
Working with database models and relations
LEARNING VOYAGE

Working with database models and relations

This lesson covers

- Creating object types for database models
- Defining a global context shared among all resolvers
- Resolving fields from database models and transforming their names and values
- Resolving one-to-one and one-to-many relations
- Working with database views and join statements

Running and connecting to databases

- The easiest way to get this project's databases up and running with sample data is to use Docker.
- Docker uses your OS virtualization to provide software in packaged containers, It's available on all three major operating systems.
- I've prepared two Docker containers for this project: one for PostgreSQL and one for MongoDB.
- They both have the database structure created and the sample data imported.

Running and connecting to databases

- Once Docker is running, you can run this command to start both databases.

```
$ npm run start-dbs
```

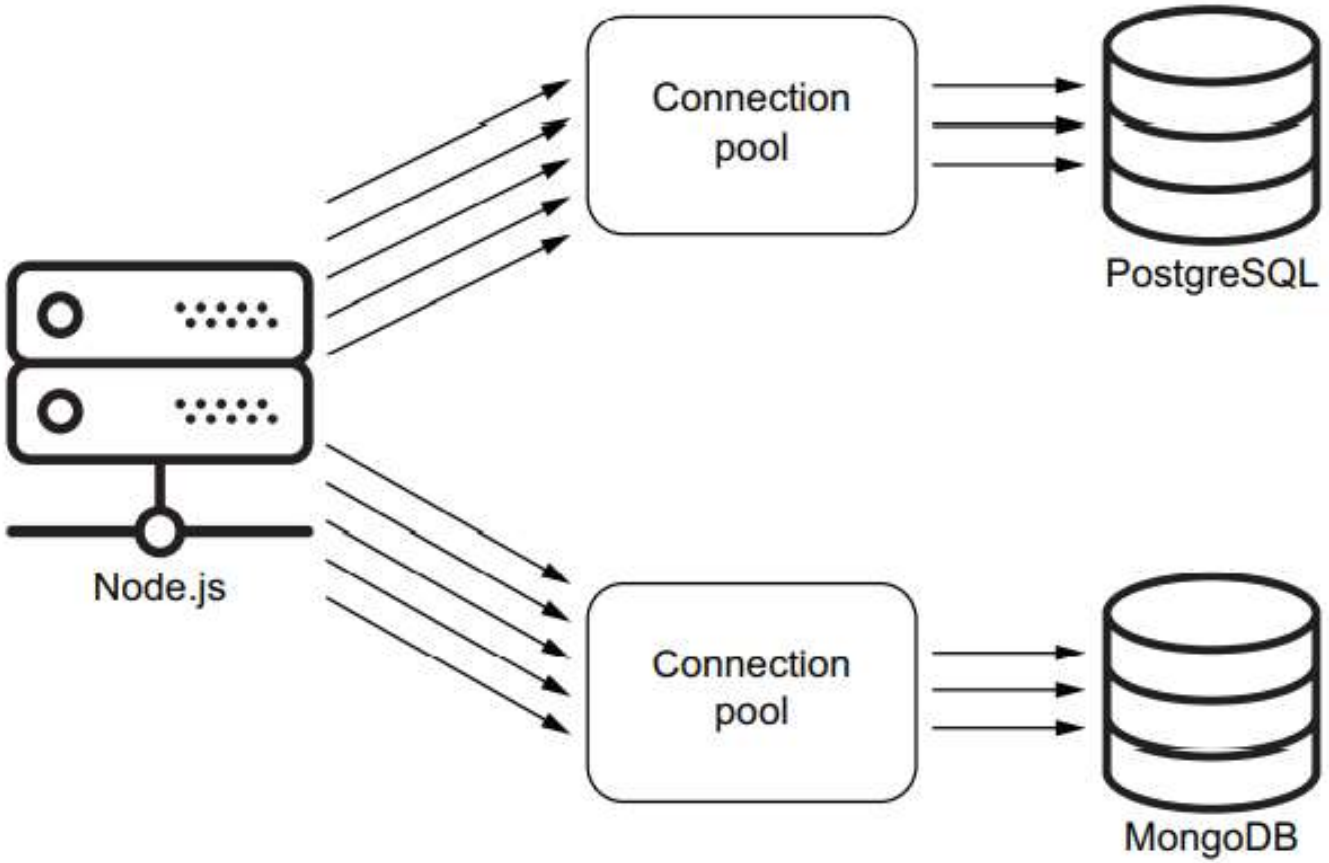
Running and connecting to databases

- Use the following SQL queries to see the data in PostgreSQL.

```
SELECT * FROM azdev.users;
SELECT * FROM azdev.tasks;
SELECT * FROM azdev.approaches;
```

