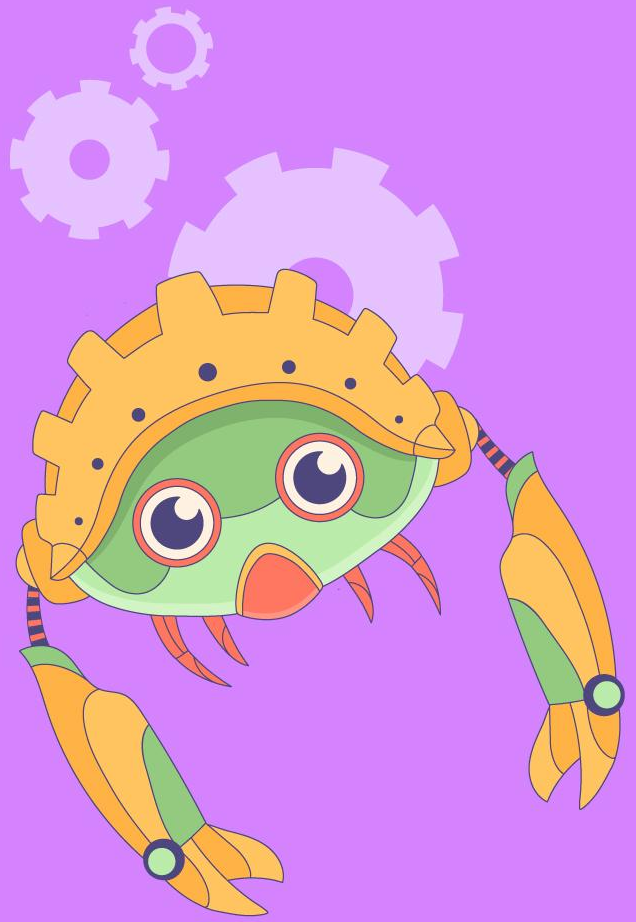
Emilio Junior Francischetti • Angelo Rendina

FROM SIMPLE OBJECT TO FEDERATION



About us

I am passionate about software development and automation.

I love to take big monoliths, super coupled, clearly without tests and then transform them into cloud-native applications in TDD, DDD, CQRS, and Event Sourcing - obviously on Kubernetes.

I therefore define myself as a Software Restorer.



Emilio Junior Francischetti, Software Restorer

About us

With a background in pure mathematics and experience as a senior software engineer, I feel right at home with Rust and its expressive type system.

I am product-focused, and equally passionate about building systems that are both highly performant and correct.



Angelo Rendina, Senior Software Engineer

Index

- Why GraphQL
- The client view
- The type system
- The server view
- The Rust stack
- Federation
- Strangler fig pattern



An example

A store frontend page, showing details about an order

```
Order
     id: 42
     status: PLACED
     total: 57.90

Customer
     email: alice@email.it
     phone: 123456789

Products
1)   Mug
          id: 1
          amount: 2
2)   Teapot
          id: 2
          amount: 1
```

In REST

- Underfetching

Need to make multiple API calls to populate the entire page

- Overfetching

Each API call returns more information than needed

Tip: a possible solution is an ad-hoc API exposed by a BE for FE

```
GET /orders/42
  → { id, status, total, customerId }

GET /customers/7
  → { id, name, email, vat, phone, ... }

GET /orders/42/lines
  → [ { productId, quantity }, ... ]

GET /products/1

GET /products/2

...
```

In GraphQL

```
query OrderPage {
  order(id: "42") {
    status
    total
    customer { name }
    lines {
      quantity
      product { name price }
    }
  }
}
```

```
{
  "data": {
    "order": {
      "status": "PLACED",
      "total": "57.90",
      "customer": { "name": "Ada" },
      "lines": [
        { "quantity": 2,
          "product": { "name": "Mug",
                       "price": "9.95" } },
        { "quantity": 1,
          "product": { "name": "Teapot",
                       "price": "38.00" } }
      ]
    }
  }
}
```

A first look at the schema

Contract between clients and server

Declares what operations are possible, and on what data they act

```
type Query {
  order(id: ID!): Order
}

type Mutation {
  purchase(cart: Cart!): Order!
}

type Order {
  id: ID!
  status: OrderStatus!
  total: Money!
}

input Cart {
  products: [ID!]!
}
```

WHAT GRAPHQL IS

- Created at Facebook in 2012
- Open-sourced in 2015
- Governed by the GraphQL Foundation since 2018.

- A query language for APIs
- A runtime that executes queries against a typed schema
- A complement to REST, with its own trade-offs

The client view: operations

- **`Query`**
- **`Mutation`**
- **`Subscription`**



The client view

Queries

- Operations that read state
- Selection set declares requested fields
- Any field can take arguments, at any depth

```
# operation type and name
query OrderRecapPage {
  # field with an argument
  order(id: "42") {
    # leaf field
    status
    # field with a selection set
    lines(limit: "10")  {
      name
    }
  }
}
```

Mutations

- Operations that change state
- Same syntax as queries: the selection set reads back the result

```
mutation {
  cancelOrder(id: "42") {
    id
    status
  }
}
```

Execution order

- Top-level mutation fields run one after the other, in order
- Top-level query fields may run in parallel

```
mutation {
  first: cancelOrder(id: "41") {
    status
  }
  second: shipOrder(id: "42") {
    carrier
    eta
  }
}
```

