David Mugnai
Coffee drinker

Niente bug se ti affidi all'elefante

Develer Techlabs

15/09/2020

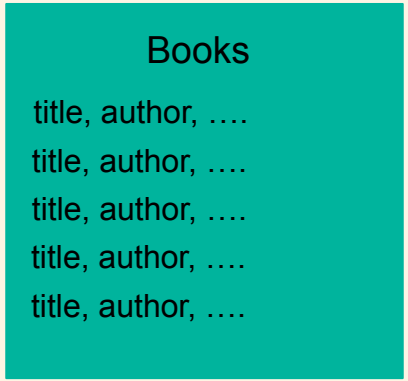
develer 20

Di cosa parleremo in questo talk?

- Postgres e Row Level Security
- Sarà un talk più pratico che teorico
- Due scenari diversi
- Useremo le RLS per implementare le feature richieste

Scenario1
“quello facile”

Scenario1 - La situazione prima del disastro



Scenario1 - Il disastro

Books

title, author, deleted, ….
title, author, deleted, ….
title, author, deleted, ….
title, author, deleted, ….
title, author, deleted, ….

```
select *
from books
where
  deleted=false
```

```
select title
from books
where author=?
 and deleted=false
```

```
select sum(price)
from books
where date < now()
 and deleted=false
```

Perché un disastro?

Bugs, bugs everywhere

La probabilità di introdurre un bug è:

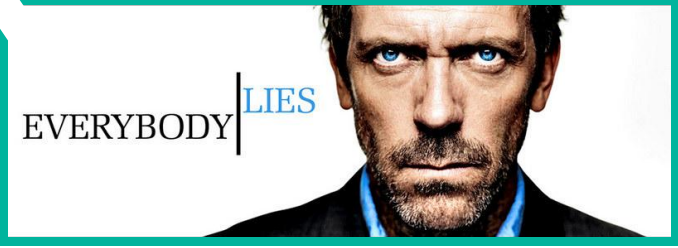
proporzionale al numero di linee di codice

inversamente proporzionale al tempo concesso per implementare la feature

NOTA: i vecchi test (se esistono) non aiutano

RLS to the rescue!

- ROW LEVEL SECURITY
- Usabili da PostgreSQL ~~9.5~~ 10
- Evitiamo il bug **senza scrivere codice**



Scenario1 - Books table

```
dvd@localhost/dvd> \d books

               Table "public.books"
Column  |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
title   | text    |           |          |
author  | text    |           |          |
visible | boolean |           |          |
```

Scenario1 - Programma di esempio

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

with psycopg2.connect(os.environ["DATABASE_URL"]) as conn:
    with conn.cursor() as cur:
        cur.execute("SELECT title, author, visible FROM books ORDER BY title")
        for (ix, (title, author, visible)) in enumerate(cur.fetchall()):
            print("{:2} - {:50} {:20} {}".format(ix, title, author, visible))
```

Scenario1 - Prima della cura

```
$ DATABASE_URL=postgresql://dvd@/dvd python3 fetch.py

 0 - Dune                                          Frank Herbert       True
 1 - Dune (draft)                                  Frank Herbert       False
 2 - The Hitchhiker's Guide to the Galaxy          Douglas Adams       True
 3 - The Lord of the Rings                         J. R. R. Tolkien    True
```

Scenario1 - Dopo la cura

```
$ DATABASE_URL=postgresql://dbuser:pwd@localhost/dvd python3 fetch.py

 0 - Dune                                                Frank Herbert       True
 1 - The Hitchhiker's Guide to the Galaxy                Douglas Adams       True
 2 - The Lord of the Rings                               J. R. R. Tolkien    True
```

Cos’è la Row Level Security?

La RLS è una feature (**SQL**) che permette un controllo preciso e puntuale su quali righe di una tabella sono accessibili.

Si attiva definendo una o più **POLICY**

Cos’è una POLICY?

- Una policy conferisce il permesso di selezionare, inserire, aggiornare o cancellare le righe che soddisfano una certa condizione.
- Le policy hanno un nome univoco per tabella.
- Le policy possono essere associate ad un comando, ad un ruolo o ad entrambi.
- Per specificare quali righe sono visibili o modificabili, è necessaria un'espressione che restituisca un risultato booleano.

Scenario1 - La cura (come si usano le Row Level Security)