- For the data in MongoDB, you can use this find command.

```
db.approachDetails.find({});
```


Connection
pool
PostgreSQL
Node.js
Connection
pool
MongoDB

The taskMainList query

- Let's start by implementing the main Task type, Here is the SDL text we prepared for it.

```
type Task implements SearchResultItem {
  id: ID!
  createdAt: String!
  content: String!
  tags: [String!]!
  approachCount: Int!

  # author: User!
  # approachList: [Approach!]!
}
```

The taskMainList query

- The first query field that will use this Task type is the list of the latest Tasks that will be displayed on the main page of the AZdev app.

- We named that field taskMainList.

```
type Query {
        taskMainList: [Task!]
}
```

The taskMainList query

- Here's a GraphQL query that we can use to start testing this feature.

```
query {
  taskMainList {
    id
    content
    tags
    approachCount
    createdAt
  }
}
```

Defining object types

```
import {
  GraphQLID,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLNonNull,
  GraphQLList,
} from 'graphql';

const Task = new GraphQLObjectType({
  name: 'Task',
  fields: {
    id: { type: new GraphQLNonNull(GraphQLID) },
```

Defining object types

```
    content: { type: new GraphQLNonNull(GraphQLString) },
    tags: {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(GraphQLString))
      ),
    },
    approachCount: { type: new GraphQLNonNull(GraphQLInt) },
    createdAt: { type: new GraphQLNonNull(GraphQLString) },
  },
});

export default Task;
```

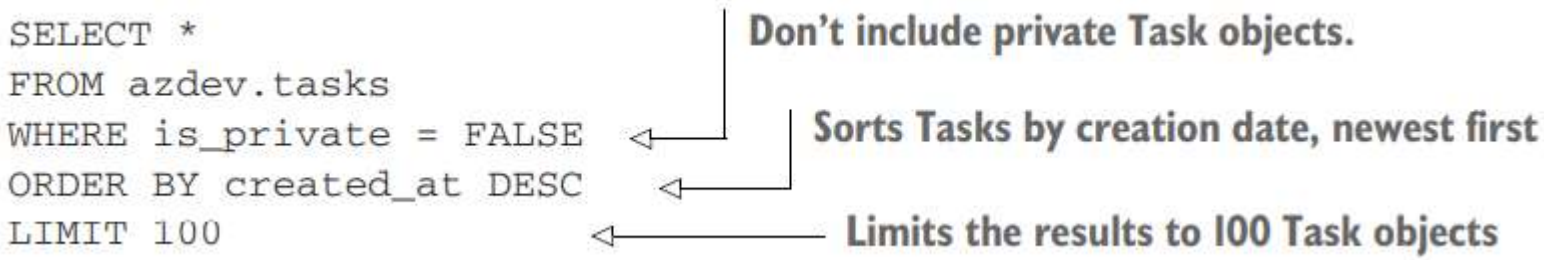
Defining object types

- The worst part about this is probably the type for the tags field.
- The simple [String!]! had to be written with nested calls of three functions:

```
new GraphQLNonNull(
  new GraphQLList(
    new GraphQLNonNull(
      GraphQLString
    )
  )
)
```

The context object

- It means the type for the taskMainList field should be new GraphQLList(new GraphQLNonNull(Task)).
- To resolve this field, we need to execute this SQL statement on the PostgreSQL database



```
// ...
import pgClient from './db/pg-client';

async function main() {
  const { pgPool } = await pgClient();
  const server = express();
  // ...

  server.use(
    '/',
    graphqlHTTP({
      schema,
      context: { pgPool },
      graphiql: true,
    }),

    );

    // ...
  }

  main();
```

The context object

- The pgPool object has a query method we can use to execute a SQL statement.
- We can use it this way to execute the SELECT statement

```
const pgResp = await pgPool.query(`
  SELECT *
  FROM azdev.tasks
  WHERE is_private = FALSE
  ORDER BY created_at DESC
  LIMIT 100
`);
```

The context object

- The pgResp object will have a rows property holding an array of objects representing the rows returned by the database

```
[
  { id: 1, content: 'Task #1', approach_count: 1,   ·-·-·},
  { id: 2, content: 'Task #2', approach_count: 1,   ·-·-·},
  ·-·-·
]
```

