Postgres in a Microservices World

BRUCE MOMJIAN

This presentation explains the value of microservices to modern organizations and how Postgres can enhance such architectures.

https://momjian.us/presentations

Last updated: February 2025

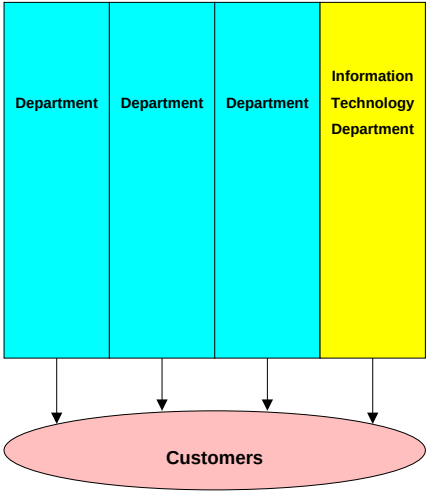
Outline

1. Changing role of information technology
2. Why microservices?
3. What are microservices?
4. How to organize persistent data for microservices?
5. Problems of per-service data
   - 5.1 call amplification
   - 5.2 cross-service joins
   - 5.3 writing across microservices
6. Postgres and microservices
7. Tying services together

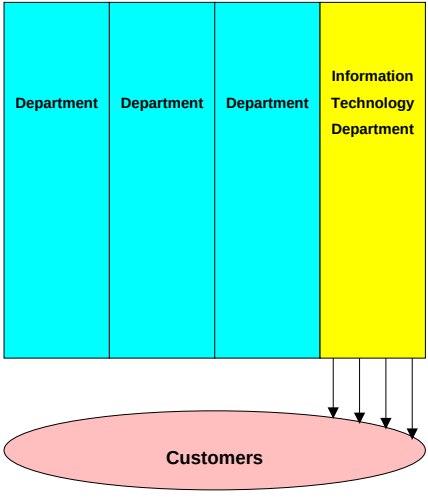
1. Changing Role of Information Technology

https://www.flickr.com/photos/flex/

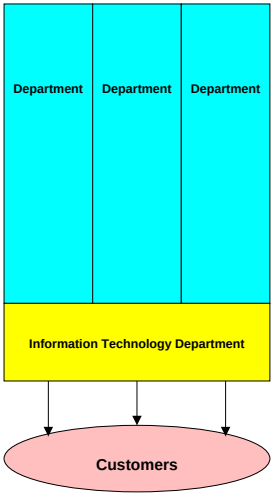
Information Technology (IT) in a Supporting Role



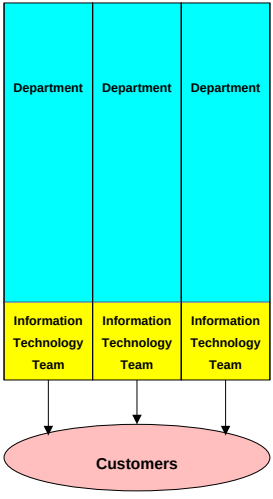
IT as the Customer-Facing Role



IT Required for Departmental Success



Departmental IT



Splintering IT Trend

- Customer communication increasingly digital
- Customer digital interaction increasingly important
- IT now critical, not just in a supporting role
- IT is increasingly nimble and less costly

2. Why Microservices?

https://www.flickr.com/photos/zaffi/

Monolithic and modular applications bad

Microservices good

~~Monolithic and modular applications bad~~

~~Microservices good~~

Microservice Benefits

Microservices can enable

- Alignment of product teams with business goals *(see previous section)*
  - domain-driven design (DDD)
- Improved software development using smaller teams
  - Agile
  - DevOps
- More frequent software deployment
  - continuous integration and continuous delivery (CICD)
- Easier unit testing
- More flexible scaling
- Customized programming languages and databases
- Improved reliability
  - event-driven architecture (EDA)

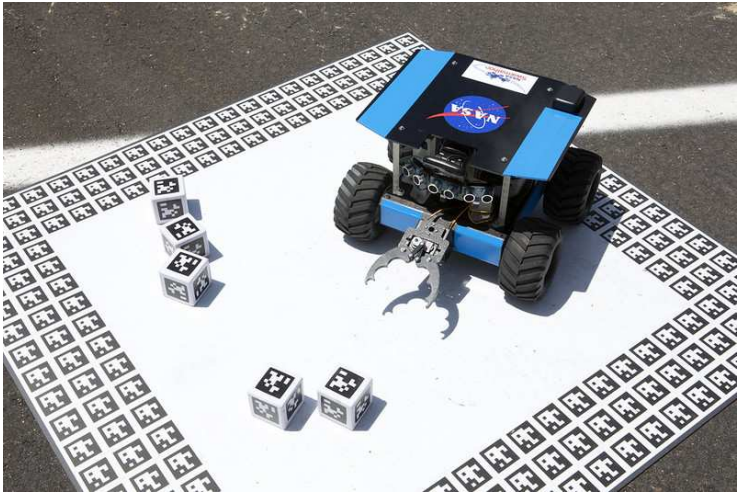
but at a cost. *Win developer bingo:* $a^2c^2d^6e^3gi^2lv$*-ops*☺

https://blog.dreamfactory.com/7-key-benefits-of-microservices/

https://medium.datadriveninvestor.com/what-are-the-benefits-of-microservices-and-will-they-pay-for-your-business-7b23f57c2234

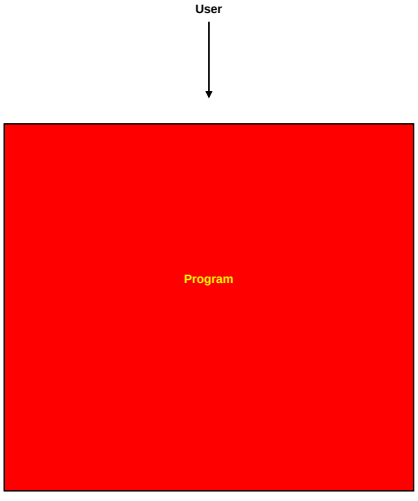
https://www.youtube.com/watch?v=wgdBVIX9ifA

3. What Are Microservices?

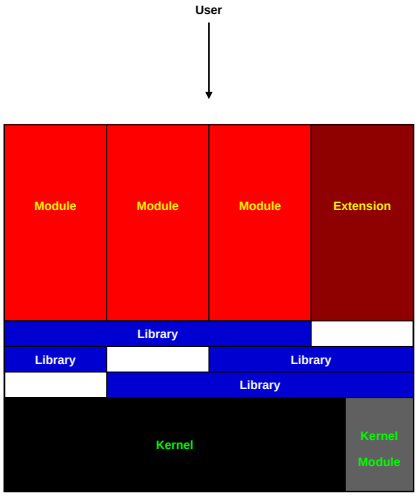


https://www.flickr.com/photos/nasakennedy/

Monolithic Architecture

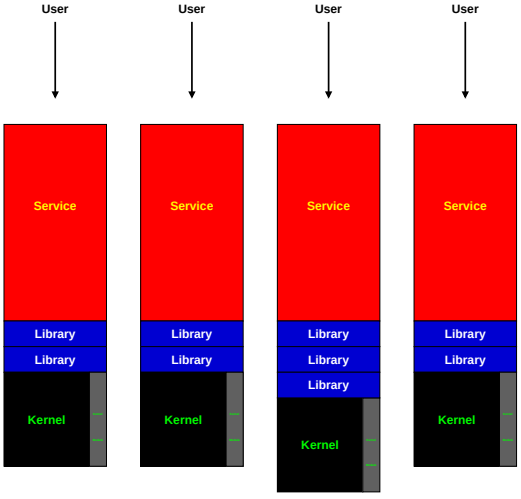


Modular Architecture



Module communication is done via function calls in the same address space.

