

NILS HARTMANN



Schema Stitching with Apollo GraphQL

Slides: <https://bit.ly/meetup-schema-stitching>

NILS HARTMANN

Software Developer from Hamburg

**JavaScript, TypeScript, React
Java**

Trainings, Workshops

nils@nilshartmann.net

@NILSHARTMANN

Beer Rating! - Apollo GraphQL × Nils

localhost:9080

The Beer Backend Java 1.8.0_73 since 814s GraphQL
Rating Backend NodeJS 8.9.3 since 1044.635s GraphQL
Stitcher NodeJS 8.9.3 since 1028.369s GraphQL

BEER RATING

<https://github.com/nilshartmann/apollo-graphql-examples>

Frydenlund 150 NOK

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?

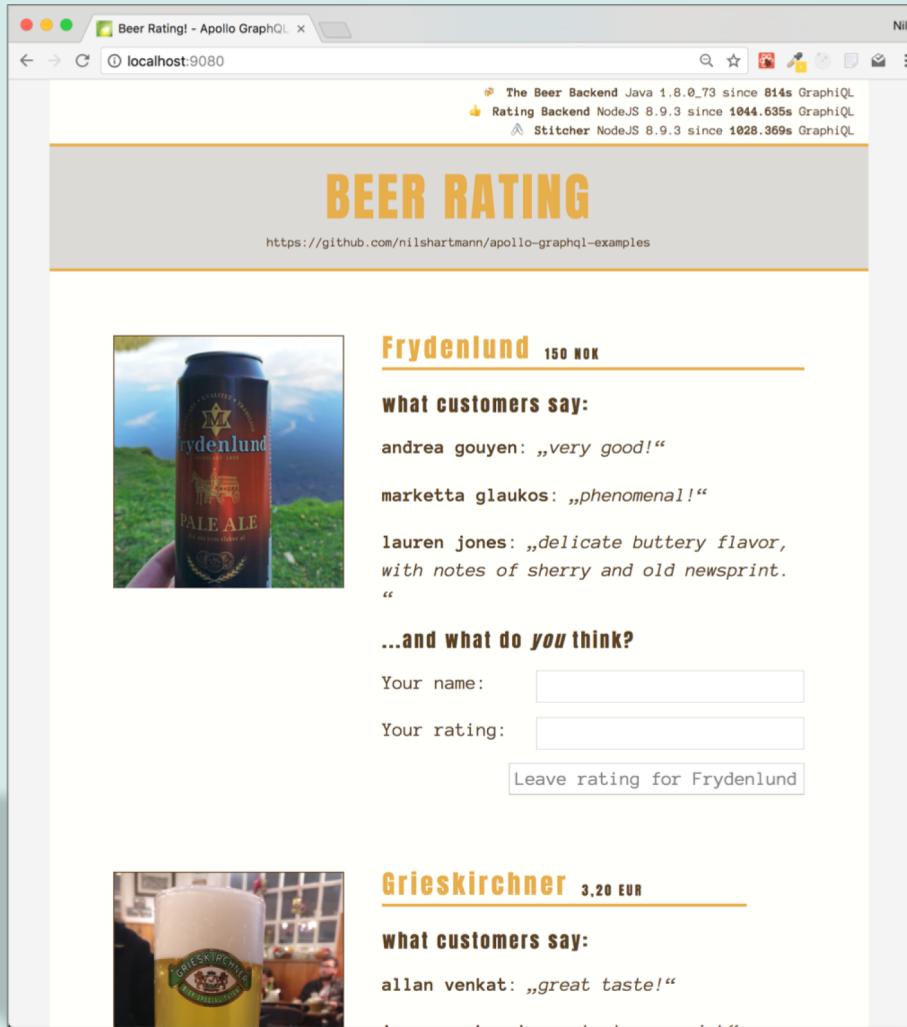
Your name:

Your rating:

Grieskirchner 3,20 EUR

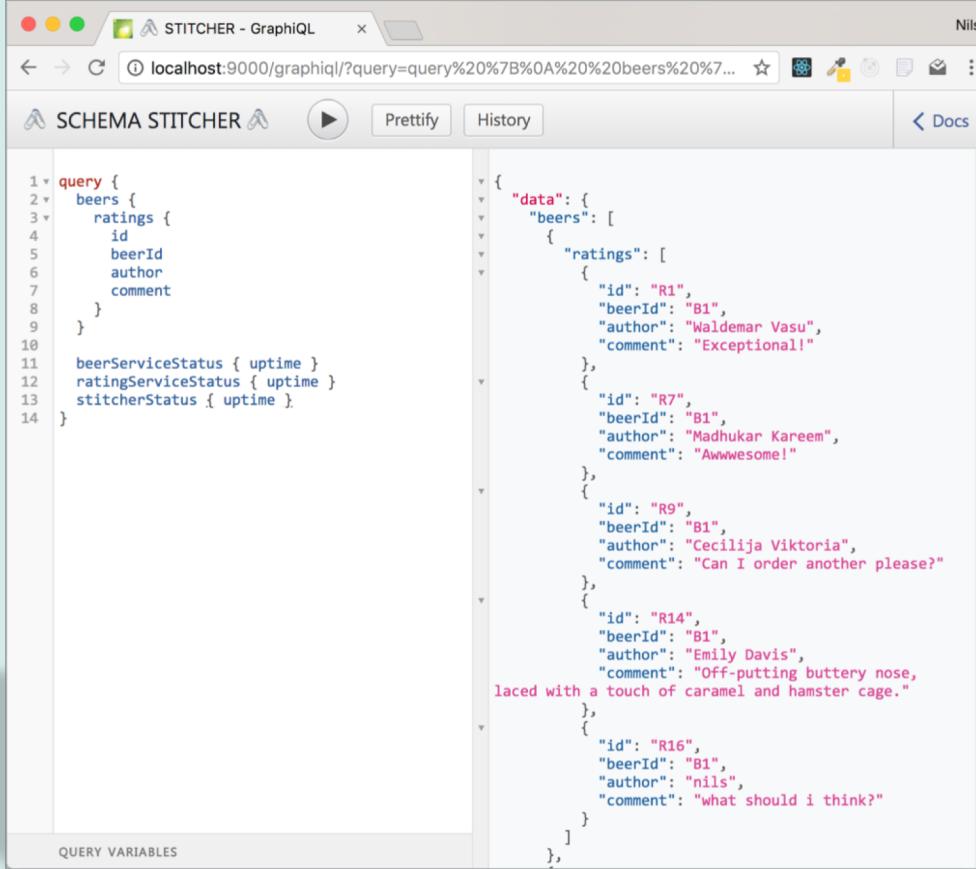
what customers say:

allan venkat: „great taste!“



Example

Source Code: <https://bit.ly/graphql-stitching-example>



The screenshot shows the GraphiQL interface running on a Mac OS X system. The title bar reads "STITCHER - GraphiQL". The main area has two panes: the left pane contains a GraphQL query, and the right pane displays the resulting JSON data.

