A Journey to GraphQL

TV 2 Denmark is a commercial broadcasting station based in Denmark. The first broadcast was sent out in 1988 from one single channel - today we broadcast on 6 different channels.

Beside traditional broadcasting TV we run a video streaming service called TV 2 PLAY serving both video on-demand and TV 2s own live channels

Jimmi Kristensen
Software Engineering Manager
Works @ TV 2 PLAY

@jimmibk

TV2 / PLAY

GR Conf EUROPE 2019

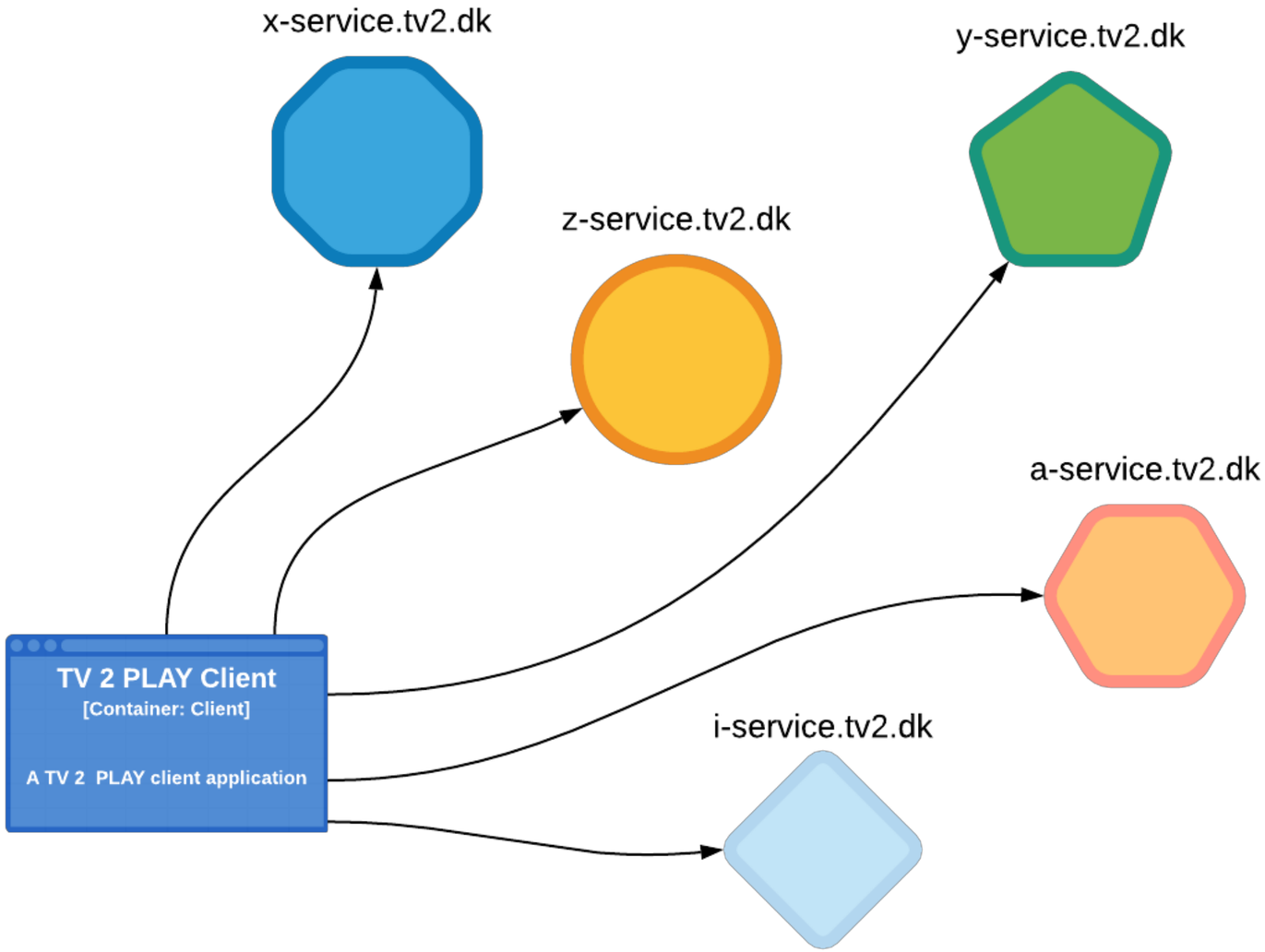
A Bit of History

- **2004:** TV 2s streaming services launched under the name **TV 2 Sputnik**
  - Built on the LAMP stack
- **2012:** Rebranded under its current name **TV 2 PLAY**
  - Still on a LAMP stack
- **2014-2015:** started to buckle under its own success

