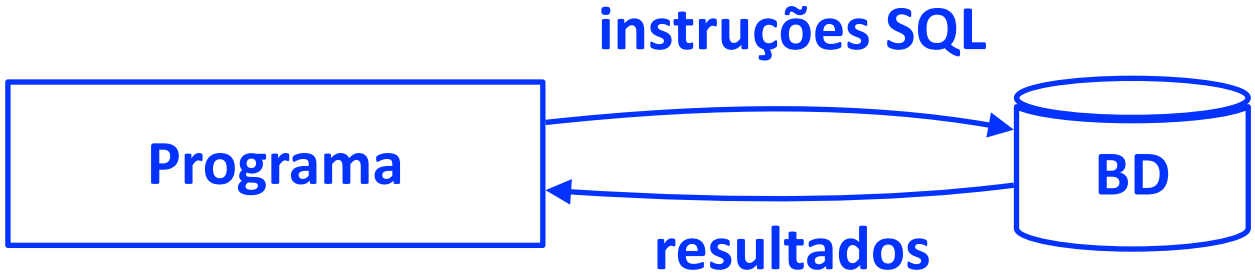
Aplicações BD com SQL embebido

Bases de Dados (CC2005)

Eduardo R. B. Marques, DCC/FCUP

Aplicações de BD com SQL embebido

- **Aplicações de BD**
  - Programas de software que de acedem a BDs.
  - Normalmente escritos numa linguagem de programação genérica (ex. Python) em que o SQL é **embebido** na lógica aplicacional em ligação a um motor de BDs.
  - Em casos mais específicos, o SQL é suportado **nativamente** numa linguagem específica.



SQL embebido

- Linguagem de programação usada é genérica. Bibliotecas/ pacotes de software provêem ao programa uma API (Application Programmer Interface):
  - 1. Estabelecer e configurar uma ligação à BD
  - 2. Subsequentemente executar comandos SQL
  - 3. Fechar a ligação à BD quando este deixa de ser necessária (ex. muitas vezes no término do programa).
- Por exemplo para acedermos à BDs SQLite em Python temos o pacote sqlite3
- Diferenças entre motores de BD são em boa medida abstraídas por standards, por exemplo ODBC, JDBC API para Java, Python PEP 249 -- Python Database API Specification v2.0

SQL embebido
Python + SQLite3

Python e SQLite3 - aspectos básicos

```
# Import sqlite3 module
import sqlite3

# Connect to database
conn = sqlite3.connect('movie_stream.db')

# Create a cursor for SQL commands
cur = conn.cursor()
cur.row_factory = sqlite3.Row
```

- Primeiros passos de um programa:
  - estabelecer ligação
  - ... e de seguida um cursor para executar comandos SQL.
- O uso de **sqlite3.Row** é útil na configuração da "row factory" do cursor para permitir aceder aos resultados da BD por nomes de colunas (ver exemplos a seguir).

Python e SQLite3 - execução de SQL

```
# Execute query
res = cur.execute(
    '''
    SELECT Title,Year
    FROM MOVIE
    WHERE Title LIKE '%star wars%'
    ORDER BY Year DESC
    ''')
# Fetch all results and use them for something
data = res.fetchall()
for row in data:
    print('Title:', row['title'],
          'Year:', row['year'])
```

- O programa fica habilitado a executar comandos SQL em interface à BD ...
- Por exemplo podemos efetuar uma interrogação e obter os seus resultado - **fetchall**() acima devolve uma lista com os resultados da interrogação

Python e SQLite3 - execução de SQL (cont.)

```
res = cur.execute(
    '''
    SELECT Title,Year
    FROM MOVIE
    WHERE Title LIKE '%star wars%'
    ORDER BY Year DESC
    ''')
data = res.fetchall()
for row in data:
   for row in data:
     print('Title:', row['title'],
           'Year:', row['year'])
```