Microservice Architecture



Service calls are done via a network request, which is more likely than function calls to fail or be delayed. For containers, the kernel is virtually private.

Architectural Differences

Monolithic: application in a single executable

Modular: application relies on libraries, kernel services, and external services; modules communicate by calling functions in the same address space

Microservices: application split into multiple executables in dedicated address spaces; all service communications are via network requests

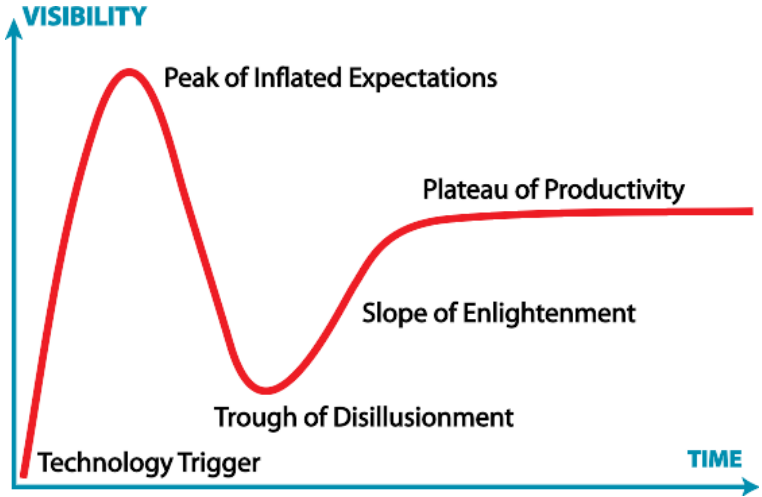
Microservices Are Part of a Larger Modularization Trend

- Monolithic kernels to modular kernels
  - VMS to Unix
  - modules can be added/removed from kernels without recompiling
- Command-line tools with many options to Unix command piping
  - traditional VMS/MS Windows command line vs. bash (and PowerShell)
- Application library usage
  - avoid copying code between applications by using custom libraries
  - avoid recompiling after library changes by using dynamically-linking libraries
  - use externally-developed libraries, often installed by package managers
- Use network services for complex tasks
  - SAN for storage
  - DNS for domain-name lookup
  - SMTP for email
  - HTTPS for website and REST
  - database

Modularization to the Extreme?

- Every kernel function a module?
  - microkernel
- Minimal command-line options?
  - no `ls -r`? Use `sort -r`
- Every function in its own library or package?
- Use network services for simple tasks
  - all function calls become network service calls

Hype Cycle



http://en.wikipedia.org/wiki/Hype_cycle

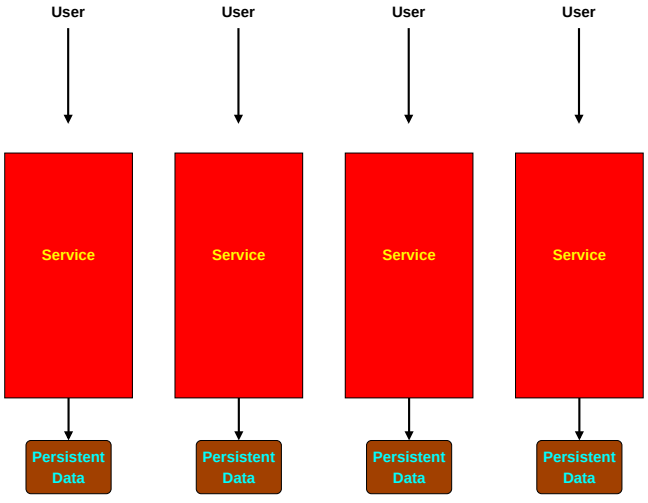
Appropriate Modularization, Appropriate Microservices

Software engineers are used to implementing appropriate modularization. Microservices also require appropriate deployment. Microservices are not a goal, like modularization is not a goal.

4. How To Organize Persistent Data for Microservices

https://www.flickr.com/photos/wikimediacommons/

Microservices with Per-Service Data



Benefits of Per-Service Data

Microservices can enable

- Alignment of product teams with business goals
- Improved software development using smaller teams
  - **team fully controls its data**
- More frequent software deployment
  - **no need to coordinate schema changes with other teams during deployment**
- Easier unit testing
- More flexible scaling
  - **data stores are smaller and can be scaled independently**
- Customized programming languages and databases
  - **team chooses its ideal language and data store**
- Improved reliability
  - **unavailability of one service does not affect other services, no central database**

but at a cost.

5. Problems of Per-Service Data

https://www.flickr.com/photos/dan_e/

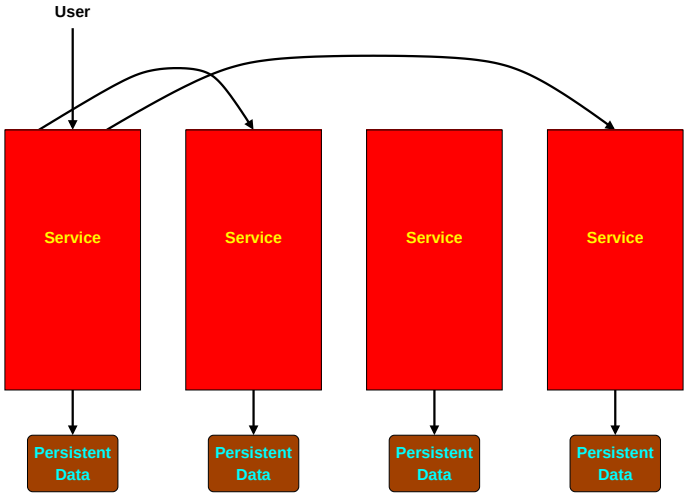
Problems of Per-Service Data

- Services are isolated
- Inter-service interactions are only through predefined APIs
- How can cross-service data needs be met?
- Anti-relational?

5.1 Call Amplification

https://www.flickr.com/photos/martin_heigan/

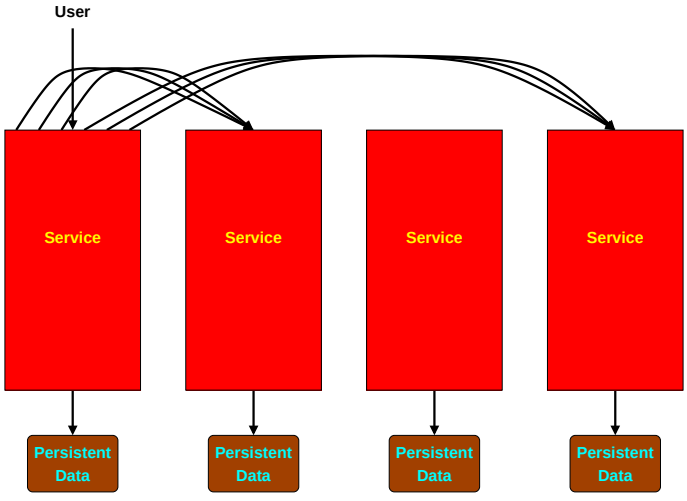
Single Read Per Request



For example, look up the customer, their shipping address, and credit limit.

