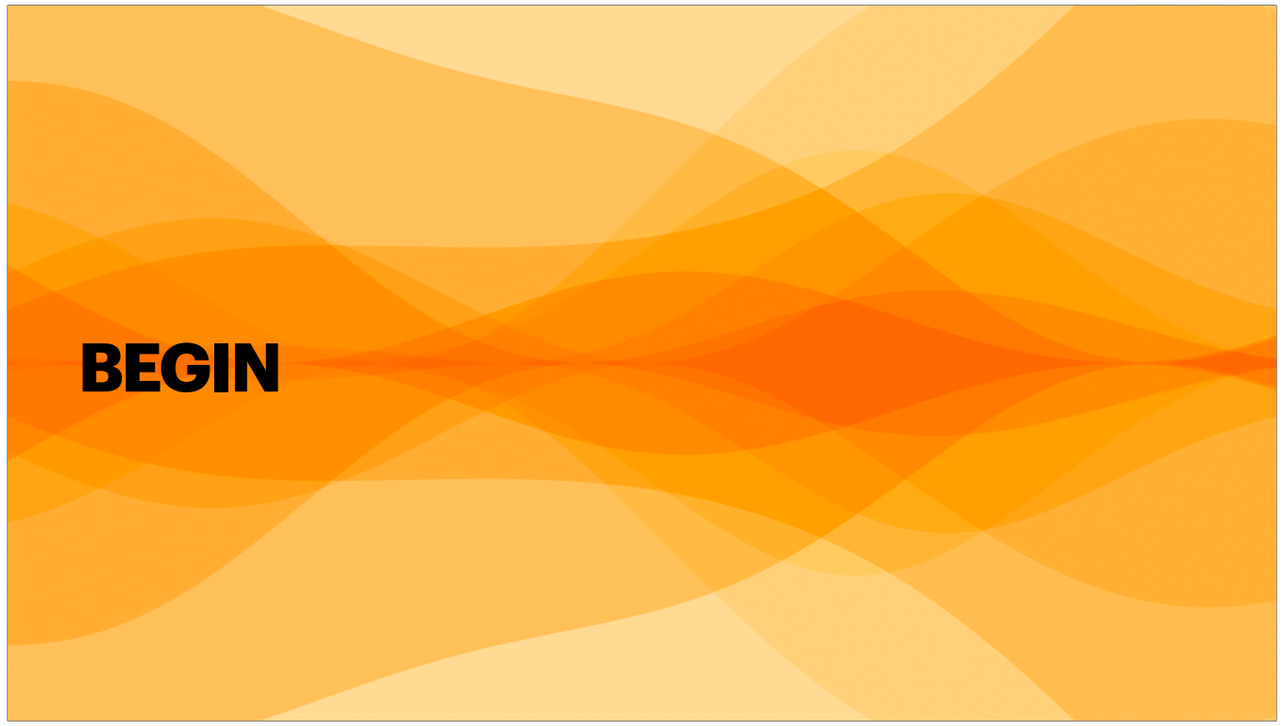
G’day everyone. I’m David Wheeler, creator of PGXN and Principal Architect at Tembo. Today I’d like to talk about the state of the Postgres Extension Ecosystem, its past, present, and future.

What I mean by the extension ecosystem is how one discovers extensions, learns about them, and installs them, and the challenges that have hindered broad adoption.



Let’s go back in time to the beginning.

LEGACY

Long history of extensibility

Two approaches

- shared_preload_libraries
- Pure SQL (including PLs)

A few intrepid adopters

- PostGIS
- BioPostgres
- PL/R
- PL/Proxy
- pgTAP

- Postgres has a long history of extensibility
- In the old days, there were two basic approaches to extending Postgres without forking it
    - 1. Loading dynamic shared objects in shared preload libraries
    - 2. Pure SQL extensions that used procedural languages and SQL features to create objects
- There were quite a few intrepid adopters in those years
    - Including PostGIS, BioPostgres, PL/R, PL/Proxy, pgTAP, and others

POSTGRES 9.1

Dimitri Fontaine adds extensions

Key features:

- **Compile and install**
- **CREATE/UPDATE/DROP EXTENSION**
- **pg_dump and pg_restore**

- Then Dimitri Fontaine submitted a patch for formal extension support, which was committed and released in Postgres 9.1 in 2011
- This work built on those existing patterns for extending Postgres, but formalized things through three key features:
  - Tooling to compile and install extensions, both DSOs and pure SQL
  - New SQL commands to create, update, and drop extensions, which bundle together objects into named units
  - And backup and restore support to ensure consistent versioning and behavior when upgrading Postgres itself

pair.control

```
# pair extension
comment = 'A key/value pair data type'
default_version = '1.0'
module_pathname = '$libdir/pair'
relocatable = true
```

- The artifact that holds these pieces together is the control file, named for the extension it manages
- This file tells Postgres the default (latest released) version of an extension,
- the path to any DSO modules, and other configuration data, like whether the extension can be moved from one schema to another

Makefile

```
EXTENSION    = pair
MODULEDIR    = $(EXTENSION)
DATA         = sql/pair--1.0.sql
MODULES      = src/pair
TESTS        = test/sql/base.sql
REGRESS      = base
PG_CONFIG   ?= pg_config

PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

- The Postgres build system was also optimized for Extensions. One need only create a Makefile with the extension name and, for DSOs, module directory
- List documentation files and SQL files to deploy versions of the extension, as well as the paths to DSOs to be build
- And configuration for testing with the pg_regress utility
- The user can point to the pg_config utility relevant for their Postgres version
- And the Makefile simply loads the PGXS Make configuration and runs it

Build and Install

```
❯ make
gcc -c -o src/pair.o src/pair.c
gcc src/pair.o -o src/pair.dylib

❯ make install
install -c -m 644 pair.control '/pgsql/share/extension/'
install -c -m 644 sql/pair--1.0.sql  '/pgsql/share/pair/'
install -c -m 644 src/pair.dylib '/pgsql/lib/'

❯ make installcheck PGUSER=postgres
ok 1         - base                       15 ms
1..1
# All 1 tests passed.