Output do programa

```
$ python3 example1.py
Title: Star Wars: Episode VII – The Force Awakens :: Year: 2015
Title: Star Wars: Episode III – Revenge of the Sith :: Year: 2005
Title: Star Wars: Episode II – Attack of the Clones :: Year: 2002
Title: Star Wars: Episode I – The Phantom Menace :: Year: 1999
Title: Star Wars: Episode VI – Return of the Jedi :: Year: 1983
Title: Star Wars: Episode V – The Empire Strikes Back :: Year: 1980
Title: Star Wars :: Year: 1977
```

Python e SQLite3 - execução de SQL (cont.)

```
# Execute query
res = cur.execute(
    '''
    SELECT COUNT(*) n,  MIN(Year) min, MAX(Year) max
    FROM MOVIE
    WHERE Title LIKE '%star wars%'
    ''')
# Fetch single row (there's only 1!)
data = res.fetchone()
print(data['n'], data['min'], data['max'])
```

Output do programa

```
$ python3 example2.py
7 1977 2015
```

- **fetchone**(): devolve resultado único (ou próximo resultado)
- **fetchall**(): devolve uma lista com todos os resultados
- **fetchmany**(**n**): devolve uma lista com próximos n resultados da interrogação

Instruções SQL parametrizadas

```
# Execute query
year = sys.argv[1]
duration = sys.argv[2]
res = cur.execute(
    '''
    SELECT Title, Year, Duration
    FROM MOVIE
    WHERE Year = ? AND Duration  <= ?
    ORDER BY Duration
    ''', [year, duration])
for row in res.fetchall():
    print(row['Title'], row['Year'], row['Duration'])
```

- **Instruções SQL parametrizadas -** O SQL é neste caso parametrizado por dados passados pelo programa.
- A "marca" de **?** (ponto de interrogação) denota um parâmetro cujo valor é fornecido pelo programa.

Instruções SQL parametrizadas (cont.)

```
# Execute query
year = sys.argv[1]
duration = sys.argv[2]
res = cur.execute(
    '''
    SELECT Title, Year, Duration
    FROM MOVIE
    WHERE Year = ? AND Duration  <= ?
    ORDER BY Duration
    ''', [year, duration])
for row in res.fetchall():
    print(row['Title'], row['Year'], row['Duration'])
```

```
$ python3 example3.py 2001 92
Zoolander 2001 90
Rush Hour 2 2001 90
Shrek 2001 90
Monsters, Inc. 2001 92
Jurassic Park III 2001 92
```

Instruções SQL parametrizadas (cont.)

```
# Execute query
year = sys.argv[1]
duration = sys.argv[2]
res = cur.execute(
    '''
    SELECT Title, Year, Duration
    FROM MOVIE
    WHERE Year = :year AND Duration  <= :duration
    ORDER BY Duration
    ''', {'year': year, 'duration': duration})
```

- Em alternativa a **?** podemos usar **:param_name** no SQL embebido e fornecer nesse caso como parâmetro um dicionário de valores (em que as chaves identificam os parâmetros).

Alterações à BD e transações

- Podemos também ter instruções SQL que alteram os dados da BD (ex. INSERT, UPDATE, DELETE) ex.

```
cur.execute('DELETE FROM MOVIE_ACTOR WHERE ActorId=?', [id])
```

- Várias dessas instruções em bloco sequencial formam o que se chama uma **transação** tal que o efeito das instruções
  - **se torna permanente na BD** se usarmos no fim a instrução **COMMIT**
  - **é ignorado (não tem efeito)** se usarmos no fim a instrução **ROLLBACK.**
- Veremos o tópico de transações em detalhe em aulas futuras.
- **Python + sqlite3:** uma transação é iniciada implicitamente que seja executado o método **commit**() ou o método **rollback**() sobre a conexão. Veja exemplo a seguir.

```
conn.commit()   # Commits all changes

conn.rollback() # Rollback all changes
```

