TÉCNICO LISBOA

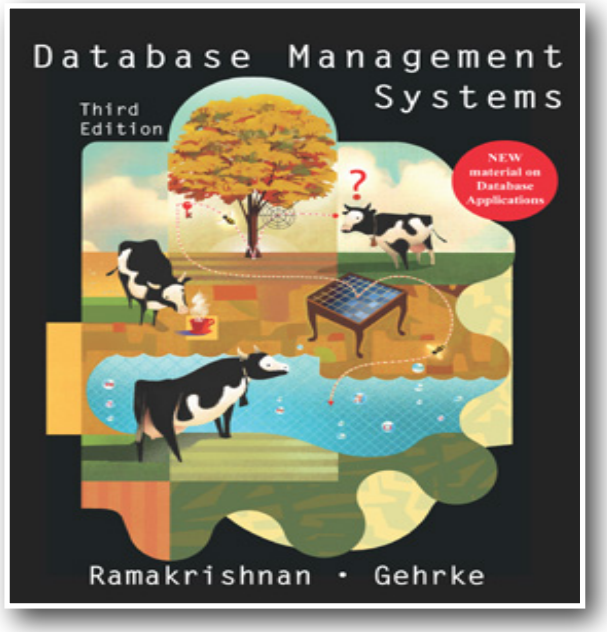
Sistemas de Informação e Bases de Dados

Class 15: Application Development (Python)

Prof. Paulo Carreira

slides não são livros

Bibliography



Chapters 6 e 7

Bibliography

- **Python Tutorial:** https://www.w3schools.com/python/

- **Python Data Access:** https://www.psycopg.org/docs/usage.html

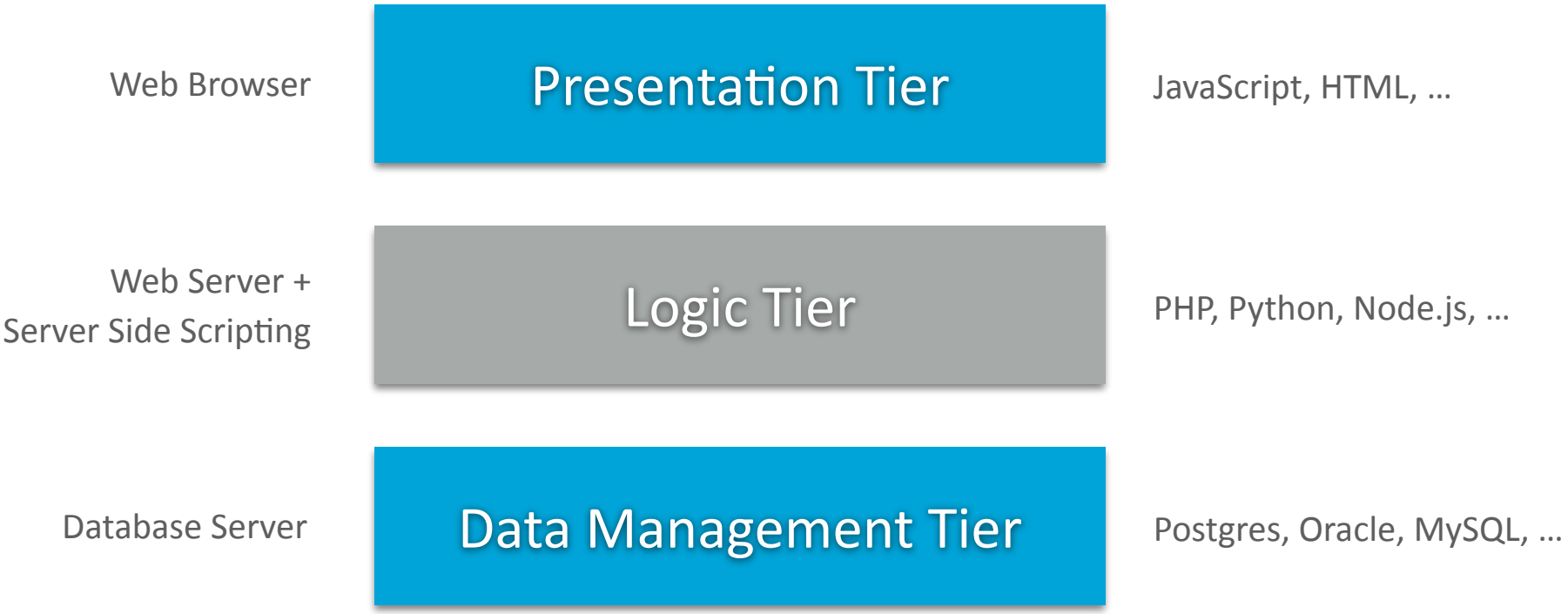
Class Outline

- Web Applications: Typical 3-tier architecture
- HTTP and HTML Basics
- Interoperability with SQL
- Python and Psycodb DB access
- Accessing page form parameters

Typical Architecture for Web Applications

Typical deployment architecture of DB applications

3-Tier architecture



Advantages of 3-Tiered architectures

- **Independence between layers**: Changes in one layer do not affect the other (as long as the API and data schemas are not changed)
- **Faster development**: Easier reuse of components already implemented in lower layers
- **Scalability**: Many clients can connect to the same server, and many nodes can be conjoined as servers
- **Thin clients**: Clients do not consume many resources. Only interpret HTML and JS.

