GraphQL from Backend to Frontend

GraphQL is a query language that allows clients to receive exactly the data the need

Describe your data

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

Ask what you want

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

Get results

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

That's it, thank you for my talk!

Just kidding!
There is much more to it!

Demo!

https://twittergql.fly.dev

Let's go back to the "Describe your data"

GraphQL Schema

- Types
- Input Types
- Interfaces
- Unions
- Enums
- Scalars
- Directives

Types

```
type Tweet {
  content: String!
  id: ID!
  insertedAt: DateTime!
  likes: Int
  likedBy(limit: Int, offset: Int): [User!]
  user: User!
}
```

Input Types

```
input UserInput {
  password: String!
  username: String!
}
```

Interfaces

```
"""An object with an ID."""
interface Node {
  """ID of the object."""
  id: ID!
}
```

```
"""Product."""
type Product implements Node {
  id: ID!
  # ...
}
```

Unions

```graphql
type Book {
  id: ID!
  title: String!
  author: Person!
}
type Movie {
  id: ID!
  title: String!
  cast: [Person!]
}

union Media = Book | Movie

type Library {
  id: ID!
  media: [Media!]
}
```

Enums

```
enum AddressTypeInput {
  billing
  shipping
}
```

Scalars

```
"""
An ISO 8601-encoded datetime
"""
scalar ISO8601DateTime
```

Directives

- @include(if: Boolean!)
- @skip(if: Boolean!)
- @deprecated(reason: String)

```
query Hero($episode: Episode, $withFriends: Boolean!) {
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

Now, let's focus on "Ask what you want"

GraphQL Query

- Queries
- Mutations
- Subscriptions
- Fragments

Queries

```
query Me {
  me {
    id
    username
    avatarUrl
  }
}
```

Fragments

```
fragment Tweet on Tweet {
  id
  content
  insertedAt
  likes
  user {
    ...User
  }
}
```

Mutations

```
mutation SignIn($username: String!, $password: String!) {
  token: signIn(username: $username, password: $password)
}
```

```
mutation CreateTweet($content: String!) {
  createTweet(content: $content) {
    ...Tweet
  }
}
```

Subscriptions

```
subscription OnCommentAdded($tweetId: ID!) {
  commentAdded(tweetId: $tweetId) {
    id
    content
  }
}
```

Now we know what the GraphQL language is, let's explore how Backend and Frontend implementations may look like

Backend approach

- Schema first
- Code first
- We'll focus on the code first approach
- In Elixir!

Basic type

```
1 input_object :user_input do
2   field :username, non_null(:string)
3   field :password, non_null(:string)
4 end
```

Resolver

```elixir
object :user do
  field :id, non_null(:id)
  field :username, non_null(:string)

  field :avatar_url, :string do
    resolve(fn _, %{source: %{id: id}} ->
      index = rem(id, 50)
      {:ok, "https://avatar.iran.liara.run/public/#{index}"}
    end)
  end
end
```

Mutation

```
1  field :sign_in, non_null(:string) do
2    arg(:username, non_null(:string))
3    arg(:password, non_null(:string))
4
5    resolve(fn _, args, _ ->
6      AccountsResolver.sign_in(args)
7    end)
8  end
```

More advanced topics

- Context
- Middlewares
- Dataloaders
- Relay Connection

Middleware and Context

```
1  field :me, :user do
2    middleware(AuthMiddleware)
3
4    resolve(fn _, _args, %{context: %{user: user}} ->
5      {:ok, user}
6    end)
7  end
```

Dataloader

GraphQL solution to N+1 queries

Without Dataloader

```elixir
object :tweet do
  field :id, non_null(:id)
  field :content, non_null(:string)
  field :inserted_at, non_null(:datetime)

  field :user, non_null(:user) do
    resolve(fn _, %{source: %{user_id: user_id}} ->
      Twitter.Users.get_user(user_id)
    end)
  end
end
```

With Dataloader

