NILS HARTMANN

Schema Stitching with Apollo GraphQL

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

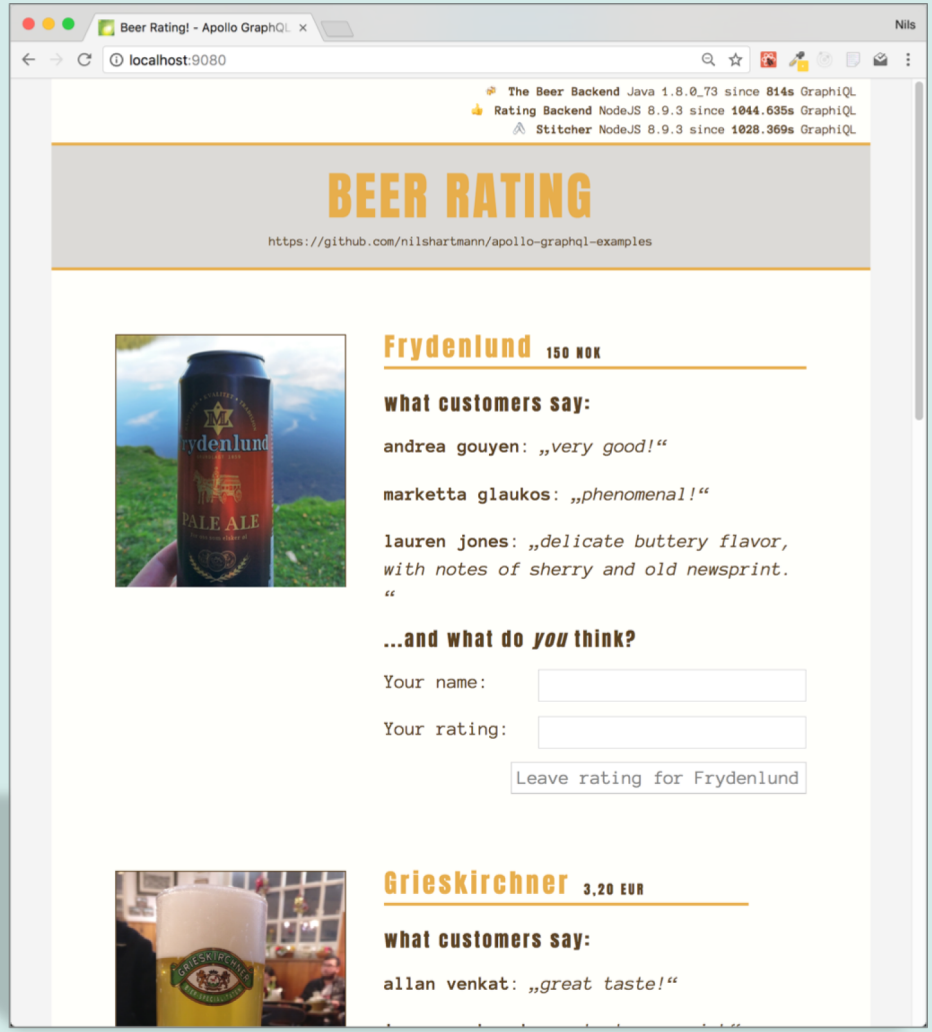
