Full Stack GraphQL mit Apollo

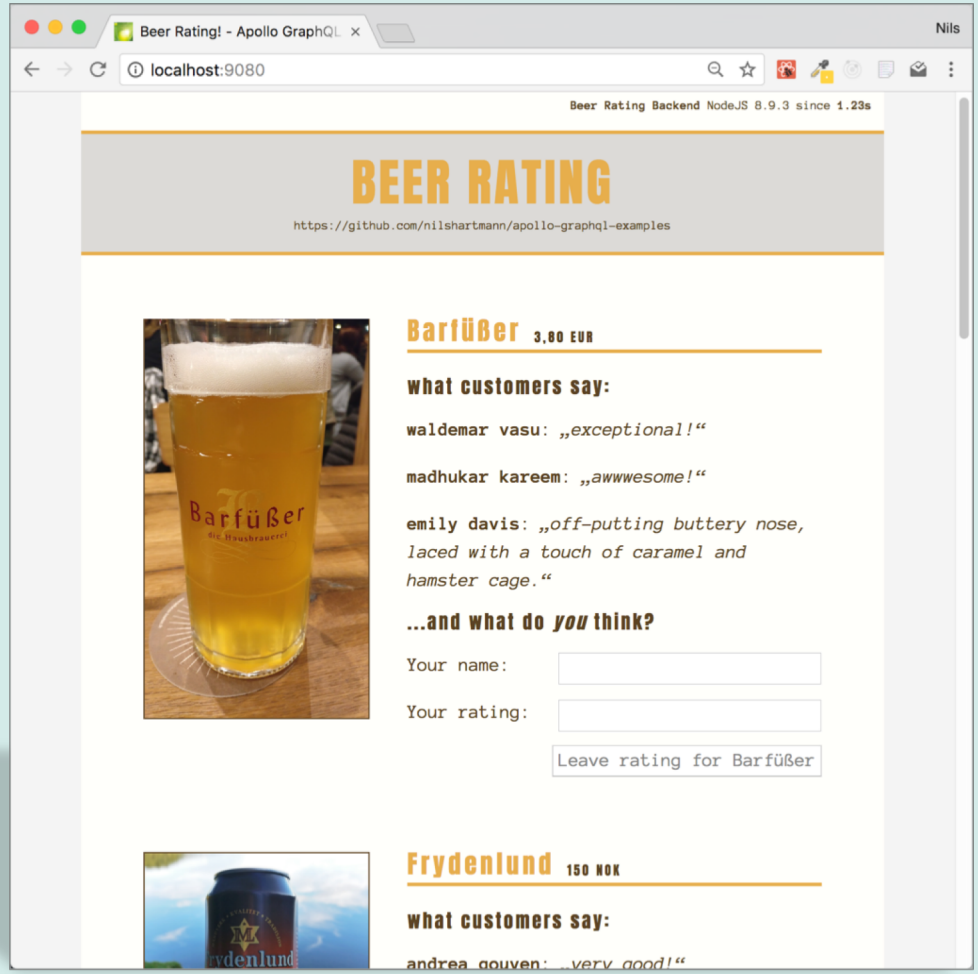
Slides: http://bit.ly/jax2018-typescript-einfuehrung

*"GraphQL is a **query language for APIs** and a **runtime for fulfilling those queries** with your existing data"*

