Boosting development using GraphQL Gateway with REST APIs

How Many times when working with REST-based microservice architectures, we end up with projects having API complexity problems, because, as the project grows

- more and more services directly depend on / care coupled with each other
- data aggregation between the services (for e.g. frontend consumption) becomes a significant burden, slowing the project down
- the number of services increases, which can be a maintainability issue if there are no communication contracts between them

On the front-end, the complications roughly mirror the issues outlined above:

- Communicating with many different REST APIs can be cumbersome
- Especially when there are data dependencies between the APIs, retrieving all of the relevant data for a single view may involve several REST API calls, making development and the application slow

Many of these complexities can be simplified by using a **GraphQL Gateway.**It unifies and standardizes communication between REST Microservices, and brings a unique access interface with well defined auto generated data types, queries and mutations to the frontend.

In short words, aGraphQL Gateway creates data relations between a growing number of Entities that belong to different services without the danger of creating deep coupling or an unstoppable mesh of REST calls.

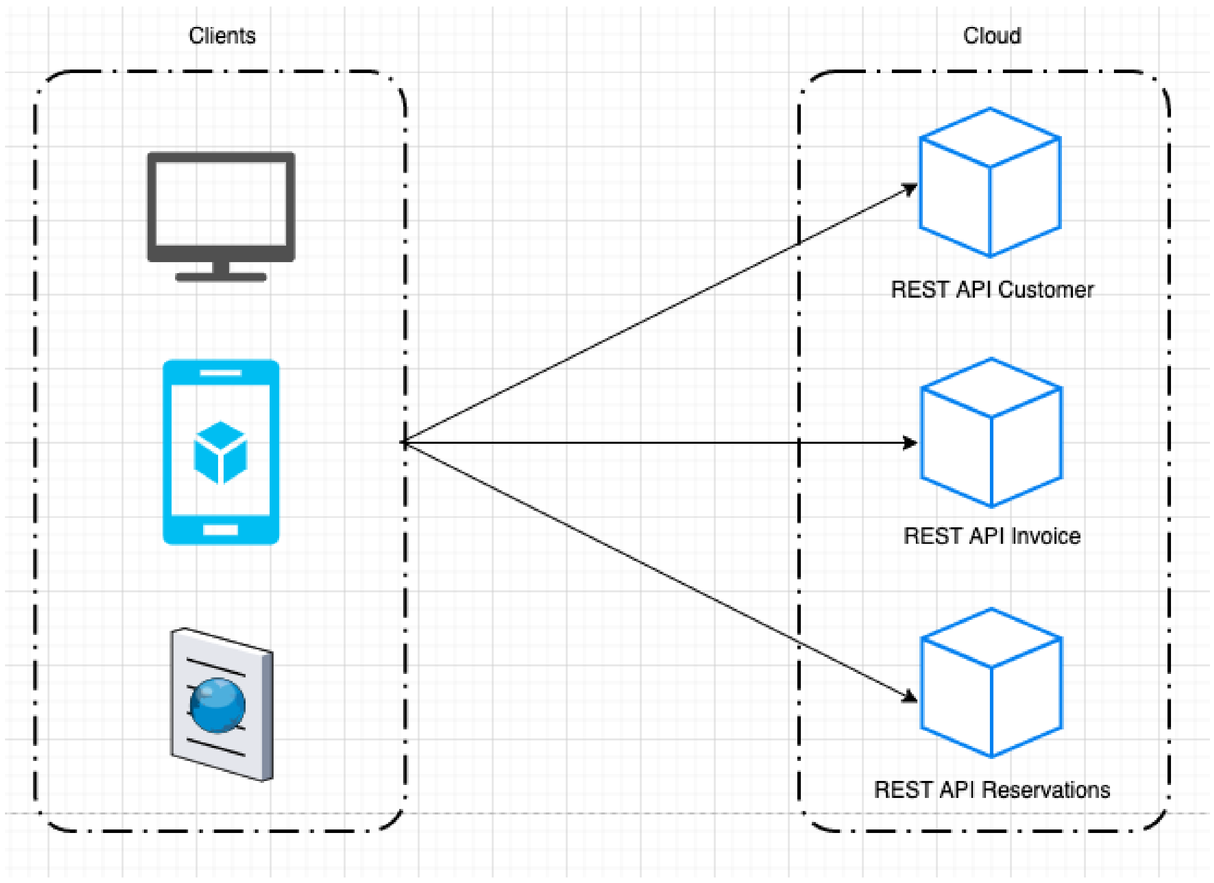
What is GraphQL

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

More on GraphQL Documentation.

