NILS HARTMANN
https://nilshartmann.net

Apollo GraphQL Client

git clone https://github.com/nilshartmann/graphql-apollo-workshop

Rückblick

- Fragen?

APOLLO CLIENT

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

- React API für GraphQL

- Ausführen von Queries, Mutation, Subscriptions
- Fehlerhandling
- Globaler Cache
- Persistent Queries

APOLLO CLIENT

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

- React API für GraphQL

- Ausführen von Queries, Mutation, Subscriptions
- Fehlerhandling
- Globaler Cache
- Persistent Queries

- Zugrunde liegender Client auch für andere Frameworks
  - https://apollo.vuejs.org/
  - https://apollo-angular.com/docs/

Alternativen für React

GraphQL Client von Facebook:

- https://relay.dev/

Leichtgewichtiger GraphQL Client:

- https://formidable.com/open-source/urql/

Data Fetching Libs:

- https://react-query.tanstack.com/graphql
- https://swr.vercel.app/docs/data-fetching#graphql

Queries und Mutations

DER APOLLO CLIENT

Zentrales Client-Objekt

- Konfiguration
  - URL, Authentifizierung, Cache...
- Methoden zum Ausführen von Queries, Mutations
- Arbeiten mit dem Cache

- Ist React unabhängig
- Wird in React über Hooks (nur) indirekt verwendet

Zentrales Client-Objekt

```
import { ApolloClient, InMemoryCache } from "@apollo/client";


// Beim Starten der Anwendung
const client = new ApolloClient({
  uri: "http://localhost:4000",
  cache: new InMemoryCache()
})
```

Ausführen von Queries

Nur als Beispiel, in React über Hooks API

```
import { ApolloClient, InMemoryCache } from "@apollo/client";


// Beim Starten der Anwendung
const client = new ApolloClient({
  uri: "http://localhost:4000",
  cache: new InMemoryCache()
})

client.query({
  query: gql`
   query Project($id: ID!) {
     project(id: $id) { id title }
   }`,

  variables: {
    id: "P1"
  }
}).then(result => ...)
```

Bereitstellen über Provider-Komponente

Die Client-Instanz wird über ApolloProvider in die React—App gegeben

```
import { ApolloClient, InMemoryCache, ApolloProvider } from "@apollo/client";


// Beim Starten der Anwendung
const client = new ApolloClient({
  uri: "http://localhost:4000",
  cache: new InMemoryCache()
})

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

Queries ausführen

👉 Beispiel: Eine Darstellung aller User (Liste und Detail)

👉 frontend_workspace

Queries ausführen

Schritt 1: Query definieren

```
import { gql } from "@apollo/client";

const ProjectsQuery = gql`
  query ProjectsQuery {
    projects { id title tasks { id title } }
  }
}
```

Queries ausführen: useQuery

Schritt 2: Query ausführen und verarbeiten

Wenn Request Status sich ändert,
wird Komponente neu gerendert,
=> neue Daten kommen zurück!

```
import useQuery from "@apollo/client";

function ProjectListPage() {
  const { data, loading, error } = useQuery(ProjectsQuery);


  if (loading)
    return <h1>Greetings loading...</h1>

  if (error)
    return <Error msg="Loading failed" />


  return <GreetingTable greetings={data} />
}
```

useQuery-Lifecycle

1. Request wird gestartet, loading: true wird zurückgegeben
2. Komponente wird (neu) gerendert, LoadingIndicator sichtbar
3. Ergebnis kommt: error oder data ist jetzt gesetzt
4. Komponente wird (neu) gerendert, kann Informationen anzeigen
5. Geladene Daten werden in den Cache geschrieben
6. Alle Komponenten, die diese Daten verwenden, werden neu gerendert

👉 Ganze Anwendung ist konsistent!

Variablen

```
import { gql } from "@apollo/client";

const ProjectsQuery = gql`
  query ProjectQuery($id: ID!) {
    project(id: $id) { id title tasks { id title } }
  }
}
```

Variablen

Konfiguration des Query als zweiten Parameter

Darin u.a. Variablen

```
function ProjectPage() {
  const { data, loading, error } = useQuery(ProjectQuery, {
    variables: {
      id: "P1"
    }
   });

