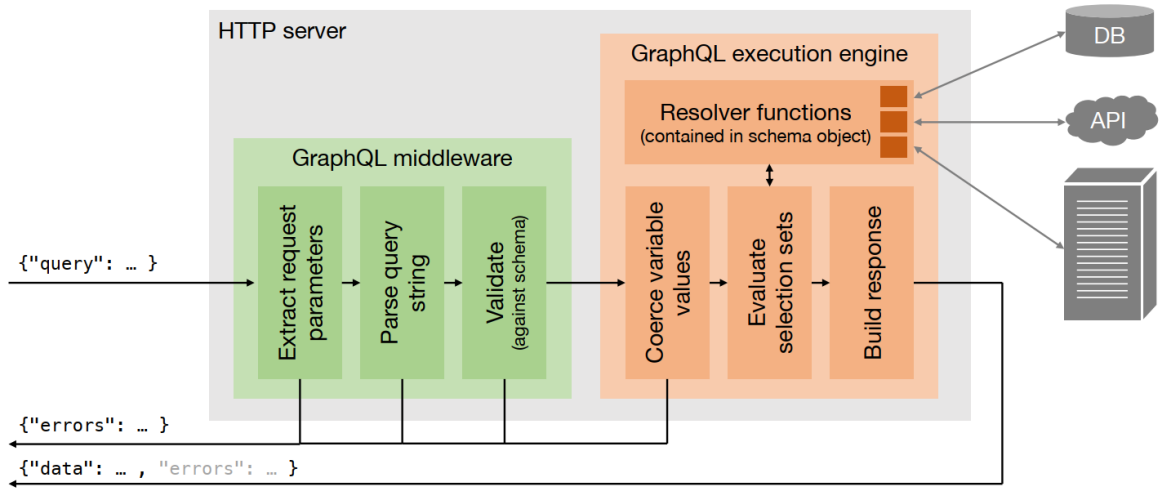
GraphQL

- GrapQL is a **query language** for networked APIs and a **runtime** for servers to fulfill queries
- GraphQL shifts control to what data is returned by the API to the clients
- The client sends a specific query at runtime and the server responds with the according data at runtime
- Clients send type-checked queries and servers respond with requested data

Query execution on a (HTTP) server



Introspection

> **Introspection** is a mechanism for clients to learn (at runtime) about the data types and operations a GraphQL server offers.

- An **introspection query** is a plain-old GraphQl query, that happens to select **meta-fields** provided by **introspection types**
- Client-tools like GraphQL rely on introspection for:
  - Showing documentation about types and operations
  - Client-side query validation
  - Auto-completion when typing queries
  - ...
- Example:

```
query IntrospectionQuery {
  __schema {
    queryType { name }
    mutationType { name }
    subscriptionType { name }
    types {
      ...FullType
    }
    directives {
      name
      locations
      args {
        ...InputValue
      }
    }
  }
}

... Directive Definitions ...
```

Pros&Cons (or only Pros?)

- Static typing
  - auto-complete
  - validation
- Fewer queries
- Predictable responses
- No over-fetching (due to more specific queries)
- In-sync documentation of type system

GraphQL benefits for providers

- Happy API consumers (because nice API)
- Simplified maintenance
  - Serve clients with diverse, changing requirements with a single endpoint
  - GraphQL API self-documents types & operations
- Improved performance and operations
  - Avoid loading / caching / exposing unneeded data
  - Understand data-use on a per-field level
- Compose heterogenous backend resources

Challenges

HTTP caching of GraphQL requests

- Problems with typical HTTP proxy/gateway caches include:
  - Often, non-safe & non-idempotent `POST` is used to send (large) queries
  - Some queried fields may become stale sooner than others, making it hard to define `Chache-Control` / `Last-modiefied` headers
- Alternatives include:
  - Cache persisted queries in proxy or gateway
  - Client-side caching based on `ID` field
  - Application caches in the data-layer ("DataLoaders") or resolver functions

Rate-limiting & Threat prevention

```
query fetchAllTheData {
  users (limit: 1000) {
    orders (first: 1000) {
      paymentDetails { # calls external API
        status
      }
    }
  }
}

= ~1000s of REST requests!
```

- Servers may need to deal with excessive queries sent by clients
  - Rate-limiting and not "x requests per time-interval"
  - Pricing requests
  - Blocking (inadvertently) threatening requests
- Options include:
  - Timeouts against threatening requests
  - Dynamic analysis
  - Static analysis
    - Query "depth" or "nesting"
    - Query "cost" or "complexity"