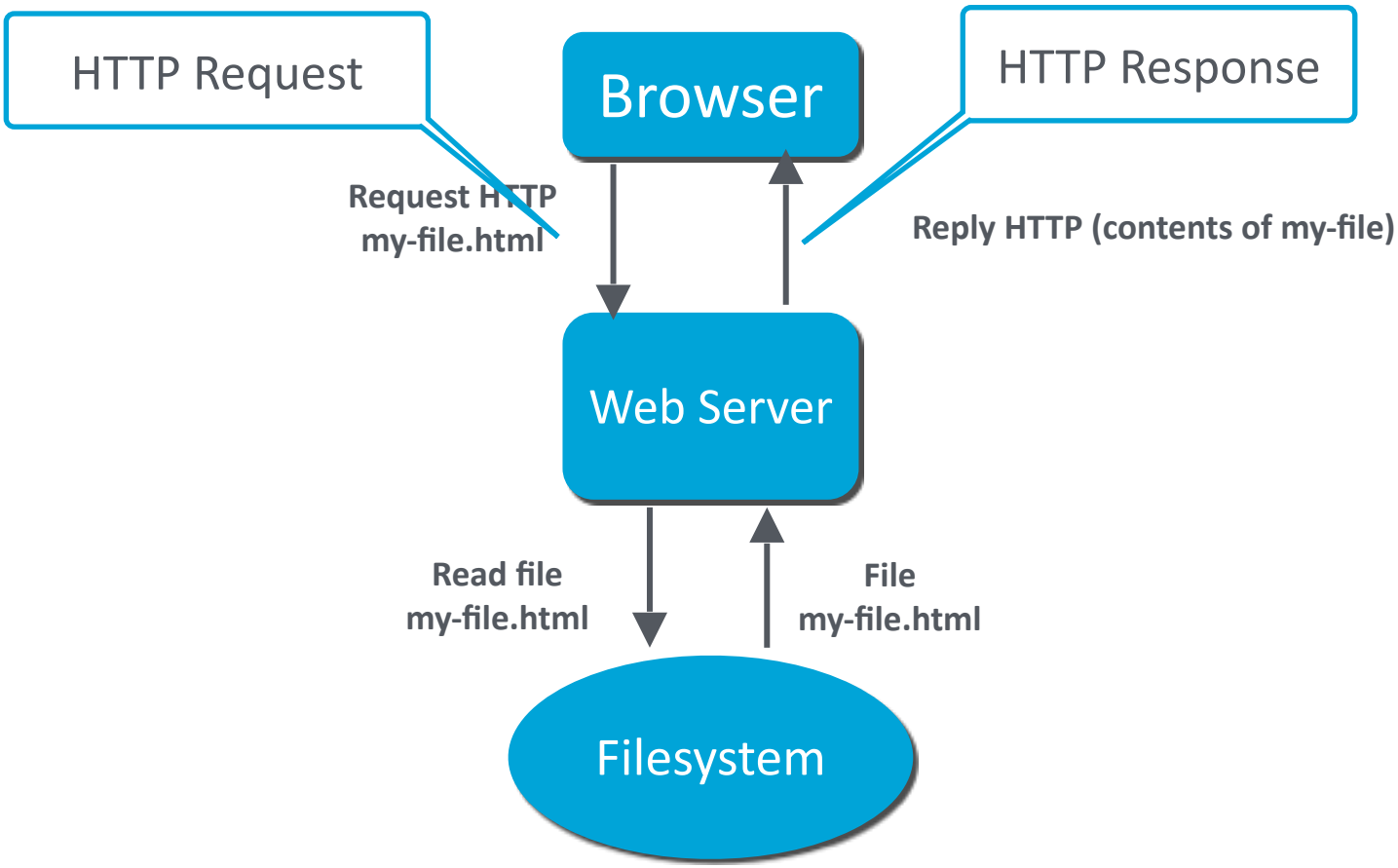
HTTP and HTML

HTTP Protocol

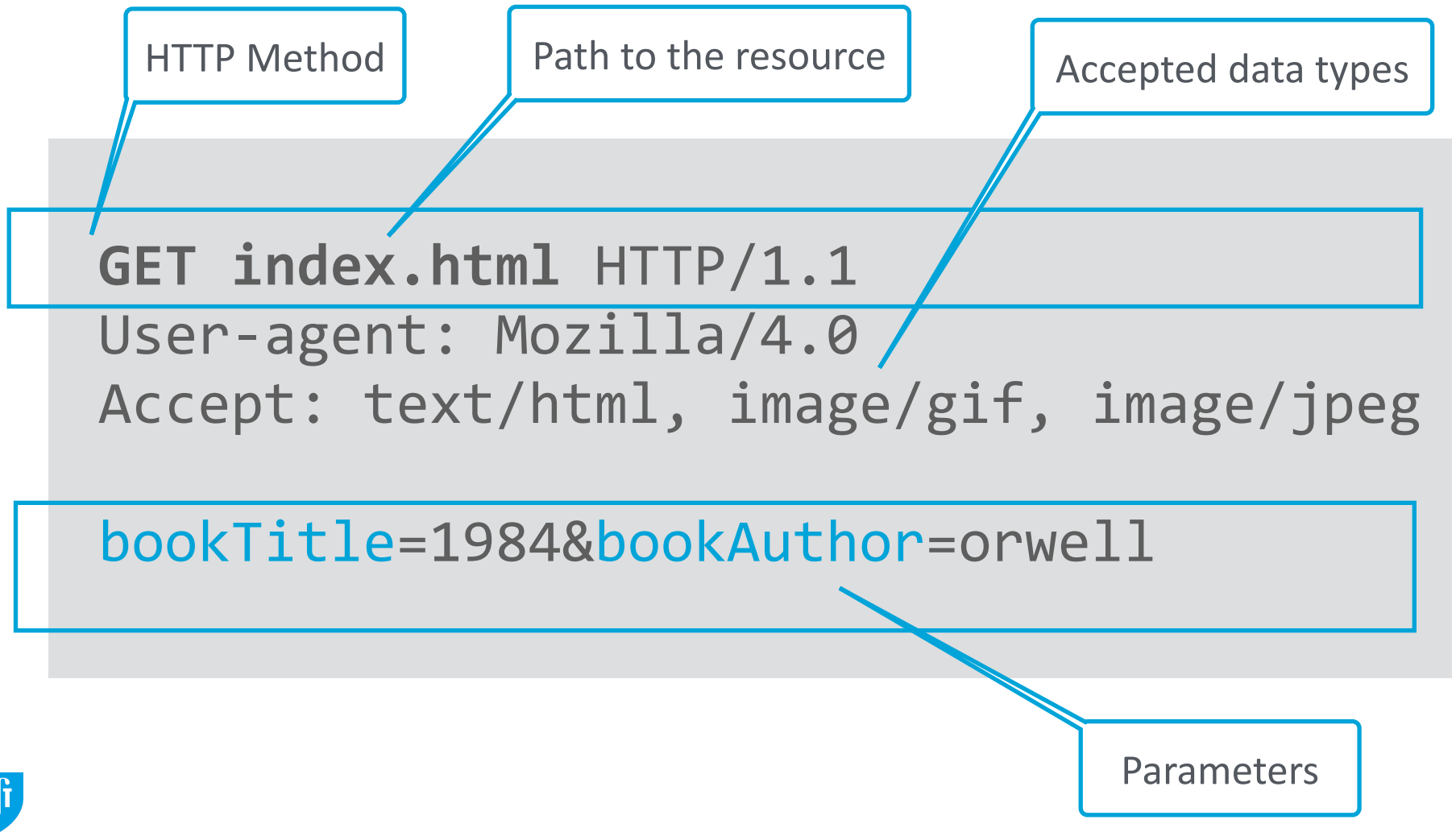
- Client (e.g. web browser or app) sends request to the HTTP server
- Server sends the response to the client

