In late 2016, we decided to rewrite our aging PHP legacy system using Python and React. With only four months to build an MVP in time for the 2017 festival season, we had to decide very carefully where to invest time.

One of the technologies we invested in was GraphQL. None of us had ever worked with it before, but we judged it to be crucial for delivering quickly and enabling people to work independently.

This turned out to be a great decision, so two years later we wanted to look back and share what we've learned since then...

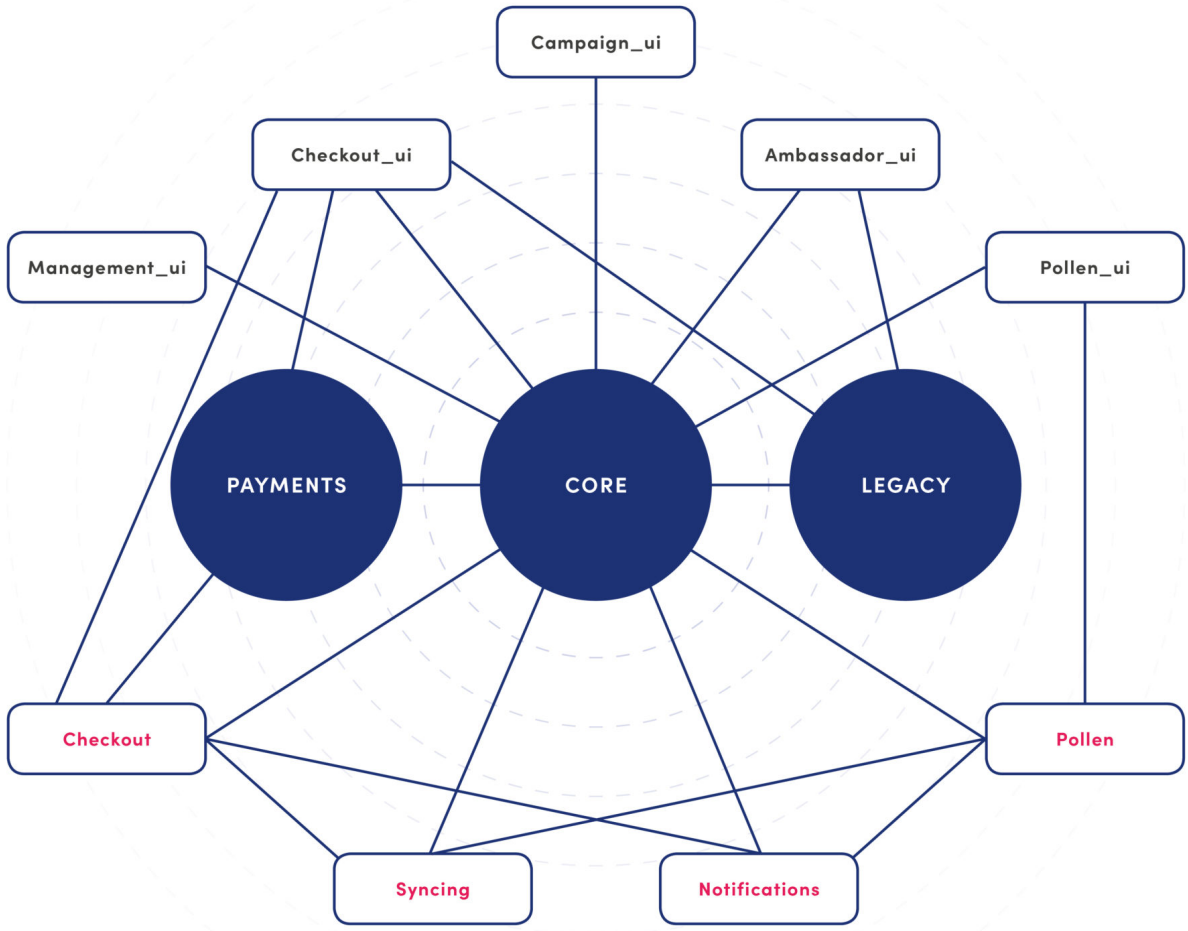
GRAPHQL: TWO YEARS LATER

Our decision to use GraphQL was heavily influenced by lessons learned from our legacy system. We were using REST APIs between a fair number of microservices, which were creating a mess of inconsistent interfaces,

varying resource identifiers and incredibly complex deployments. Any API change had to be deployed simultaneously to all services using that API to avoid downtime, which often went wrong and resulted in long release cycles. Using GraphQL in a single API gateway, we would simplify the service landscape drastically. We also decided to use Relay, giving us a single, global way of identifying resources and a straightforward way of organising our GraphQL schema.

