Working with database models and relations

LEARNING VOYAGE

Table of Contents

- **Running and connecting to databases**
- The taskMainList query
- Error reporting
- Resolving relations

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

- If the database servers run successfully, you should have six Tasks with their Approaches and some extra dynamic data elements in MongoDB for each Approach.
- Use the following SQL queries to see the data in PostgreSQL.

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

Running and connecting to databases

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

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

Running and connecting to databases

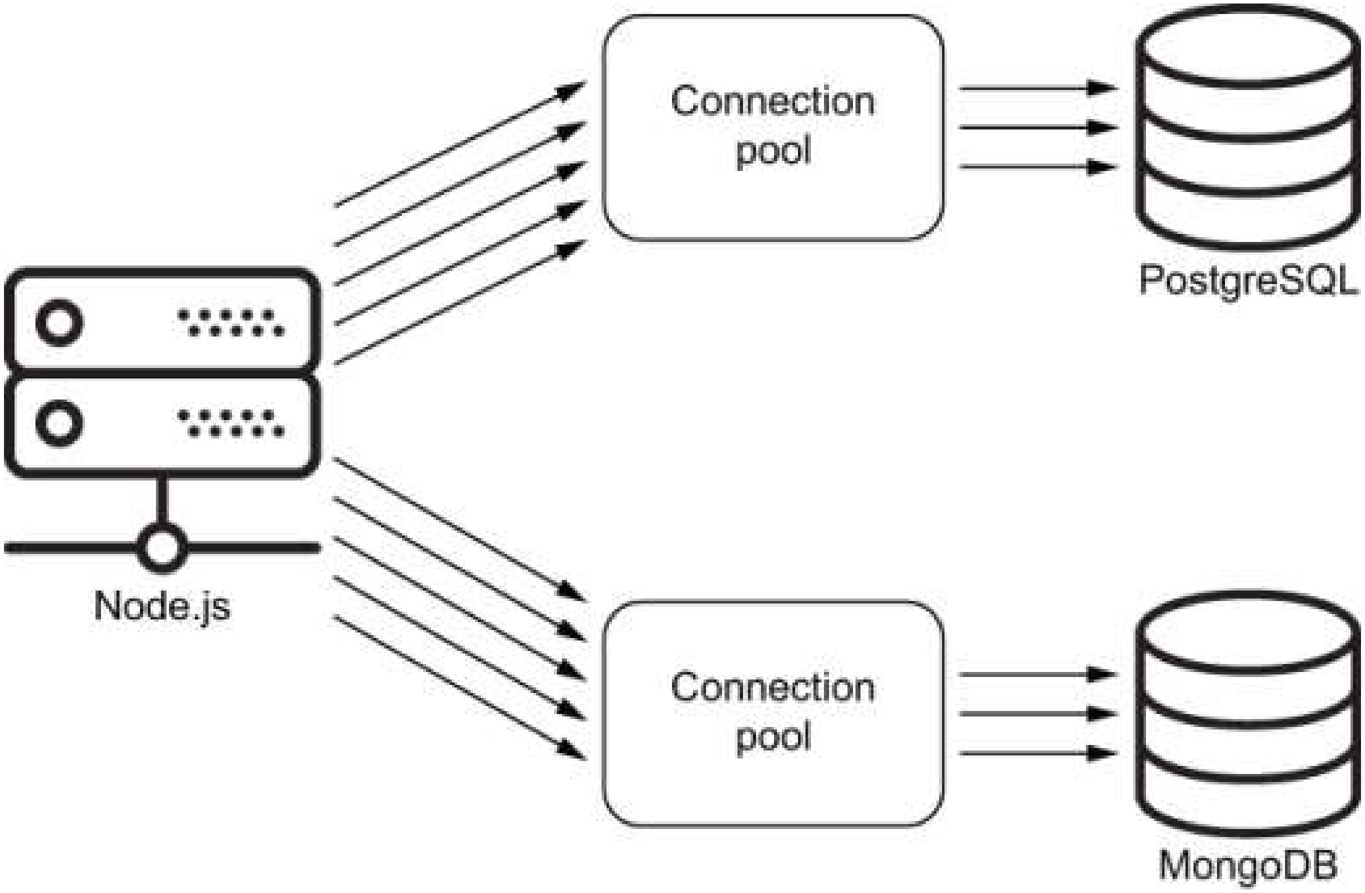


Table of Contents

- Running and connecting to databases
- **The taskMainList query**
- Error reporting
- Resolving relations

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.

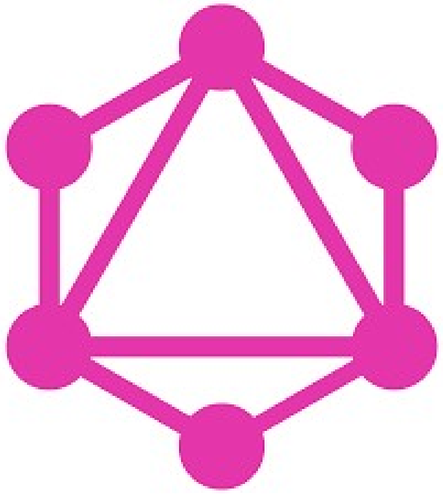
- We named that field taskMainList.

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

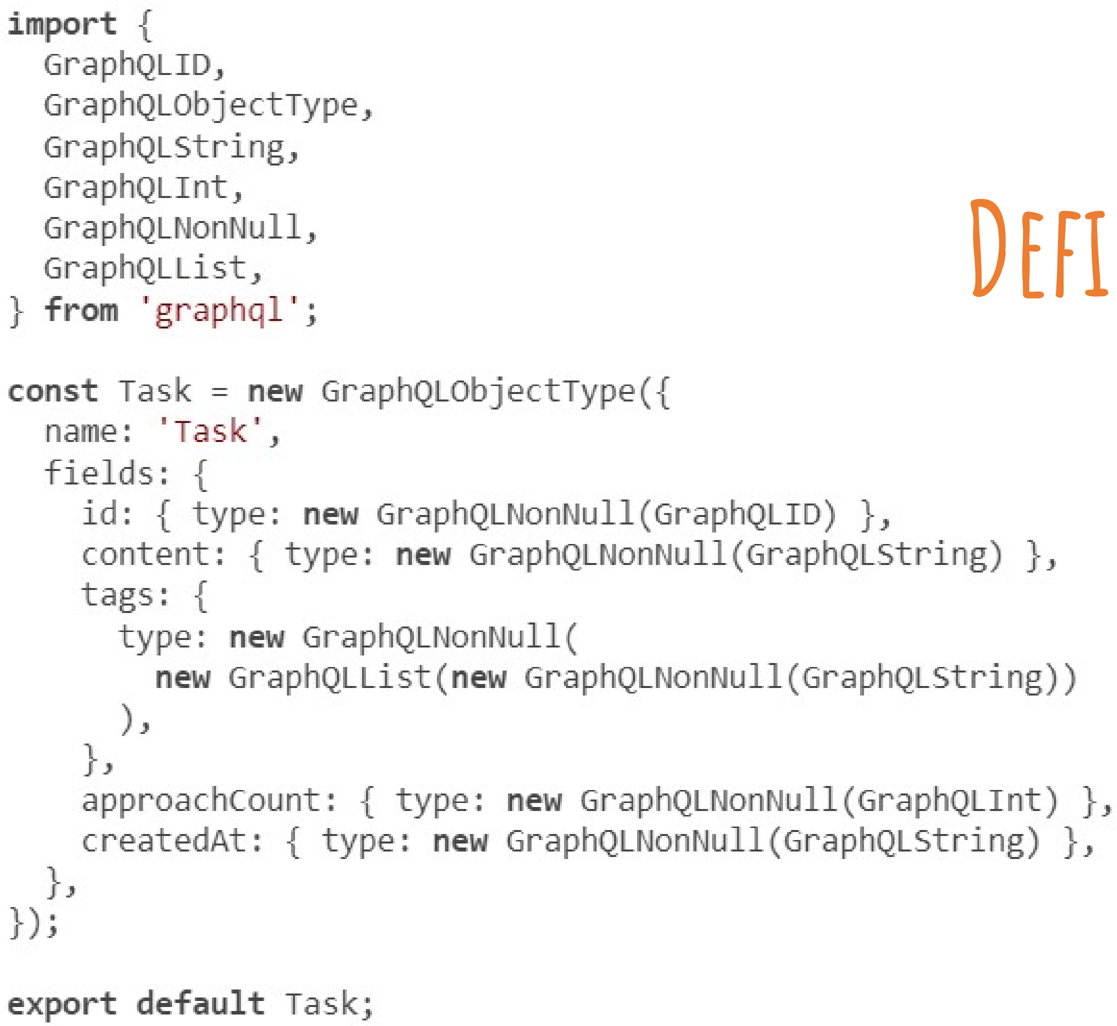
The taskMainList query

- A GraphQL query that we can use to start testing this feature.

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



Defining object types



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

const Task = new GraphQLObjectType({
  name: 'Task',
  fields: {
    id: { type: new GraphQLNonNull(GraphQLID) },
    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 simple [String!]! had to be written with nested calls of three functions:

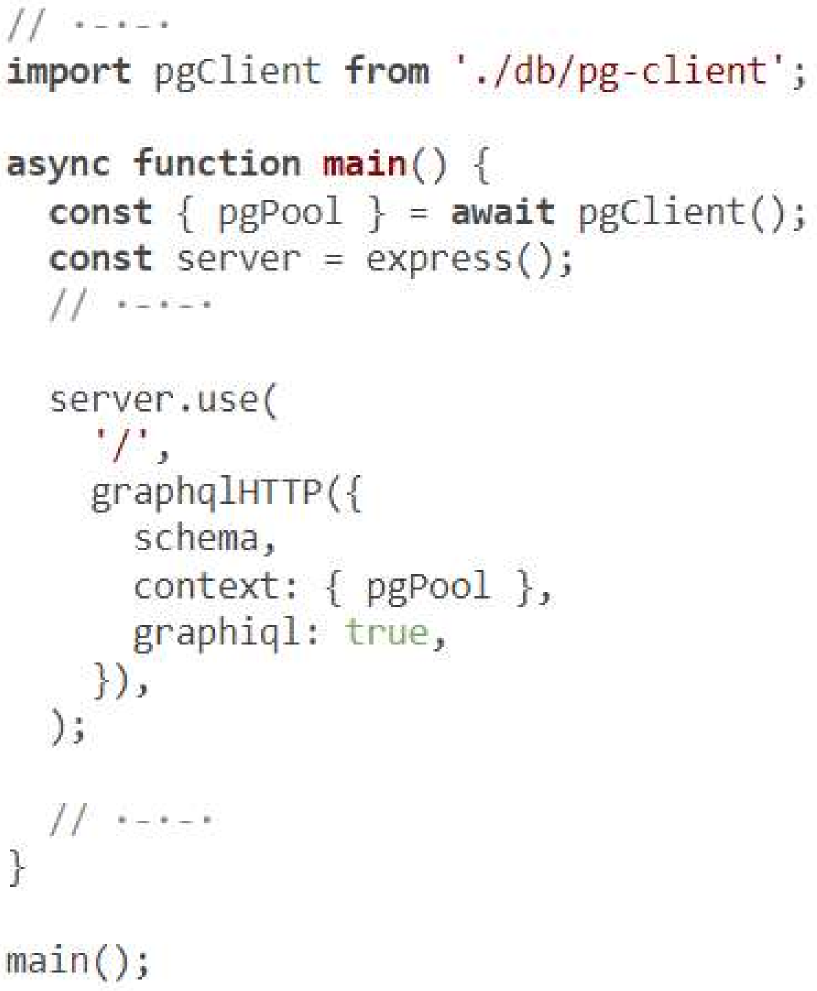
```
new GraphQLNonNull(
  new GraphQLList(
    new GraphQLNonNull(
      GraphQLString
    )
  )
)
```

