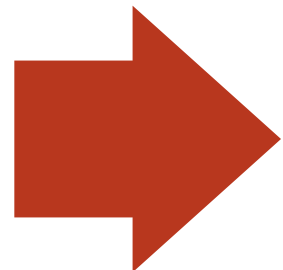


# Web Programming

## **Server-side programming III.**

# Server-side programming

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



**State 1: global variables**

# Example

- A global variable is read and written to

```
app = Flask(__name__)
```

```
postcodes = {  
    "0001": "Oslo"  
    ...  
}
```

Global variable in **app.py**

```
@app.route("/addpostnumber", methods=["POST"])
```

```
def addEntry():
```

```
    number = request.form.get("number", "")
```

```
    city = request.form.get("city", "")
```

```
    postcodes[number] = city
```

```
    return render_template("added.html")
```

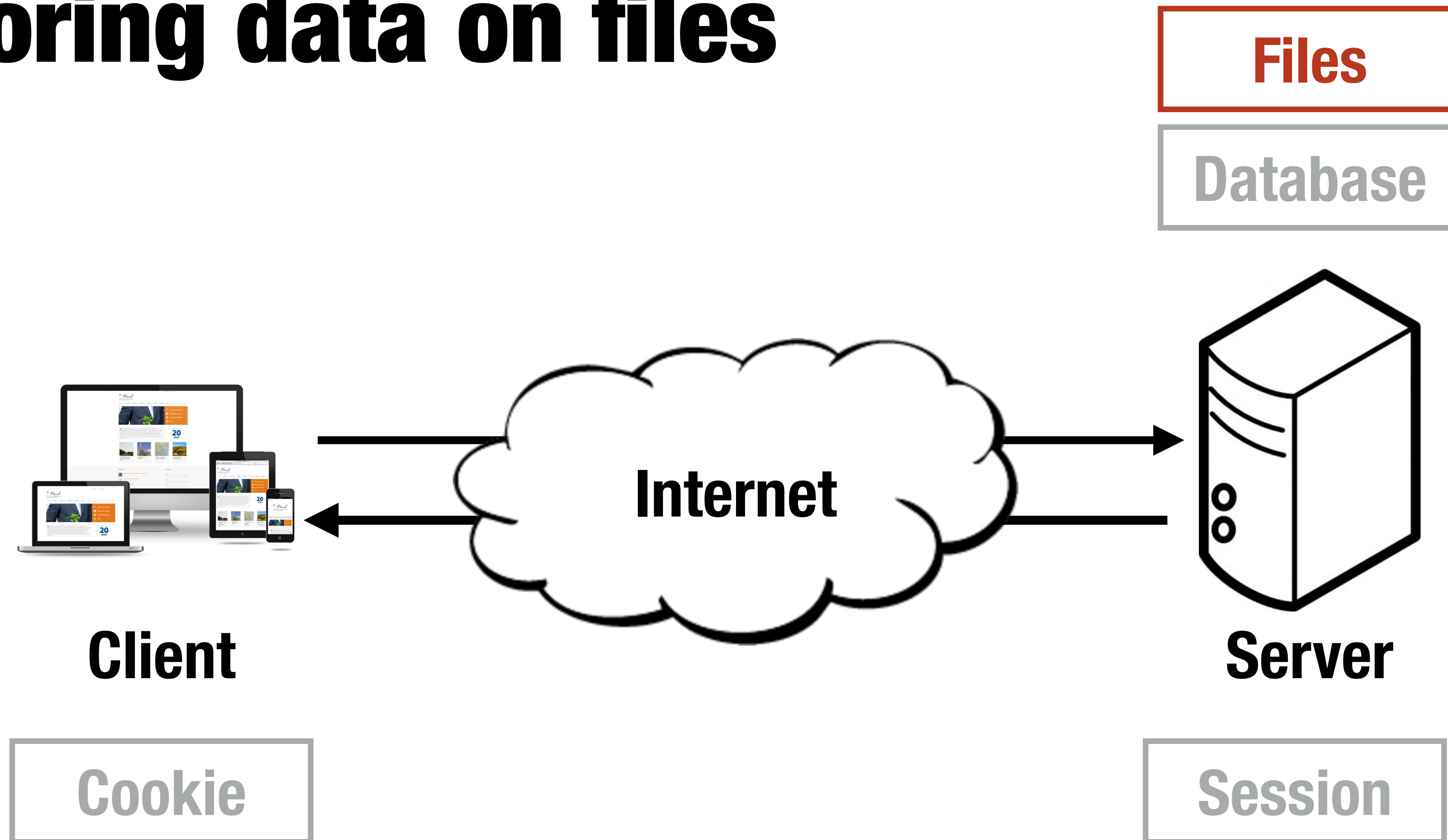
Updated in route

# State in global variable

- State is not persisted when program stops
- Not thread safe:
  - When multiple clients are connected, this may give:
    - Incorrect values
    - Program crashes

Not good in production. Use Lock.  
Ok for testing.

# Storing data on files



# JSON

- JavaScript Object Notation
- Lightweight data-interchange format
- Language independent
- Two structures
  - Collection of name-value pairs (object)
    - a.k.a. record, struct, dictionary, hash table, associative array
  - Ordered list of values (array)
    - a.k.a. vector, list

# JSON

- Values can be
  - string (in between "...")
  - number
  - object
  - array
  - boolean (true/false)
  - null



# Example JSON

```
{  
  "name": "John Smith",  
  "age": 32,  
  "married": true,  
  "interests": [1, 2, 3],  
  "other": {  
    "city": "Stavanger",  
    "postcode": 4041  
  }  
}
```

# JSON with Python

🔗 [examples/ajax/json/json\\_python.py](#)

- **json** is a standard module
- **json.dumps(data)**