Alterações à BD e transações (cont.)

```
name = sys.argv[1]
res = cur.execute('SELECT ActorId FROM ACTOR WHERE Name = ?',
[name]).fetchone()
if res == None:
    print(name, ': actor not found')
    sys.exit(1)

id = res['ActorId']
res = cur.execute('DELETE FROM MOVIE_ACTOR WHERE ActorId=?', [id])
print(res.rowcount,'rows in MOVIE_ACTOR deleted')

res = cur.execute('DELETE FROM ACTOR WHERE ActorId = ?', [id]);
print(res.rowcount,'rows in ACTOR deleted')

if input('Commit  changes (Y/N)? ') == 'Y':
    conn.commit()
    print('Changes commited!')
else:
    conn.rollback()
    print('Changes rolled back!')
```

Alterações à BD e transações (cont.)

```
$ python3 example5.py 'Tom Cruise'
22 rows in MOVIE_ACTOR deleted
1 rows in ACTOR deleted
Commit  changes (Y/N)? N
Changes rolled back!

$ python3 example5.py 'Tom Cruise'
22 rows in MOVIE_ACTOR deleted
1 rows in ACTOR deleted
Commit  changes (Y/N)? Y
Changes commited!

$ python3 example5.py 'Tom Cruise'
Tom Cruise : actor not found
```

Dados a remover existem mas mudanças não são "commited"

Dados (ainda existem e) são desta vez removidos.

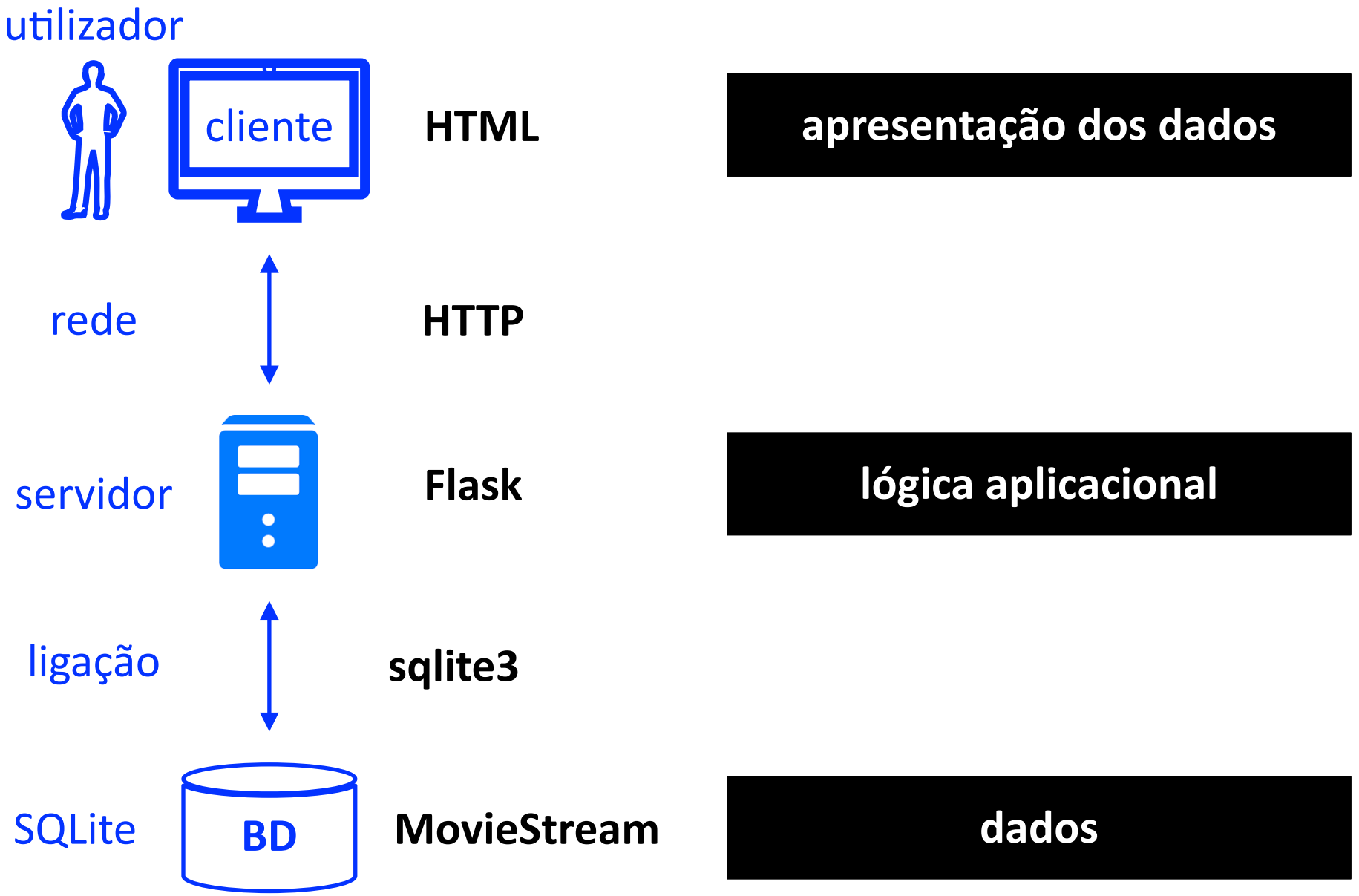
Dados já não existem.