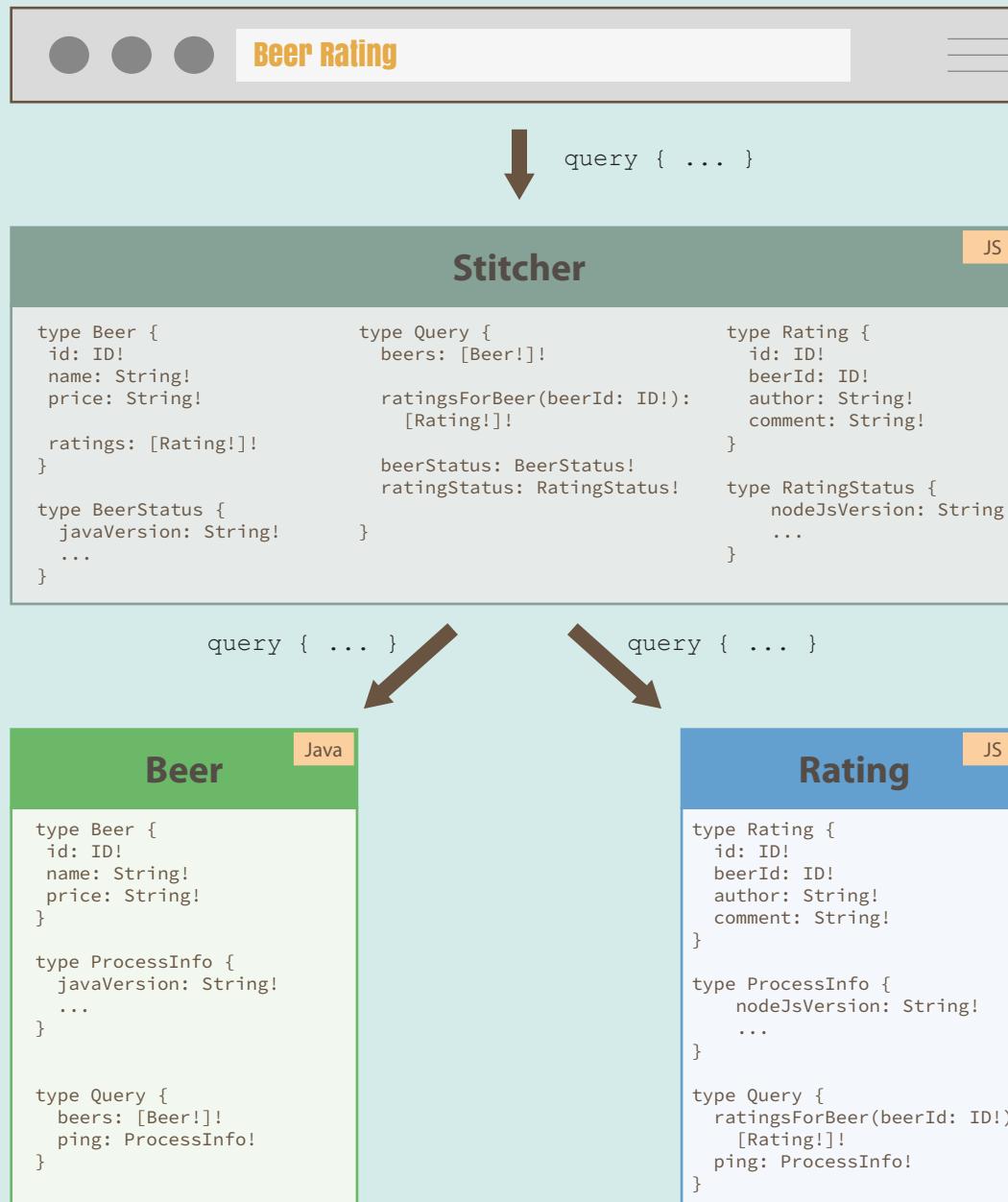
```
query {
  beers {
    ratings {
      id
      beerId
      author
      comment
    }
  }
  beerServiceStatus { uptime }
  ratingServiceStatus { uptime }
  stitcherStatus { uptime }
}
```

```
[{"id": "R1", "beerId": "B1", "author": "Waldemar Vasu", "comment": "Exceptional!"}, {"id": "R7", "beerId": "B1", "author": "Madhukar Kareem", "comment": "Awwesome!"}, {"id": "R9", "beerId": "B1", "author": "Cecilija Viktoria", "comment": "Can I order another please?"}, {"id": "R14", "beerId": "B1", "author": "Emily Davis", "comment": "Off-putting buttery nose, laced with a touch of caramel and hamster cage."}, {"id": "R16", "beerId": "B1", "author": "nils", "comment": "What should i think?"}]
```