***URI** (Uniform Resource Identifiers): Naming scheme to locate resources on the internet*

Handling an HTML Request



Example of an HTTP request



Example of an HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 04 Mar 2002 12:00:00 GMT
Content-Length: 1024
Content-Type: text/html
Last-Modified: Mall, 22 JUIl 1998 09:23:24 GMT
<HTML>
<HEAD>
</HEAD>
<BODY>
 <H1>Barns and Nobble Internet Bookstore</H1>
 Our inventory:
 <H3>Fiction</H3>
 <B>1984 by George Orwell</B>
</BODY>
</HTML>
```

HTML

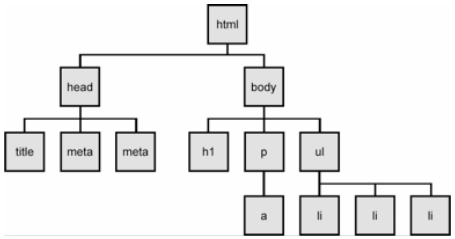
- A ***Markup Language*** that annotates text with start tag and an end tag:
  - `<HTML> … </HTML>`
  - `<TITLE> … </TITLE>`
  - `<H3> … </H3>`
  - `<A HREF="…">This is a link</A>`
- Web Browsers present the contents graphically to the user

HTML Rendering

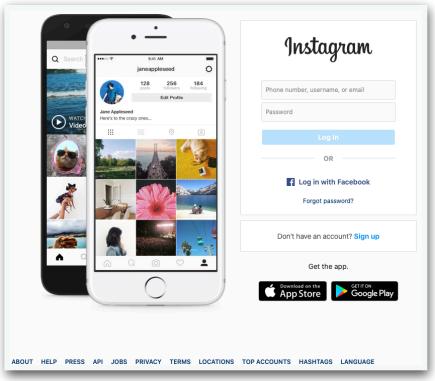
HTML

```
<!DOCTYPE html>
<html lang="en" class="no-js not-logged-in client-root">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">

        <title>
Instagram
</title>


        <meta name="robots" content="noimageindex, noarchive">
        <meta name="apple-mobile-web-app-status-bar-style" content="default">
        <meta name="mobile-web-app-capable" content="yes">
        <meta name="theme-color" content="#ffffff">
        <meta id="viewport" name="viewport" content="width=device-width, initial-scale=1,
minimum-scale=1, maximum-scale=1, viewport-fit=cover">
        <link rel="manifest" href="/data/manifest.json">

        <link rel="preload" href="/static/bundles/es6/ConsumerUICommons.css/71b101bb9e76.css"
as="style" type="text/css" crossorigin="anonymous" />
<link rel="preload" href="/static/bundles/es6/ConsumerAsyncCommons.css/871d296b773f.css"
as="style" type="text/css" crossorigin="anonymous" />
<link rel="preload" href="/static/bundles/es6/Consumer.css/15430a8f31c7.css" as="style"
type="text/css" crossorigin="anonymous" />
<link rel="preload" href="/static/bundles/es6/LandingPage.css/8751804c0079.css" as="style"
type="text/css" crossorigin="anonymous" />
<link rel="preload" href="/static/bundles/es6/Vendor.js/c911f5848b78.js" as="script"
type="text/javascript" crossorigin="anonymous" />
<link rel="preload" href="/static/bundles/es6/en_US.js/c3365f7d8e25.js" as="script"
type="text/javascript" crossorigin="anonymous" />
<link rel="preload" href="/static/bundles/es6/ConsumerLibCommons.js/6a97ea9cdf23.js" as="script"
type="text/javascript" crossorigin="anonymous" />
```

Element Tree



Render



A basic (static) HTML table

This content should come from the database

```
<!DOCTYPE html>
<html>
<body>

<h2>Basic Employee Table</h2>

<table style="width:100%">
  <tr>
    <th>ID</th>
    <th>Name</th>
    <th>Birthdate</th>
  </tr>
  <tr>
    <td>1</td>
    <td>Alice</td>
    <td>1995-10-10</td>
  </tr>
  <tr>
    <td>2</td>
    <td>Bob</td>
    <td>1996-03-02</td>
  </tr>
  <tr>
    <td>4</td>
    <td>Daniel</td>
    <td>1998-04-04</td>
  </tr>
</table>

