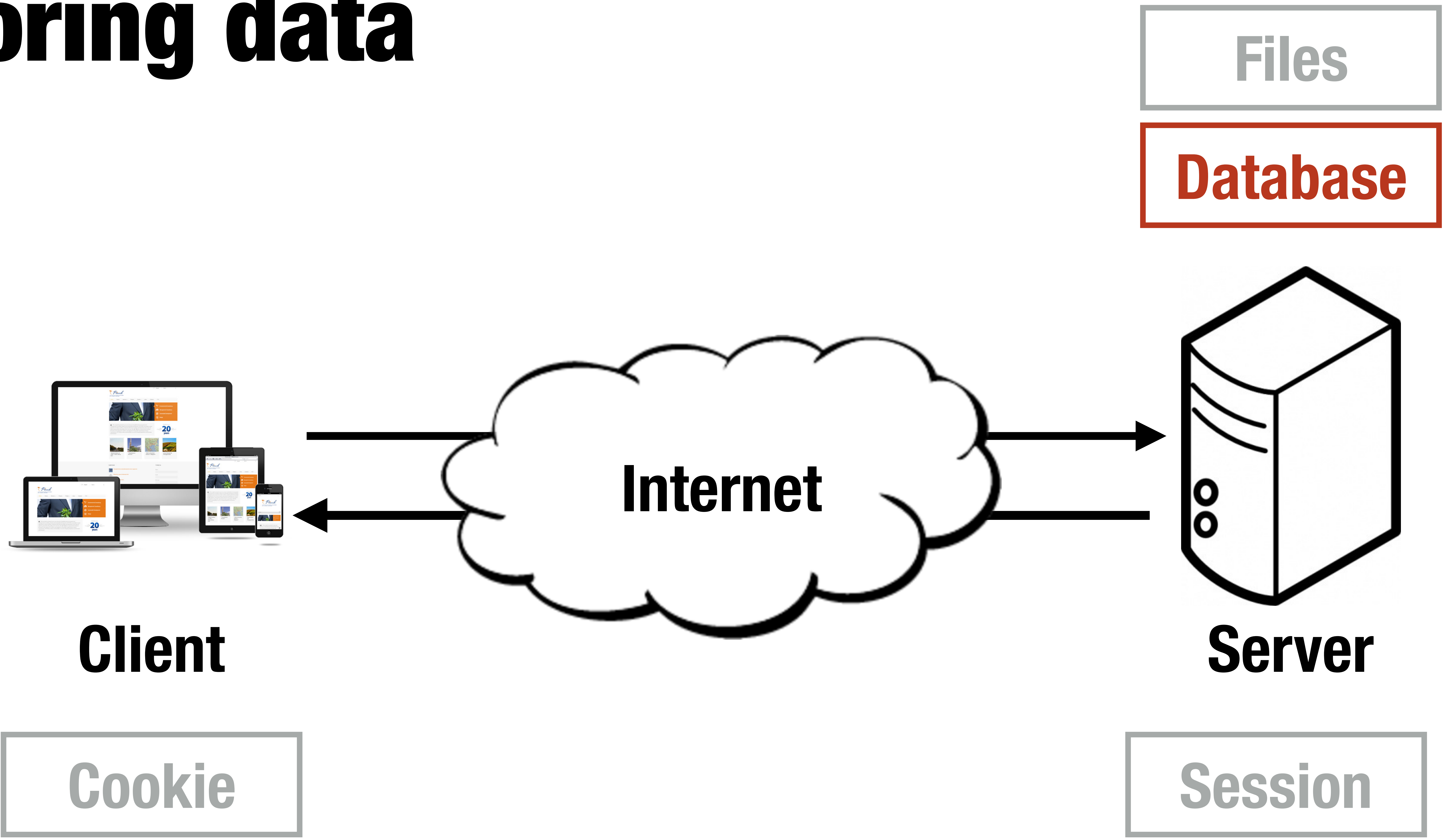
Web Programming
Server-side programming III.

