

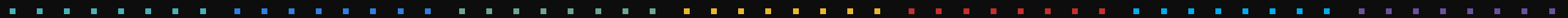
WMNBE-2203 | BACK-END DEVELOPMENT

Flask #5

Data Persistence - SQLite 3



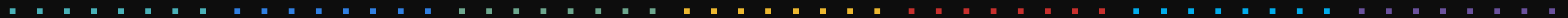
ΠΕΡΙΕΧΟΜΕΝΑ



Περιεχόμενα

- Connection
- Data Retrieval
 - **fetchall**, **fetchone**
 - Query params
 - **row_factory**
- Connection management
- Transactions
 - Context manager
- **rowcount**, **lastrowid**

SQLITE3



sqlite3

SQLite

Η **SQLite** είναι στην ουσία μια **C** βιβλιοθήκη, η οποία παρέχει μια απλή **disk-based** βάση δεδομένων, χωρίς την ανάγκη εγκατάστασης ενός *Database Server*.

Η πρόσβαση στη βάση γίνεται με μια *nonstandard* **SQL** διάλεκτο.

sqlite3

SQLite

Χρησιμοποιείται συχνά για την αποθήκευση ρυθμίσεων/δεδομένων εφαρμογών (όπως π.χ. εφαρμογών **Android**).

Μπορεί άνετα να υποστηρίξει ένα μεσαίου μεγέθους site (π.χ. με ~100K hits/day) και μπορεί να λειτουργήσει ως "σκαλοπάτι" για μια μεταγενέστερη λύση, π.χ. **MariaDB**, **PostgreSQL**, κ.λπ.

sqlite3

Connection

Η *Standard Library* της **Python** περιέχει ήδη module για τη σύνδεση με βάση **SQLite**.

```
import sqlite3  
  
con = sqlite3.connect('path/to/database.db')
```

Στη θέση του *file path* μπορεί να χρησιμοποιηθεί και το **:memory:**, για μια βάση δεδομένων αποκλειστικά στη μνήμη.

sqlite3

Data Retrieval

Για την ανάκτηση δεδομένων, γίνεται χρήση του **Cursor** object και της μεθόδου **execute**.

Το αποτέλεσμα της **execute** μπορεί να χρησιμοποιηθεί ως *iterator*, π.χ. για τη χρήση σε μία εντολή **for**.

```
cur = con.cursor()

for row in cur.execute('SELECT id, title FROM products'):
    print(row)
```


sqlite3

fetchall

Μία καλύτερη προσέγγιση είναι η χρήση της μεθόδου **fetchall**.

```
products = cur.execute(
    '''
    SELECT id, title, description, price
    FROM products
    '''
).fetchall()

for product in products:
    print(product)
```

sqlite3

fetchall

Το αποτέλεσμα της μεθόδου είναι μία λίστα με της γραμμές του πίνακα σε **tuples**.

```
...
print(products)          # [(1, ...), (2, ...), (3, ...)]
print(products[0])       # (1, 'Some product', 'Some description', 12.34)
print(products[0][0])    # 1
print(products[0][1])    # 'Some product'
```

sqlite3

fetchone

Εναλλακτικά μπορεί να χρησιμοποιηθεί η μέθοδος **fetchone**, όταν περιμένουμε το πολύ ένα αποτέλεσμα.

```
products = cur.execute(
    '''
    SELECT id, title, description, price
    FROM products
    WHERE id = ?
    ''',
    (id,) # query params have to be a tuple
).fetchone()

if product is not None:
    print(product)
```

sqlite3

Query params

Είτε πρόκειται για τη μέθοδο **fetchall**, είτε για τη **fetchone**, πολύ συχνά υπάρχει ανάγκη για τη χρήση παραμέτρων στο **query**.

Στο προηγούμενο παράδειγμα, χρησιμοποιούνται *positional parameters*, όπου κάθε **?** αντικαθιστάται με τη σειρά από τα στοιχεία ενός **tuple**.

Το **tuple** αυτό παρέχεται ως δεύτερο όρισμα της μεθόδου **execute**.

sqlite3

SQL Injection

Σε καμία περίπτωση δε χρησιμοποιούμε κάποιο μηχανισμό για *string interpolation / concatenation* της γλώσσας, αλλά χρησιμοποιούμε αποκλειστικά τις παραμέτρους του **sqlite3** module.

Διαφορετικά η εφαρμογή μας κινδυνεύει από **SQL Injection**.

Το ίδιο ισχύει γενικά, για οποιοδήποτε συνδυασμό γλώσσας / *db access framework*.

sqlite3

Named params

Οι *positional* παράμετροι μπορεί να οδηγήσουν σε πολλά προβλήματα.