Aplicação exemplo

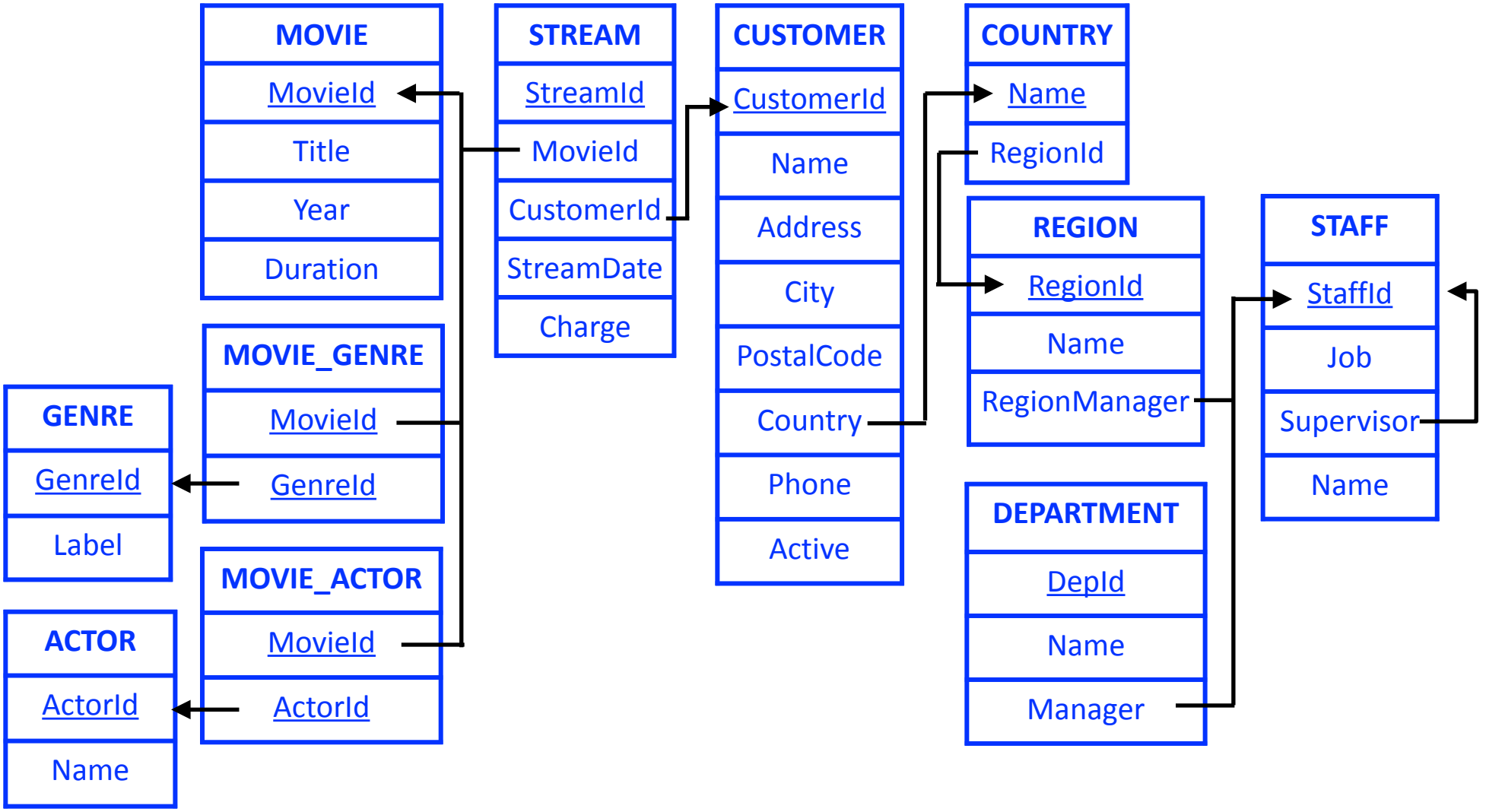
Aplicação exemplo

- Base de dados MovieStream relativa a filmes catalogados no IMDB (detalhes a seguir)
- A aplicação exemplo é desenvolvida em Python e emprega as biblioteca Flask: para desenvolvimento de aplicações Web, em conjunto com sistema de "templates" Jinja.

Aplicação exemplo (cont.)



Esquema da BD



Acesso à BD — exemplo 1

app.py

```
@APP.route('/movies/')
def list_movies():
    movies = db.execute(
      '''
      SELECT MovieId, Title, Year, Duration FROM MOVIE
      ORDER BY MovieId
      ''').fetchall()
    return render_template('movie-list.html', movies=movies)
```

- SQL é embebido na aplicação por forma a comunicar com a BD.
- Neste exemplo é feita uma consulta para obter os dados de todos os filmes [via `db.execute()`, definido em `db.py`]
- Seguidamente é retornado à aplicação cliente o HTML gerado a partir de uma template Jinja (ilustrada a seguir) [via `render_template`, definido pela biblioteca Flask].

Acesso à BD — exemplo 1 (cont.)

```
...
{% for movie in movies %}
  <tr>
    <td>
      <a href="/movies/{{ movie.MovieId }}">{{ movie.MovieId }}</a>
    </td>