https://graphql.org

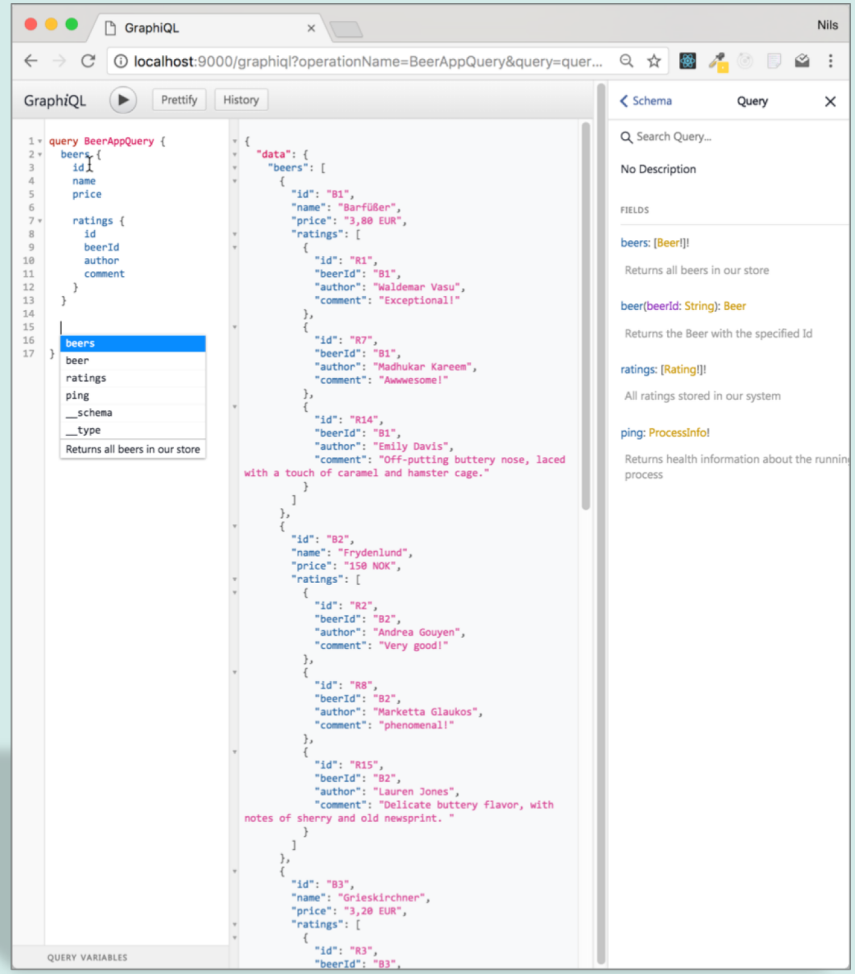
GraphQL

A QUERY LANGUAGE FOR YOUR API



Beispiel Anwendung

Source: https://bit.ly/fullstack-graphql-example



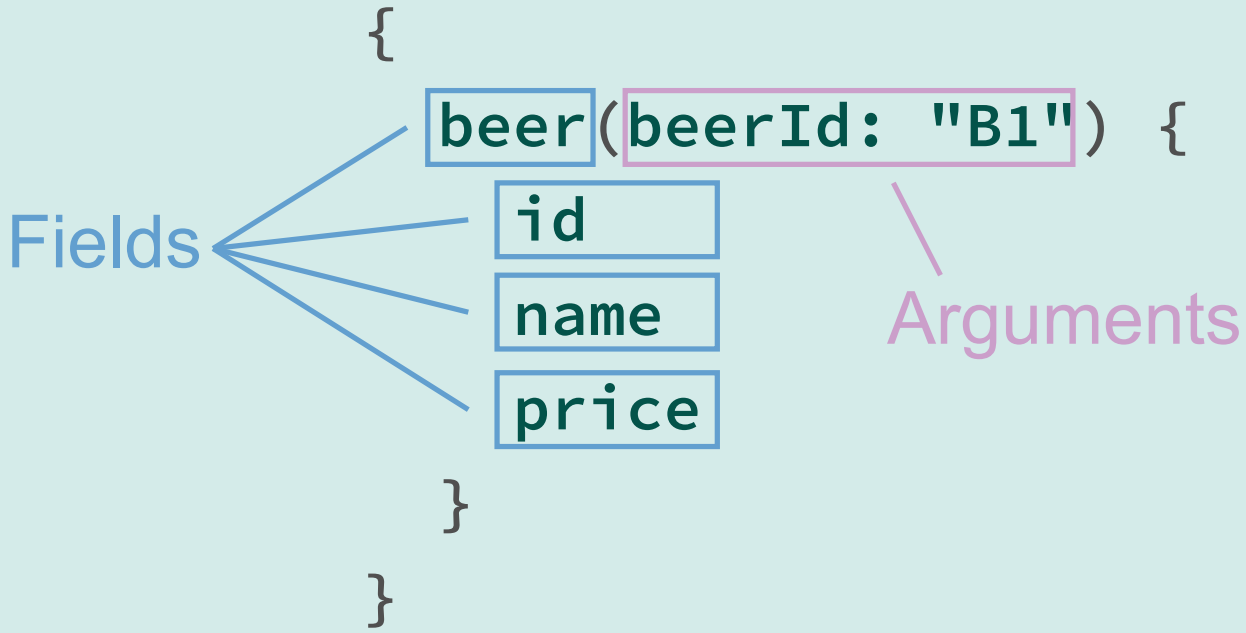
GraphiQL

