WEB 3, Session 5

GraphQL

Apollo Server

Agenda

- **Same-origin Policy**
- GraphQL
- Apollo Server
- GraphQL Client

Same-origin Policy (SOP)

- Browsers are very strict about calling services from another origin (address and port)
- Scenario:
  - You log in to your bank - mybank.com
  - In another tab you are logged in to - evilbankhackers.org
  - Because you are logged in, the evil bank hackers could in theory hijack your session and empty your bank account
- Therefore, SOP doesn't allow a page on evilbankhackers.org to call services (RESTful or otherwise) on mybank.com
- Note: this protection is in the browser. It protect end-users, not the service.

Origins

- An origin consists of
  - protocol
  - host
  - port
- These are different domains:
  - https://mydomain.dk:2348/
  - https://otherdomain.dk:2348/
  - https://mydomain.dk:2234/
  - http://mydomain.dk:2348/

Cross-Origin Resource Sharing (CORS)

- CORS are rules you can set up to loosen SOP
- Another scenario:
  - Your bank wants a new domain: myonlinebank.com
  - But the services are still on mybank.com
  - The bank sets up CORS headers on the responses from mybank.com
  - The browser can now know that myonlinebank is okay to call mybank
- Note: This is not the server blocking calls from specific sites
  - It's the browser not allowing calls unless CORS is set up to allow it

CORS in Express

```
gameserver.use((_, res, next) => {
    res.header("Access-Control-Allow-Origin", "http://localhost:5173");
    res.header("Access-Control-Allow-Headers",
               "Origin, X-Requested-With, Content-Type, Accept");
    res.header("Access-Control-Allow-Methods", "GET, POST, PATCH");
    next();
});
```

Using cors middleware

```
app.use(cors({
  origin: 'http://localhost:5173',
  methods: ['GET', 'POST, OPTIONS']
}))
```

Using regular expressions

```
app.use(cors({
  origin: /:\/\/localhost:/,
  methods: ['GET', 'POST', 'OPTIONS']
}))
```

Dynamic origins

```
function dynamicOrigins(origin?: string) {
  if (origin === undefined) return defaultOrigins
  const portIndex = origin.lastIndexOf(':')
  if (portIndex === -1) return defaultOrigins
  const port = origin.slice(portIndex + 1)
  return `http://localhost:${port}`
}

app.use(cors({
  origin: (origin, callback) => callback(null, dynamicOrigins(origin)),
    methods: ['GET', 'POST, OPTIONS']
}))
```

Agenda

- Same-origin Policy
- **GraphQL**
- Apollo Server
- GraphQL Client

GraphQL

- A standard for making an interface
- Often used for API gateways
- Still need to implement the interface
- We'll be using Apollo Server for that

GraphQL queries only ask for what they need

```
pending_games {
  id
  players
}
```

Scalar types

- Int
- Float
- String
- Boolean
- ID
  - This is a GUID/UUID - a string not a number

GraphQL type

```
type PendingGame {
  id: ID!,
  pending: Boolean,
  creator: String!,
  players: [String!]!,
  number_of_players: Int!
}
```

Arrays

- Arrays are indicated by putting [] *around* the type: [String]
- Example:

```
players: [String!]!
```

!

! means NOT NULL:

```
type Credentials {
  username: String!
  password: String!
}
```

- ! and arrays:
  - `[String]!` - the array is mandatory, but can contain nulls
  - `[String!]` - the array is optional, but can't contain nulls
  - `[String!]!` - the array is mandatory, and can't contain nulls

empty array still allowed

Referring to other types

```
type Score {
  slot: String!,
  score: Int
}

type ActiveGame {
  id: ID!,
  …
  scores: [[Score!]!]!
}
```