The context object

- The context object is exposed to each resolver function as the third argument (after source and args).

```
resolve: (source, args, context, info) => {}
```

```
import {
  // ...
  GraphQLList,
} from 'graphql';
// ...
import Task from './types/task';

const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // ...

    taskMainList: {
      type: new GraphQLList(new GraphQLNonNull(Task)),
      resolve: async (source, args, { pgPool }) => {
        const pgResp = await pgPool.query(`
          SELECT *
          FROM azdev.tasks
          WHERE is_private = FALSE
          ORDER BY created_at DESC
          LIMIT 100
        `);
        return pgResp.rows;
      },
    },
  },
});
// ...
```

The context object

- Go ahead and test things now.
- The API should be able to answer this query.

```
{
  taskMainList {
    id
    content
  }
}
```

```
{
  taskMainList {
    id
    content
  }
}
```

QUERY VARIABLES


```
{
  "data": {
    "taskMainList": [
      {
        "id": "1",
        "content": "Make an image in HTML change based on the theme color mode (dark or light)"
      },
      {
        "id": "2",
        "content": "Get rid of only the unstaged changes since the last git commit"
      },
      {
        "id": "3",
        "content": "The syntax for a switch statement (AKA case statement) in JavaScript"
      },
      {
        "id": "4",
        "content": "Calculate the sum of numbers in a JavaScript array"
      },
      {
        "id": "6",
        "content": "Create a secure one-way hash for a text value
```

Transforming field names

- In some cases, we need the API to represent columns and rows in the database with a different structure.
- Maybe the database has a confusing column name; or maybe we want the API to consistently use camel-case for all field names, and the database uses snake-case for its columns.
- This latter situation is exactly what we have to deal with next.

METHOD #1

- For example, if we have a function caseMapper that takes an object and makes all of its properties camel-case, we can modify the resolver of taskMainList as follows.

```
resolve: async (source, args, { pgPool }) => {
  const pgResp = await pgPool.query(
    // ....
  );
  return pgResp.rows.map(caseMapper);
},
```

METHOD #2

- We can create custom resolvers for the fields that need to be converted.
- For example, we can change the createdAt field to include this resolve function.

```
const Task = new GraphQLObjectType({
  name: 'Task',
  fields: {
      // .-.-.
      createdAt: {
        type: new GraphQLNonNull(GraphQLString),
        resolve: (source) => source.created_at,
      },
    },
});
```

METHOD #2

- We can create custom resolvers for the fields that need to be converted.
- For example, we can change the createdAt field to include this resolve function.

```
const Task = new GraphQLObjectType({
  name: 'Task',
  fields: {
      // .-.-.
      createdAt: {
        type: new GraphQLNonNull(GraphQLString),
        resolve: (source) => source.created_at,
      },
    },
});
```

METHOD #3

- For example, here's a version of the taskMainList resolver function to implement this method

```
resolve: async (source, args, { pgPool }) => {
  const pgResp = await pgPool.query(`
    SELECT id, content, tags,
           approach_count AS "approachCount", created_at AS "createdAt"
    FROM azdev.tasks
    WHERE // ----
  `);
  return pgResp.rows;
},
```

Transforming field values

- let's serialize all date-time fields for the AZdev API using UTC ISO format.
- We can use the JavaScript toISOString method for this.
- We'll need to implement the createdAt field's resolver function using the following.

```
createdAt: {
  type: new GraphQLNonNull(GraphQLString),
  resolve: (source) => source.createdAt.toISOString(),
},
```

Transforming field values

```
{
  taskMainList {
    id
    content
    createdAt
  }
}
```


```
{
  "data": {
    "taskMainList": [
      {
        "id": "1",
        "content": "Make an image in HTML change based on the theme color mode (dark or light)",
        "createdAt": "2020-08-01T00:03:02.032Z"
      },
      {
        "id": "2",
        "content": "Get rid of only the unstaged changes since the last git commit",
        "createdAt": "2020-08-01T00:03:02.032Z"
      },
```

Transforming field values

- We need to transform this value into an array of strings instead (so, ["node", "git"]). We do that with a custom resolver function.

```
tags: {
  type: new GraphQLNonNull(
    new GraphQLList(new GraphQLNonNull(GraphQLString))
  ),
  resolve: (source) => source.tags.split(','),
},
```

