

NILS HARTMANN

Apollo Client 2.0

GraphQL

als State Management-Werkzeug für React?

Slides: <http://bit.ly/enterjs-apollo-graphql>

NILS HARTMANN

Programmierer aus Hamburg

Bock auf React Entwicklung in HH?
<http://bit.ly/eos-react-developer>

Trainings, Workshops zu TypeScript, React,:
nils@nilshartmann.net

*"A community building flexible open source **tools** for
GraphQL."*

- <https://github.com/apollographql>

GraphQL

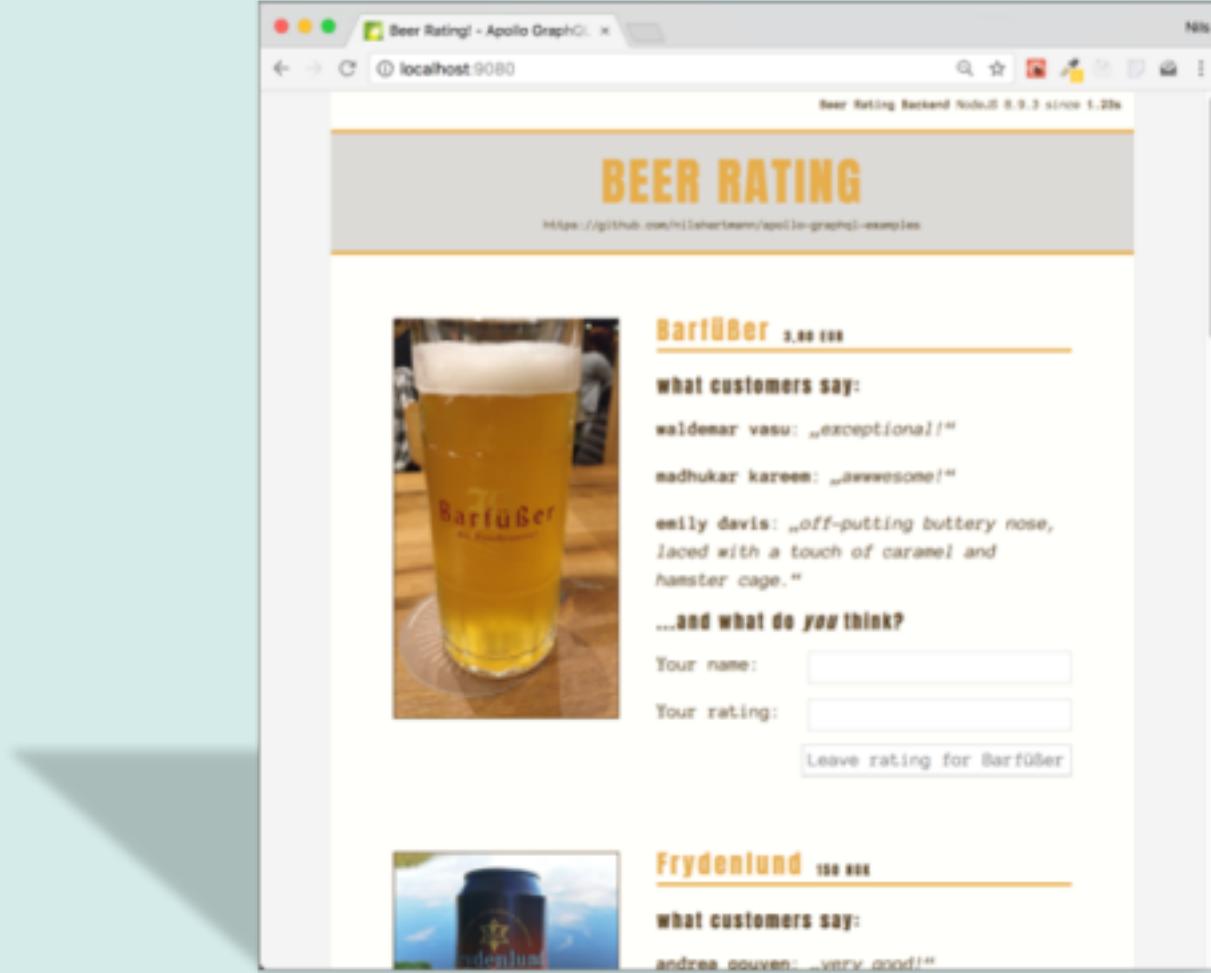
*"A community building flexible open source **tools** for GraphQL."*

- <https://github.com/apollographql>

"React Apollo allows you to fetch data from your GraphQL server and use it in building ... UIs using the React framework."

