Big Picture (Architecture)

GraphQL has been released only as a *specification*. This means that GraphQL is in fact not more than a long document that describes in detail the behaviour of a *GraphQL server.*

Use Cases

In this section, we'll walk you through 3 different kinds of architectures that include a GraphQL server:

1. GraphQL server *with a connected database*
2. GraphQL server that is a *thin layer in front of a number of third party or legacy systems* and integrates them through a single GraphQL API
3. A *hybrid approach of a connected database and third party or legacy systems* that can all be accessed through the same GraphQL API

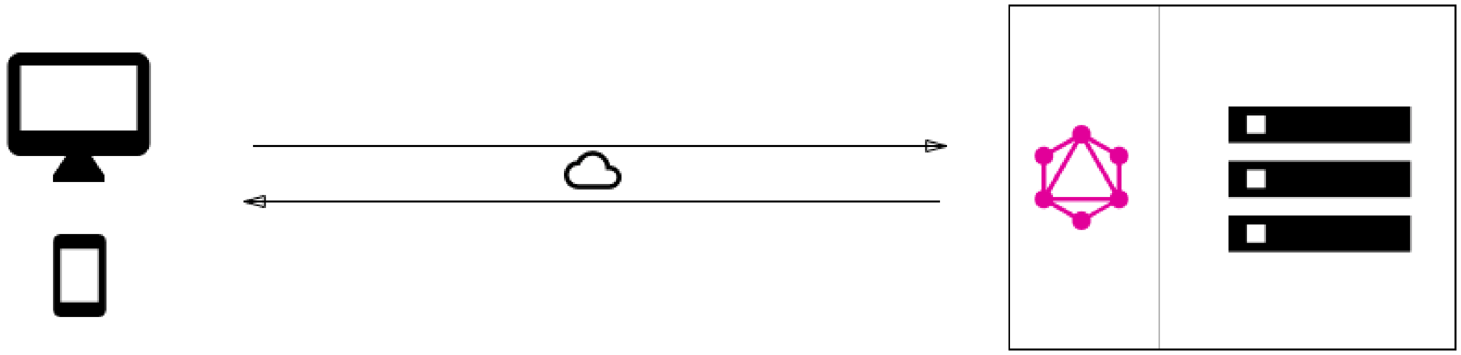
All three architectures represent major use cases of GraphQL and demonstrate the flexibility in terms of the context where it can be used.

1. GraphQL server with a connected database

This architecture will be the most common for *greenfield* projects. In the setup, you have a single (web) server that implements the GraphQL specification. When a query arrives at the GraphQL server, the server reads the query's payload and fetches the required information from the database. This is called *resolving* the query. It then constructs the response object as described in the official specification and returns it to the client.

It's important to note that GraphQL is actually *transport-layer agnostic*. This means it can potentially be used with any available network protocol. So, it is potentially possible to implement a GraphQL server based on TCP, WebSockets, etc.

GraphQL also doesn't care about the database or the format that is used to store the data. You could use a SQL database like AWS Aurora or a NoSQL database like MongoDB.