- With that, the resolver will return an array when asked for the tags property

```
{
  taskMainList {
    id
    content
    tags
  }
}
```

```
{
  "data": {
    "taskMainList": [
      {
        "id": "1",
        "content": "Make an image in HTML change based on the theme
color mode (dark or light)",
        "tags": [
          "code",
          "html"
        ]
      },
      {
        "id": "2",
        "content": "Get rid of only the unstaged changes since the
last git commit",
        "tags": [
          "command",
          "git"
        ]
      },
```

Separating interactions with PostgreSQL

- let's do a small refactoring. Instead of using SQL statements directly in resolver functions, let's introduce a module responsible for communicating with PostgreSQL and just use that module's API in the resolver functions.
- This separation of responsibilities will generally improve the readability of the API's code.
- The logic to fetch things from PostgreSQL will not be mixed with the logic to transform raw data into the public API.

```
import pgClient from './pg-client';
import sqls from './sqls';

const pgApiWrapper = async () => {
  const { pgPool } = await pgClient();
  const pgQuery = (text, params = {}) =>
    pgPool.query(text, Object.values(params));

  return {
    taskMainList: async () => {
      const pgResp = await pgQuery(sqls.tasksLatest);
      return pgResp.rows;
    },
  };
};

export default pgApiWrapper;
```

The tasksLatest SQL statement is already in api/src/db/sqls.js.

Separating interactions with PostgreSQL

```
// ·-·-·
import pgApiWrapper from './db/pg-api';

async function main() {
  const pgApi = await pgApiWrapper();

  // ·-·-·

  server.use(
    '/',
    graphqlHTTP({
      schema,
      context: { pgApi },
      graphiql: true,
    })
  );

  // ·-·-·
}
```

This line replaces the pg-client import line.

This line replaces the pgClient() call line.

Separating interactions with PostgreSQL

- Finally, we need to change the resolve function for taskMainList to use the new pgApi instead of issuing a direct SQL statement

```
taskMainList: {
  type: new GraphQLList(new GraphQLNonNull(Task)),
  resolve: async (source, args, { pgApi }) => {
    return pgApi.taskMainList();
  },
},
```

Error reporting

- Let's look at an example, Fake an error anywhere in the code that resolves data for the taskMainList field, something like the following.

```
const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // ...
    taskMainList: {
      type: new GraphQLList(new GraphQLNonNull(Task)),
      resolve: async (source, args, { pgApi }) => {
        return pgApi.taksMainList();
      },
    },
  },
});
```

Typo!

Error reporting

- Now observe what happens when you ask for the taskMainList field in GraphiQL.

```
{
  taskMainList {
    id
    content
    createdAt
    tags
  }
}
```

```
async function main() {
  // ·-·-·

  server.use(
    '/',
    graphqlHTTP({
      schema,
      context: { pgApi },
      graphiql: true,
      customFormatErrorFn: (err) => {
        const errorReport = {
          message: err.message,
          locations: err.locations,
          stack: err.stack ? err.stack.split('\n') : [],
          path: err.path,
        };
        console.error('GraphQL Error', errorReport);
        return config.isDev
          ? errorReport
          : { message: 'Oops! Something went wrong! :(' };
      },
    }),
  );

  // ·-·-·
}
```

Makes the error stack show up in development, which is very handy

Logs the error in the server logs

Returns a generic error in production

Resolving relations

- The remaining fields on the Task type are author and approachList, We'll need to implement two new GraphQL types for them.
- I'll name them Author and Approach. These fields will not be leaf fields in a query. They represent relations.
- A Task has one Author and many Approaches, To resolve these fields, the GraphQL server will have to execute SQL statements over many tables and return objects from these tables.

```
{
  taskMainList {
    id
    content
    tags
    approachCount
    createdAt

    author {
      id
      username
      name
    }

    approachList {
      id
      content
      voteCount
      createdAt

      author {
        id
        username
        name
      }
    }
  }
}
```

Resolving a one-to-one relation

- We have to execute another SQL statement to get information about the User who authored it.
- You can find that SQL statement under sqls.usersFromIds (in api/src/db/sqls.js).

```
// $1: userIds
usersFromIds: `
  SELECT id, username,
         first_name AS "firstName", last_name AS "lastName",
         created_at AS "createdAt"
  FROM azdev.users
  WHERE id = ANY ($1)
`,
```