http://localhost:9000

IDE Support

Beispiel: Intellij IDEA

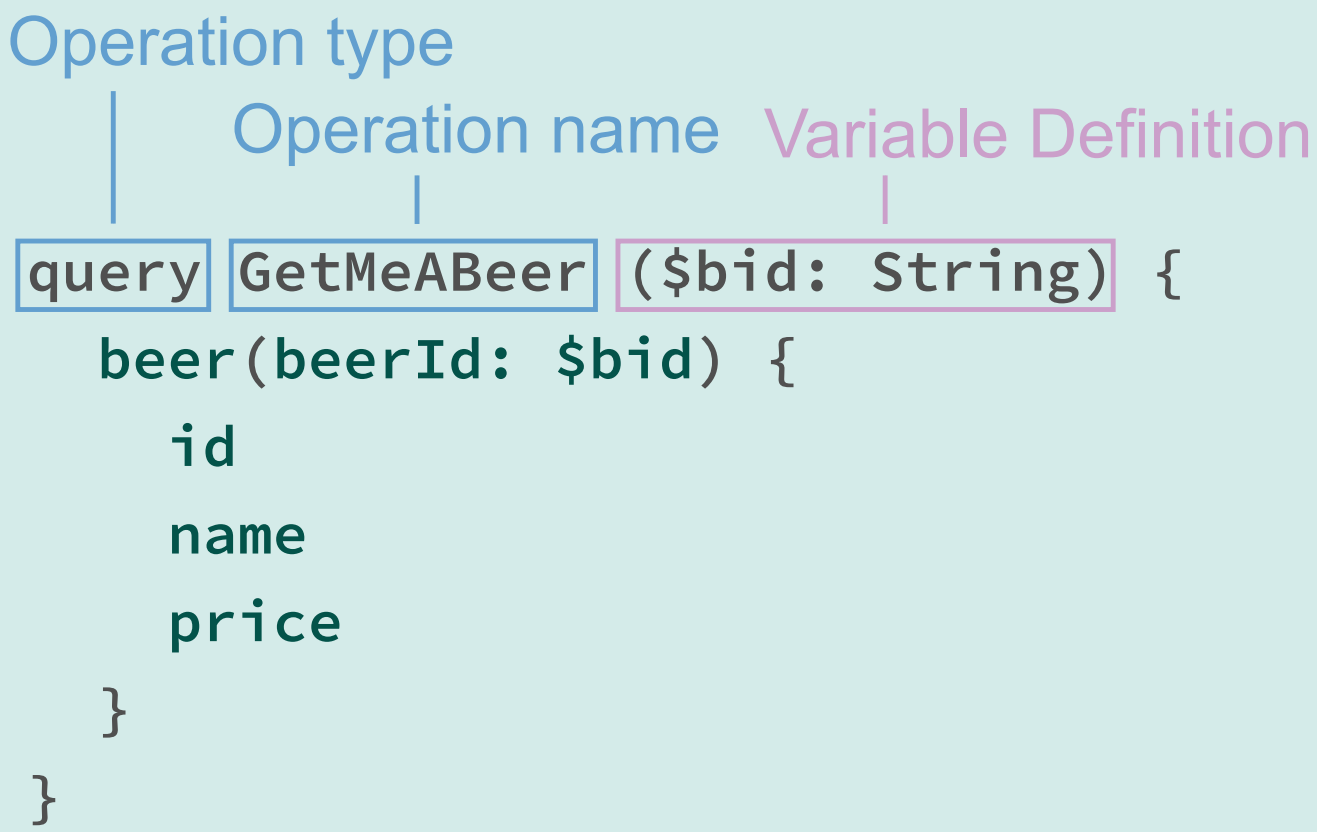
QUERY LANGUAGE



- Strukturierte Sprache, um Daten von der API abzufragen
- Abgefragt werden **Felder**
- Felder können **Argumente** haben

