Databaser i webapplikasjoner

Av Frode Børli, 2024.

Innledning

Dette dokumentet er pensum i WebApp modul 3: Databaser. Dokumentet gir informasjon om bruk av databaser i webapplikasjoner, og sikkerhetsaspekter som er viktige.

Innhold

Introduksjon til databaser og Python

Når vi utvikler moderne applikasjoner, spiller databaser en nøkkelrolle i lagring og henting av data på en strukturert og effektiv måte. Python tilbyr gode biblioteker og moduler for å samhandle med databaser. For å forstå hvordan dette samspillet fungerer må vi se på grunnleggende konsepter som er involvert i tilkoblingen mellom en Python-applikasjon og en database.

Autentisering og sikkerhet

Når applikasjonen bruker en slik modul for å kobler seg til database-serveren, så autentiserer applikasjonen seg med serveren. Dette bruker databaseserveren for å håndtere hvilke databaser applikasjonen får jobbe med, og hvilke handlinger denne applikasjonen kan gjøre.

Ikke gjenbruke tilkoblinger

De tre viktigste operasjonene man forholder seg til når man jobber med databaser i en webapplikasjon er:

1. Koble seg til databasen.
2. Utføre databasespørringer.
3. Koble seg fra databasen.

Generelt bør man ikke gjenbruke en tilkobling til databasen på tvers av forskjellige brukere og forespørsler til webapplikasjonen. Et av problemene med å gjenbruke en tilkobling (for eksempel ved å lagre den i en global variabel), er at man vanskelig kan kontrollere om en annen forespørsel blir håndtert samtidig over den samme tilkoblingen. Et annet problem er at man risikerer å gjøre databaseoperasjoner inne i en transaksjon som er startet av en annen forespørsel til webapplikasjonen.

Interaksjon med databaser

Opprette forbindelse til databasen

Før en Python-applikasjon kan lese fra eller skrive til en database, må den først opprette en forbindelse til database-serveren. Dette involverer vanligvis spesifisering av detaljer som serverens adresse (f.eks., hostname eller IP), portnummer, databasenavn, samt autentiseringsopplysninger (brukernavn og passord). Denne prosessen sikrer at kun autoriserte applikasjoner og brukere kan aksessere og manipulere databasen.

PostgreSQL

```
import psycopg2  # For PostgreSQL
# Opprette forbindelse til database
conn = psycopg2.connect(
  dbname='min_database',
  user='brukernavn',
  password='passord',
  host='database.server.com')
```

MySQL

```
import mysql.connector

# Opprette forbindelse til MySQL-database
conn = mysql.connector.connect(
  host='database.server.com',
  user='brukernavn',
  password='passord',
  database='min_database'
)
```

Sqlite3

Sqlite3 er en databaseform som *ikke benytter en server.* Dette betyr at applikasjonen jobber mot en fil i filsystemet. Denne varianten er svært enkel å ta i bruk, men egner seg ikke veldig godt for webapplikasjoner som har mange brukere. Den er imidlertid utmerket i applikasjoner som installeres på brukerens PC og dermed ikke deles av mange brukere samtidig. Den kan også brukes i utviklingen av en applikasjon.

```
import sqlite3

conn = sqlite3.connect('eksempel.db')
```

Åpning og lukking av databasetilkoblinger

Når forbindelsen er etablert, kan applikasjonen utføre SQL-spørringer for å manipulere data (CRUD-operasjoner: Create, Read, Update, Delete). Etter at nødvendige operasjoner er utført, er det viktig å lukke databasetilkoblingen for å frigjøre ressurser og opprettholde systemets ytelse og sikkerhet.

```
# Utføre en spørring
cur = conn.cursor()
cur.execute("SELECT * FROM min_tabell")
# Hente resultater
rows = cur.fetchall()
for row in rows:
  print(row)
# Lukke tilkoblingen
cur.close()
conn.close()
```