Breaking down the Monolith

- We knew the concepts but had no experience, so the first few attempts failed
- Eventually we got a good grasp of it and succeeded
- We chose the JVM instead of PHP
- We knew insight into each microservice was essential - we chose Datadog
- Ran with Docker for containerization
- Chose Jenkins as our CI/CD tool
- Started out on Rancher for container orchestration
- Later transitioned to Kubernetes

Breaking down the Monolith

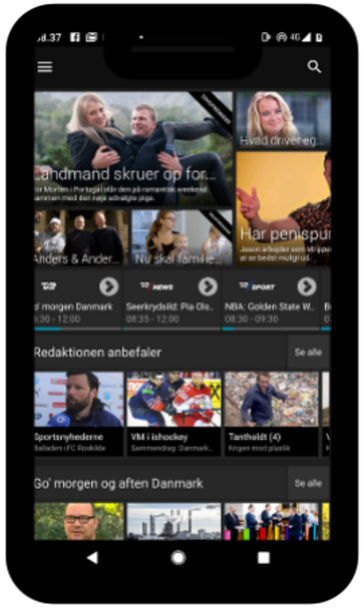


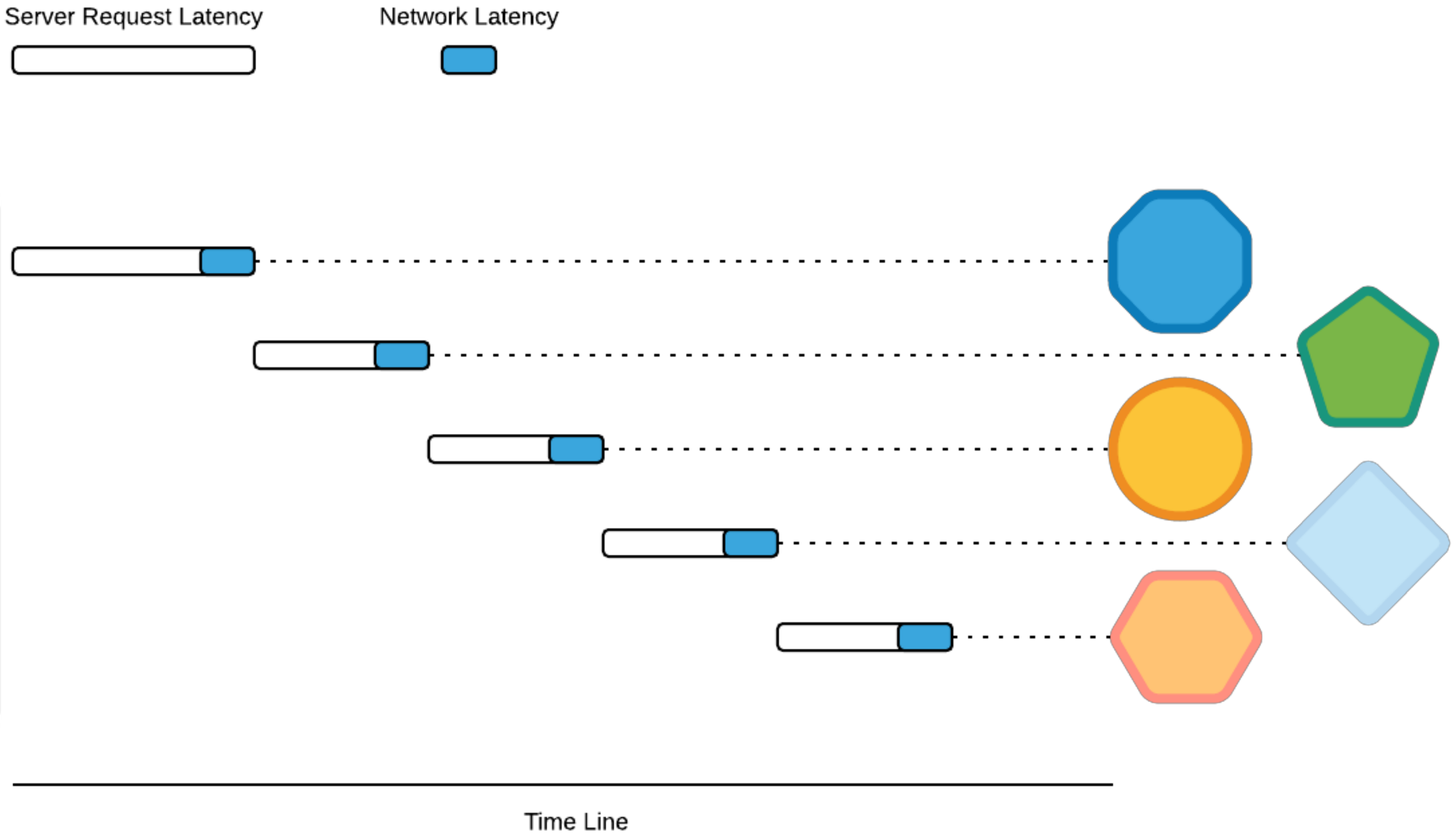
A Complex Client Landscape

Different Challenges

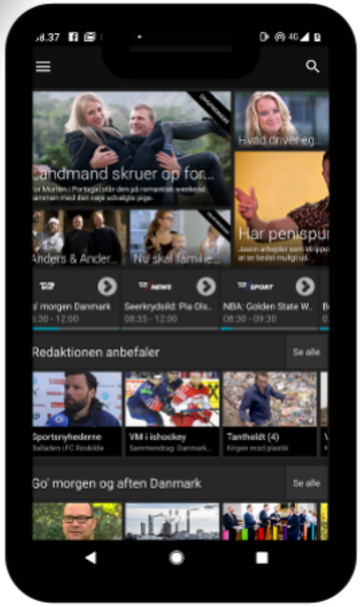
- Business logic sneaked into the client app
- Different teams works on different clients
- Documentation was interpreted differently by each team
- Implementation ended up being different for each client
- A general lag of collaboration and communication
- Not all clients had the same needs and requirements

