GraphQL APIs for Gateways

• Query language and runtime
• Facebook tech
• MIT-licensed
• Better *Different* approach to API design

“GraphQL” and its logo are ©2018 Facebook Inc.
Query Language: Intro

- SQL, Xpath, Gremlin, etc
- Example schema:
• Summarize the effort behind Stockfish

```sql
SELECT u.name, u.org, array_agg(pg.group_name) AS groups 
FROM project_group_rel pg 
LEFT JOIN group_user_rel gu ON gu.group_name = pg.group_name 
INNER JOIN users u ON u.name = gu.user_name 
WHERE project_name = 'Stockfish' 
GROUP BY u.name, u.org;
```

| name  |        org        | groups          |
|-------+-------------------+-----------------|
| Ada   | Cyberdyne Systems | {ML}            |
| Sally | Cyberdyne Systems | {UX,ML}         |

![Diagram showing relationships between Stockfish, Sally, Ada, UX, ML, and SkyNet]
• Summarize the effort behind Stockfish

In:
{
  projects(name: "Stockfish") {
    groups {
      name
      users { name org }
    }
  }
}

Out:
{
  "project": {
    "groups": [
      {
        "name": "ML",
        "users": [
          {
            "name": "Ada",
            "org": "Cyberdyne Systems"
          },
          {
            "name": "Sally",
            "org": "Cyberdyne Systems"
          }
        ]
      },
      {
        "name": "UX",
        "users": [
          {
            "name": "Sally",
            "org": "Cyberdyne Systems"
          },
          {
            "name": "UX",
            "users": [n
              {
                "name": "Sally",
                "org": "Cyberdyne Systems"
              }
            ]
          }
        ]
      }
    ]
  }
  /* ... various closing brackets ... */
Query Language: Type System

- **Server:**
  ```graphql
  type Project {
    name: String!
    alignment: Alignment!
    groups: [Group]!
  }
  type Group {
    name: String!
    title: String!
    projects: [Project]!
    users: [User]!
  }
  type User {
    name: String!
    org: String!
    groups: [Group]!
  }
  enum Alignment {
    lawful good
    neutral good
    ... etc
  }
  type Query {
    project(name: String!): Project
    projects(): [Project]!
    group(name: String!): Group
    groups(): [Group]!
    user(name: String!): User
    users(): [User]!
  }
  ```
Runtime: vs. REST

**REST**

GET /projects/Stockfish/groups
for $groupURL in groups:
  GET $groupURL/users
  for $userURL in $users:
    GET $userURL

**GraphQL**

GET /graphql?query={
  projects(name: "Stockfish") {
    groups {
      name
      users { name org }
    }
  }
}

type Query {
  project(name: String!): Project
  projects(): [Project]!
  group(name: String!): Group
  groups(): [Group]!
  user(name: String!): User
  users(): [User]!
}
## Runtime: vs. REST

### REST

- **GET** /projects/Stockfish/groups
  - for $groupURL in groups:
    - **GET** $groupURL/users
      - for $userURL in $users:
        - **GET** $userURL

- **GET** /projects/Stockfish/summary
- **GET** /projects/Stockfish
  - ?include=groups,users

### GraphQL

```
GET /graphql?query=

projects(name: "Stockfish") {
  groups {
    name
    users { name org }
  }
}
```

```
type Query {
  project(name: String!): Project
  projects(): [Project]!
  group(name: String!): Group
  groups(): [Group]!
  user(name: String!): User
  users(): [User]!
}
```
### Runtime: vs. REST

#### REST

- GET `/projects/Stockfish/groups`
  - for `$groupURL` in `groups`:
    - GET `$groupURL/users`
    - for `$userURL` in `$users`:
      - GET `$userURL`

- GET `/projects/Stockfish/summary`
- GET `/projects/Stockfish`
  - `?include=groups,users`
- GET `/projects/Stockfish/related`

#### GraphQL

- GET `/graphql?query={
    projects(name: "Stockfish") {
        groups {
            name
            users { name org }
        }
        projects { name }
    }
}

```graphql
type Query {
  project(name: String!): Project
  projects(): [Project]!
  group(name: String!): Group
  groups(): [Group]!
  user(name: String!): User
  users(): [User]!
}
```
Agility & Expressiveness, at a price

**Pros**
- Responds well to requirement change
- Less dependency between front- and back-end teams
- Supports multiple consumer types readily
- Consistent
- Less latency

**Cons**
- Non-standard aesthetic
- More latency
- “Production-izing” more costly