NILS HARTMANN

Software Developer from Hamburg

JavaScript, TypeScript, React
Java

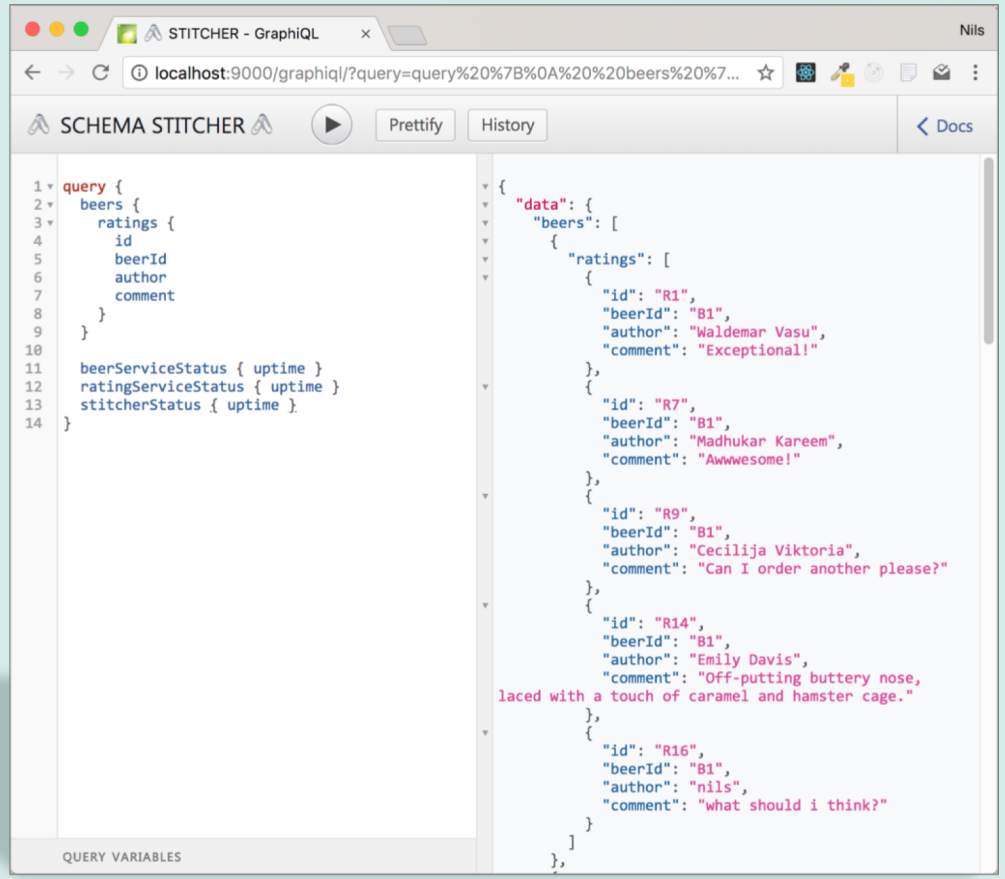
Trainings, Workshops