  // Rest wie gesehen
}
```

Optionen

useQuery kennt noch weitere Optionen und Rückgabe-Werte

sehen wir uns später an

ÜBUNG - VORBEREITUNG

1. git pull oder git clone

2. Starten Backend:
   - app/userservice: **npm install** und **npm start**
   - app/10_ 10_naive_implementierung: **npm install** und **npm start**

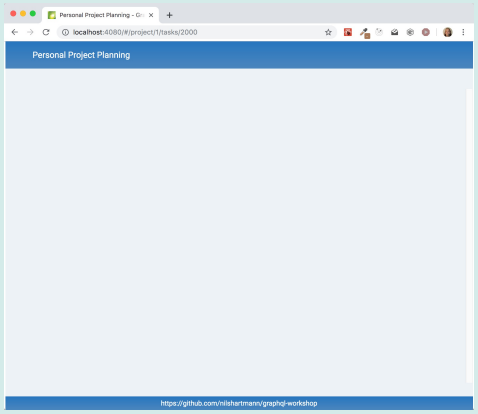
3. Installation Frontend:
   - code-frontend/workspace: **npm install** und **npm start**
   - Frontend läuft unter **http://localhost:3000**

ÜBUNG 1: EINE GRAPHQL QUERY

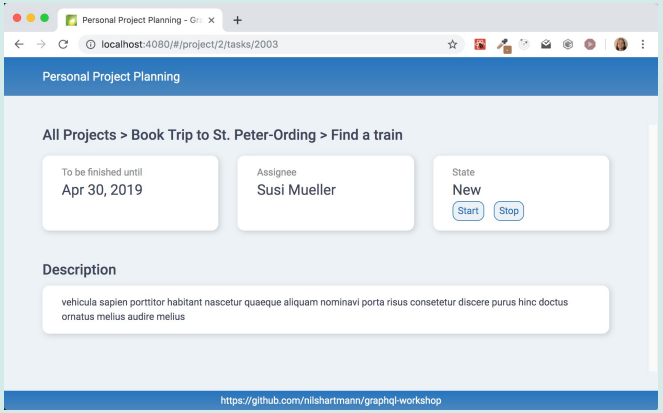
Die Task-Ansicht bauen

In TaskDetailsPage.js musst Du einen Query definieren, ausführen und das Ergebnis verarbeiten

- In der Datei TaskDetailsPage.js sind TODOs eingetragen
  - Achtung! Nur "ÜBUNG 1", TODO 1 und TODO 2 machen (ÜBUNG 2 ignorieren)
- Wenn Du die Datei speicherst:
  - wird sie automatisch im Browser aktualisiert
  - Darstellung wird aktualisiert, sonst Reload drücken



Vorher 😰



Nachher 😍

MUTATIONS

Mutation ausführen

👉 Beispiel: Hinzufügen eines Users

Mutation ausführen

Schritt 1: Mutation definieren, analog zu Query

```
import { gql } from "@apollo/client";

const AddTaskMutation = gql`
  mutation AddTask($id: ID!, $input: AddTaskInput!) {
    addTask(id: $id, input: $input) {
      id state
  }
}
```

Mutation ausführen: useMutation

useMutation liefert Array mit Funktion und Lifecycle-Objekt zurück

Die Funktion wird verwendet, wenn die Mutation ausgeführt werden soll

Hier können Variablen übergeben werden (wie useQuery)

```
function AddTaskPage() {
  const [ addTask ] = useMutation(AddTaskMutation);

  function onAddTask() {
    // z.B. bei Button klick
    addTask({variables: {
        id: taskId, input: { title: "...", description: "..." }
    } })
  }

  return <...><button onClick={onAddTask}>Save</button>...;
```

Mutation ausführen: Lifecycle

useMutation als 2. Parameter das Lifecycle-Objekt zurück

Ausführung der Mutation kann wie bei useQuery getracked werden

```
function AddTaskPage() {
  const [ addTask, {loading, error, data} ] = useMutation(...);

  function onAddTask() { ... }

  if (loading) {
     // Mutation wird ausgeführt
  }

  if (error) { ... }

  return ...;
}
```

Mutation ausführen: Lifecycle

useMutation als 2. Parameter das Lifecycle-Objekt zurück

Ausführung der Mutation kann wie bei useQuery getracked werden

```
function AddTaskPage() {
  const [ addTask, {loading, error, data} ] = useMutation(...);

  function onAddTask() { ... }

  if (loading) {
     // Mutation wird ausgeführt
  }

  if (error) { ... }

  return ...;
}
```

Mutation ausführen: Lifecycle

Funktion zum Ausführen liefert ein Promise zurück

Auch hierüber kann mit dem Mutation-Ergebnis gearbeitet werden

Zum Beispiel für Redirect nach erfolgreicher Mutation