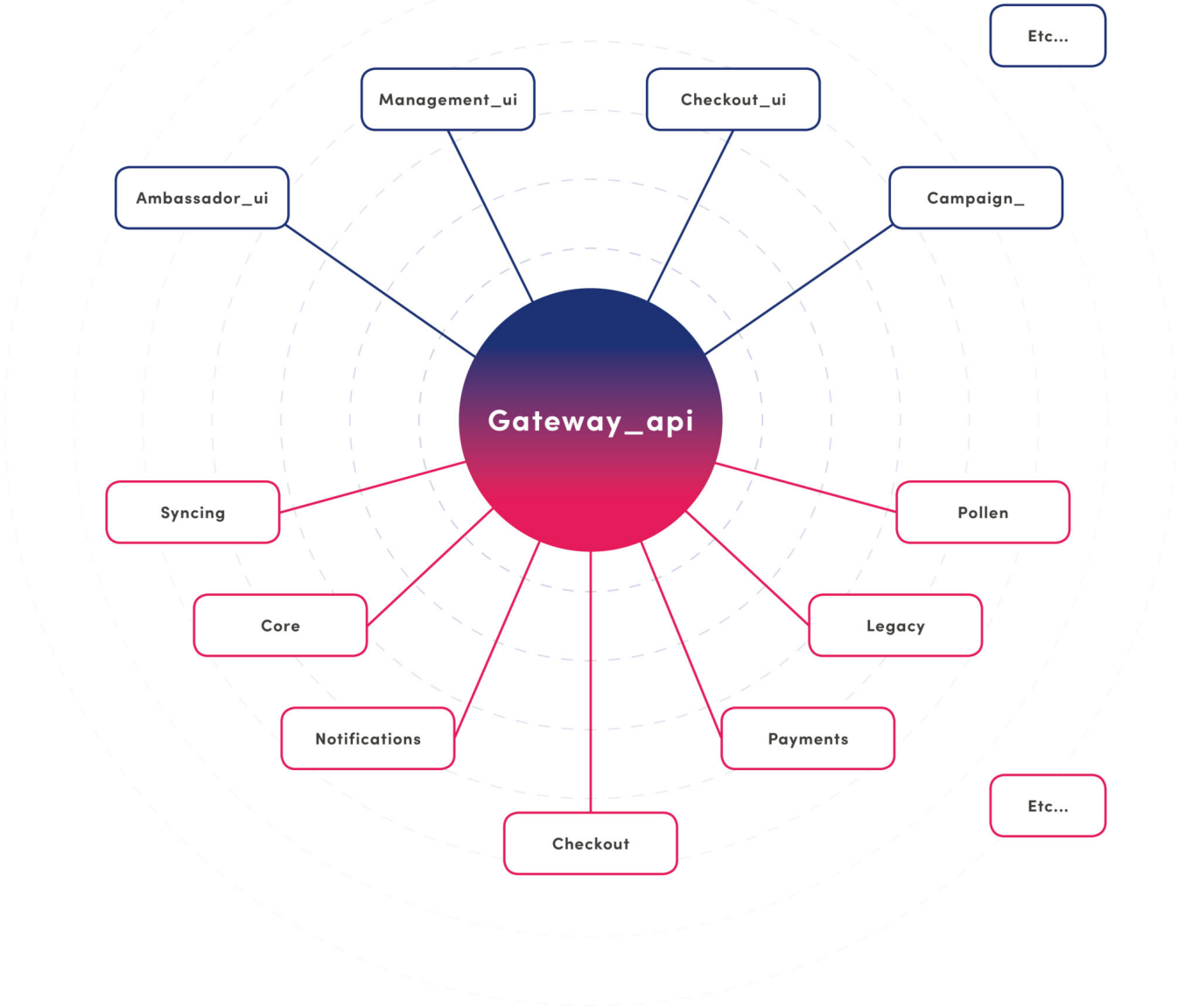
We use a single service as our GraphQL server, which in turn makes requests to various backend services – most of which have REST APIs, but because they only communicate with the gateway they're free to use anything they like. The gateway is designed to be completely stateless, which is great for scalability. Caching happens within the GraphQL gateway, so it's easy to scale up our whole system by just scaling up the number of gateway instances.

API gateways aren't the norm in the GraphQL world, so why do we use them, even though it means making additional requests from the gateway to backend services? For us, the biggest reason is reducing API interdependencies. Without a gateway, our service structure would look roughly like this:



There are many services talking to many other services, resulting in a number of API connections that grows roughly quadratically with the number of services. This is not only impossible to keep in anyone's brain, it also adds a lot of complexities when dealing with outages, maintenance and API changes.