  - returns JSON representation of the data
- **json.loads(json\_value)**
  - decodes a JSON value
- **json.dumps()** and **json.loads()** work with strings
- **json.dump()** and **json.load()** work with file streams

# Example

examples/python/flask/5\_json

fileaccess\_json.py

```
FILENAME = "postcodes.json"

def create_file(filename):
    open(filename, 'x')

def readJSON(filename):
    ...

def writeJSON(filename, data):
    jsonstring = json.dumps(data)
    with open(filename, "w") as f:
        f.write(jsonstring)

if __name__ == "__main__":
    postcodes = {
        "0001": "Oslo",
        ...
    }
    create_file(FILENAME)
    writeJSON(FILENAME, postcodes)
```

- readJSON returns parsed json or empty dict.
- writeJSON writes new object to file.
- run fileaccess\_json.py to create postcodes.json with init data.

Carefull, where the file is created.

# Example

examples/python/flask/5\_json

app.py

```
from fileaccess_json import readJSON,
writeJSON, FILENAME

app = Flask(__name__)
postcodes = readJSON(FILENAME)

@app.route("/addpostnumber",
methods=["POST"])
def addEntry():
    number = request.form.get("number",
    "")
    city = request.form.get("city", "")
    postcodes[number] = city
    writeJSON(FILENAME, postcodes)
```

- import readJSON and writeJSON from fileaccess\_json.py
- call readJSON to init global variable
- call writeJSON to update file

# State in JSON files

- State is persisted when program stops
- Not thread safe:
  - When multiple clients are connected, this may give:
    - Incorrect values
    - Program crashes
- Complex to update or read only parts
- No guarantees that data is correct

Not good in production. Use Lock.  
Ok for testing.