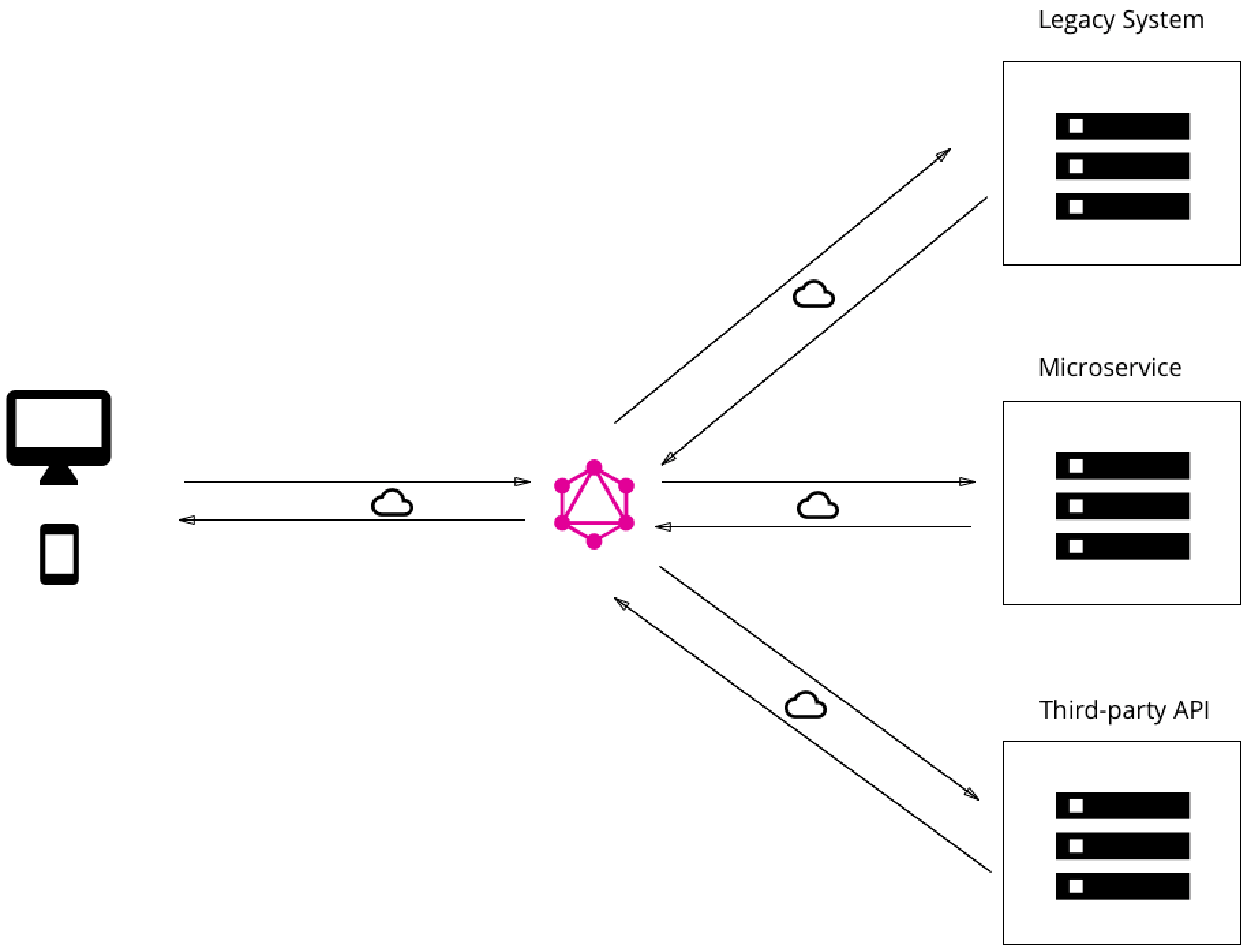
A standard greenfield architecture with one GraphQL server that connects to a single database.

2. GraphQL layer that integrates existing systems

Another major use case for GraphQL is the integration of multiple existing systems behind a single, coherent GraphQL API. This is particularly compelling for companies with legacy infrastructures and many different APIs that have grown over years and now impose a high maintenance burden. One major problem with these legacy systems is that they make it practically impossible to build innovative products that need access to multiple systems.

In that context, GraphQL can be used to *unify* these existing systems and hide their complexity behind a nice GraphQL API. This way, new client applications can be developed that simply talk to the GraphQL server to fetch the data they need. The GraphQL server is then responsible for fetching the data from the existing systems and package it up in the GraphQL response format.

Just like in the previous architecture where the GraphQL server didn’t care about the type of database being used, this time it doesn’t care about the data sources that it needs to fetch the data that’s needed to *resolve* a query.