❯
```

- This setup enables straightforward compilation of C and SQL extensions by running “make”
- Then “make install” installs the relevant files, including the control file, SQL files, and DSOs
- And finally, “make installcheck” runs pg_regress tests against the newly-installed extension

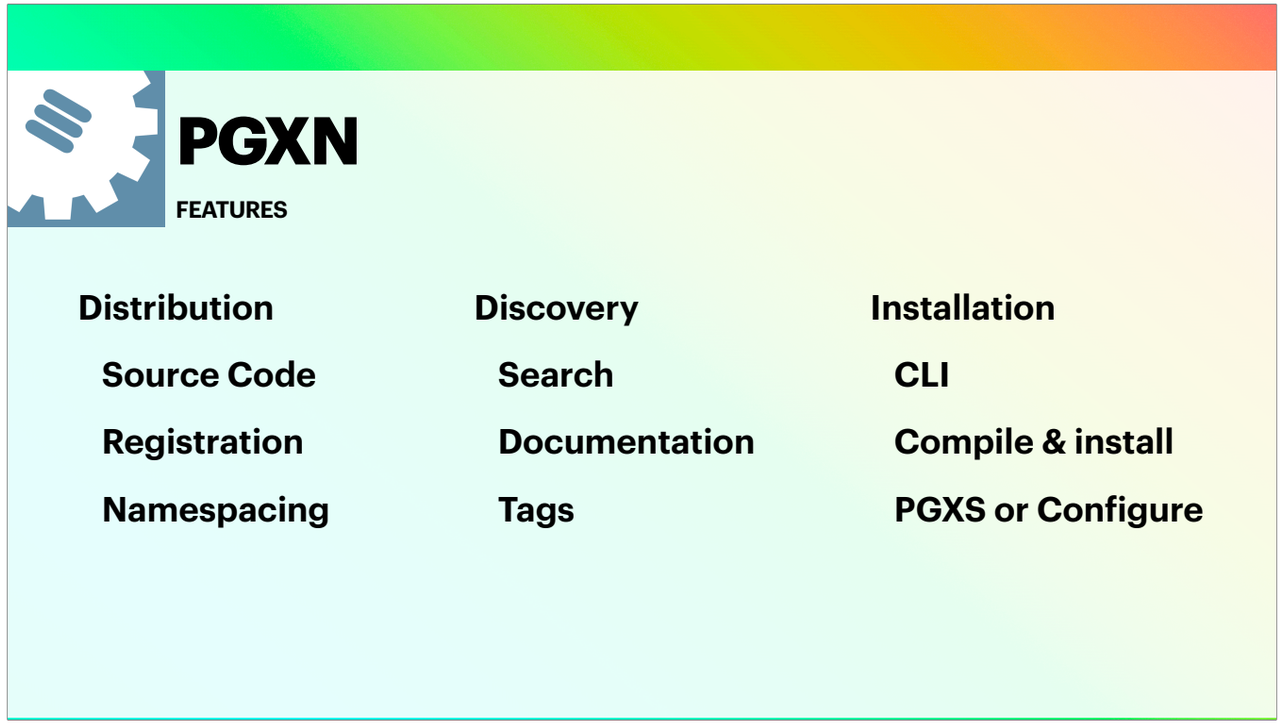
CREATE EXTENSION

```
❯ psql -d try -c 'CREATE EXTENSION pair'
CREATE EXTENSION

❯ pg_dump -d try
-- Name: pair; Type: EXTENSION; Schema: -; Owner: -
--

CREATE EXTENSION IF NOT EXISTS pair WITH SCHEMA public;

-- Name: EXTENSION pair; Type: COMMENT; Schema: -; Owner:
--
COMMENT ON EXTENSION pair IS 'A key/value pair data type';
```

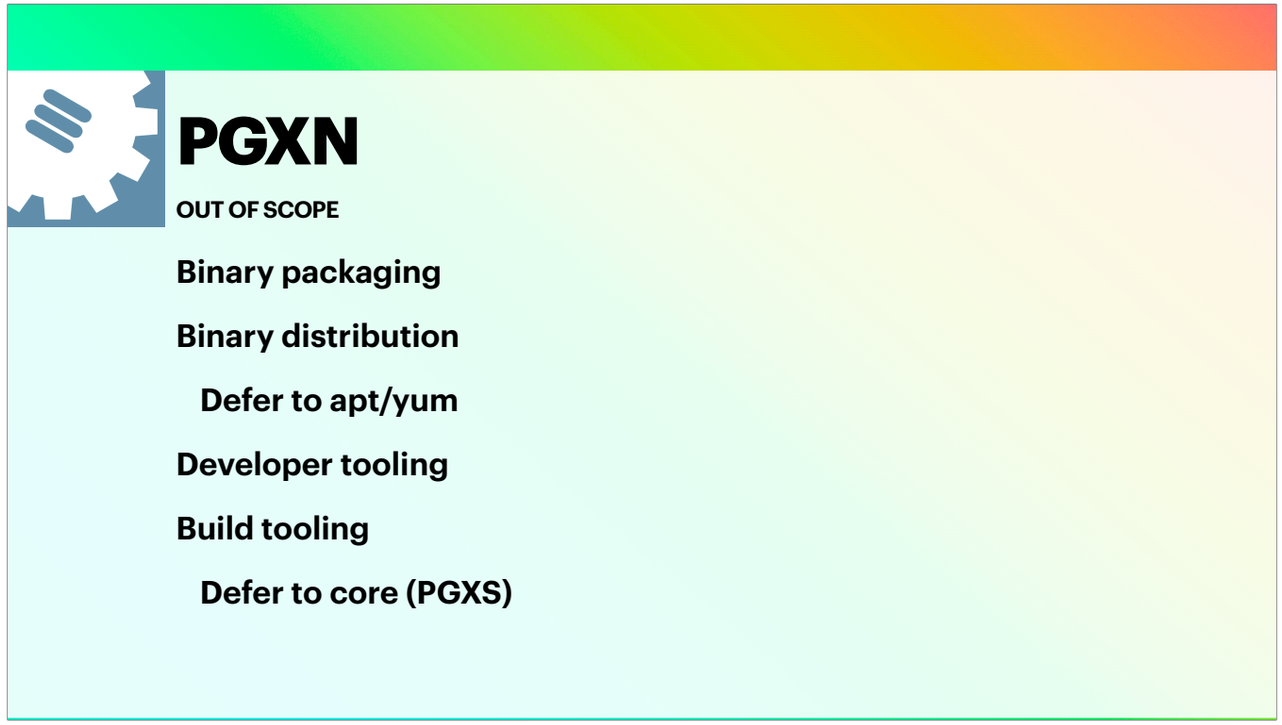
- And now users can use “CREATE EXTENSION” to add the extension to a database
- And thanks to bundling of features into a single unit, pg_dump output can likewise simply install the extension as before.

- All this infrastructure created new opportunities for the extensibility of Postgres.

I was pretty excited at the possibilities around that time, publicly claiming that “PostgreSQL today is not merely a database, it’s an application development platform.”



- As a result, I proposed to create PGXN, a service for distributing Postgres extensions, with the goal to be the canonical source for all publicly-available extensions
- The features were to include distribution of source code, as well as user registration and namespace management, so there would be no conflict between named extensions
- PGXN also aimed to provide comprehensive search features, and the ability to browse and read well-written documentation. Individual releases could also be tagged for better discoverability.
- A command-line client, meanwhile, would provide a simple interface to download, compile, and install extensions.



- Now, a number of features were out-of-scope of that vision
- These include binary packaging; PGXN would be for source code distribution only
    - It seemed too much to take on support for a wide variety of platforms, so we deferred binary distribution to the community Apt and Yum repositories
- PGXN also would not include developer tooling
- Or Build tooling. There were already solutions for building extensions
    - Most notably PGXS and whatever future directions the Core project might take to support extension authors

- I launched the project in 2010, around the time Dimitri started developing formal extension support
- By December, the little fundraiser I set up had met its goal and it was time to get to work
- The site, pgxn.org, launched in April of 2011 with the first few extensions and a public REST API
- Meanwhile, Daniele Varrazzo took it upon himself to write the command-line client. It was on my list but he beat me to it! Which was cool, because he wrote it in Python, deftly demonstrating that any language could use the APIs and interfaces
- Almost to prove the point, Dickson Guedes released a suite of dev utils written in Ruby in June 2011. The community involvement was so great!

Here's what pgxn.org looks like today. It currently hosts over 2200 releases of almost 400 extensions amongst 430 users. It's pretty cool, I think, you should check it out.

Distribution API

- **Name:** `dist`
- **Returns:** `application/json`
- **URI Template Variables:** `{dist}`
- **Availability:** Mirror Server, API Server

Returns JSON describing all releases of a distribution. This method requires that the distribution name be known; if no distribution exists with the specified name, a 404 response will be returned.

Mirror API Structure

The structure of this JSON file on mirror servers is quite simple. A few examples:

- countnulls.json
- pair.json
- pgmp.json

The contents constitute a single JSON object with the following keys:

Key	Type	Description
`name`	String	The name of the distribution.
`releases`	Object	Lists all releases of the distribution.

Pages 21

Quick Links

Basics

- Introduction
- Terminology
- Data Types
- Status Codes
- index
- JSONP Callbacks

Mirror APIs

- download
- readme
- meta
- dist
- extension
- user
- tag
- stats
- mirrors
- spec

API Server APIs

As I said there is also a robust REST API, documented in the GitHub wiki for the pgxn-api project. Here the distribution API allows a client to find the latest release of an extension.