Leander Jehl | University of Stavanger

Server-side programming

- Part I. handling requests
- Part II. templating
- Part III. MySQL
- Part IV. cookies and sessions

Storing data



Using MySQL from Python

Connectors

- Low level connectors vs. Object-relational mapping (ORM)
- Many packages for low level connection
  - Most of them are compliant with the Python Database API Specification (PEP 249) https://www.python.org/dev/peps/pep-0249/
- We will be using **MySQL Connector/Python**
  - "Official" connector
  - https://dev.mysql.com/doc/connector-python/en/
  - Install using pip

    ```
    python -m pip install mysql-connector-python
    ```

  - Part of Anaconda, but needs to be installed

    ```
    conda install mysql-connector-python
    ```

Python Database API Specification

- Two main objects
  - Connection
  - Cursor
- Connection methods
  - **`cursor()`** returns a new Cursor
  - **`close()`** closes connection to DB
  - **`commit()`** commits any pending transactions
  - **`rollback()`** rolls back to the start of any pending transaction (optional)

Connecting to a DB

```
import mysql.connector


conn = mysql.connector.connect(user='root', password=‘foobarfoo',
                               host='127.0.0.1', database='dat310')

# do some stuff

conn.close()
```

- The **`connect()`** constructor creates a connection to the MySQL server and returns a **`MySQLConnection`** object

Error Handling

examples/python/mysql/mysql1.py

```
from mysql.connector import errorcode


try:
    conn = mysql.connector.connect(…)
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Invalid username/password.")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exist.")
    else:
        print(err)
else:
    # do some stuff
   conn.close()
```

Python Database API Specification

- Cursor methods/attributes
  - **`execute()`** executes a database operation or query
  - **`rowcount`** read-only attribute, number of rows that the last execute command produced (SELECT) or affected (UPDATE, INSERT, DELETE)
  - **`close()`** closes the cursor
  - **`fetchone()`** fetches the next row of a query result set
  - **`fetchmany()`** fetches the next set of rows of a query result
  - **`fetchall()`** fetches all (remaining) rows of a query result
  - **`arraysize`** read/write attribute, specifying the number of rows to fetch at a time with **`fetchmany()`** (default is 1)

Creating a Table

examples/python/mysql/mysql1.py

```python
cur = conn.cursor()
try:
    sql = ("CREATE TABLE postcodes ("
           "postcode VARCHAR(4), "
           "location VARCHAR(20), "
           "PRIMARY KEY(postcode))")
    cur.execute(sql)
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
        print("Error: Table already exists.")
    else:
        print("Error: {}".format(err.msg))
else:
    print("Table created.")
finally:
    cur.close()
```

Dropping a Table

examples/python/mysql/mysql1.py

```python
cur = conn.cursor()
try:
    sql = "DROP TABLE postcodes"
    cur.execute(sql)
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_BAD_TABLE_ERROR:
        print("Error: Table does not exist.")
    else:
        print("Error: {}".format(err.msg))
else:
    print("Table dropped.")
finally:
    cur.close()
```

Inserting Data

examples/python/mysql/mysql1.py

```
sql = "INSERT INTO postcodes (postcode, location) VALUES (%s, %s)"
try:
    cur.execute(sql, (k, v))  # data is provided as a tuple
    conn.commit()  # commit after each row
except mysql.connector.Error as err:
    print("Error: {}".format(err.msg))
```

- Data is provided as a tuple (list of values)
- DELETE and UPDATE work the same way
- You must **commit** the data after these statements

Inserting Data (2)

```
add_salary = ("INSERT INTO salaries "
              "(emp_no, salary, from_date, to_date) "
              "VALUES (%(emp_no)s, %(salary)s, %(from_date)s, %(to_date)s)")


# Insert salary information
data_salary = {
  'emp_no': emp_no,
  'salary': 50000,
  'from_date': tomorrow,
  'to_date': to_date,
}

cursor.execute(add_salary, data_salary)
```