Demo: GraphQL

<http://localhost:9000>

ARCHITECTURE



ALIGN MULTIPLE APIs

Rename ambiguous types and fields

Stitcher JS

```
type Beer {  
  id: ID!  
  name: String!  
  price: String!  
  
  ratings: [Rating!]!  
}  
  
type BeerStatus {  
  javaVersion: String!  
  ...  
}  
  
type Query {  
  beers: [Beer!]!  
  
  ratingsForBeer(beerId: ID!): [Rating!]!  
  
  beerStatus: BeerStatus!  
  ratingStatus: RatingStatus!  
}  
  
type Rating {  
  id: ID!  
  beerId: ID!  
  author: String!  
  comment: String!  
}  
  
type RatingStatus {  
  nodeJsVersion: String!  
  ...  
}
```

query { ... }

query { ... }

Beer Java

```
type Beer {  
  id: ID!  
  name: String!  
  price: String!  
}  
  
type ProcessInfo {  
  javaVersion: String!  
  ...  
}  
  
type Query {  
  beers: [Beer!]!  
  ping: ProcessInfo!  
}
```

Rating JS

```
type Rating {  
  id: ID!  
  beerId: ID!  
  author: String!  
  comment: String!  
}  
  
type ProcessInfo {  
  nodeJsVersion: String!  
  ...  
}  
  
type Query {  
  ratingsForBeer(beerId: ID!): [Rating!]!  
  ping: ProcessInfo!  
}
```

ALIGN MULTIPLE APIs

Add fields to existing types by linking to schemas

Stitcher

JS

```
type Beer {  
  id: ID!  
  name: String!  
  price: String!  
  ratings: [Rating!]!  
}  
  
type BeerStatus {  
  javaVersion: String!  
  ...  
}  
  
type Query {  
  beers: [Beer!]!  
  ratingsForBeer(beerId: ID!): [Rating!]!  
  beerStatus: BeerStatus!  
  ratingStatus: RatingStatus!  
}  
  
type Rating {  
  id: ID!  
  beerId: ID!  
  author: String!  
  comment: String!  
}  
  
type RatingStatus {  
  nodeJsVersion: String!  
  ...  
}
```

query { ... }

query { ... }

Beer

Java

```
type Beer {  
  id: ID!  
  name: String!  
  price: String!  
}  
  
type ProcessInfo {  
  javaVersion: String!  
  ...  
}  
  
type Query {  
  beers: [Beer!]!  
  ping: ProcessInfo!  
}
```

Rating

JS

```
type Rating {  
  id: ID!  
  beerId: ID!  
  author: String!  
  comment: String!  
}  
  
type ProcessInfo {  
  nodeJsVersion: String!  
  ...  
}  
  
type Query {  
  ratingsForBeer(beerId: ID!): [Rating!]!  
  ping: ProcessInfo!  
}
```

APOLLO GRAPHQL

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

- Open-Source-Framework for building GraphQL Server (JavaScript)
- Exposes Types and Resolver (Schema) via HTTP endpoint

TASKS

Implementing a Schema Stitcher using Apollo Server...

1. **Create a "Remote Schema" for each endpoint**
2. Rename the `ProcessInfo` type from each schema
3. Create the "link" between the two schemas
 - Add `ratings` field to `Beer` type
4. Merge schemas into one new Schema
5. Expose merged Schema via HTTP Endpoint

CREATE REMOTE SCHEMAS

1. Configure Network Connection

```
import { HttpLink } from "apollo-link-http";
import fetch from "node-fetch";
import { introspectSchema, makeRemoteExecutableSchema } from "graphql-tools";

async function createRemoteSchema(uri) {

  const link = new HttpLink({ uri, fetch });

  const schema = await introspectSchema(link);

  return makeRemoteExecutableSchema({
    schema,
    link
  });
}

const beerSchema = await createRemoteSchema("localhost:9010/graphql");
const ratingSchema = await createRemoteSchema("localhost:9020/graphql");
```