Interfaces

```
interface Game {
  id: ID!,
  pending: Boolean!
}

type ActiveGame implements Game {
  id: ID!,
  pending: Boolean!
  …
}

type PendingGame implements Game {…}
```

mandatory - not implied by implements

Unions

```
union Game = ActiveGame | PendingGame
```

Defining Queries and Mutations

- Queries and Mutations are defined using 2 special types type Query and type Mutation
- In order to make Queries and Mutations work, we need to implement *resolvers*
- Queries can be either with or without parameters
- Mutations *can* be without parameters, but will rarely be so
- The parameters of Mutations need to be either primitive types or a special *input* type

type Query

```
type Query {
  games: [ActiveGame!]!,
  game(id: ID!): ActiveGame,
  pending_games: [PendingGame!]!
  pending_game(id: ID!): PendingGame
}
```

type Mutation

not type

```
input BlogInput {
    text: String!,
    username: String!,
    tags: [String!]
}

type Mutation {
  createBlog(blog: BlogInput!): Blog,
  addComment(blogId: ID!, comment: CommentInput!): Comment
}
```

Agenda

- Same-origin Policy
- GraphQL
- **Apollo Server**
- GraphQL Client

Apollo Server

- Implements GraphQL based on
  - Type definitions (.sdl)
  - Resolvers
- A resolver takes (most important)
  - parent - properties from parents (for resolver chaining)
  - args - properties from arguments /parameters
  - context - server-wide properties

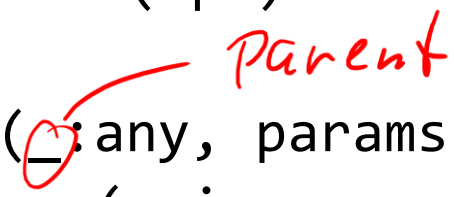
Helper functions for the resolver

```
async function respond_with_error(err: ServerError): Promise<never> {
  throw new GraphQLError(err.type)
}


async function pending_game(api: API, id: string)
    : Promise<PendingGame | undefined> {
  const res = await api.pending_game(id)
  return res.resolve({
    onSuccess: async g => g,
    onError: async e => undefined
  })
}
```

The Query Property of the resolver object

```
Query: {
  async games() {
    return games(api)
  },
  async game(_:any, params: {id: string}) {
    return game(api, params.id)
  },
  …
  async pending_game(id: string) {
    return pending_game(api, id)
  }
},
```



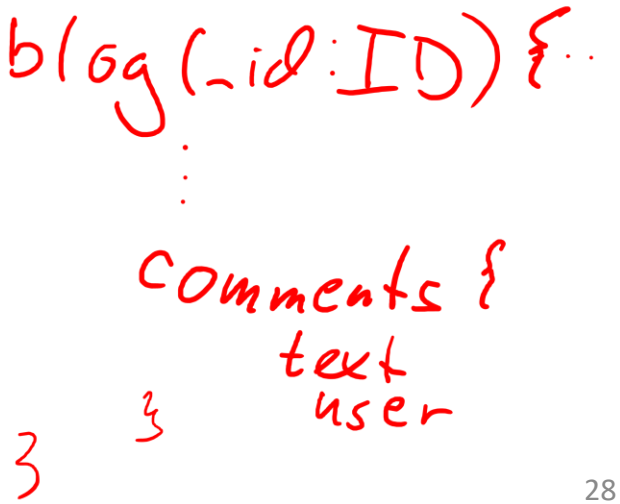
The Mutation property of the resolver object

```
Mutation: {
  async join(_:any, params: {id: string, player: string}) {
      return join(api, params)
   },
   …
},
```

