View online

Download PDF

Don’t break GraphQL, extend it!

Paolo Insogna

Node.js TSC, Principal Engineer @ **Platformatic**

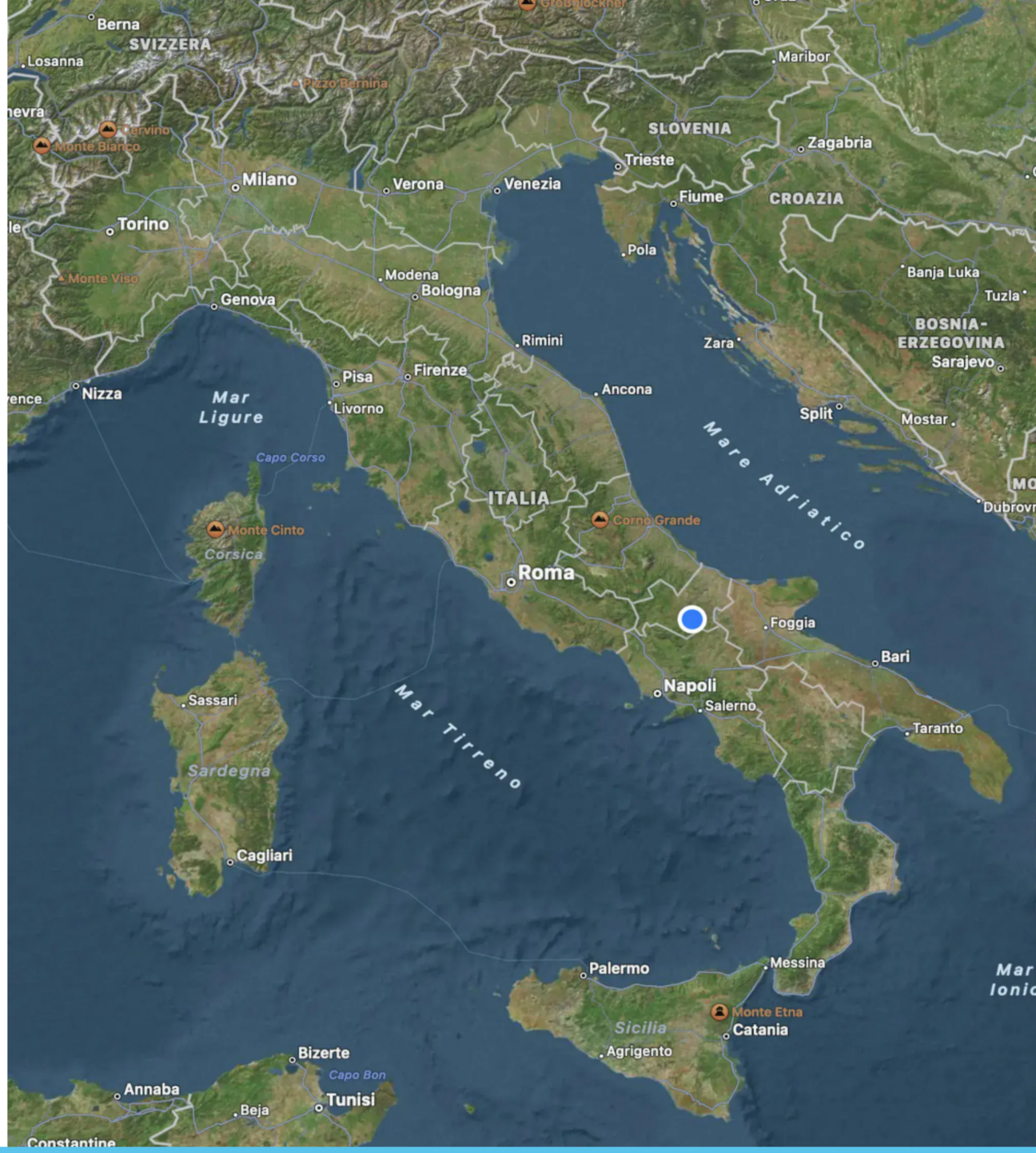
Being kind never hurts!

Hello, I'm **Paolo**!