- It is also possible to provide data in a dict

Querying Data

examples/python/mysql/mysql1.py

```
cur = conn.cursor()
try:
    sql = ("SELECT postcode, location FROM postcodes "
           "WHERE postcode BETWEEN %s AND %s")
    cur.execute(sql, ("4000", "5000"))
    for (postcode, location) in cur:
        print("{}: {}".format(postcode, location))
except mysql.connector.Error as err:
    print("Error: {}".format(err.msg))
finally:
    cur.close()
```

Object-Relational Mapping

- For Object-Relational Mapping (ORM), see SQLAlchemy
    - https://www.sqlalchemy.org/
    - Flask extension: http://flask.pocoo.org/docs/0.12/patterns/sqlalchemy/

```
users = Table('users', metadata,
    Column('user_id', Integer, primary_key=True),
    Column('name', String(40)),
    Column('age', Integer),
    Column('password', String),
)
users.create()

i = users.insert()
i.execute(name='Mary', age=30, password='secret')

s = users.select(users.c.age < 40)
rs = s.execute()
```

Using MySQL from Flask

Flask Contexts

- Flask provides two contexts
- **`request`** variable is associated with the current request

```
from flask import request
```

- **`g`** is associated with the "global" application context

```
from flask import g
```

  - typically used to cache resources that need to be created on a per-request case, e.g., DB connections
  - resource allocation: **`get_X()`** creates resource X if it does not exist yet, otherwise returns the same resource
  - resource deallocation: **`teardown_X()`** is a tear down handler

Example

examples/python/flask/5_mysql/app.py

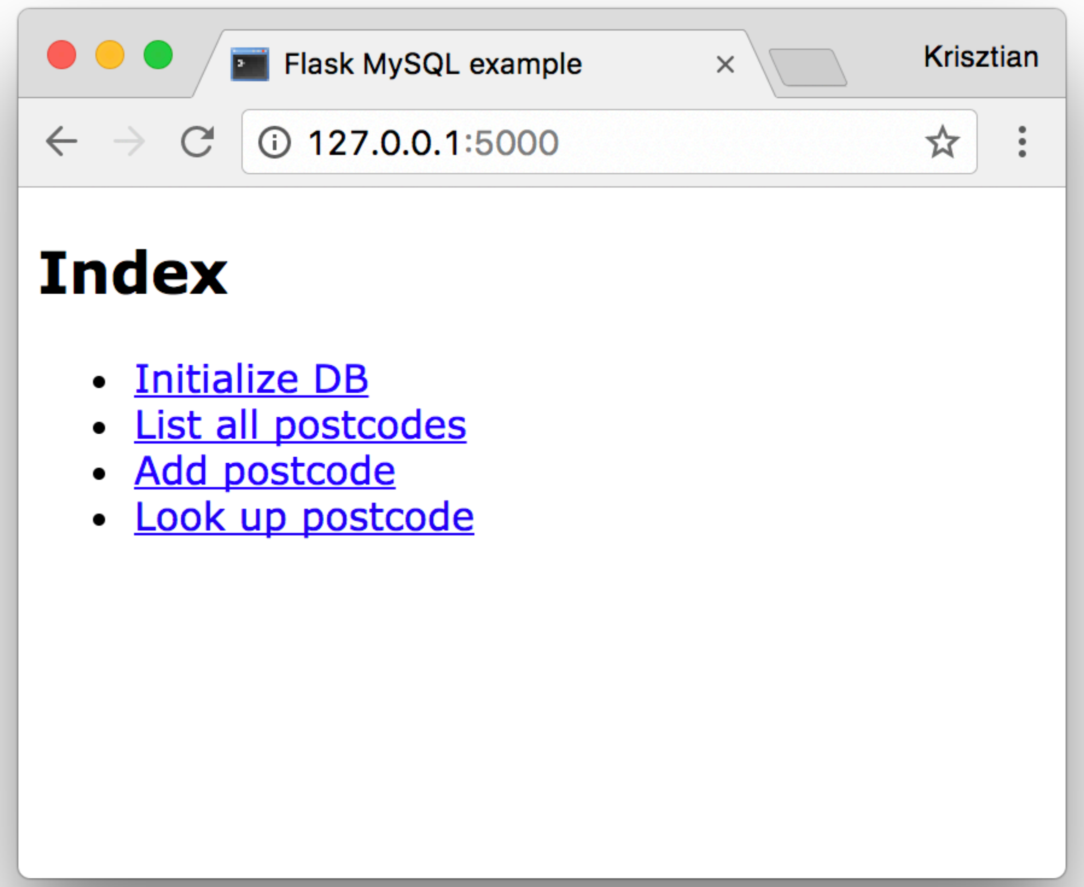
```python
def get_db():
    if not hasattr(g, "_database"):
        g._database = mysql.connector.connect(…)
   return g._database


@app.teardown_appcontext
def teardown_db(error):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()


@app.route("/listall")
def list_all():
    """List all postcodes."""
    db = get_db()
    cur = db.cursor()
```