The context object

- We need to execute this SQL statement on the PostgreSQL database to resolve field.

```
SELECT *
FROM azdev.tasks
WHERE is_private = FALSE
ORDER BY created_at DESC
LIMIT 100
```

The context object

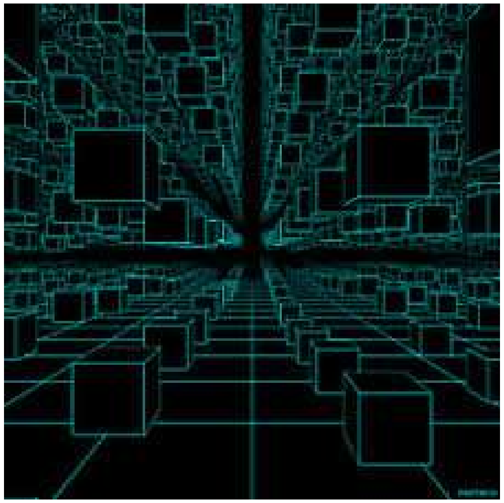


```
// ·-·-·
import pgClient from './db/pg-client';

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.

```
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.

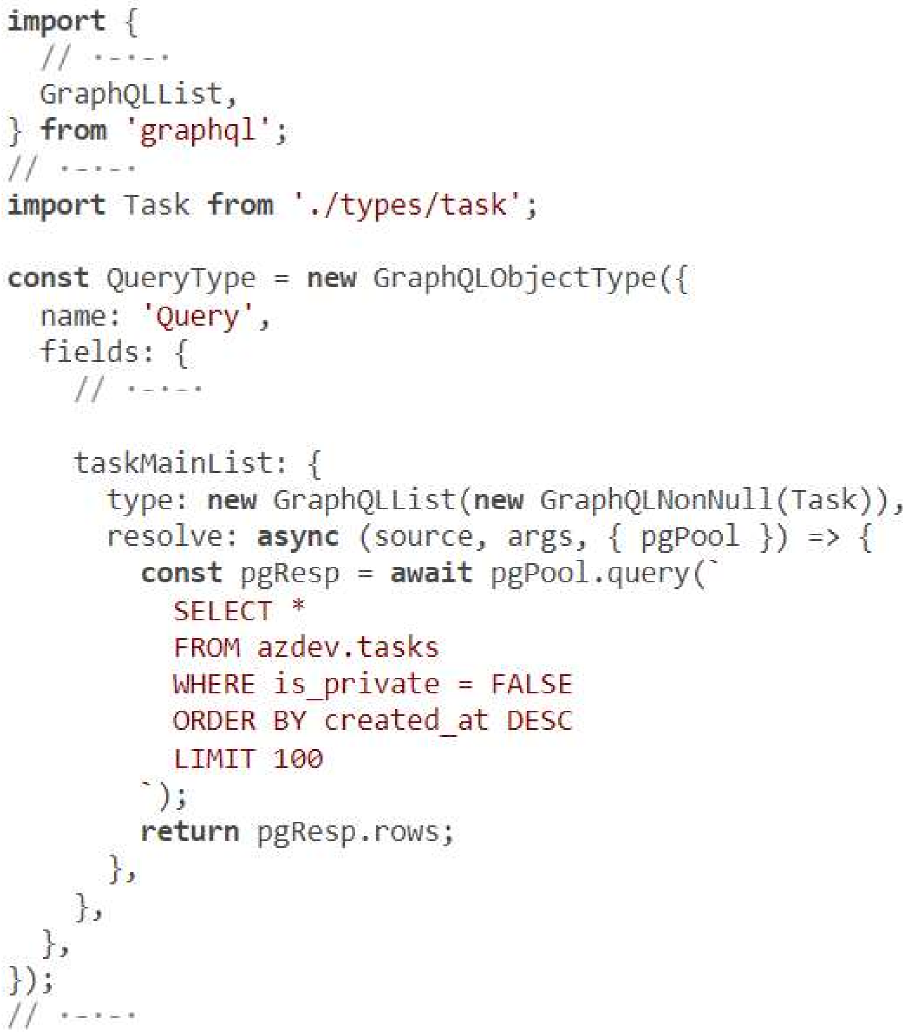
```
[
  { id: 1, content: 'Task #1', approach_count: 1,
•−•−•},
  { id: 2, content: 'Task #2', approach_count: 1,
•−•−•},
  •−•−•
]
```

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) => {}
```

The context object



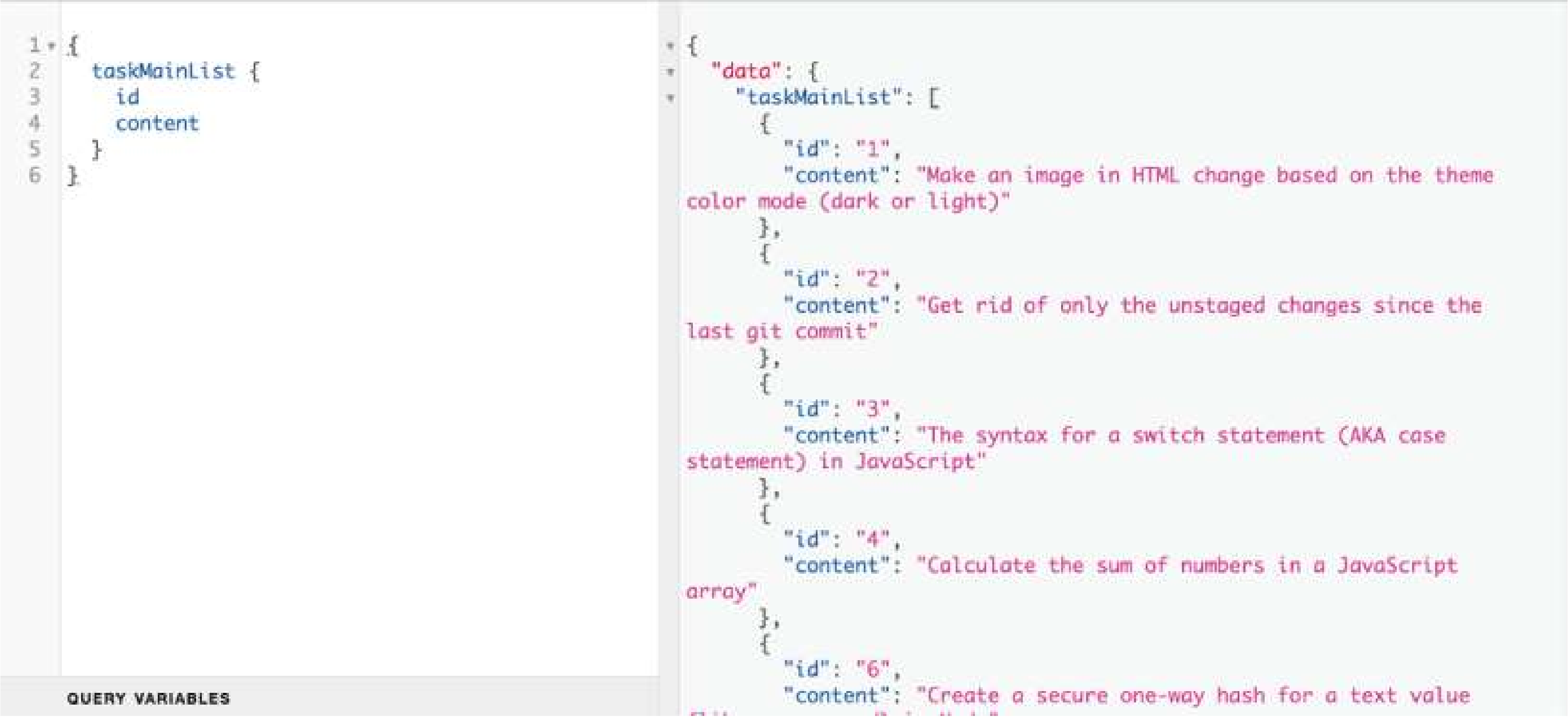
```
import {
  // ·-·-·
  GraphQLList,
} from 'graphql';
// ·-·-·
import Task from './types/task';

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

- Lets test things now. The API should be able to answer this query (see next slide):

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

The context object



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

```
{
  "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.

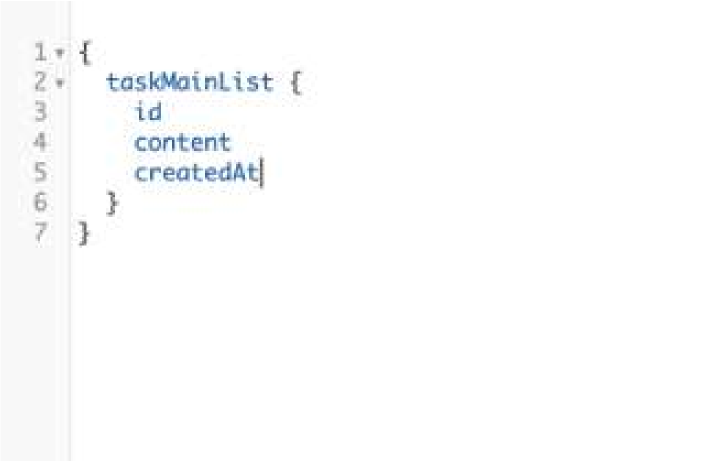
Method #1

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

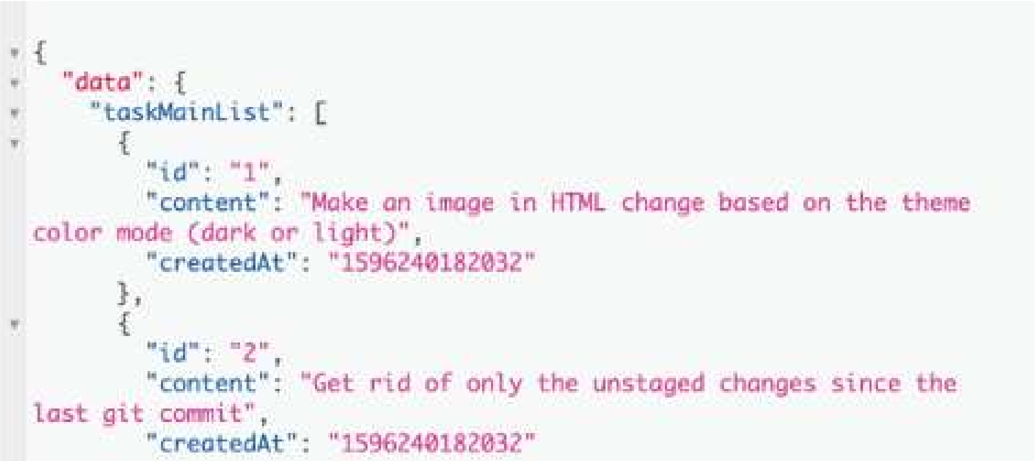
Method #2

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

Method #2



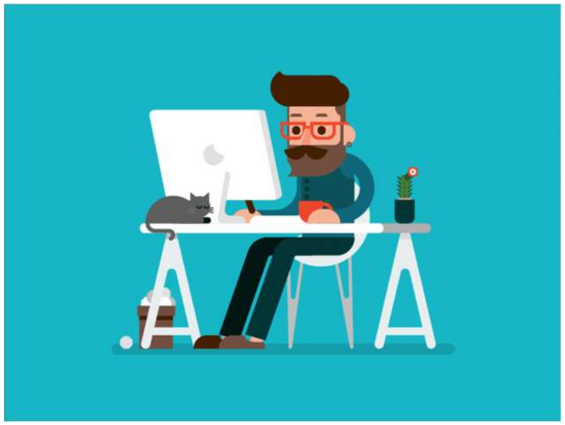
```
{
  taskMainList {
    id
    content
    createdAt
  }
}
```