Let's take, as an example, this repository,it contains the implementation of two microservices. Customer and Invoice, both expose a REST API documented using OpenAPI/Swagger. Now, we would like to show on the Frontend the invoice paid amount and the customer name. How would we implement this ?

Option 1



We could move the business logic to the clients (browser, phone etc), so it creates two REST calls, one to the Customer API and another one to the Invoice API, in that case **calls continue growing along with the amount of new microservices**.

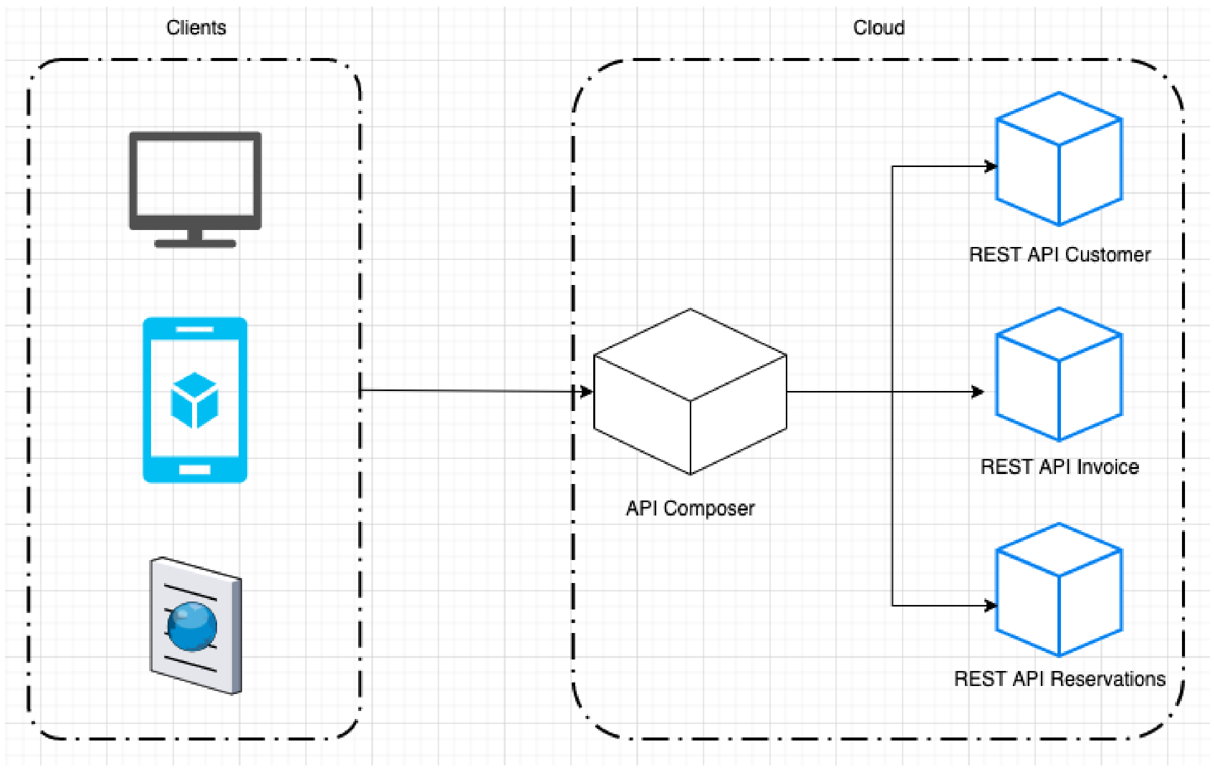
In this approach the **Frontend needs to be aware of multiple APIs**, which is hard to maintain when the amount of microservices are continuously increasing.

Also, when new microservices are incorporated into the architecture our Frontend needs to add new URLs, endpoints if there are more than one and deal with the different types of HTTP requests and their configuration.

In this case the web page is showing the relation between the Entities customer and invoice(Per customer a list of invoices).To access the same data aggregation on the phone app it needs to be implemented there again because the **entities relations can't be reused**.

In addition to the previous, controlling REST APIs response payload's properties doesn't come out of the box. Which could mean, everytime we request the customer or invoice data it comes the Entity with all the properties whether we need it or not. Which **could cause performance issues due to network latencies and the transference of big data chunks over the internet**.