```
CREATE ROLE dbuser WITH LOGIN ENCRYPTED PASSWORD 'pwd';
ALTER TABLE books OWNER TO dbuser;
CREATE POLICY only_visible_rows ON books USING (visible=true);
ALTER TABLE books ENABLE ROW LEVEL SECURITY;
ALTER TABLE books FORCE ROW LEVEL SECURITY;
                Table "public.books"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 title   | text    |           |          |
 author  | text    |           |          |
 visible | boolean |           |          |
Policies (forced row security enabled):
    POLICY "only_visible_rows"
      USING ((visible = true))
```

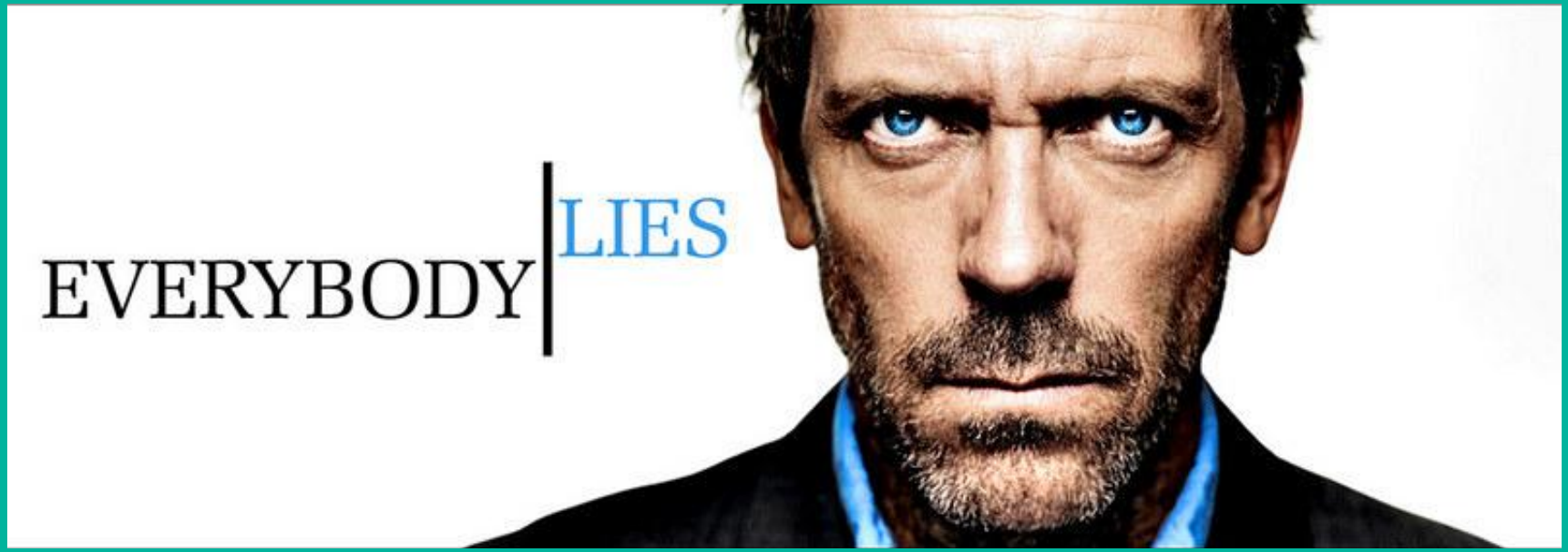
Explain me analyzer!

Ok, ma va veloce?