</body>
</html>
```

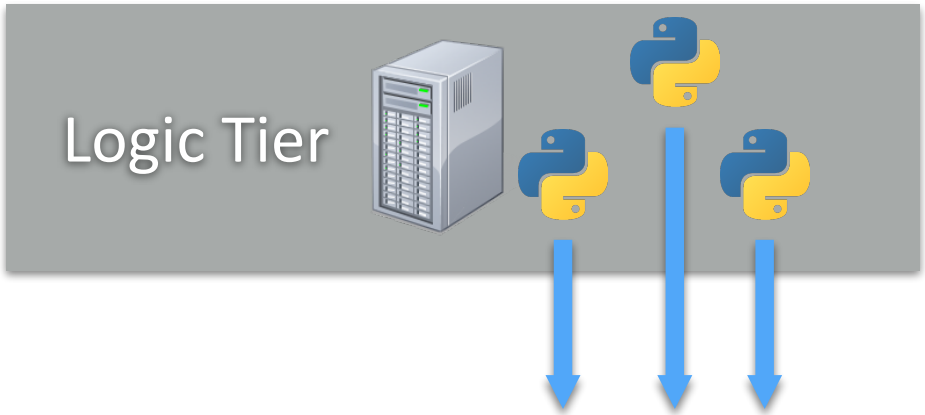
Basic Employee Table

ID	Name	Birthdate
1	Alice	1995-10-10
2	Bob	1996-03-02
4	Daniel	1998-04-04

Web & Python

Web Clients

Web Server
(Apache)



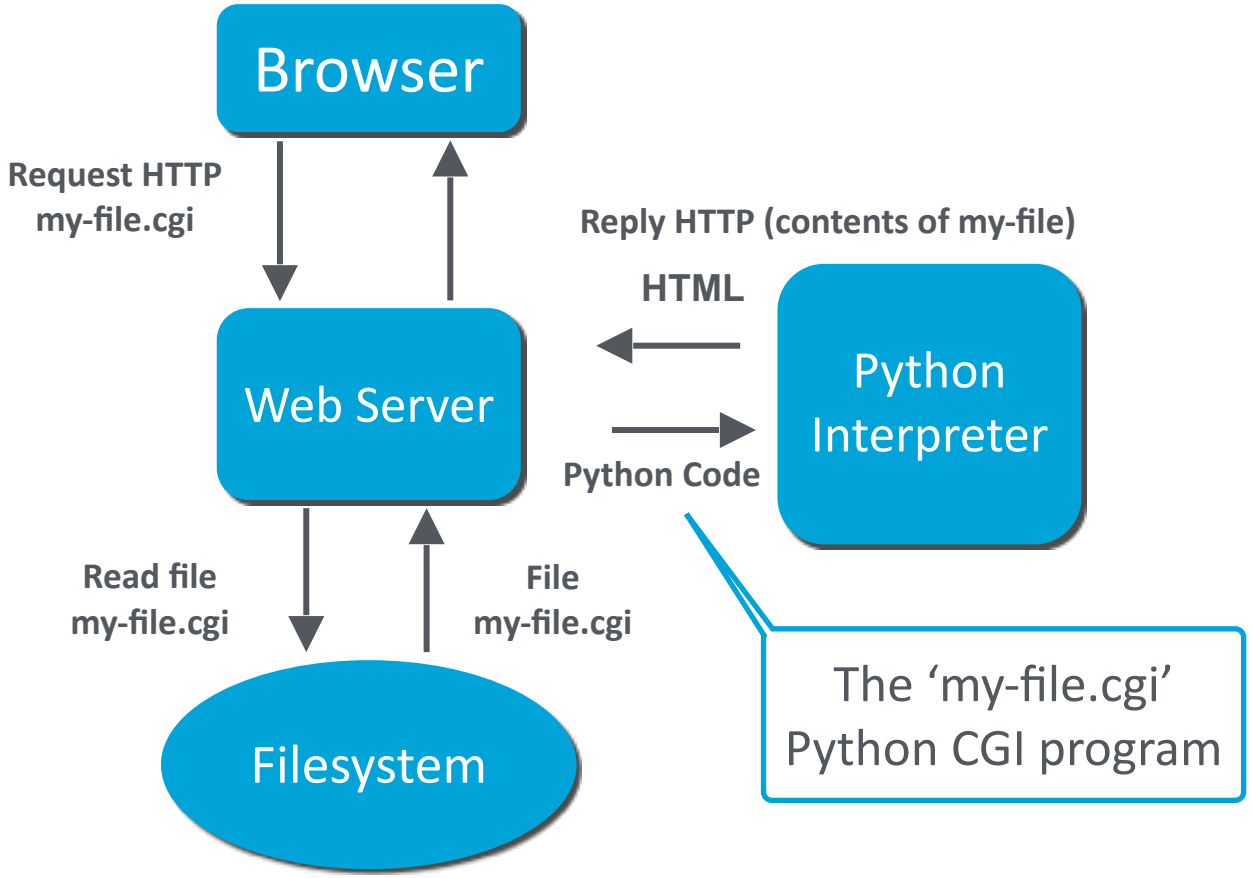
Database
Server

Python Language

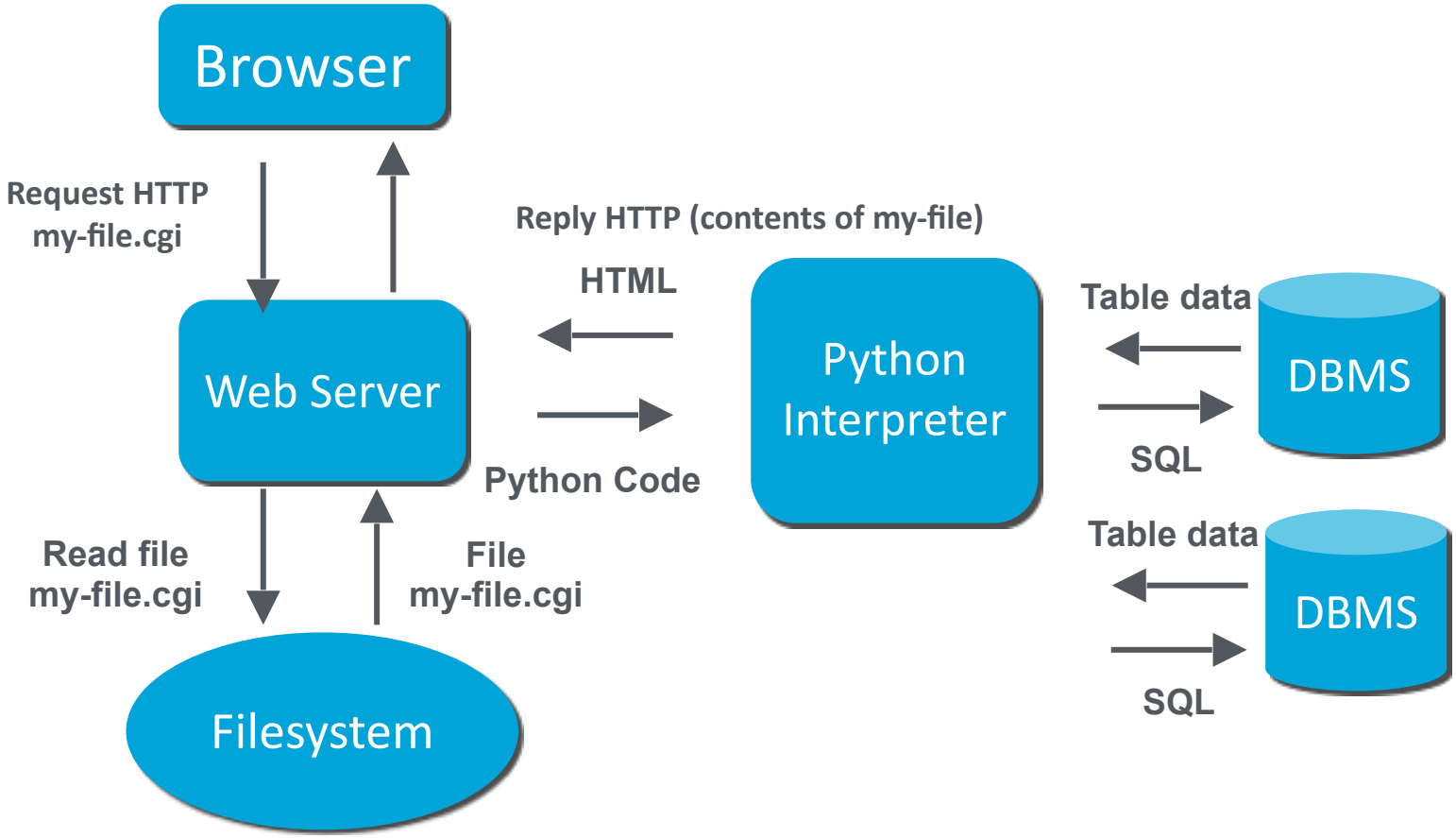
Logic Tier

- Scripting language
- Runs on many platforms
- Dynamic generation of web pages
- Integrated with multiple Web servers
- Support for many DBMS

Handeling a Python CGI Request

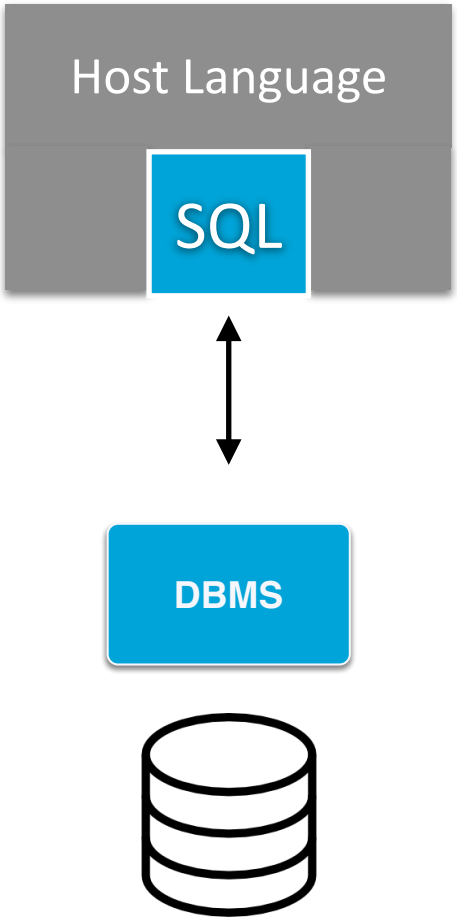


Handling a Python CGI request with DB access



Interoperability with SQL

The *Impedance Mismatch Problem*



- SQL handles **tables** and **rows**
- Host language manipulates **variables**, **objects**, **pointers**
- Typically, no construction in the host language that allows us to manipulate (relational) **tables** and **rows**!

Why not use just one language?

- **Forgetting SQL**: It's not a good idea! (why?)
- **Extending SQL**: Also not a good idea!

Two approaches to Integration

A. **Static Approach**: SQL embedded in the language

- Embedded SQL
- SQLJ, an extension to Java

B. **Dynamic approach**: APIs to invoke SQL commands

- Dynamic SQL
- JDBC (Java DataBase Connectivity)
- Python (via Psycopg)

The approach used nowadays!

Data Adaptation

- The output of an SQL query is a set of rows (a Table)
- Since the Table type of data does not normally exist in programming languages, SQL integration uses the Cursor as an adaptation mechanism
- Cursor objects or data structures enable iterating over the rows of a result

⚠ Cursors are the abstraction used by most languages

Introduction to Python & Psycopg

Hello World