nils@nilshartmann.net



Example

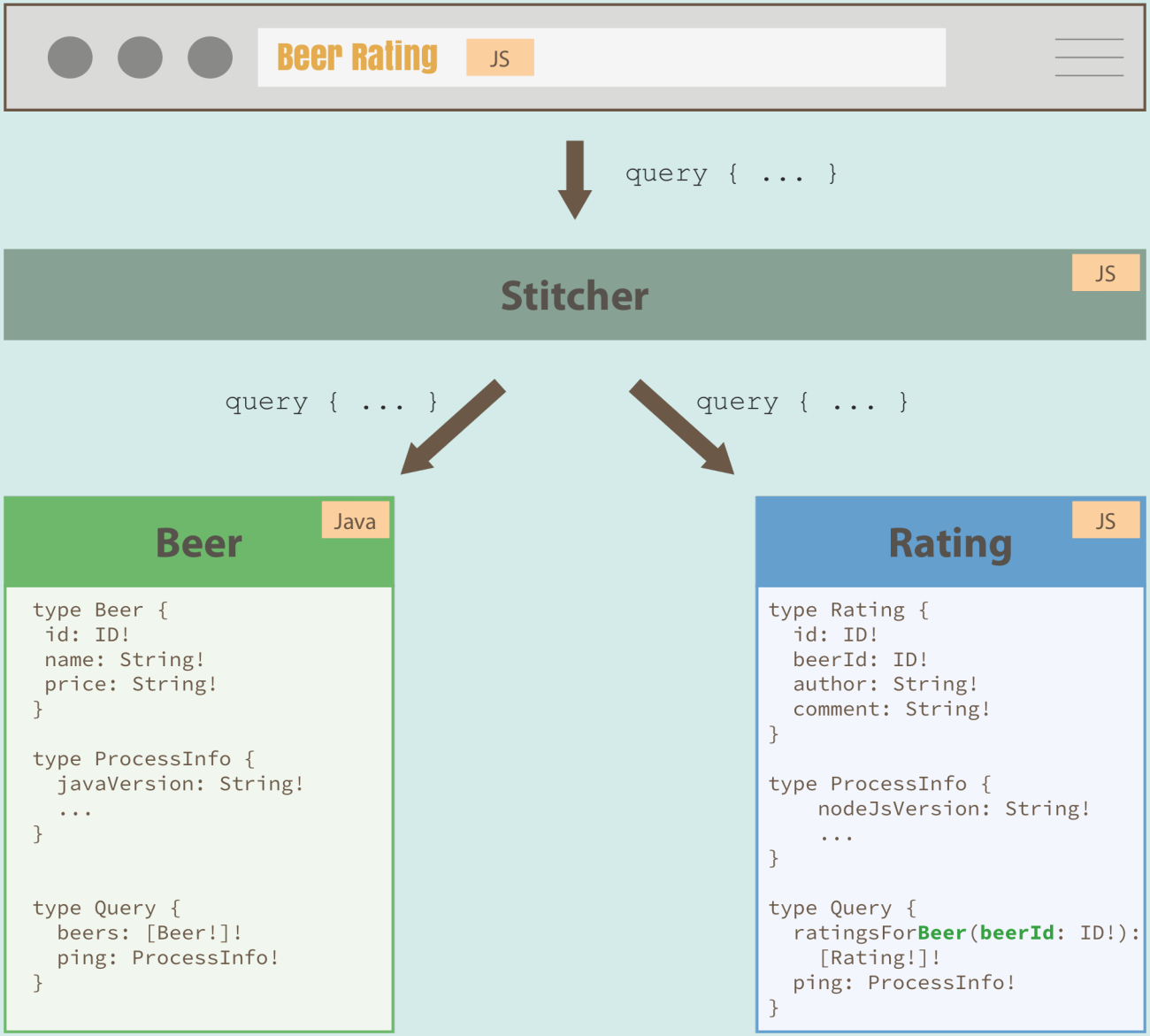
Source Code: https://bit.ly/graphql-stitching-example