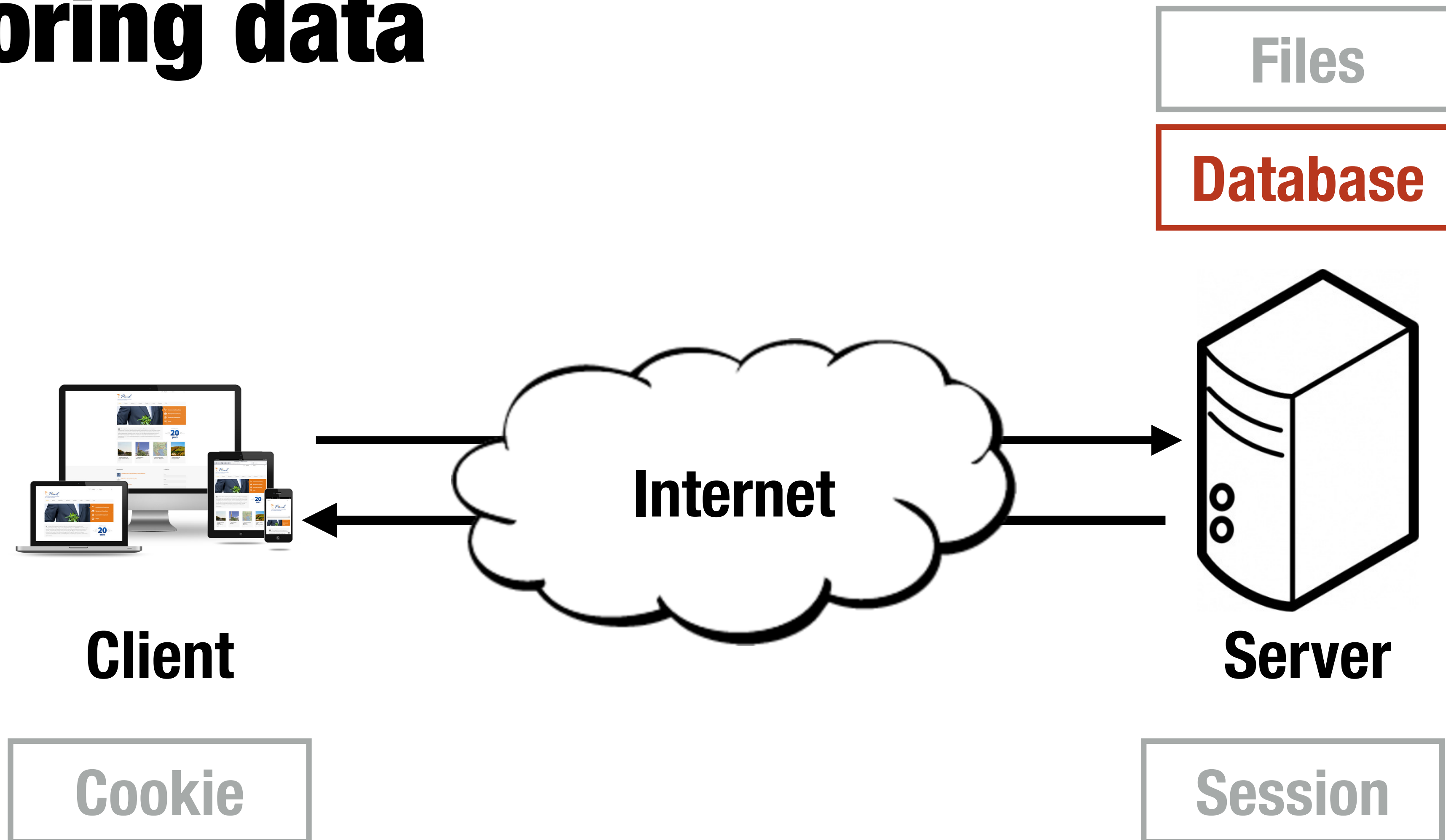
Here a CSV file would be better.