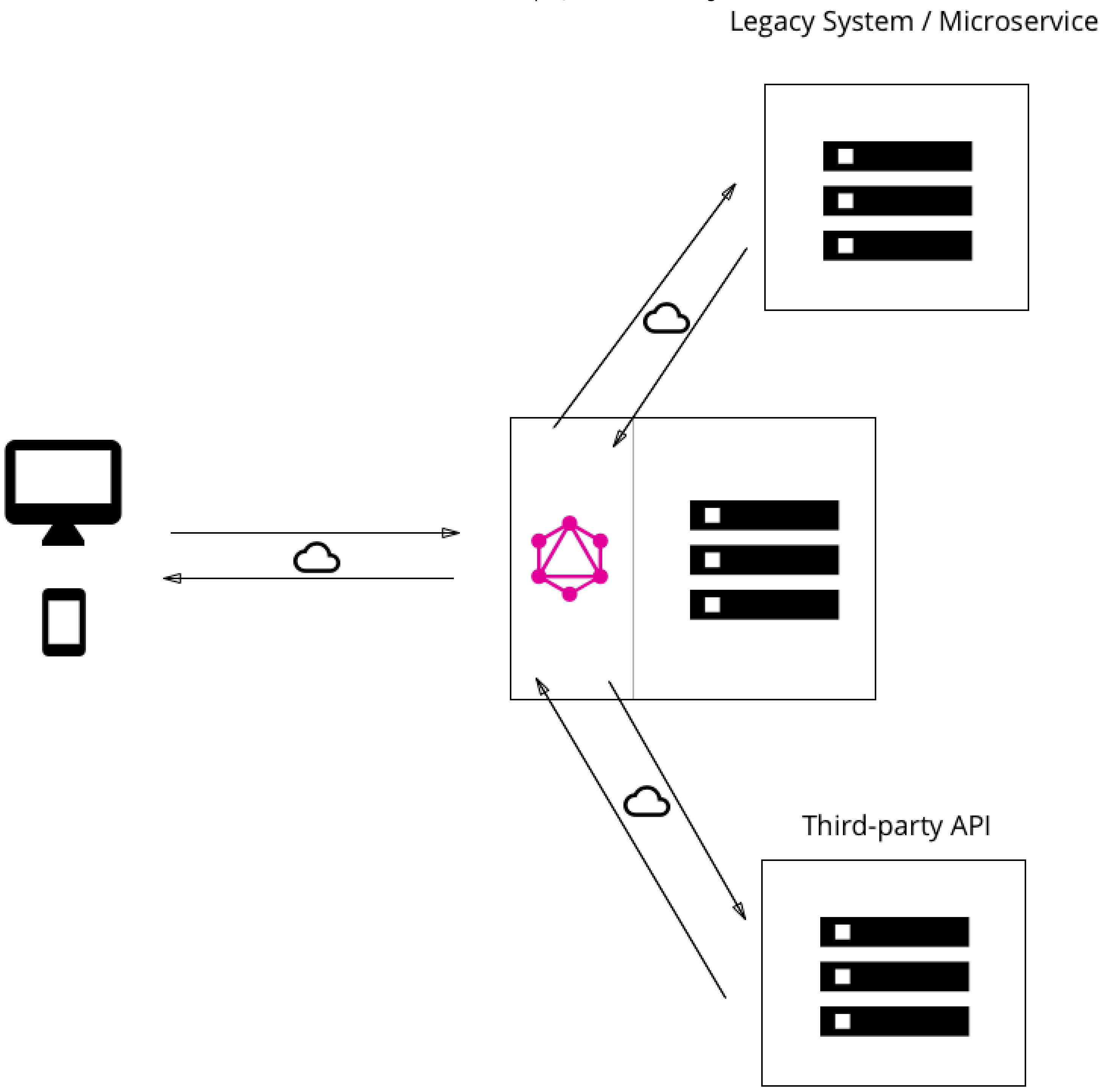


GraphQL allows you to hide the complexity of existing systems, such as microservices, legacy infrastructures or third-party APIs behind a single GraphQL interface.

3. Hybrid approach with connected database and integration of existing system

Finally, it’s possible to combine the two approaches and build a GraphQL server that has a connected database but still talks to legacy or third—party systems.

When a query is received by the server, it will resolve it and either retrieve the required data from the connected database or some of the integrated APIs.