    <td>{{ movie.Title }}</td>
    <td>{{ movie.Year }}</td>
    <td>{{ movie.Duration }}</td>
  </tr>
{% endfor %}
...
```

- Template Jinja para geração de HTML.
- A negrito **anotações Jinja** que permitem fazer ligação com os dados.

Acesso à BD — exemplo 1 (cont.)

/movies

```
<tr>
  <td><a href="/movies/1">1</a></td>
  <td>The Hobbit: An Unexpected Journey</td>
  <td>2012</td>
  <td>169</td>
</tr>
```

Id	Title	Year	Duration (m)
1	The Hobbit: An Unexpected Journey	2012	169
2	Night at the Museum	2006	108
3	21	2008	123
4	Mission: Impossible - Ghost Protocol	2011	132
5	Sully	2016	96
6	Resident Evil	2002	100

- Fragmento da página visualizada no "browser" cliente para o HTML gerado (endpoint "/movies"). Pode ver o seu HTML fonte no "browser" (ex. "Show Page Source" no Firefox)

Acesso à BD — exemplo 2

app.py

```
@APP.route('/movies/<int:id>/')
def get_movie(id):
  movie = db.execute(
      '''
      SELECT MovieId, Title, Year, Duration
      FROM MOVIE WHERE movieId = ?
      ''',
      [id]
    ).fetchone()
...
```

- Neste caso consultamos os dados de um filme em particular. A consulta à BD leva em conta o identificador **id** de um filme passado pela aplicação cliente.

