Practical GraphQL with Symfony & Doctrine

Miro Hudak <mhudak@enscope.com>

PHP UG DD — 23. May 2019

About Me

- Software Engineer
- Started programming in early 1990s in QBasic on 386
- Experience with desktop and server software in consumer and enterprise segments
- Currently C#, PHP and Swift and software architecture
- … and photography, electronics, LEGO, flying, etc.

Motivation

REST API

- Let's assume JSON payload over HTTP, which is now de-facto standard REST API implementation;
- We need to define required server API interfaces (methods, DTOs) in cooperation with team developing client application(s);
- Often, adding a new consumer implementation (another client application) leads to adding new interfaces to avoid overfetching and underfetching;
- This leads to more API interface versions and code bloat.

Overfetching and Underfetching

- Call to an API method yields results in specified format
- Results are defined by API schema, returning fixed amount of fields and entities in specified DTOs; This can be:
  - … more information we need - overfetching - resulting in unnecessary data processing and transfer.
  - … less information we need - underfetching - resulting in another API calls (e.g. via hypermedia links), therefore more I/O (network) latency, application slowdown and more possible failures.
- Flexible returned fields are possible with custom API fetching routines, but that is often clunky and hard to maintain.

Documentation

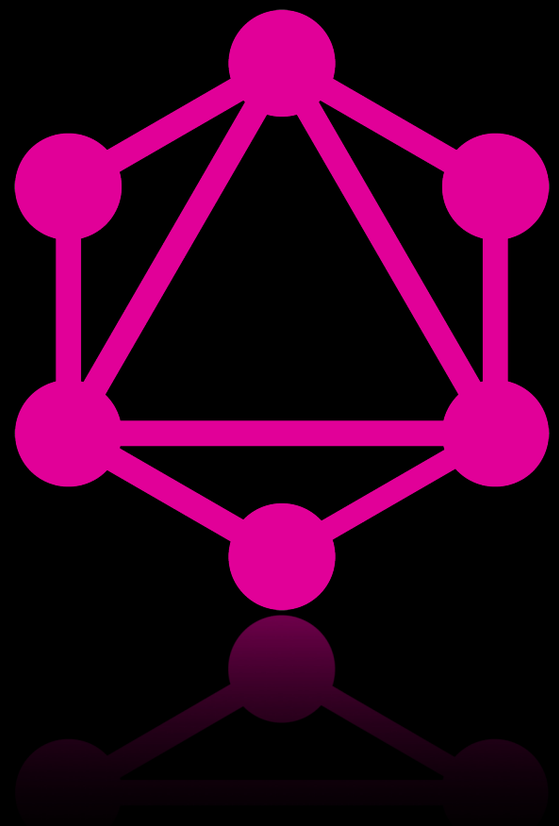
- API must be properly documented, therefore it is necessary to either maintain the documentation by hand or extend API to allow self-documentation.

- How many APIs have you seen using Swagger?

GraphQL?

What is GraphQL?

- Open-source data query and manipulation language
- Developed internally by Facebook in 2012, published in 2015, moved to separate GraphQL Foundation in 2018
- Client and server libraries available in all major programming languages and environments
- https://www.graphql.org



So how does it help us?

- Schema is defined on server;
- Clients can specify, which fields and sub-entities are they interested in; therefore server implementation can expose more data than needed and client receives only data it really needs;
- Only one API endpoint, actual method calls are specified in *queries* or *mutations* *(or subscriptions, for long-running requests)*.
- Schema documentation is part of the specification and it is trivial to add description of queries, mutations and fields during schema definition.