- **Oppretter en cursor (**cur = conn.cursor()**):**
    - En cursor (markør) er et objekt som lar deg utføre SQL-spørringer mot databasen og hente ut resultater.
    - conn representerer den åpne databasetilkoblingen du har etablert tidligere i koden.
- **Utfører en SQL SELECT-spørring (**cur.execute("SELECT * FROM min_tabell")**):**
    - SQL-spørringen "SELECT * FROM min_tabell" instruerer databasen til å hente alle kolonner (*) fra tabellen som heter "min_tabell".
    - cur.execute()-funksjonen utfører den spesifiserte spørringen mot databasen.
- **Henter resultater (**rows = cur.fetchall()**):**
    - cur.fetchall()-funksjonen samler alle de returnerte radene fra spørringen og lagrer dem i variabelen rows.
- **Prosesserer resultater (**for row in rows: print(row)**):**
    - Løkken itererer (gjentas) gjennom hver rad i rows-listen:
        - row representerer en enkelt rad i resultatet.
        - print(row) skriver ut den nåværende raden til konsollen. Antagelig vil du gjøre noe mer nyttig med dataene fra raden, som for eksempel å vise dem på en nettside.
- **Lukker markør og tilkobling (**cur.close()**,** conn.close()**):**
    - Disse linjene frigjør ressurser som er knyttet til både markøren (cur) og databasetilkoblingen (conn). Dette er viktig for å forhindre ressurslekkasjer og sikre optimal ytelse.

Viktige punkter

- **Datatype:** Datalinjen (row) som du henter ut fra databasen vil sannsynligvis være i formatet tuple eller liste, avhengig av det spesifikke databasebiblioteket du bruker. Hvert element i tupelen/listen vil representere en enkelt verdi fra en kolonne i tabellen.

- **Databasebibliotek:** Jeg brukte her et generelt eksempel uten å spesifisere om det gjelder PostgreSQL, MySQL eller Sqlite3. Husk å importe riktig bibliotek og justere tilkoblingsdetaljene i overensstemmelse med databasen du bruker.
- **Sikkerhet:** Alltid vær varsom med SQL-injeksjon, særlig når du konstruerer SQL-spørringer fra brukerinput. Les opp på "parameterized queries" eller "prepared statements" for å sikre at du lager trygge databseinteraksjoner.

Skrive egne klasser i Python

Skrive egne klasser i Python

I dette kapitlet skal vi se på hvordan vi kan lage egne klasser i Python. Dette er et nyttig verktøy for å organisere kode og gjenbruke funksjonalitet. Vi skal også se på hvordan vi kan importere moduler og bruke klasser fra andre moduler.

Moduler og import

I Python kan vi organisere kode i moduler. En modul er en fil som inneholder Python-kode. Vi kan importere moduler til andre Python-filer for å bruke funksjonaliteten de inneholder.

For å importere et modul bruker vi import-setningen. For eksempel:

```
import modulnavn
```

Dette importerer alle funksjoner og variabler fra modulnavn til den nåværende filen. Vi kan også velge å importere bare bestemte elementer fra et modul:

```
from modulnavn import funksjon, variabel
```

Dette importerer bare funksjon og variabel fra modulnavn til den nåværende filen.

Klasser og instanser

En klasse er en mal for å lage objekter. Et objekt er en instans av en klasse. En klasse kan inneholde attributter (variabler) og metoder (funksjoner).

For å definere en klasse bruker vi class-setningen:

```
class Klassenavn:
    attributt1 = verdi1
    attributt2 = verdi2

    def metode1(self, parameter1, parameter2):
        ...

    def metode2(self):
        ...
```

Dette definerer en klasse med navnet Klassenavn. Klassen har to attributter, attributt1 og attributt2, og to metoder, metode1 og metode2.

For å lage en instans av en klasse bruker vi ()-operatoren:

```
objekt = Klassenavn()
```

Dette lager en ny instans av Klassenavn og lagrer den i variabelen objekt.