Option 2

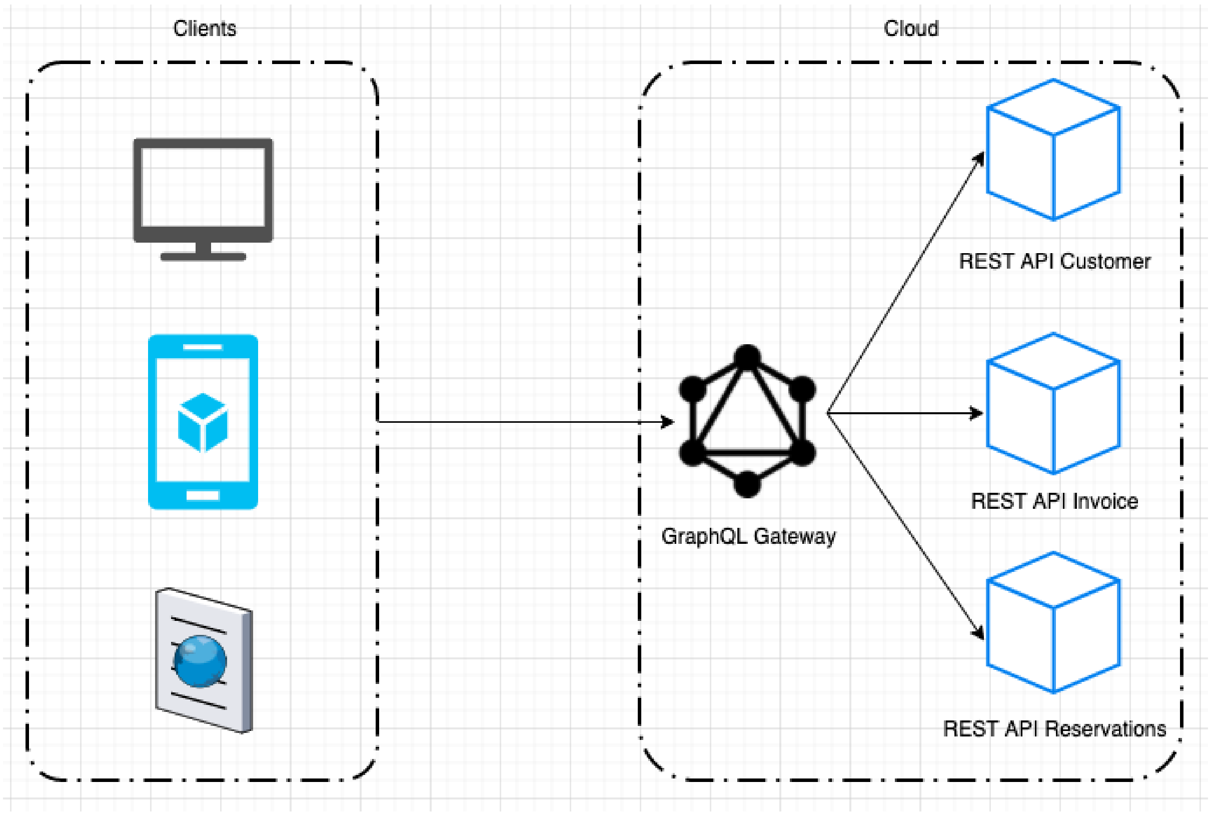


Another way is to create an [API Composer service](), which should expose endpoints that allow us to retrieve the entities with their aggregations by doing only one call on the Frontend. **Ex** one customer with all associated invoices.

The service no doubt performs better than the previous Option, since reduce the amount of calls, but **relations continue to be non reusable**, we could get the relationship customer and invoices from the API Composer service on a particular GET endpoint but can't do the same on another endpoint that uses POST and returns the same Customer Entity without manually implementing the aggregation of data, so again another solution needs to be found.

Each endpoint on the API Composer service returns a relation between entities(**Ex** : Customer - Invoice, Customer - Reservation, Invoice - Reservation), but as soon as new relation is needed like Customer - Invoice - Reservation **the response payload becomes huge** because REST calls does not exclude properties from the response without implementing it and **previous relations can't be reused between endpoints**, needs to be reimplemented turning into an unmanageable grow.

GraphQL Gateway as solution:



By introducing the GraphQL Gateway service on our architecture the **Frontend has a single entry point to access the data** no matter how many services are in the Backend.