Siehe auch: https://dev-blog.apollodata.com/the-anatomy-of-a-graphql-query-6dffa9e9e747

QUERY LANGUAGE: OPERATIONS



- ...
- Operations: query, mutation, subscription

Siehe auch: https://dev-blog.apollodata.com/the-anatomy-of-a-graphql-query-6dffa9e9e747

SCHEMA BESCHREIBUNG

- TODO

Framework für GraphQL Server- und Client-Anwendungen (JavaScript)

Apollo GraphQL

HTTPS://WWW.APOLLOGRAPHQL.COM/

Teil 1:
GraphQL Server

APOLLO-SERVER

Apollo Server: https://www.apollographql.com/docs/apollo-server/

- Basiert auf JavaScript GraphQL Referenzimplementierung
- Adapter für diverse Webserver (Connect, Express, Hapi, ...)
- Gute Grundlage zum Starten
- Typings für TypeScript vorhanden

- Apollo Server v2 wird noch einfacher 😉
  - Zurzeit Beta-Version

GRAPHQL SERVER MIT APOLLO

Aufgaben

1. Typen und Schema definieren
2. Resolver für das Schema implementieren
   - Wie/woher kommen die Daten für eine Anfrage
3. Schema erzeugen und
4. Veröffentlichen über Webserver (z.B. Express)
   - (optional in Version 2)

SCHRITT 1: TYPEN DEFINITION

Schema Definition Language: https://graphql.org/learn/schema/

- Bestandteil der GraphQL Spec
- Legt fest, welche Typen es gibt und wie sie aussehen

```
type Beer {
  id: ID!
  name: String!
  price: String!
  ratings: [Rating!]!
 }

type Rating {
  id: ID!
  author: String!
  comment: String!
}
```

SCHRITT 1: TYPEN DEFINITION

Schema Definition Language: https://graphql.org/learn/schema/

- Auch **Query**, **Mutation** und **Subscription** sind "Typen"
- **Query** ist Pflicht, legt "Einstieg" zur API fest

```
type Query {
  beers: [Beer!]!
  beer(id: ID!): Beer
}

type Mutation {
  addRating(beerId: ID!, author: String!, comment: String): Rating!
}
```

Type-Definition in Apollo Server

- Erfolgt in der Regel über Schema-Definition-Language

Schema Definition

Root-Fields (erforderlich)

```
// Server.js

const typeDefs = `
  type Beer {
    id: ID!
    name: String!
    price: String!
    ratings: [Rating!]!
  }

  type Rating { . . . }

  type Query {
    beers: [Beer!]!
    beer(beerId: String!): Beer
  }
`;
```

SCHRITT 2: RESOLVER

Resolver-Funktion: Funktion, die Daten für ein Feld ermittelt

- Rückgabewert wird zum Client gesendet
- Laufzeitumgebung prüft auf korrektheit gemäß Schema-Definition
- Müssen mindestens für Root-Felder implementiert werden
  - ab da per "Property-Pfad" weiter (root.a.b.c)
  - oder per speziellem Resolver

Beispiel 1: Root-Resolver

Schema Definition

```
type Query {
  beers: [Beer!]!


}
```

Root-Resolver

```
const resolvers = {
  Query: {
    beers: => beerStore.all(),


  }
}
```

Beispiel 2: Root-Resolver mit Argumenten

- Argumente werden der Resolver-Funktion als Parameter übergeben
- Laufzeitumgebung sorgt dafür, dass nur gültige Werte übergeben werden

Schema Definition

```
type Query {
  beers: [Beer!]!
  beer(beerId: String): Beer
}
```

Root-Resolver

Root-Resolver mit Argumenten

```
const resolvers = {
  Query: {
    beers: => beerStore.all(),
    beer: (_, args) => beerStore.all()
                .find(beer => beer.id === args.beerId)
  }
}
```

Beispiel 3: Resolver für *ein* Feld eines Types

- Erlaubt individuelle Behandlung für einzelne Felder
- (zum Beispiel Performance-Optimierung / Security, ...)