CREATE REMOTE SCHEMAS

2. Run GraphQL *introspection query*

Standard query for retrieving a schema

```
import { HttpLink } from "apollo-link-http";
import fetch from "node-fetch";
import { introspectSchema, makeRemoteExecutableSchema } from "graphql-tools";

async function createRemoteSchema(uri) {

  const link = new HttpLink({ uri, fetch });

  const schema = await introspectSchema(link);

  return makeRemoteExecutableSchema({
    schema,
    link
  });
}

const beerSchema = await createRemoteSchema("localhost:9010/graphql");
const ratingSchema = await createRemoteSchema("localhost:9020/graphql");
```

CREATE REMOTE SCHEMAS

3. Create an executable schema

Executable Schema is Apollo's schema abstraction

```
import { HttpLink } from "apollo-link-http";
import fetch from "node-fetch";
import { introspectSchema, makeRemoteExecutableSchema } from "graphql-tools";

async function createRemoteSchema(uri) {

  const link = new HttpLink({ uri, fetch });

  const schema = await introspectSchema(link);

  return makeRemoteExecutableSchema({
    schema,
    link
  });
}

const beerSchema = await createRemoteSchema("localhost:9010/graphql");
const ratingSchema = await createRemoteSchema("localhost:9020/graphql");
```

CREATE REMOTE SCHEMAS

4. Create the remote schemas

Need to do this for each remote endpoint

```
import { HttpLink } from "apollo-link-http";
import fetch from "node-fetch";
import { introspectSchema, makeRemoteExecutableSchema } from "graphql-tools";

async function createRemoteSchema(uri) {

  const link = new HttpLink({ uri, fetch });

  const schema = await introspectSchema(link);

  return makeRemoteExecutableSchema({
    schema,
    link
  });
}

const beerSchema = await createRemoteSchema("localhost:9010/graphql");
const ratingSchema = await createRemoteSchema("localhost:9020/graphql");
```

TASKS

Implementing a Schema Stitcher...

1. Create a "Remote Schema" for each endpoint
2. **Rename the ProcessInfo type from each schema**
3. Create the "link" between the two schemas
 - Add `ratings` field to `Beer` type
4. Merge schemas into one new Schema
5. Expose merged Schema via HTTP Endpoint

TRANSFORMING SCHEMAS

Rename Types and Root-Fields for our overall Schema

1. Type `ProcessInfo` to `{System}Status`

The diagram shows a transformation from the original schema to a new schema. On the left, under the heading "Beer", is the original schema:

```
type ProcessInfo {  
    name: String!  
    javaVersion: String!  
}
```

A dashed arrow points from this schema to the right, where the transformed schema is shown:

```
type BeerStatus {  
    name: String!  
    javaVersion: String!  
}
```

The diagram shows a transformation from the original schema to a new schema. On the left, under the heading "Rating", is the original schema:

```
type ProcessInfo {  
    name: String!  
    nodeJsVersion: String!  
}
```

A dashed arrow points from this schema to the right, where the transformed schema is shown:

```
type RatingStatus {  
    name: String!  
    nodeJsVersion: String!  
}
```

TRANSFORMING SCHEMAS

Step 2: Rename Types and Root-Fields

1. Type `ProcessInfo` to `{System}Status`

Beer

```
type ProcessInfo {  
  name: String!  
  javaVersion: String!  
}
```

```
type BeerStatus {  
  name: String!  
  javaVersion: String!  
}
```

2. Root-Field `ping` to `{system}Status`

```
Query {  
  beers: [Beer!]!  
  ping: ProcessInfo!  
}
```

```
Query {  
  beers: [Beer!]!  
  beerStatus: BeerStatus!  
}
```

Rating

```
type ProcessInfo {  
  name: String!  
  nodeJsVersion: String!  
}
```

```
type RatingStatus {  
  name: String!  
  nodeJsVersion: String!  
}
```

```
Query {  
  ratings: [Rating!]!  
  ping: ProcessInfo!  
}
```

```
Query {  
  ratings: [Rating!]!  
  ratingStatus: RatingStatus!  
}
```

TRANSFORMING SCHEMAS

1. **transformSchema** runs a list of Transform operations

- Returns new Apollo Schema instance

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

function renameInSchema(schema) {
  return transformSchema(schema, [
    ...
  ]);
}
```

TRANSFORMING SCHEMAS

2. Rename Type `ProcessInfo` to `{System}Status`