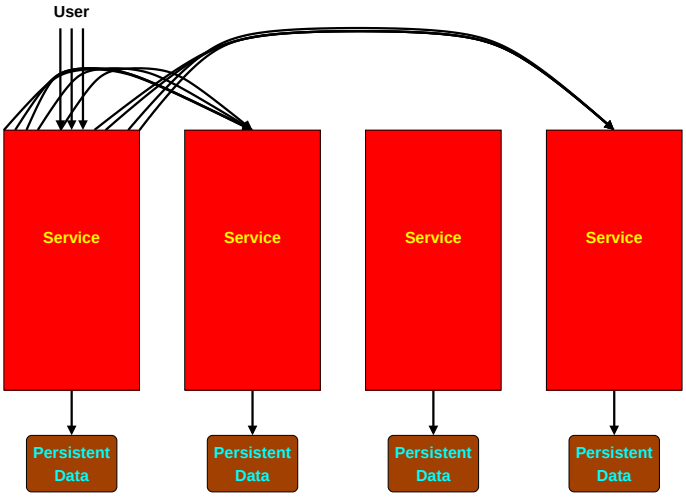
https://www.youtube.com/watch?v=E8-e-3fRHBw

Multiple Reads Per Request



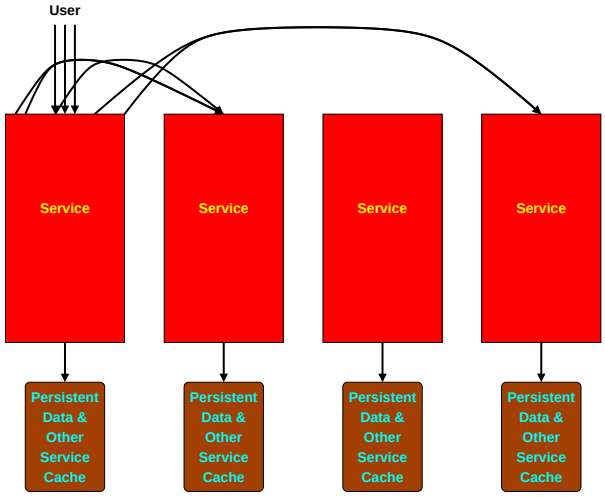
For example, look up the customer, their orders and shipments.

Multiple Requests with Multiple Reads



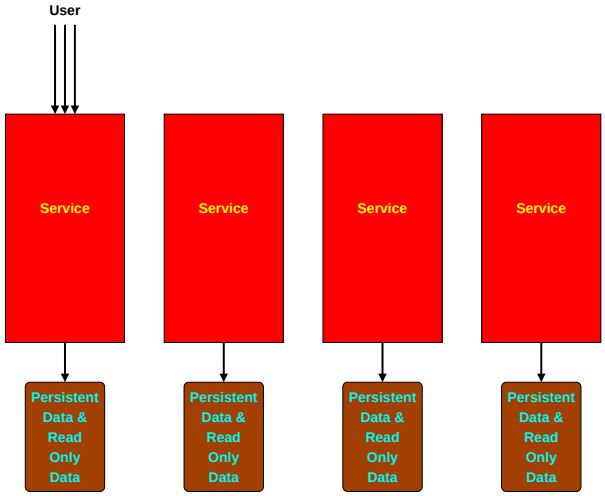
For example, look up all customers who have ordered in the past week, their orders and shipments.

Fixing the Microservice Call Amplification with Caches



The cache is populated via service calls or event subscriptions; might need adjustment for data schema changes.

Fixing with Read-Only Copies

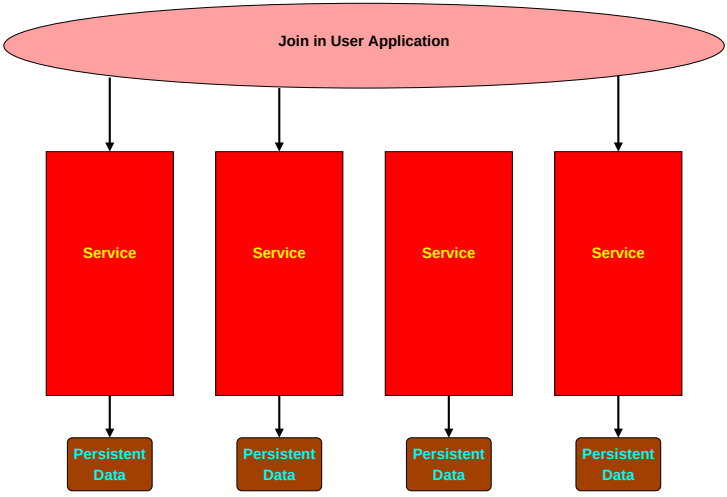


The read-only copies can be from microservice calls, an event stream, or a central data store. Schema changes might require external team coordination.

5.2 Cross-Service Joins

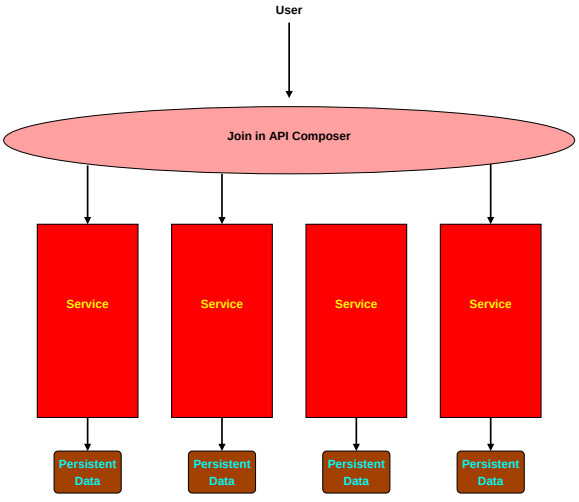
https://www.flickr.com/photos/19779889@N00/

Join in User Applications



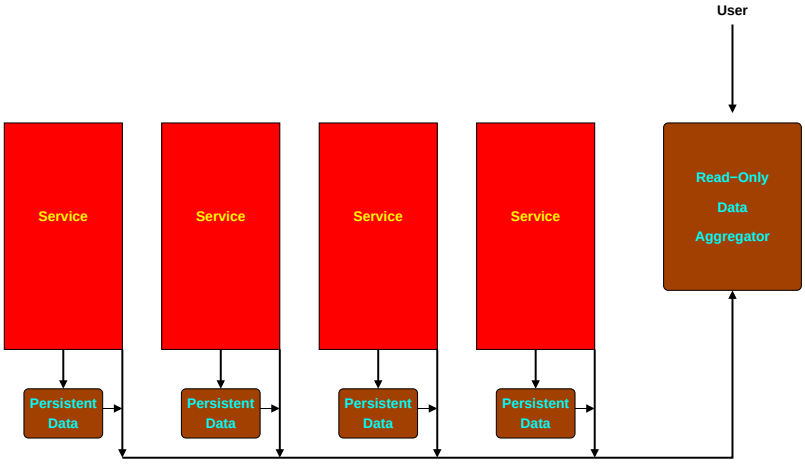
The application probably does a nested-loop join with inner index scan. It does not have row counts or column selectivity estimates like the optimizer of a relational database.

Join in API Composer



API composer joins have similar limitations to joins in user applications.