Μία καλύτερη προσέγγιση είναι η χρήση των *named params*.

```
# || is for string concatenation

products = cur.execute(
    '''
    SELECT id, title, description, price
    FROM products
    WHERE title LIKE :q || '%' AND category_id = :id
    ''',
    {'q': search, 'id': category_id}
).fetchall()

for product in products:
    print(product)
```

sqlite3

Named params

Σε αυτή την περίπτωση ορίζουμε τις παραμέτρους με ένα όνομα, αυτές διακρίνονται από το χαρακτήρα **:** στα αριστερά τους.

Οι τιμές για τις παραμέτρους προκύπτουν από το δεύτερο όρισμα της **execute** που σε αυτή την περίπτωση είναι ένα **dict**.

Τα **keys** του λεξικού πρέπει να συμφωνούν με τα ονόματα που δώσαμε στις παραμέτρους.

sqlite3

Rows

Τα αποτελέσματα που επιστρέφουν από τη βάση είναι στην ουσία μία λίστα από πλειάδες. Οι πλειάδες περιέχουν τα στοιχεία κάθε εγγραφής που ανακτήθηκε από τη βάση.

Αυτό από μόνο του δεν είναι κακό, αλλά μας δυσκολεύει στο να προσπελάσουμε με ευκολία τα δεδομένα. Θα πρέπει να αναφερόμαστε στις στήλες κάθε εγγραφής με έναν αριθμητικό δείκτη.

Παράδειγμα

sqlite3

```
@app.get('/courses')
def show_course_list():
    cur = get_con().cursor()
    courses = cur.execute(
        'SELECT * FROM courses'
    ).fetchall()

    return render_template('courses/list.html',
                           courses=courses)
```

```
{# list.html #}

{% for course in courses %}
    {{ course[0] }} - {{ course[1] }}
{% endfor %}
```

sqlite3

row_factory

Μιας και η προσπέλαση μέσω αριθμητικών δεικτών δεν είναι ιδανική, μπορεί να χρησιμοποιηθεί το **sqlite3.Row**, για την αναπαράσταση των εγγραφών της βάσης.

Ο τύπος αυτός επιτρέπει την αναφορά στις εγγραφές / στήλες ως *key-value pairs*.

Αυτό ορίζεται ως εξής:

```
con = sqlite3.connection(DATABASE_FILE_PATH)
con.row_factory = sqlite3.Row
```

Παράδειγμα

sqlite3

```
@app.get('/product/<int:id>')
def product_details():
    cur = get_con().cursor()
    product = cur.execute(
        'SELECT * FROM products WHERE id = :id', {'id': id}
    ).fetchone()

    return render_template('products/details.html',
                           product=product)
```

```
{# details.html #}

<h1>{{ product['title'] }}</h1>
<p>{{ product['description'] }}</p>
<p>€{{ product['price'] }}</p>
```

sqlite3

Connection management

Για την ευκολότερη διαχείριση των συνδέσεων στη βάση, μπορεί να χρησιμοποιηθεί το **g** object μαζί με κάποια utility **function**.

Ιδανικά θέλουμε ένα **connection** ανά **request**.

```
from pathlib import Path

DATABASE_PATH = Path(__file__).parent / 'data/e-shop.db'

def get_con():
    if 'con' not in g:          # hasattr(g, 'con')
        con = sqlite3.connect(DATABASE_PATH)
        con.row_factory = sqlite3.Row
        g.con = con            # setattr(g, 'con', con)
    return g.con
```

sqlite3

Connection closing

Η καλή πρακτική ορίζει πως πρέπει κάθε σύνδεση στη βάση να κλείνει σωστά και στο σωστό χρόνο.

Μια καλή ιδέα είναι να αυτοματοποιηθεί η διαδικασία αυτή, στο τέλος κάθε **request**. Το **teardown_appcontext** θεωρείται ως καταλληλότερο για τη χρήση αυτή μιας και θα "πυροδοτηθεί" ακόμη και στην περίπτωση κάποιου **exception**.

```
@app.teardown_appcontext
def close_connection(error):
    '''
    Close connection on appcontext teardown
    This will fire whether there was an exception or not
    '''
    if con := g.pop('con', None):
        con.close()
```

sqlite3

Persistence / **commit**

Προφανώς, εκτός από **DQL** ερωτήματα (**SELECT**), μπορούν να εκτελεστούν και "εντολές" **DML** (**SELECT...INTO, INSERT, UPDATE, DELETE**).

Σε αυτή την περίπτωση, πρέπει να κληθεί η μέθοδος **commit**, ώστε να "παραμείνουν" οι αλλαγές στη βάση (*persist changes*).

```
con = get_con()
con.execute(
    'INSERT INTO book (isbn, title) VALUES (:isbn, :title)',
    {
        'isbn': request.form.get('isbn'),
        'title': request.form.get('title')
    }
)
con.commit()
```

sqlite3

Transactions / rollback

Στα συστήματα διαχείρισης βάσης δεδομένων υπάρχει η έννοια του *transaction*.