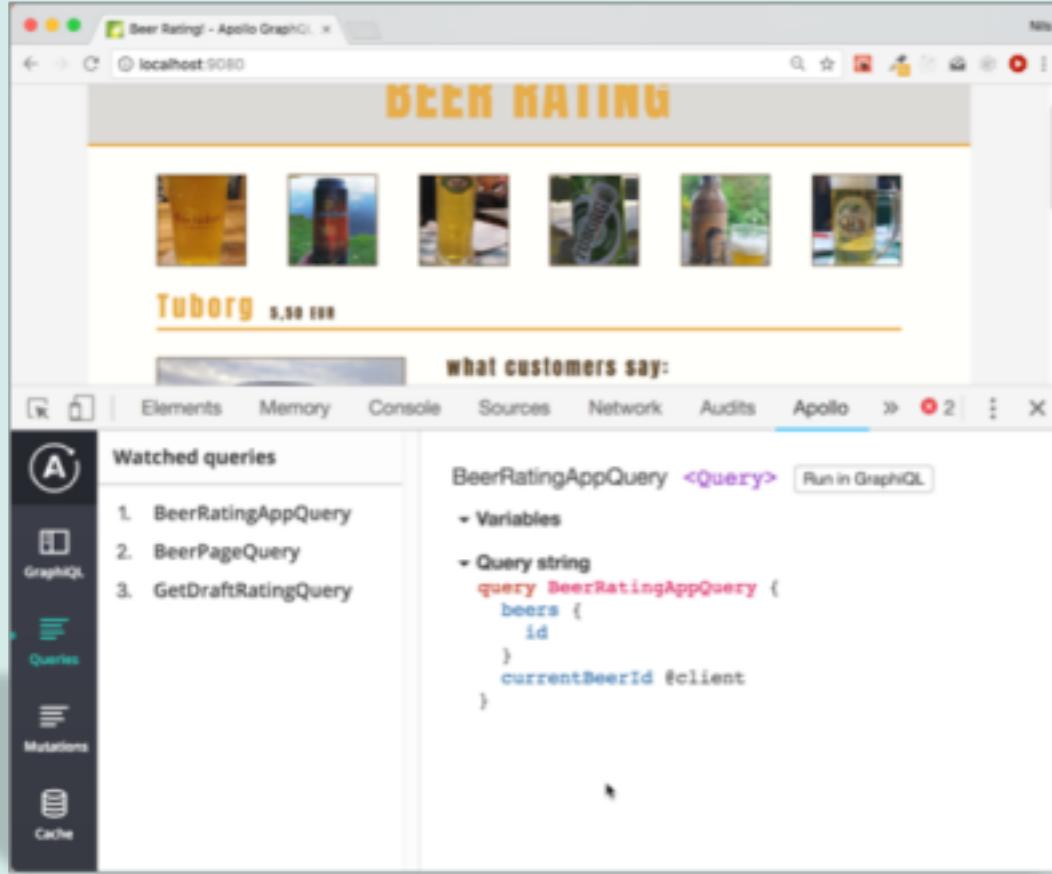
- <https://github.com/apollographql/react-apollo>

Apollo



Beispiel

Source (TypeScript): <http://bit.ly/enterjs-apollo-graphql-example>



Demo: Apollo Dev Tools



React Apollo

MIT APOLLO UND REACT

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

- React-Komponenten zum Zugriff auf GraphQL APIs
 - funktioniert mit allen GraphQL Backends
- **apollo-boost** hilft bei Konfiguration für viele Standard Use-Cases
 - bringt alle notwendigen Dependencies mit
 - <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, Authentifizierung...