Join Using a Data Aggregator

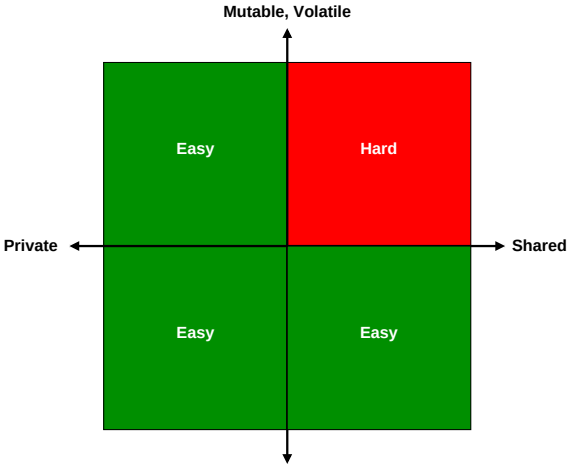


Publication of service events or database replication can be used to populate the data aggregator. The data can be stored as materialized views for specific queries, or ad hoc queried if using a relational database. Separating reads and writes is sometimes called Command Query Responsibility Segregation (CQRS).

5.3 Writing Across Services

https://www.flickr.com/photos/pellesten/

Shared Mutable State Is Hard



Mutable data stored in multiple places requires locking and synchronization, making sharing difficult.

https://momjian.us/main/blogs/pgblog/2021.html#June_16_2021

Decoupled Atomic Commits

- Multiple per-service writable copies of data would be too hard to lock and synchronize in a decoupled way
- Two-phase commit would also couple microservices too closely

6. Postgres and Microservices

https://www.flickr.com/photos/lapine403/

Adjust Relational Systems

- Transaction isolation level
- Durability per
  - transaction
  - table
  - server
  - streaming replica
- Normalization

https://www.youtube.com/watch?v=tRteUhd9eAI

Microservices: A Dial

> *Don't think of microservices as flipping a switch; think about it as turning a dial.*

Monolith to Microservices, Sam Newman

https://www.youtube.com/watch?v=GBTdnfD6s5Q

https://www.youtube.com/watch?v=MjIfWe6bn4O

https://www.flickr.com/photos/61629877@N04/

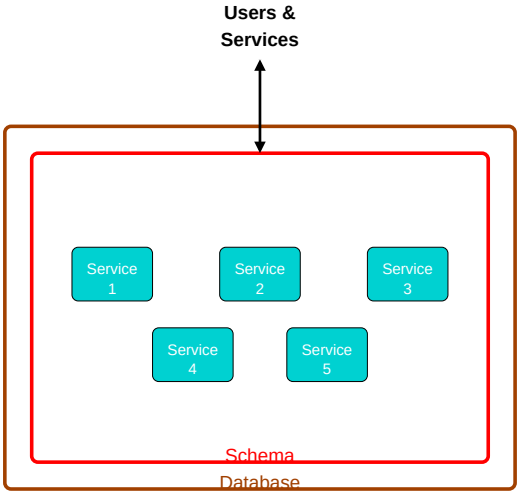
Data Encapsulation

Data encapsulation allows data to be hidden inside a class or module — data interaction from outside the class/module is controlled by access functions (“getters” and “setters”) or other methods:

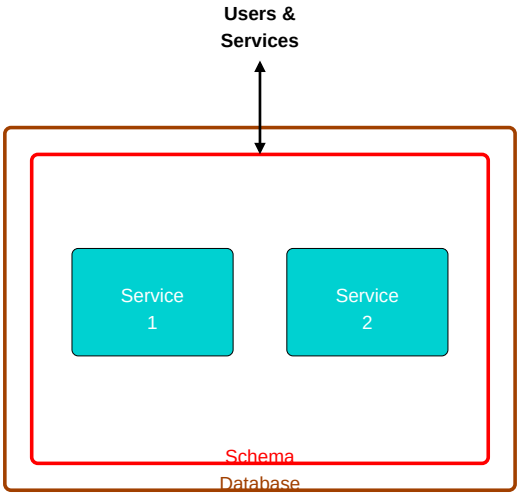
```
public class Employee {
    private BigDecimal salary = new BigDecimal(0);

   public BigDecimal setSalary() {
    }
…
```

https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)

Monolithic Data Layout



Monolithic Data Layout



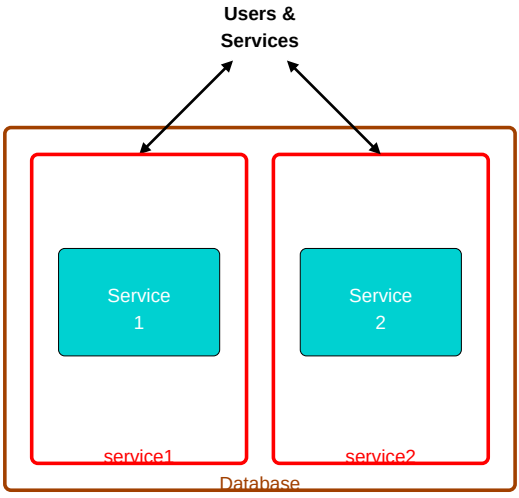
Only two services

Monolithic Data Layout

```
-- create role
CREATE ROLE service;

-- create schema
CREATE SCHEMA service;
ALTER SCHEMA service OWNER TO service;

-- become 'service' user
SET SESSION AUTHORIZATION service;

-- create table
CREATE TABLE service.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- all services have full access
```

All queries used in this presentation are available at https://momjian.us/main/writings/pgsql/microservices.sql.

Per-Service Schema



Logical isolation of services

Per-Service Schema

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create per-service schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Per-Service Schema

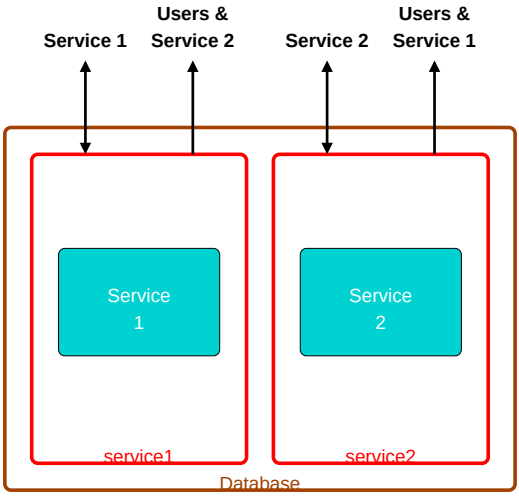
```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create table
CREATE TABLE service1.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- give all services full access
GRANT ALL ON SCHEMA service1 TO service;
GRANT ALL ON service1.employee TO service;
```

Benefit of Per-Service Schema

Microservices can enable

- **Alignment of product teams with business goals**
- Improved software development using smaller teams
- More frequent software deployment
- Easier unit testing
- More flexible scaling
- Customized programming languages and databases
- Improved reliability

Read-Only for Non-Owners



Authorization isolation of services

Read-Only for Non-Owner

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create per-service schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Read-Only for Non-Owner

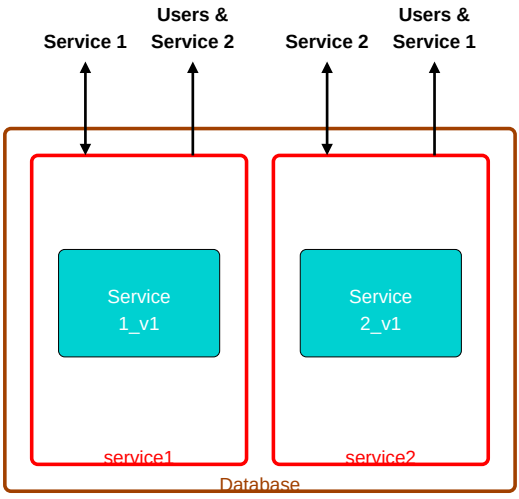
```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create table
CREATE TABLE service1.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- give other services read-only access
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee TO service;
```