Data relations/aggregations can be reused by different clients(web app, phone app) because they are established between Entities. Once we declare that Customer Entity has a relation with Invoice Entity, it becomes available to all operations(Query, Mutation) that use those Entities.

GraphQL Types and resolvers for Queries and Mutations are automatically generated, which means that the service creates on its own GraphQL Types out of the OpenAPI documentation and knows where to find the data and which HTTP methods (POST, GET, etc) to use.

The Frontend requests and receives **only** the Entity's properties needed, which **reduces the amount of data travelling over the internet** between our Backend and Frontend. The previous advantage also contributes to getting the Customer and associated Invoices by doing only one request.

Adding to our Backend **new services does not require any changes on the Frontend** to access new Data Types, Queries and Mutations.

What does GraphQL Gateway actually do ?

GraphQL Gateway is using the package gql-gateway internally, which is in charge of connecting to each one of the microservices, reading, interpreting, transforming automatically all the OpenApi/Swagger documents into GraphQL Types and auto generating resolvers for all Queries and Mutations.

The service creates an Apollo Server with all its features, takes care of validating the Input data and resolving only those properties requested by the client.

So, when the Frontend requests the entities Customer and Invoice the service knows precisely to which service connects to retrieve the data from.

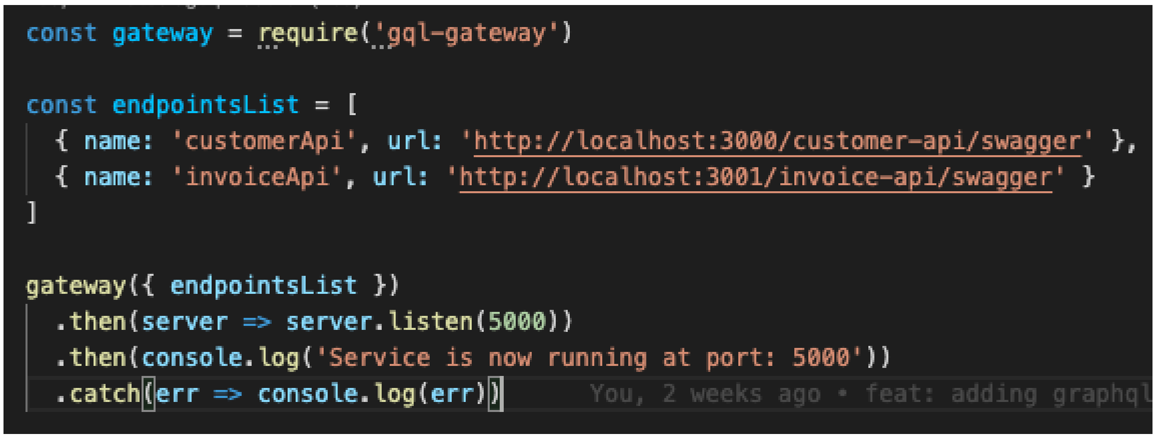
The service is completely stateless which allows us to scale it according to the amount of request and usage.

How hard would it be to implement this approach?

The creation of the gateway with all the auto generated GraphQL Types, Queries and Mutations should take no more than a few lines of code.

Let's start from the beginning …

Imagine we have two microservices that are exposing the OpenApi/Swagger documentation on ports `3000` and `3001` which describe endpoints to retrieve customers and invoices.