The schema as a contract

- Lists every type, field and operation the server supports
- Written in SDL (Schema Definition Language) or generated from code
- Every operation is validated against it before execution

Built-in scalars

Atomic basic types

- **`Int`**: signed 32-bit integer
- **`Float`**: double precision
- **`String`**: UTF-8 text
- **`Boolean`**: True or False
- **`ID`**: unique identifier, serialized as a string

Custom scalars

- Domain values with their own parsing and validation
- `@specifiedBy` links to the format specification

```
"ISO8601 datetime with timestamp"
scalar DateTime
@specifiedBy(url: ...)

"Amount with 2 decimal digits"
scalar Money
```

Enums

- A closed set of values
- Validated on input and on output
- Leaves, like scalars

```
enum OrderStatus {
  DRAFT
  PLACED
  SHIPPED
  CANCELLED
}
```

Object types

A named set of fields, each with a type

```
type Order {
  id: ID!
  status: OrderStatus!
  total: Money!
  placedAt: DateTime!
  customer: Customer!
}
```

Nulls and lists

- Fields are nullable by default
- Brackets represent lists
- Lists and nullables can be composed

```
type Employee {
  # never null
  name: String!

  # may be null
  email: String

  # null if not a manager;
  # elements must be non-null
  manages: [Employee!]
}
```

Field arguments

- Arguments are typed
- Default values for omitted arguments

```
type Order {
  lines(
    limit: Int = 10
  ):[OrderLine!]!
}
```

Unions

- A polymorphic type with a closed set of variants
- Tagged by type name, so variants need to be all different

```
union SearchResult
  = Order
  | Customer
```

Querying unions

```graphql
query {
  search(term: "ada") {
    __typename
    ... on Order { id status }
    ... on Customer { name }
  }
}
```

```json
{
  "data": {
    "search": [
      {
        "__typename": "Order",
        "id": "42",
        "status": "SHIPPED"
      },
      {
        "__typename": "Customer",
        "name": "ada"
      }
    ]
  }
}
```

Interfaces

- Defines a set of fields that all implementors must provide
- Open-set polymorphism by declaring types that implement the interface

```
interface Product {
  id: ID!
  name: String!
  price: Money!
}

type PhysicalProduct implements Product {
  id: ID!  name: String!  price: Money!
  weightGrams: Int!
}

type DigitalProduct implements Product {
  id: ID!  name: String!  price: Money!
  downloadUrl: String!
}
```

Querying interfaces

```
query {
  products {
    __typename
    price
    ... on PhysicalProduct {
      weightGrams
    }
    ... on DigitalProduct {
      downloadUrl
    }
  }
}
```

```
{
  "data": {
    "products": [
      {
        "__typename": "PhysicalProduct",
        "price": "24.99",
        "weightGrams": "160"
      },
      {
        "__typename": "DigitalProduct",
        "price": "1.99",
        "downloadUrl": "http://..."
      },
      {
        "__typename": "CustomerSupport",
        "price": "86.99"
      },
    ]
  }
}
```

Input types

- Structured arguments
- Fields can be scalars, enums or other input types
- `@oneOf`: exactly one field must be provided and non-null

```
input PlaceOrderInput {
  customerId: ID!
  lines: [OrderLineInput!]!
}

input OrderLineInput {
  productId: ID!
  quantity: Int! = 1
}

input ProductBy @oneOf {
  id: ID
  code: String
}
```

Root types and directives

```
"schema.graphql"

type Query {
  order(id: ID!): Order
}

type Mutation {
  placeOrder(input: PlaceOrderInput!): PlaceOrderPayload!
}

type Product {
  sku: String @deprecated(reason: "Use `code`")
  code: String!
}
```

The server view

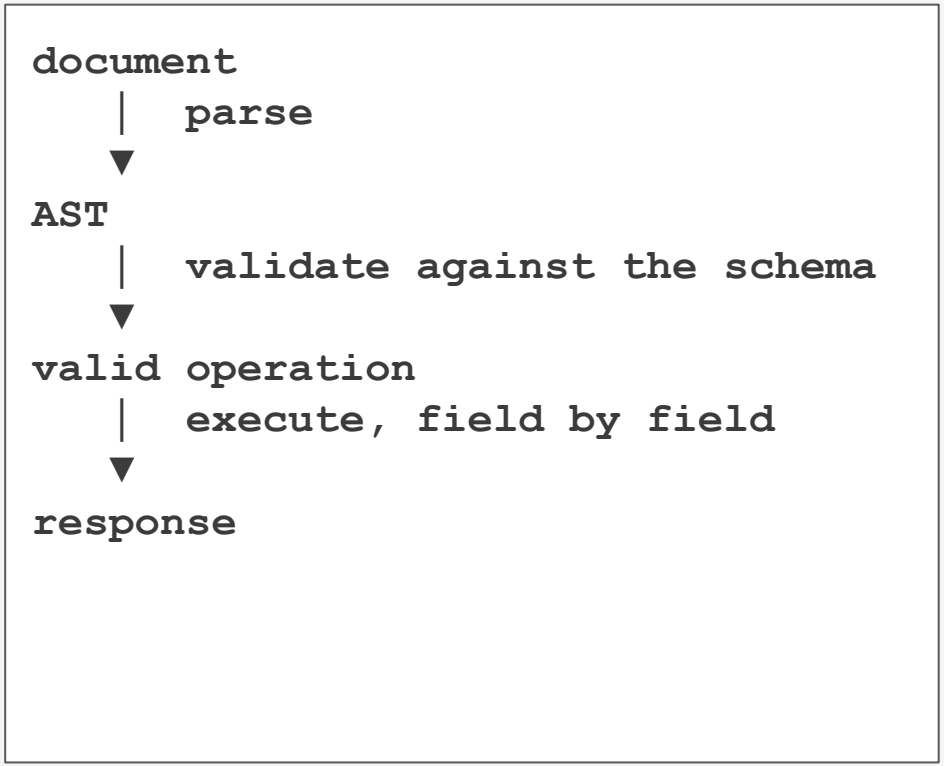
How a server turns an operation and a schema into a response.