Benefit of Read-Only for Non-Owner

Microservices can enable

- Alignment of product teams with business goals
- **Improved software development using smaller teams**
- More frequent software deployment
- Easier unit testing
- More flexible scaling
- Customized programming languages and databases
- Improved reliability

Versioning



Decoupling of table schema changes; schema names can be versioned instead

Versioning

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create per-service schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Versioning

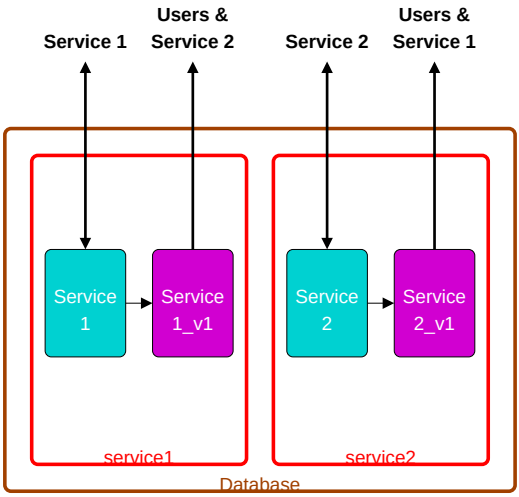
```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create table
CREATE TABLE service1.employee_v1 (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- give other services read-only access
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Benefit of Versioning

Microservices can enable

- Alignment of product teams with business goals
- Improved software development using smaller teams
- **More frequent software deployment**
- Easier unit testing
- More flexible scaling
- Customized programming languages and databases
- Improved reliability

View



Views decouple the private and public table schemas. We could also output JSON using a function API.

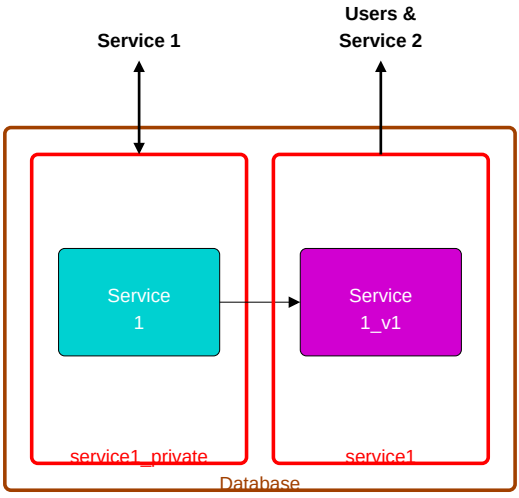
View

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create per-service schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

View

```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create table
CREATE TABLE service1.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- create view
CREATE VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Public Schema



Only one service shown, separate public schema hides private implementation

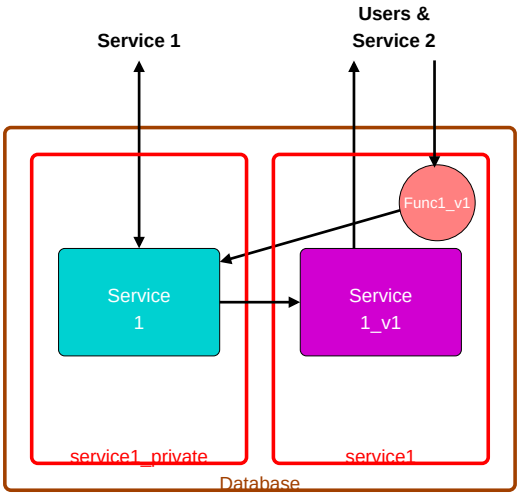
Public Schema

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create private schema
CREATE SCHEMA service1_private;
ALTER SCHEMA service1_private OWNER TO service1;

-- create public schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Public Schema

```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create private table
CREATE TABLE service1_private.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- create view
CREATE VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1_private.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Setter Function



Setter function is a public API for writes

Setter Function

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create private schema
CREATE SCHEMA service1_private;
ALTER SCHEMA service1_private OWNER TO service1;

-- create public schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Setter Function

```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create private table
CREATE TABLE service1_private.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- create view
CREATE VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1_private.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Setter Function

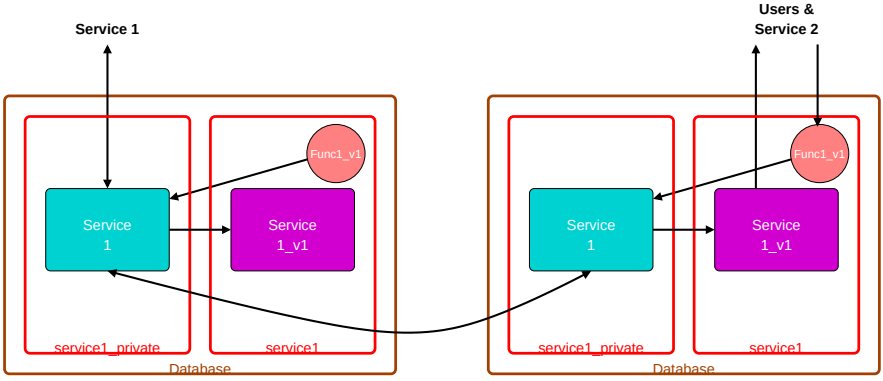
```
-- create setter function for salary
-- use a transaction block so PUBLIC permission is never visible
BEGIN WORK;
CREATE FUNCTION service1.set_employee_salary_v1
        (emp_id service1_private.employee.emp_id%TYPE,
         new_salary service1_private.employee.salary%TYPE)
        RETURNS VOID
AS $$
        UPDATE service1_private.employee
        SET salary = new_salary
        WHERE service1_private.employee.emp_id = emp_id
$$
LANGUAGE sql SECURITY DEFINER;
        return this.salary;

GRANT EXECUTE ON FUNCTION service1.set_employee_salary_v1 TO service;
REVOKE EXECUTE ON FUNCTION service1.set_employee_salary_v1 FROM PUBLIC;
COMMIT WORK;
```

Benefit of Setter Function

Microservices can enable

- Alignment of product teams with business goals
- Improved software development using smaller teams
- More frequent software deployment
- **Easier unit testing**
- More flexible scaling
- Customized programming languages and databases
- Improved reliability

Foreign Data Wrapper (FDW)



Using FDWs, services can now be on different hosts

Foreign Data Wrapper: Private

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create private schema
CREATE SCHEMA service1_private;
ALTER SCHEMA service1_private OWNER TO service1;

-- create public schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Foreign Data Wrapper: Private

```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create private table
CREATE TABLE service1_private.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- create view
CREATE VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1_private.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Foreign Data Wrapper: Private

```
-- create setter function for salary
-- use a transaction block so PUBLIC permission is never visible
BEGIN WORK;
CREATE FUNCTION service1.set_employee_salary_v1
        (emp_id service1_private.employee.emp_id%TYPE,
         new_salary service1_private.employee.salary%TYPE)
        RETURNS VOID
AS $$
        UPDATE service1_private.employee
        SET salary = new_salary
        WHERE service1_private.employee.emp_id = emp_id
$$
LANGUAGE sql SECURITY DEFINER;

GRANT EXECUTE ON FUNCTION service1.set_employee_salary_v1 TO service;
REVOKE EXECUTE ON FUNCTION service1.set_employee_salary_v1 FROM PUBLIC;
COMMIT WORK;
```

Foreign Data Wrapper: Public

