GraphQL

Introduction

- What is GraphQL
  - API *specification* Open Sourced by Facebook in 2015
    - Developed to address two primary shortcomings of REST: *flexibility* and *efficiency*
    - Described as being compatible with any language and framework
  - Key conceptual difference to REST
    - REST provides unique resource endpoints that return fixed data structures
    - GraphQL provides a single endpoint that responds to queries
      - This is described as 'declarative data fetching' because clients express what they need and receive nothing more
- Three primary motivations behind GraphQL
  - Conserving network: increased mobile requires more efficient data loading
  - Abstraction: a heterogeneous client-side landscape makes it difficult to build a single REST API that meets all client needs
  - Flexibility: it is easier to change a client that obtain data from a single queryable endpoint than it is to modify the design of a REST API

Core concepts

- Implements a type system that you define with a 'schema' for the API. Schemas are typically a collection of GraphQL types. The schema is one of the most important concepts in GraphQL because it represents the *contract between client and server: clients can expect these X capabilities and here's how you access them*
  - Schemas are defined with the Schema Definition Language (SDL)

```
type Person {
  name: String!
  age: Int!
}
```

  - Relations

```
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
```

```
type Post {
  title: String!
  author: Person!
}
```

- CRUD - request / response
  - Queries: reading 'fields'
    - Simple

```
{
  allPersons {
    name
    age
  }
}
```

    - With arguments (if specified in the schema)

```
{
  allPersons(last: 2) {
    name
    age
  }
}
```

    - Get all Persons and, for each of them, all of their posts

```
{
  allPersons {
    name
      posts  {
        title
      }
  }
}
```

      - GraphQL seems to simplify the process of retrieving relations
        - Compare this to something 'RESTful' like https://github.com/orgs/nationalarchives/teams/digital-services/members
  - Mutations provide for: Creating, Updating and Deleting
    - Create a person and send me the created person's ID in the response.

```
mutation {
  createPerson(name: "Bob", age: 36) {
    id
  }
}
```

      - Creating and retrieving in a single round-trip
- Subscriptions: sockets
  - Client subscribes to a new Person being created. Whenever that event happens, the server pushes this information to the client

```
subscription {
  newPerson {
    name
    age
  }
}
```

    - Provide for 'realtime' updates
    - Initiates a continuous connection to the server (socket). This is a stream rather than the request/response cycle elsewhere in GraphQL

Key architectural points

- GraphQL is only a specification that describes how a GraphQL server has to behave
  - You write your own implementation (but there are many reference implementations available)
  - GraphQL is transport layer agnostic and can be used with any available network protocol (incl. TCP or WebSockets etc.)
- Schema and Types
  - Uses a strong type system to define the capabilities of an API
  - All types that are exposed by the API are described in a schema (using GraphQL Schema Definition Language)
    - This schema serves as a contract between client and server
      - Which allows frontend, backend - or different micro services - teams to work independently of each other
- Resolver functions
  - A GraphQL server has one resolver function per field (and the sole purpose of the function is to retrieve the data for that field)
- Architectural use cases (not necessarily mutually exclusive):
  - GraphQL with a connected database
    - Here the GraphQL server resolves queries and constructs a response from data it retrieves from a database
  - GraphQL as a thin integration layer in front of a number of third party or legacy systems.
    - Acts to unify many different (existing) APIs and hide complexity of data fetching logic
      - This might include databases, web services, 3rd party APIs etc.
    - Provides considerable flexibility for the creation and amendment of clients

GraphQL vs REST

- Pros
  - Avoids overfetching and underfetching
    - Example: details page, related research guide, traces through time
    - In REST: 3 requests with a lot of unnecessary data being requested
      - Commonly dealt with in supposedly RESTful applications by creating endpoints that provide only the data needed for a specific view
        - But this tight coupling slows the ability to respond to changed requirements for the front end
    - In GraphQL: single request
      - POST request - the body of which contains a *query* that describes all the data requirements of the client
        - Note: this means that more information is required at the point of request
  - Supports fine grained analytics about what data is being requested by clients
    - This information can be used to evolve an API to deprecate fields that are no longer needed (because you can see what they are)
  - Versioning
    - Because GraphQL APIs are open for extension this avoids problems associated with the common practice of versioning RESTful APIs.
- Cons
  - Error handling (in HTTP)
    - Always results in a 200 status code, with the error message provided in the response
  - Loss or URIs and 'resources'
  - Loss of HTTP caching
    - While REST can take advantage of HTTP caching, GraphQL clients must manage caching themselves