The server view

Request lifecycle

- Parse or validation errors stop the request before execution
- Validation is static and can be cached per document



The server view

Validation

Invalid operations are rejected before any application code runs

```
query {
  # missing required argument "id"
  order { status }

  product(by: { code: "MUG" }) {
    name
    # unknown field on Product
    color
  }

  # Money is a scalar: no selection
  products { price { amount } }
}
```

Resolvers

- Every field is backed by a resolver function
- If omitted, a **default** resolver extracts the homonymous field from the parent

```
resolve(
  parent     # value of enclosing field
  args       # arguments of this field
  context    # injected state (DB, auth)
  info       # metadata: path, selection
) -> value | error
```

The resolver tree

```
Query.order(id: "42")          # loads order from db by id
├─ Order.status                # default: reads order.status
├─ Order.total                 # default: reads order.total
├─ Order.customer              # loads customer by order.customer_id
│  └─ Customer.name            # default: reads customer.name
└─ Order.lines                 # loads [line] by order.id
   ├─ OrderLine.quantity       # default: reads line.quantity
   └─ OrderLine.product        # loads product by line.product_id
      ├─ Product.name          # default: reads product.name
      └─ Product.price         # default: reads product.price
```

Same query as the order page. Execution walks the tree; sibling fields may resolve concurrently.

Response shape

- **data**: optional JSON object whose shape matches the selection set
- **errors**: optional list of errors occurred while handling the request
- **extensions**: optional JSON free-form object carrying request metadata

```
{
  "data": { ... },
  "errors": [ ... ],
  "extensions": { ... }
}
```

The server view

Partial results

- A failing field becomes **`null`**
- The rest of the response is still returned
- Possible only because **`tracking`** is nullable

```
{
  "data": {
    "order": {
      "status": "SHIPPED",
      "tracking": null
    }
  },
  "errors": [{
    "message": "Carrier API timed out",
    "path": ["order", "tracking"]
  }]
}
```

The server view

Null propagation

- A non-null field cannot be `null`
- The null moves up to the nearest nullable parent
- Too many `!` and one failure can erase large parts of the response

```
type Query {
  order(id: ID!): Order     # nullable
}
type Order {
  status: OrderStatus!      # non-null
}

# status fails, so order becomes null:
# { "data": { "order": null },
#   "errors": [{ "path":
#     ["order", "status"] }] }
```

GraphQL over HTTP: request

```
POST /graphql HTTP/1.1
Content-Type: application/json
Accept: application/graphql-response+json, application/json

{
  "query": "query OrderPage($id: ID!) { order(id: $id) { status } }",
  "variables": { "id": "42" },
  "operationName": "OrderPage"
}
```

The Rust stack

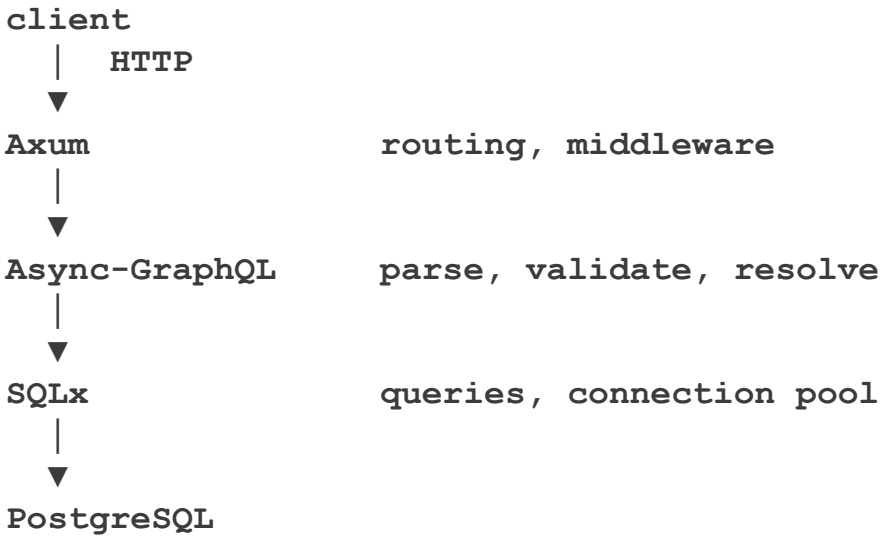
- Axum: handles HTTP
- SQLx: the database
- Async-GraphQL: the GraphQL layer



Why Rust

- Static types map naturally onto a typed schema
- Derive macros generate the schema from Rust types at compile time
- No garbage collector: predictable latency across many small resolvers
- Async on Tokio: sibling fields resolve concurrently
- `Send` and `Sync` catch data races at compile time

Architecture



Schema-first or code-first

async-graphql is **code-first**: Rust types are the schema.