```
dvd@[local]/dvd> ALTER TABLE books ADD COLUMN date TIMESTAMP NOT NULL DEFAULT now();
ALTER TABLE

dvd@[local]/dvd> CREATE INDEX ix_books_date ON books(date);
CREATE INDEX

dvd@[local]/dvd> INSERT INTO books
   SELECT 'Books #' || n, 'Postgres', random() < 0.5, t
   FROM (
       SELECT row_number() OVER() AS n, t
       FROM generate_series('2021-01-01 00:00:00'::timestamp, '2021-07-01 00:00:00', '1 minute')
AS t
   ) serie;
INSERT 0 260641
```

```
dvd@[local]/dvd> SELECT visible, count(*) FROM books GROUP BY 1;
visible | count
---------+--------
f       | 130367
t       | 130278

dvd@[local]/dvd> \c postgresql://dbuser:pwd@localhost/dvd

dbuser@[local]/dvd> SELECT visible, count(*) FROM books GROUP BY 1;
visible | count
---------+--------
t       | 130278
```

```
dbuser@localhost/dvd> EXPLAIN ANALYZE
  SELECT * FROM BOOKS WHERE DATE BETWEEN '2021-02-14' AND '2021-02-15';
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
Index Scan using ix_books_date on books  (cost=0.42..60.14 rows=699 width=31) (actual
time=0.033..0.545 rows=753 loops=1)
  Index Cond: ((date >= '2021-02-14 00:00:00'::timestamp without time zone) AND (date <=
'2021-02-15 00:00:00'::timestamp without time zone))
  Filter: visible
  Rows Removed by Filter: 688
Planning Time: 0.090 ms
Execution Time: 0.609 ms
(6 rows)
```


EVERYBODY LIES

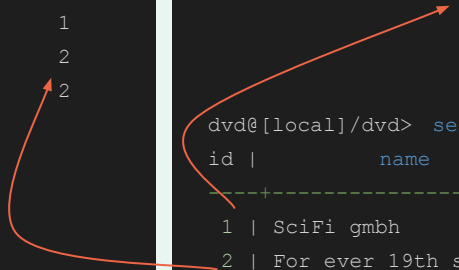
Scenario 2
“quello non-semplice”

Scenario2 - Il database

```
dvd@[local]/dvd>  select * from books2;
                title                  |        author        | customer_id
---------------------------------------+----------------------+------------
-
The Lord of the Rings                  | J. R. R. Tolkien     |           1
The Hitchhiker's Guide to the Galaxy   | Douglas Adams        |           1
Dune                                   | Frank Herbert        |           1
War and Peace                          | Leo Tolstoy          |           2
A Study in Scarlet                     | Arthur Conan Doyle   |           2
```

```
dvd@[local]/dvd>  select * from users ;
username | customer_id
----------+-------------
dvd      |           1
l        |           1
jmp      |           2
```

```
dvd@[local]/dvd>  select * from customers;
id |          name
---+-----------------------
 1 | SciFi gmbh
 2 | For ever 19th s.r.l.
```



Scenario2 - Il database

```
dvd@[local]/dvd> \d books2
                Table "public.books2"
  Column     |  Type   | Collation | Nullable | Default
-------------+---------+-----------+----------+---------
title        | text    |           | not null |
author       | text    |           | not null |
customer_id | integer |           | not null |
Foreign-key constraints:
    "books2_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customers(id)
Policies (row security disabled):
    POLICY "same_customer_policy"
      USING (((current_setting('app.customer_id'::text))::integer = customer_id))
      WITH CHECK (((current_setting('app.customer_id'::text))::integer = customer_id))
```

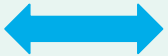
Scenario2 - Il database

```
CREATE POLICY same_customer_policy ON books2
   USING (current_setting('app.customer_id')::int = customer_id)
   WITH CHECK (current_setting('app.customer_id')::int = customer_id);
```

Un valore impostato dall'applicazione con lo scope limitato **alla transazione corrente**

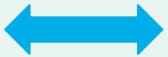
Scenario2 - Il codice