```
❯ pgxn install semver
INFO: best version: semver 0.32.1
INFO: saving /tmp/tmpbr43m_p3/semver-0.32.1.zip
INFO: unpacking: /tmp/tmpbr43m_p3/semver-0.32.1.zip

INFO: building extension
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-stat
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-stat
cp sql/semver.sql sql/semver--0.32.1.sql

INFO: installing extension
ginstall -c -m 644 ./semver.control '/pgsql/extension/'
ginstall -c -m 644 ./sql/semver--0.10.0--0.11.0.sql ./sql/semver—0.10.0
ginstall -c -m 755 ./src/semver.so '/pgsql/lib/'
ginstall -c -m 644 ./doc/semver.mmd '/pgsql/share/doc/extension/'
```

- The CLI that Daniele wrote uses that API. Here we ask it to install the semver extension
- It uses the API to find the latest stable version, and a second to download its zip file
- Then it uses the core-provided PGXS pipeline to compile the extension
- And then to install all the parts where PostgreSQL wants to find them. That’s it!

PGXN

SHORTCOMINGS

- Little development since 2012
- Search limitations
  - Docs preferred
  - Most distributions have few
- Source of Record
  - Minority of available extensions
  - Releases uneven or abandoned

- Over time we've recognized a number of shortcomings to PGXN
- First and foremost, there has been little work on it since 2012. Over the years I've kept it running and tweaked a few things here and there. A couple years ago, for example, I finally made it look good on mobile devices
- Another problem is that PGXN has always suffered from significant search limitations
  - Namely, it has defaulted to searching extension documentation
  - But most distributions don't provide extension documentation, only a README, at best.
- Sadly, PGXN has also not become the source of record for Postgres extensions
  - It contains at best 40% of publicly-available extensions.
  - Furthermore, releases of those extensions have been uneven, and many abandoned, in part because the release process was quite manual until I developed a GitHub test-and-release workflow a few years ago.

• In other words, I would say, “In classic SDLC fashion, the PGXN POC shipped as an MVP and was neglected.”

Meanwhile, things have not stood still the in the rest of the Postgres extensions ecosystem.

CLOUDY DAYS

Non-Core Extension Counts:

Azure: 25

GCP 29

AWS: 48

PGXN: 369

joelonsql/PostgreSQL-EXTENSIONs.md: 1,186

- Chief among the changes has been the emergence of “Postgres as a service” providers. They curate extensions for their users. As of this spring…
- Azure provides 25 non-core extensions
- GCP provides 29
- AWS 48
- PGXN meanwhile has 369 distributions with 396 extensions
- But as I said, that’s a fraction of those available. Joel on SQL has inventoried almost 1200 publicly-available extensions in a wide variety of locations, especially GitHub

- This leads me to wonder: why has uptake been so…modest?

LOST OPPORTUNITIES

POSTGRES EXTENSIONS:

Difficult to find and discover

Under-documented and difficult to understand

Maturity difficult to gauge

Hard to configure and install

No comprehensive binary packaging

Centralized source distribution insufficient

Insufficient developer tooling

- Despite the best of intentions, some opportunities were never met.
- Postgres extensions remain quite difficult to find and discover, since they're scattered to the internet winds
- Most that one can find, including on PGXN, are under-documented and difficult to understand
- When you do find them, it can be tricky to gauge the maturity, reliability, and stability of an extension, especially if there isn't much documentation
- Furthermore, they're difficult to configure and install. Most users just want the add them to their dev and production clusters, and not have to install a bunch tooling like compilers and devel packages on those servers
- This is because there is no comprehensive binary packaging system covering a wide variety of platforms, architectures, and Postgres versions. It's catch-as-catch-can between the community Apt and Yum repositories and tools like Homebrew. Windows is pretty much omitted altogether.
- Clearly the centralized distribution of source code is insufficient to meet the needs of people who just want to quickly find, install and use extensions.
- Part of the problem is insufficient developer tooling. One has to suss out how to create and maintain extensions by following scattered blogs and example repositories, and the lack of features like documentation standards increase the difficulty of providing consistent product

But as I said, things haven’t stood still. A number of new projects have emerged in the last couple years that attempt to fill some of the development and distribution gaps.

TLE

TRUSTED LANGUAGE EXTENSIONS

Goal: Empower app developers

No C, only trusted languages

SQL, PL/pgSQL, PL/Perl, PL/v8, PL/Rust, etc.

Easy install via SQL functions

Portable (no compilation)

No file system access

Hooks into `CREATE EXTENSION`

Supports custom data types

Plans for FDWs, background workers

Inspiration for core development

From AWS & Supabase

- One such development is Trusted Language Extensions, or TLEs
- The TLE project wants to empower application developers to build database functionality without having to do the full compile/install dance on their servers
- By supporting only trusted languages, TLEs eliminate the need to install files on the server file system as C extensions require
    - Any trusted language works, including SQL, PL/pgSQL, PL/Perl, PL/v8, PL/Rust, and more
- The Pg_TLE extension provides functions for easy installation
- And TLEs can be ported between systems and architectures, since they require no compilation
- Once an extension is installed, pg_tle has hooks into CREATE EXTENSION to allow seamless loading
- The system supports custom data types via its API, which exposes the underlying Postgres APIs without requiring C access
- This pattern will be used to add support for other hooks not normally accessible to non-C extensions, including foreign data wrappers and background workers
- The project also aims to inspire or at least inform designs for adding SQL-only extensions to the core
- Our friends at AWS and Supabase steer this project

```
SELECT pgtle.install_extension(
  'pair',
  '0.1.0',
  'Arbitrary key/value pair data type',
  $$
    CREATE TYPE pair AS ( k text, v text );

    CREATE FUNCTION pair(text, text)
    RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair;';

    CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, FUNCTION = pair);
  $$
);

CREATE EXTENSION pair;
```

Control file info

No files

TLE Hooks into command

- Here's a TLE example that installs an extension named "pair"
- Note that the first few arguments to the install_extension() function are identical to the contents of the control file we saw earlier
- We pass the contents of the normal SQL files that create the extension objects as subsequent arguments
- Once installed, thanks to pg_tle's hooks into the guts of Postgres, a normal CREATE EXTENSION command will automagically load the extension, just as if it has been installed via the file system as we saw earlier.

PGRX

Native Rust extensions

Build shared object libraries

Bridges C interfaces & internals

Full access to Postgres features

- Data types, functions, triggers, FDWs, etc.

Use crates.io to build quickly

Developer-friendly tooling

- Manage test clusters, run tests, install, package, etc.
- Automatic schema generation

Lots of community excitement

Under active development

From ZomboDB & PgCentral Foundation

- Another recent source of excitement is pgrx,
- A framework for building extensions in Rust
- It’s like C in that it builds shared object libraries to be loaded into Postgres
- But it provides fairly transparent bridges from the Postgres C APIs to Rust
- This allows full access to nearly all Postgres features
    - Including custom data types, functions, triggers, foreign data wrappers, background workers, and more
- And since pgrx is just Rust, it’s easy to quickly add functionality from Rust crates without having to write something from scratch.
- pgrx also provides developer-friendly tooling
    - To manage test clusters, run tests, install and package the extension, and more
    - As well as automatic schema generation, simplifying the management of the SQL used to create an extension
- There’s a ton of community excitement and a slew of new Rust extensions released every month
- The project itself is under active development
- Thanks to our friends at ZomboDB and the PG Central Foundation