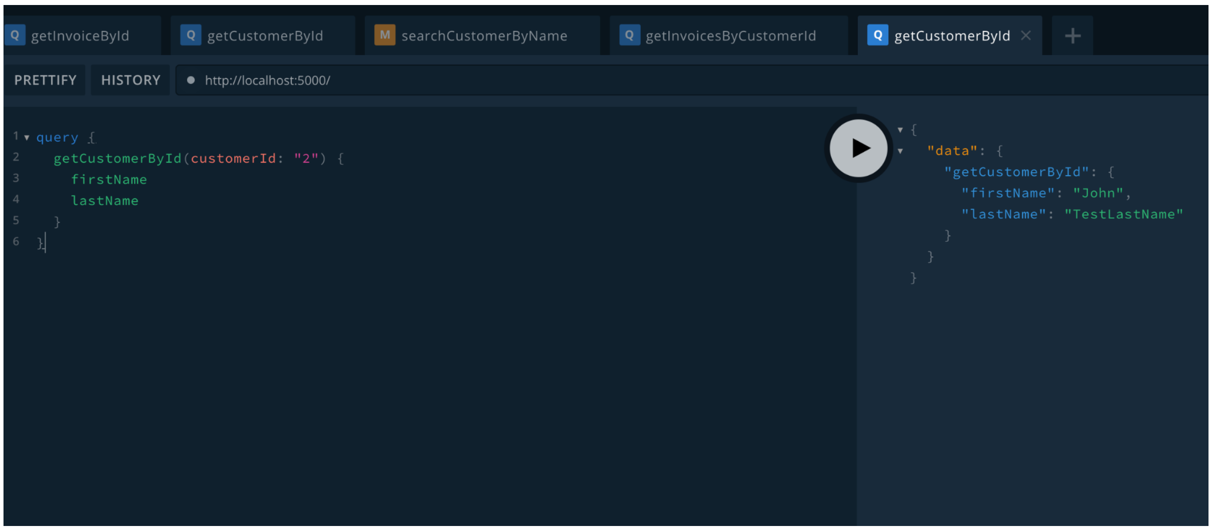
```
const gateway = require('gql-gateway')

const endpointsList = [
  { name: 'customerApi', url: 'http://localhost:3000/customer-api/swagger' },
  { name: 'invoiceApi', url: 'http://localhost:3001/invoice-api/swagger' }
]

gateway({ endpointsList })
  .then(server => server.listen(5000))
  .then(console.log('Service is now running at port: 5000'))
  .catch(err => console.log(err))      You, 2 weeks ago • feat: adding graphql
```

The constant `endpointsList` contains a list of all the microservices OpenApi/Swagger documentation. This configuration is passed to the **gql-package** which takes care of creating the Apollo Server which listens on port 5000.

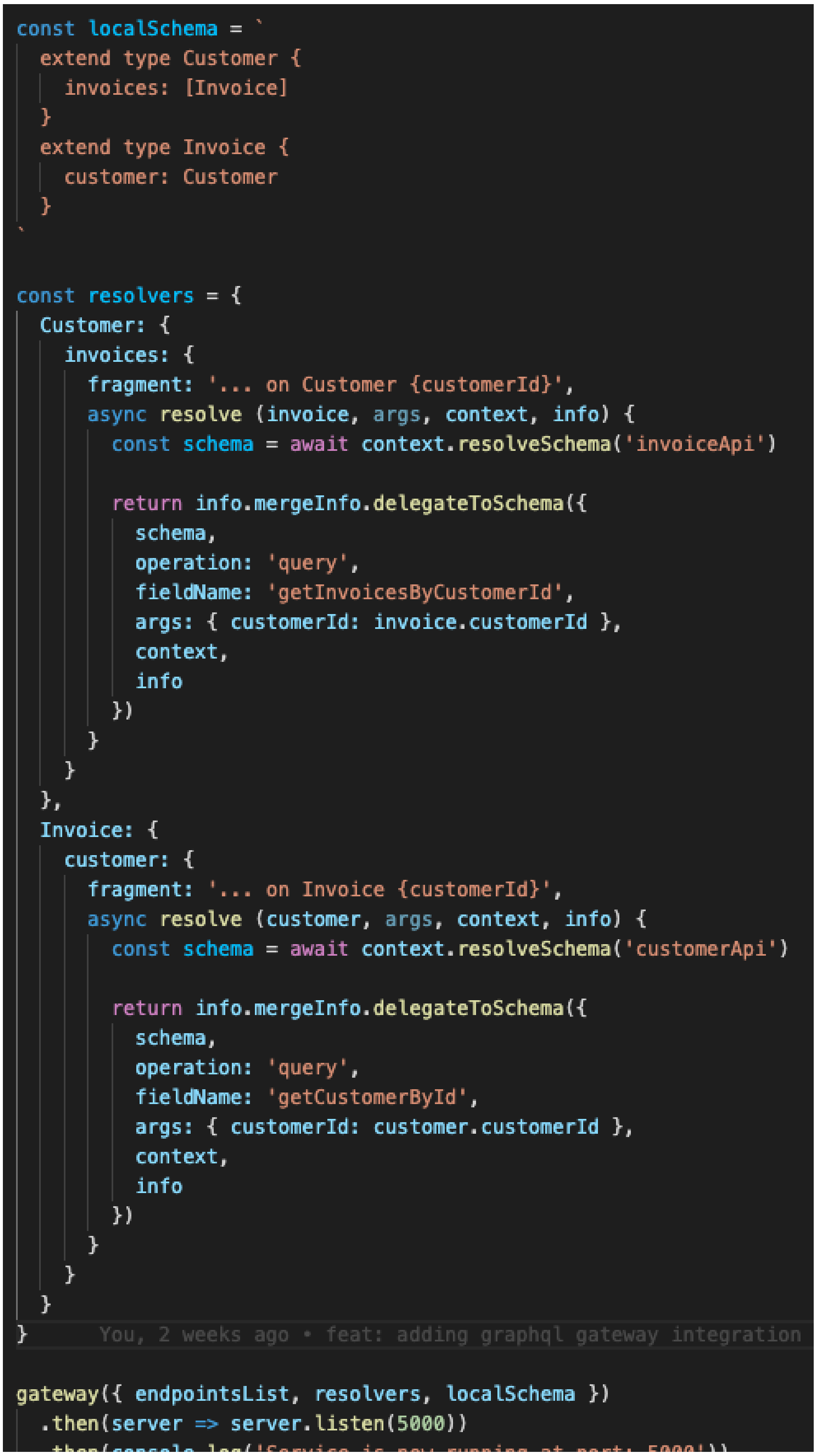
We can access the apollo playground by navigating to `http://localhost:5000` which gives an idea of all the auto generated Types, Queries and Mutations available.



```
query {
  getCustomerById(customerId: "2") {
    firstName
    lastName
  }
}
```

```
{
  "data": {
    "getCustomerById": {
      "firstName": "John",
      "lastName": "TestLastName"
    }
  }
}
```