Query

- Fetching data from server;
  must not modify the data!
- Begins with `query` keyword,
  but does not have to, as it is
  implied when it is not
  provided
- Query name can be provided
- Can be parametrised to
  specify what data should be
  retrieved

```
query BasicQuery {
  title(id: "tt0848228") {
    id
    title: primaryTitle
    releaseYear
    rating
  }
}
```

Nomenclature

- Operation - query, mutation, subscription
- Selection Set - set of required information
- Field - a piece of information
- Field alias - an alternative name of the field
- Argument - behaviour altering parameter to field
- *... and many other entities, which we are not about to discuss here to keep it simple and comprehensible.*

Type System

- Scalar types `Int`, `Float`, `String`, `Boolean`
- `ID` is a special `String` type, which indicates unique identifier and can be used as such; GraphQL server implementation can coerce the type when necessary and assure it's uniqueness
- Non-scalars `Null`, `Enum`, `List`, `Object`
- Special types `Union`, `Interface` and `Input Object`
- It is possible to add user-defined types
- Can be marked non-nullable by appending `!` (`String!`)

Issuing a Query

- Best way to issue a query is via some GraphQL client library (you can check the list at https://graphql.org/code/) or an API tool (Graph*i*QL, Insomnia,...)
- Low-level, GraphQL is usually a JSON POST request with *query string* in request body
- It is also possible to issue GET request with `?query={query}`
- When named query is provided, `"operationName"` indicates, which query to perform, allowing multiple queries per request

```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: insomnia/6.5.3
Content-Type: application/json
Accept: */*
Content-Length: 140

{"query":"query BasicQuery {\n
title(id: \"tt0848228\") {\n
id\n    title: primaryTitle\n
releaseYear\n  }
\n}","operationName":"BasicQuery"
}
```

Query Result

- Resulting data is in well-formed JSON format with HTTP 200 OK
- Successful query result contains `"data"` key and no `"errors"` key

```
{
  "data": {
    "title": {
      "id": "tt0848228",
      "title": "The Avengers",
      "releaseYear": 2012,
      "rating": 8.1
    }
  }
}
```

Query Error

- Error response should be well-formed JSON as well
- Failure query result contains `"errors"` key and can contain `"data"` key
- Specification requires error item to contain `"message"` and `"path"` keys, optionally `"locations"` key as well
- HTTP status code is `200 OK`, as GraphQL processing was still successful

```
{
  "errors": [
    {
      "message": "Not found.",
      "extensions": ...,
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "title"
      ]
    }
  ],
  "data": {
    "title": null
  }
}
```

Nested Objects

- You can nest objects up to depth limit specified in your server implementation
- To obtain nested object, you need to specify fields that should be returned
- Depending on an actual implementation of *resolver*, each nested object may require additional database access

```
query NestedObjects {
  title(id: "tt0848228") {
    id
    title: primaryTitle
    releaseYear
    rating
    principals {
      name: primaryName
      birthYear
    }
  }
}
```

Field Parameters

- Schema can specify parameters for fields
- These parameters can be used to specify details of retrieved field; e.g. specify type of objects to return or specify constraints for pagination

```
query FieldParameters {
  title(id: "tt0848228") {
    id
    title: primaryTitle
    releaseYear
    rating
    names(type: "actor") {
      name: primaryName
      birthYear
    }
  }
}
```

Traditional Pagination

- It is not specified, how to implement pagination, so it is up to the developer
- Pagination and filtering is provided as parameters
- Pagination query returns metadata and items array with requested fields

```
query OffsetPagination {
  titles(
    primaryTitle: "The Avengers",
    type: "movie",
    offset: 3)
  {
    offset
    limit
    filteredCount
    items {
      title: primaryTitle
      releaseYear
      rating
    }
  }
}
```

Cursor-based Pagination

- More flexible than traditional pagination but more complicated to implement
- Provided by third party frameworks like Relay
- Cursor is usually base64-encoded data, that can be used to continuation of listing

```
query CursorPagination {
  titles(
    title: "The Avengers",
    type: "movie",
    first: 3,
    after: "c29tZS1jdXJzb3I=")
  {
    totalCount
    edges {
      node {
        title: primaryTitle
        releaseYear
        rating
      }
      cursor
    }
    pageInfo {
      endCursor
      hasNextPage
    }
  }
}
```

Mutations

- Expected to modify the data on server
- Begins with `mutation` keyword
- Syntax is the same as with queries, it is only semantical difference

```
mutation BasicMutation {
  AddToWatchList(
    id: "tt0848228")
  {
    id
    primaryTitle
    releaseYear
    inWatchlist
  }
}
```

Enough theory...!