# Exercises #1

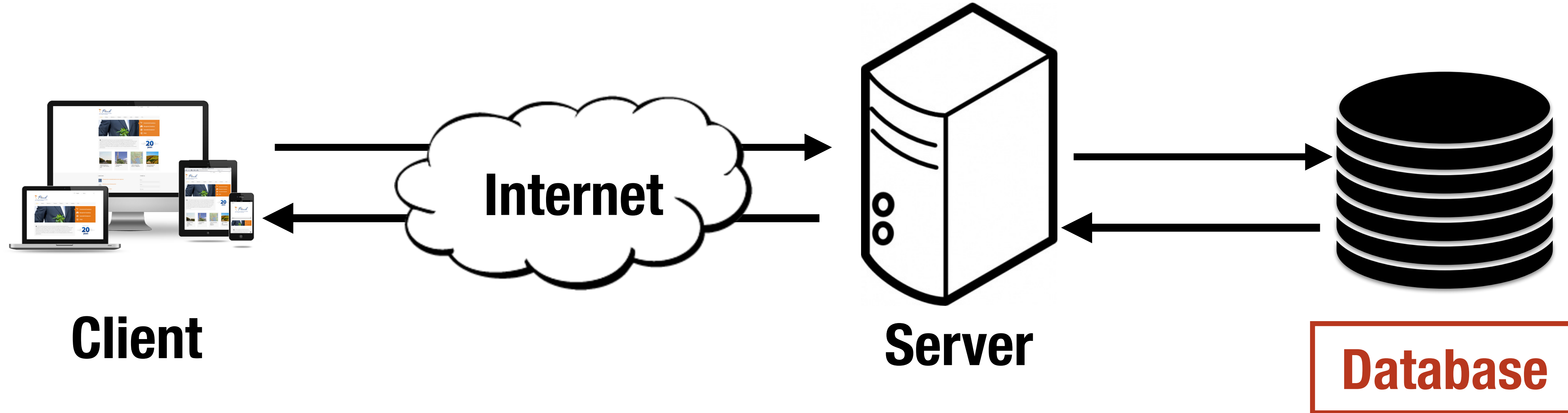


[github.com/dat310-2023/info/tree/master/  
\*\*exercises/python/flask3\*\*](https://github.com/dat310-2023/info/tree/master/exercises/python/flask3)