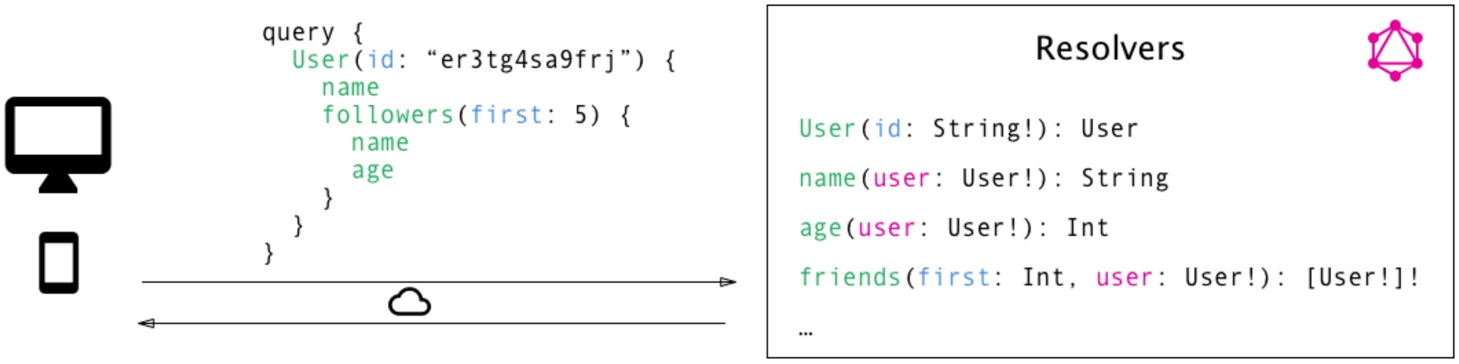
Both approaches can also be combined and the GraphQL server can fetch data from a single database as well as from an existing system - allowing for complete flexibility and pushing all data management complexity to the server.

Resolver Functions

But how do we gain this flexibility with GraphQL? How is it that it's such a great fit for these very different kinds of use cases?

As you learned in the previous chapter, the payload of a GraphQL query (or mutation) consists of a set of *fields*. In the GraphQL server implementation, each of these fields actually corresponds to exactly one function that's called a *resolver*. The sole purpose of a resolver function is to fetch the data for its field.

When the server receives a query, it will call all the functions for the fields that are specified in the query's payload. It thus *resolves* the query and is able to retrieve the correct data for each field. Once all resolvers returned, the server will package data up in the format that was described by the query and send it back to the client.



```
query {
  User(id: "er3tg4sa9frj") {
    name
    followers(first: 5) {
      name
      age
    }
  }
}
```

```
User(id: String!): User

name(user: User!): String

age(user: User!): Int

friends(first: Int, user: User!): [User!]!

…
```

The above screenshot contains some of the resolved field names. Each field in the query corresponds to a resolver function. The GraphQL calls all required resolvers when a query comes in to fetch the specified data.

GraphQL Client Libraries

GraphQL is particularly great for frontend developers since it completely eliminates many of the inconveniences and shortcomings that are experienced with REST APIs, such as over- and underfetching. Complexity is pushed to the server-side where powerful machines can take care of the heavy computation work. The client doesn't have to know where the data that it fetches is actually coming from and can use a single, coherent and flexible API.

Let's consider the major change that's introduced with GraphQL in going from a rather imperative data fetching approach to a purely declarative one. When fetching data from a REST API, most applications will have to go through the following steps:

1. construct and send HTTP request (e.g. with `fetch` in Javascript)
2. receive and parse server response
3. store data locally (either simply in memory or persistent)
4. display data in the UI

With the ideal *declarative data fetching* approach, a client shouldn't be doing more than the following two steps:

1. describe data requirements
2. display data in UI

All the lower-level networking tasks as well as storing the data should be abstracted away and the declaration of data dependencies should be the dominant part.

This is precisely what GraphQL client libraries like Relay or Apollo will enable you to do. They provide the abstraction that you need to be able to focus on the important parts of your application rather than having to deal with the repetitive implementation of infrastructure.