Example Application

- The rest of this talk will refer the example application
- It is available on GitHub at: https://github.com/enscope/talk-graphql

```
composer create-project ↩
  symfony/website-skeleton ↩
  talk-graphql-example
```

- Application requires public IMDb dataset (large!), but there is a small subset included in sources
- Basis of the application is nothing special, just a usual empty Symfony skeleton

```
composer require ↩
  symfony/apache-pack

composer require --dev ↩
  symfony/web-server-bundle
```

GraphQL Bundle

- Example implementation uses `Overblog/GraphQLBundle`
- https://github.com/overblog/GraphQLBundle
- Current version can be integrated with Symfony 3.1+
- Additionally, installation of *GraphiQL* for development environment is recommended

```
composer require ↩
  overblog/graphql-bundle

// accept all contribs

composer require --dev ↩
  overblog/graphiql-bundle

// accept all contribs
```

Schema Configuration

- Main configuration of the bundle is in `config/graphql.yaml`
    - Application is set to pre-generate schema classes to enhance performance and then point composer auto-loader to these classes
- Routes are provided in `config/routes/graphql.yaml`
- All types are located in `config/graphql/types` with `Query.types.yaml` and `Mutation.types.yaml` being the root query and mutation respectively
    - All other files refer to specific types in schema, which are directly or indirectly connected to the root query or mutation

Schema Type

- YAML file with root key as the name of the custom type
- `type:` a type of the custom type
- `config:` configuration contains:
  - `description:` human-readable string provided by tools as a documentation for this custom type
  - `resolveField:` resolver link; the resolver is usually an `InvokeResolver`, then fields do not require separate resolvers
- `fields:` is a list of fields available in this schema; each field has a `type:` and can have `description:`, `resolve:` and `args:`
  - `args:` contains arguments that can be provided to field to alter it's behaviour; each argument has a `type:` and `description:`

Resolvers

- Each type has a *Resolver*, which is a class, that provides the actual data for a result or performs the mutation
- Resolvers are located in `GraphQL\Resolver` namespace, but actual namespace assignment is up to the developer
- A resolver class is specified via `resolve:` key in configuration:
  - Resolver implementing `ResolveInterface` is for queries
  - Resolver implementing `MutationInterface` is for mutations
- Resolver method can be specified using double-colon `::` or it can implement `InvokeResolver` and the `__invoke()` is used to obtain a result value. Example application combines both approaches.

Query Resolver

- In example application, almost all query resolvers are extended from `AbstractInvokeResolver`, which implements `__invoke()` method, that automatically performs *getter* when specific resolve method is not found
- Also, type resolve is abstracted to `doResolveInternal( ... )`
- Query resolvers implement `doResolveInternal( ... )`; other fields only on as-needed basis.
- `AbstractPaginatingResolver` adds additional functionality for resolvers, which provide pagination

Mutation Resolver

- Implemented in a similar way to Query Resolver
- Root Mutation must be specified in `config/graphql.yaml`
- Actual fields of Root Mutation type file are the same
- Resolver is specified via `@=mutation`
- Mutation resolver must implement `MutationInterface`
- Example application provides `WatchlistMutation` resolver

Security

- Access Token authentication based on built-in security
- Example application provides `MockApiUserProvider`, which provides one single user for any `OAuth2` access token
- GraphQL schema defines access and visibility rules:
  - `access:` defines an expression, which controls access to the field; when evaluates to `false`, error is thrown, when field is accessed
  - `public:` defines an expression, which controls visibility of the field in schema; when evaluates to `false`, it is not possible to use the field
- The same applies to mutation operations

Debugging

- GraphQL is best debugged via some API tool as issuing direct requests is tedious and error prone…
  - … that is why we've installed Graph*i*QL for `development` environment and it will available at localhost:8000/graphiql whenever you run `./bin/console server:run`.
- Another option is *Insomnia*, which now supports GraphQL as well *(unlike Paw…)*; https://insomnia.rest

Demo Time!

Questions?

Thank You!

- Example application uses IMDb public data source, which is needed to run in local environment
- Source code: https://github.com/enscope/talk-graphql
- If you have further questions or inquiries, feel free to contact me at mhudak@enscope.com
- Thank you for your attention!