```rust
/// json_schema_validates_json validates `data` against `schema`.
#[pg_extern(immutable, strict, parallel_safe, name = "jsonschema_validates")]
fn json_schema_validates_json(data: Json, schema: Json) -> bool {
    let schemas = [schema.0];
    run_validate!(id_for!(&schemas[0]), &schemas
}

/// jsonb_schema_validates_jsonb validates `data` against `schema`.
#[pg_extern(immutable, strict, parallel_safe, name = "jsonschema_validates")]
fn jsonb_schema_validates_jsonb(data: JsonB, schema: JsonB) -> 
    let schemas = [schema.0];
    run_validate!(id_for!(&schemas[0]), &schemas, dat
}
```

Transparent type mapping

Polymorphism!

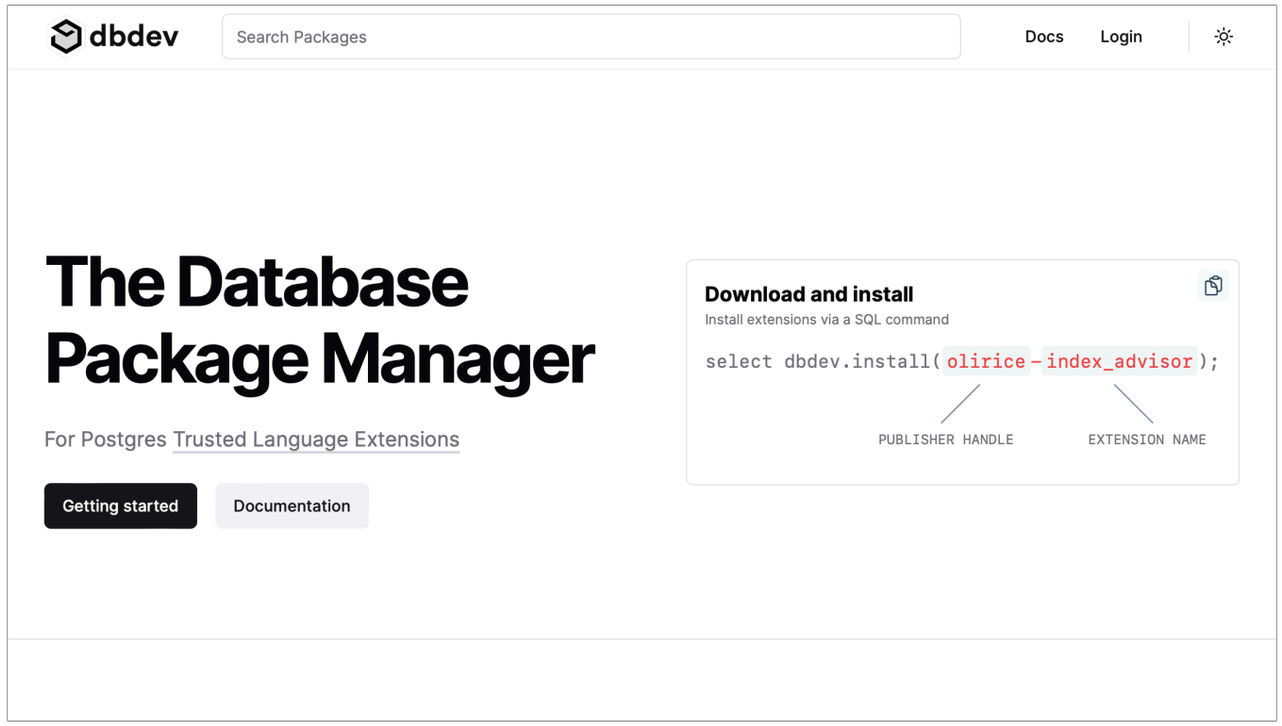
All core types

- Here’s a quick example from my jsonschema project, which uses the boon crate to provide full jsonschema v2020 validation functions
- These functions define the Postgres function interface; note the types in the function signature. The Json type transparently maps Postgres JSON to Rust serde objects
- A second function does the same for JSONB; all the core types have corresponding pgrx types, including arrays, variadic arguments, and ranges
- The pg_extern macro tells pgrx how to define the SQL function. Here it’s named jsonscema_validates, which differs from its Rust name
- An identical macro for the JSONB version uses the same name, mapping the Rust functions to polymorphic Postgres functions

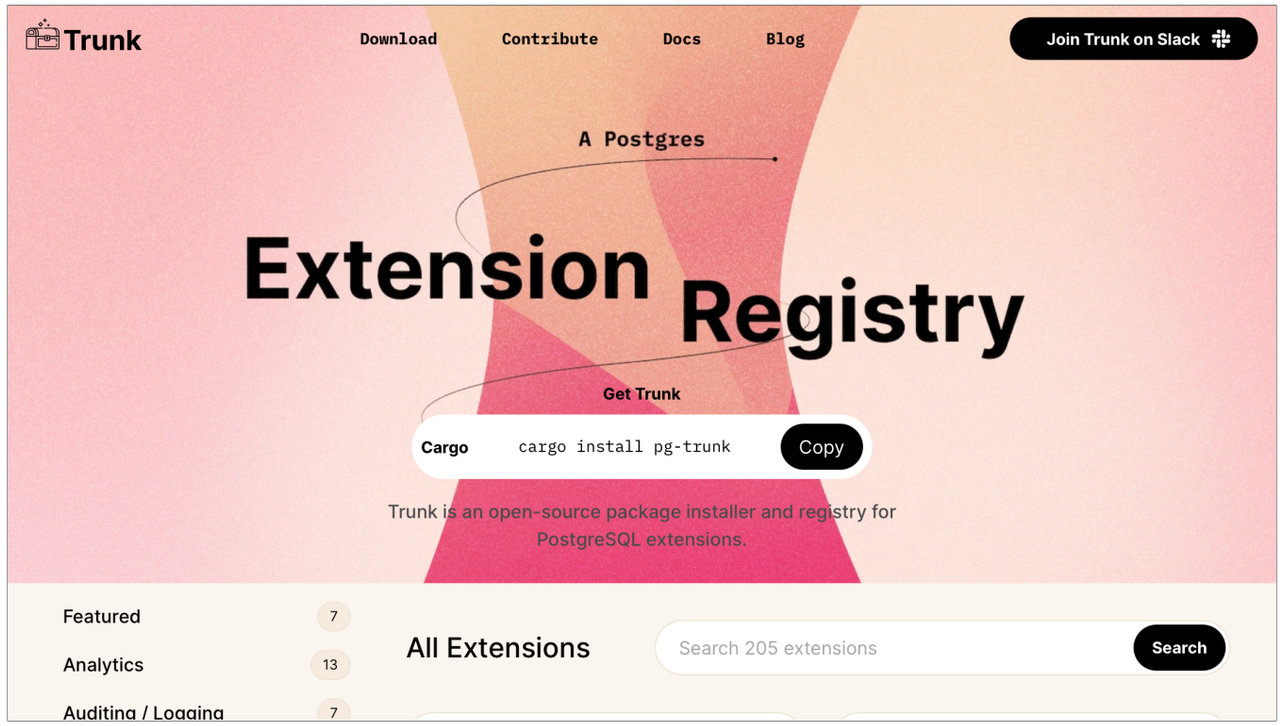
```sql
CREATE  FUNCTION "jsonschema_validates"(
    "data" json, /* pgrx::datum::json::Json */
    "schema" json /* pgrx::datum::json::Json */
) RETURNS bool /* bool */
IMMUTABLE STRICT PARALLEL SAFE
LANGUAGE c /* Rust */
AS 'MODULE_PATHNAME', 'json_schema_validates_json_wrapper';

