Psycopg, and...

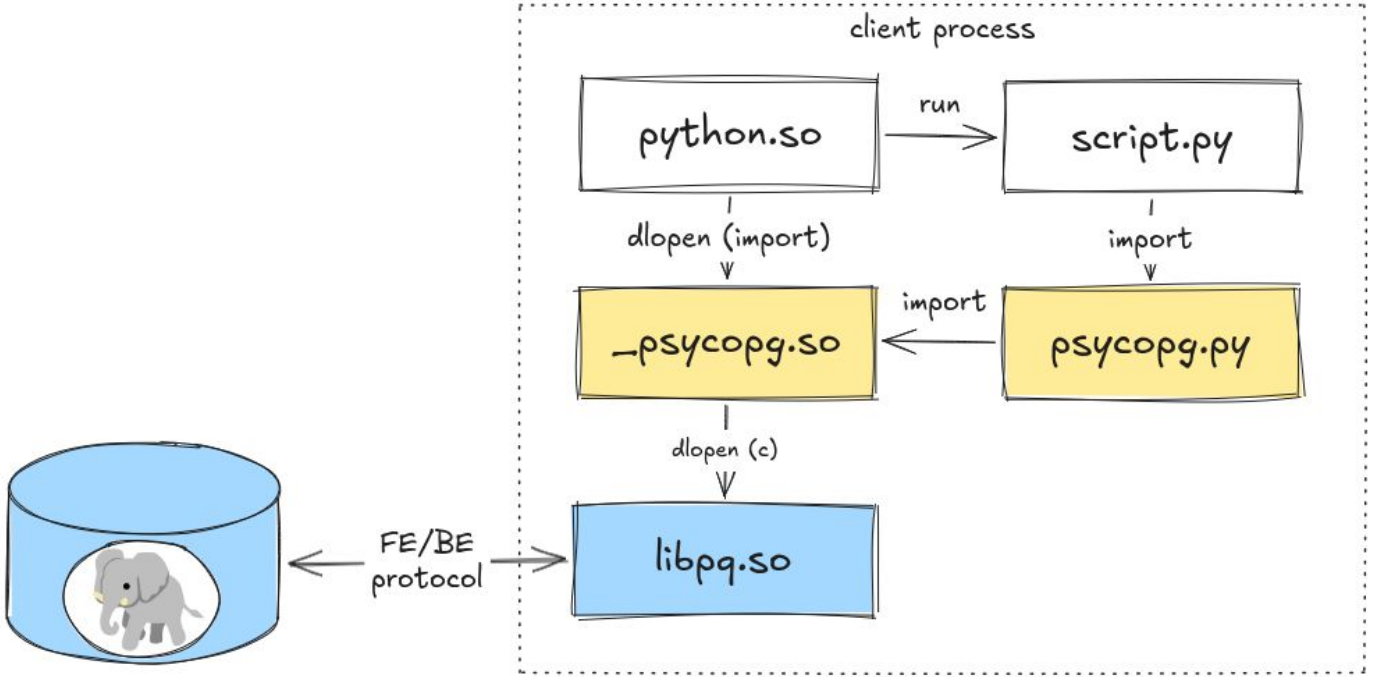
...20 years of mostly friendly *coexistence with the libpq*