Node.js Technical Steering Committee Member

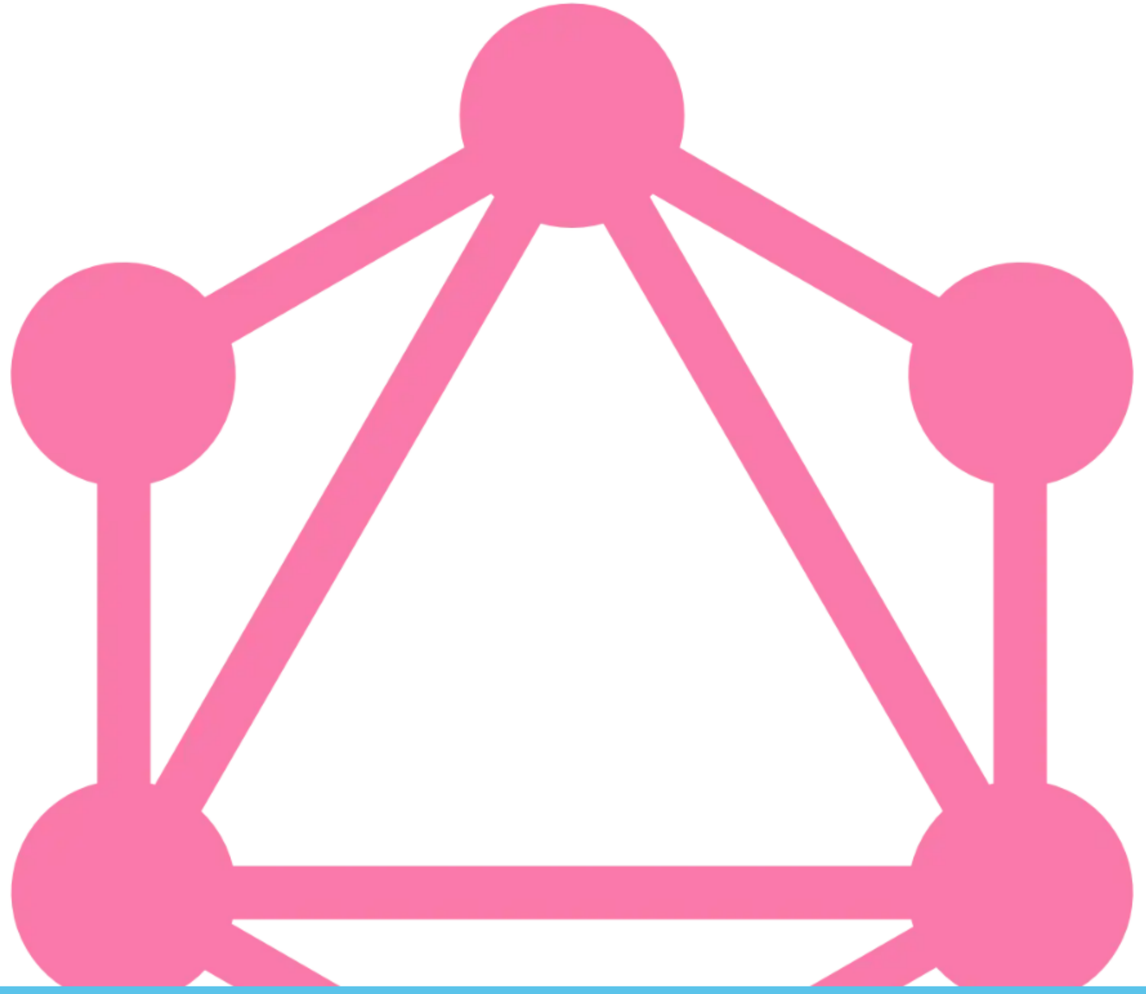
Platformatic Principal Engineer

`paoloinsogna.dev` `ShogunPanda` `p_insogna` `pinsogna`



Let’s celebrate GraphQL!

We all know how this technology has made our life easier.



Why is it good?

Many resources, less overhead, expressive language

You can ask for multiple resources in a single request and save network bandwidth.

No overfetching or underfetching

The server will return exactly the data we asked for. **This will return.**

Federation

Multiple schemas can be easily joined.

Federation

Split the schema in subgraphs

This is great for separation of concerns.

Integration with remote services

Each subgraph is handled by a separate service, possibly remote.

Only works with GraphQL service

You can’t directly integrate with REST APIs or similar.

Serialization

GraphQL does not enforces a serialization format

Developers are free to use whatever they want to.

The network stack is your choice

Neither the data format or the transfer protocol are mandated by the spec.