```
import { transformSchema, RenameTypes, RenameRootFields } from "graphql-tools";

function renameInSchema(schema, systemName) {
  return transformSchema(schema, [
    new RenameTypes(
      name => (name === "ProcessInfo" ? `${systemName}Status` : name
    ),
    new RenameRootFields(
      (op, name) => name === "ping" ? : `${systemName}Status` : name
    )
  ]);
}
```

TRANSFORMING SCHEMAS

3. Rename Root-Field `ping` to `{system}Status`

```
import { transformSchema, RenameTypes, RenameRootFields } from "graphql-tools";

function renameInSchema(schema, systemName) {
  return transformSchema(schema, [
    new RenameTypes(
      name => (name === "ProcessInfo" ? `${systemName}Status` : name
    ),
    new RenameRootFields(
      (op, name) => name === "ping" ? : `${systemName}Status` : name
    )
  ]);
}
```

TRANSFORMING SCHEMAS

4. Run the transformation in both Schemas

```
import { transformSchema, RenameTypes, RenameRootFields } from "graphql-tools";

function renameInSchema(schema, systemName) {
  return transformSchema(schema, [
    new RenameTypes(
      name => (name === "ProcessInfo" ? `${systemName}Status` : name
    ),
    new RenameRootFields(
      (op, name) => name === "ping" ? : `${systemName}Status` : name
    )
  ]);
}

const renamedBeerSchema = renameInSchema(beerSchema, "Beer");
const renamedRatingSchema = renameInSchema(ratingSchema, "Rating");
```

TASKS

Implementing a Schema Stitcher...

1. Create a "Remote Schema" for each endpoint
2. Rename the `ProcessInfo` type from each schema
3. **Create the "link" between the two schemas**
 - **Add ratings field to Beer type**
4. Merge schemas into one new Schema
5. Expose merged Schema via HTTP Endpoint

LINK SCHEMAS

Enhance existing schema

1. Add ratings to Beer Schema

Beer

```
type Beer {  
    id: ID!  
    name: String!  
    price: String  
}
```

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

Rating

```
type Rating { . . . }
```

LINK SCHEMAS

Enhance existing schema

1. Add ratings to Beer Schema
2. Implementation: Delegate ratings to Rating API

Beer

```
type Beer {  
    id: ID!  
    name: String!  
    price: String  
}
```

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

Rating

```
type Rating { . . . }  
  
Query {  
    ratingsForBeer(beerId: ID!): [Ratings!]!  
}
```

implemented by

LINK SCHEMAS

1. Extend existing Type

- Add ratings field to Beer Schema

```
function createLinkedSchema() {  
  return {  
    linkedTypeDefs: `  
      extend type Beer {  
        ratings: [Rating!]!  
      }  
    `,  
  }  
}
```

LINK SCHEMAS

2. Add Resolver function

Naming convention as in "regular" Apollo Resolvers

```
function createLinkedSchema(ratingSchema) {  
  return {  
    linkedTypeDefs: `extend type Beer {ratings: [Rating!]! }`,  
    linkedResolvers: {  
      Beer: {  
        ratings: {  
          fragment: `fragment BeerFragment on Beer { id }`,  
          resolve: (parent, args, context, info) =>  
            info.mergeInfo.delegateToSchema({  
              schema: ratingSchema,  
              operation: "query",  
              fieldName: "ratingsForBeer",  
              args: { beerId: parent.id },  
              context, info  
            });  
        }  
      }  
    }  
  }  
}
```

LINK SCHEMAS

3. Use Fragment to add fields needed for resolver

Ensures that all required fields are queried, even if not specified in user's query