`'2026-05-20'::date`

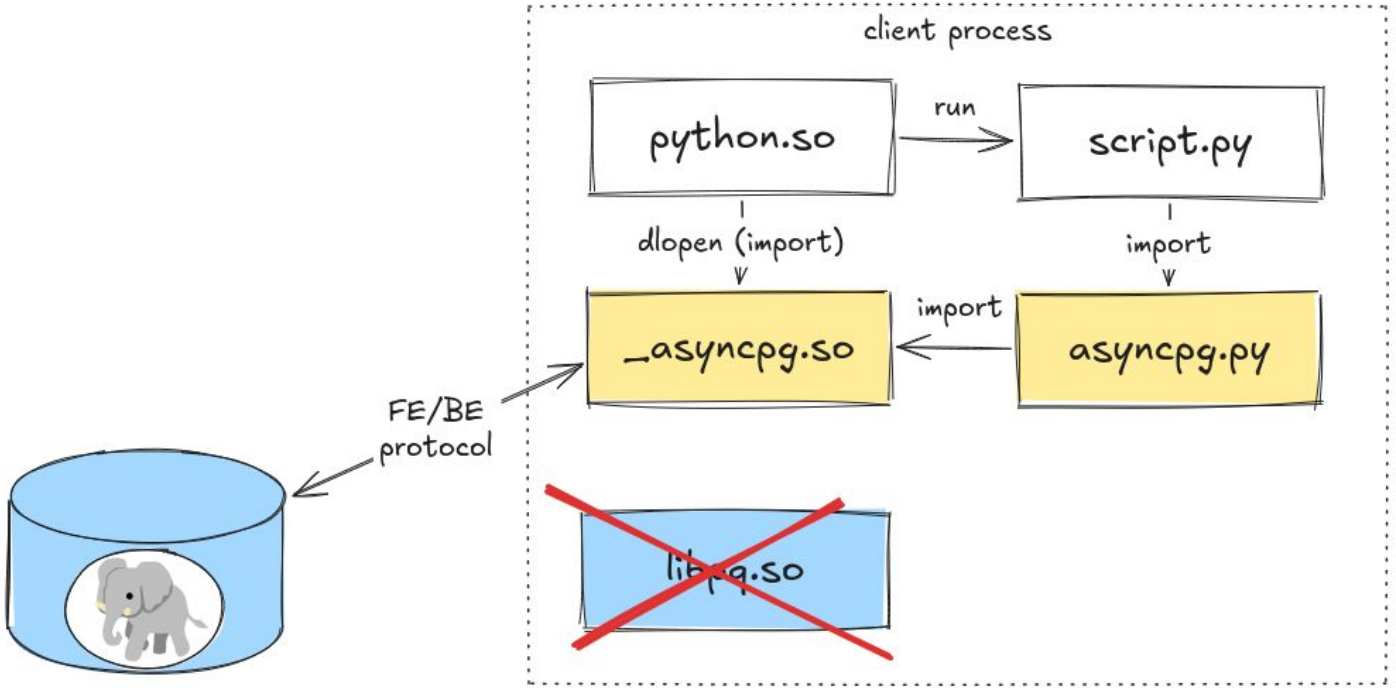
whoami

- Daniele – "piro"
- Python since 2003 (2.2)
- Postgres since 2003 (7.4)
- Contributing to Psycopg since 2006 (2.0 alpha)
- Maintaining psycopg2 since 2010 (2.2)
- Designed and developed Psycopg 3 in 2020-21