Resolving a one-to-one relation

- Let's design that function to accept a userId value as an argument.

```
const pgApiWrapper = async () => {
  // ....
  return {

    // ....
    userInfo: async (userId) => {
      const pgResp = await pgQuery(sqls.usersFromIds, { $1: [userId] });
      return pgResp.rows[0];
    },
  };
};
```

Passes $1 to the SQL statement as [userId]

Resolving a one-to-one relation

- To make the GraphQL server aware of the new author field, we need to define the User type.
- Everything in a GraphQL schema must have a type.
- In the SDL text, we had this structure for the User type.

```
type User {
  id: ID!
  username: String!
  name: String
  taskList: [Task!]!
}
```

We'll implement the taskList field under the me root field.

```
import {
  GraphQLID,
  GraphQLObjectType,
  GraphQLString,
  GraphQLNonNull,
} from 'graphql';

const User = new GraphQLObjectType({
  name: 'User',
  fields: {
    id: { type: new GraphQLNonNull(GraphQLID) },
    username: { type: GraphQLString },
    name: {
      type: GraphQLString,
      resolve: ({ firstName, lastName }) =>
        `${firstName} ${lastName}`,
    },
  },
});

export default User;
```

Resolving a one-to-one relation

```
import User from './user';

const Task = new GraphQLObjectType({
  name: 'Task',
  fields: {
    // ....

    author: {
      type: new GraphQLNonNull(User),
      resolve: (source, args, { pgApi }) =>
        pgApi.userInfo(source.userId),
    },
  },
});
```

Resolving a one-to-one relation

- That will do it, You can test the new relation with this query.

```
{
  taskMainList {
    content
    author {
      id
      username
      name
    }
  }
}
```

Resolving a one-to-one relation

```
{
  taskMainList {
    content
    author {
      id
      username
      name
    }
  }
}
```


```
{
  "data": {
    "taskMainList": [
      {
        "content": "Make an image in HTML change based on the theme color mode (dark or light)",
        "author": {
          "id": "1",
          "username": "test",
          "name": "null null"
        }
      },
      {
        "content": "Get rid of only the unstaged changes since the last git commit",
        "author": {
          "id": "1",
          "username": "test",
          "name": "null null"
        }
      },
```

DEALING WITH NULL VALUES

- There is a small problem in the data response in previous figure: the Author name was returned as null null.
- Why is that? The null concept is confusing, Different coders associate different meanings with it.
- You need to be careful to always consider the possibility of dealing with null, You should ask, “What if this is null?” about every variable you use in your code.
- This is one reason why languages like TypeScript and Flow are popular: they can help detect these problems.

DEALING WITH NULL VALUES

- There are many ways to implement that. Here's one.

```
name: {
  type: new GraphQLNonNull(GraphQLString),
  resolve: ({ firstName, lastName }) =>
    [firstName, lastName].filter(Boolean).join(' '),
},
```

DEALING WITH NULL VALUES

```
{
  taskMainList {
    content
    author {
      id
      username
      name
    }
  }
}
```

```
{
  "data": {
    "taskMainList": [
      {
        "content": "Make an image in HTML change based on the theme
color mode (dark or light)",
        "author": {
          "id": "1",
          "username": "test",
          "name": ""
        }
      },
      {
        "content": "Get rid of only the unstaged changes since the
last git commit",
        "author": {
          "id": "1",
          "username": "test",
          "name": ""
        }
      },
```