- **Schema-first**: write SDL, then resolvers that match it
- **Code-first**: write code, generate the SDL from it
- Code-first cannot drift from the implementation
- Schema-first suits design reviews before implementation

Axum: routing and state

```rust
#[derive(Clone)]
struct AppState { pool: sqlx::PgPool }

async fn order_status(
    State(state): State<AppState>,
    Path(id): Path<String>,
) -> String {
    // query state.pool ...
}

let app = Router::new()
    .route("/health", get(|| async { "ok" }))
    .route("/orders/{id}/status", get(order_status))
    .with_state(state);

let listener = tokio::net::TcpListener::bind("0.0.0.0:3000").await?;
axum::serve(listener, app).await?;
```

Axum and async-graphql

```rust
let schema = Schema::build(Query, Mutation, Subscription)
    .data(pool)
    .finish();

let app = Router::new()
     .route("/graphql", post(GraphQL::new(schema)))
     .with_state(/* pool */);

let listener =
tokio::net::TcpListener::bind("0.0.0.0:3000").await?;

axum::serve(listener, app).await?;
```

async-graphql: building the schema

```rust
struct Query;

#[Object]
impl Query {
    async fn order(&self, ctx: &Context<'_>, id: ID)
     -> Result<Option<Order>> {
        let pool = ctx.data::<sqlx::PgPool>()?;
        // load the order ...
    }
}

let schema = Schema::build(Query, EmptyMutation, EmptySubscription)
    .data(pool) // available in every resolver through ctx.data()
    .finish();
```

async-graphql: resolvers

```rust
struct Order { id: ID, customer_id: String }

#[Object]
impl Order {
    async fn id(&self) -> &ID { &self.id }

    async fn customer(&self, ctx: &Context<'_>) -> Result<Customer> {
        let pool = ctx.data::<sqlx::PgPool>()?;
        let row = sqlx::query_as::<_, CustomerRow>(
                "SELECT id, name, email FROM customers WHERE id = $1")
            .bind(&self.customer_id)
            .fetch_one(pool)
            .await?;
        Ok(row.into())
    }
}
```

SQLx: connection pool

```
# async, pure Rust, plain SQL;
# cheap to clone and shared across requests.
let pool = PgPoolOptions::new()
    # set optimal configuration here
     .max_connections(5)
    ...
     .connect(&std::env::var("DATABASE_URL")?)
     .await?;
```

SQLx: querying the DB

```rust
#[derive(sqlx::FromRow)]
struct CustomerRow {
    id: String,
    name: String,
    email: Option<String>,
}

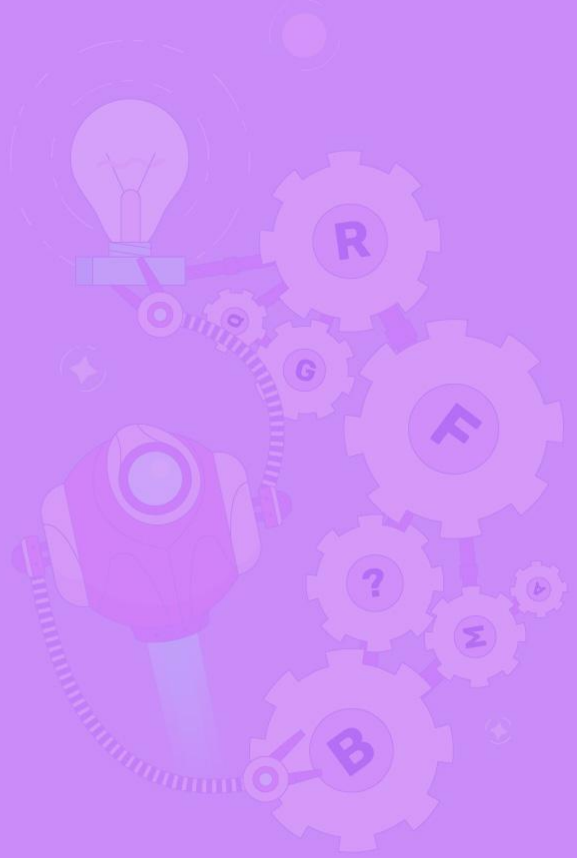
let customer = sqlx::query_as::<_, CustomerRow>(
    "SELECT id, name, email FROM customers WHERE id = $1",
)
    .bind(&id)
    .fetch_optional(&pool)
    .await?;
```