What is Psycopg — behind the scene

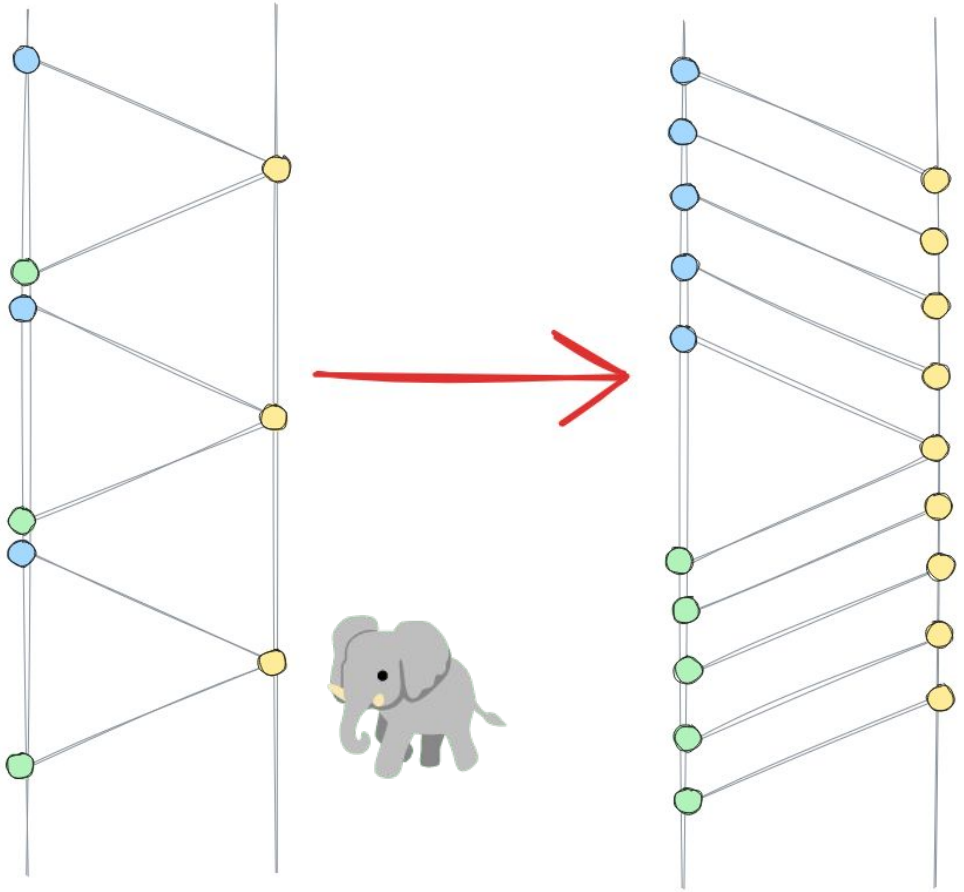


What is Psycopg — behind the scene



Some libpq stories

Pipeline mode



Pipeline mode

- Capability of the BE-FE protocol since v3
- Available since PostgreSQL 7.4 (2003)
- Not exposed by libpq until PostgreSQL 14 (2021)
  - 14 - 7.4 = 17 major versions!
- Used by Psycopg since 3.1 (August 2022)

Pipe

```
piro@sherekhan:~/dev/fs/postgres$ git log acb7e4eb6b1c6^!
commit acb7e4eb6b1c614c68a62fb3a6a5bba1af0a2659
Author: Álvaro Herrera <alvherre@alvh.no-ip.org>
Date:   Mon Mar 15 18:13:42 2021 -0300

    Implement pipeline mode in libpq

    Pipeline mode in libpq lets an application avoid the Sync messages in
    the FE/BE protocol that are implicit in the old libpq API after each
    query.  The application can then insert Sync at its leisure with a new
    libpq function PQpipelineSync.  This can lead to substantial reductions
    in query latency.

    Co-authored-by: Craig Ringer <craig.ringer@enterprisedb.com>
    Co-authored-by: Matthieu Garrigues <matthieu.garrigues@gmail.com>
    Co-authored-by: Álvaro Herrera <alvherre@alvh.no-ip.org>
    Reviewed-by: Andres Freund <andres@anarazel.de>
    Reviewed-by: Aya Iwata <iwata.aya@jp.fujitsu.com>
    Reviewed-by: Daniel Vérité <daniel@manitou-mail.org>
    Reviewed-by: David G. Johnston <david.g.johnston@gmail.com>
    Reviewed-by: Justin Pryzby <pryzby@telsasoft.com>
    Reviewed-by: Kirk Jamison <k.jamison@fujitsu.com>
    Reviewed-by: Michael Paquier <michael.paquier@gmail.com>
    Reviewed-by: Nikhil Sontakke <nikhils@2ndquadrant.com>
    Reviewed-by: Vaishnavi Prabakaran <VaishnaviP@fast.au.fujitsu.com>
    Reviewed-by: Zhihong Yu <zyu@yugabyte.com>
```

Pipeline mode

- Systems using the BE-FE protocol **might** have used it
  - some did, some didn't, some don't
- Pretty hard problem — libpq or not! For Psycopg:
  - 2 devs (Denis Laxalde [Dalibo], me)
  - 3 major redesigns
  - 4-6 months of work