```
-- connect to service2 database server
\connect - - - 5433

-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create private schema
CREATE SCHEMA service1_private;
ALTER SCHEMA service1_private OWNER TO service1;

-- create public schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Foreign Data Wrapper: Public

```
CREATE SERVER postgres_fdw_service1
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', port '5432', dbname 'test');

CREATE USER MAPPING FOR PUBLIC
SERVER postgres_fdw_service1
OPTIONS (user 'service1', password '');

IMPORT FOREIGN SCHEMA service1_private LIMIT TO (employee)
FROM SERVER postgres_fdw_service1 INTO service1_private;

-- create view
CREATE VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1_private.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Foreign Data Wrapper: Public

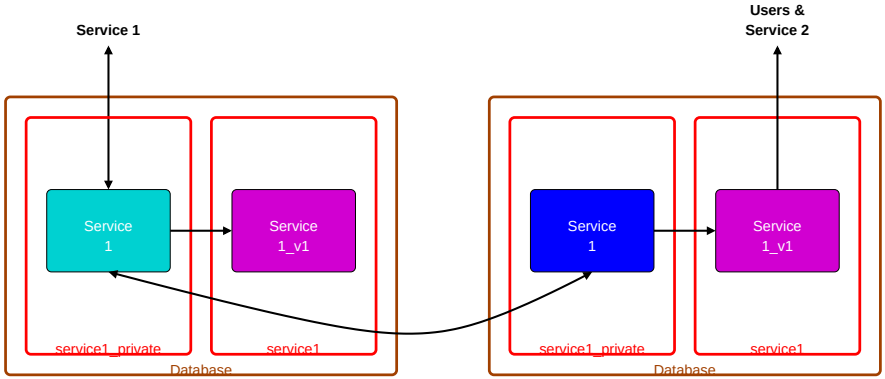
```
-- create setter function for salary
-- use a transaction block so PUBLIC permission is never visible
BEGIN WORK;
CREATE FUNCTION service1.set_employee_salary_v1
        (emp_id service1_private.employee.emp_id%TYPE,
         new_salary service1_private.employee.salary%TYPE)
        RETURNS VOID
AS $$
        UPDATE service1_private.employee
        SET salary = new_salary
        WHERE service1_private.employee.emp_id = emp_id
$$
LANGUAGE sql SECURITY DEFINER;

GRANT EXECUTE ON FUNCTION service1.set_employee_salary_v1 TO service;
REVOKE EXECUTE ON FUNCTION service1.set_employee_salary_v1 FROM PUBLIC;
COMMIT WORK;
```

Benefits of Foreign Data Wrapper

Microservices can enable

- Alignment of product teams with business goals
- Improved software development using smaller teams
- More frequent software deployment
- Easier unit testing
- **More flexible scaling**
- **Customized programming languages and databases**
- Improved reliability

Materialized FDW View



Materialized FDW views decouple the effects of cross-service downtime, but prevent cross-service writes.

Materialized FDW View: Private

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create private schema
CREATE SCHEMA service1_private;
ALTER SCHEMA service1_private OWNER TO service1;

-- create public schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Materialized FDW View: Private

```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create private table
CREATE TABLE service1_private.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- create view
CREATE VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1_private.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Materialized FDW View: Public

```
-- connect to service2 database server
\connect - - - 5433

-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN IN ROLE service;
CREATE ROLE service2 WITH LOGIN IN ROLE service;

-- create private schema
CREATE SCHEMA service1_private;
ALTER SCHEMA service1_private OWNER TO service1;

-- create public schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Materialized FDW View: Public

```
CREATE SERVER postgres_fdw_service1
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', port '5432', dbname 'test');

CREATE USER MAPPING FOR PUBLIC
SERVER postgres_fdw_service1
OPTIONS (user 'service1', password '');

IMPORT FOREIGN SCHEMA service1_private LIMIT TO (employee)
FROM SERVER postgres_fdw_service1 INTO service1_private;
```

Materialized FDW View: Public

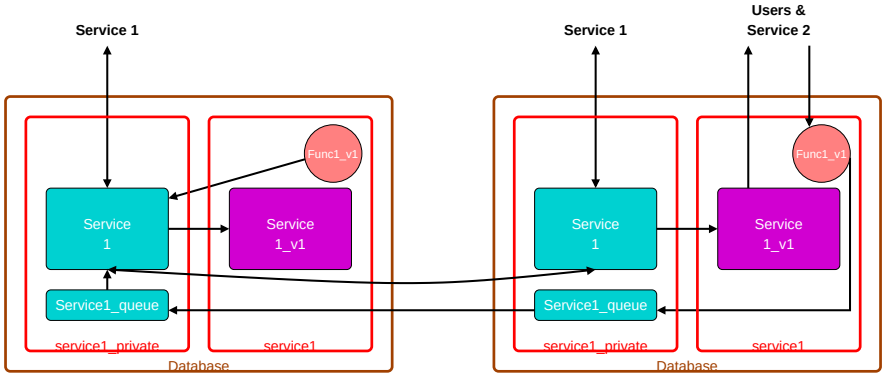
```
-- create materialized view
CREATE MATERIALIZED VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1_private.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;

-- needed for refresh
CREATE UNIQUE INDEX i_employee_v1 ON service1.employee_v1 (emp_id);

REFRESH MATERIALIZED VIEW CONCURRENTLY service1.employee_v1;
```

Benefit of Materialized FDW View

Microservices can enable

- Alignment of product teams with business goals
- Improved software development using smaller teams
- More frequent software deployment
- Easier unit testing
- More flexible scaling
- Customized programming languages and databases
- **Improved reliability**

Logical Replication



Allows access more current data, while also allowing access during disconnection from other services

Logical Replication: Private

```
-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN REPLICATION IN ROLE service;
CREATE ROLE service2 WITH LOGIN REPLICATION IN ROLE service;

-- create private schema
CREATE SCHEMA service1_private;
ALTER SCHEMA service1_private OWNER TO service1;

-- create public schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Logical Replication: Private

```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create private employee table
CREATE TABLE service1_private.employee (
        emp_id SERIAL PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- create read-only queue table
CREATE TABLE service1_private.queue (
        emp_id INTEGER,
        new_salary numeric(10,2),
        entry_date TIMESTAMP WITH TIME ZONE
);
```

Logical Replication: Private

```
-- create view
CREATE VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1_private.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Logical Replication: Private

```
-- create setter function for salary
-- use a transaction block so PUBLIC permission is never visible
BEGIN WORK;
CREATE FUNCTION service1.set_employee_salary_v1
        (emp_id service1_private.employee.emp_id%TYPE,
         new_salary service1_private.employee.salary%TYPE)
        RETURNS VOID
AS $$
        INSERT INTO service1_private.queue VALUES (emp_id, new_salary, CURRENT_TIMESTAMP);
$$
LANGUAGE sql SECURITY DEFINER;

GRANT EXECUTE ON FUNCTION service1.set_employee_salary_v1 TO service;
REVOKE EXECUTE ON FUNCTION service1.set_employee_salary_v1 FROM PUBLIC;
COMMIT WORK;
```

Logical Replication: Private

```
CREATE FUNCTION service1_private.queue_trigger_func()
        RETURNS TRIGGER
AS $$
        BEGIN
        UPDATE service1_private.employee
        SET salary = new.new_salary
        WHERE service1_private.employee.emp_id = new.emp_id;
        RETURN new;
        END
$$
LANGUAGE plpgsql;
```

Logical Replication: Private