```
function AddTaskPage() {
  const [ addTask ] = useMutation(...);

  async function onAddTask() {
    const result = await addTask({variables: ... });

    history.push("/projects");
  }

  return ...;
}
```

Mutation ausführen: Cache

Mit den gelesenen Daten der Mutation wird Cache aktualisiert

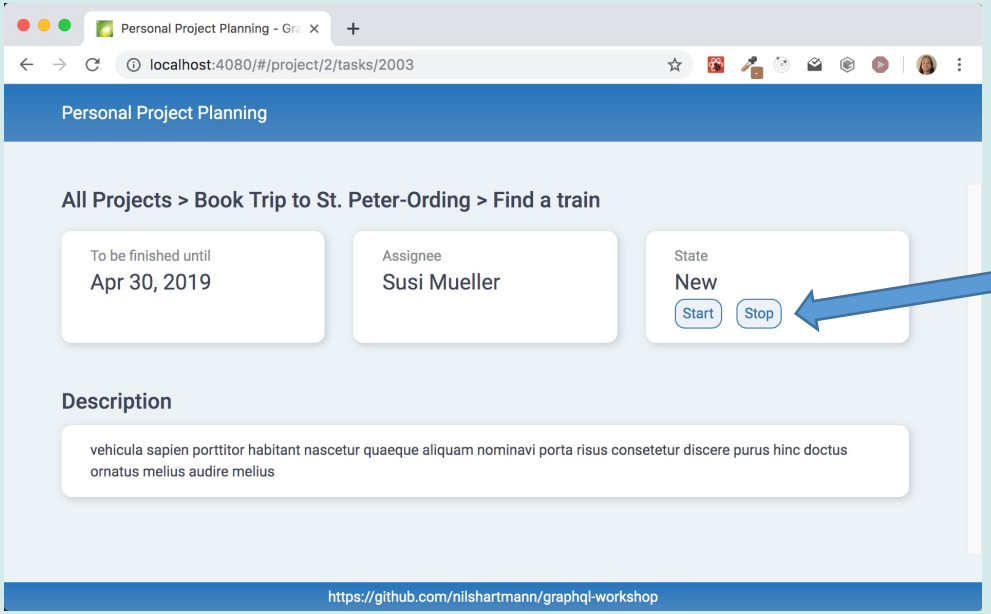
Analog zu useQuery, deswegen auch hier immer id mit abfragen!

ÜBUNG 2: MUTATIONS

Die Task-Ansicht vervollständigen

In TaskDetailsPage.js musst Du eine Mutation ausführen, damit der Task State aktualisiert werden kann

- In der Datei TaskDetailsPage.js sind TODOs eingetragen (ÜBUNG 2)



Button sollten funktionieren, das Label darüber wird aktualisiert

Der Apollo Cache

Der Apollo Cache

Konsistente Darstellung Eurer Daten in der gesamten App

Einsparen von Netzwerk-Requests

Apollo Client Dev Tools:

- Chrome: https://chrome.google.com/webstore/detail/apollo-client-devtools/jdkknkkbebbapilgoeccciglkfbmbnfm

- Firefox: https://addons.mozilla.org/de/firefox/addon/apollo-developer-tools/

Der Apollo Cache

TODO: Sidebar einschalten!

- 👉 Demo: Task-State
- 👉 Demo: Dev Tools

- 👉 Demo: Task-Liste mit AddTaskPage (fetchPolicy cache-and-network)
- 👉 Demo: Task-Liste mit QuickAddTask (Refresh)

Queries ausführen: Caching

Geladene Daten werden normalisiert und gecached

Immer id-Feld mit abfragen (auch wenn man es nicht selbst braucht)!

```
const ProjectsQuery = gql`
  query ProjectsQuery {
    id title tasks { id title }
  }
}
```

Apollo fügt jedem Objekt automatisch das __typename-Feld implizit hinzu

Cache-Key: `__typename:id-Feld`

```
Project:P1: { id: P1, title: ..., tasks: [ T1, T2, T3 ] },
Project:P2: { id: P2, title: ..., tasks: [ T2 ] },
Task:T1: { id: T1, description: ... },
Task:T2: { id: T2, description: ... }
```

Cache aktualisieren: Strategien

- Query neu ausführen (forcieren):
  - Fetch-Policy
  - Polling
  - Refetch
- Direkte Aktualisierung des Caches per API

DER APOLLO CACHE

Fetch-Policy: gibt an, wann ein Query erneut ausgeführt wird

https://www.apollographql.com/docs/react/data/queries/#supported-fetch-policies

Zum Beispiel:

- cache-first: Guckt in den Cache, lädt nur, wenn dort nicht verfügbar
- cache-and-network: Erst Cache, dann aber auch Netzwerk, um zu aktualisieren
- network-only: Immer erneut laden
- cache-only: Nur aus dem Cache nehmen, nie Netzwerk