Pipeline mode

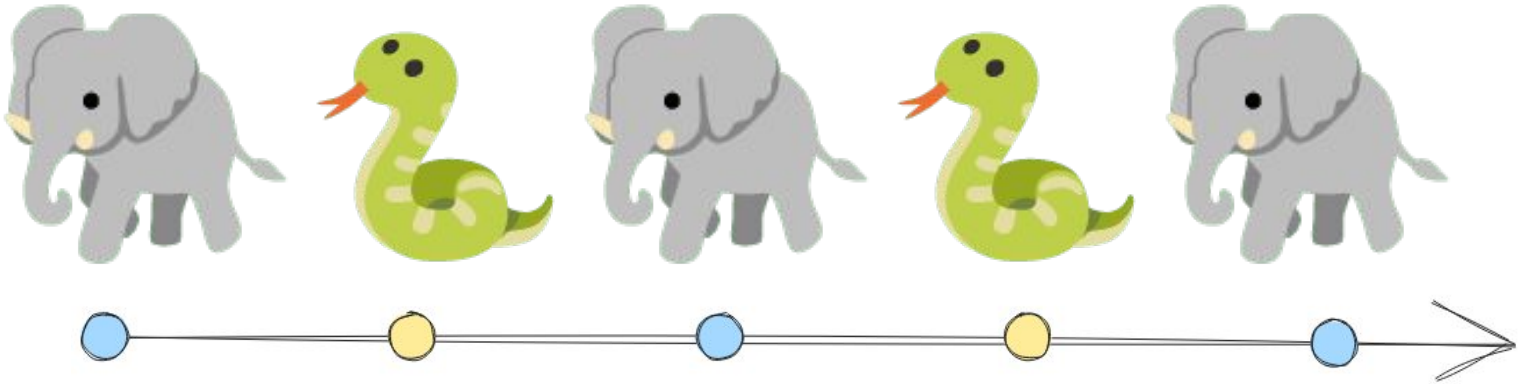
- Systems using the BE-FE protocol **might** have used it
  - some did, some didn't, some don't
- Pretty hard problem — libpq or not! For Psycopg:
  - 2 devs (Denis Laxalde [Dalibo], me)
  - 3 major re
  - 4-6 month

```
with conn.pipeline():
    cur.execute("QUERY 1")
    with conn.transaction():
        cur.executemany("QUERY 2", args_list)
        row = cur.execute("QUERY 3").fetchone()
```

libpq has shaped the landscape

- Python, Perl, Ruby obviously affected (libpq users)
- Npgsql (C#): explicit `NpgsqlBatch` API, since 2021
- pgx (Go): `StartPipeline()`, since August 2022
- pgjdbc (Java): only batch INSERT
- tokyo-postgres (Rust): since 2019
  - implicit in async concurrent model

Replication



Physical, logical replication

- `COPY_BOTH` messages at protocol level
  - `PQgetCopyData()` to receive data
  - `PQputCopyData()` to return keepalives
- Replication messages (XLogData) are standard
  - Same envelope for physical, logical
  - Yet low level protocol access is required

Physical, logical replication

- `COPY_BOTH` messages at protocol level
  - `PQgetCopyData()` to receive data
  - `PQputCopyData()` to return keepalives
- Rep
  -
  -

```python
def confirm(self, msg: ReplicationMessage[Any]) -> None:
    """Acknowledge *msg* as processed, flushed, and applied."""
    lsn_int = int(msg.lsn)
    self._conn.wait(
        _repl_write(
            self._pgconn,
            struct.pack(">cQQQQB", b"r", lsn_int, lsn_int, lsn_int, 0, 0),
        )
    )
```