```
function read_handler(user):
  tx = open_tx()
  … use tx
```

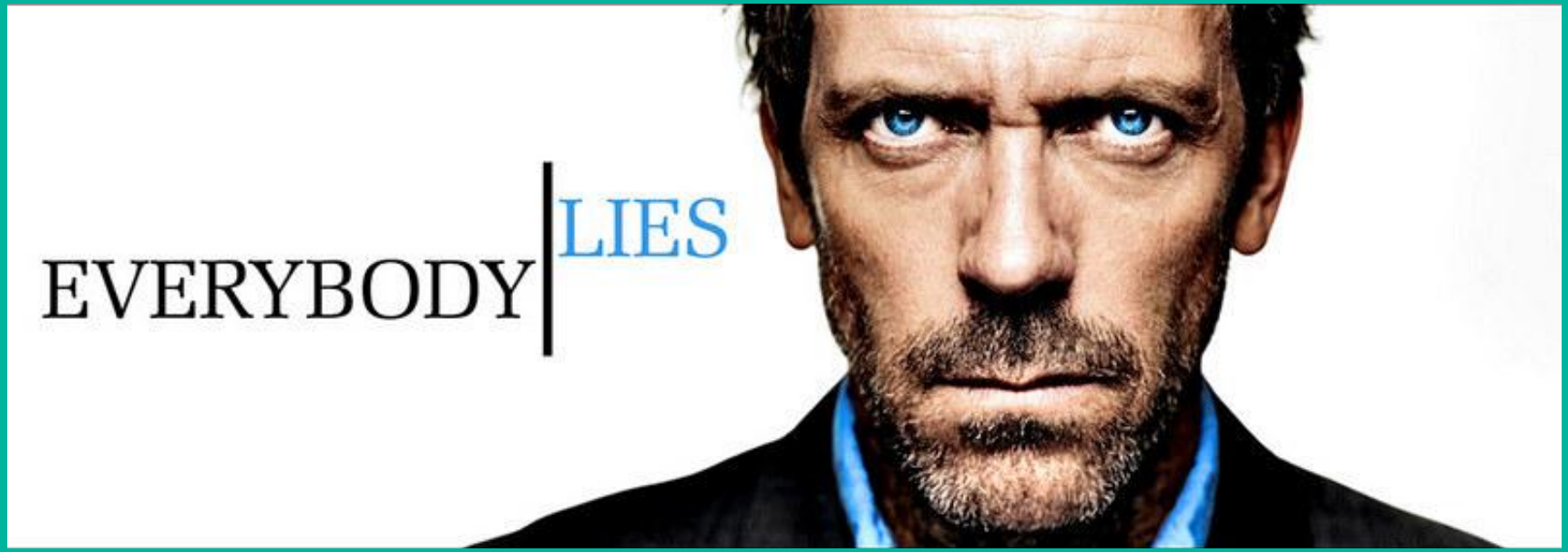


```
function open_tx():
  conn = … // maybe ask the pool?
  tx = conn.execute("BEGIN")
  return tx
```

```
function read_handler(user):
  tx = open_tx(user)
  … use tx
```



```
function open_tx(user):
  conn = … // maybe ask the pool?
  tx = conn.execute("BEGIN")
  tx.execute(
    "SET LOCAL app.customer_id = " + user.customer_id)
  return tx
```



Ma alcune bugie sono a fin di bene :)

Abbiamo trasformato un problema “globale” in uno “locale”

Variabili di sessione

```
SELECT current_setting('app.customer_id')::text);
```

Attenzione! “current_settings” termina con un errore se la variabile non esiste (💃💃💃), ma solo fino a quando una transazione non la imposta, dopodiché restituisce una stringa vuota (💀💀💀).

Tips & Tricks

Usare almeno due utenti

Ovviamente vi serve un utente soggetto alle RLS perché il gioco funzioni.

Ma avere anche un utente **che può ignorare** le RLS è molto comodo per debug.

Tips & Tricks

Scrivere policy noiose

Trattenetevi dall'inserire subquery nelle policy, o da metterne troppe in cascate (magari mischiando per bene AND / OR)

Il pericolo, come a tutte le astrazioni di altissimo livello, è di perdere il contatto con la realtà (👏👏 performance, 👏👏 debug)

Tips & Tricks

Verificate i permessi dell'utente a runtime

Un piccolo check all'avvio dell'applicazione può risparmiarvi un sacco di problemi!

Terminiamo (o al limite un bel log) se l'utente **ha troppi permessi!**

David Mugnai

dvd@develer.com

Vuoi rimanere aggiornato sugli eventi Develer?
Seguici nei nostri canali social:

develer | 20

www.develer.com