```
print('Content-type:text/html\n\n')
print('<html>')
print('<head>')
print('<title>Python CGI Test</title>')
print('</head>')
print('<body>')
#
# More Python code here…
#
print('</body>')
print('</html>')
```

HTML Tags

Python code

Generic Block

```
# Print header
# Get credentials and build the DSN string
connection = None
try:
  # Create a connection
  connection = psycopg2.connect(dsn)
  print(“<p>Connected</p>");

  # code for query/update here

  # Free the connection
  connection.close()

except Exception as error:
  print('<h1>An error occurred.</h1>')
  print('<p>{}</p>'.format(error))
finally:
 if connection is not None:
   connection.close()
# Print page footer
```

SQL Queries and iteration over results

Error handling

Creating a Connection

```
#!/usr/bin/python3
import psycopg2

ist_id = 'ist12345'
host = 'db.tecnico.ulisboa.pt'
port = 5432
password = 'xpto123'
db_name = ist_id

dsn = 'host={} port={} user={} password={} dbname={}'.format(host, port, ist_id,
     password, db_name)

# Page header code …

try:
  # Creating connection
  connection = psycopg2.connect(dsn)
  print('<p>Connected to Postgres with: {}.</p>'.format(dsn))

  # More code…
```

Connection String

Connection object

Submitting an SQL query

```
cursor = connection.cursor()
sql = 'SELECT * FROM account;'
cursor.execute(sql)
```

Get a new cursor from the connection object

Execute the query and populate the cursor

Finding the number of columns and rows

```
result = cursor.fetchall()

row_count = len(result)
print('<p>How many rows: {}</p>'.format(row_count))

col_count = len(cursor.description)
print('<p>How many columns: {}</p>'.format(col_count))
```

Processing the results

```
print('<table border=“5”>');

for row in result:
    print('<tr>')
    for value in row:
       print(‘<td>{}</td>'.format(value))
    print('</tr>')

print ('</table>');
```

Iterate for each row

Iterate for each value

Termination

Make sure to always close the cursors and the connection

```
 # DB access body!

 # Closing the cursor
  cursor.close()
  print('<p>Test completed successfully.</p>')

except Exception as error:
  print('<h1>An error occurred.</h1>')
  print('<p>{}</p>'.format(error))
finally:
  if connection is not None:
        connection.close()
        print('<p>Connection closed.</p>')
```

Inserting records in the DB

```
cursor = connection.cursor()
sql = "INSERT INTO account VALUES('B-101', 'Downtown',
   500);"
cursor.execute(sql)
connection.commit()
```

SQL statement

Changes only visible in the current transaction

Make changes permanent

Changes are only visible to the application. To make changes permanent, the current transaction has to be commited.

Parametric query

```
cursor = connection.cursor()
sql = """
   SELECT * FROM account
   WHERE account_number = %s;"""
cursor.execute(sql, ['A-101'])
result = cursor.fetchall()
```

Make the statement parametric

Pass the value as a python Tuple

Inserting records in the DB

```
cursor = connection.cursor()
sql = "INSERT INTO account VALUES(%s, %s, %s)"
cursor.execute(sql, ['B-102', 'Downtown', 500])
connection.commit()
```

Make the statement parametric

Pass the values as a python Tuple

Similar for **INSERT** and **DELETE**

Updating the DB