Questions, then hands-on



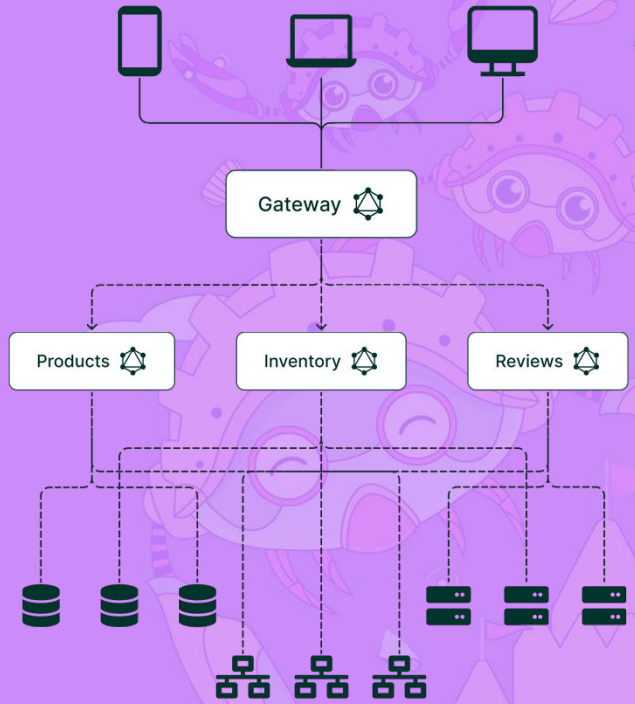
Federation

- **Why Federation**
- **How to Federate**
- **Conflicts Management**



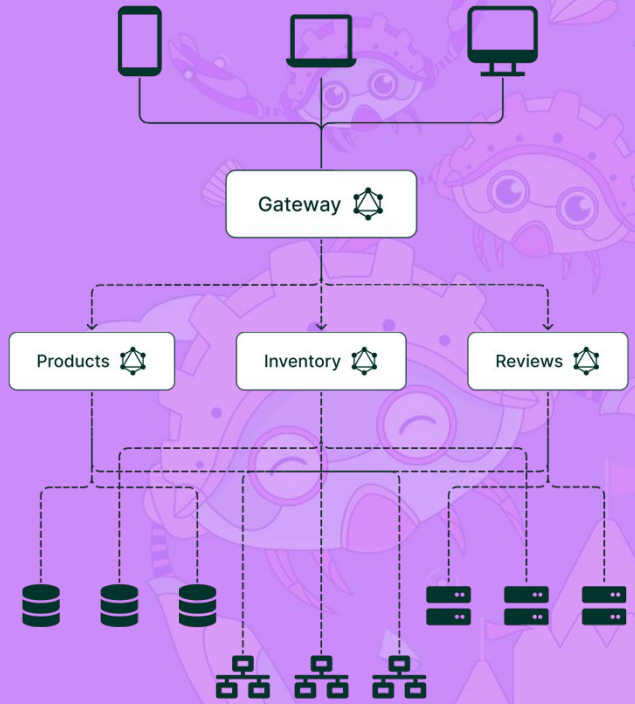
How to Federate

It solves the problem of retrieving information from several schemas keeping the graph endpoint



How to Federate

- The Router hold the super schema
- Analyze the Query to optimize the retrieval
- Collect the payloads and arrange the response



How to Federate

- Now Order has only a Foreign Key related to Product
- @key attributes instruct the router how to resolve data across subschemas

```
type Order {
        id: ID!
        amount: Float!
        items: [Product!]!
}

type Product @key(fields: "id") {
        id: ID!
}
```

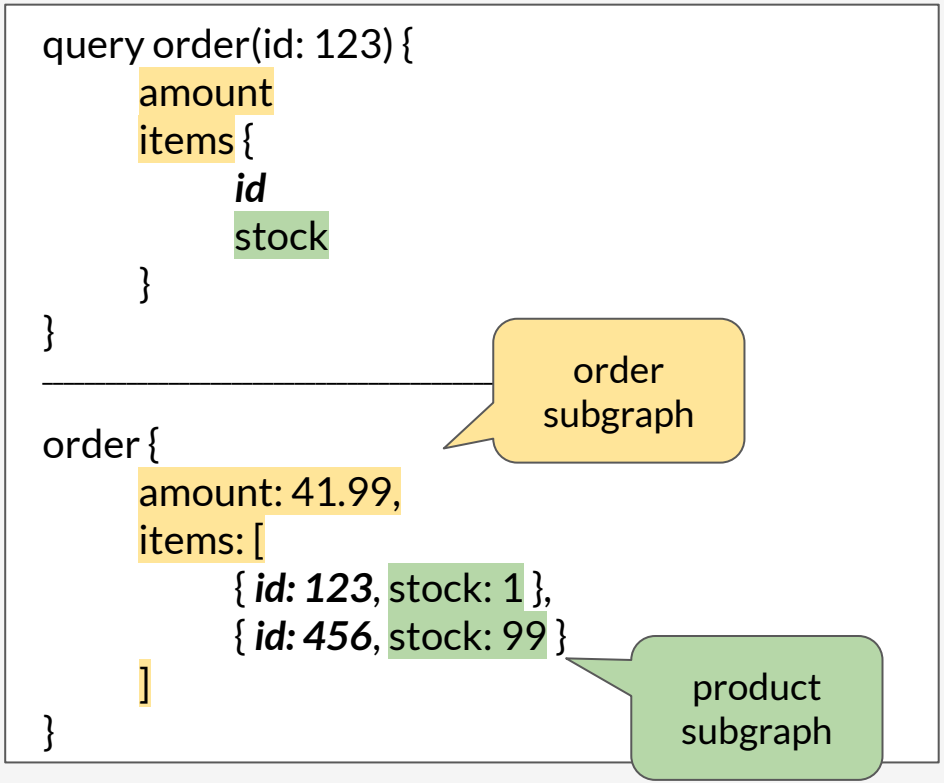
order subgraph

```
type Product @key(fields: "id") {
        id: ID!
        description: String!
        stock: Int!
}
```

product subgraph

How to Federate

The query knows nothing about the subschemas.