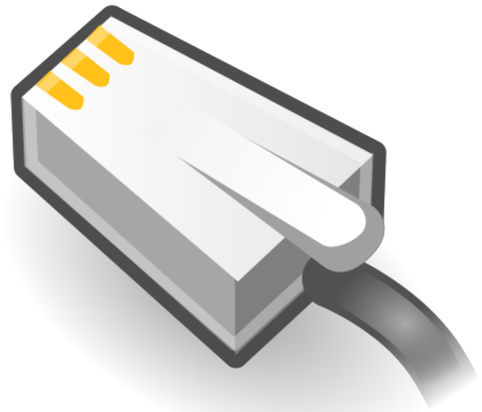
Fat Clients

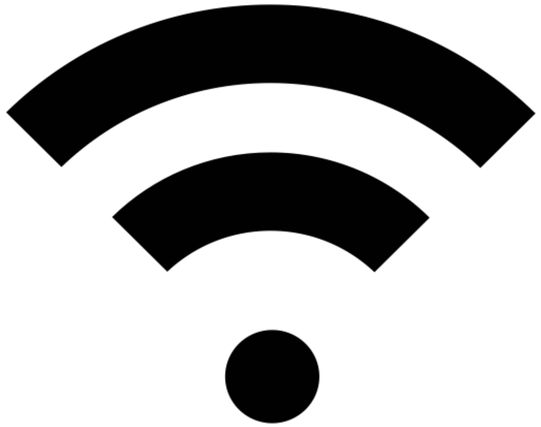

Redaktionen anbefaler
Se alle
Go' morgen og aften Danmark
Se alle


Server Request Latency
Network Latency
Time Line

Client Diversity

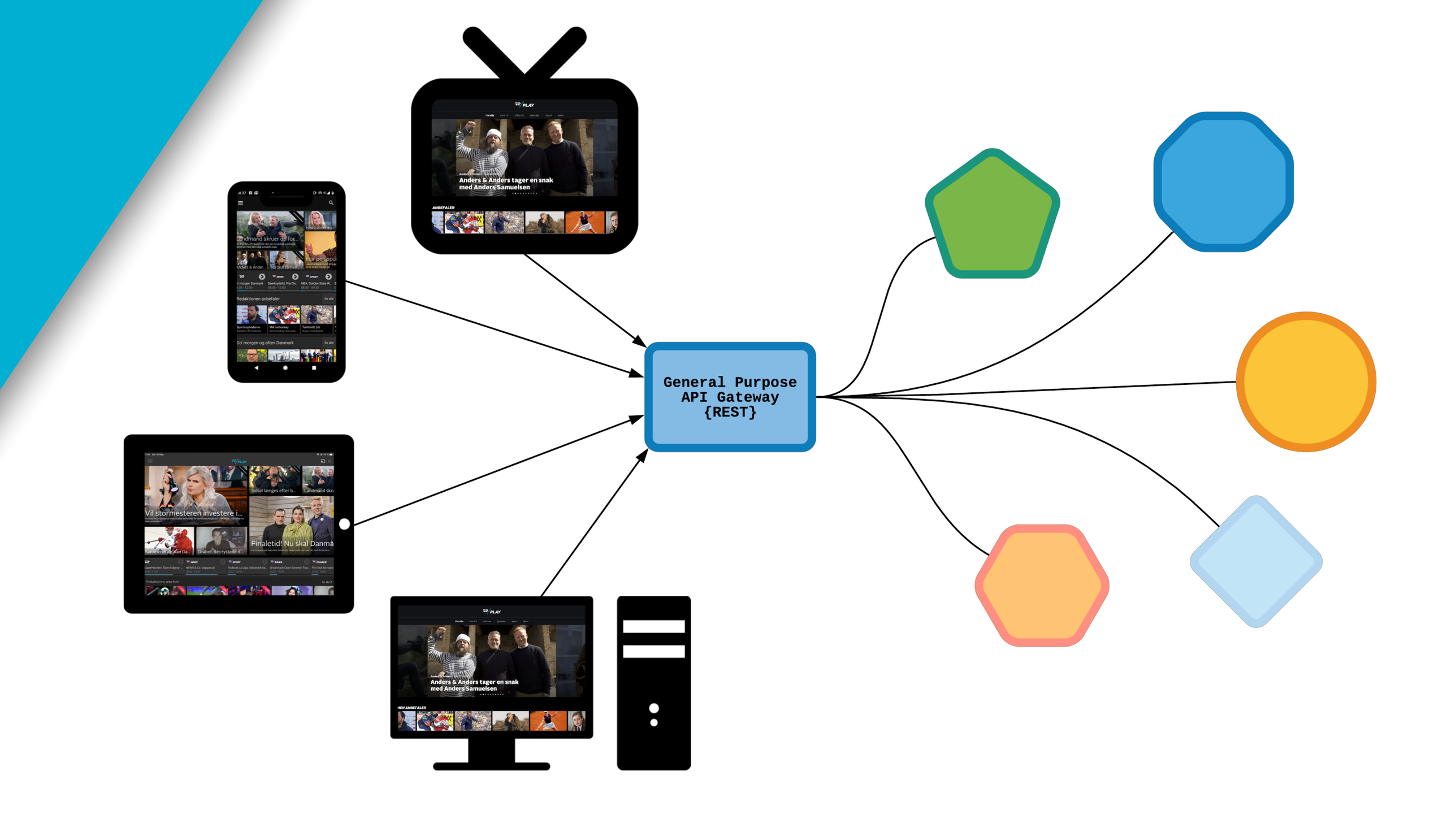


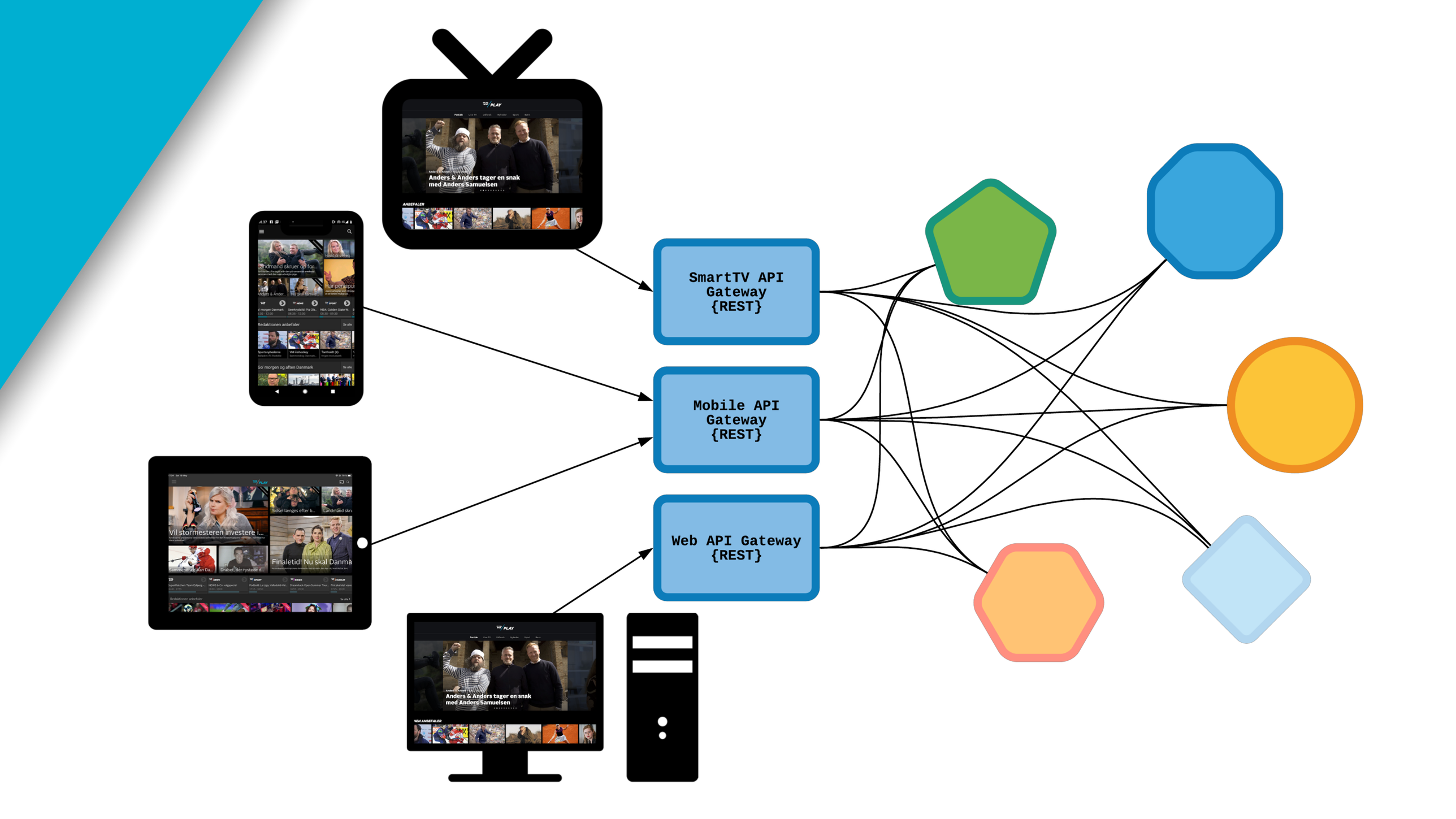




Our Findings

- Standard REST endpoints was not the ideal solution to support the needs of our different client apps
- We wanted thinner clients by getting rid of the business logic in the client to maximize consistency across client apps
- The client app should not have to know the context and host name of each microservice


Anders & Anders tager en snak med Anders Samuelsen
Redaktionen anbefaler
Go' morgen og aften Danmark
Vil stormesteren investere i...
Finaletid! Nu skal Danma
General Purpose API Gateway {REST}


Anders & Anders tager en snak med Anders Samuelsen
ANBEFALER
Redaktionen anbefaler
Go' morgen og aften Danmark
Vil stormesteren investere i...
Finaletid! Nu skal Danma
SmartTV API Gateway {REST}
Mobile API Gateway {REST}
Web API Gateway {REST}

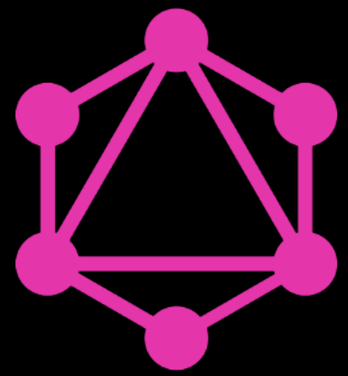
Different Approaches

Both approaches removes the business logic from the client app
but...