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

IN REACT APOLLO

QUERIES

Queries – Daten vom Server abfragen

- Werden mittels gql-Funktion angegeben und geparsst

Query parsen

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

const BEER_PAGE_QUERY = gql`  
  query BeerPageQuery($beerId: ID!) {  
    beer(beerId: $beerId) {  
      id  
      name  
      price  
  
      ratings { . . . }  
    }  
  }  
`;
```

Query ausführen innerhalb einer React-Komponente

```
import { gql, Query } from "react-apollo";  
  
const BEER_PAGE_QUERY = gql`...`;  
  
const BeerPage({beerId}) => (  
);
```

Die React Komponente

Query-Komponente

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

```
import { gql, Query } from "react-apollo";  
  
const BEER_PAGE_QUERY = gql`...`;  
  
const BeerPage({beerId}) => (  
  <Query query={BEER_PAGE_QUERY} variables={{beerId}}>  
    </Query>  
);
```

Query Komponente

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_PAGE_QUERY = gql`...`;

const BeerPage({beerId}) => (
  <Query query={BEER_PAGE_QUERY} variables={{beerId}}>
    {({ loading, error, data }) => {
      if (loading) return <div>Loading</div>;
      if (error) return <div>Error</div>;
      return <div>{data.beer}</div>;
    }}
  </Query>
);
```

Callback-Funktion als
Children

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_PAGE_QUERY = gql`...`;

const BeerPage({beerId}) => (
  <Query query={BEER_PAGE_QUERY} variables={{beerId}}>
    {({ loading, error, data }) => {
      if (loading) { return <h1>Loading...</h1> }
      if (error) { return <h1>Error!</h1> }

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

Ergebnis (samt Fehler)
auswerten

Query-Komponente Weitere Features

- Paginierung / fetch more: Daten nachladen
- Aktualisierung basierend auf Subscriptions
 - Client erhält automatisch neue Daten
- Cache
- Typ-Sicherheit mit TypeScript und Flow

TYP-SICHERE VERWENDUNG

Beispiel TypeScript: Typ-Sichere Verwendung der Komponente

```
import { BeerPageQueryResult, BeerPageQueryVars } from "...";  
  
const BeerRatingPage(...) => (  
  <Query<BeerPageQueryResult, BeerPageQueryVars>  
    query={BEER_PAGE_QUERY} variables={{bier: beerId}} >  
    {({ loading, error, data }) => {  
      // . . .  
  
      return <BeerList beers={data.biere} />  
    }}  
  </Query>  
) ;
```

Compile-Fehler!

Compile-Fehler!

BEISPIEL: TYP-SICHERE QUERIES

apollo-codegen: Generiert Typen für Flow und TypeScript

- Schema wird vom Server geladen
- Fehler in **Queries** werden schon beim Generieren erkannt

Falscher Typ

```
$ apollo-codegen generate 'src/**/*.tsx' ...
BeerRatingApp.tsx:
  Variable "$newBeerId" of type "String"
  used in position expecting type "ID".
```

Unbekanntes Feld

```
$ apollo-codegen generate 'src/**/*.tsx' ...
BeerPage.tsx:
  Cannot query field "alc" on type "Beer".
```

...and what do you think?

Your name:

Klaus

Your rating:

perlt!

Leave rating for Barfüßer



Beispiel: Speichern der Ratings auf dem Server

Mutations

DATEN VERÄNDERN

MUTATIONS

Mutations

- Mutation wird ebenfalls per gql geparsst

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

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

MUTATIONS

Mutation innerhalb einer React Komponente

React Komponente

```
import { gql, Mutation } from "react-apollo";
const ADD_RATING_MUTATION = gql`...`;
const RatingFormController(props) => (
);

```

MUTATIONS

Mutation-Komponente: Führt Mutations aus

- Funktionsweise ähnlich wie Query-Komponente

Mutation Komponente

```
import { gql, Mutation } from "react-apollo";
const ADD_RATING = gql`...`;
const RatingFormController(props) => (
  <Mutation mutation={ADD_RATING} variables={...}>
    </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 = gql`...`;  
  
const RatingFormController(props) => (  
  <Mutation mutation={ADD_RATING} variables={...}>  
    {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 onSubmit={({  
        newRating }) => addRating({  
          variables: {newRating}  
        })}  
      } />  
    }  
  }  
  </Mutation>  
) ;
```

