CS-A1150 Tietokannat

13.5.2019

Oppimistavoitteet: tämän luennon jälkeen

- Tiedät lisää siitä, miten SQL-käskyjä voidaan liittää toisella ohjelmointikielellä kirjoitetettuun ohjelmaan.
- 29.5. käsiteltiin ORM-työkaluja ja Python-kirjaston käyttöä. Tällä luennolla
  - Käydään läpi lisää esimerkkejä tietokannan käytöstä Python-ohjelmassa sqlite3-kirjaston avulla.
  - Katsotaan esimerkkejä tietokannan käytöstä Scala-ohjelmassa anorm-kirjaston avulla.
  - Esitellään hyvin lyhyesti SQL-käskyjen kirjoittamista ohjelmaan suoraan (embedded SQL).

Kertausta: SQL-käskyjen liittäminen Python-ohjelmaan

- Python-kirjaston sqlite3 avulla Python-ohjelmassa voidaan luoda ja käyttää SQLite-tietokanta.
- Python-ohjelman aluksi on määriteltävä, että ohjelma käyttää tätä moduulia:

```
import sqlite3
```

- Muodostetaan yhteys haluttuun tietokantaan:

```
conn = sqlite3.connect("webstoredatabase.db")
```

- Luodaan kursori, jonka avulla voidaan tehdä tietokantaan kyselyitä ja muutoksia.

```
cursor = conn.cursor()
```

Kertausta: SQL-käskyjen suorittaminen

- Kursorin avulla voidaan suorittaa SQL-käskyjä metodin `execute` avulla.

```
cursor.execute("""CREATE TABLE Products (
                  number TEXT PRIMARY KEY,
                  prodName TEXT,
                  description TEXT,
                  price REAL,
                  manufID TEXT)""")

cursor.execute("""INSERT INTO Products
VALUES('P-48221', 'Teema 26', 'plate', 16.00,'F542')""")
```

- Kun halutut päivitykset on tehty, on yhteydelle suoritettava `commit`-operaatio.

```
conn.commit()
```

- Kun tietokantaa ei enää tarvita, yhteys siihen voidaan sulkea

```
conn.close()
```

Kertausta: kyselyiden tekeminen ja tulosten läpikäynti

- Kyselyitä voidaan tehdä kursorin `execute`-metodilla:

```
cursor.execute("SELECT * FROM Products WHERE price < 99")
```

- Tulosrivit voidaan hakea joko yksi kerrallaan `fetchone`-metodilla tai sitten kaikki kerralla `fetchall`-metodilla, esim.

```
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Tulostetaan vain osa attribuuteista:

```
rows = cursor.fetchall()
for row in rows:
    print(row[1], row[3])
```

Kertausta: käyttäjän syötteen käyttäminen kyselyissä

- SQL-käskyihin voi lisätä käyttäjältä saatua tietoa muuttujien avulla.
- Muuttujien arvoja ei saa kuitenkaan liittää suoraan kyselyssä käytettävään merkkijonoon, koska se mahdollistaisi *SQL-injektion*: vihamielinen käyttäjä antaa syötteen sijaan SQL-käskyjä, jotka tekevät ei-toivottuja asioita.
- SQL-injektion riskin välttämiseksi SQL-käskyt välitetään SQLitelle parametrisoituna, esimerkiksi:

```
name1 = input("Give the name of the product: ")
cursor.execute("SELECT * FROM Products WHERE name =?",
               (name1,))
```

Esimerkkejä 1

- Pyydetään käyttäjältä tuotteen kuvaus ja maksimihinta. Haetaan ehdon täyttävien tuotteiden tiedot:

```
import sqlite3

conn = sqlite3.connect("webstoredatabase.db")
cursor = conn.cursor()
desc1 = input("Give the description: ")
price1 = float(input("Give the maximum price: "))
query = """SELECT * FROM Products
           WHERE description =? AND  price <=?"""
cursor.execute(query, (desc1, price1))
rows = cursor.fetchall()
for row in rows:
    print(row)
conn.close()
```

Esimerkkejä 2

- Lisätään Products-tauluun tuote, jonka tiedot saadaan käyttäjältä