```
{
  "data": {
    "taskMainList": [
      {
        "id": "1",
        "content": "Make an image in HTML change based on the theme
color mode (dark or light)",
        "createdAt": "1596240182032"
      },
      {
        "id": "2",
        "content": "Get rid of only the unstaged changes since the
last git commit",
        "createdAt": "1596240182032"
```

Method #3

```
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;
},
```



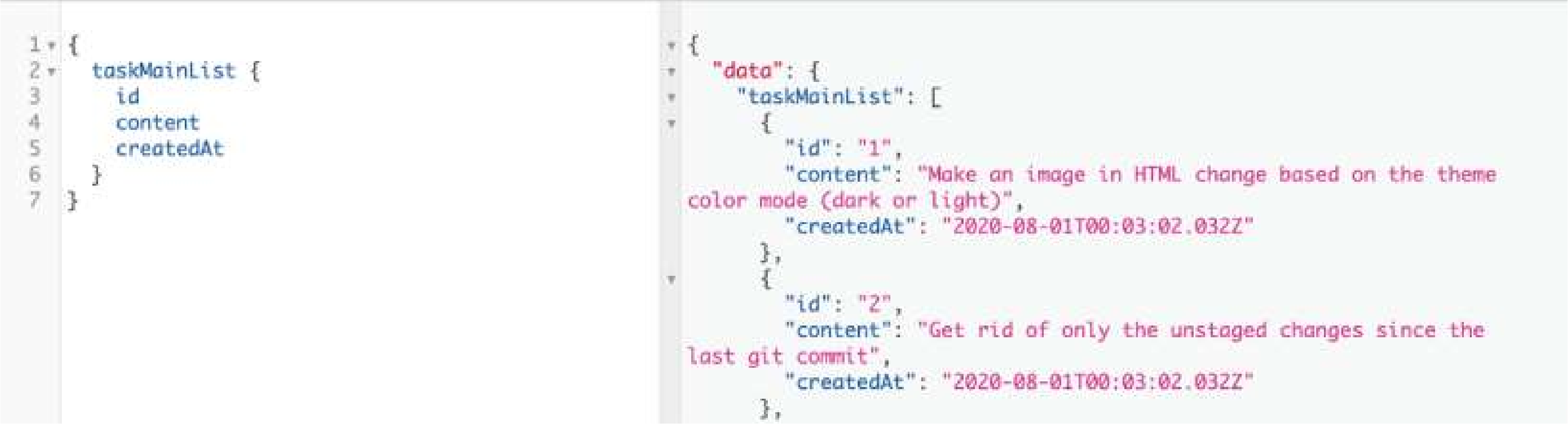
Transforming field values

- 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

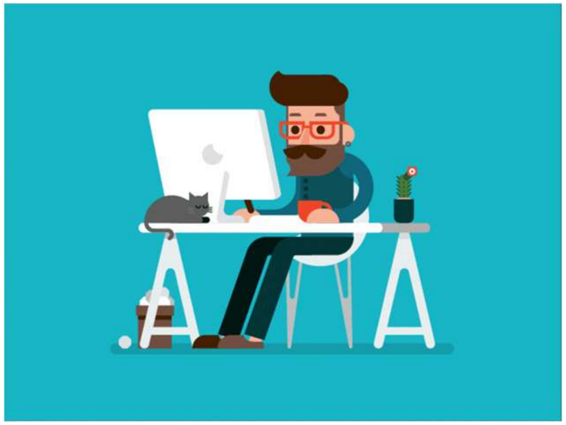
- Now the API displays values of createdAt using the ISO format



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

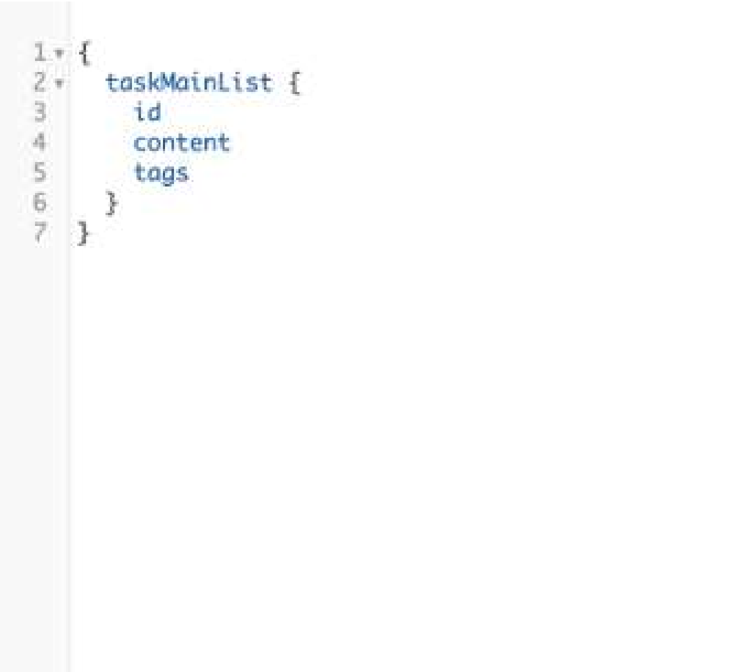
```
{
  "data": {
    "taskMainList": [
      {
        "id": "1",
        "content": "Make an image in HTML change based on the theme
color mode (dark or light)",
        "createdAt": "2020-08-01T00:03:02.032Z"
      },
      {
        "id": "2",
        "content": "Get rid of only the unstaged changes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

Transforming field values

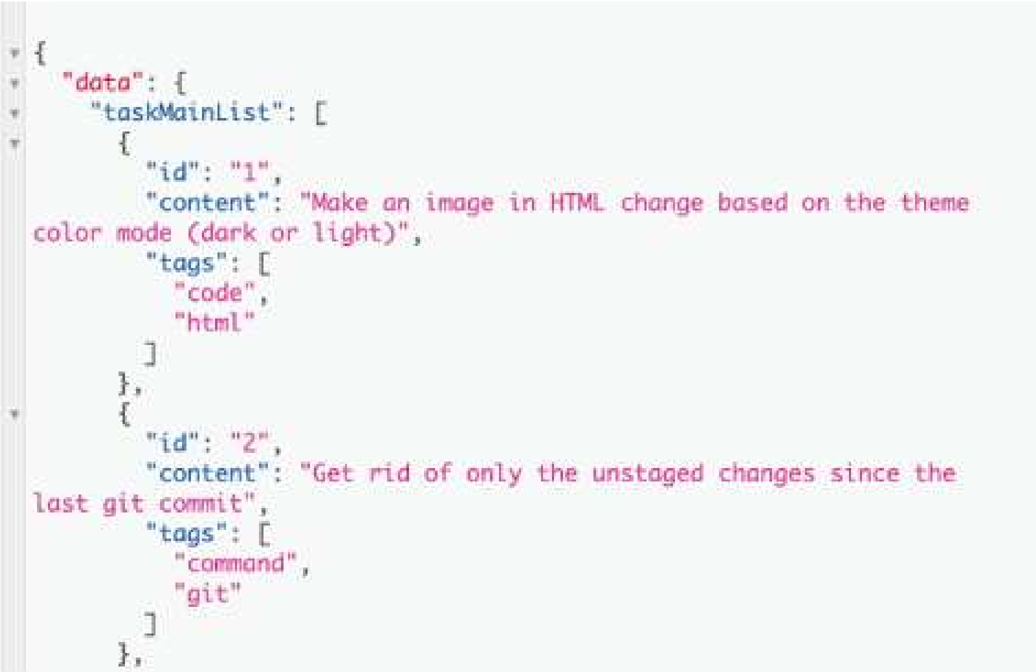


```
tags: {
  type: new GraphQLNonNull(
    new GraphQLList(new
GraphQLNonNull(GraphQLString))
  ),
  resolve: (source) => source.tags.split(','),
},
```