SCHRITT 3: MUTATIONS

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}) => {
```



Enthält die Daten, die der Server als Antwort auf die Mutation geschickt hat

```
    }>
    . . .
  </Mutation>
);
```

SCHRITT 3: MUTATIONS

Mutation-Komponente: Cache aktualisieren

- Callback-Funktionen zum aktualisieren des lokalen Caches
 - Query-Komponenten werden automatisch aktualisiert (manchmal)

```
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>
);
```

Local State

GRAPHQL FÜR DEN ANWENDUNGSZUSTAND

BEISPIEL 1: SELECTED BEER

Screenshot of a web browser showing a beer rating application. The page title is "Deer Rating! - Apollo GraphQL". The URL is "localhost:9080". The page header says "BEER RATING".

The interface is divided into two main sections:

- BeerRack:** A horizontal row of beer cans and bottles. One can in the center is highlighted with a blue arrow pointing to it, labeled "currentBeerId".
- Frydenlund:** A detailed view of a Frydenlund Pale Ale can. The can features a logo with a star and the text "Frydenlund PALE ALE".

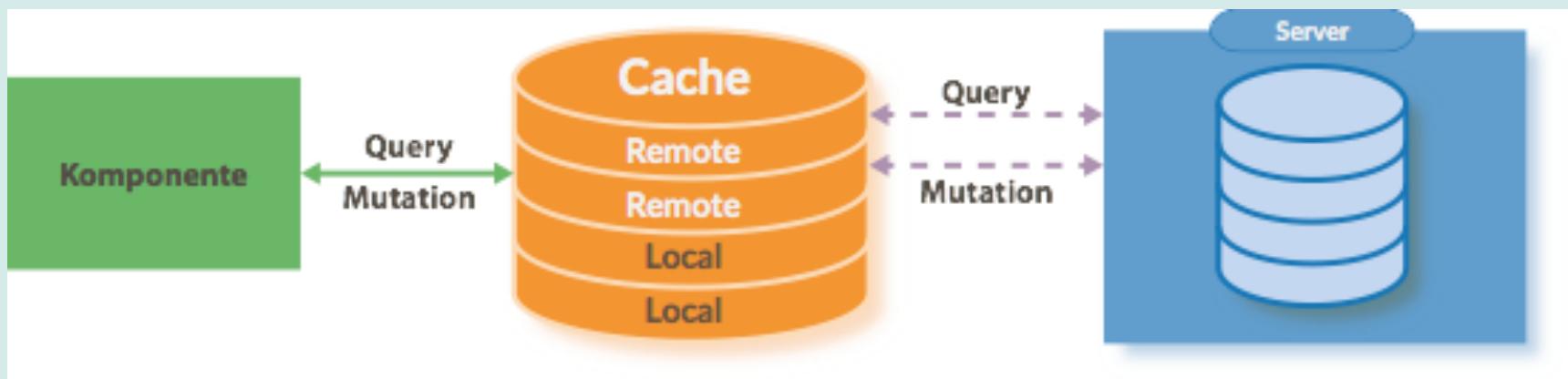
The "BeerPage" section contains the following content:

- what customers say:**
- andrea gouyen: „very good!“
- marketta glaukos: „phenomenal!“
- lauren jones: „delicate buttery flavor, with notes of sherry and old newsprint. “
- and what do you think?**

WAS BEDEUTET "LOCAL" STATE

Local State – Globale Daten in der Anwendung

- Entspräche Redux Store
- "Local", weil nicht remote über API geladen
 - Unglücklicher Begriff. Eigentlich: "Global" State? "App" State?



Apollo Cache – Zentrale Ablage aller gelesenen Daten

- in normalisierter Form
- kann per API gelesen und verändert werden

The screenshot shows the Apollo GraphQL DevTools interface with the "Cache" tab selected. On the left, there's a sidebar with icons for Cache (selected), GraphQL, Queries, Mutations, and Cache. The main area displays a hierarchical tree of data from a query named "ROOT_QUERY". The tree structure is as follows:

- beer({ "beerId": "B1" }): Beer
- Beer:B1
- beers: [Beer]
 - 0: Beer:B1
 - 1: Beer:B2
 - 2: Beer:B3
 - 3: Beer:B4
 - 4: Beer:B5
 - 5: Beer:B6
- currentBeerId: "B1"

LOCAL DATA

Lokale Daten aus dem Cache lesen – mit GraphQL Queries

- Abfragen mit `@client` directive
- liest Daten aus dem lokalen Cache, nicht vom Server

Daten vom Server