Schema Definition

```
type Query { . . . }

type Beer {
  ratings: [Rating!]!
  . . .
}
```

Resolver für nicht-Root-Field

```
const resolvers = {
  Query: { . . . }
  Beer: {
    ratings: (beer) => ratingStore.all()
        .filter(rating => rating.beerId === beer.id)
  }
}
```

SCHRITT 3: SCHEMA VERÖFFENTLICHEN

Schema-Instanz erzeugen

- besteht aus Type-Definition und passenden Resolvern

```
import { makeExecutableSchema } from "graphql-tools";
```

Schema erzeugen

```
const schema = makeExecutableSchema({
    typeDefs,
    resolvers
  });
```

Schema-Instanz veröffentlichen

- Beispiel hier: über Instanz von Express-Server

```
import { makeExecutableSchema } from "graphql-tools";
import { graphqlExpress,                    }
                          from "apollo-server-express";
```

Schema erzeugen

```
const schema = makeExecutableSchema({
    typeDefs,
    resolvers
  });

const app = express();
```

Schema veröffentlichen

```
app.use("/graphql", graphqlExpress({ schema }));
```

SCHRITT 3: SCHEMA VERÖFFENTLICHEN

Optional: GraphiQL

```
import { makeExecutableSchema } from "graphql-tools";
import { graphqlExpress, graphiqlExpress }
                            from "apollo-server-express";
```

Schema erzeugen

```
const schema = makeExecutableSchema({
    typeDefs,
    resolvers
  });

const app = express();
```

Schema veröffentlichen

```
app.use("/graphql", graphqlExpress({ schema }));


app.get("/graphiql",
          graphiqlExpress({ endpointURL: "/graphql" }));
```

Teil 2:
GraphQL Client

MIT APOLLO UND REACT

APOLLO-SERVER

React Apollo: https://www.apollographql.com/docs/react/

- React-Komponenten zum Zugriff auf GraphQL APIs
  - funktioniert mit allen GraphQL Backends

- Sehr modular aufgebaut, viele npm-Module
- **apollo-boost** hilft bei Konfiguration für viele Standard Use-Cases
  - https://github.com/apollographql/apollo-client/tree/master/packages/apollo-boost

SCHRITT 1: ERZEUGEN DES CLIENTS UND PROVIDERS

Client ist zentrale Schnittstelle

- Netzwerkverbindung zum Server, Caching, ...

Bootstrap der React-Anwendung

```
import ApolloClient from "apollo-boost";
```

Client erzeugen

```
const client = new ApolloClient
  ({ uri: "http://localhost:9000/graphql" });
```

SCHRITT 1: ERZEUGEN DES CLIENTS UND PROVIDERS

Provider stellt Client in React Komponenten zur Verfügung

- Nutzt React Context API

Bootstrap der React-Anwendung

```
import ApolloClient from "apollo-boost";
import { ApolloProvider } from "react-apollo";
```

Client erzeugen

```
const client = new ApolloClient
  ({ uri: "http://localhost:9000/graphql" });
```

Apollo Provider um Anwendung legen

```
ReactDOM.render(
  <ApolloProvider client={client}>
    <BeerRatingApp />
  </ApolloProvider>,
  document.getElementById(...)
);
```

Queries

- Werden mittels gql-Funktion angegeben und geparst

Query parsen

```
import { gql } from "react-apollo";

const BEER_RATING_APP_QUERY = gql`
  query BeerRatingAppQuery {
    beers {
      id
      name
      price

      ratings { . . . }
    }
  }
`;
```

Queries

- Werden mittels gql-Funktion angegeben und geparst

Query parsen

```
import { gql } from "react-apollo";

const BEER_RATING_APP_QUERY = gql`
  query BeerRatingAppQuery {
    beers {
      id
      name
      price

      ratings { . . . }
    }
  }
`;
```

Query-Komponente

- Führt Query aus, kümmert sich um Caching, Fehlerbehandlung etc

React Komponente

```
import { gql, Query } from "react-apollo";

const BEER_RATING_APP_QUERY = gql`...`;

const BeerRatingApp(props) => (
  <Query query={BEER_RATING_APP_QUERY}>




  </Query>
);
```