```
cursor = connection.cursor()
  sql = """
     UPDATE account
     SET balance = %s
     WHERE account_number = %s"""
  cursor.execute(sql, [1000, 'B-102'])

  connection.commit()

  count = cursor.rowcount
  print(count, ' rows updated')
```

Account to be changed

New amount

We can always find the number of rows affect after each statement using `cursor.rowcount`

Parametric SQL

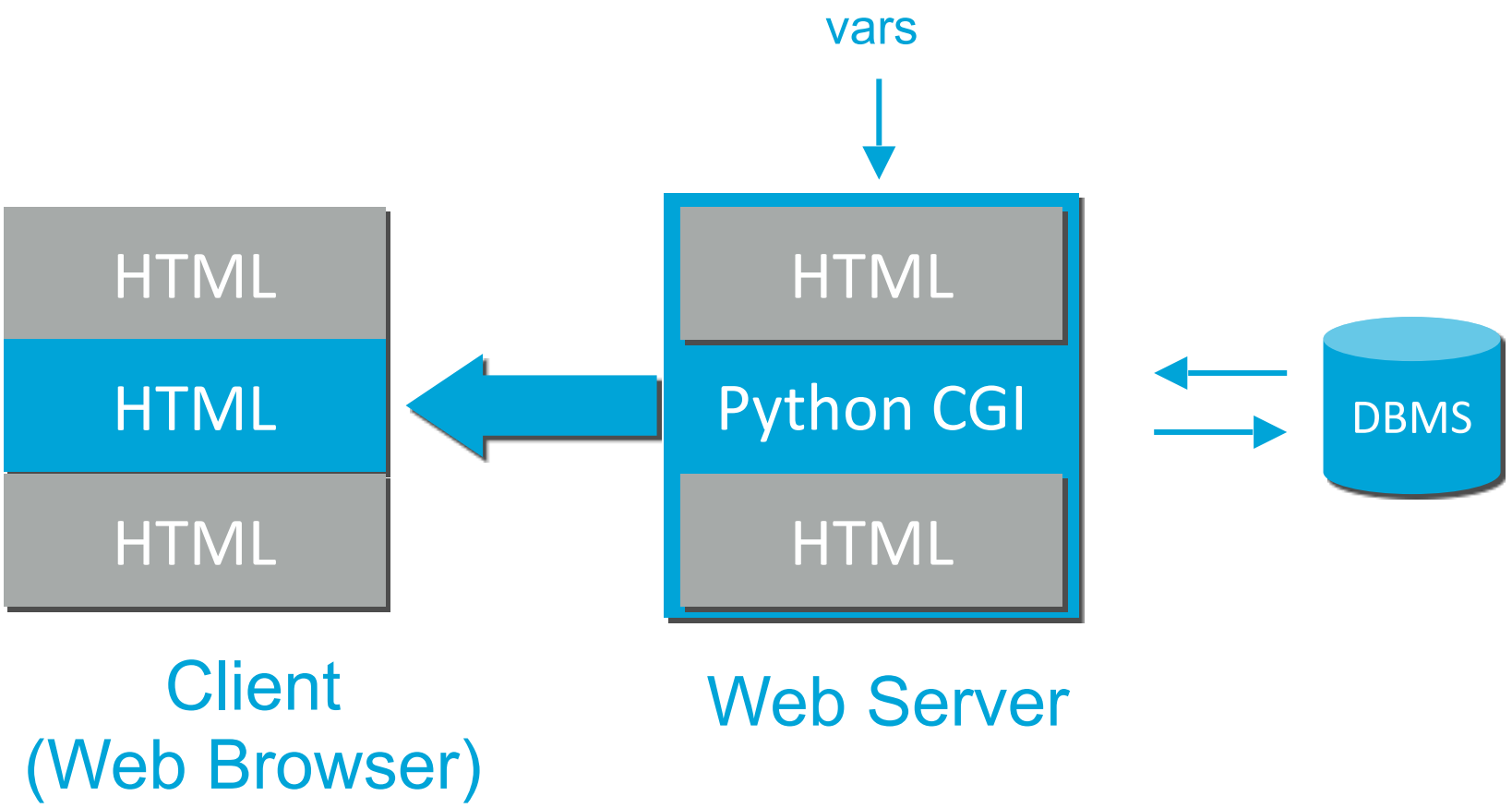
- Specify the names as ***%(name)s***
- Call `cursor.execute` with a dictionary
- Use the correct native types

```
# Remember to import datetime

cursor = connection.cursor()

sql = "INSERT INTO employee VALUES(%(eid)s, %(name)s, %
   (birthdate)s)"

cursor.execute(sql, {'eid': 8, 'name': 'Hubert',
   'birthdate': datetime.date(1999, 11, 18)})

connection.commit()
```

Page Parameters

Generation of HTML from Python CGI

URL http://host.domain/path?var1=val1 …



Parameters from URLs

URL: http://example.com/myexample.cgi?account=A-101

```
import cgi

# 1- Get the web page parameters
form = cgi.FieldStorage()
acct_num = form.getvalue('account')

# 2- Query the table account based on the parameter
cursor = connection.cursor()
sql = """
  SELECT * FROM account
 WHERE account_number = %s;"""
cursor.execute(sql, [acct_num])
result = cursor.fetchall()

# 3- Display the results…
```

TÉCNICO LISBOA

Sistemas de Informação e Bases de Dados

Aula 16: Application Development (cont)

Prof. Paulo Carreira

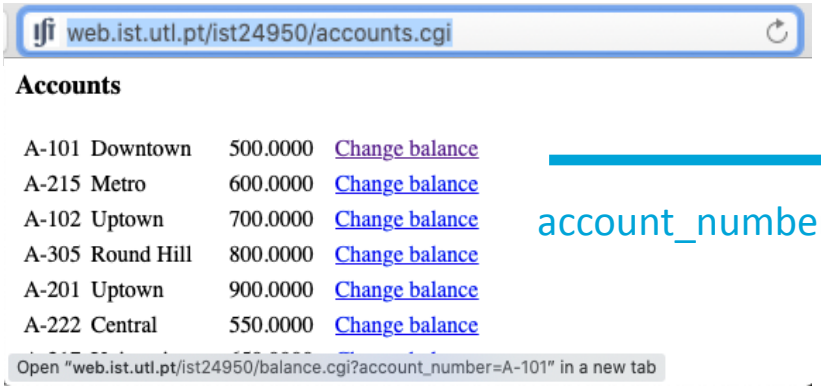
Class Outline

- Dynamic links and forms
- Error handling CGI page errors
- Introduction to transactions
- Handeling transactions in Python
- SQL-Injection

Dynamic Links and Forms

Simple forms

accounts.cgi



account_number=A-101

balance.cgi

account_number=A-101
&value=200

update.cgi

Balance update successful

Dynamic Links

```
# …

# Displaying results
result = cursor.fetchall()
print('<table border="0" cellspacing="5">')
for row in result:
   print('<tr>')
    acct_num = row[0]
   print('<td>{}</td>'.format(acct_num))
   print('<td>{}</td>'.format(row[1]))
   print('<td>{}</td>'.format(row[2]))
   print('<td><a href="balance.cgi?account_number={}">Change
  balance</a></td>'.format(acct_num))
   print('</tr>')
print('</table>')

#…
```

This dynamic link will call the ***balance.cgi*** script with the *account-number* parameter set

Forms