```
useQuery(TaskListQuery, {
  variables: { ... },
  fetchPolicy: "cache-and-network"
})
```

DER APOLLO CACHE

Polling: Query wird automatisch alle x-ms ausgeführt

https://www.apollographql.com/docs/react/data/queries/#supported-fetch-policies

Variante 1: per Property

```
useQuery(TaskListQuery, {
  variables: { ... },
  pollInterval: 5000
})
```

Variante 2: Funktion

```
const {startPolling} = useQuery(TaskListQuery);

startPolling(5000);
```

DER APOLLO CACHE

Refetch: Ein Query manuell erneut ausführen

https://www.apollographql.com/docs/react/data/queries/#supported-fetch-policies

useQuery liefert eine refetch-Funktion zurück

Diese führt den Query erneut aus

Dabei können neue Variablen etc. mitgegeben werden

Nützlich z.B. für "Reload"-Button

```
const {refetch} = useQuery(TaskListQuery, {
  variables: { taskId: "T1" }
});

function onReload() {
  refetch({
    taskId: "T2"
  })
}
```

Nach einer Mutation

- Es gibt mehrere Strategien, den Cache nach dem Ausführen einer Mutation zu aktualisieren
- Wenn ein einzelnes Objekt gelesen wurde, wird dieses im Cache ausgetauscht
- Das gilt auch für Objekt-Graphen, *aber nicht für Listen*

Nach einer Mutation: Optimistic Responses

- Wenn das Ergebnis einer Mutation "erahnt" werden kann, kann man das erwartete Ergebnis übergeben
- Dieses wird dann sofort in den Cache gesetzt, so dass der User nicht warten muss
- Wenn das "echte" Ergeniss dann vom Server kommt, wird der Cache ggf. nochmal aktualisiert
- Kann zu "flüssigerem" Erlebnis für User führen

```
const [addTask] = useMutation(UpdateTaskState);

function onReload() {
  addTask({
    variables: { taskId: "T1", newState: "RUNNING" },
    optimisticResponse: {
      // Struktur muss zum erwarteten Ergebnis passen
      taskId: "T1",
      newState: "RUNNING",
      title: "My Task"
    }
  });
}
```

Nach einer Mutation: Queries erneut ausführen

- Beim Ausführen einer Mutation kann eine Liste von *Queries* (samt Variablen) angegeben werden, die dann erneut ausgeführt werden.
- Mit deren Ergebnissen wird der Cache aktualisiert

```
const [addTask] = useMutation(UpdateTaskState, {
  refetchQueries: [
    {
      query: TaskListPageQuery,
      variables: { projectId: "1" }
    }
  ]
});
```

Cache direkt modifizieren

- Auf den Cache kann per API zugegriffen werden
- Über die API kann der Cache gelesen und verändert werden
- Per GraphQL Query, per GraphQL Fragment oder direkt per API

```
cache.writeFragment({
  id: "T5",
  fragment: gql`
    fragment UpdatedTask on Task {
      newState
    }
  `,
  data: {
    newState: "RUNNING"
  }
});
```

Cache direkt modifizieren: Per API

- Besonders hilfreich, wenn Listen angepasst werden müssen (Elemente hinzufügen, Löschen)
- API ist sehr gewöhnungsbedürftig und m.E. auch zu fragil

ÜBUNG 3: CACHE AKTUALISIEREN

Die TaskListPage sollte immer aktuell sein...

1. Erweitere den useQuery-Aufruf in TaskListPage.js, so dass dieser zusätzlich zum Cache-Lookup auch einen Netzwerk-Request macht
   - Wenn Du AddNewTask einen neuen Task anlegst, sollte dieser nun unmittelbar in der Liste sichtbar werden (kein LoadingIndicator!)

2. Auch nach dem "Quick Add Task" direkt auf der TaskListPage sollte die Liste aktualisiert werden
   - Aktiviere die QuickAddTaskForm-Komponente in TaskListPage, so dass diese angezeigt wird (ist auskommentiert dort vorhanden)
   - Wenn eine in der QuickAddTaskForm-Komponente ein neuer Task gespeichert wurde, soll dieser oben in der Liste auftauchen (ohne dass User Browser Fenster neu laden muss)
   - Wie kannst Du das implementieren?

Der Apollo Client - Ausblick

Mögliche Themen

- Eure Anwendung
- Client State und Reactive Variables
- TypeScript Support für Client und Server
- React Server Components

Vielen Dank!

Repository: https://github.com/nilshartmann/graphql-apollo-workshop

Fragen und Kontakt: nils@nilshartmann.net