The first time **`get_db()`** is called the connection will be established

Example

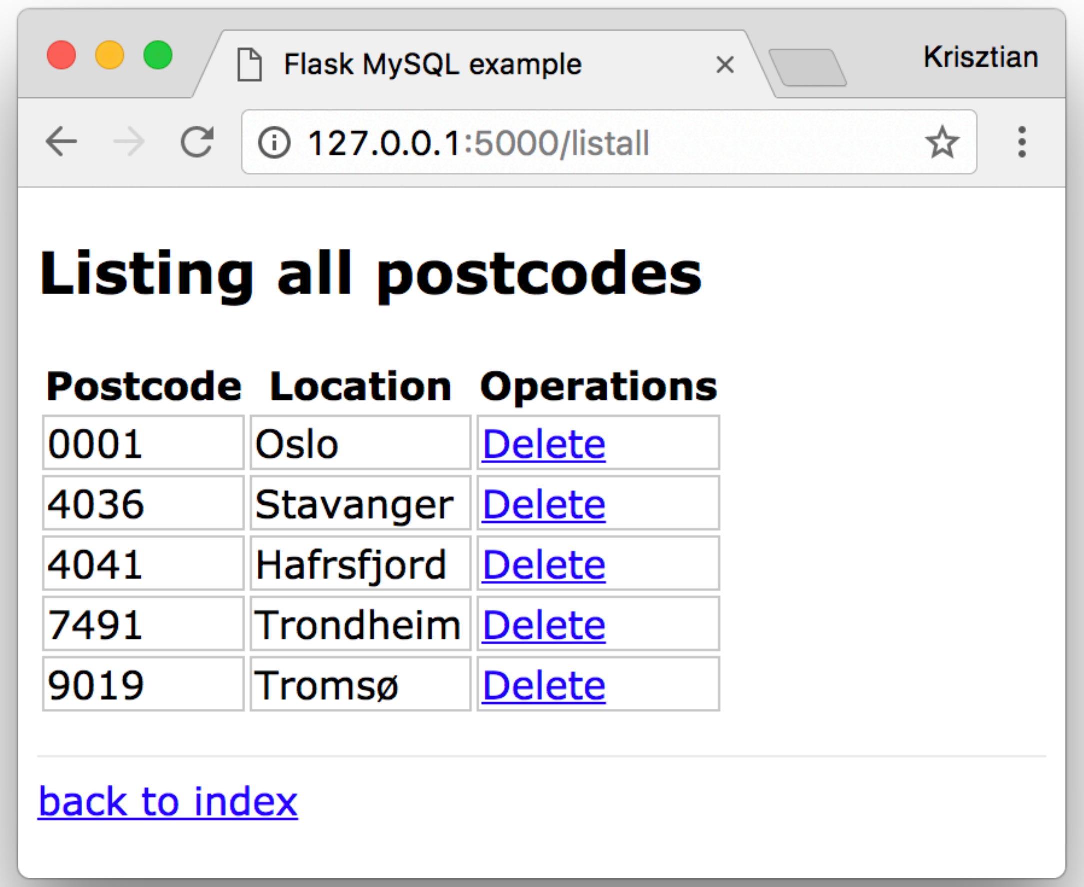
examples/python/flask/5_mysql/app.py

- Contains examples of CREATE TABLE, INSERT, SELECT (single/multiple records), DELETE
- Uses flashing for success messages