```
1  object :tweet do
2    field :id, non_null(:id)
3    field :content, non_null(:string)
4    field :inserted_at, non_null(:datetime)
5
6    field :user, non_null(:user) do
7      resolve(dataloader(User))
8    end
9  end
```

Dataloader definition

```elixir
def context(ctx) do
  loader =
    Dataloader.new()
    |> Dataloader.add_source(User, Dataloader.Ecto.new(Twitter.Repo))

  Map.put(ctx, :loader, loader)
end
```

Relay "Connection" specification

```
user {
  id
  name
  friends(first: 10, after: "opaqueCursor") {
    edges {
      cursor
      node {
        id
        name
      }
    }
    pageInfo {
      hasNextPage
      hasPreviousPage
    }
  }
}
```

In practice

```
1  connection field :tweets, node_type: :tweet do
2    resolve(fn pagination_args, _ ->
3      Twitter.Tweets.list_tweets(pagination_args)
4    end)
5  end
```

Time for Frontend

- GraphQL over HTTP
- GraphQL clients
- Codegen
- Normalized cache
- Partial results
- Request policy
- Optimistic responses

GraphQL over HTTP

```
fetch('http://localhost:4000/api/graphql', {
  method: 'POST',
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({
    query: `{
      me {
        id
        username
        avatarUrl
      }
    }`
  })
})
.then(res => res.json())
.then(res => console.log(res.data.me))
```

GraphQL clients

- Apollo Client
- Relay
- AWS Amplify
- urql
- rtk-query-gql
- GraphQL Request
- graphqurl

- graphql-hooks
- GraphQL-WS
- Lokka
- nanographql
- GraphQL-SSE
- Grafoo
- GraphQL-HTTP
- ...and probably many more

urql - query

```
export const MeQuery = gql`
  query Me {
    me {
      id
      username
      avatarUrl
    }
  }
`

const [{ data, fetching, error }] = useQuery(MeQuery)

if (fetching) return <p>Loading...</p>
if (error) return <p>Oh no... {error.message}</p>
return <p>{data.me.username}</p>
```

urql - mutation

```
export const CreateTweetDocument = gql`
  mutation CreateTweet($content: String!) {
    createTweet(content: $content) {
      id
    }
  }
`