Με ένα *transaction* μπορούμε να εξασφαλίσουμε πως μία ομάδα από εντολές θα λειτουργήσουν ως ένα σύνολο. Είτε θα εκτελεστούν όλες επιτυχώς ή καμία.

Αυτή η λειτουργικότητα είναι προφανώς εξαιρετικά χρήσιμη σε περιπτώσεις που είναι πολύ κρίσιμο να μη διακοπεί κάποια σειρά ενεργειών στη βάση πρόωρα.

sqlite3

Transactions / rollback

...

```
con = get_con()
try:
    amount = request.form.get('amount')
    con.execute(
        'UPDATE account SET balance = balance - :a WHERE id = :id',
        {'a': amount, 'id': request.form.get('from_account_id')}
    )
    con.execute(
        'UPDATE account SET balance = balance + :a WHERE id = :id',
        {'a': amount, 'id': request.form.get('to_account_id')}
    )
    con.commit()
except sql.Error as err:
    con.rollback()
    app.logger.error(err)

# Not the actual way a bank transaction is done...
```


sqlite3

Context manager

Η διαχείριση ενός *transaction* μπορεί να γίνει και με τη βοήθεια ενός *context manager* της **Python**.

Αν δημιουργηθεί ένα *connection* στη βάση με τη βοήθεια του *keyword with*, τότε θα εκτελεστεί αυτόματα ένα **commit** στο τέλος του *with block*.

Αν εμφανιστεί κάποιο σφάλμα, τότε ο *context manager* θα φροντίσει για το **rollback**.

Προσοχή, την περίπτωση του σφάλματος, κατά την επικοινωνία στη βάση, δεν την διαχειρίζεται ο *context manager*, άρα παραμένει η ανάγκη για τη χρήση ενός **try...except**.

sqlite3

Context manager

...

```
try:
    with get_con() as con: # won't close the connection
        con.cursor().execute(
            '''
            INSERT INTO user (username, password, fullname)
            VALUES (:username, :password, :fullname)
            ''',
            {'username': u, 'password': p, 'fullname': f}
        )
        # Context manager will deal with the commit
    flash('Registration successful', category='success')
except Exception as err:
    # Context manager will deal with the rollback
    app.logger.error(err)
    flash('Something went wrong...', category='error')
finally:
    con.close()
```

sqlite3

rowcount

Το **Cursor** object μπορεί να μας δώσει μερικές χρήσιμες πληροφορίες. Για παράδειγμα πόσες γραμμές επηρεάστηκαν από το προηγούμενο "ερώτημα".

```
con = get_con()
cur = con.cursor()
cur.execute(
    'UPDATE "product" SET "price" = 12.34'
)    # forgot the WHERE clause

if cursor.rowcount == 1:
    con.commit()
else:
    con.rollback()
```

sqlite3

lastrowid

Η ποιο είναι το **rowid** (*autoincrement value*) του στοιχείου που μόλις εισήχθε σε πίνακα της βάσης.

```
with get_con() as con:
    cur = con.cursor()
    cur.execute(
        '''
        INSERT INTO "product" ("title", "desc", "price")
        VALUES (:title, :desc, :price)
        ''',
        {'title': title, 'desc': desc, 'price': price}
    )
    print(cur.lastrowid)
```

Χρήσιμα links



Datatypes In SQLite Version 3

<https://www.sqlite.org/datatype3.html>

Using SQLite 3 with Flask — Flask Documentation (2.1.x)

<https://flask.palletsprojects.com/en/2.1.x/patterns/sqlite3/>



sqlite3 — DB-API 2.0 interface for SQLite databases — Python 3.9.5 documentation

<https://docs.python.org/3/library/sqlite3.html>



Flask - SQLite - Tutorialspoint

https://www.tutorialspoint.com/flask/flask_sqlite.htm


Define and Access the Database — Flask Documentation (2.1.x)

<https://flask.palletsprojects.com/en/2.1.x/tutorial/databases/>

Extra info

 SQLite Tutorial - Tutorialspoint
<https://www.tutorialspoint.com/sqlite/index.htm>

 SQLite Python
<https://www.sqlitetutorial.net/sqlite-python/>

 Squeezing Performance from SQLite: Indexes? Indexes! | by Jason Feinstein | Medium
<https://medium.com/@JasonWyatt/squeezing-perfor...>

 Getting the most out of SQLite3 with Python
<https://remusao.github.io/posts/few-tips-sqlite-perf.html>

THANK YOU!