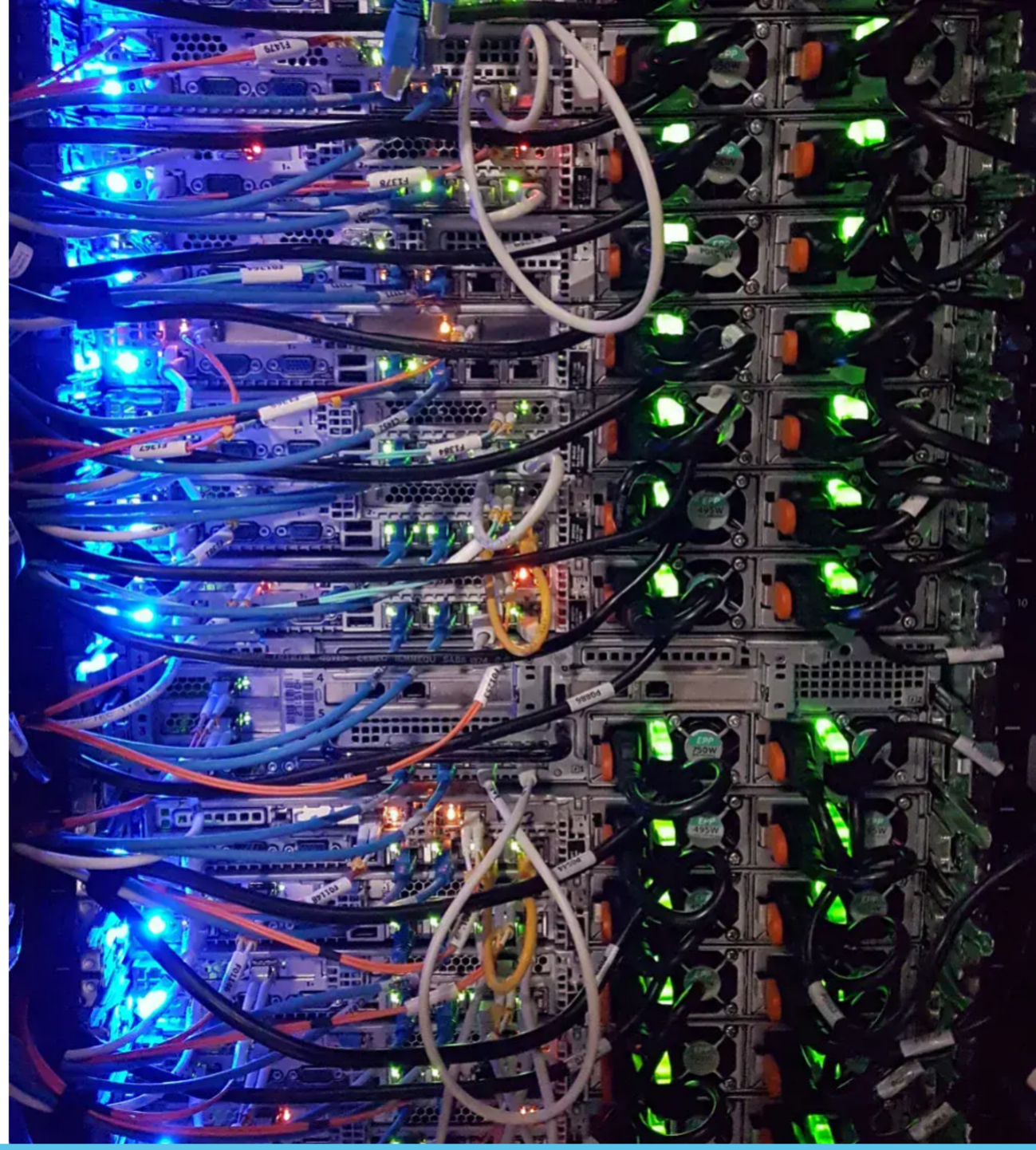
Let’s face the honest truth

99% (total arbitrary) of the times we are talking about JSON data exchanged over HTTP.

The server knows it better

It’s relative easy to attach new data sources to the server.

How to push additional data to a client that didn’t ask for it?



How to be proactive?

Happy case: we control everything

1. **Add new data to the server's schema**
   You should not break backward compatibility.

2. **Verify the clients are still working**
   Double checking that nothing broke is never a bad choice.

3. **Update the clients**
   Update queries in the client to use the new data.

Are we done?

You already know the answer...

The happy case is mostly theoretical

You have to be in control of the clients

You also have to consider users that don’t usually upgrade their applications often.