Acesso à BD — exemplo 2

Movie info

`/movies/151`

Id: 151

Title: Pulp Fiction

Year: 1994

Duration: 154 minutes

Genres:

- Crime
- Drama

Actors:

- Bruce Willis
- John Travolta
- Samuel L. Jackson
- Uma Thurman

Streamed:

- 2018-11-14 11:30:00
- 2018-07-27 15:20:00
- 2018-06-30 02:54:00

Injeção de SQL

Exemplo 1

```
year = sys.argv[1]
res = cur.execute(
    '''
    SELECT Title, Year, Duration
    FROM MOVIE
    WHERE YEAR =''' + year + ' ORDER BY Title')
for row in res.fetchall():
    print(row['Title'], row['Year'], row['Duration'])
```

- Não é usada uma instrução parametrizada. Ao invés o valor de **`year`** é usado para definir dinamicamente o código SQL a executar. Este valor é arbitrário dado não ser sujeito a qualquer validação ou filtragem.
- O programa pode executar "normalmente" ... mas também de forma "inesperada" já que o código SQL pode ser adulterado.

Exemplo 1 (cont.)

Execução "normal"

```
$ python3 injection_example1.py 1981
Das Boot 1981 149
Raiders of the Lost Ark 1981 115
The Evil Dead 1981 85
```

Execução "inesperada"

```
$ python3 injection_example1.py '1981 UNION SELECT
CustomerId,Name,Country FROM CUSTOMER  --'
1 Mary Smith Japan
2 Patricia Johnson United States
3 Linda Williams Greece
4 Barbara Jones Myanmar
5 Elizabeth Brown Taiwan
. . .
Das Boot 1981 149
Raiders of the Lost Ark 1981 115
The Evil Dead 1981 85
```

DADOS DA TABELA CUSTOMER!

Exemplo 1 (cont.)

```
year = sys.argv[1]
res = cur.execute(
    '''
    SELECT Title, Year, Duration
    FROM MOVIE
    WHERE YEAR =''' + year + ' ORDER BY Title')
for row in res.fetchall():
    print(row['Title'], row['Year'], row['Duration'])
```

- Com **year = '1981 UNION SELECT CustomerId,Name,Country FROM CUSTOMER --'** programa o código SQL executado é ...

```
SELECT Title, Year, Duration
    FROM MOVIE
    WHERE YEAR = 1981
UNION SELECT CustomerId, Name, Country FROM CUSTOMER-- ORDER BY Title'
```

- Nota: **ORDER BY Title** é ignorado já que **--** inicia um comentário até ao fim da linha

Exemplo 2

```
id = sys.argv[1]
res = cur.execute('DELETE FROM STREAM WHERE StreamId = ' + id);
print(res.rowcount, 'rows deleted')
```

- Podemos adulterar o código SQL executado por forma a eliminar todos os registos da tabela em causa em vez de apenas um! Por exemplo se **id = '1 OR TRUE'** o código executado torna-se

```
DELETE FROM STREAM WHERE StreamId = 1 OR TRUE
```

Execução "normal"

```
$ python3 injection_example2.py 1
Before DELETE: 10162 rows
1 rows deleted
After DELETE: 10161 rows
```

Execução "inesperada"

```
$ python3 injection_example2.py '1 OR TRUE'
Before DELETE: 10162 rows
10162 rows deleted
After DELETE: 0 rows
```