Resolver chaining case

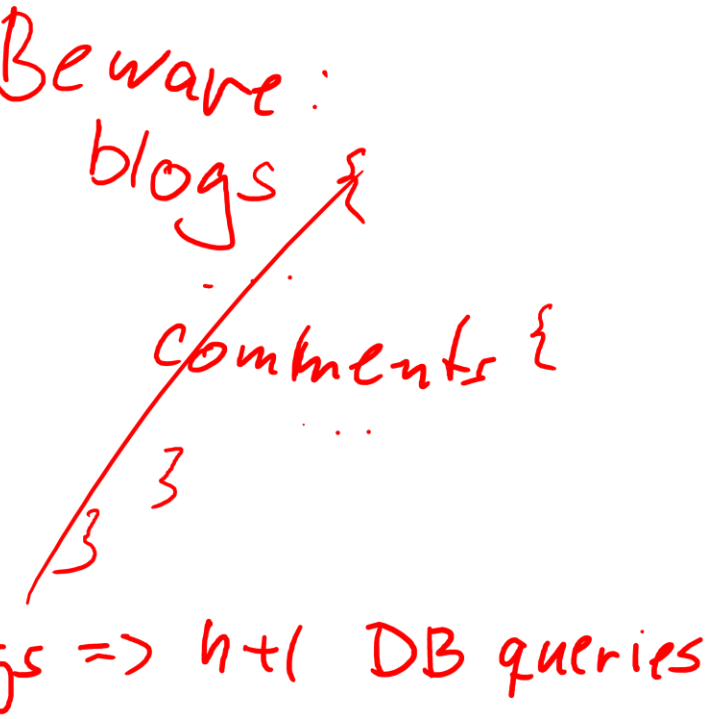
```
type Comment {
    text: String!,
    user: String!,
    likes: [String!]!
}

type Blog {
    _id: ID!,
    …
    comments: [Comment!]!
}
```



Resolver chaining

- Query.blog()
  - Resolver for the blog query
- Blog.comments()
  - Resolver for the comments property of blogs
  - Uses the parent parameter to look up the blog id
- Chaining: Query.blog() --> Blog.comments()
  - When querying blogs
  - If any details from comments are requested
  - Calls Blog.comments()



Resolver for Chaining

```
export async function blogComment(parent) {
    return runSession(db =>
        commentsForBlog(db.collection("blogs"), parent._id)
    )
}
```

Chaining resolver in the resolver object

```
Blog: {
  comments: Resolver.blogComment
},
```

Type Resolver

- Type resolvers are used when a type can be seen as ambiguous
- This might be an interface with several implementations
- It can also be a union type
- The resolver is the same in either case

Type resolver in the resolver object

```
Game: {
  __resolveType(obj:any) {
    if (obj.pending)
      return 'PendingGame'
    else
      return 'ActiveGame'
  }
},
```

Agenda

- Same-origin Policy
- GraphQL
- Apollo Server
- **GraphQL Client**

GraphQL Clients

- GraphQL runs over HTTP(S)
- You can call it with
  - fetch
  - axios
  - RxJS
  - even XMLHttpRequest (God forbid)
- You can also use a dedicated GraphQL Client
  - I use the ApolloClient (install @apollo/client)

Using fetch (I)

```
const query = `query PendingGame($id: String!) {
  pending_game(id: $id) {
    id
    number_of_players
    pending
    creator
    players
  }
}`
const variables = {
  'id': '3'
}
```

Using fetch (II)

```
const headers = {
  'Content-Type': 'application/json',
  Accepts: 'application/json'
}
const res = await fetch('/graphql', {
  method: 'POST',
  body: JSON.stringify({query, variables}),
  headers})
const data = await res.json()
```

ApolloClient (I)

```
import {
  ApolloClient,
  HttpLink,
  InMemoryCache,
  gql } from "@apollo/client/core"

const httpLink = new HttpLink({ uri: 'http://localhost:4000/graphql'})

const client = new ApolloClient({
  link: httpLink,
  cache: new InMemoryCache(),
})
```

important

ApolloClient (II)

```
const query = `query PendingGame($id: String!) {
  pending_game(id: $id) {
    id
    number_of_players
    pending
    creator
    players
  }
}`
const variables = {
  'id': '3'
}
```

ApolloClient (III)

destructuring

returns object { data ... }

Payload

```
const { data } = await apolloClient.query({
  query,
  variables,
  fetchPolicy: 'network-only' })
```

No caching