Nothing can go out of sight

If even a single component is not updated at the right time, compatibility problems will arise.

Specification will be broken

Even if the client is able to handle data it didn’t ask for, **we are breaking the specification**.

BAD!

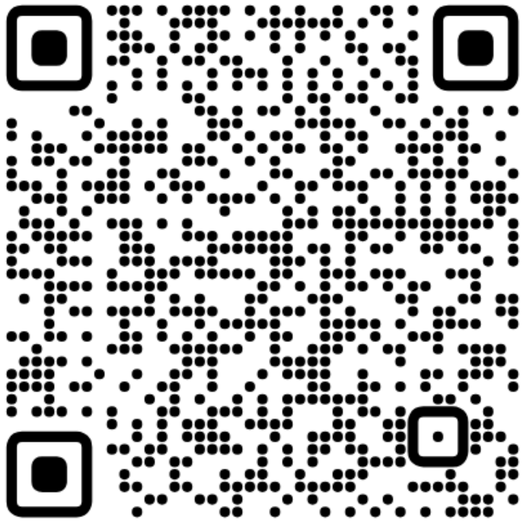
DON'T!

Do we have a choice?

Yes, let's make an enriching proxy!

Check it out!

We can reuse parsing and serialization from the reference GraphQL Javascript implementation.



https://github.com/ShogunPanda/graphql-enrich-proxy

How it works

1. **Analyze the query**
   Parse and validate the query received by the client. In case of request error, stop here.

2. **Ensure types information with temporary modifications**
   Each selection set must contain the type to give all information to the enriching handler.

3. **Execute the query**
   This can be done directly on the server or we can create an enriching GraphQL proxy server.

4. **Fetch the additional data**
   Using a tree traversal algorithm, fetch additional data for each field according to the handler.

5. **Enrich the response**
   Store the additional data in the extensions field, using the JSONPath selector as the key.

Meet GraphQL extensions

The `extensions` field is already documented in the specification and it's perfect to ensure both compatibility and expandability.

Extensions are for developers

The specification states that the field is reserved for developers.

Existing clients will ignore them

Unaware clients will ignore the field and the server is still specification compliant.

Overview

```
import fastify from 'fastify'
import { parse, print, visit } from 'graphql/language/index.js'

/* ... */

server.post('/graphql', async function handleQuery(req, reply) {
  // Step 1: Parse the query and check for syntax error
  const document = parse(req.body.query)

  // Step 2: Add types information to the query
  const [enrichedAst, enrichedId] = addTypesInformation(document)

  // Step 3: Execute the query on the upstream
  const response = await graphql('https://api.geographql.rudio.dev', print(enrichedAst))

  // Step 4 and 5: Execute the enriching handler to fetch additional data
  const extensions = await enrich(response.data, enrichedId, addWeatherInformation)

  // Return to the client
  return { ...response, extensions: { ...response.extensions, ...extensions }}
})
```

Leveraging types

We make sure `__typename` is in all selection sets so we can easily parse the response.

The added fields are temporary
Never return these to the client.

Leverage field aliasing
Use it to easily spot the fields added.

```
query test {
  hero {
    __typename
    name
    friends {
      __enriched: __typename
      name
      homeglobe {
        aliasedType: __typename
        __enriched: __typename
        name
        climate
      }
    }
  }
}
```

Ensuring type information

The visit API and the AST from the GraphQL reference implementation makes our life very easy.

```
function injectedTypeField(id) {
  return {
    kind: 'Field',
    alias: { kind: 'Name', value: id },
    name: { kind: 'Name', value: '__typename' }
  }
}

function addTypesInformation(document) {
  const injectedFieldId = `enrichType_${Date.now()}`
  const injectedField = injectedTypeField(injectedFieldId)

  const updatedDocument = visit(document, {
    SelectionSet(node) {
      // Check if the type is already included unaliased
      for (const f of node.selections) {
        if (f.name.value === '__typename' && !f.alias) {
          return
        }
      }

      // If we got there, we still have to add the typename
      node.selections.unshift(injectedField)
      return node
    }
  })

  return [updatedDocument, injectedFieldId]
}
```

Cache the queries

Parsing and executing GraphQL is expensive. Cache them when possible.

Cache the original query...