Federation

Conflicts Management

What happen if two type define the same field?

```
type Order {
        id: ID!
        amount: Float!
        items: [Product!]!
}

type Product @key(fields: "id") {
        id: ID!
        description: String!
        stock: Int!
}

_____________________________________________

type Product @key(fields: "id") {
        id: ID!
        description: String!
        stock: Int!
}
```

Conflicts Management

- @sharable means: this field can be resolved from both the subschems
- @key fields are shareable by default
- @override(from: XXX): force resolution from specific subschemas

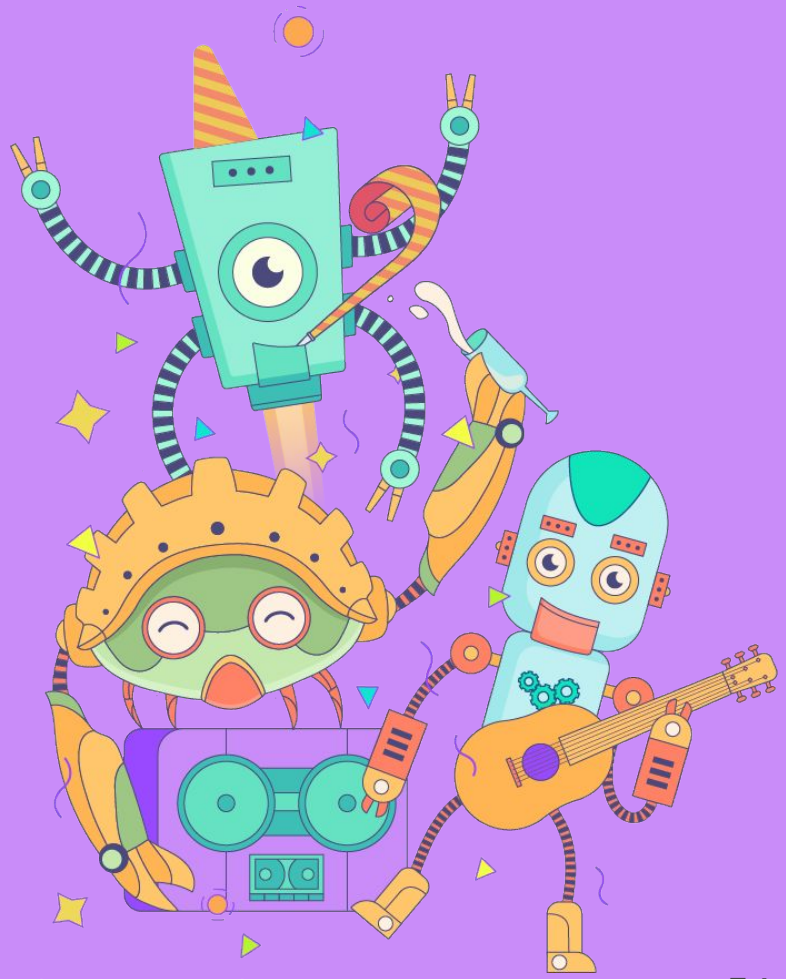
```
type Order {
        id: ID!
        amount: Float!
        items: [Product!]!
}

type Product @key(fields: "id") {
        id: ID!
        description: String!
        stock: Int! @shareable
}

_____________________________________________

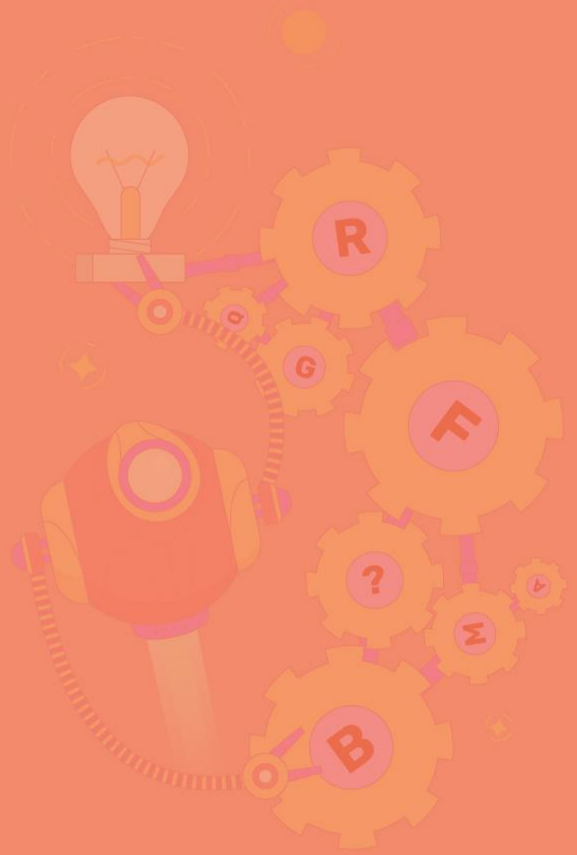
type Product @key(fields: "id") {
        id: ID!
        description: String! @override(from: "orders")
        stock: Int! @shareable
}
```