```
# …

acct_num = form.getvalue('account_number')
print('<h3>Change balance for account {}</h3>'.format(acct_num))

# The form will send the info needed for the SQL query
print('<form action="update.cgi" method="post">')
print('<p><input type="hidden" name="account_number" value="{}"/></
    p>'.format(acct_num))
print('<p>New balance: <input type="text" name="balance"/></p>')
print('<p><input type="submit" value="Submit"/></p>')
print('</form>')


#…
```

This form will call the ***update.cgi*** script with the *account-number* and *balance* parameters set

update.cgi

```
# …

acct_num = form.getvalue('account_number')
balance = form.getvalue('balance')

# create the query
sql = """
   UPDATE account
   SET balance = %s
   WHERE account_number = '%s';"""

data = (balance, acct_num)
print('<p>Executing: {}</p>'.format(sql % data))

cursor.execute(sql, data)


#…
```

Page Errors

Page Error

Executing a CGI script sometimes result in an execution error that will not be printed on the page

Running the script in the console

The best way to find what is wrong with the python CGI script is to run it directly in the console

1. Enter the web folder
2. Give the script execution privilege
3. Execute it…

```
sigma03:~$ cd web
sigma03:~/web$ chmod +x my_script.cgi
sigma03:~/web$ ./my_script.cgi
```

Script execution

The result of executing the script (HTML printout) will be shown in the console along with any errors

```
Content-type:text/html
<html>
<head>
<title>Lab 09</title>
</head>
<body>
<h3>Accounts</h3>
<h1>An error occurred.</h1>
<p>FATAL:  password authentication failed for user
"ist1XXXXX"
FATAL:  password authentication failed for user "ist1XXXXX"
</p>
</body>
</html>
```

Transactions

Definition of Transaction

A **Transaction** is the abstract view that the DBMS has of a logical and coherent unit (a sequence) of reads and writes that change the state of the database

Multiple transactions can be executed at the same time

DBMS ensure that transactions execute independently (without interfering with one another.)

Example

Transferring money from account A to account B:

```
UPDATE account
SET balance = balance - 50
WHERE account_num = A;
```

```
UPDATE account
SET balance = balance + 50
WHERE account_num = B;
```

```
1. read(A)
2. A := A – 50
3. write(A)

4. read(B)
5. B := B + 50
6. write(B)
```

Same idea for a trip reservation

Example

100 💶

200 💶

t

```
UPDATE account
SET balance = balance - 50
WHERE account_num = A;
```

```
UPDATE account
SET balance = balance + 50
WHERE account_num = B;
```

```
UPDATE account
SET balance = balance * 0.01
WHERE account_num = A;
```

```
UPDATE account
SET balance = balance * 0.01
WHERE account_num = B;
```

50.5 💶

252.5 💶

Example

100 

t

200 

```
UPDATE account
SET balance = balance * 0.01
WHERE account_num = A
```

```
UPDATE account
SET balance = balance * 0.01
WHERE account_num = B
```

```
UPDATE account
SET balance = balance - 50
WHERE account_num = A;
```

```
UPDATE account
SET balance = balance + 50
WHERE account_num = B;
```

51 

252 

Example

100 💶

t

200 💶

```
UPDATE account
SET balance = balance * 0.01
WHERE account_num = A;
```

```
UPDATE account
SET balance = balance - 50
WHERE account_num = A;
```

```
UPDATE account
SET balance = balance + 50
WHERE account_num = B;
```

```
UPDATE account
SET balance = balance * 0.01
WHERE account_num = B;
```

51 💶

202.5 💶

Preventing interference

100 

```
START TRANSACTION;

UPDATE account
SET balance = balance - 50
WHERE account_num = A;

UPDATE account
SET balance = balance + 50
WHERE account_num = B;

COMMIT;
```

50.5 

51 

200 

```
START TRANSACTION;

UPDATE account
SET balance = balance * 0.01
WHERE account_num = A;

UPDATE account
SET balance = balance * 0.01
WHERE account_num = B;

COMMIT;
```

252.5 

252 

Explicit Rollback

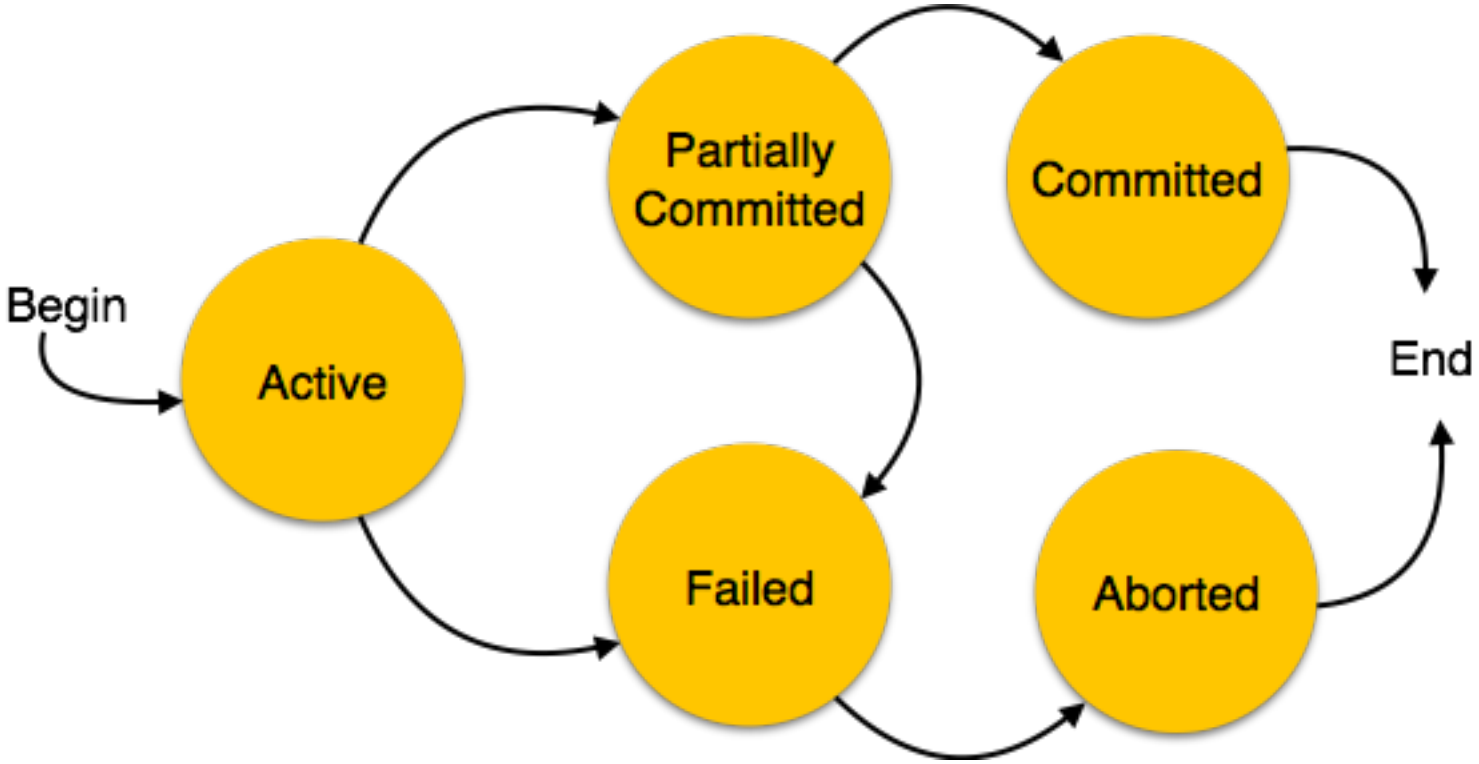
```
START TRANSACTION;
```