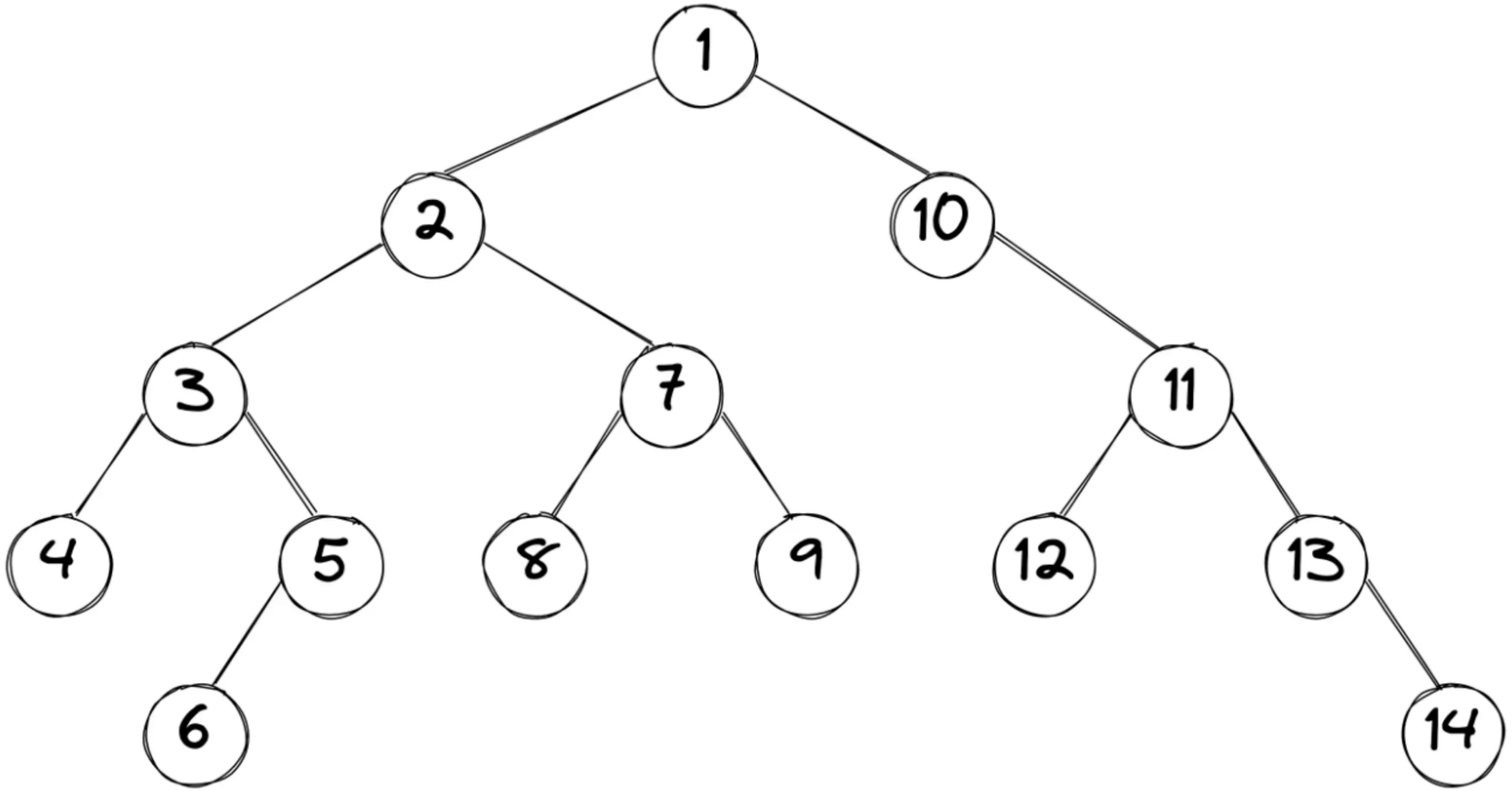
This also includes invalid queries as it will speed up the handling of misbehaving clients.

...and the enriched query

Traversing a complex query to ensure types information can be time consuming.

Let me introduce two friends...

Depth first tree traversal



JSONPath

It is a string syntax for selecting and extracting values within a JSON value.

Well known
You are already using JSONPath or a similar syntax.

Easily implementable
The syntax is easy to understand and implement.

```
shogun@panda:~/example$ cat example.json
{
  "countries": [
    { "name": "Iceland", capital: "Reykjavík", },
    { "name": "Italy", capital: "Roma" }
  ]
}

shogun@panda:~/example$ jq ".countries[1].capital" example.json
"Roma"
```

… and now the show goes on!