Now, Let’s create some Entity relations.



```javascript
const localSchema = `
  extend type Customer {
    invoices: [Invoice]
  }
  extend type Invoice {
    customer: Customer
  }
`

const resolvers = {
  Customer: {
    invoices: {
      fragment: '... on Customer {customerId}',
      async resolve (invoice, args, context, info) {
        const schema = await context.resolveSchema('invoiceApi')

        return info.mergeInfo.delegateToSchema({
          schema,
          operation: 'query',
          fieldName: 'getInvoicesByCustomerId',
          args: { customerId: invoice.customerId },
          context,
          info
        })
      }
    }
  },
  Invoice: {
    customer: {
      fragment: '... on Invoice {customerId}',
      async resolve (customer, args, context, info) {
        const schema = await context.resolveSchema('customerApi')

        return info.mergeInfo.delegateToSchema({
          schema,
          operation: 'query',
          fieldName: 'getCustomerById',
          args: { customerId: customer.customerId },
          context,
          info
        })
      }
    }
  }
}       You, 2 weeks ago • feat: adding graphql gateway integration

gateway({ endpointsList, resolvers, localSchema })
  .then(server => server.listen(5000))
  .then(console.log('Service is now running at port: 5000'))
```

All this might look like a complicated code but is actually the standard way of creating GraphQL Schema Delegation, which means that the resolution of a particular property within the Entity is going to be delegated to a specific Query or Mutation.

In this case, to indicate that the Customer is going to have a list of associated invoices we just extend the Customer Type on the `localSchema` constant by adding an array of the Invoice Type. Then, on the resolvers we indicate that the invoices property of the Customer is going to be resolved by delegating it to the Query `getInvoicesByCustomerId` that belongs to the `invoiceApi`.

For a deeper look into the `gql-package` check the README or the advanced example.

From now on, everytime a Customer is included in the response of any of the graphql queries or mutations we also have access to the Invoices and the other way around too. Delegations and Schema Stitching are an elegant way to resolve and create data relations in an scalable way.

Of course, the GraphQL Gateway per se is not the solution to all problems, and this is just a simplified representation in the direction of boosting development speed. To successfully apply solutions like the ones described above, implementation should come accompanied by good development practices, APIs gateways as intermediary between our services and the internet, load balancers, well designed cache policies and good CI/CD strategies just to mention a few. There is no silver bullet to tackle big infrastructures technical issues, but DOING continues to be the best way to learn how to.

For a deep dive about related topics, you can take a look at:

- API Composition Pattern
- Aggregator vs API Composition Pattern
- Graphql Gateway Review
- Apollo Gateway
- Apollo Federation
- Graphql Schema Delegation
- Graphql Schema Stitching
- OpenAPI/Swagger

Terms used on the article:

- REST (Representational state transfer)
- API (Application programming interface)