```
function createLinkedSchema(ratingSchema) {  
  return {  
    linkedTypeDefs: `extend type Beer {ratings: [Rating!]! }`,  
    linkedResolvers: {  
      Beer: {  
        ratings: {  
          fragment: `fragment BeerFragment on Beer { id }`,  
          resolve: (parent, args, context, info) =>  
            info.mergeInfo.delegateToSchema({  
              schema: ratingSchema,  
              operation: "query",  
              fieldName: "ratingsForBeer",  
              args: { beerId: parent.id },  
              context, info  
            });  
        }  
      }  
    }  
  }  
}
```

LINK SCHEMAS

4. Delegate execution to other schema ("Rating" in our example)

```
function createLinkedSchema(ratingSchema) {
  return {
    linkedTypeDefs: `extend type Beer {ratings: [Rating!]! }`,
    linkedResolvers: {
      Beer: {
        ratings: {
          fragment: `fragment BeerFragment on Beer { id }`,
          resolve: (parent, args, context, info) =>
            info.mergeInfo.delegateToSchema({
              schema: ratingSchema,
              operation: "query",
              fieldName: "ratingsForBeer",
              args: { beerId: parent.id },
              context,
              info
            });
        }
      }
    }
}
```

LINK SCHEMAS

4. Delegate execution to other schema ("Rating" in our example)

```
function createLinkedSchema(ratingSchema) {  
  return {  
    linkedTypeDefs: `extend type Beer {ratings: [Rating!]! }`,  
    linkedResolvers: {  
      Beer: {  
        ratings: {  
          fragment: `fragment BeerFragment on Beer { id }`,  
          resolve: (parent, args, context, info) =>  
            info.mergeInfo.delegateToSchema({  
              schema: ratingSchema,  
              operation: "query",  
              fieldName: "ratingsForBeer",  
              args: { beerId: parent.id },  
              context, info  
            });  
        }  
      }  
    }  
  }  
}
```

Runs on Rating Schema:

```
query {  
  ratingsForBeer(beerId: ...) {  
    ...  
  }  
}
```

TASKS

Implementing a Schema Stitcher...

1. Create a "Remote Schema" for each endpoint
2. Rename the `ProcessInfo` type from each schema
3. **Create the "link" between the two schemas**
 - **Add ratings field to Beer type**
4. Merge schemas into one new Schema
5. Expose merged Schema via HTTP Endpoint

MERGE SCHEMAS

Recap: Create, Transform and Link Schemas

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

const beerSchema = await createRemoteSchema("localhost:9010/graphql");
const ratingSchema = await createRemoteSchema("localhost:9020/graphql");

const renamedBeerSchema = renameInSchema(beerSchema, "Beer");
const renamedRatingSchema = renameInSchema(ratingSchema, "Rating");

const { linkedTypeDefs, linkedResolvers } = createLinkedSchema(renamedRatingSchema);

const mergedSchema = mergeSchemas({
  schemas: [
    renamedBeerSchema,
    renamedRatingSchema,
    linkedTypeDefs
  ],
  resolvers: linkedResolvers
});
```

MERGE SCHEMAS

1. Merge Schemas together

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

const beerSchema = await createRemoteSchema("localhost:9010/graphql");
const ratingSchema = await createRemoteSchema("localhost:9020/graphql");

const renamedBeerSchema = renameInSchema(beerSchema, "Beer");
const renamedRatingSchema = renameInSchema(ratingSchema, "Rating");

const { linkedTypeDefs, linkedResolvers } = createLinkedSchema(renamedRatingSchema);

const mergedSchema = mergeSchemas({
  schemas: [
    renamedBeerSchema,
    renamedRatingSchema,
    linkedTypeDefs
  ],
  resolvers: linkedResolvers
});
```

PUBLISH MERGED SCHEMA

Publish Schema

Example: Using HTTP Endpoint with Express

```
import { graphqlExpress } from "apollo-server-express";
import express from "express";
import bodyParser from "body-parser";

const mergedSchema = mergeSchemas(. . .);

const app = express();

app.use("/graphql",
  bodyParser.json(),
  graphqlExpress({ schema: mergedSchema })
);

app.listen(PORT);
```



Thank you!

Slides: <https://bit.ly/meetup-schema-stitching>

Sample-Code: <https://bit.ly/graphql-stitching-example>

@NILSHARTMANN