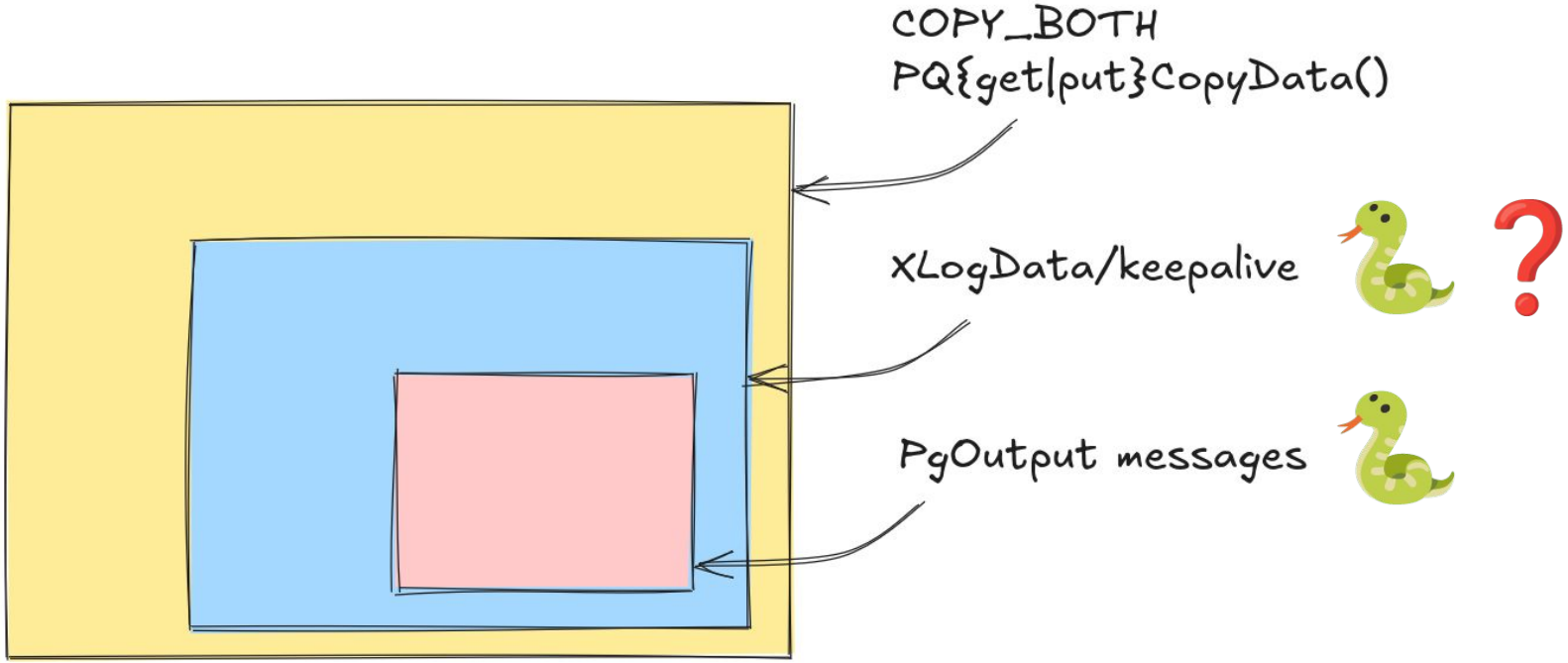
Some policies are best practice

- How often to send feedback
  - “Not at every query” seems the consensus
    - pg_recvlogical does it
    - psycopg2 introduced it as bugfix
  - But also not get disconnected
- Maybe the libpq is the right place for such policy?

Logical replication decoding

- One built-in plugin: **pgoutput**
  - Complete, well maintained, evolving
  - But no library decoder
    - No reference implementation

Logical replication decoding



replication decoding in Psycopg

- psycopg2: low level `consume_stream()`, callback based
  - Thank you Alexander Kukushkin!
  - No pgoutput parsing
  - Replisome built on top a souped-up wal2json
- Psycopg ???: high level logical replication interface
  - iterator based
  - auto keepalives
  - pgoutput parsing to Python data

replication decoding in Psycopg

- psycopg2: low level `consume_stream()`, callback based
  -
  -
  -
- Psy
  -
  -
  -

```python
# A replication connection
with psycopg.connect(dsn, replication="database", autocommit=True) as repl_conn:

    # A Replication to consume the stream, a Plugin to decode it.
    plugin = PgoutputPlugin("my_publication")
    with Replication(repl_conn, slot="my_slot", plugin=plugin) as repl:
        for msg in repl:
            if isinstance(msg.decoded, InsertMessage):
                print(msg.decoded.new_row)
                break
```