Vi kan nå bruke attributtene og metodene til objektet:

```
objekt.attributt1 = "Ny verdi"
objekt.metode1(parameter1, parameter2)
```

Eksempel

La oss se på et eksempel. Vi skal lage en klasse for å representere en person:

```
class Person:
    navn = ""
    alder = 0

    def __init__(self, navn, alder):
        self.navn = navn
        self.alder = alder

    def hils(self):
        print(f"Hei, jeg heter {self.navn} og er {self.alder} år gammel.")


person1 = Person("Ola", 25)
person2 = Person("Kari", 30)

person1.hils()
person2.hils()
```

Dette kodesnippetet definerer en klasse Person med to attributter, navn og alder, og en metode, hils. Vi lager deretter to instanser av Person, person1 og person2, og setter attributtene for hver instans. Til slutt kaller vi hils-metoden på begge instansene.

Fordeler med å bruke klasser

Det er mange fordeler med å bruke klasser i Python:

- **Organisering av kode:** Klasser kan brukes til å organisere kode i moduler og gjenbruke funksjonalitet.
- **Encapsulation:** Klasser kan brukes til å skjule detaljer om implementasjonen av en funksjonalitet.
- **Polymorphism:** Klasser kan brukes til å definere forskjellige typer objekter som kan interagere med hverandre på en enhetlig måte.

Eksempelklasse for databasehåndtering i Flask

Denne eksempelklassen demonstrerer hvordan man kan lage en klasse i Python for å håndtere kobling til og frakobling fra en database, og utføre spørringer med mindre repetering av kode. Klassen er designet for å brukes i en Flask-applikasjon.

```
from flask import Flask
from sqlalchemy import create_engine

class Database:
    def __init__(self, app):
        self.app = app
        self.engine = None

    def connect(self):
        """Kobler til databasen."""
        database_uri = self.app.config["DATABASE_URI"]
        self.engine = create_engine(database_uri)

    def disconnect(self):
        """Frakobler fra databasen."""
        if self.engine is not None:
            self.engine.dispose()

    def execute_query(self, query):
        """Utfører en spørring mot databasen."""
        with self.engine.connect() as connection:
            return connection.execute(query)

app = Flask(__name__)

# Konfigurer database-URI
app.config["DATABASE_URI"] = "sqlite:///database.db"

# Opprett en database-instans
database = Database(app)

@app.route("/")
def index():
    # Koble til databasen
    database.connect()

    # Utfør en spørring
    results = database.execute_query("SELECT * FROM users")

    # Behandle resultater
    for row in results:
        print(row)

    # Frakoble fra databasen
    database.disconnect()

    return "<h1>Hjemmeside</h1>"

if __name__ == "__main__":
    app.run(debug=True)
```

Forklaring av koden

- Klassen Database har tre funksjoner:
    - connect: Kobler til databasen ved å bruke URI-en spesifisert i Flask-applikasjonens konfigurasjon.
    - disconnect: Frakobler fra databasen.
    - execute_query: Utfører en SQL-spørring mot databasen og returnerer resultatet.
- I index-funksjonen:
    - Vi kobler til databasen ved å kalle database.connect.
    - Vi utfører en spørring ved å kalle database.execute_query.
    - Vi behandler resultatene fra spørringen.
    - Vi frakobler fra databasen ved å kalle database.disconnect.

Fordeler med å bruke en slik klasse

- **Mindre repetering av kode:** Vi unngår å gjenta koden for å koble til og frakoble fra databasen for hver spørring.
- **Enkel å bruke:** Klassen tilbyr et enkelt grensesnitt for å utføre spørringer mot databasen.
- **Testetbar:** Klassen kan enkelt testes for å sikre at den fungerer korrekt.

Utvidelse av klassen

Denne klassen kan enkelt utvides for å støtte mer avanserte funksjoner, for eksempel:

- Utføring av INSERT-, UPDATE- og DELETE-spørringer.
- Håndtering av transaksjoner.
- Caching av database resultater.

Denne eksempelklassen er et godt utgangspunkt for å håndtere databasetilkoblinger og spørringer i Flask-applikasjoner.