- **General Purpose API Gateway** requires specialized endpoints or a large number of query parameters to cope with client diversity
- **BFF API Gateways** is able to handle the specialized needs of the different clients, but what about shared functionality?
- We eventually looked into **GraphQL**

What is GraphQL?

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more.



What does that mean?

Some things we really liked

- Schema based
- Uses the well known HTTP protocol usually via POST messages
- Is a query language meaning client developers decide the returned data set (deterministic)
- Supports both reads and writes via Queries and Mutations
- Is typed language

Some things where we needed to rewire our brains

- POST message for both reads and writes?
- Queries and Mutations instead of GET, POST, PUT, DELETE, PATCH, etc.
- We had to think about caching a bit different

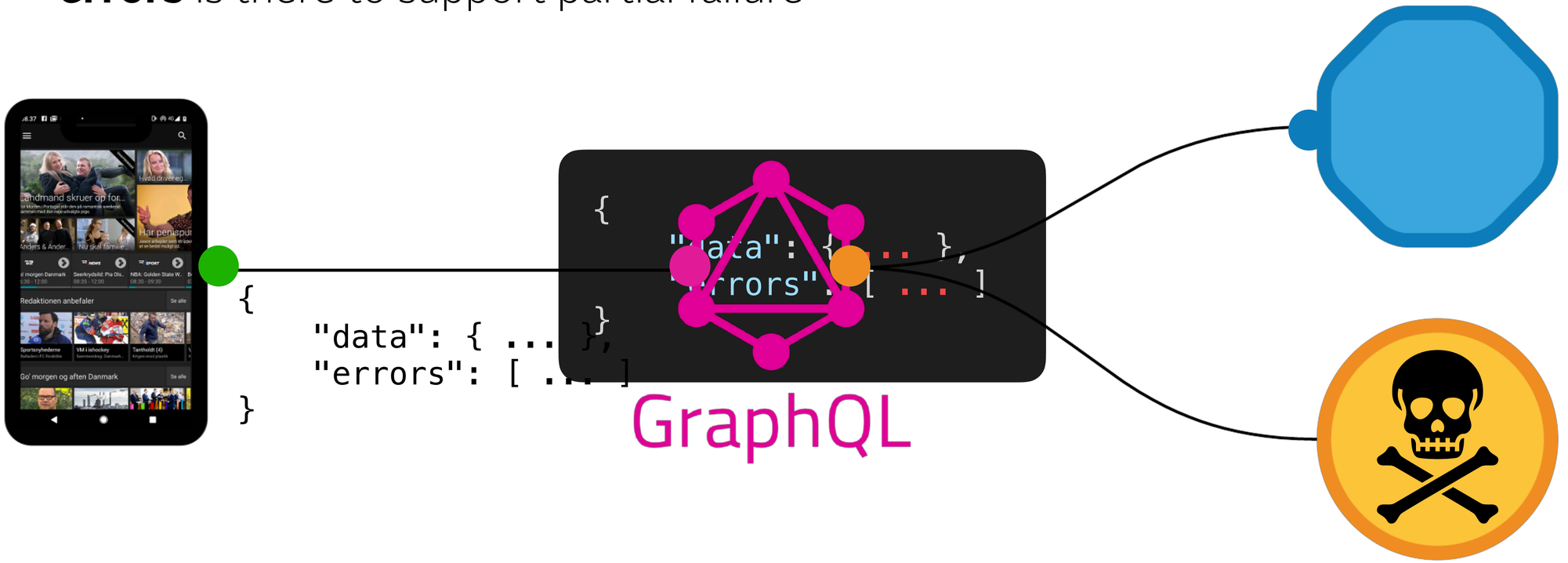
Requesting GraphQL Server

- Is usually served over HTTP as a GET or POST request
- Is formatted as JSON of content type application/json
- **query** contains your GraphQL query
- **variables** is a JSON formatted set of variables used in the query
- **operationName** contains the name of the query operation
- Some server implementation for the JVM:
  - **Java** has the graphql-java library
  - **Micronaut** support via the micronaut-graphql module which gives you a *GraphQLController*

```
{
    "query": "",
    "variables":"",
    "operationName":""
}
```