```
CREATE TRIGGER queue_trigger
BEFORE INSERT ON service1_private.queue
FOR EACH ROW
EXECUTE FUNCTION service1_private.queue_trigger_func();

-- force trigger to run on replica
ALTER TABLE service1_private.queue ENABLE ALWAYS TRIGGER queue_trigger;

-- become superuser
RESET SESSION AUTHORIZATION;

-- create publication
CREATE PUBLICATION pub_employee FOR TABLE service1_private.employee;
```

Logical Replication: Public

```
-- connect to service2 database server
\connect - - - 5433

-- create roles
CREATE ROLE service WITH NOLOGIN;

CREATE ROLE service1 WITH LOGIN REPLICATION IN ROLE service;
CREATE ROLE service2 WITH LOGIN REPLICATION IN ROLE service;

-- create private schema
CREATE SCHEMA service1_private;
ALTER SCHEMA service1_private OWNER TO service1;

-- create public schema
CREATE SCHEMA service1;
ALTER SCHEMA service1 OWNER TO service1;
```

Logical Replication: Public

```
-- become 'service1' user
SET SESSION AUTHORIZATION service1;

-- create read-only employee table
CREATE TABLE service1_private.employee (
        emp_id INTEGER PRIMARY KEY,
        name TEXT UNIQUE NOT NULL,
        salary numeric(10,2)
);

-- create write-only queue table
CREATE TABLE service1_private.queue (
        emp_id INTEGER,
        new_salary numeric(10,2),
        entry_date TIMESTAMP WITH TIME ZONE
);
```

Logical Replication: Public

```
-- create view
CREATE VIEW service1.employee_v1 AS
        SELECT emp_id, name FROM service1_private.employee;

-- give other services read-only access to two columns
GRANT USAGE ON SCHEMA service1 TO service;
GRANT SELECT ON service1.employee_v1 TO service;
```

Logical Replication: Public

```
-- create setter function for salary
-- use a transaction block so PUBLIC permission is never visible
BEGIN WORK;
CREATE FUNCTION service1.set_employee_salary_v1
        (emp_id service1_private.employee.emp_id%TYPE,
         new_salary service1_private.employee.salary%TYPE)
        RETURNS VOID
AS $$
        INSERT INTO service1_private.queue VALUES (emp_id, new_salary, CURRENT_TIMESTAMP);
$$
LANGUAGE sql SECURITY DEFINER;

GRANT EXECUTE ON FUNCTION service1.set_employee_salary_v1 TO service;
REVOKE EXECUTE ON FUNCTION service1.set_employee_salary_v1 FROM PUBLIC;
COMMIT WORK;
```

Logical Replication: Public

```
-- become superuser
RESET SESSION AUTHORIZATION;

-- create publication
CREATE PUBLICATION pub_queue FOR TABLE service1_private.queue;

-- create subscription
CREATE SUBSCRIPTION sub_service1_private_employee
        CONNECTION 'dbname=test host=localhost port=5432 user=service1'
        PUBLICATION pub_employee;

-- connect to service1 database server
\connect - - - 5432

-- create subscription
CREATE SUBSCRIPTION sub_service1_private_queue
        CONNECTION 'dbname=test host=localhost port=5433 user=service1'
        PUBLICATION pub_queue;
```

Logical Replication: Example

```
INSERT INTO service1_private.employee
        VALUES (DEFAULT, 'Sam', 100);

SELECT * FROM service1_private.employee;
 emp_id | name | salary
--------+------+--------
      1 | Sam  | 100.00

SELECT service1.set_employee_salary_v1(1, 200);
 set_employee_salary_v1
------------------------
 (null)
```

Logical Replication: Example

```
-- the queue
SELECT * FROM service1_private.queue;
 emp_id | new_salary |          entry_date
--------+------------+-------------------------------
      1 |     200.00 | 2022-02-22 13:10:01.213059-05

SELECT * FROM service1_private.employee;
 emp_id | name | salary
--------+------+--------
      1 | Sam  | 200.00
```

Logical Replication: Example

```
-- connect to service2
\connect - - - 5433

SELECT * FROM service1_private.employee;
 emp_id | name | salary
--------+------+--------
      1 | Sam  | 200.00
```

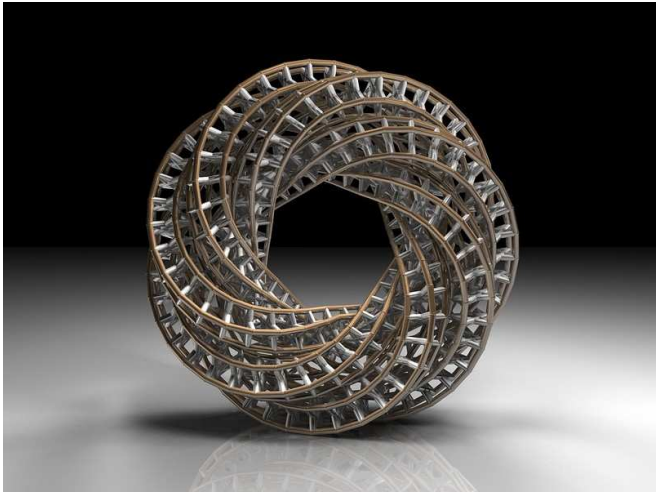
Logical Replication: Example

```
-- using public API
SELECT * FROM service1.employee_v1;
 emp_id | name
--------+------
      1 | Sam

SELECT service1.set_employee_salary_v1(1, 300);
 set_employee_salary_v1
------------------------
 (null)

SELECT * FROM service1_private.queue;
 emp_id | new_salary |          entry_date
--------+------------+-------------------------------
      1 |     300.00 | 2022-02-22 13:10:40.706198-05