Index

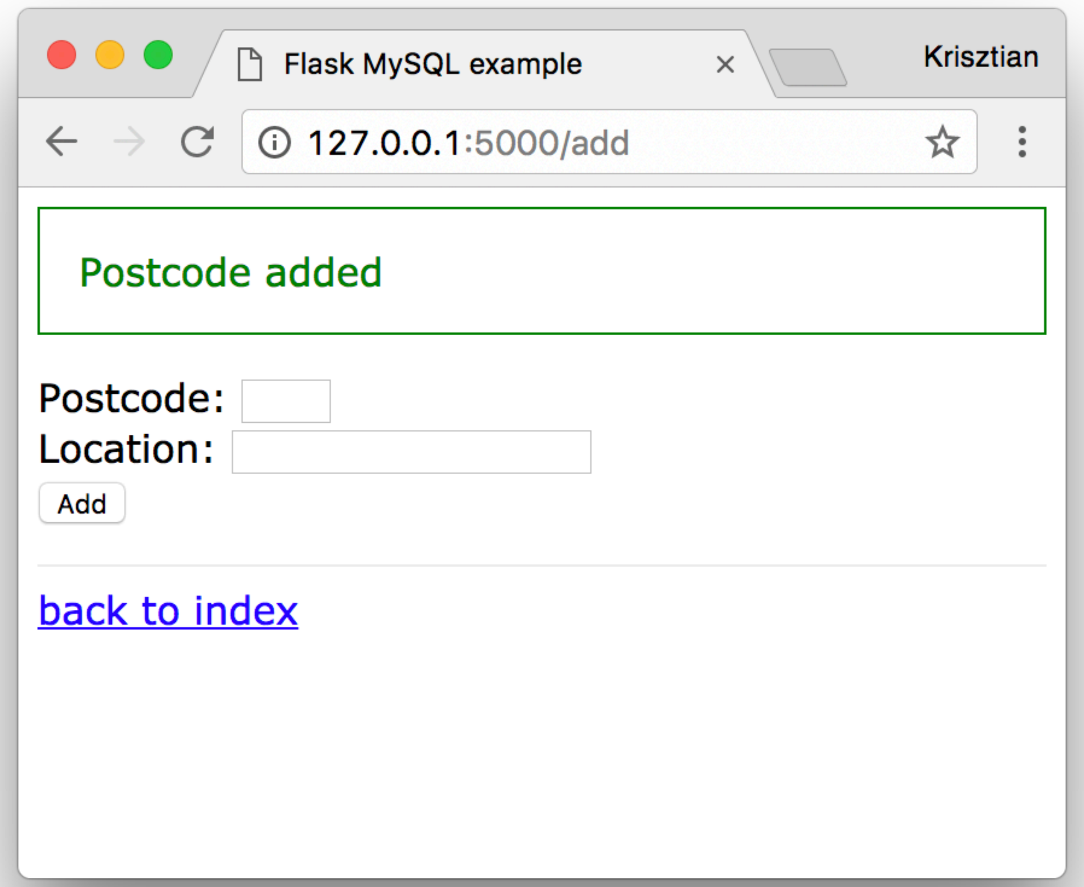
- Initialize DB
- List all postcodes
- Add postcode
- Look up postcode



Listing all postcodes

Postcode	Location	Operations
0001	Oslo	Delete
4036	Stavanger	Delete
4041	Hafrsfjord	Delete
7491	Trondheim	Delete
9019	Tromsø	Delete

back to index



Postcode added

Postcode:
Location:
Add

back to index

Exercises #1, #2

github.com/dat310-spring20/course-info/tree/master/
exercises/python/flask3

Resources

- Python Database API Specification
  https://www.python.org/dev/peps/pep-0249/
- MySQL Connector/Python
  https://dev.mysql.com/doc/connector-python/en/
- Flask
  http://flask.pocoo.org/docs/0.12/quickstart/#