Transforming field values

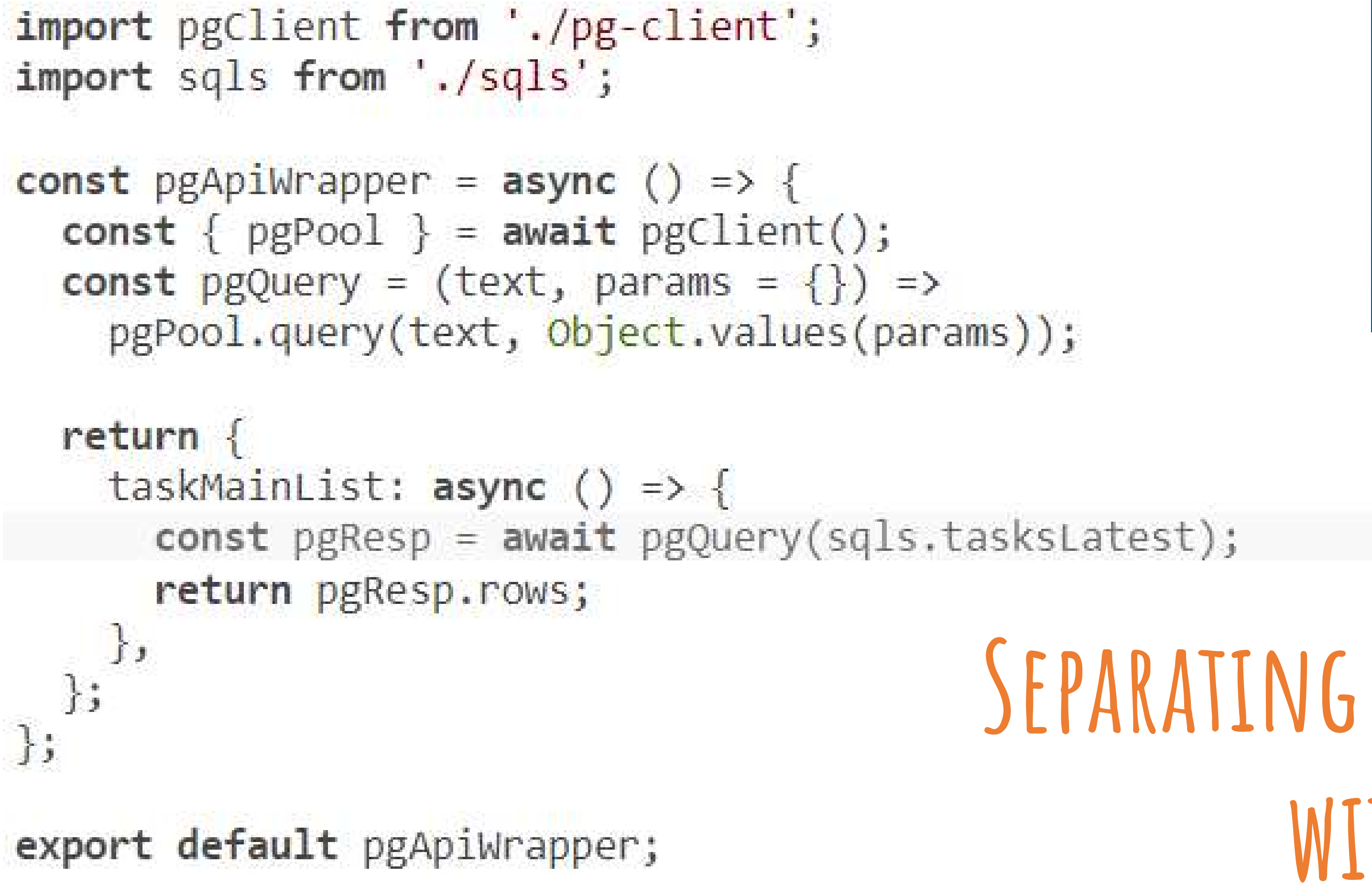


```
{
  taskMainList {
    id
    content
    tags
  }
}
```



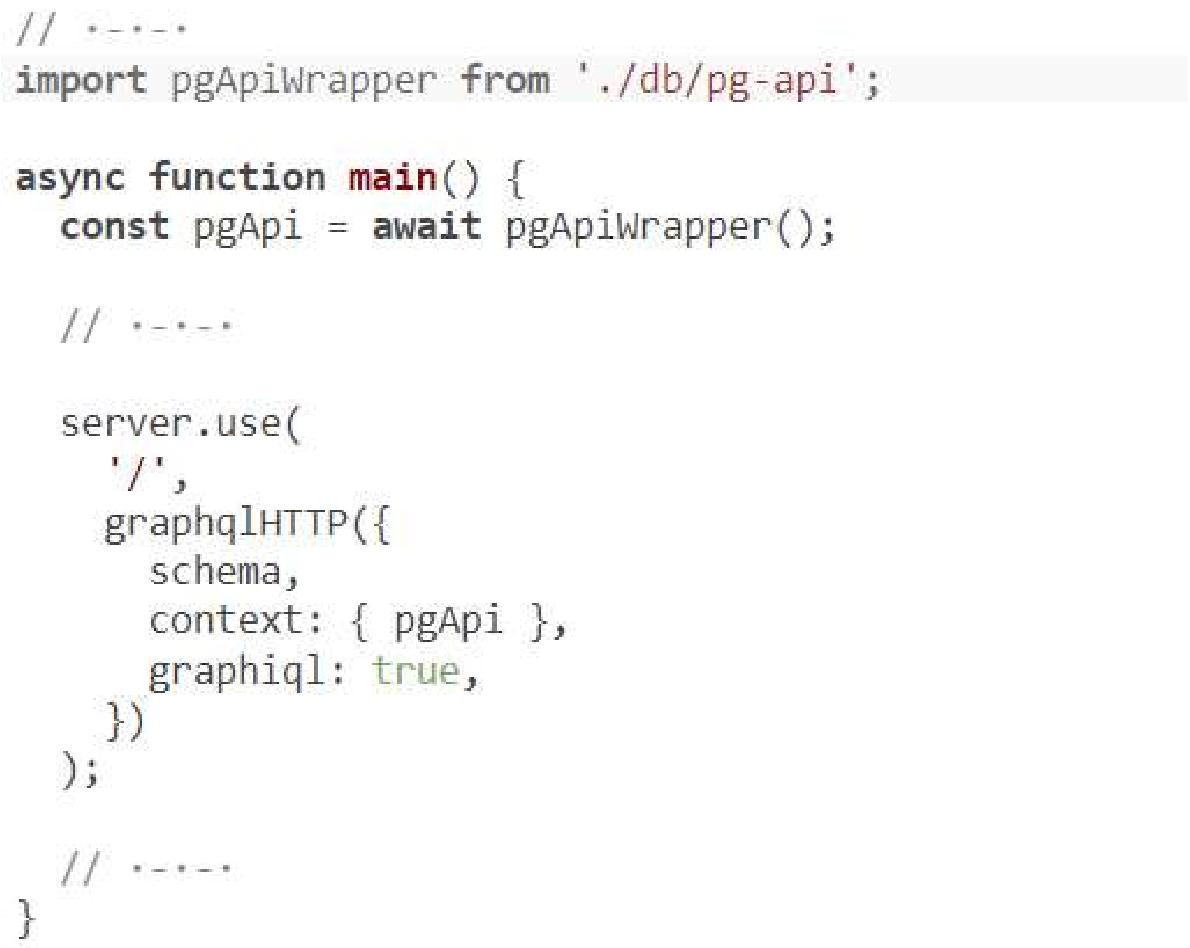
```
{
  "data": {
    "taskMainList": [
      {
        "id": "1",
        "content": "Make an image in HTML change based on the theme
color mode (dark or light)",
        "tags": [
          "code",
          "html"
        ]
      },
      {
        "id": "2",
        "content": "Get rid of only the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

Separating interactions with PostgreSQL



```
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;
```

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,
    })
  );

  // ·-·-·
}
```

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();
  },
},
```

Table of Contents

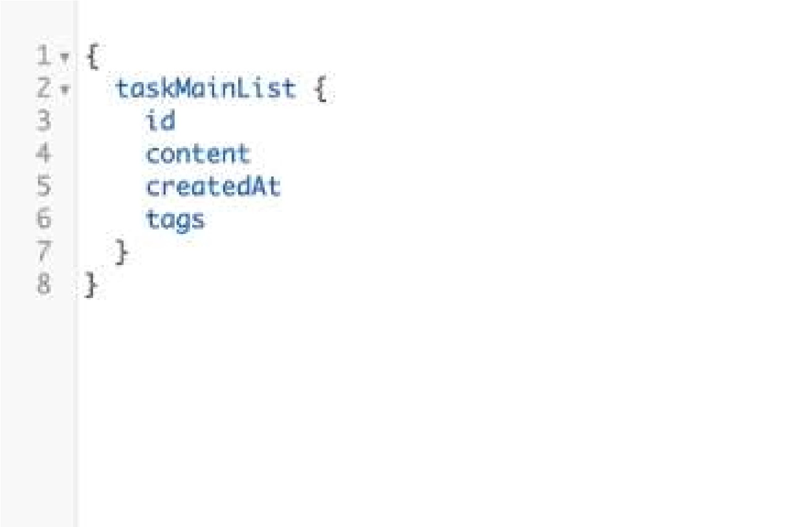
- Running and connecting to databases
- The taskMainList query
- **Error reporting**
- Resolving relations

Error reporting

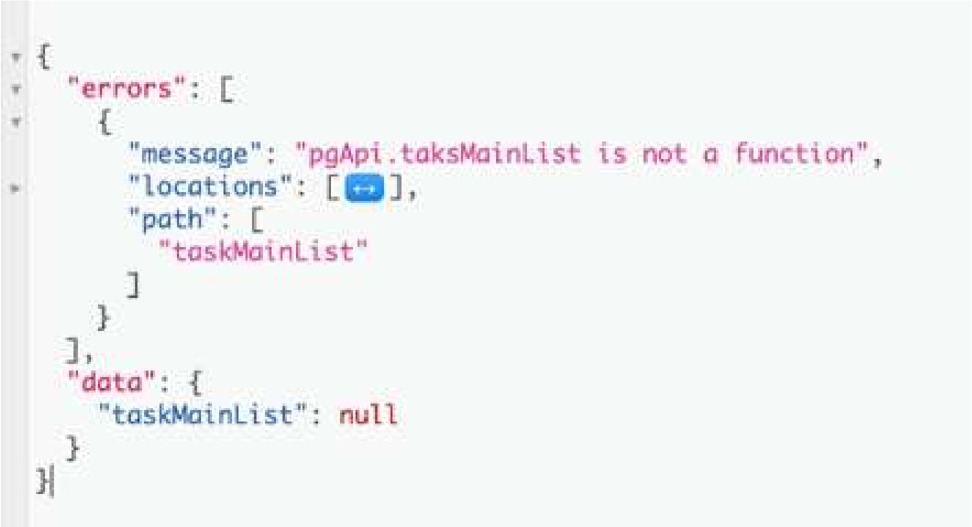
```
const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // ·-·-·
    taskMainList: {
      type: new GraphQLList(new GraphQLNonNull(Task)),
      resolve: async (source, args, { pgApi }) => {
        return pgApi.taksMainList();
      },
    },
  },
});
```

Error reporting

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

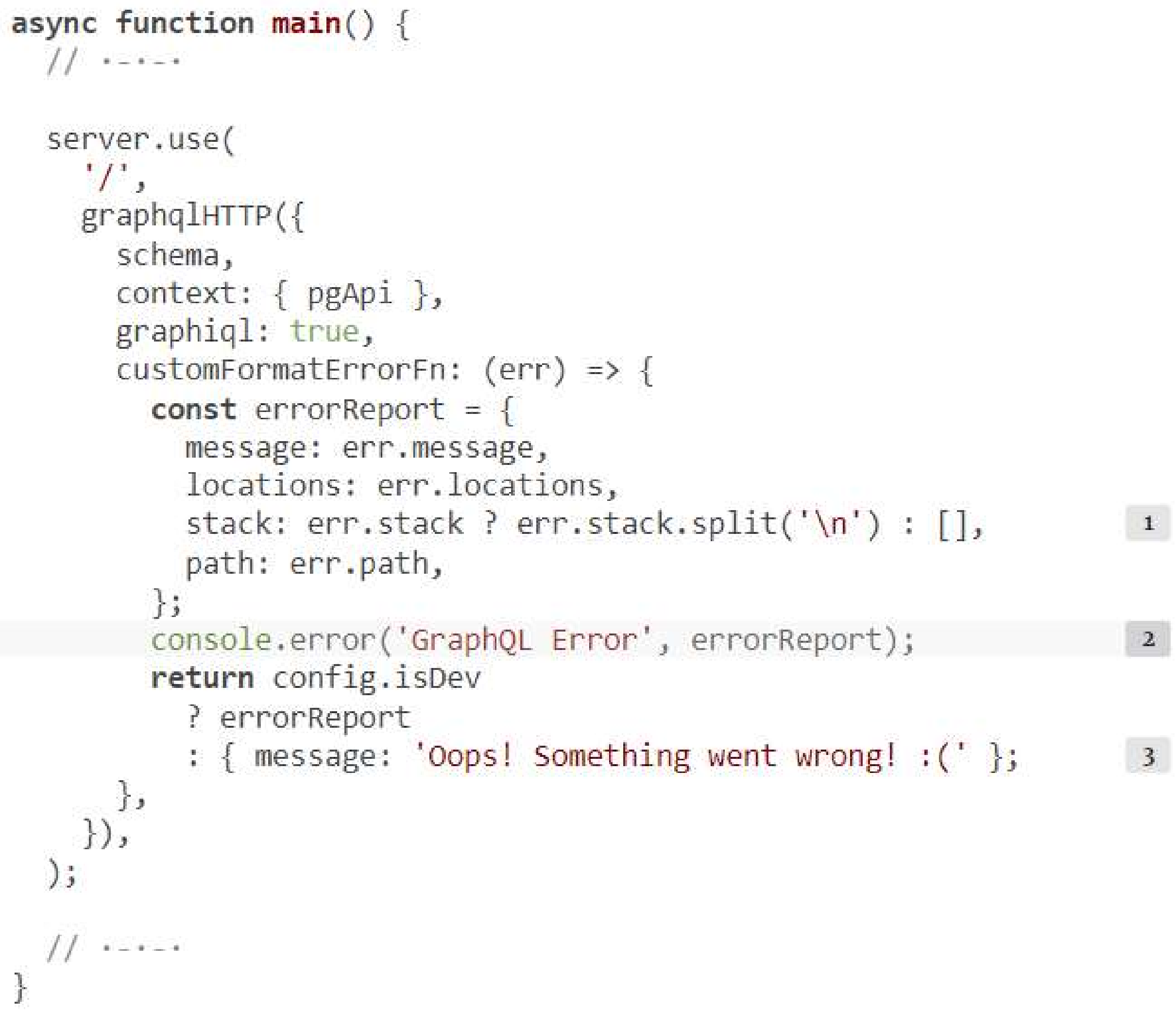