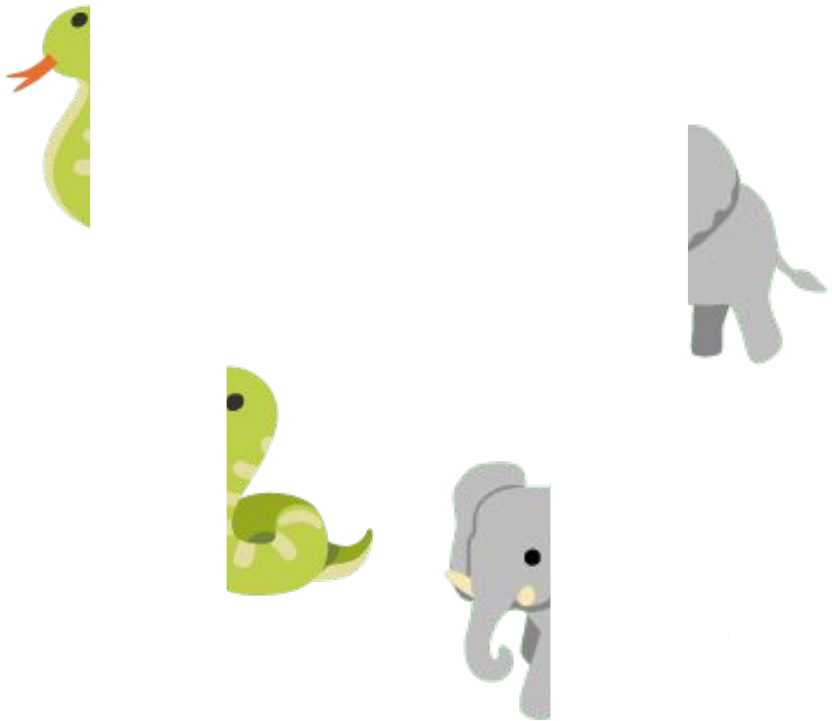
Replication ecosystem

- Java, C#: high abstraction API (no libpq based)
- Ruby, psycopg2: raw bytes only
- Go, Ruby, Node, Php: no support or external projects
  - Every project has to reimplement pgoutput or write their plugin

Looking for a pattern

- If there is no client for a library feature, it may stagnate
- The Postgres `/bin` directory is an important client
  - No program use pipeline mode
  - No client for replication

Async, you said?



Why async operations

- Not all runtimes support blocking+threads for concurrency
  - Erlang processes
  - Goroutines
  - Python green threads
- Libpq supports async modes too
  - Connection
  - Query execution

Async connection in libpq

- Libpq connection function come in sync/async flavour
  - `PQconnectdb()` (blocked by `poll()`, `select()`...)
  - `PQconnectStart()`/`PQconnectPoll()` (and you block)
- But they are almost the same
  - Except for...

Async connection in libpq

- Libpq connection function come in sync/async flavour
  - `PQconnectdb()` (blocked by `poll()`, `select()`...)
  - `PQconnectStart()`/`PQconnectPoll()` (and you block)
- But they are almost the same
  - Excep

> The `connect_timeout` connection parameter is ignored when using `PQconnectPoll`; it is the application's responsibility to decide whether an excessive amount of time has elapsed. Otherwise, `PQconnectStart` followed by a `PQconnectPoll` loop is equivalent to `PQconnectdb`.

Async connection in libpq

- Easy enough!
  - Just `PQconninfoParse()`, check for `connect_timeout`
- But...
  - `"host=db1.example.com,db2.example.com"`
- The first host has to time out before to try the second

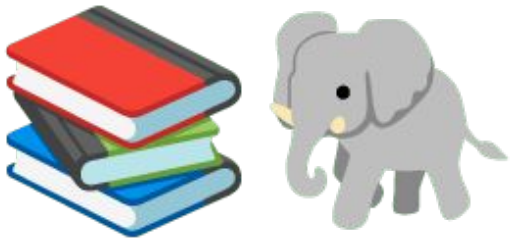
Async connection in libpq

- Connection-string parsing
- `PG*` env vars fallback
- Service file parsing
  - `pg_service.conf`, with the right precedence rules for `PGSERVICEFILE` vs `~/.pg_service.conf` vs system path
- DNS resolution to multiple IPs
- Looping over (host, IP, port) tuples with per-attempt timeout
- `target_session_attrs=prefer-standby` check
- `load_balance_hosts=random` randomization
  - Is DNS randomised too by the way?
- All consistent with libpq's sync behavior

Searching for a pattern

- Libraries are shaped by earliest/more influential users
  - Not an uncommon pattern
- /bin programs are mostly C/single-user/blocking
  - No more the main pattern of the wider ecosystem
  - Less polished libpq support for other patterns