```
import { gql } from "react-apollo";  
  
const BEER_RATING_APP_QUERY = gql`  
query BeerRatingAppQuery {
```

Daten aus lokalem Cache

```
    beers {  
        id  
    }  
  
    currentBeerId @client  
}  
`;
```

Verwendung wie gewohnt

```
<Query query={BEER_RATING_APP_QUERY}>...</Query>
```

Lokale Daten verändern – mit GraphQL Mutations

- ebenfalls @client Directive

```
const SET_CURRENT_BEER_ID_MUTATION = gql`  
  mutation SetCurrentBeerIdMutation($newBeerId: ID!) {  
    setCurrentBeerId(beerId: $newBeerId) @client  
  }  
`;
```

Lokale Daten verändern – mit GraphQL Mutations

- ebenfalls @client Directive

```
const SET_CURRENT_BEER_ID_MUTATION = gql`  
  mutation SetCurrentBeerIdMutation($newBeerId: ID!) {  
    setCurrentBeerId(beerId: $newBeerId) @client  
  }  
`;
```

Mutation wie gewohnt

```
<Mutation mutation={SET_CURRENT_BEER_ID_MUTATION}>  
  {setCurrentBeerId => (  
    <div onClick={() => setCurrentBeerId({  
      variables: { newBeerId: beer.id }})>...</div>  
  )}  
</Mutation>
```

Schema definieren

- in GraphQL Schema Definition Language (SDL)
- optional, zurzeit nur für apollo-codegen und GraphiQL

```
const typeDefs = `

  type Query {
    currentBeerId: ID!
  }

  type Mutation {
    setCurrentBeerId(beerId: ID!): ID!
  }
`
```

Default-Werte Vorbelegung für den Cache

- optional, je nach Fachlichkeit

```
const typeDefs = `

  type Query {
    currentBeerId: ID!
  }

  type Mutation {
    setCurrentBeerId(beerId: ID!): ID!
  }
`


const defaults = {
  currentBeerId: "B1"
}
```

Resolver Funktionen Zum Lesen / Schreiben in den Cache

- Für Queries (optional) und Mutations
- Funktion bekommt Argumente aus Query und Cache übergeben

```
const resolvers = {  
  Query: {  
    // in unserem Fall nicht notwendig  
  },  
  
  Mutation: {  
    setCurrentBeerId: (_, { beerId }, { cache }) => {  
      cache.writeData({ data: { currentBeerId: beerId } });  
      return beerId;  
    }  
  }  
}
```

Lokalen State beim ApolloClient bekannt geben

```
const typeDefs = ...;
const defaults = ...;
const resolver = ...;

const client = new ApolloClient({
  uri: "http://localhost:9000/graphql",
  clientState: {
    typeDefs,
    defaults,
    resolver
  }
});
```

AKTUALISIERUNG NACH MUTATIONS

Komponenten-Updates funktionieren nicht immer automatisch

- zum Beispiel, wenn Objekte in eine Liste eingefügt werden
- refetchQueries kann remote-Zugriffe auslösen!
- Das ist in Redux nicht notwendig

```
// client.mutate ist Alternative zu Mutation-Komponente

client.mutate({
  mutation: UPDATE_DRAFT_RATING,
  variables: { beerId, author, comment },
  refetchQueries: [
    { query: GET_DRAFT_RATING_QUERY, variables: { beerId } },
    { query: BEERS_QUERY }
  ]
});
```

Löst Remote Zugriff aus!

BEISPIEL 2: DRAFT RATING

The screenshot shows a user interface for rating a draft beer. At the top, there's a horizontal row of six small images representing different beers. The first two images have a red box around them, and the second one has a blue arrow pointing to it from the left. Above this row is the text "BeerRack". Below the row, the name "Barfüßer" is displayed in large orange letters, followed by "3,80 EUR". To the right of the price, there's a "What customers say:" section containing four customer reviews. A blue box highlights the word "hasDraftRating" in the third review. At the bottom, there's a form for users to leave their own rating, with a red box around the "RatingForm" button.

BeerRack

Barfüßer 3,80 EUR

What customers say:

waldemar nessu: „exceptional!“

madhukar kareem: „awwwesome!“

cecilia viktoria: „...hasDraftRating... please?“

emily davis: „off-putting buttery nose, laced with a touch of caramel and hamster cage.“

...and what do you think?

Your name: Klaus

Your rating:

Leave rating for Barfüßer

RatingForm

REMOTE- UND LOCAL DATA KOMBINIEREN

Client kann Felder vom Remote-Schema ergänzen