CREATE  FUNCTION "jsonschema_validates"(
    "data" jsonb, /* pgrx::datum::json::JsonB */
    "schema" jsonb /* pgrx::datum::json::JsonB */
) RETURNS bool /* bool */
IMMUTABLE STRICT PARALLEL SAFE
LANGUAGE c /* Rust */
AS 'MODULE_PATHNAME', 'jsonb_schema_validates_jsonb_wrapper';
```

- The resulting SQL looks like this.
- Note the JSON arguments
- Same goes for the JSONB version, using the same function name! pgrx automagically generates this SQL code.

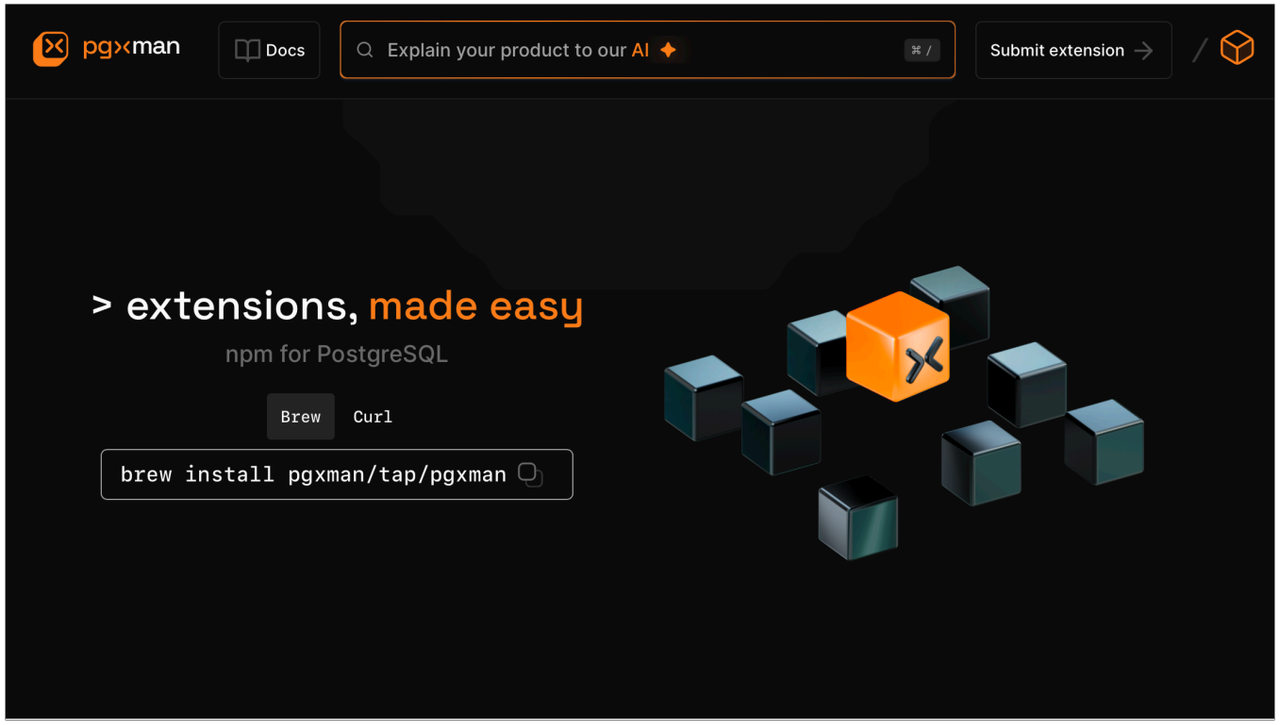
- Meanwhile, a number of projects have cropped up to try to improve upon the PGXN discovery and installation issues.



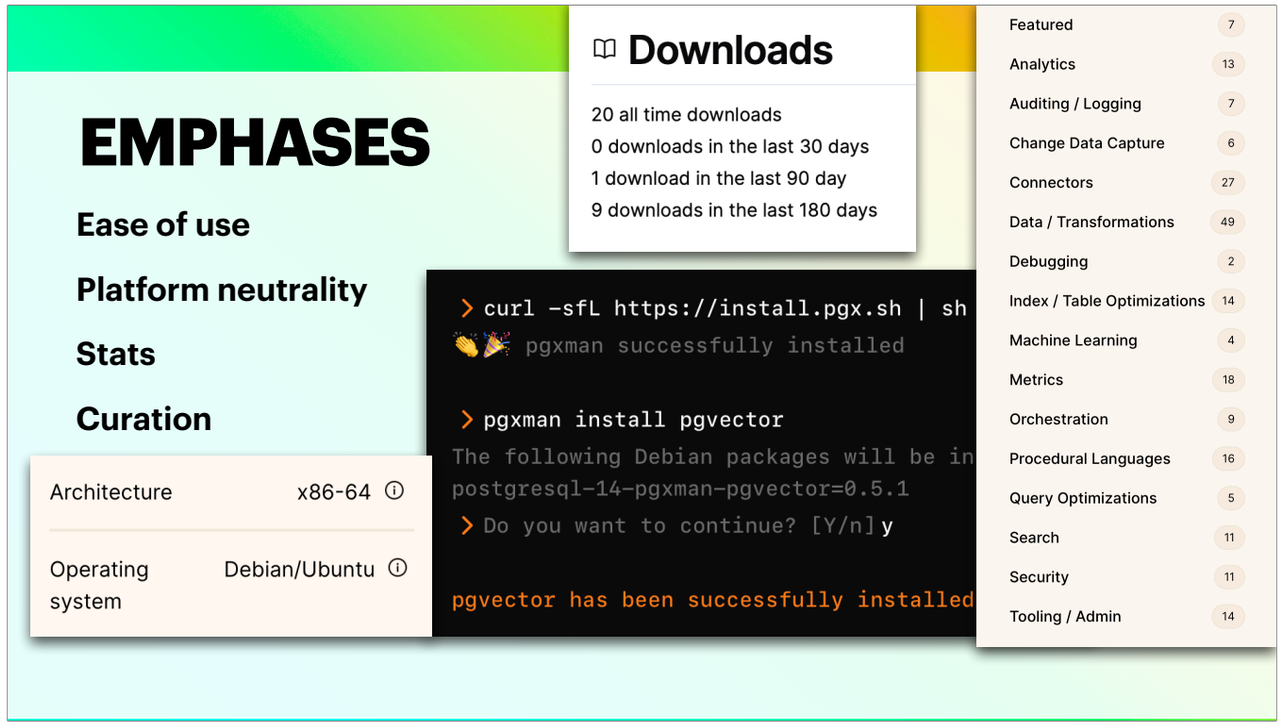
database.dev, or “DB Dev”, is a new packaging registry for trusted language extensions. This service makes it easy to download and install TLEs right in the database!



- Tembo, meanwhile, developed Trunk, hosted at pgt.dev, which provides binary packaging for over 200 extensions. Today it supports only AMD64 Linux, but plans to support a wide variety of platforms and OSes in the future.



Similarly, PGXMan from Hydra provides Debian packages for many extensions, and plans to support for macOS packages.



- These binary registries have emphasized features that PGXN has not. These include
- Ease of use, as in this demonstration of installing PGXMan and a first extension in just two commands.
- Platform neutrality, as in Trunk showing the available platforms for an extension
- Statistics, such as DB Dev listing download numbers for an extension, which can be helpful for evaluating maturity and community acceptance
- And curation, as in Trunk's use of categories for extensions, which have proven a popular vector for discovering extensions to address issues in specific problem domains

- All these recent developments have invigorated the broader extension ecosystem. But many of the promises we saw early on sadly have yet to be fulfilled.

STATE OF AFFAIRS

Despite early promise…

No single source

Many incomplete attempts

Poor discovery

Difficult installation

Insufficient docs

Low adoption

But…

To whit

- There is no single source for a complete list of all extensions.
    - Every attempt and doing so remains incomplete
- This makes discovery difficult, as one has to know to search PGXN, GitHub, GitLab, Bitbucket, the Postgres Wiki, various cloud providers, specific web sites like postgis.org, and more
- Installation is difficult outside the packages provided by your chosen installer (if you can find a list of them). No one wants to compile extensions on their production database server.
- Most extensions that do exist have very little documentation, usually just a README. And the lack of a documentation standard means that docs, when they exist, vary tremendously in quality and format
- All this has led to pretty low adoption rates. Few databases use extensions at all, and the number using more than one is vanishingly small
- Still, as we've seen…

- “There sure has been a lot of excitement around extensions lately.”

Which brings us to the future. What new opportunities are there to improve the extensions ecosystem for developers and users?

PGXN V2

The New World Order

Started this year

Made possible by Tembo

Consider emerging patterns

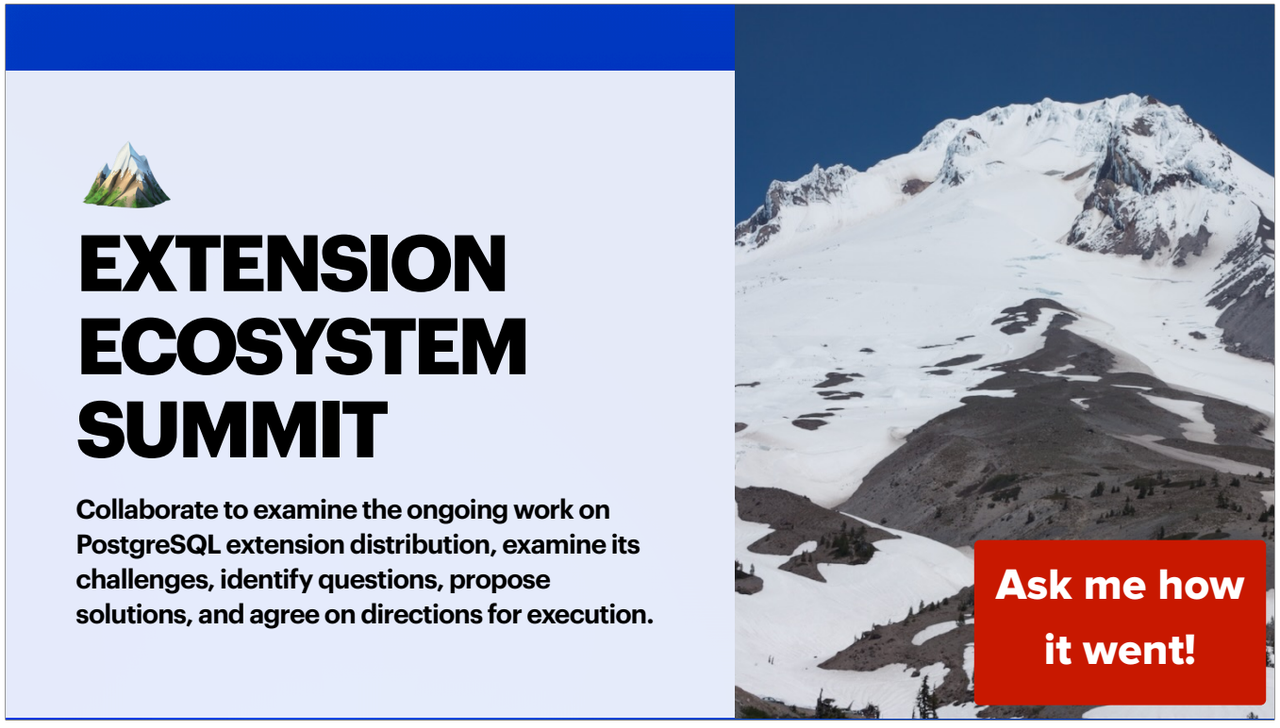
Meditate on deficiencies

Engage deeply with the community

Design and implement new architecture

Serve the community for next decade

- To answer those questions, we've launched a project, code-named "PGXN v2"
- To find out and make it happen.
- This came about thanks to Tembo, the creators of Trunk, who hired me specifically to
- Consider the emerging patterns, like binary registries and development frameworks