Authentication libraries



Auth #1: GSSAPI

- Enterprise-grade authentication
- It depends on the external `libgssapi`
- Enabled by default

GSSAPI

- Several issues have since been raised
  - Psycopg 2 #1242 (4s slowdown)
  - PHP #13339 (SIGABRT)
  - Psycopg 3 #1136 (crash on macOS)
  - Psycopg 3 #1213 (Segfault on Linux)
  - PgJDBC #1868 (no libpq but same problem)
  - ...
- `"gssencmode=disable"` has become a copy-paste solution
  - In Psycopg 3.3 disabled by default in bundled libpq

Auth #2: OAuth/OIDC

- Introduced in PostgreSQL v18
- Part of SASL (already in the libpq)
- OAuth Device Authorization flow
- Bearer Token flow
- Bearer Token flow improvements (in v19)

Bearer token: great flow!

- C structure to pass data to-from client
- Create your flow in the libpq client
- Socket pair for BE-FE coordination
- In v19: passing back error from flow to client

Device flow:

- Blocking only
- libcurl dependency
  - Optional dependency: great! 👍
- JSON parser

What would other clients care about?

- In Psycopg: we expose the `PQAUTHDATA_PROMPT_OAUTH_DEVICE`
- But we also reimplement it in Python on `PQAUTHDATA_OAUTH_BEARER_TOKEN[_V2]`
  - Easier dependency management
  - Async integration
  - Existing http/json integration
- About ~60 Python LoC vs. >3KLoC in `oauth-curl.c`

What woul

- In Psycopg: w
  PQAUTHDATA_PF
- But we also r
  PQAUTHDATA_OA
  - Easier de
  - Async int
  - Existing h
- About ~60 Py

```python
async def _run_device_flow(
    handler: DeviceFlowHandler, request: OauthBearerToken
) -> bool:
    if not request.openid_configuration:
        return False

    # Resolve client_id early so a missing oauth_client_id fails before
    # any HTTP call.
    client_id = handler.client_id

    async with httpx.AsyncClient() as client:
        resp = await client.get(request.openid_configuration)
        resp.raise_for_status()
        config: dict[str, Any] = resp.json()

        device_endpoint: str | None = config.get(
            "device_authorization_endpoint")
        token_endpoint: str | None = config.get("token_endpoint")
        if not device_endpoint or not token_endpoint:
            return False

        params: dict[str, str] = {"client_id": client_id}
        if request.scope:
            params["scope"] = request.scope

        resp = await client.post(device_endpoint, data=params)
        resp.raise_for_status()
        device: dict[str, Any] = resp.json()
```

My take

- OAuth Device Flow *is not a libpq feature*
  - *It is a psql feature*
  - Maybe it does make sense for other `/bin` apps?
  - `pg_dump`?
- Other clients are better served by the lower lever flow
  - Build the high level in the target language
  - Better integration with whatever runtime model
  - Better use of higher-level language libraries

What would other clients care about?

- The global hook seems a bad idea
  - (ask Andreas Karlsson, he's here somewhere)
- Maybe there should be a "connection factory"?
  - An object to configure, create connections from
  - Or a well-defined hook between `PQconnectStart()` and `Poll()`?

Question:

Who is the libpq for?

In its current form

- Single-user or multi-thread C application?
- Institutional clients?
  - GSS-API probably needed in a bank
  - not by a macOS guy on a laptop
  - but *the defaults are the same*
- Can we provide different configuration sets in the same runtime?

In its current form

- PostgreSQL /bin clients as major design driver
  - No pipeline demand -> feature long lagged
  - No async client -> async support lagging
  - Single user programs -> global structures
  - No replication client -> no replication parsing
  - Covering all /bin client needs -> library bloat?

Who is the libpq for?

- I *might* be biased 👀 but...
- Likely most of the libpq users use it via drivers
- C “final” applications are outnumbered

How can libpq evolve for everyone

- Context objects instead of globals
- SASL hooks for auth configuration?
- Maybe separate /bin dependencies into intermediate package?
  - “libpq-porcelain” vs. “libpq-high-level”

Thank you for the amazing work!

Support the Psycopg project!

- Sponsorships welcome
- Collaborations welcome
- A new package in preparation
  - Follow us on https://www.psycopg.org/