# Storing data



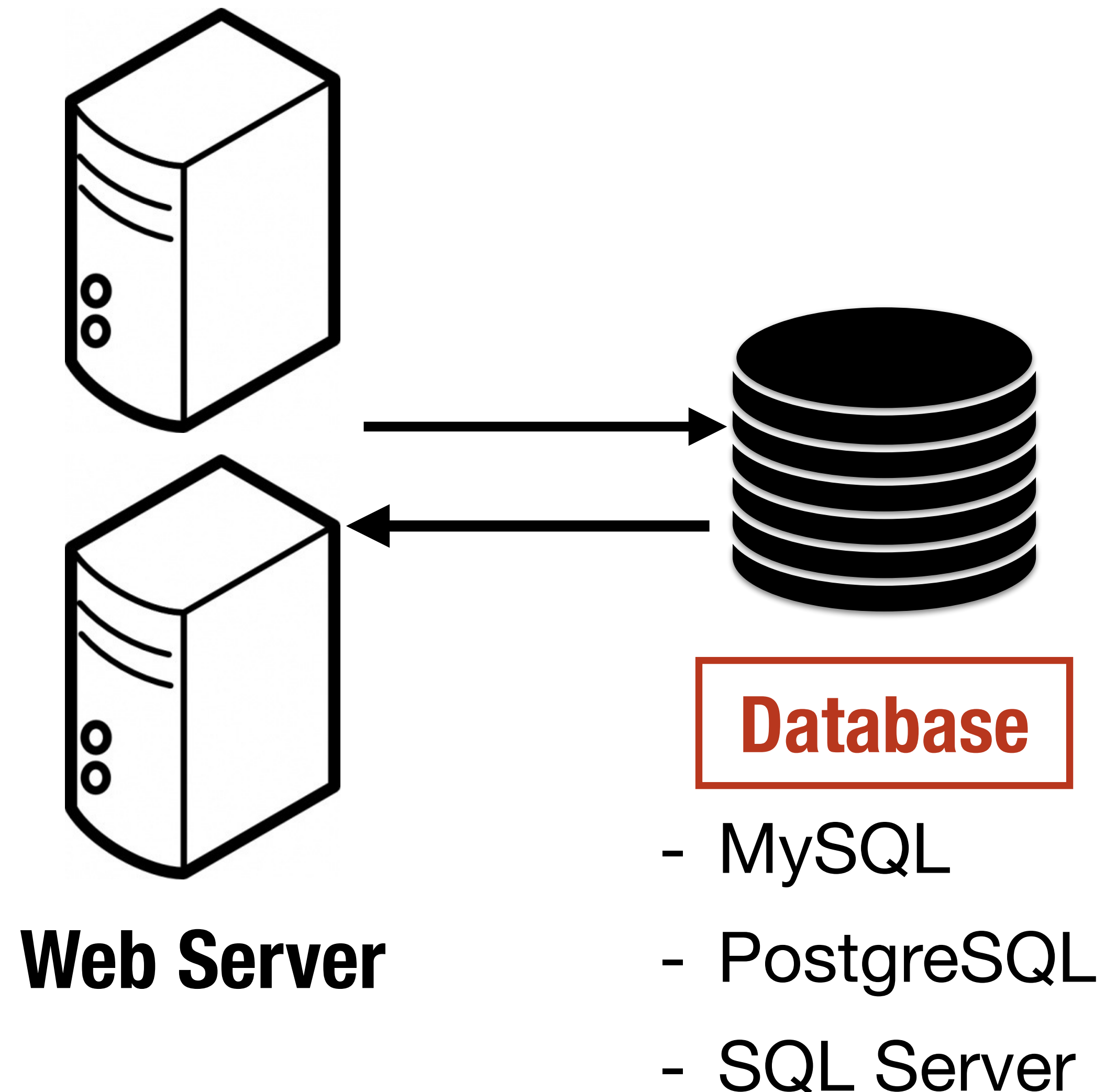
# Architecture





# Architecture

- Database server:
  - maintains state
  - stays consistent
- Web servers:
  - process client request
  - access state from database servers
  - may cache state but otherwise stateless



# SQLite

- Lightweight database:
  - Store database in a file
  - Good for prototyping and examples
  - Only for single webserver
  - Tutorial: <https://www.sqlitetutorial.net/>
  - Try it editor: <https://www.sqlitetutorial.net/tryit/>

# Learning goal:

- (Very) basic understanding of operations
  - Advantages of a database
- Use database from python

Not goal:

- Write complex SQL

**Passing the course does  
not require writing SQL.**

# Databases store data in Tables:

- Defined column names
- Same columns on every row

Can add information and constraints to column, e.g. type, length, non-empty

Postcodes	
number	city
"0001"	"Oslo"
"4036"	"Stavanger"
"4041"	"Hafslsfjord"
"7491"	"Trondheim"
"9019"	"Tromsø"

# CREATE TABLE

- Create a table with row names:

```
CREATE TABLE postcode (number, city);
```

```
CREATE TABLE <tablename>  
(<rowname>, <rowname>, ...);
```

Postcodes	
number	city
"0001"	"Oslo"
"4036"	"Stavanger"
"4041"	"Hafslsfjord"
"7491"	"Trondheim"
"9019"	"Tromsø"

# INSERT

- Insert a row into a table

```
INSERT INTO postcode (number, city) VALUES ('0001', 'Oslo');
```

```
INSERT INTO <tablename> (<rowname>,<rowname>) VALUES (value, value);
```

- Insert multiple values at once

```
INSERT INTO postcode (number, city) VALUES ('4036', 'Stavanger'), ('4024', 'Stavanger');
```

# SELECT

- Select named columns

```
SELECT number, city FROM postcode;
```

- Select all columns

```
SELECT * FROM postcode;
```

- Select rows with specific values

```
SELECT city FROM postcode WHERE number = '4036';
```

- Apply function to result, e.g. count rows:

```
SELECT COUNT(number) FROM postcode WHERE city = 'Stavanger';
```

# DELETE & UPATE

- DELETE rows with specific values

```
DELETE FROM postcode WHERE city = 'Stavanger' AND number = '4024';
```

- UPDATE rows with specific values

```
UPDATE postcode SET city = 'Svg.' WHERE city = 'Stavanger';
```



# Exercises #2



[github.com/dat310-2023/info/tree/master/](https://github.com/dat310-2023/info/tree/master/exercises/python/flask3)  
**exercises/python/flask3**

# Constraints

- Create a table with types and constraints

```
CREATE TABLE postcode
( number TEXT UNIQUE NOT NULL
    CHECK(length(number) == 4),
  city TEXT NOT NULL CHECK(length(number) > 0));
```

- Types:
  - TEXT, INTEGER, REAL

**Sometimes: PRIMARY KEY  
is used instead of UNIQUE**

Postcodes	
number	city
"0001"	"Oslo"
"4036"	"Stavanger"
"4041"	"Hafslsfjord"
"7491"	"Trondheim"
"9019"	"Tromsø"

# Rowid

- In SQLite, by default every row has a rowid

```
SELECT rowid, number, city FROM postcode;
```