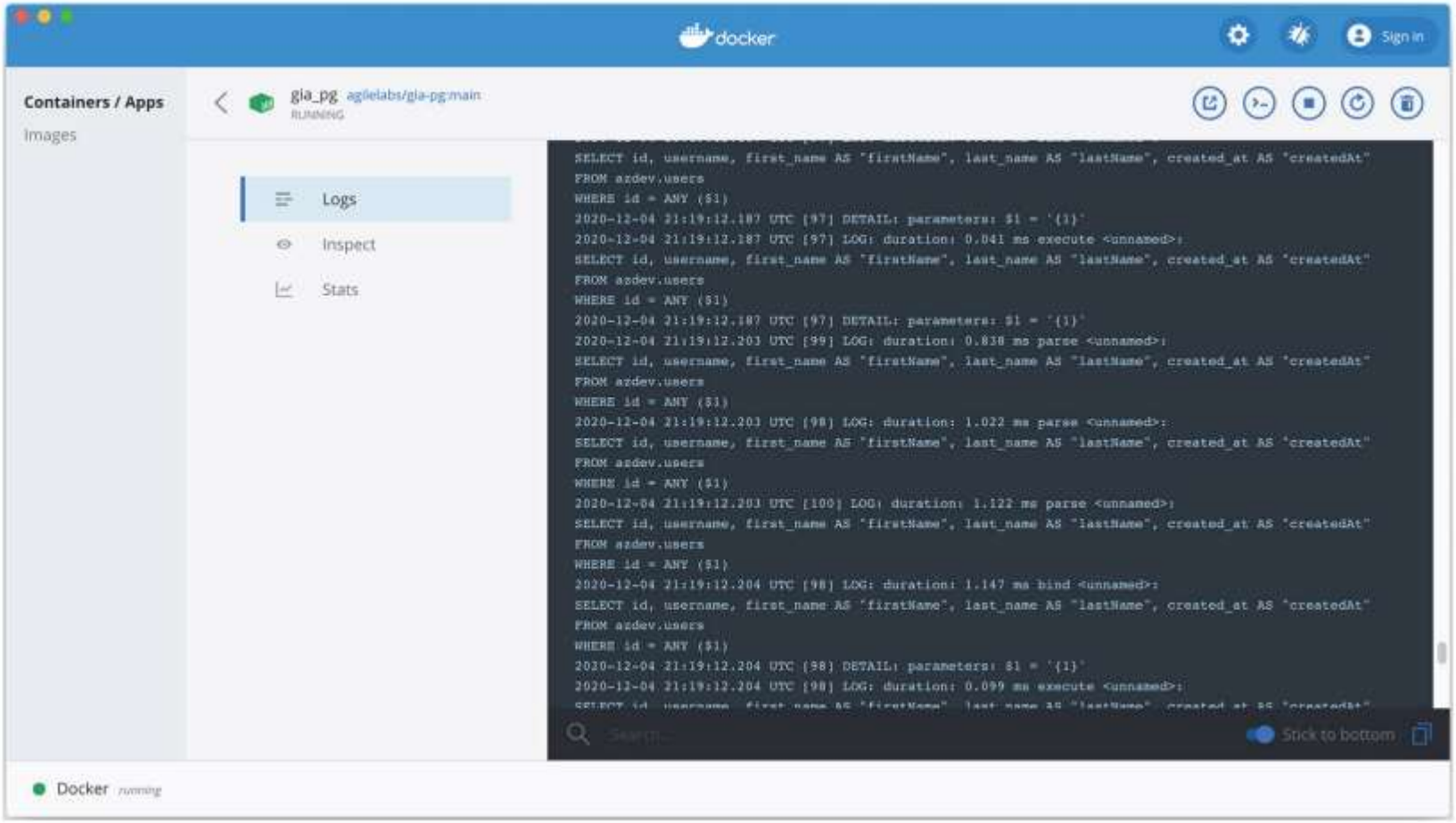
THE N+1 QUERIES PROBLEM

- Now that we have implemented a relation and made the GraphQL server execute multiple SQL statements for it, we can talk about the N+1 queries problem.
- It is the first big challenge when implementing GraphQL services.
- To see this problem in action, you'll need to enable logging for your PostgreSQL service and tail the logs while you execute GraphQL queries.

- Here are the SQL queries that were executed on my PostgreSQL server when I tested this.

```
LOG: statement:
SELECT ·-·-·
FROM azdev.tasks WHERE ·-·-·
LOG: execute <unnamed>:
SELECT ·-·-·
FROM azdev.users WHERE id = ANY ($1)
DETAIL:  parameters: $1 = '1'
LOG: execute <unnamed>:
SELECT ·-·-·
FROM azdev.users WHERE id = ANY ($1)
DETAIL:  parameters: $1 = '1'
LOG: execute <unnamed>:
SELECT ·-·-·
FROM azdev.users WHERE id = ANY ($1)
DETAIL:  parameters: $1 = '1'
LOG: execute <unnamed>:
SELECT ·-·-·
FROM azdev.users WHERE id = ANY ($1)
DETAIL:  parameters: $1 = '1'
LOG: execute <unnamed>:
SELECT ·-·-·
FROM azdev.users WHERE id = ANY ($1)
DETAIL:  parameters: $1 = '1'
```

"1" is the ID value for the user I used in the sample data.