```
{
  taskMainList {
    id
    content
    createdAt
    tags
  }
}
```



```
{
  "errors": [
    {
      "message": "pgApi.taksMainList is not a function",
      "locations": [↔],
      "path": [
        "taskMainList"
      ]
    }
  ],
  "data": {
    "taskMainList": null
  }
}
```

Error reporting



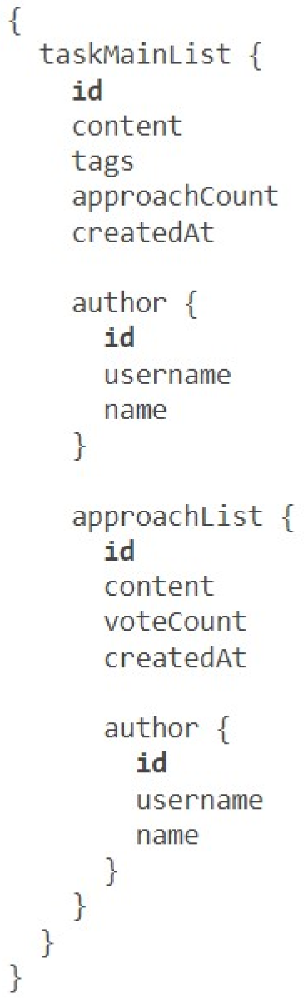
```
async function main() {
  // ·-·-·

  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') : [],        1
          path: err.path,
        };
        console.error('GraphQL Error', errorReport);            2
        return config.isDev
          ? errorReport
          : { message: 'Oops! Something went wrong! :(' };      3
      },
    }),
  );

  // ·-·-·
}
```

Table of Contents

- Running and connecting to databases
- The taskMainList query
- Error reporting
- **Resolving relations**

Resolving relations

- When we’re done implementing the author and approachList fields, the API server should accept and reply to this query.



```
{
  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

```
// $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

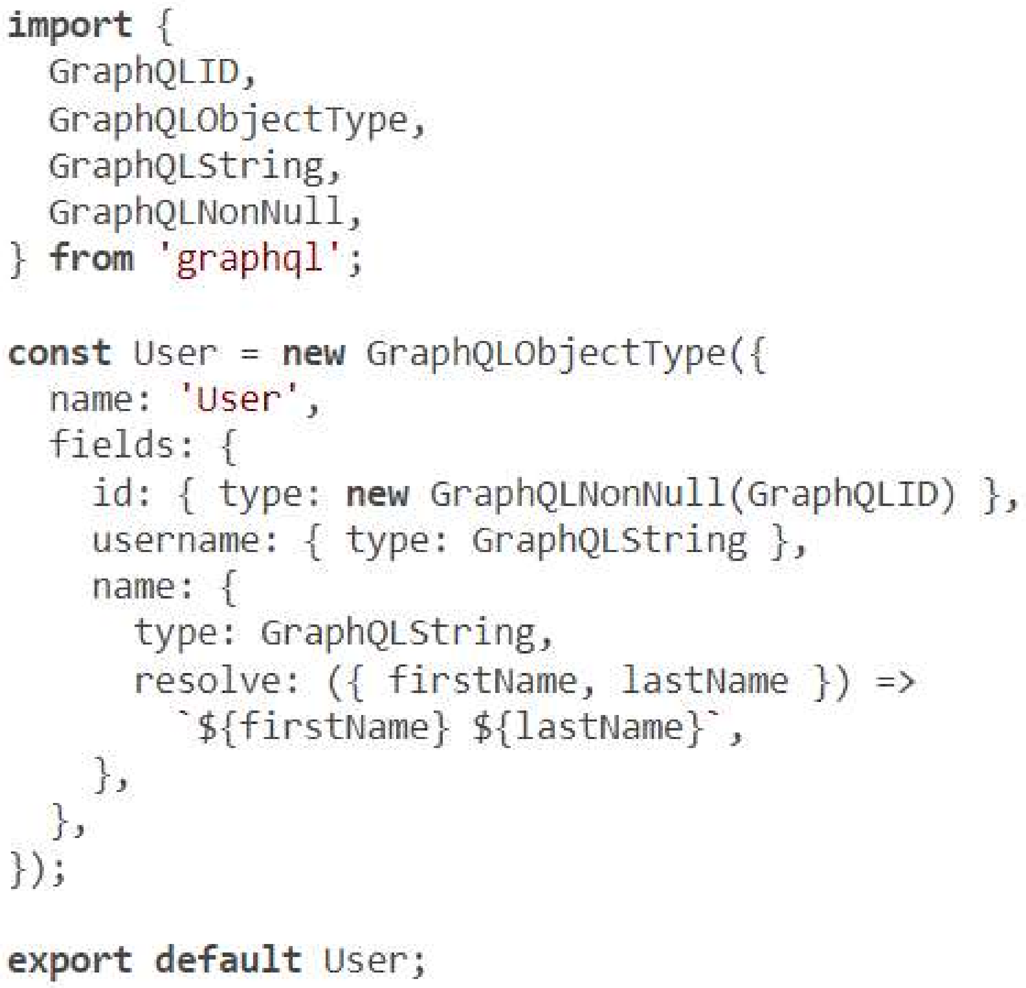
```
const pgApiWrapper = async () => {
  // ·-·-·
  return {
    // ·-·-·
    userInfo: async (userId) => {
      const pgResp = await
pgQuery(sqls.usersFromIds, { $1: [userId] });
      return pgResp.rows[0];
    },
  };
};
```

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!]!
}
```

Resolving a one-to-one relation



```
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

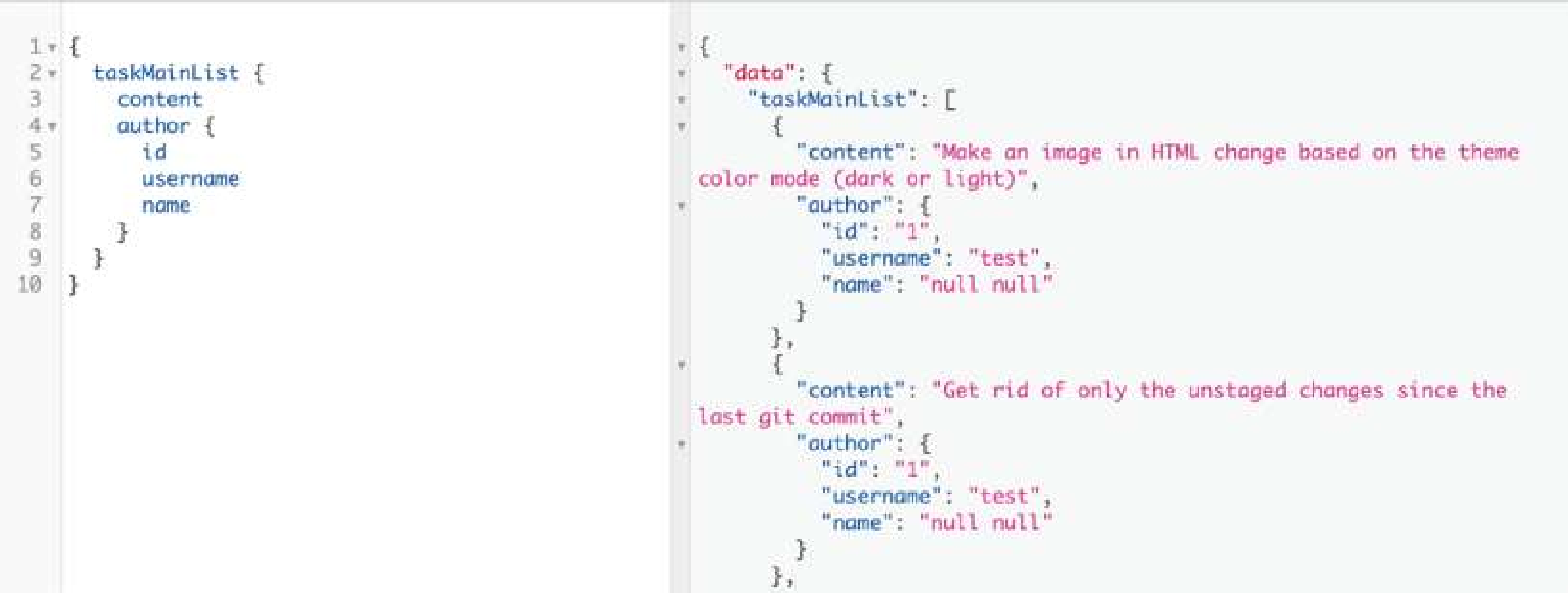
```
import User from './user';
const Task = new GraphQLObjectType({
  name: 'Task',
  fields: {
    // ·-·-·

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

Resolving a one-to-one relation

- You can test the new relation with this query.