- rowid is useful as object identity

Postcodes	
number	city
"0001"	"Oslo"
"4036"	"Stavanger"
"4041"	"Hafslsfjord"
"7491"	"Trondheim"
"9019"	"Tromsø"

# FOREIGN KEY

- Constraint connecting two tables
- Make sure student exists.

```
CREATE TABLE grades (  
  student INTEGER NOT NULL,  
  grade TEXT NOT NULL,  
  FOREIGN KEY (student)  
    REFERENCES students (student_no);
```

Grades

grade	student
"A"	"123456"
"B"	"222222"

```
CREATE TABLE students (  
  student_no INTEGER UNIQUE NOT NULL,  
  name TEXT NOT NULL;
```

Student

student_no	name
"123456"	"Tom"
"222222"	"Alice"

# Using SQLite 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 **PySQLite Connector/Python**
  - Included in the standard library
  - Similar interface to database servers
  - Tutorial: <https://www.sqlitetutorial.net/sqlite-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 sqlite3

conn = sqlite3.connect("database_file.db")

# do some stuff

conn.close()
```

- The **connect()** constructor creates a connection to the SQLite database and returns a **Connection** object
- **connect()** takes the path to a database file (absolute or relative). If the file does not exist a new database is created.



# Error Handling

📄 examples/python/sqlite/sqlite1.py

```
from sqlite3 import Error

try:
    conn = sqlite3.connect("database_file.db")
except Error as err:
    print(err)
else:
    # do some stuff
    conn.close()
```

All database statements should be done  
inside `try: except:`

# 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/sqlite/sqlite1.py

```
cur = conn.cursor()
try:
    sql = ("CREATE TABLE postcodes ("
           "postcode VARCHAR(4), "
           "location VARCHAR(20), "
           "PRIMARY KEY(postcode))")
    cur.execute(sql)
except Error as err:
    print("Error: {}".format(err))
else:
    print("Table created.")
finally:
    cur.close()
```

# Dropping a Table

📄 [examples/python/sqlite/sqlite1.py](#)

```
cur = conn.cursor()
try:
    sql = "DROP TABLE postcodes"
    cur.execute(sql)
except Error as err:
    print("Error: {}".format(err))
else:
    print("Table dropped.")
finally:
    cur.close()
```

# Inserting Data

📄 examples/python/sqlite/sqlite1.py

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

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

# Querying Data

📄 examples/python/sqlite/sqlite1.py

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

- Use **cur.fetchall()** to get list of row values

# 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 SQLite 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\_sqlite/app.py