Query-Komponente

- Führt Query aus, kümmert sich um Caching, Fehlerbehandlung etc
- Ergebnis (Daten, ...) wird per Render-Prop (Children) übergeben

```
import { gql, Query } from "react-apollo";

const BEER_RATING_APP_QUERY = gql`...`;

const BeerRatingApp(props) => (
  <Query query={query}>
    {({ loading, error, data }) => {



    }}
  </Query>
);
```

Query Ergebnis (wird ggf mehrfach aufgerufen)

Query-Komponente

- Führt Query aus, kümmert sich um Caching, Fehlerbehandlung etc
- Ergebnis (Daten, ...) wird per Render-Prop (Children) übergeben

Ergebnis (samt Fehler) auswerten

```
import { gql, Query } from "react-apollo";

const BEER_RATING_APP_QUERY = gql`...`;

const BeerRatingApp(props) => (
  <Query query={query}>
    {({ loading, error, data }) => {
      if (loading) { return <h1>Loading...</h1> }
      if (error) { return <h1>Error!</h1> }

      return <BeerList beers={data.beers} />
  }}
  </Query>
);
```

Query-Komponente in TypeScript: Typ-sicherer Zugriff auf Ergebnis

- Wird typisiert mit Query-Resultat
- Typen können mit apollo code-gen generiert werden

```
import { gql, Query } from "react-apollo";
import { BeerRatingAppQueryResult } from "./__generated__/...";

class BeerRatingQuery extends Query<BeerRatingAppQueryResult> {}

const BeerRatingApp(props) => (
  <BeerRatingQuery query={query}>
    {({ loading, error, data }) => {
      // . . .

      return <BeerList beers={data.biere} />
  }}
  </BeerRatingQuery>
);
```

Compile-Fehler!

Mutation-Komponente: Führt Mutations aus

- Mutation wird ebenfalls per gql geparst

```
import { gql } from "react-apollo";

const ADD_RATING_MUTATION = gql`
  mutation AddRatingMutation($input: AddRatingInput!)
  {
    addRating(ratingInput: $input) {
      id
      beerId
      author
      comment
    }
  }
`;
```

Mutation-Komponente: Führt Mutations aus

- Funktionsweise ähnlich wie Query-Komponente

```
import { gql, Mutation } from "react-apollo";

const ADD_RATING_MUTATION = gql`...`;

const RatingFormController(props) => (
  <Mutation mutation={ADD_RATING_MUTATION }>




  </Mutation>
);
```

Mutation-Komponente: Führt Mutations aus

- Funktionsweise ähnlich wie Query-Komponente
- Render-Property erhält Funktion zum Ausführen der Mutation

```
import { gql, Mutation } from "react-apollo";

const ADD_RATING_MUTATION = gql`...`;

const RatingFormController(props) => (
  <Mutation mutation={ADD_RATING_MUTATION }>
    {addRating => {




      }
    }
  </Mutation>
);
```

SCHRITT 3: MUTATIONS

Mutation-Komponente: Führt Mutations aus

- Funktionsweise ähnlich wie Query-Komponente
- Render-Property erhält Funktion zum Ausführen der Mutation

```
import { gql, Mutation } from "react-apollo";

const ADD_RATING_MUTATION = gql`...`;

const RatingFormController(props) => (
  <Mutation mutation={ADD_RATING_MUTATION }>
    {addRating => {
      return <RatingForm onNewRating={
        newRating => addRating({
          variables: {ratingInput: newRating)}
        })
       } />
      }
    }
  </Mutation>
);
```

Mutation-Komponente: Cache aktualisieren

- Callback-Funktionen zum aktualisieren des lokalen Caches
  - Aktualisiert automatisch sämtliche Ansichten

```
const RatingFormController(props) => (
  <Mutation mutation={ADD_RATING_MUTATION }
   update={(cache, {data}) => {
     // "Altes" Beer aus Cache lesen
     const oldBeer = cache.readFragment(...);

     // Neues Rating dem Beer hinzufügen
     const newBeer = ...;

     // Beer im Cache aktualisieren
     cache.writeFragment({data: newBeer});
   }>
   . . .
  </Mutation>
);
```

Vielen Dank!

Slides: http://bit.ly/jax2018-typescript-einfuehrung

Beispiel-Code: https://bit.ly/fullstack-graphql-example

Fragen?