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

Resolving a one-to-one relation



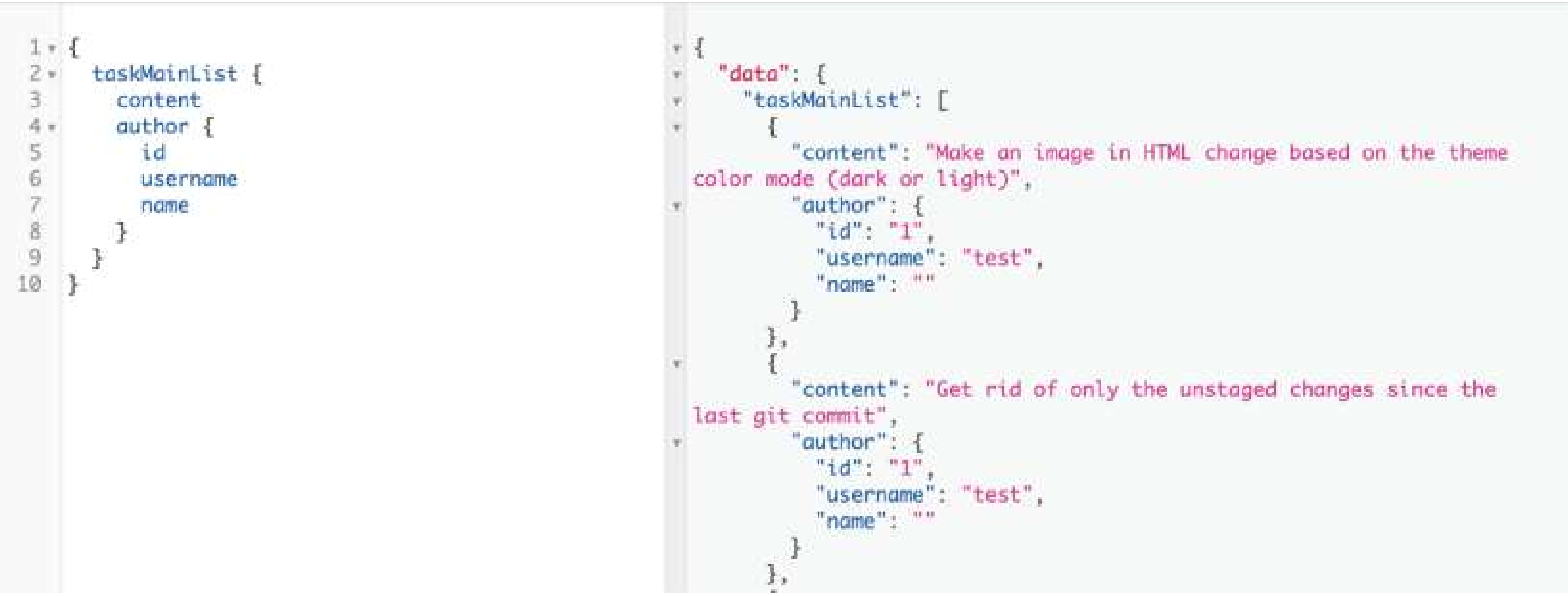
```
{
  taskMainList {
    content
    author {
      id
      username
      name
    }
  }
}
```

```
{
  "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"
        }
      },
```

Resolving a one-to-one relation

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

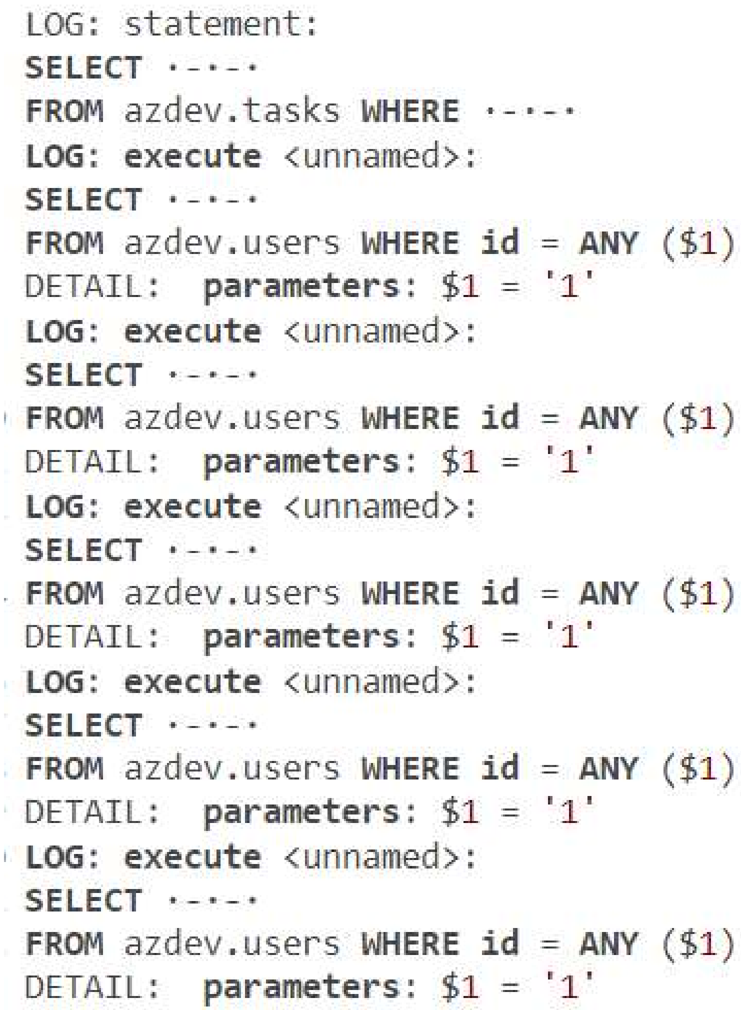
Resolving a one-to-one relation



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

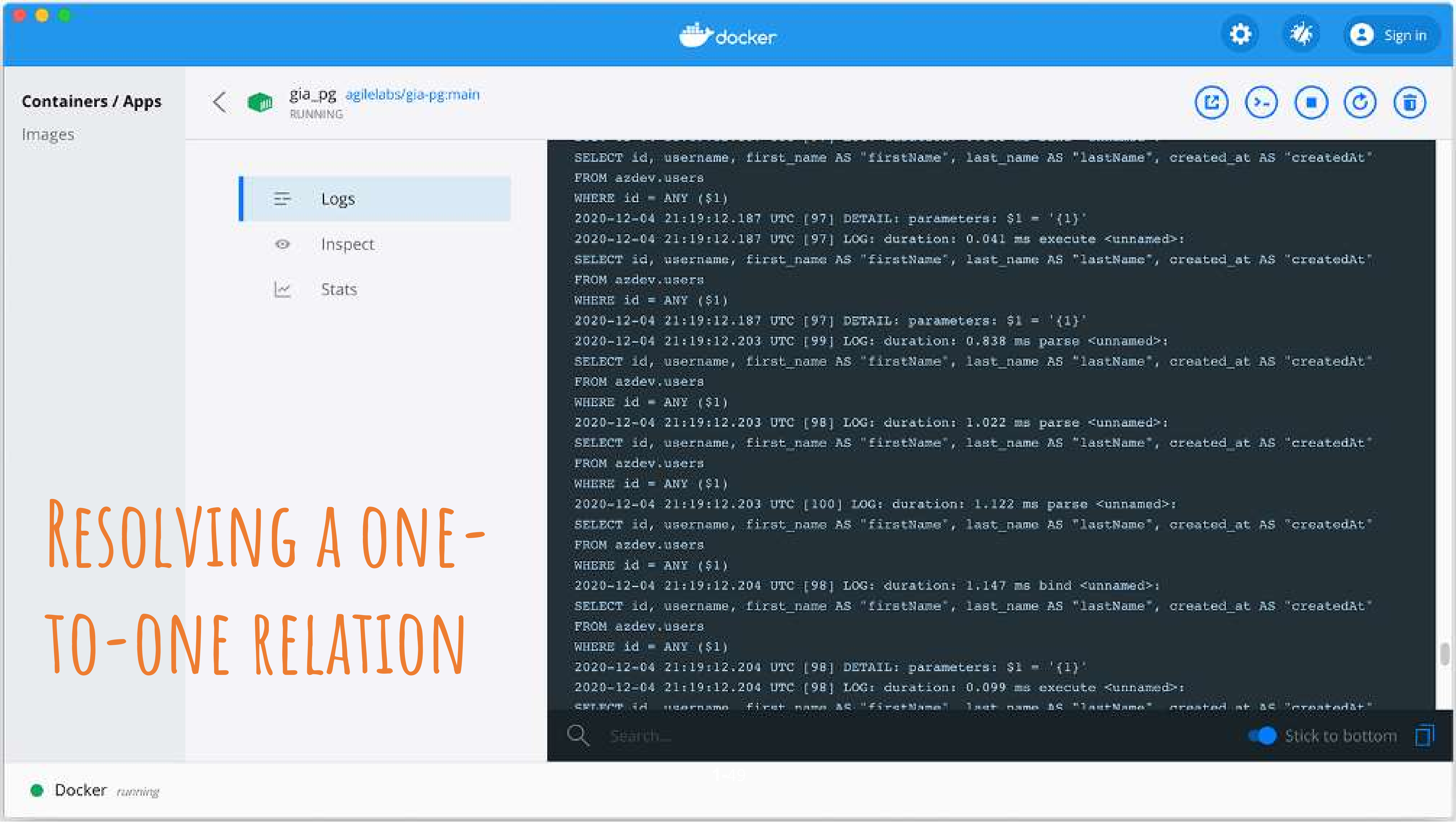
```
{
  "data": {
    "taskMainList": [
      {
        "content": "Make an image in HTML change based on the theme
color mode (dark or light)",
        "author": {
          "id": "1",
          "username": "test",
          "name": ""
        }
      },
      {
        "content": "Get rid of only the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

Resolving a one-to-one relation