Enrich the data

We traverse the upstream response, executing the handler on each node.

If data is returned from the handler, we append to the extension using JSONPath.

```
async function enrich(data, enrichedId, handler) {
  const extensions = {}

  await traverse(data, async function (value, path) {
    // Execute the handler on the node
    // and eventually add the returned data
    const additional = await handler(
      value.__typename || value[enrichedId],
      path,
      value
    )

    if (additional) {
      const jsonPath = path
        .join('.')
        .replace(/\.(\d+)\./g, '[$1].')

      extensions[jsonPath] = additional
    }

    // Make sure we remove any fields we added
    value[enrichedId] = undefined
  })

  return extensions
}
```

Tree traversal

Implementing a depth first tree traversal is quite easy when using recursion.

```
async function traverse(current, path, visitor) {
  // This is to handle the initial call
  if (typeof path === 'function') {
    visitor = path
    path = ['$']
  }

  // First of all, call the visitor on the current object
  await visitor(current, path)

  // For each enumerable property in the object,
  // perform a depth first traverse of the property
  // if it is an array of objects or an object.
  for (const [key, val] of Object.entries(current)) {
    if (Array.isArray(val)) {
      for (let i = 0; i < val.length; i += 1) {
        await traverse(val[i], path.concat(key, i), visitor)
      }
      // Say thanks to JS typing for the null checking
    } else if (typeof val === 'object' && val !== null) {
      await traverse(val, path.concat(key), visitor)
    }
  }
}
```

Fetch additional data

Using the type and/or the path we can choose whether the node needs additional data.

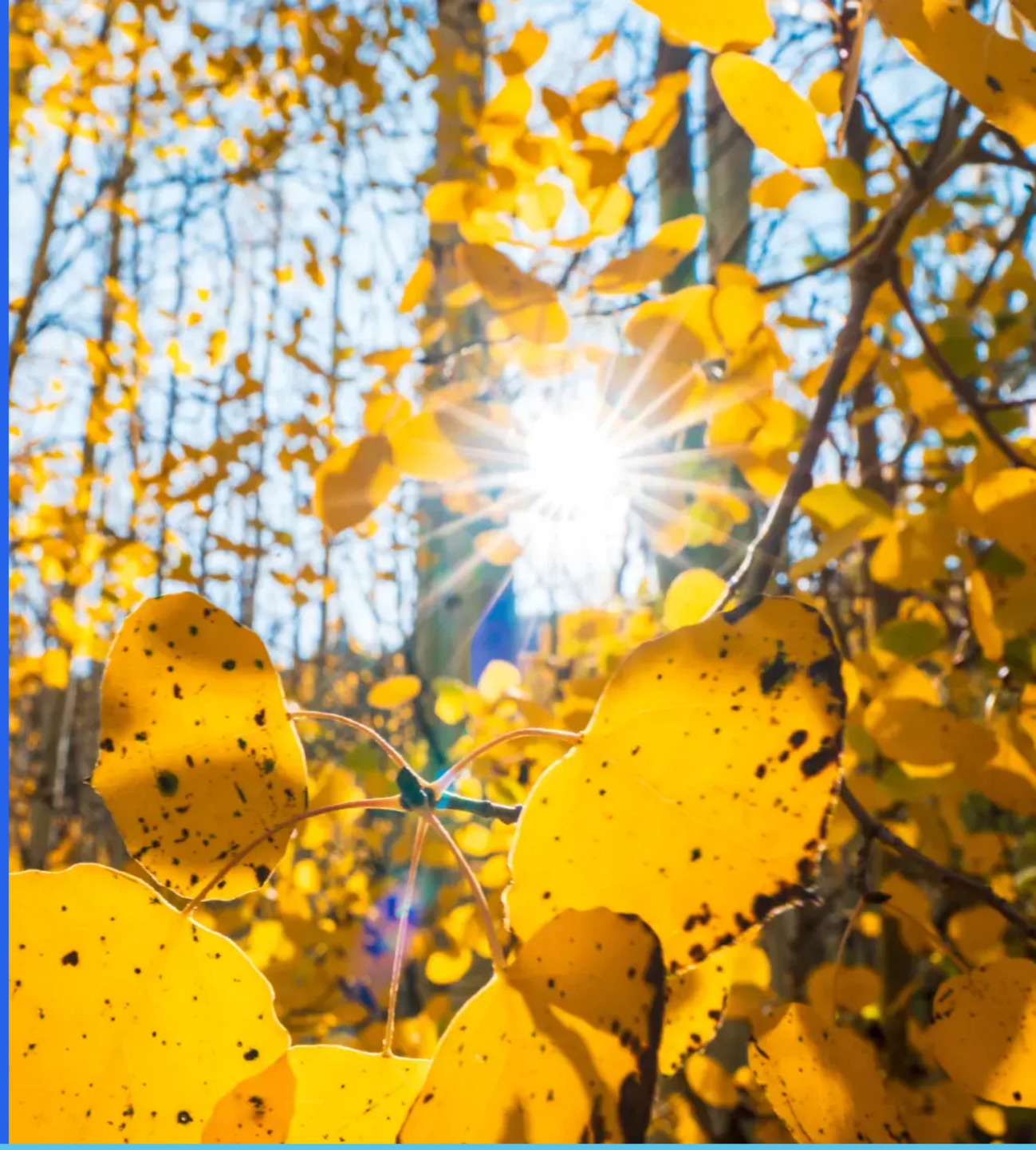
```
async function addWeatherInformation(type, path, value) {
  if (type !== 'City') {
    return
  }

  // Get weather information for today for the city
  const response = await undici.request(
    'https://goweather.herokuapp.com',
    {
      method: 'GET',
      path: `/weather/${value.name}`,
      dispatcher: agent
    }
  )

  const body = JSON.parse(await getStream(response.body))

  return response.statusCode === 200
    ? { temperature: body.temperature }
    : {
        error: {
          statusCode: response.statusCode,
          body
        }
      }
}
```