SELECT * FROM service1_private.employee;
 emp_id | name | salary
--------+------+--------
      1 | Sam  | 300.00
```

Complexities

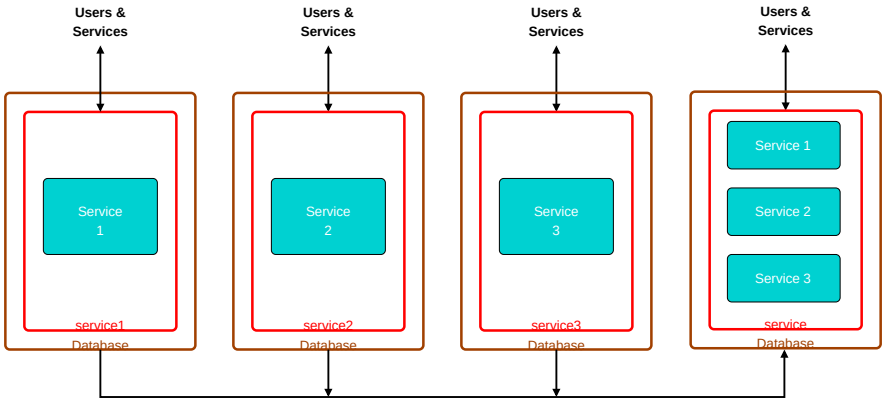
- The service's external API can be table-based or function-based
  - I chose reads as table-based, and writes as function-based
- Materialized views need periodic refreshes; data too stale to allow a write API
- Changes to table schemas will require possible recreation of logical replicas and materialized views
- Multi-database deployments need the ability to update public APIs in all databases
- Remove service queue entries can be added in the same transaction as local service changes
  - however, conflicting service changes might be overwritten
- Old queue table rows will need removed occasionally by each service user
  - having *entry_date* helps with this

7. Tying Microservices Together



https://www.flickr.com/photos/fdecomite/

Data Aggregation



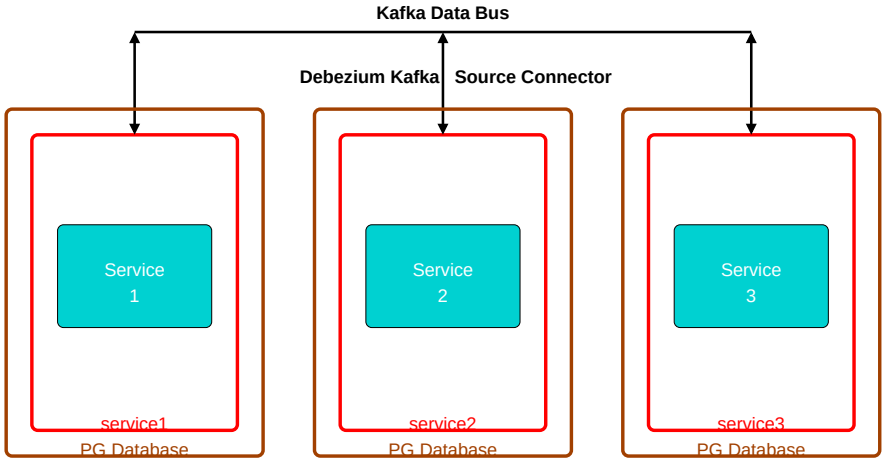
Service data is aggregated in the right server; can be used to perform cross-service joins and data analysis.

Deploying Schema Changes

- Flyway
- Liquibase

https://moduscreate.com/blog/microservices-databases-migrations/

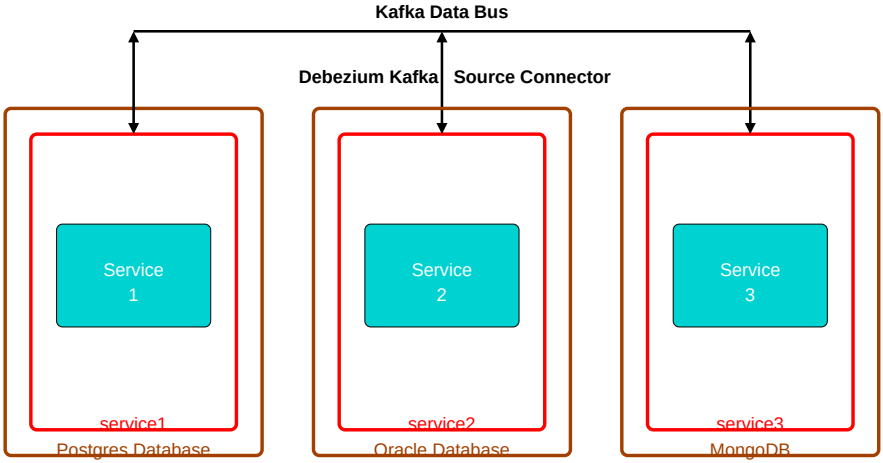
Debezium Simplifies Adding/Removing Data-Subscribing Services



Debezium reads the Postgres logical replication log to broadcast changes to other services.

https://debezium.io/documentation/reference/stable/connectors/postgresql.html

Debezium Can Communicate with Different Database Types



https://debezium.io/documentation/reference/stable/architecture.html

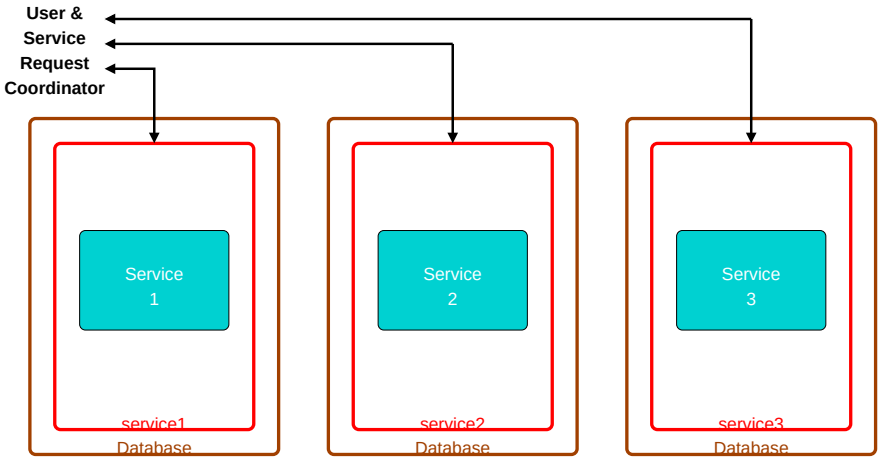
Saga Pattern

The saga pattern is like a transaction manager, but is asynchronous with optimistic commit and undo. Saga patterns create a chain of events to perform multi-service requests. If any event fails, a set of reversing events undoes the request.

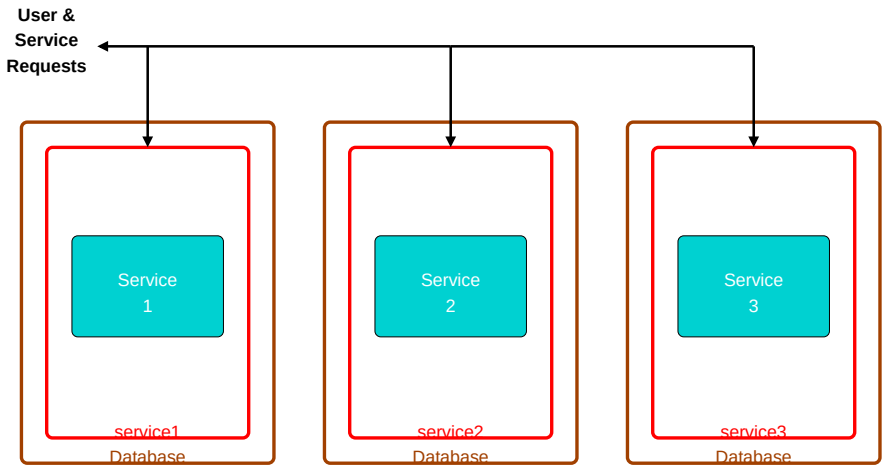
https://www.baeldung.com/cs/saga-pattern-microservices

https://www.youtube.com/watch?v=YPbGW3Fnmbc
https://www.youtube.com/watch?v=STKCRSUsyP0

Saga Orchestration



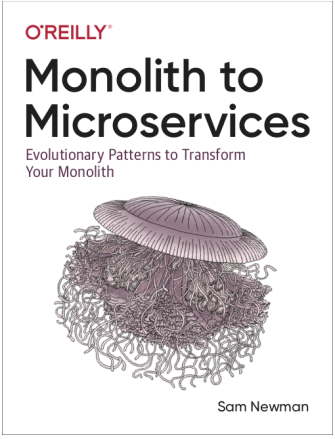
Saga Choreography, Event-Driven



Choreography requests are sent on an event bus and services take action on appropriate messages.

Monolith to Microservices by Sam Newman

1. Just Enough Microservices
2. Planning A Migration
3. Splitting The Monolith
4. **Decomposing The Database (81 pages)**
5. Growing Pains
6. Closing Words



https://samnewman.io/books/monolith-to-microservices/

Conclusion

https://momjian.us/presentations

https://www.flickr.com/photos/146651768@N06/