Questions, then hands-on

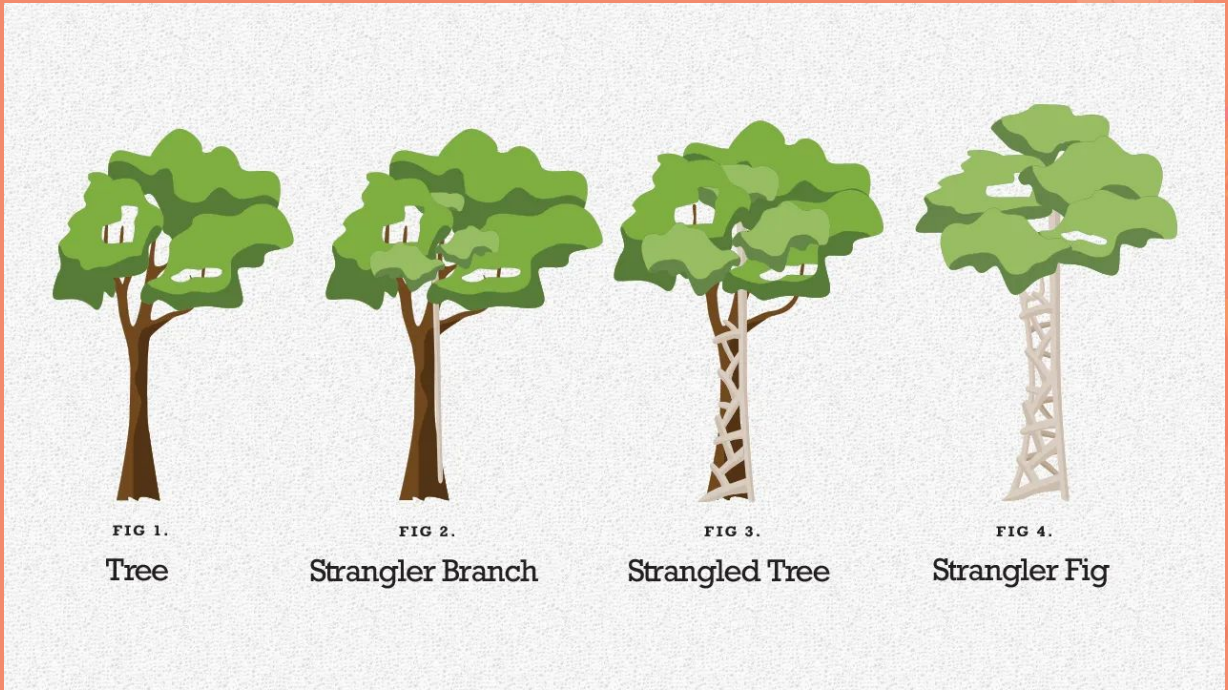


Strangler fig pattern via federation

How to override schemas with feature flags

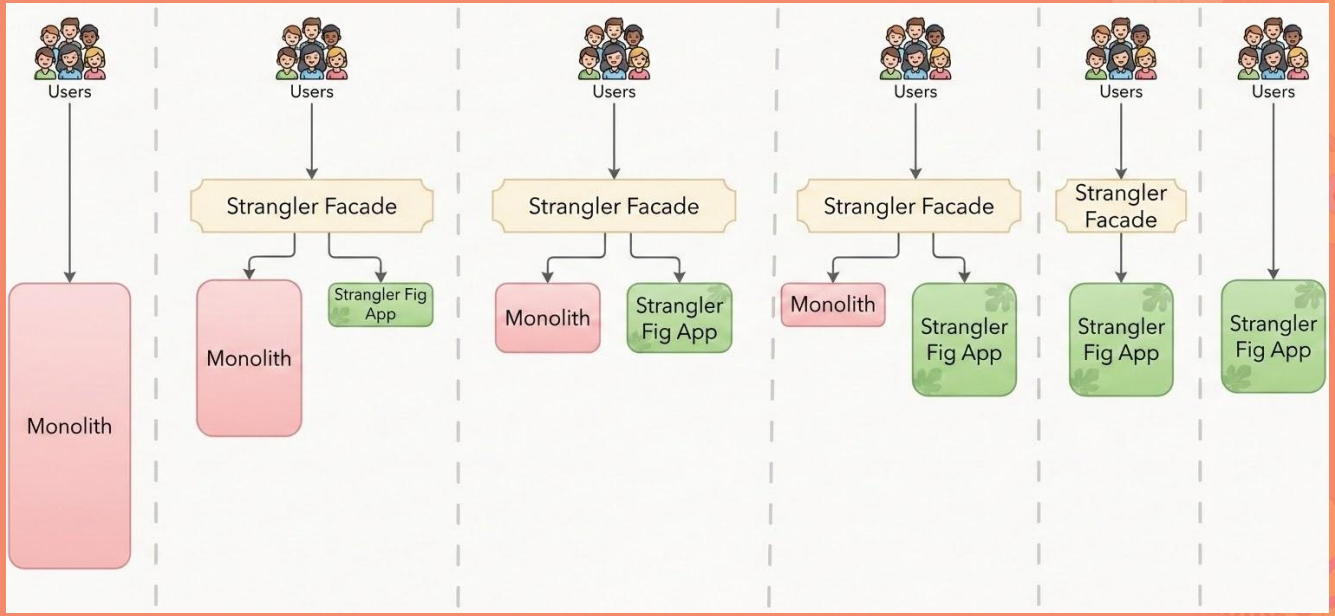


Strangler fig



Strangler fig pattern via federation

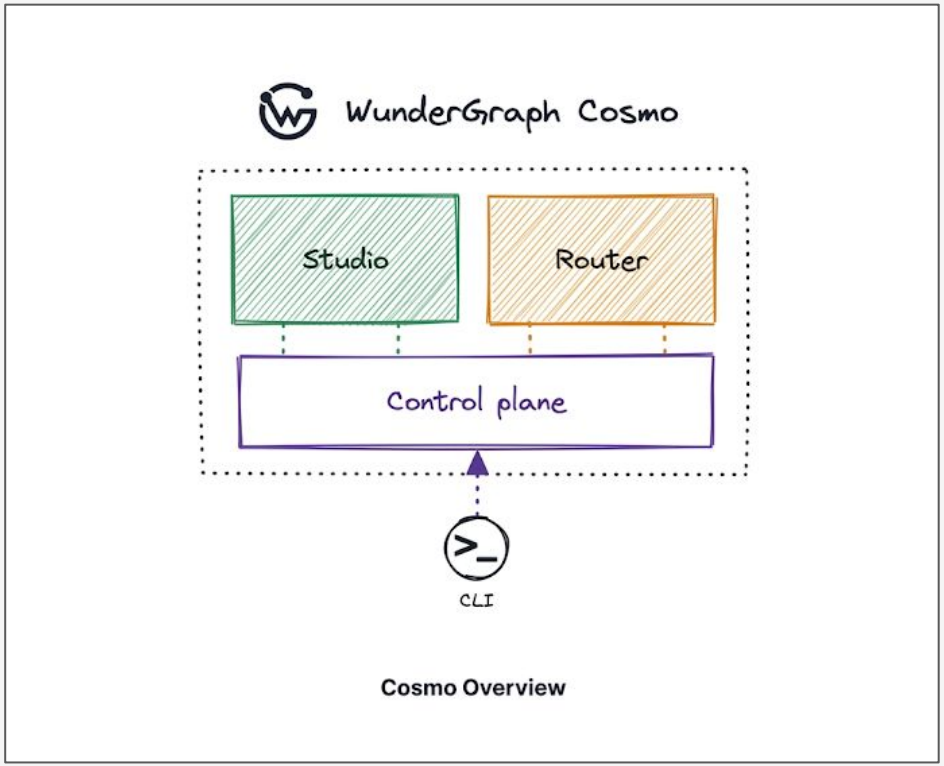
Strangler fig



Strangler fig pattern via federation

WunderGraph