```
const BEER_RATING_APP_QUERY = gql`  
query BeerRatingAppQuery {  
  beers {  
    id  
  
    hasDraftRating @client  
  }  
  
  currentBeerId @client  
}
```

REMOTE- UND LOCAL DATA KOMBINIEREN

Resolver-Funktion erhält umschließendes Objekt

```
const resolvers = {  
  Beer: {  
    hasDraftRating: (beer, _, { cache }) => {  
      const res = cache.readFragment({  
        fragment: gql`fragment draftRating on DraftRating { id }`,  
        cacheKey: `DraftRating:${beer.id}`  
      });  
  
      return res !== null;  
    }  
  },  
  Query: { ... }  
}
```

REMOTE- UND LOCAL DATA KOMBINIEREN

Resolver-Funktion erhält umschließendes Objekt

- `readFragment` liefert Objekte aus dem Cache

```
const resolvers = {
  Beer: {
    hasDraftRating: (beer, _, { cache }) => {
      const res = cache.readFragment({
        fragment: gql`fragment draftRating on DraftRating { id }`,
        cacheKey: `DraftRating:${beer.id}`
      });

      return res !== null;
    }
  },
  Query: { ... }
}
```



FAZIT UND BEWERTUNG

GraphQL mit dem Apollo Client

- Query- und Mutation-Komponenten funktionieren sehr gut
 - TypeScript-Support weitgehend sehr gut
 - Leider nicht **durchgehend** typisiert
- Durch starke Modularisierung schwer überschaubar
 - wo werden Fehler gemeldet? Wo Doku?
 - Doku teilweise inkonsistent
- Diverse Möglichkeiten, ein Ziel zu erreichen
 - (HOCs vs Komponenten, Mutations vs direkter Cache Zugriff, mehrere APIs für direkten Cache Zugriff)
- Apollo Dev Client hat Probleme
 - Ansichten teilweise nicht aktuell
 - Führt andere Queries aus als der Client (Client fügt __typename für Caching hinzu)

Lokaler State mit Apollo: GraphQL für APIs? Oder für Client?

- Achtung! Version 0.4, dh noch "experimental"
 - Zahlreiche Issues im Bug Tracker
 - Tools passen teilweise nicht zusammen (Client/Source, Dev Tools, codegen)
- Queries müssen nach Mutations manuell neu ausgeführt werden
 - in Redux nicht nötig
- Dokumentation / Beispiele nur sehr eingeschränkt
 - Verbreitungsgrad? Kopplung an **Apollo GraphQL**
- Arbeiten mit der Cache API "gewöhnungsbedürftig"
 - Reducer in Redux wesentlich einfacher
- Im Vergleich mit Redux weniger Architektur-Pattern (meine Meinung)
 - Redux (logischerweise) viel höhere Verbreitung, viel ausgereifter
 - Mutations => Actions ?
 - Selectors => GraphQL Abfrage? Resolver ?

"EMPFEHLUNG"

Meine persönliche Einschätzung:

- Query und Mutation Komponenten gut einsetzbar
- Lokaler State mit Apollo: noch fragil, weitere Entwicklung abwarten
 - (dort wo es funktioniert ganz cool)
- **Weiterhin gilt: "normaler" React State ist "erlaubt"!**
 - Alles andere nur machen, wenn der React State nicht ausreicht



Vielen Dank!

Slides: <http://bit.ly/enterjs-apollo-graphql>

Beispiel-Code: <http://bit.ly/enterjs-apollo-graphql-example>