, timeout=10):
    """
    Uitgebreide en veilige versie van requests.get() met ingebouwde retry-logica.

    Deze functie probeert een HTTP GET-verzoek uit te voeren naar de opgegeven URL.
    Als het verzoek faalt door een netwerkfout of een HTTP-fout (zoals 5xx of 4xx-status),
    wordt het verzoek automatisch opnieuw geprobeerd tot een maximum van `tries` keer.
    Na elke mislukte poging wordt `delay` seconden gewacht alvorens opnieuw te proberen.

    Parameters:
    - url (str): De URL waarnaar het GET-verzoek wordt verzonden.
    - params (dict, optional): Optionele query parameters toe te voegen aan het verzoek.
    - headers (dict, optional): Optionele headers om mee te sturen met het verzoek.
    - tries (int): Aantal pogingen bij fouten. Standaard is 3.
    - delay (int or float): Wachtijd (in seconden) tussen pogingen. Standaard is 2.
    - timeout (int or float): Maximum wachttijd voor een antwoord van de server. Standaard is 10 seconden.

    Retourneert:
    - response (requests.Response): Het response-object als het verzoek succesvol was.
```

```
    Raises:
    - requests.exceptions.RequestException: Als alle pogingen mislukken of er een andere fout optreedt.

    Gebruik:
    response = safe_requests_get("https://api.example.com/data", tries=5, delay=1)

    """
    # Lokale kopie van tries maken zodat de oorspronkelijke waarde behouden blijft bij hergebruik
    _tries = tries
    while _tries > 1:
        try:
            response = requests.get(url, params=params, headers=headers, timeout=timeout)
            # Roep een uitzondering op bij een HTTP-statuscode die een fout aangeeft (4xx of 5xx)
            response.raise_for_status()
            return response
        except (requests.exceptions.RequestException, requests.exceptions.HTTPError) as e:
            print(f"⚠️ Request fout: {e}. Nog {_tries-1} pogingen... Wacht {delay}s.")
            # Wacht voor het opgegeven aantal seconden
            time.sleep(delay)
            _tries -= 1
    # Laatste poging buiten de loop: als deze faalt, wordt de uitzondering niet meer opgevangen
    response = requests.get(url, params=params, headers=headers, timeout=timeout)
    response.raise_for_status()
    return response
```

## Inhoud uit script: `src/utils/sqlalchemy_model_utils.py`

In [ ]:

```
"""
sqlalchemy_model_utils.py

Hulpfuncties voor inspectie van SQLAlchemy-modellen.

Bevat tools om automatisch een overzicht te genereren van alle modellen die afstammen van een
gegeven basisklasse, inclusief hun kolomnamen en optionele beschrijvingen.

Geschikt voor debugging, documentatie, of het dynamisch genereren van gebruikersinterfaces
op basis van SQLAlchemy-modeldefinities.
```

```
Voorbeeldgebruik:
    from sqlalchemy_model_utils import alle_modellen_en_kolommen
    print(alle_modellen_en_kolommen(Base))
"""

def alle_modellen_en_kolommen(base_class):
    """
    Genereert een overzicht van alle SQLAlchemy-modellen die afstammen van een gegeven basisklasse,
    samen met hun kolomnamen en optionele beschrijvingen.

    Voor elke subklasse van `base_class` (typisch gemaakt met declarative_base()) wordt gecontroleerd
    of deze daadwerkelijk gekoppeld is aan een database-tabel (via __tablename__ en __table__).
    Vervolgens worden alle kolommen weergegeven, inclusief eventuele beschrijvingen die zijn toegevoegd
    via het `info`-attribuut van SQLAlchemy.

    Daarnaast worden ook constraints (zoals primaire sleutels, foreign keys en unieke restricties)
    en indexen per tabel weergegeven, voor een volledig beeld van de tabelstructuur.

    Parameters:
    - base_class : declarative_base()
        De basisklasse waarvan alle SQLAlchemy-modellen afstammen.
        Voorbeeld: Base = declarative_base()

    Returns:
    - str
        Een overzichtelijke string die de naam van elk model toont, gevolgd door:
            - de lijst van kolommen (met optionele beschrijving)
            - de constraints op de tabel
            - de indexen op de tabel
    """
    uitvoer = []

    # Doorloop alle subklassen (modellen) die van base_class zijn afgeleid
    for cls in base_class.__subclasses__():
        # Controleer of het een geldig SQLAlchemy-model is met een gekoppelde tabel
        if hasattr(cls, '__tablename__') and hasattr(cls, '__table__'):
            uitvoer.append(f"Model: {cls.__name__} (tabel: {cls.__tablename__})")

            # Doorloop alle kolommen van de tabel
            for kolom in cls.__table__.columns:
```

```
                kolom_naam = kolom.name  # naam van de kolom
                # Haal optionele beschrijving op (indien aanwezig)
                beschrijving = kolom.info.get("beschrijving", None)

                # Voeg de kolom toe aan de uitvoer, met of zonder beschrijving
                if beschrijving:
                    uitvoer.append(f"  - {kolom_naam}: {beschrijving}")
                else:
                    uitvoer.append(f"  - {kolom_naam}")

            # Voeg constraint-informatie toe
            if cls.__table__.constraints:
                uitvoer.append("\n  Constraints:")
                for constraint in cls.__table__.constraints:
                    uitvoer.append(f"    - {type(constraint).__name__}: {str(constraint)}")

            # Voeg index-informatie toe
            if cls.__table__.indexes:
                uitvoer.append("\n  Indexen:")
                for index in cls.__table__.indexes:
                    uitvoer.append(f"    - {index.name}: columns={[col.name for col in index.columns]}")



            uitvoer.append("")

    return "\n".join(uitvoer)
```