Only an example can enlighten us!



Input query

```
{
  country(iso2: "US") {
    aliasedType: __typename
    name
    cities(page: {first: 2}) {
      __typename
      edges {
        node {
          name
        }
      }
    }
  }
}
```

Query executed from the upstream

```
{
  enrichType_1677506561773: __typename
  country(iso2: "US") {
    enrichType_1677506561773: __typename
    aliasedType: __typename
    name
    cities(page: {first: 2}) {
      __typename
      edges {
        enrichType_1677506561773: __typename
        node {
          enrichType_1677506561773: __typename
          name
        }
      }
    }
  }
}
```

Upstream response

It contains the enriched type information that will be removed from the final response.


```
{
  "data": {
    "enrichType_1677506561773": "Query",
    "country": {
      "enrichType_1677506561773": "Country",
      "aliasedType": "Country",
      "name": "United States",
      "cities": {
        "__typename": "CityConnection",
        "edges": [
          {
            "enrichType_1677506561773": "CityEdge",
            "node": {
              "enrichType_1677506561773": "City",
              "name": "Abbeville"
            }
          },
          {
            "enrichType_1677506561773": "CityEdge",
            "node": {
              "enrichType_1677506561773": "City",
              "name": "Alabaster"
            }
          }
        ]
      }
    }
  }
}
```

Proxy response (1/2)

The client will receive the original data requested ...


```
{
  "data": {
    "country": {
      "aliasedType": "Country",
      "name": "United States",
      "iso": "US",
      "cities": {
        "__typename": "CityConnection",
        "edges": [
          {
            "node": {
              "name": "Abbeville"
            }
          },
          {
            "node": {
              "name": "Adamsville"
            }
          }
        ]
      }
    }
  },
  "extensions": {
    /* ... */
  }
}
```

Proxy response (2/2)

...and all our enriched data is in the extensions field.


```
{
  "data": {
    /* ... */
  },
  "extensions": {
    "$.country.cities.edges[0].node": {
      "temperature": "+5 °C"
    },
    "$.country.cities.edges[1].node": {
      "error": {
        "statusCode": 404,
        "body": {
          "message": "NOT_FOUND"
        }
      }
    }
  }
}
```

Mission completed!

Take home lessons

What can we learn from this long journey?

Read the specification

The specifications are formal and verbose but they **might already contain what you need**.

Be compliant

Even if peers are lenient with specification break, they might stop at any time. **Don’t risk!**

Analyse the environment

Even if you develop the experience end to end, you will never be able to **cannot control everything**.

One last thing™

“You are remembered for the rules you break.”

Douglas MacArthur

Thank you!

Paolo Insogna

Node.js TSC, Principal Engineer

@p_insogna

paolo.insogna@platformatic.dev

Platformatic