Demo: GraphiQL

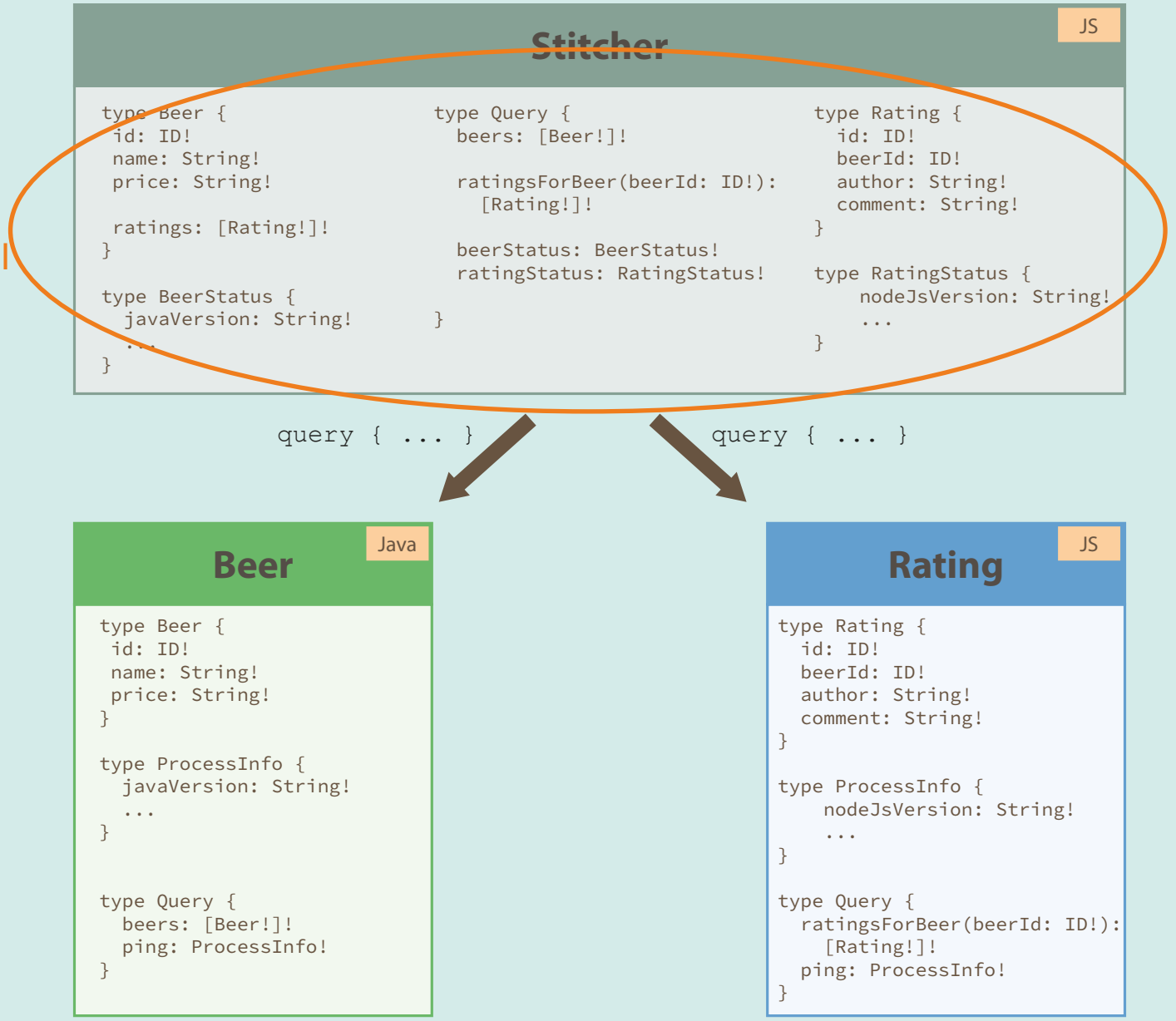
http://localhost:9000

ARCHITECTURE



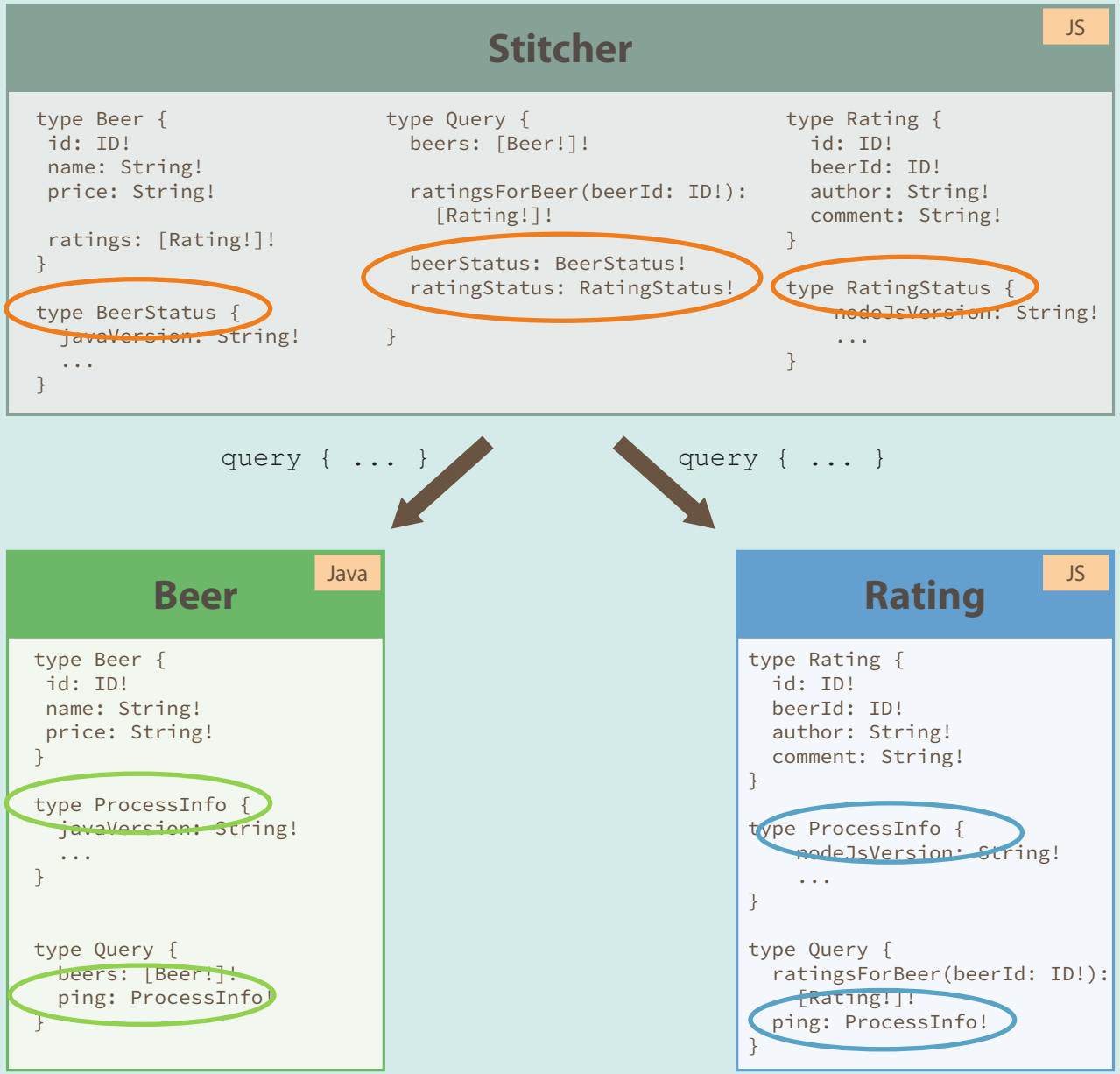
ALIGN MULTIPLE APIS

Provide **one** API for the Client



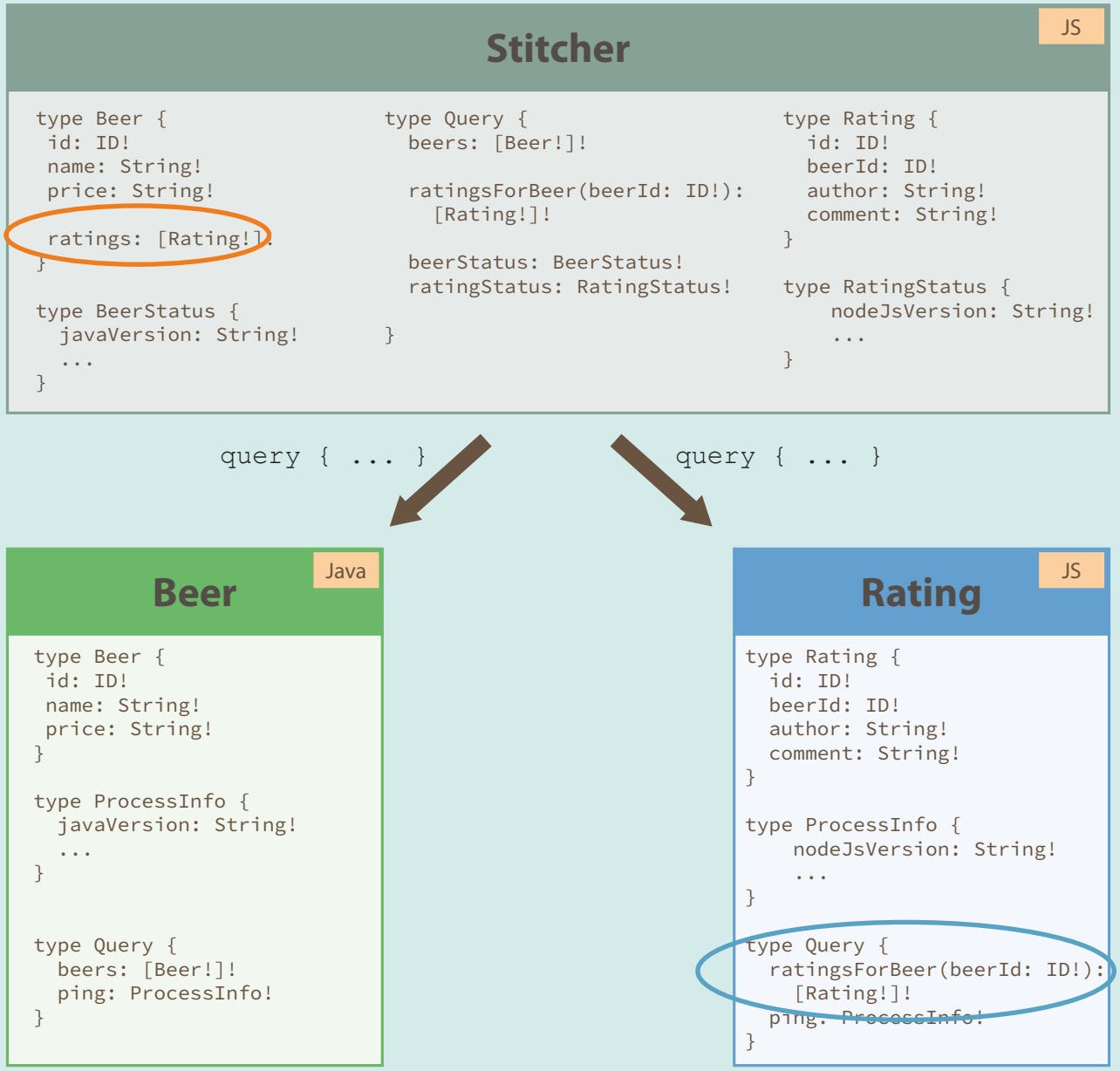
ALIGN MULTIPLE APIS

Rename ambigous types and fields



ALIGN MULTIPLE APIS

Add fields to existing types by linking to schemas



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

Example: GraphQL Server on one slide

```
const express = require("express");
const bodyParser = require("body-parser");
const { graphqlExpress } = require("apollo-server-express");
const { makeExecutableSchema } = require("graphql-tools");

const typeDefs = `
  type Query { hello: String! }
`;

const resolvers = {
  Query: { hello: () => "Hello, World" }
};

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

const app = express();
app.use("/graphql", bodyParser.json(), graphqlExpress({ schema }));
app.listen(9090);
```

TASKS

Implementing a Schema Stitcher using Apollo Server...

1. Create a "Remote Schema" for each endpoint
2. Rename `ProcessInfo` type and `ping` field 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

TASKS

Implementing a Schema Stitcher using Apollo Server...

1. **Create a "Remote Schema" for each endpoint**
2. Rename `ProcessInfo` type and `ping` field 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 `ProcessInfo` type and `ping` field 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 for our target Schema

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

Beer

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

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

Rating

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

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

TRANSFORMING SCHEMAS

Rename Root-Fields

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

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

Beer

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

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

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

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

Rating

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

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

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

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

TRANSFORMING SCHEMAS

1. transformSchema runs a list of Transform operations on a given Schema

- Returns new Apollo Schema instance

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

function renameInSchema(schema) {
  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");
```

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
    )
  ]);
}


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

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
    )
  ]);
}


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

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 `ProcessInfo` type and `ping` field 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!]!
}
```

implemented by

Rating

```
type Rating { . . . }

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

LINK SCHEMAS

1. Create new Schema & extend existing Type

- Add **`ratings`** field to **`Beer`** type

```
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 `ProcessInfo` type and `ping` field 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