Response from Server

- Regardless of transport protocol, the response should be returned as JSON
- The response might contain some data and some errors
  - **data** contains the results
  - **errors** is there to support partial failure



Fields

Asking for specific fields on an object

```
query taxo {
  taxonomy {
    securitySchemes {
      nodes {
        key
        value
      }
    }
  }
}
```

Response has the same structure as the query
It is deterministic - you know what you get


```
{
  "data": {
    "taxonomy": {
      "securitySchemes": {
        "nodes": [
          {
            "key": "widevine",
            "value": "Widevine DRM"
          },
          {
            "key": "playready",
            "value": "PlayReady"
          },
          {
            "key": "fairplay",
            "value": "FairPlay"
          }
        ]
      }
    }
  }
}
```

Arguments

With REST you have a single set of query params.
GraphQL every field and object can have a set of args

```
query ent {
  entity(id: 123456) {
    title
    description
    art(type: "keyframe") {
      nodes {
        url
        type
      }
    }
  }
}
```

```
{
  "data": {
    "entity": {
      "title": "To grader fra en
klimakatastrofe",
      "description": "Den legendariske tv-
vært
og naturhistoriker sir David Attenborough er
vært i denne storslåede BBC-dokumentar,
der ser på konsekvenserne af de
ødelæggende…”,
      "art": {
        "nodes": [
          {
            "url": “https://img.svr.dk/
key.jpeg",
            "type": "keyframe"
          }
        ]
      }
    }
  }
}
```

Aliases

Querying for the same field with different arguments results in conflict
Aliases lets you change the field result to something else

```
query ent {
  entity1: entity(id: 1234) {
    title
    type
  }
  entity2: entity(id: 4321) {
    title
    type
  }
}
```

```
{
  "data": {
    “entity1": {
      "title": "Julia Sofia & Friends",
      "type": "series"
    },
    “entity2": {
      "title": "Fremtidens kød",
      "type": "episode"
    }
  }
}
```

Fragments

Client applications some times needs complicated sets of data
Fragments lets us reuse sets of fields where we need them

```
query ent {
  entity1: entity(id: 1234) {
    ... entityFields
  }
  entity2: entity(id: 4321) {
    ... entityFields
  }
}

fragment entityFields on Entity {
  title
  type
  references {
    web
  }
}
```

```
{
  "data": {
    "entity1": {
      "title": "Julia Sofia & Friends",
      "type": "series",
      "references": {
        "web": “/programmer/dokumentar/serier/
julia-sofia-friends/"
      }
    },
    “entity2": {
      "title": "Fremtidens kød",
      "type": "episode",
      "references": {
        "web": “/programmer/magasiner/serier/
tantholdt/fremtidens-koed-179939/"
      }
    }
  }
}
```

In-line Fragments

GraphQL supports interfaces and unions
Entity is an interface, we use in-line fragments to access the concrete types
Series and **Episode**

```
query ent {
  entity1: entity(id: 1234) {
    title
    type
    ... on Series {
      episodeCount
      seasonCount
    }
  }
  entity2: entity(id: 4321) {
    title
    type
    ... on Episode {
      episodeNumber
      seasonNumber
    }
  }
}
```

```
{
  "data": {
    "entity1": {
      "title": "Julia Sofia & Friends",
      "type": "series",
      "episodeCount": 7,
      "seasonCount": 1
    },
    "entity2": {
      "title": "Fremtidens kød",
      "type": "episode",
      "episodeNumber": 1,
      "seasonNumber": 1
    }
  }
}
```

Variables

To support dynamic arguments we use variables
Avoid doing string interpolation to construct queries from user's input

```
query ent($entityId: Int) {
  entity(id: $entityId) {
    title
    type
    ... on Series {
      episodeCount
      seasonCount
    }
  }
}

# Variables
{
  "entityId": 1234
}
```

```
{
  "data": {
    "entity": {
      "title": "Julia Sofia & Friends",
      "type": "series",
      "episodeCount": 7,
      "seasonCount": 1
    }
  }
}
```