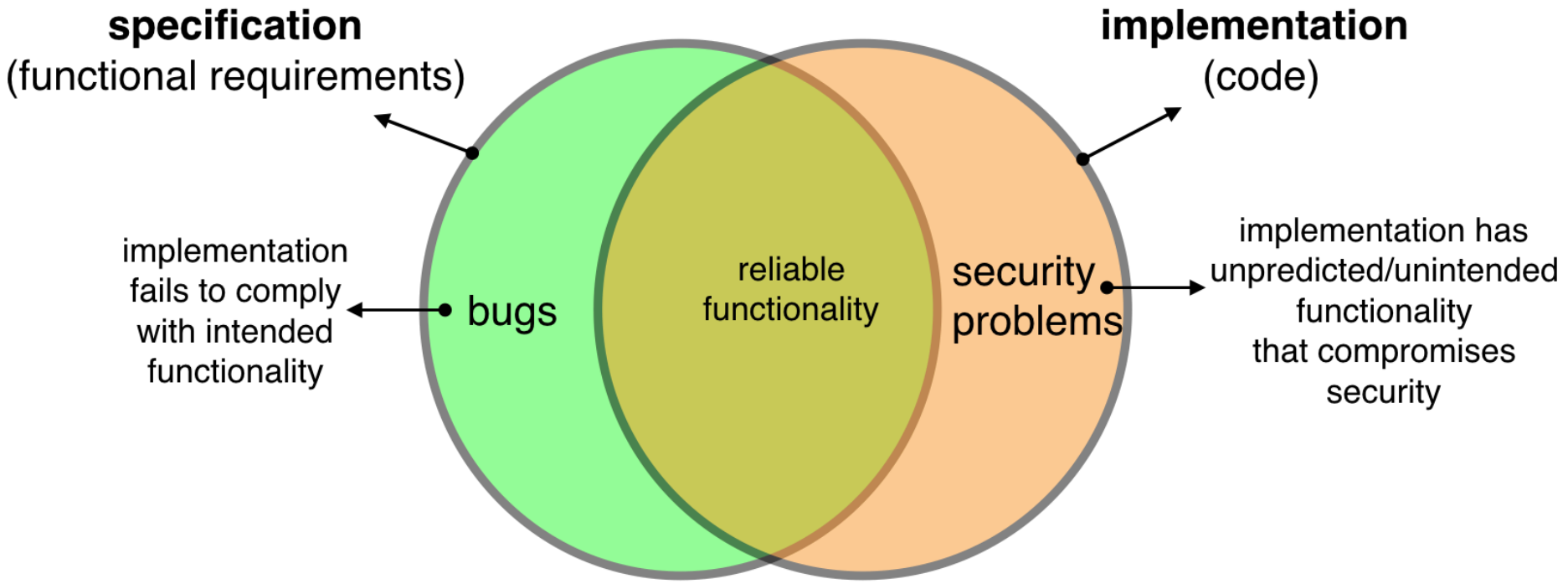
Injeção de SQL — quão relevante é o problema?

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function

- "SQL injection" aparece em 1º lugar na lista CWE/SANS Top 25 Most Dangerous Software Errors
- A CWE é o registo de referência para classes de vulnerabilidade de segurança.

Contexto geral — segurança vs fiabilidade

(Imagem: Eduardo Marques, Questões de Segurança em Eng. Software, Mestrado em Segurança Informática)