docker
Sign in
Containers / Apps
Images
gia_pg agilelabs/gia-pg:main
RUNNING
Logs
Inspect
Stats
SELECT id, username, first_name AS "firstName", last_name AS "lastName", created_at AS "createdAt"
FROM azdev.users
WHERE id = ANY ($1)
2020-12-04 21:19:12.187 UTC [97] DETAIL: parameters: $1 = '{1}'
2020-12-04 21:19:12.187 UTC [97] LOG: duration: 0.041 ms execute <unnamed>:
SELECT id, username, first_name AS "firstName", last_name AS "lastName", created_at AS "createdAt"
FROM azdev.users
WHERE id = ANY ($1)
2020-12-04 21:19:12.187 UTC [97] DETAIL: parameters: $1 = '{1}'
2020-12-04 21:19:12.203 UTC [99] LOG: duration: 0.838 ms parse <unnamed>:
SELECT id, username, first_name AS "firstName", last_name AS "lastName", created_at AS "createdAt"
FROM azdev.users
WHERE id = ANY ($1)
2020-12-04 21:19:12.203 UTC [98] LOG: duration: 1.022 ms parse <unnamed>:
SELECT id, username, first_name AS "firstName", last_name AS "lastName", created_at AS "createdAt"
FROM azdev.users
WHERE id = ANY ($1)
2020-12-04 21:19:12.203 UTC [100] LOG: duration: 1.122 ms parse <unnamed>:
SELECT id, username, first_name AS "firstName", last_name AS "lastName", created_at AS "createdAt"
FROM azdev.users
WHERE id = ANY ($1)
2020-12-04 21:19:12.204 UTC [98] LOG: duration: 1.147 ms bind <unnamed>:
SELECT id, username, first_name AS "firstName", last_name AS "lastName", created_at AS "createdAt"
FROM azdev.users
WHERE id = ANY ($1)
2020-12-04 21:19:12.204 UTC [98] DETAIL: parameters: $1 = '{1}'
2020-12-04 21:19:12.204 UTC [98] LOG: duration: 0.099 ms execute <unnamed>:
Search...
Stick to bottom
Docker running

THE N+1 QUERIES PROBLEM

```
const views = {
  tasksAndUsers: `
    SELECT t.*,
        u.id AS "author_id",
        u.username AS "author_username",
        u.first_name AS "author_firstName",
        u.last_name AS "author_lastName",
        u.created_at AS "author_createdAt"
    FROM azdev.tasks t
    JOIN azdev.users u ON (t.user_id = u.id)
  `,
};
// ----
```

```
/ # psql azdev postgres
psql (12.2)
Type "help" for help.

azdev=#     SELECT t.*,
azdev-#         u.id AS "author_id",
azdev-#         u.username AS "author_username",
azdev-#         u.first_name AS "author_firstName",
azdev-#         u.last_name AS "author_lastName",
azdev-#         u.created_at AS "author_createdAt"
azdev-#     FROM azdev.tasks t
azdev-#     JOIN azdev.users u ON (t.user_id = u.id);
 id |                                content
thor_id | author_username | author_firstName | author_lastName |
----+-------------------------------------------------------------
--------+-----------------+------------------+-----------------+-
  1 | Make an image in HTML change based on the theme color mode
     1 | test            |                  |                 |
  2 | Get rid of only the unstaged changes since the last git com
     1 | test            |                  |                 |
  3 | The syntax for a switch statement (AKA case statement) in :
```

THE N+1 QUERIES PROBLEM

- To use the tasksAndUsers view, instead of selecting from the azdev.tasks table for the sqls.tasksLatest SQL query, we can select from the new tasksAndUsers view.

```
taskMainList: `
  SELECT id, content, tags, ----
    "author_id", "author_username", "author_firstName",
    "author_lastName", "author_createdAt"
  FROM (${views.tasksAndUsers})
  WHERE is_private = FALSE
  ORDER BY created_at DESC
  LIMIT 100
`,
```

THE N+1 QUERIES PROBLEM

```
// -.-.-
import { extractPrefixedColumns } from '../../utils';

const Task = new GraphQLObjectType({
  name: 'Task',
  fields: {
    // -.-.-

    author: {
      type: new GraphQLNonNull(User),
      resolve: prefixedObject =>
        extractPrefixedColumns({ prefixedObject, prefix: 'author' }),
    },
  },
});
```

```
export const extractPrefixedColumns = ({
  prefixedObject,
  prefix,
}) => {
  const prefixRexp = new RegExp(`^${prefix}_(.*)`);
  return Object.entries(prefixedObject).reduce(
    (acc, [key, value]) => {
      const match = key.match(prefixRexp);
      if (match) {
        acc[match[1]] = value;
      }
      return acc;
    },
    {},
  );
};
```

match[1] will be the prefixed column name without the prefix part.