```
def get_db():  
    if not hasattr(g, "_database"):  
        g._database = sqlite3.connect("database.db")  
    return g._database
```

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

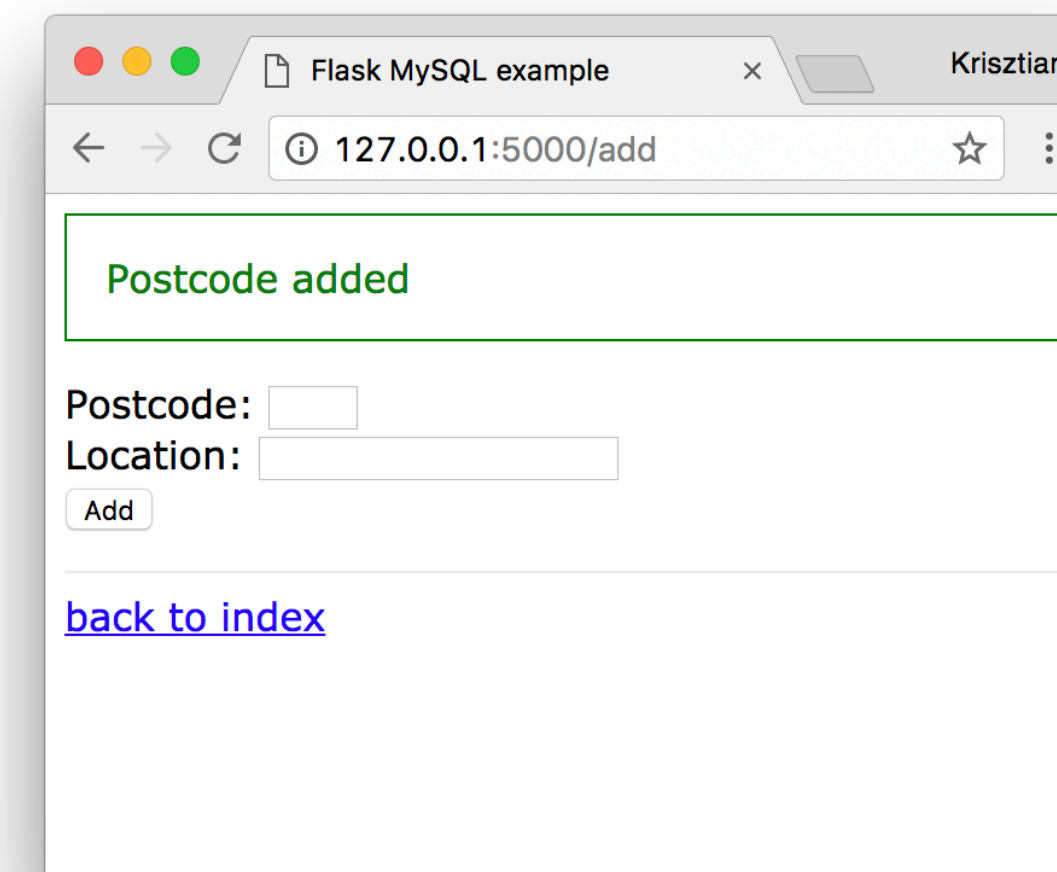
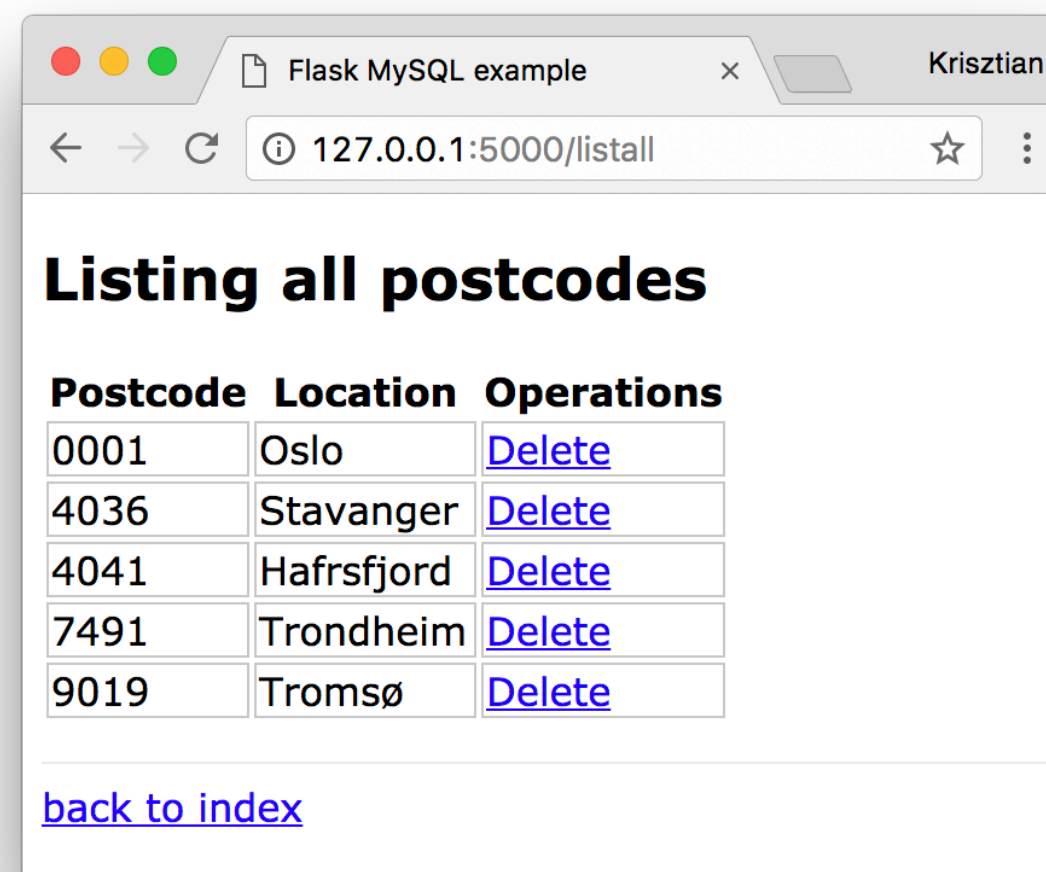
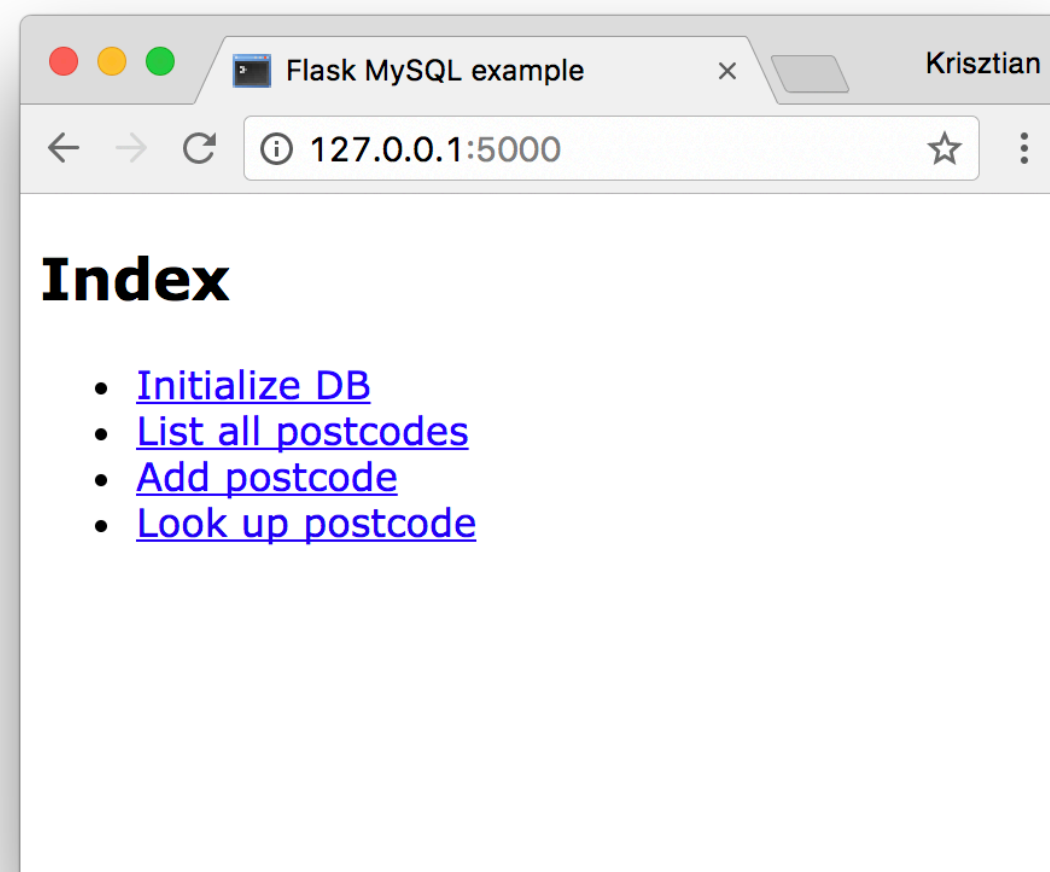
```
@app.route("/listall")  
def list_all():  
    """List all postcodes."""  
    db = get_db()  
    cur = db.cursor()
```

The first time **get\_db()** is called the connection will be established

# Example

🔗 [examples/python/flask/5\\_sqlite/app.py](#)

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



# Exercises #1, #2



[github.com/dat310-spring21/course-info/tree/master/](https://github.com/dat310-spring21/course-info/tree/master/exercises/python/flask3)  
**exercises/python/flask3**

# Resources

- Python Database API Specification  
<https://www.python.org/dev/peps/pep-0249/>
- SQLite3 Connector/Python  
<https://docs.python.org/3/library/sqlite3.html>
- Flask SQLite  
<https://flask.palletsprojects.com/en/1.1.x/patterns/sqlite3/>
- SQLite CLI  
<https://sqlite.org/cli.html>