- Meditate on the deficiencies, including discovery and ease of installation
- To engage deeply with the broader community of Postgres developers, extension authors, and users
- In order to understand the use cases well enough to design and implement a new architecture
- That will serve the community for the next decade or more

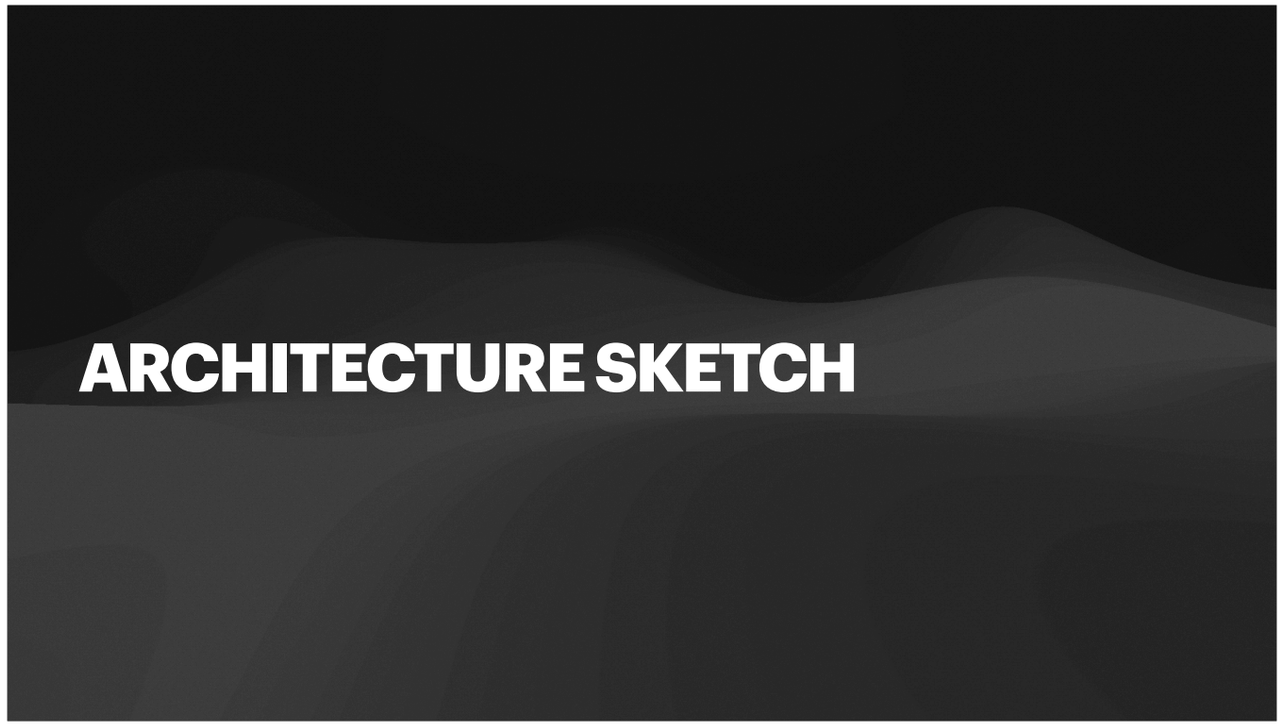


- One of the key points of community engagement has been the Extension Ecosystem summit. We had six virtual “mini-summits” to share ideas, discover new patterns, and discuss issues, as well as an in-person Summit at PGConf.dev on May 28th.
- The summit took place after recording this talk, so ask me how it went in the Discord!

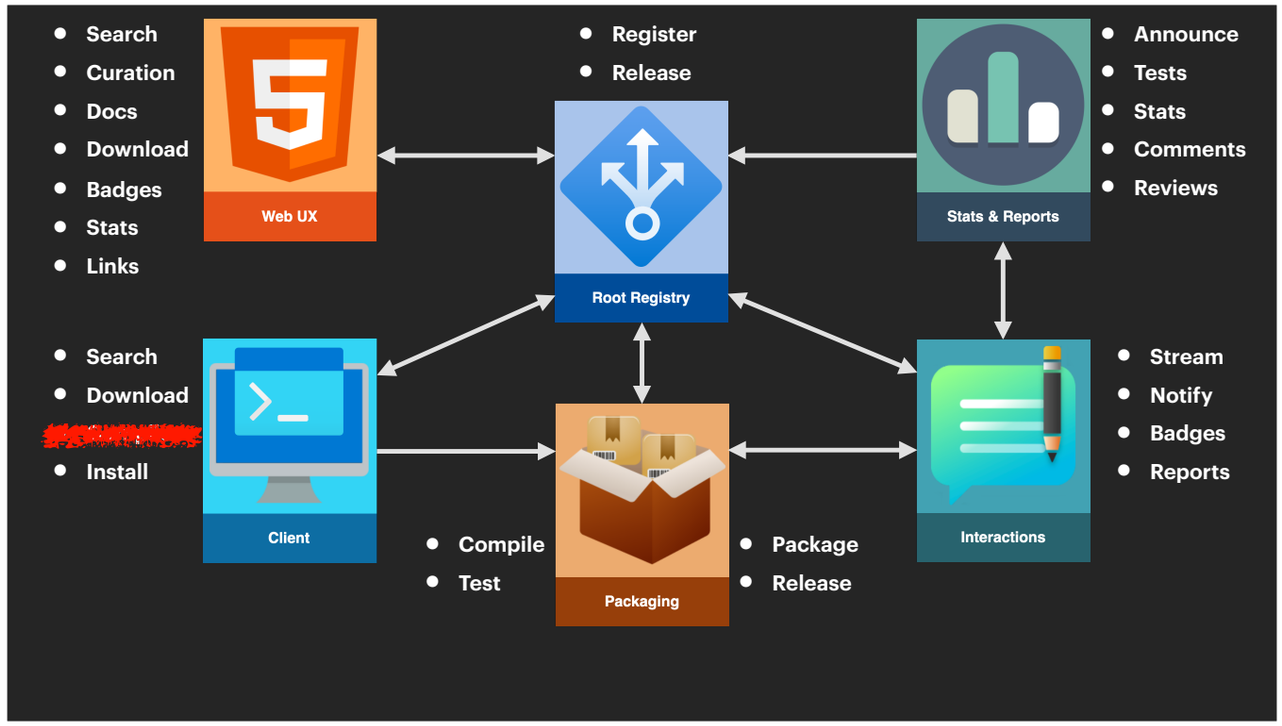
PGXN V2 JOBS

Authoritative
Beneficial
Integrated
Discoverable
Informative
Programmable
Installable
Trusted
Manageable

- We've also worked to identify the key jobs to be done for the PGXN extension ecosystem. It must be:
- **Authoritative**: Be the official, canonical source of record for all Postgres extensions
- **Beneficial**: It's expected that extension authors benefit from developing and publishing their extensions there
- **Integrated**: It's easy for developers to start writing and publishing extensions
- **Discoverable**: Make extensions easy to find and understand
- **Informative**: Make extension documentation consistent, standardized, and comprehensive
- **Programmable**: where distribution services provide stable, easy-to-use, and comprehensive APIs with which downstream developers can build other tools and products
- **Installable**: Provide automatic binary packaging for wide variety of platforms
- **Trusted**: Validate extensions and protect from supply chain vulnerabilities
- **Manageable**: Provide intuitive, powerful interfaces for installing and managing extensions



I’ve made a high-level architecture sketch to capture the categories of tools and services to fulfill these jobs.



- At root is, of course, the root registry
- Where authors register accounts and release extension source code
- This is complemented by a well-known web site
- for users to find those extensions, read their docs, look at curation information, and download them
- Next is a command-line client
- which allows one to quickly find, download, compile, and install any of the indexed extensions. These services build on PGXN and other precedents.
- Brand new is the idea of Interactions
- These services stream registry changes like new releases, which subscribers can use for their own purposes. It will also provide WRITE APIs where trusted clients can submit reports, badges, stats, and other metadata about extensions. Think test matrices or Security scores.
- Stats and reports are the categories of services that provide that kind of data, getting notifications from Interactions for recent relases, downloading sources from the root, and submitting results to interactions, which publishes them in the root registry
- From there the Web UX picks them up to show links, badges, and simple stats to help users evaluate extension quality
- Last but not least we have binary packaging services that support a variety of OSes, architectures, and Postgres versions. They subscribe to interactions to learn about new releases
- They pull the source from the root registry
- And use the client to build and package them, then push the data back to the root registry so the appropriate availability data can be visible in the UX.
- The CLI will then also install from the registry, eliminating the need for DBAs to compile from source

PRODUCT SKETCH

- Let’s imagine what that will look like. Here’s a mockup that imagines using the CLI to manage extensions

Install Management