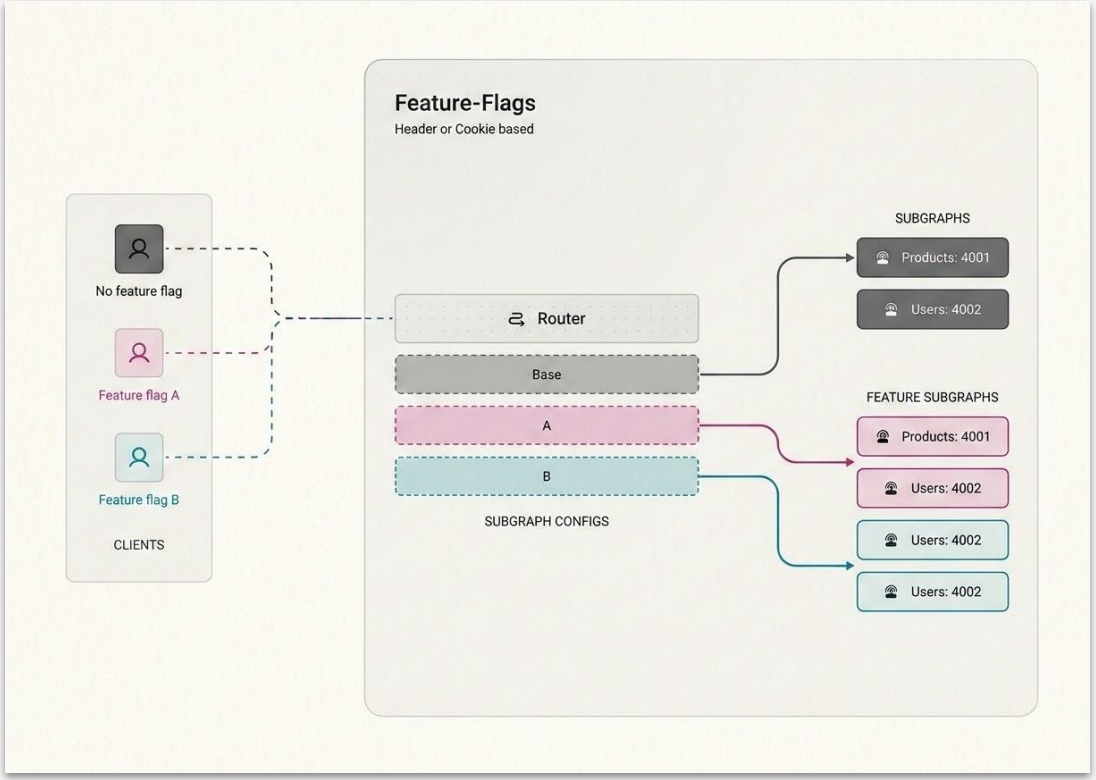
- Router: merge all the subschemas and expose the super graph to the clients
- Control Plane: keep in sync the router configuration and change it live
- Studio: it's web UI to interact with the control plane



Strangler fig pattern via federation

WunderGraph

Clients see a different super graph based on feature flags



Thank you