THE N+1 QUERIES PROBLEM

```
LOG: statement:
SELECT ·-·-·
    FROM (
    SELECT ·-·-·
    FROM azdev.tasks t ,
    JOIN azdev.users u ON (t.user_id = u.id)
) tau WHERE ·-·-·
```

Resolving a one-to-many relation

```
// ·-·-·
import Approach from './approach';

const Task = new GraphQLObjectType({
  name: 'Task',
  fields: {
    // ·-·-·
    approachList: {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(Approach))
        ),
        resolve: (source, args, { pgApi }) =>
          pgApi.approachList(source.id),
      },
    },
  });
```

Approach is the new GraphQL type we need to introduce.

pgApi.approachList receives the ID of a Task object (source.id) and should return a list of Approach objects.

Resolving a one-to-many relation

- Let's implement the Approach type next, This is the schema-language text we have for it.

```
type Approach implement SearchResultItem {
  id: ID!
  createdAt: String!
  content: String!
  voteCount: Int!
  author: User!
  task: Task!
  detailList: [ApproachDetail!]!
}
```

We'll implement the task and detailList fields

```
import {
  GraphQLID,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLNonNull,
} from 'graphql';

import User from './user';

const Approach = new GraphQLObjectType({
  name: 'Approach',
  fields: {
    id: { type: new GraphQLNonNull(GraphQLID) },
    content: { type: new GraphQLNonNull(GraphQLString) },
    voteCount: { type: new GraphQLNonNull(GraphQLInt) },
    createdAt: {
      type: new GraphQLNonNull(GraphQLString),
      resolve: ({ createdAt }) => createdAt.toISOString(),
    },
    author: {
      type: new GraphQLNonNull(User),

      resolve: (source, args, { pgApi }) =>
        pgApi.userInfo(source.userId),
    },
  },
});

export default Approach;
```

Resolving a one-to-many relation

```
tasksApproachLists: `
  SELECT id, content, user_id AS "userId", task_id AS "taskId",
         vote_count AS "voteCount", created_at AS "createdAt"
  FROM azdev.approaches
  WHERE task_id = ANY ($1)
  ORDER BY vote_count DESC, created_at DESC
`,
```

The columns are aliased as camel-case.

This statement needs a Task ID value to be passed as $1.

Sorts Approaches by their vote count (and then timestamp, if many records have the same vote count)

Resolving a one-to-many relation

- The sqls.approachesForTaskIds statement will be used by the pgApi.approachList function that we will implement next.

```
const pgApiWrapper = async () => {
  // ...-...

  return {
    // ...-...
    approachList: async (taskId) => {
      const pgResp = await pgQuery(sqls.approachesForTaskIds, {
        $1: [taskId],
      });
      return pgResp.rows;
    },
  };
};
```

Passes $1 to the SQL statement as [taskId]

```
{
  taskMainList {
    id
    content
    tags
    approachCount
    createdAt

    author {
      id
      username
      name
    }

    approachList {
      id
      content
      voteCount
      createdAt

      author {
        id
        username
        name
      }
    }
```

QUERY VARIABLES


```
{
  "data": {
    "taskMainList": [
      {
        "id": "1",
        "content": "Make an image in HTML change based on the theme
color mode (dark or light)",
        "tags": [
          "code",
          "html"
        ],
        "approachCount": 1,
        "createdAt": "2020-08-01T00:03:02.032Z",
        "author": {…},
        "approachList": [
          {
            "id": "1",
            "content": "<picture>\n  <source\n    srcset=\"settings-
dark.png\"\n    media=\"(prefers-color-scheme: dark)\"\n  />\n
<source\n    srcset=\"settings-light.png\"\n    media=\"(prefers-
color-scheme: light), (prefers-color-scheme: no-preference)\"\n  />\n
<img src=\"settings-light.png\" loading=\"lazy\" />\n</picture>",
            "voteCount": 0,
            "createdAt": "2020-08-01T00:03:02.035Z",
            "author": {…}
          }
        ]
```

Summary

- Use realistic, production-like data in development to make your manual tests relevant and useful.
- Start with the simplest implementations you can think of. Make things work, and then improve on your implementations.
- You can use the GraphQL context object to make a pool of database connections available to all resolver functions.

"Complete Lab"