```
import sqlite3
conn = sqlite3.connect("webstoredatabase.db")
cursor = conn.cursor()
print("Give information of the product to be inserted")
number1 = input("number: ")
name1 = input("name: ")
desc1 = input("description: ")
price1 = float(input("price: "))
manufid1 = input("manufacturer id: ")
insertion = "INSERT INTO Products VALUES(?, ?, ?, ?, ?)"
cursor.execute(insertion, (number1, name1, desc1,
                           price1, manufid1))
conn.commit()
conn.close()
```

Huomattavaa

- Lisää `sqlite3`-kirjaston tarjoamista mahdollisuuksistaon kerrottu tutoriaalissa `http://zetcode.com/db/sqlitepythontutorial/` ja Pythonin omassa dokumentaatiossa `http://docs.python.org/3/library/sqlite3.html`.
- Esimerkeistä oli jätetty virheenkäsittely pois. Kirjastofunktioiden suorituksissa tulleita ongelmia voi käsitellä `sqlite3.Error`-tyyppisinä poikkeuksina.

Esimerkkejä 3: virheenkäsittely esimerkkiin 2

```
import sqlite3

conn = None
try:
    conn = sqlite3.connect("webstoredatabase.db")
    cursor = conn.cursor()
    print("Enter information of the product to be inserted")
    number1 = input("number: ")
    name1 = input("name: ")
    desc1 = input("description: ")
    try:
        price1 = float(input("price: "))
    except ValueError:
        print("Price is not a decimal number. 0.0 used instead")
        price1 = 0.0
    manufid1 = input("manufacturer id: ")
    insertion = "INSERT INTO Products VALUES(?, ?, ?, ?, ?)"
    # continues
```

Esimerkkejä 3 jatkuu

```
    cursor.execute(insertion,
                   (number1, name1, desc1, price1, manufid1))
    conn.commit()
except sqlite3.Error:
    print("Database error.")
if conn != None:
    conn.close()
```

- Esimerkki (kuten myös esimerkit 4 ja 5) on tehty käyttämällä vain sellaisia Python-kielen ominaisuuksia, jotka on opetettu kurssilla CS-A1111 Ohjelmoinnin peruskurssi Y1. Sen vuoksi esimerkeissä ei ole käytetty esimerkiksi `with`-rakennetta eikä `finally`-osaa toisin kuin usein vastaavissa tilanteissa käytetään.

Esimerkkejä 4

- Haetaan halutun valmistajan kaikkien tuotteiden numero, nimi ja hinta. Jos taulussa `Products` tuotteen hinta on NULL, tulostetaan hinnan paikalle Not available.
- Python-ohjelma käsittelee NULL arvoja arvona None.

```
import sqlite3

conn = None
try:
    conn = sqlite3.connect("webstoredatabase.db")
    cursor = conn.cursor()
    manufacturer = input("Enter the id of the manufacturer: ")
    query = """SELECT number, prodName, price FROM Products
               WHERE manufID = ?"""
    cursor.execute(query, (manufacturer,))
```

Esimerkkejä 4 jatkuu

```
        rows = cursor.fetchall()
        print("All products manufacturer by", manufacturer)
        for row in rows:
            if row[2] == None:
                priceinfo = "Not available"
            else:
                priceinfo = row[2]
            print(row[0], row[1], priceinfo)
except sqlite3.Error:
        print("Database error")
if conn != None:
        conn.close()
```

Useamman rivin lisääminen yhdellä käskyllä

- Metodin `executemany` avulla voi suorittaa parametrina annetun käskyn useita kertoja.
- Esimerkiksi

```
insertion = "INSERT INTO Products VALUES(?, ?, ?, ?, ?)"
cursor.executemany(insertion, new_products)
```

Suorittaa käskyn `INSERT`-käskyn kerran jokaista listassa `new_products` olevaa alkiota kohti.

Esimerkkejä 5: usean rivin lisääminen

```
import sqlite3

conn = None
try:
    conn = sqlite3.connect("webstoredatabase.db")
    cursor = conn.cursor()
    new_products = [('T-33442', 'NX 300', 'camera', 399.0, 'S123'),
                    ('T-33455', 'Cyber', 'camera', 463.0, 'L711'),
                    ('T-29783', '40 Smart', 'TV', 519.50, 'L711')]
    insertion = "INSERT INTO Products VALUES(?, ?, ?, ?, ?)"
    cursor.executemany(insertion, new_products)
    conn.commit()
except sqlite3.Error:
    print("Database error.")
if conn != None:
    conn.close()
```

SQL-tietokannat Scala-ohjelmissa

- Sivulla `http://manuel.bernhardt.io/2014/02/04/a-quick-tour-of-relational-database-access-with-scala/` on esitetty erilaisia vaihtoehtoja käyttää SQL-käskyjä Scala-ohjelmassa.
- Seuraavaksi esitellään lyhyesti Anorm-nimistä työkalua, joka tarjoaa yhden kirjastopohjaisen ratkaisun.
- Esitetyt esimerkit on muokattu Anormin dokumentaation pohjalta eikä niitä ole testattu.

Anormin käyttö: aloitus

- Lataa (esimerkiksi Eclipseen) seuraavat kirjastot ja lisää ne projektiin: `play`, `jdbc`, `anorm`.
- Ohjelmatiedoston alussa ota käyttöön tarvittavat kirjastot:

```
import anorm._
import play.api.db.DB
```

- Määritellään oletustietokanta. Tässä esimerkissä se on SQLite-tietokanta, joka on tallennettu tiedostoon nimeltä `/path/to/db-file` (alku on polku siihen hakemistoon, joka sisältää ko. tiedoston).

```
db.default.driver=org.sqlite.JDBC
db.default.url="jdbc:sqlite:/path/to/db-file"
```

Anorm: yhteys tietokantaan ja SQL-käskyjen suorittaminen

- Avataan yhteys tietokantaan ja suoritetaan ensimmäinen kysely:

```
DB.withConnection { implicit c => {
  val result: Boolean = SQL("Select 1").execute()
  // muita kyselyita ym}
}
```

  Muiden saman tietokantayhteyden aikana suoritettavien SQL-käskyjen suorittamiseksi tarvittava koodi kirjoitetaan kohtaan
  `// muita kyselyita ym`

- Seuraavaksi esimerkkejä käskyistä, joita tähän kohtaan voi kirjoittaa.

Anorm: esimerkki lisäyksestä

```
val id: Option[Long] = SQL(
  """
    insert into BelongsTo(orderNo, productNo, count)
    values ({number1}, {number2}, {count1})
  """
  ).on('number1 -> "469666",
       'number2 -> "S-65221",
       'count1 -> 2
  ).executeInsert()
```

Anorm: esimerkki kyselyn määrittelemistä

```
val selectProducts = SQL(
                       """
                         select number, name
                         from Prodcuts
                         where description = {desc1};
                       """
                     ).on("desc1" -> "cellphone")
```

- Tässä ei vielä suoriteta kyselyä, vaan määritellään se ja tallennetaan se muuttujaan `selectProducts`.
- Jälleen `{desc1}` on paikanvaraaja, joka korvataan `on`-kohdassa määritellyllä arvolla.

Anorm: esimerkkikyselyn suorittaminen ja tulosten tallentaminen listaan

- Suoritetaan edellisellä kalvolla määritelty kysely, muutetaan saadut rivit pareiksi (tuotteen numero, tuotteen nimi), ja kootaan näistä pareista lista.
- Koottuun listaan viitataan muuttujalla `productinfo`

```
val productinfo = selectProducts().map(row =>
  row[String]("number") -> row[String]("name")
).toList
```

SQL-käskyjen kirjoittaminen ohjelmaan suoraan

- C-ohjelmiin voidaan kirjoittaa suoraan SQL-käskyjä ympäröitynä sopivilla tunnuksilla, jotka C-kääntäjän esiprosessori käsittelee. Tekniikkaa kutsutaan upotetuksi SQL:ksi (embedded SQL).
- C-ohjelmaan kirjoitetut SQL-käskyt aloitetaan aina ilmauksella EXEC SQL
- Lisätietoa oppikirjassa, seuraavalla kalvolla joitakin keskeisiä käsitteitä.

Upotettun SQL:n käyttöön liittyviä keskeisiä käsitteitä

- SQL-käskyn ja isäntäkielisen (C-kielellä kirjoitun) ohjelman avulla voidaan välittää tietoa *yhteisten muuttujien* (shared variables) avulla, joita voi käyttää sekä C-kielisessä osassa että SQL-käskyissä. Kun näitä muuttujia käytetään SQL-käskyissä, niiden nimen eteen kirjoitetaan kaksoispiste.
- Muuttuja SQLSTATE on viiden merkin mittainen merkkijono, jonka arvo kertoo kunkin SQL-käskyn suorituksen jälkeen käskyn onnistumisesta.
- Jos suoritetaan kyselyjä, joiden tuloksena voi olla useita rivejä, voidaan tulosrivejä käydä läpi *kursorin* avulla (eri kuin sqlite3-kirjaston cursor).

ORMit, SQL:n yhdistäminen Python- tai Scala-ohjelmiin ja upotettu SQL tentissä

- Tentissä riittää, että tietää periaatteessa, mistä näissä asioissa on kysymys.
- Tentissä ei pyydetä kirjoittamaan SQL-käskyjä sisältäviä Python- tai Scala-ohjelmia tai upotettua SQL:ää.