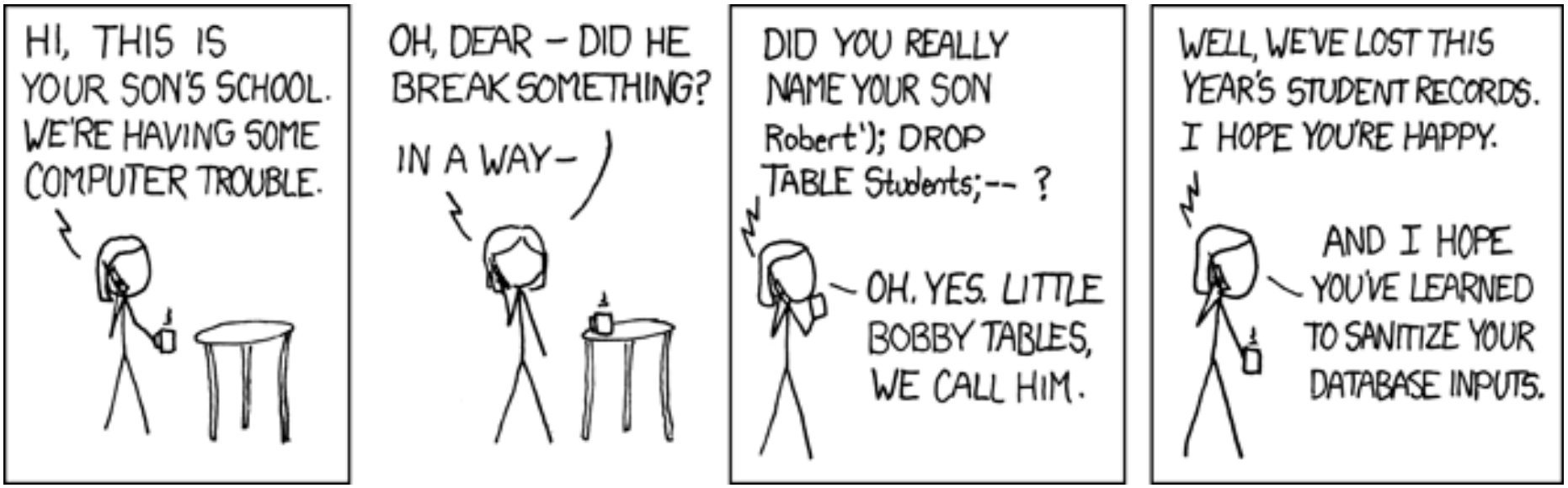


- ***"Reliable software does what it is supposed to do. Secure software does what is supposed to do, and nothing else."*** — Ivan Arce

Contexto de segurança (cont.)

- Essência do problema em injeção de SQL
  - Distinção dados vs. código fica difusa.
  - ... uma causa comum a outras classes de vulnerabilidade.
- Conselhos gerais
  - Tratar os dados de input a uma aplicação como não confiáveis por omissão ("**Trust no input**").
  - Possíveis estratégias: sanitisação, filtragem, ...
- Conselhos específicos para injeção SQL
  - **Usar instruções SQL parametrizadas** em vez de deixar que input não confiável afecte o código executado em interface à BD.
  - A geração "dinâmica" de SQL pode na esmagadora maioria das vezes ser evitada.

Outros apontadores



"Exploits of a mom" — https://xkcd.com/327/

- Mais informação
  - https://bobby-tables.com/
  - SQL injection hall of shame
  - The History of SQL Injection, the Hack That Will Never Go Away
- Humor
  - Did Little Bobby Tables migrate to Sweden?
  - ; DROP TABLE "COMPANIES";-- LTD

Outras formas de SQL embebido

SQL embebido em extensões de linguagens

- Na aplicação exemplo o SQL é embebido de forma dinâmica.
  - Código SQL é definido em tempo de execução pela aplicação. Esta é a forma mais comum de definição de SQL embebido.
  - Standards como JDBC para Java ou PEP 249 para Python facilitam o interface a SGBDs diferentes usando o mesmo código.
- Outra aproximação baseia-se na **extensão de linguagens** (ex. C, Java, ...) para a inclusão directa de SQL (SQL embebido "puro"):
  - A validação do SQL é feita em tempo de compilação, o que pode tornar mais fiável a detecção de erros e evitar problemas de segurança.
  - **Exemplos**: Pro*C/C++ (Oracle), ECPG (C/PostgreSQL), SQLJ (Java / Oracle; entretanto descontinuado).
  - Estas linguagens estão associadas a SGBDs específicos.

Exemplo — Oracle Pro*C

```
EXEC SQL BEGIN DECLARE SECTION;
...
int    empno;                 /* employee number  */
VARCHAR ename[10 + 1];  /* employee name      */
int    deptno;                /* department number  */
VARCHAR dname[14 + 1];  /* department name */
VARCHAR job[9 + 1];        /* employee job */
int    sal;                   /* employee salary */
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT DNAME
INTO :dname
FROM DEPT
WHERE DEPTNO = :deptno;
 dname.arr[dname.len] = '\0';
...
EXEC SQL INSERT INTO EMP(EMPNO,ENAME,JOB,HIREDATE,SAL,DEPTNO)
VALUES (:empno,:ename,:job,sysdate,:sal,:deptno);

printf("\n%s added to the %s department as employee number %d\n",
       ename.arr,dname.arr,empno);
```

- Fragmento de addempPROC.pc , parte do conjunto Pro*C sample programs, Oracle.
- "Pré-compilador" transforma código Pro*C em C.
- Código SQL embebido em directivas **EXEC SQL**. Variáveis C são usadas em associação a instruções SQL usando a notação **:var**.
- Outras linguagens como ECPG e SQLJ têm uma aproximação similar.