GraphQL can still help with backwards-compatibility even in a network like this, but here's what happens when you put a simple gateway in between services:



Suddenly, there is only a linear number of connections, with each new service adding exactly one new connection to the network graph. API changes only need to affect their source service and the gateway.

The API Gateway is the only way that services communicate with each other, which cuts down complexity drastically. It also creates a great central location for caching, scaling, monitoring and profiling. Generally, having a single service responsible for so many things isn't a great idea – it's a single point of failure.

However, the API Gateway is stateless. It has no database, no local resources and no authentication. This means it can be horizontally scaled

extremely easily, and because it's also responsible for caching, just increasing the number of gateway instances helps significantly with traffic spikes.

Of course, a gateway isn't free: a single request is now turned into two, and if a backend service wants to communicate with another backend service, it now has to go via the gateway. That's great for creating a more maintainable central interface, but it doesn't look great for performance. This is where a stateless gateway suddenly really shines. Because it doesn't matter where the gateway code runs, there is nothing stopping us from just having every backend service act as its own gateway. Instead of making two network requests, we move the GraphQL interface into each service and make requests directly, without using a GraphQL server at all, while still maintaining all the advantages of a central GraphQL schema. And because we have the GraphQL schema defined in Python, we decided to go one step further and make this useable directly in Python, by auto-generating an API wrapper out of the GraphQL schema.

The result is that code which communicates between services can now look like this:

```
In [10]: from streetteam.graph.api import api

In [11]: campaign = api.get_campaign('1b65bcf1-ebd1-4fea-b51c-be6493c62625')

In [12]: campaign.title
Out[12]: 'Bestival 2018'

In [13]: type(campaign)
Out[13]: streetteam.graph.api_types.CampaignApi

In [14]: campaign.brand.title
Out[14]: 'Bestival'
```

The API wrapper for the GraphQL schema is completely auto-generated from the graphene schema, so the services don't even need a copy of the schema file. There are no wasted requests, authentication is handled transparently in the background and fields are lazily resolved as they are accessed.

Now, there are some requirements to being a good API citizen in an environment like this. The backend APIs can mostly do anything they like, but they do have to play nice with how we do caching and permission checking. The rules we use in our backend APIs look like this:

AVOID NESTING OBJECTS, JUST RETURN IDS OF RELATED OBJECTS

Returning nested objects in a REST API can be a great way to reduce the number of requests needed, but it also makes caching extremely difficult and it can cause overfetching, which GraphQL is supposed to combat. We generally avoid bigger, more complicated requests in favour of more but easier-to-cache and flatter requests.

IF YOU DO NEED TO NEST, *NEVER* NEST OBJECTS WITH ADDITIONAL PERMISSIONS

Sometimes the performance needs outweigh the need for simplicity and we will return, for example, a long list of related objects nested within an API response. However, we only ever do this if the permissions for the nested objects are no more strict than for the parent object, because the response otherwise becomes impossible to cache.

We use graphene and graphene-django to actually run our server. We don't use graphene-django's ability to map Django models automatically because all data comes from external requests, we only use it for compatibility with the rest of our stack and its familiarity. The whole gateway server is essentially just a single GraphQLView, which we have extended slightly to allow for some improvements for our frontend:

- We prettify errors to break out Django REST Framework errors from the backend servers. DRF can have more than one error per field, which doesn't natively work in graphene-django, so we extended the view to give accurate error information for every field.
- We extended the error logging to more easily report various errors. 4xx errors for example can mean actual user errors, but because the gateway calls a different API it can also mean that the gateway is using our own API

wrong. DRF doesn't log 4xx errors on the backend service, so we do this in the Gateway when an error is actually caused by us, and not the user.

- Monitoring: the GraphQLView is a great place to add various bits of performance monitoring. We track execution time per query, hashing the queries to make it easy to aggregate response times for the same query with different inputs.

GraphQL is great for us, but we also made plenty of mistakes. We sometimes struggle with keeping our API truly backwards-compatible and had to invest in extra monitoring for deprecated fields in addition to performance monitoring and better error reporting. Updating the GraphQL schema with every API change manually can be tedious, and by making it extremely easy to communicate between backend services via GraphQL we accidentally broke some of our own service boundaries. But in the end, it has helped us develop faster while keeping the mental model of our infrastructure easy and giving our teams significantly more autonomy.

This post was written by Rob Kirberich, Verve's Principal Engineer and resident LED specialist.

TERMS & CONDITIONS

Registered in England under registration No 09750608. VAT No 231457819.