```
SELECT balance
INTO bal_var
WHERE account_num = A;
```

```
IF balance - 50 < 0 THEN
   ROLLBACK;
ELSE
    UPDATE account
    SET balance = balance - 50
    WHERE account_num = A;

    UPDATE account
    SET balance = balance + 50
    WHERE account_num = B;
END IF;
```

```
COMMIT;
```

Transaction States



Transaction States

- **Active** – The transaction is being executed; no permanent changes have been made; changes can be undone.
- **Partially Committed** – The transaction has committed some of the changes it has made (these changes cannot be undone) but has more changes to make
- **Failed** – One of the statements of the transaction has failed
- **Committed** – The transaction executes all its operations successfully and changes are now persistent in the database system
- **Aborted** –The transaction has reached a failed state and all changes are rolls back

Transactions in Python

General Pattern

```
# …

connection = None
try:
    connection = psycopg2.connect(dsn)
    connection.autocommit = false
    cursor = connection.cursor()

    # execute 1st statement
    cursor.execute(statement_1)

    # execute 2nd statement
    cursor.execute(statement_2)

    if something_is_wrong:
        cursor.rollback()

    # commit the transaction
    cursor.commit()

    # close the database communication
    cursor.close()
except psycopg2.DatabaseError as error:
    print(error)
    connection.rollback()
finally:
    if connection is not None:
        connection.close()
```

If anything goes wrong, an exception is thrown and the connection is rolled back

The code can decide to rollback explicitly

If everything is ok, them commit all the changes at once

Example: Balance transfer

```
# Transfer an amount between two accounts
try:
    connection.autocommit = false
    cursor = connection.cursor()

    # Withdrawal from account A
    sql_withdrawal = “""
         UPDATE accounts
         SET balance = balance - 50
         WHERE account_num = 'A-101'"""
    cursor.execute(sql_withdrawal)

    # Deposit to account B
    sql_deposit = """
         UPDATE account
         SET balance = balance + 50
         WHERE account_num = 'A-102'"""
    cursor.execute(sql_deposit)

    #Commit changes
    connection.commit()
    print ("<p>Transfer successful<p>""")
except:
   # Handle exceptions
```

Both statements are executed, or none is executed

Example: Interest deposit

```
try:
    connection.autocommit = false
    cursor = connection.cursor()

    # deposit interest on the accounts
    for acct_num in [‘A-101’, 'A-102']
       sql_interest = """
             UPDATE accounts
             SET balance = balance * 1.01
             WHERE account_num = ‘%s’"""
    cursor.execute(sql_interest, acct_num)

    #Commit changes
    connection.commit()
    print ("<p>Transfer successful<p>""")
except:
   # Handle exceptions
```

Both statements are executed, or none is executed

Example: Multiple tables

```
def add_employee(eid, ename, managed_depart_ids):
    connection = None

    try:
        connection = psycopg2.connect(get_params())
        connection.autocommit = false
        cursor = connection.cursor()

        insert_emp_sql = """
            INSERT INTO employee(eid, ename) VALUES(%s, %s)"""
        cursor.execute(insert_emp_sql, (eid, ename))

        insert_managed_dep_sql = """
           INSERT INTO manages(eid, did) VALUES(%s, %s)"""
        for dep_id in managed_depart_ids:
            cursor.execute(insert_managed_dep_sql, (eid, dep_id))

        # commit changes
        connection.commit()
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        if connection is not None:
            connection.close()
```

Insert the employees along with all the departments they manage in the ***employee*** and ***manages*** tables

SQL Injection

SQL Injection

- SQL Injection (SQLi) is an attack wherein an attacker **executes arbitrary SQL statements** by tricking **a web application into processing an malicious input as part of an SQL statement**
- SQL injection works by:
  1. Interrupting the query; and
  2. Executing malicious code

Vulnerable Query: Example 1

user input


```
value = form.getvalue('user_input');


cursor.execute("INSERT INTO table (my_column) VALUES ('%s')",
    value);
```

Possible attack values for `user_input`

```
hello'); DROP TABLE table; --
```

⚠ The statement that is executed is:

```
INSERT INTO table(my_column) VALUES('hello'); DROP TABLE table;--')
```

Will make cause the table to be dropped

Vulnerable Query: Example 2

```
value = form.getvalue('user_input');
sql = """
      SELECT username
      FROM users
      WHERE id = %s""";
 cursor.execute(sql, value)
 result = result = cursor.fetchall()
 for row in result:
      print('<tr>')
      for value in row:
          print('<td>{}</td>'.format(value))
      print('</tr>')
```

Possible attack value for `user_input`

```
1 UNION SELECT password FROM users WHERE id=1
```

Will reveal the data of the user id = 1 displayed

Preventing SQL injection

1. Use safe (named) parameters in queries

```
value = form.getvalue('user_input');
cursor.execute("INSERT INTO table (my_column) VALUES ('%(param)s')",
    {'param': value});
```

2. Log all errors
   - Avoid sending information back to the browser
   - Prevents the attacker to get error feed-back and use trial and error

Safe and unsafe bindings

```
cursor.execute("SELECT admin FROM users WHERE username = '" +
    username + "');
cursor.execute("SELECT admin FROM users WHERE username = '%s' %
    username);
cursor.execute("SELECT admin FROM users WHERE username =
    '{}'".format(username));
cursor.execute(f"SELECT admin FROM users WHERE username =
    '{username}'");
```

Unsafe! Never do this!

```
cursor.execute("SELECT admin FROM users WHERE username = %
    (username)s", {'username': username});
```

Safe! Do this.

SQL Injection taken too far