const [{data, fetching, error}, executeMutation] = useMutation(CreateTweetDocument);
```


CODEGEN
HD
HISTORY.COM
imgflip.com

@graphql-codegen/cli

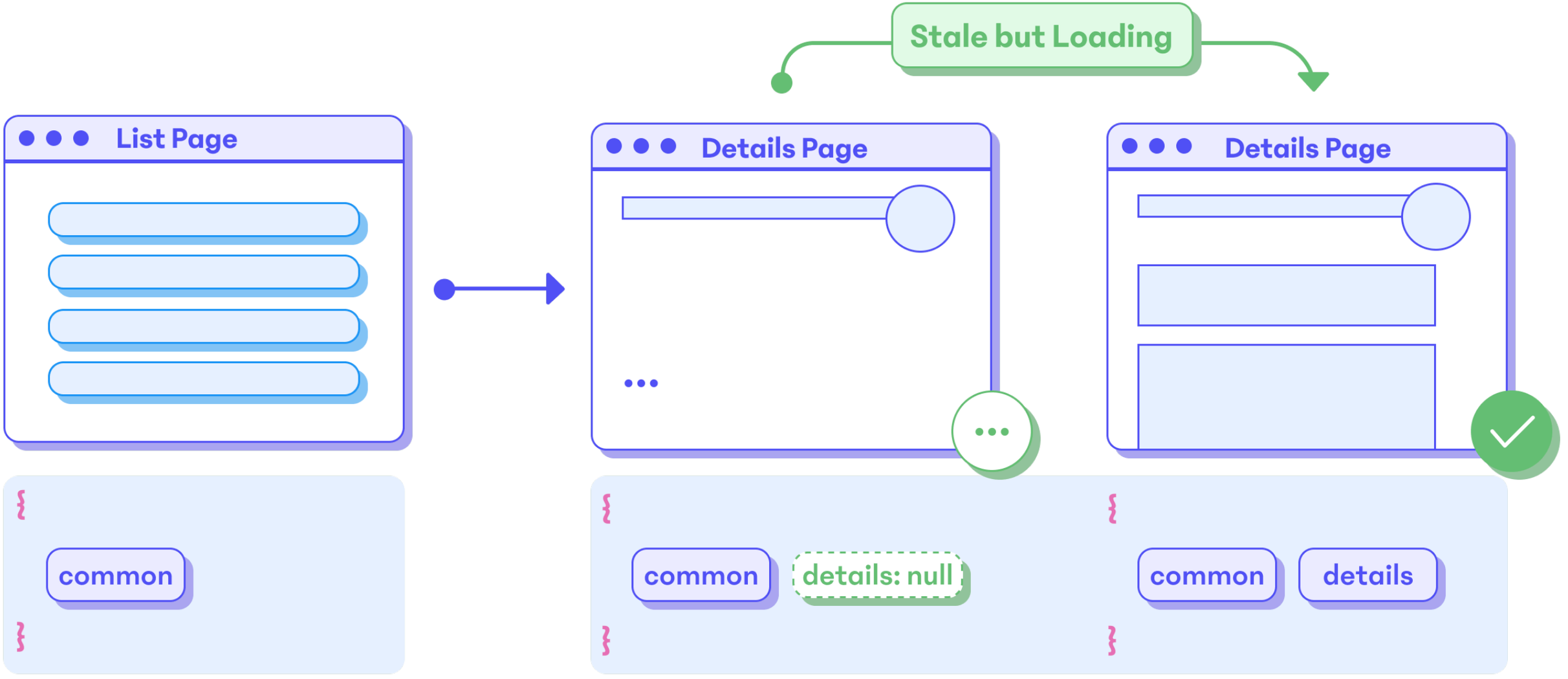
```
1  const config: CodegenConfig = {
2    schema: "http://localhost:4000/api/graphql",
3    documents: ["src/gql/**/*.gql"],
4    generates: {
5      "./schema.gql": {
6        plugins: ["schema-ast"],
7      },
8      "./src/gql/generated.ts": {
9        plugins: [
10         "typescript",
11         "typescript-operations",
12         { "typescript-urql": { withHooks: true } },
13       ],
14     },
15   },
16 };
```

But why?

- Typesafety from Backend to Frontend
- Compile time validation
- Local documentation

More advanced topics

- Normalized cache and partial results
- Request policy
- Optimistic updates


Stale but Loading
List Page
Details Page
Details Page
common
common
details: null
common
details

urql request policy

- cache-first - prefers cached results and falls back to sending an API request when no prior result is cached
- cache-and-network - returns cached results but also always sends an API request, which is perfect for displaying data quickly while keeping it up-to-date
- network-only - will always send an API request and will ignore cached results
- cache-only - will always return cached results or null

Optimistic updates

```
mutation FavoriteTodo(id: $id) {
  favoriteTodo(id: $id) {
    id
    favorite
    updatedAt
  }
}
const cache = cacheExchange({
  optimistic: {
    favoriteTodo(args, cache, info) {
      return {
        __typename: 'Todo',
        id: args.id,
        favorite: true,
      };
    },
  },
});
```

Pros and cons

Pros

- Typed!
- Self documented
- Versionless
- Great for mobile - lower transfer, cache friendly
- Speed of development (codegen, power to the clients)
- Dataloaders
- Great for public APIs (e.g. GitHub API)

Cons

- More complex Frontend
- Normalized cache (and caching) is complex
- Overkill in smaller projects

GraphQL & Rest: A burger comparison

```
https://your-api.com/burger/
```

⇒

```
query getBurger {
  burger {
    bun
    patty
    bun
    lettuce
  }
}
```

⇒


Backend
Frontend
200 OK
200 OK
{
error:
"Internal Server Error",
}
status: 500

Demo!

Thank you!

- https://twittergql.fly.dev
- https://github.com/baransu/gql-twitter
- https://github.com/baransu/gql-twitter-presentation
- https://graphql.org/learn/