```
❯ pgxn manage ls

      install      |                        path
-------------------+-------------------------------------------------
 homebrew/14.12    | /opt/homebrew/Cellar/postgresql@14/14.12
 pgenv/16.3        | /Users/david/.pgenv/pgsql-16.3
 pgenv/17devel     | /Users/david/.pgenv/pgsql-17devel
 postgres.app/16.3 | /Applications/Postgres.app/Contents/Versions/16/

❯ pgxn manage pgenv/16.3
Now managing extensions for pgenv/16.3
❯
```

- First, let’s see what Postgres installs we can manage
- This host has four installs: one from Homebrew, two from pgenv, and the Postgres.app
- The “manage list” command also shows the root directory for each installation, like this one for the pgenv-installed Postgres 16.3
- Let’s tell the CLI to manage that install

Package Management

```
❯ pgxn package ls
Packages installed in pgenv/16.3:

      package       | installed | latest |                   description
--------------------+-----------+--------+---------------------------------
 core/citext        | 1.6.0     | ✅     | data type for case-insensitive ch
 core/jsonb_plperl  | 1.0.0     | ✅     | transform between jsonb and plper
 core/unaccent      | 1.1.0     | ✅     | text search dictionary that remov
 theory/semver      | 0.32.0    | 0.32.1 | Semantic version data type

❯
```

- Now use the “package list” command to show the extension packages that the CLI manages for that install
- There are three extensions from the Postgres core, plus the semver package from PGXN user theory
- Note the “latest” version listed here. That means there is a newer version available to be installed

Package Management

```
❯ pgxn upgrade -v theory/semver

Upgrading theory/semver 0.32.1 in pgenv/16.3…
Found theory-semver-0.32.1+1.pg16+2.darwin_arm64.zip
Unpacking theory-semver-0.32.1+1.pg16+1.darwin_arm64.zip
Installing lib/semver.dylib in ~/.pgenv/pgsql-16.3/lib/
Installing share/semver–0.32.1.sql in ~/.pgenv/pgsql-16.3/share/extensions
Installing share/semver.control in ~/.pgenv/pgsql-16.3/share/extensions/
Installing doc/semver.md in ~/.pgenv/pgsql-16.3/share/doc/