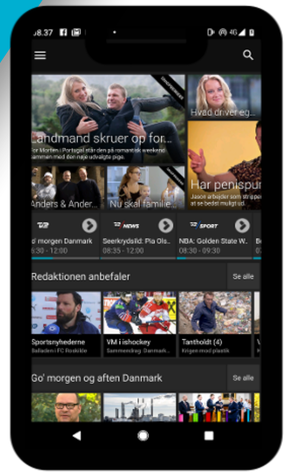
Directives

Directives are another type of variable used to dynamically change the structure of queries

- **@include(if: Boolean)** include this field in the result if arg is **true**
- **@skip(if: Boolean)** skip this field in the result if arg is **true**

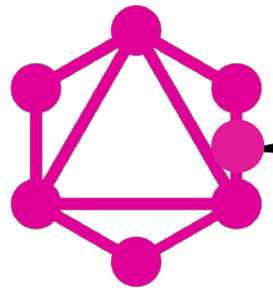
```
query ent($entityId: Int, $includeRef: Boolean!) {
  entity(id: $entityId) {
    title
    type
    references @include(if: $includeRef) {
      web
    }
  }
}

{
  "entityId": 2911,
  "includeRef": false
}
```

```
{
  "data": {
    "entity": {
      "title": "Julia Sofia & Friends",
      "type": "series"
    }
  }
}
```

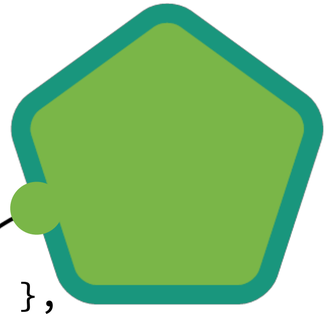


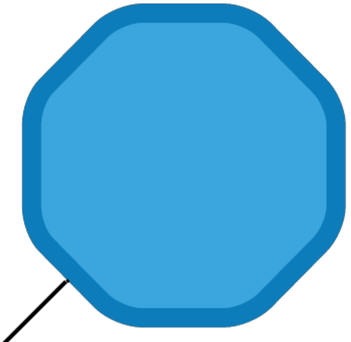

```
page(path: "/" platform: play_ios) {
  title
  path
  structures(limit: 3) {
    nodes {
      title
    }
  }
}
```

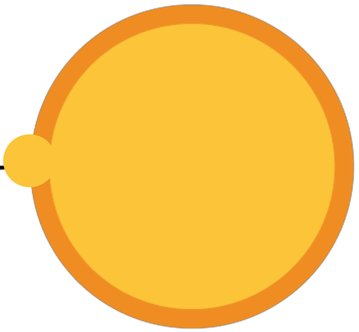
```
"page": {
      "title": "Forside",
      "path": "/",
      "structures": {
        "nodes": [
          { "title": "Vi anbefaler” },
          { "title": "Redaktionen anbefaler” },
          { "title": "I gang med at se” }
        ]
      }
    }
  }
}
```

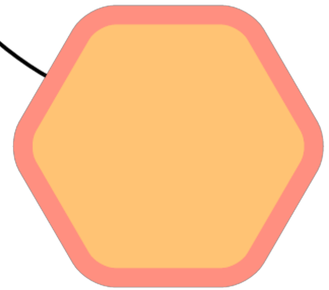


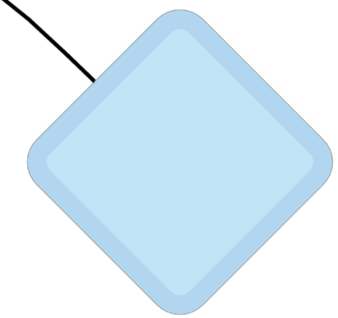
GraphQL

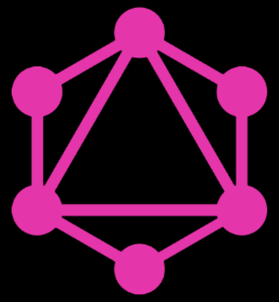












GraphQL

- Query for fetching data objects based on fields
- Mutations for changing data objects
- Supports Interfaces and different implementations
- Is a typed language: Scalars, Lists, Enums, Objects
- Supports both mandatory and optional arguments