```
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'
```

Resolving a one-to-one relation

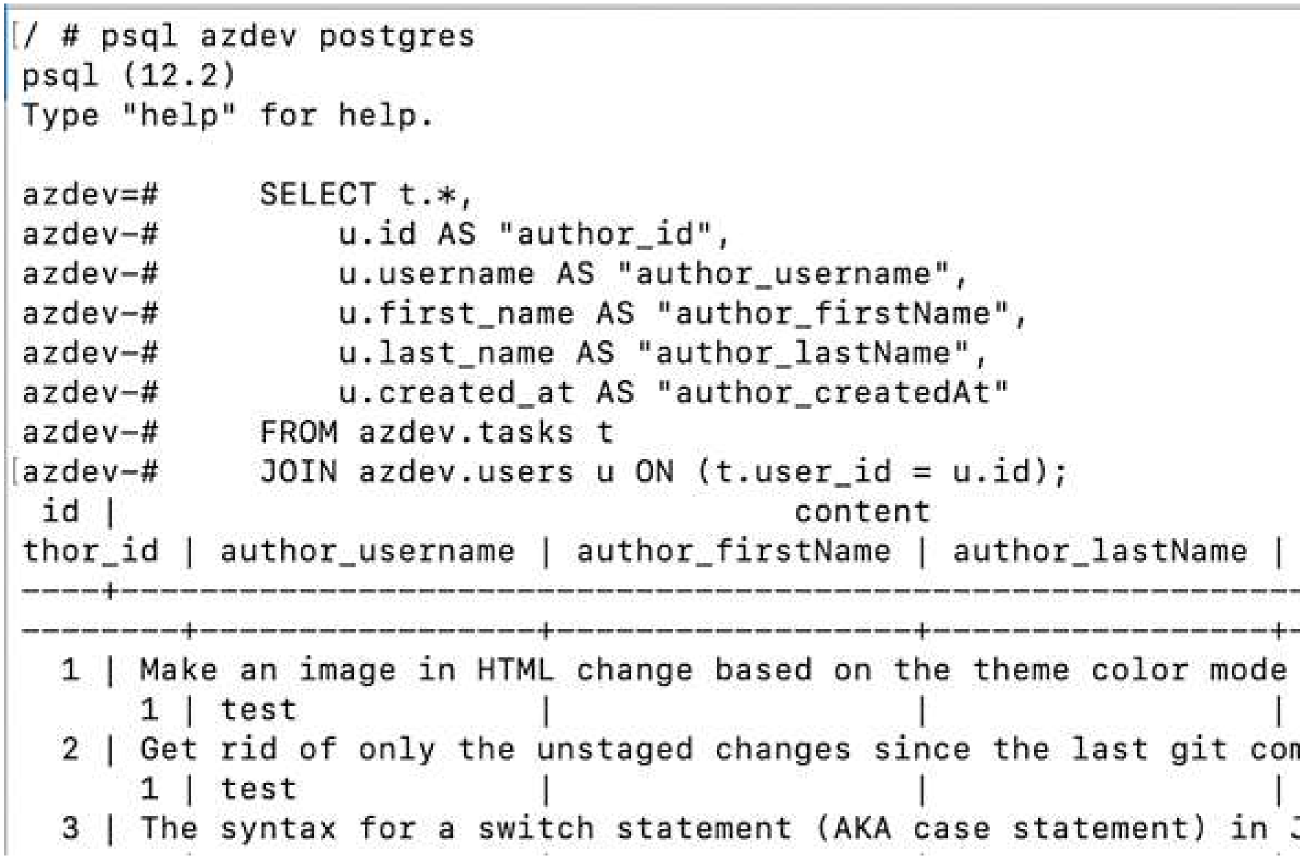


```
SELECT id, username, first_name AS "firstName", last_name AS "lastName", created_at AS "createdAt"
FROM azdev.users
WHERE id = ANY ($1)
2020-12-04 21:19:12.187 UTC [97] DETAIL: parameters: $1 = '{1}'
2020-12-04 21:19:12.187 UTC [97] LOG: duration: 0.041 ms execute <unnamed>:
SELECT id, username, 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>:
SELECT id, username, first_name AS "firstName", last_name AS "lastName", created_at AS "createdAt"
```

Resolving a one-to-one relation

```
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)
  `
   ,
};
// .-.-.
```

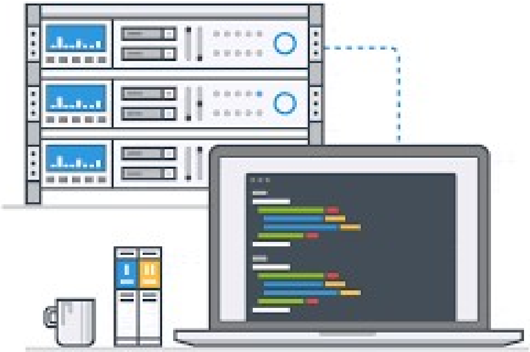
Resolving a one-to-one relation



```
/ # 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
```

Resolving a one-to-one relation

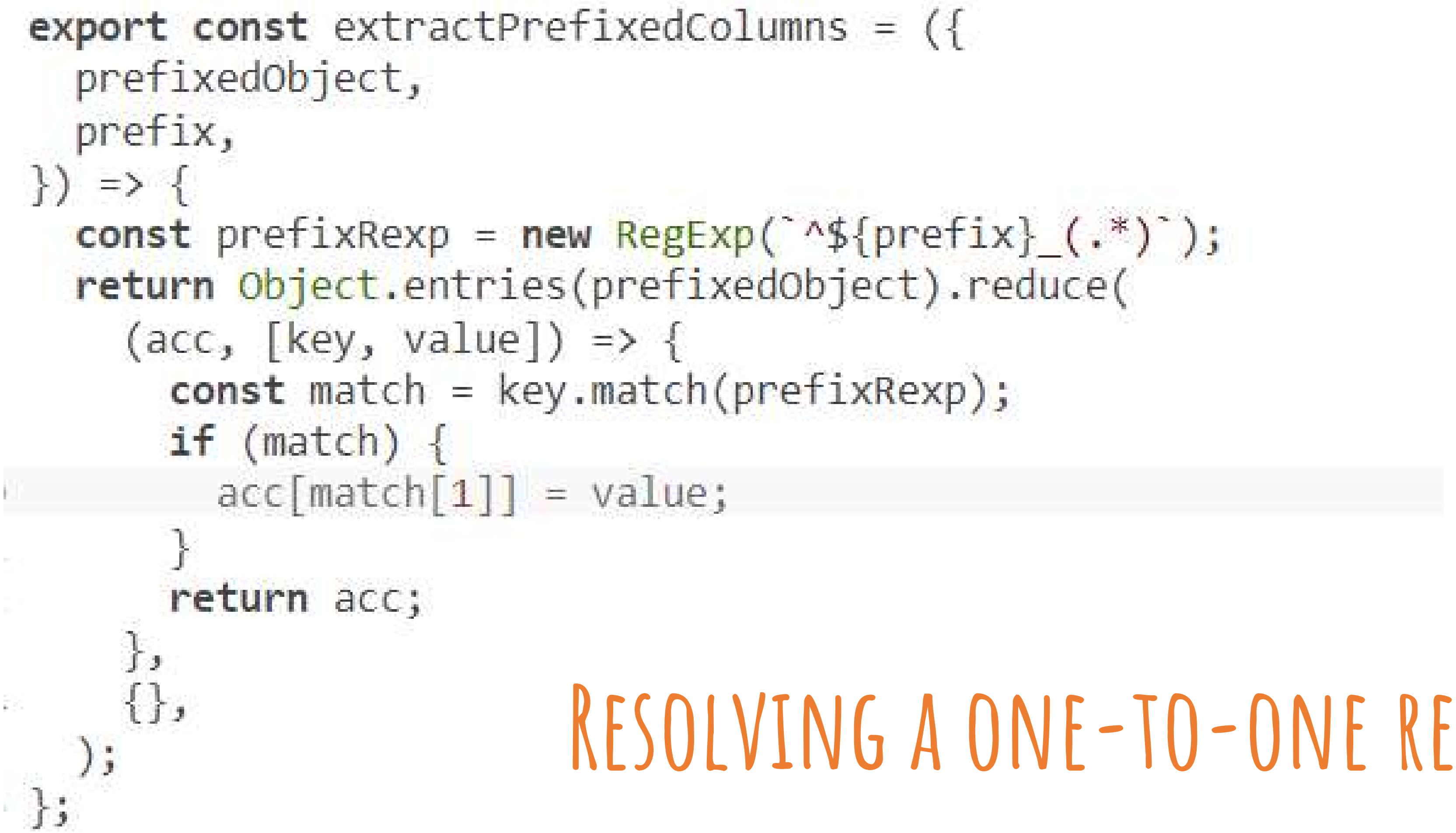
```
taskMainList: `
  SELECT id, content, tags, ·-·-·
    "author_id", "author_username",
"author_firstName",
    "author_lastName", "author_createdAt"
  FROM (${views.tasksAndUsers})
  WHERE is_private = FALSE
  ORDER BY created_at DESC
  LIMIT 100
`,
```



RESOLVING A ONE-TO-ONE RELATION

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

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

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

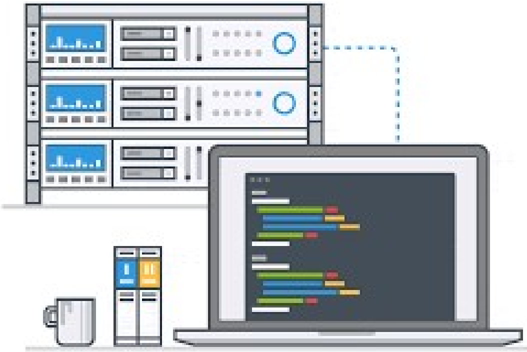
Resolving a one-to-one relation



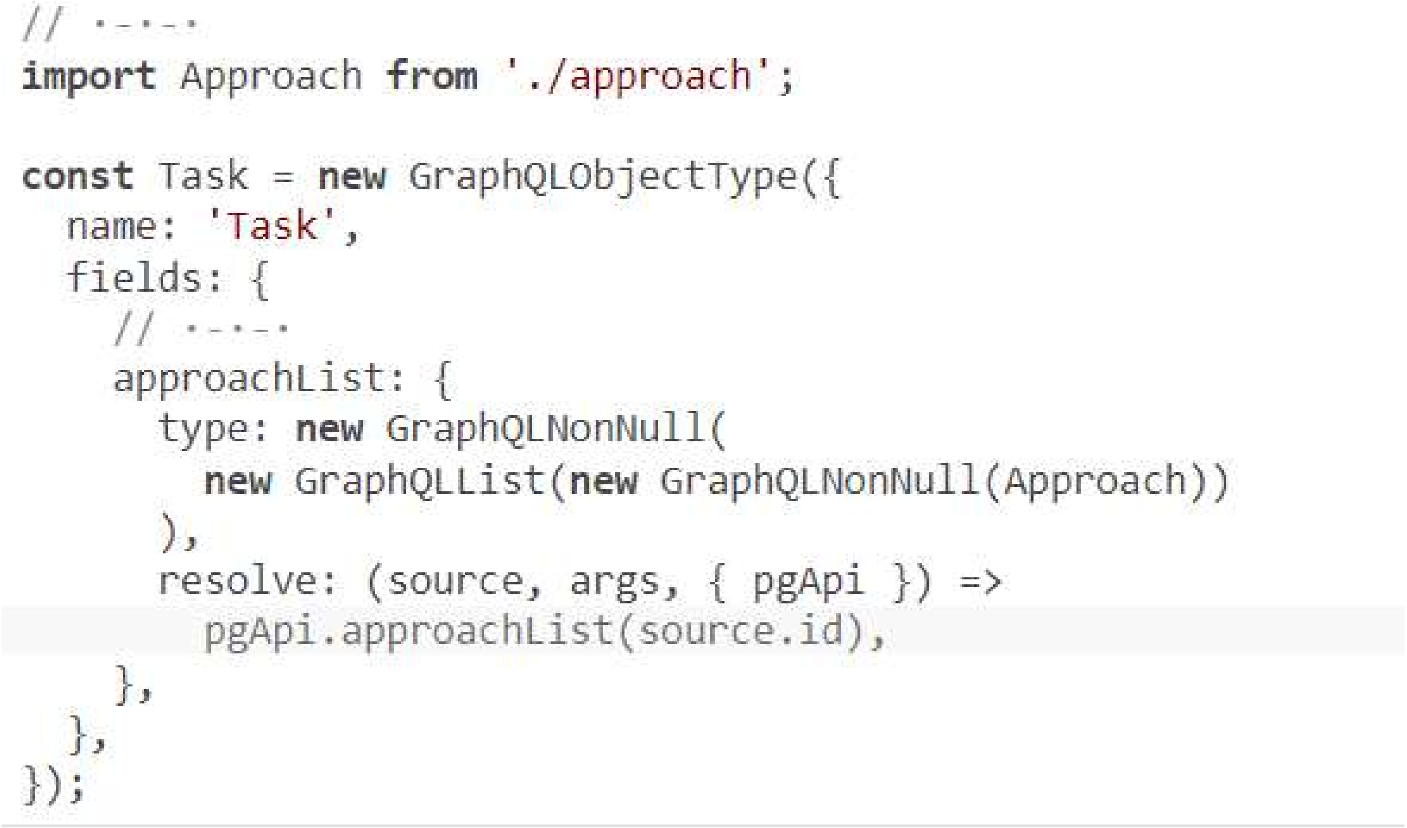
```
export const extractPrefixedColumns = ({
  prefixedObject,
  prefix,
}) => {
  const prefixRexp = new RegExp(`^${prefix}_(.*)`);
  return Object.entries(prefixedObject).reduce(
    (acc, [key, value]) => {
      const match = key.match(prefixRexp);
      if (match) {
        acc[match[1]] = value;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