Done!
❯
```

Also Linux, *BSD, Windows, etc

- Let’s use the upgrade command to upgrade it!
- Note that it found a binary package for Postgres 16, macOS/Darwin, and the arm64 architecture
- It then installs the DSO in the libdir. No compilation!
- The remaining SQL, control, and documentation files also go where they belong
- And that’s it. We plan to at least support Linux, BSD, Windows, and likely others

Package Management

```
❯ pgxn search json

  rating  |          package           | version |    date    |
----------+----------------------------+---------+------------+-----------
 ★★      | hadi/json_fdw              | 1.0.0   | 2013-11-18 | Foreign Dat
 ★★★★    | theory/jsonschema          | 0.1.0   | 2024-04-30 | JSON Schema
 ★★★     | jma/yamltodb               | 0.10.0  | 2022-11-08 | Database ch
 ★★      | shanekilkelly/bedquilt     | 2.1.0   | 2016-09-12 | A json docu
 ★★★     | furstenheim/is_jsonb_valid | 0.0.1   | 2017-08-17 | JSONB valid
 ★★★     | postgrespro/jsonb_schema   | 1.0.0   | 2020-08-24 | Decouple js
 ★★      | kungfury/jsoncdc           | 0.1.0   | 2017-11-23 | Translates

❯
```

- Now, lets’s say we want to do some JSON validation. We search the registry for “json” to see what’s what.
- It shows the search results, including some stats — in this moc averages from some sort of ratings service
- This one looks decent. It has 4 stars, does JSON schema validation, and has a pretty recent release

Package Management

```
❯ pgxn install -v theory/jsonschema

Installing theory/jsonschema 0.1.0 into pgenv/16.3…
Found theory-jsonschema-0.1.1+2.pg16+1.darwin_arm64.zip
Unpacking theory-jsonschema-0.1.1+2.pg16+1.darwin_arm64.zip
Installing lib/jsonschema.dylib in ~/.pgenv/pgsql-16.3/lib/
Installing share/jsonschema–0.1.0sql in ~/.pgenv/pgsql-16.3/share/extensio
Installing share/jsonschema.control in ~/.pgenv/pgsql-16.3/share/extension
Installing doc/jsonschema.md in ~/.pgenv/pgsql-16.3/share/doc/

Done!
❯
```

- Let’s install it.
- As before, the CLI finds a binary package appropriate to this system and Postgres version
- Once again installing the pre-compiled DSO
- As well as the control, SQL, and documentation files

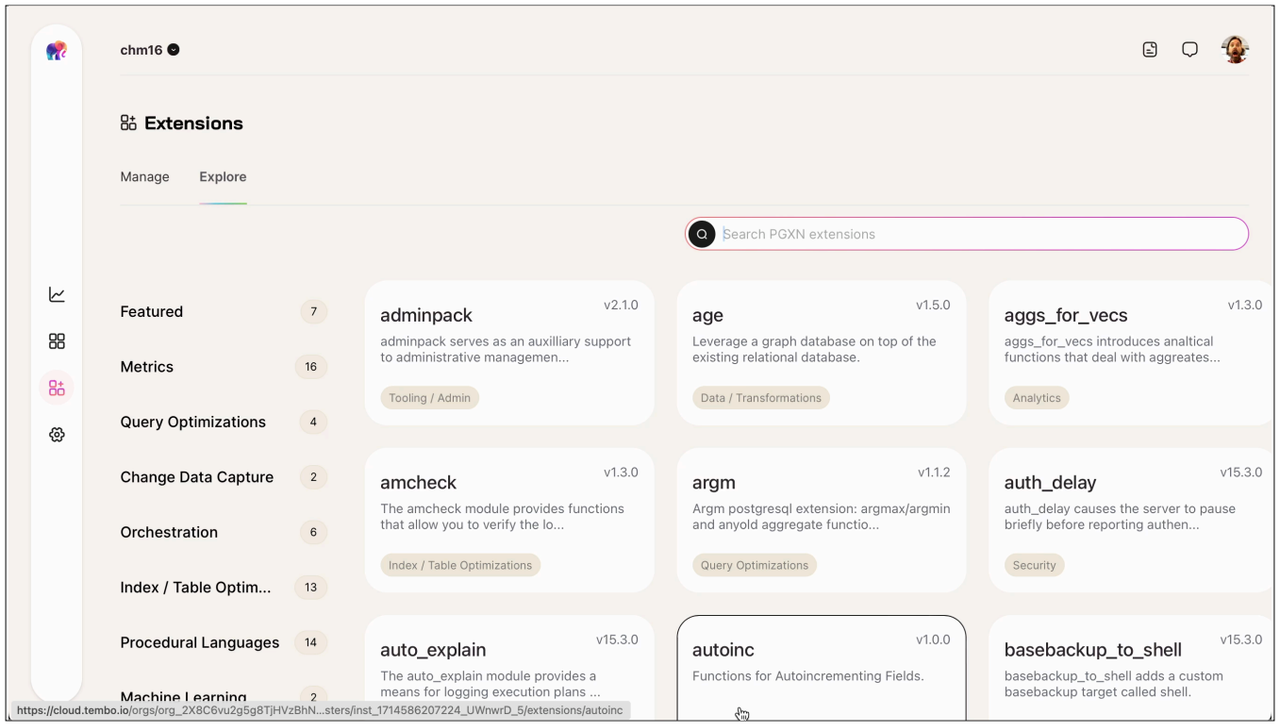
Package Management

```
❯ pgxn package ls
Packages installed in pgenv/16.3:

      package       | installed | latest |                   description
--------------------+-----------+--------+---------------------------------
 core/citext        | 1.6.0     | ✅     | data type for case-insensitive ch
 core/jsonb_plperl  | 1.0.0     | ✅     | transform between jsonb and plper
 core/unaccent      | 1.1.0     | ✅     | text search dictionary that remov
 theory/jsonschema  | 0.1.0     | ✅     | JSON Schema validation functions
 theory/semver      | 0.32.0    | ✅     | Semantic version data type

❯
```

- So what does the list of packages look like now?
- There we go
- Note that semver and jsonschema are now the latest versions. Cool!

FEDERATE TO PROVIDERS

- A crucial idea is that all this is driven by RESTful APIs. So the functionality isn’t limited to the CLI. Anyone can use it.



- For example, a Postgres as a service provider like Tembo will integrate the API to enable easy installation of binaries via a web UX
- So perhaps we search for semver among all PGXN extensions
- Then, having found it
- Click the install button and get the extension
- This functionality is identical to the CLI installation example, and uses the APIs. All public and private Postgres services will be able to use the service, or to federate and curate extensions for their customers.

JOIN US

Background: https://wiki.postgresql.org/wiki/PGXN_v2

Architecture: https://wiki.postgresql.org/wiki/PGXN_v2/Architecture

Project: https://github.com/orgs/pgxn/projects/1

Tasks: https://github.com/pgxn/planning/issues

Writing: https://justatheory.com/tags/pgxn/

Slack: https://pgtreats.info/slack-invite

- So that’s the vision. We’d love to have you join the effort. Feedback and ideas, in particular, are greatly appreciated.
- The home page for the project is in the Postgres Wiki
- As is a document that goes into much more detail on the proposed high-level architecture
- The project to build all this is managed in a PGXN GitHub project
- As are the tasks to be carried out for each service, tool, and feature
- I’ve also blogged extensively on the project since the beginning of the year at justatheory.com.
- And the best place to stay in the loop and get in involved is the #extensions channel on the community Slack, which you can join here.

I hope to see you there! Thanks for tuning in!