Resolving a one-to-one relation

```
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

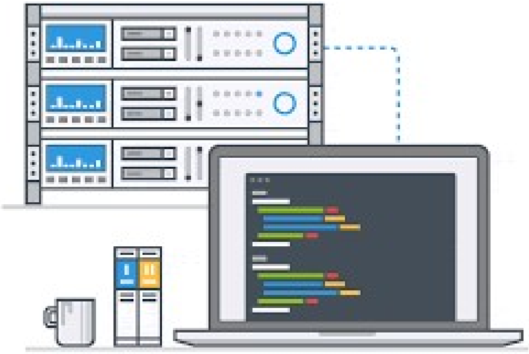


```
// ·-·-·
import Approach from './approach';

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

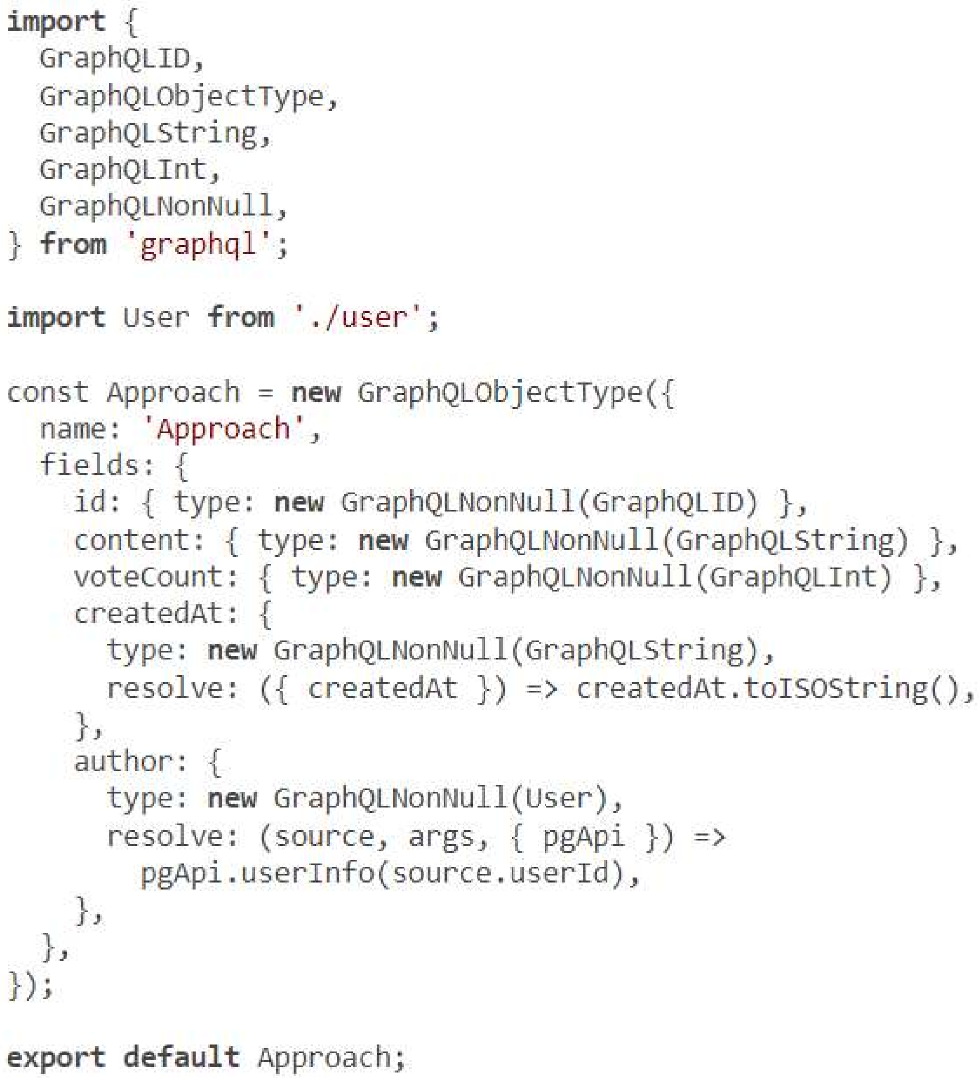
Resolving a one-to-many relation

- 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!]!
}
```



Resolving a one-to-many relation



```
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;
```

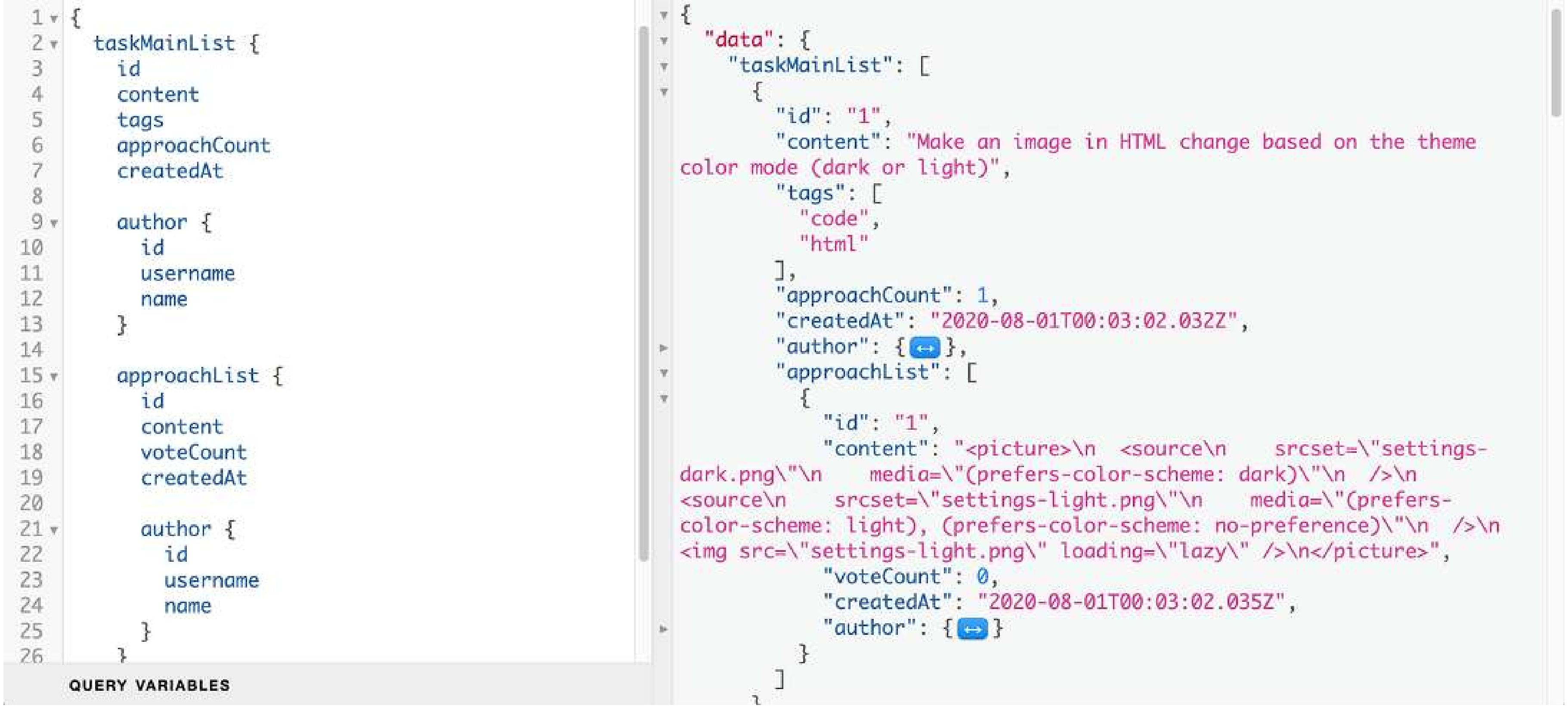
Resolving a one-to-many relation

```
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
`,
```

Resolving a one-to-many relation

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

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

Resolving a one-to-many relation



```
{
  taskMainList {
    id
    content
    tags
    approachCount
    createdAt

    author {
      id
      username
      name
    }

    approachList {
      id
      content
      voteCount
      createdAt

      author {
        id
        username
        name
      }
    }
```

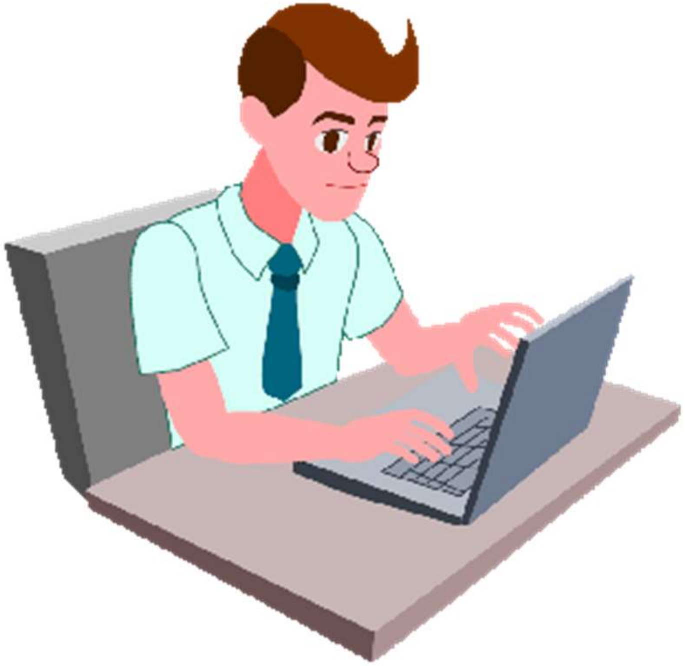
QUERY VARIABLES

```
{
  "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.

- You can use the GraphQL context object to make a pool of database connections available to all resolver functions.



"Complete Lab"