Optimizing data fetching

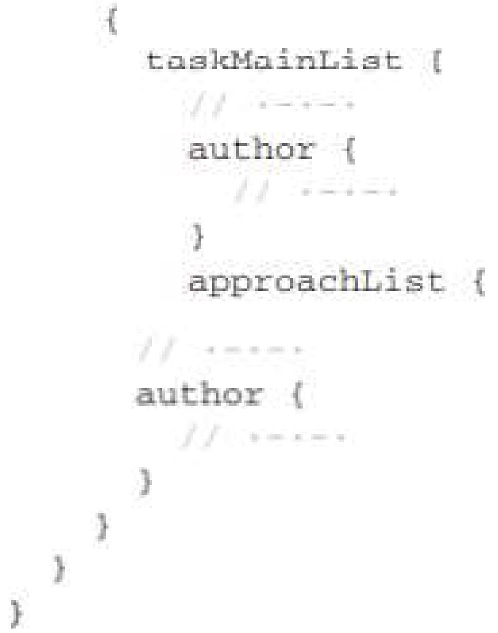
LEARNING VOYAGE

Optimizing data fetching

This lesson covers

- Caching and batching data-fetch operations
- Using the DataLoader library with primary keys and custom IDs
- Using GraphQL’s union type and field arguments
- Reading data from MongoDB

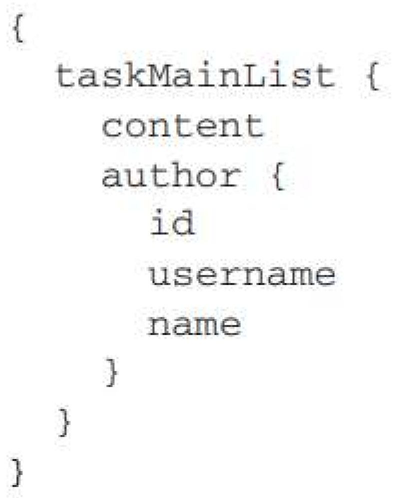
Optimizing data fetching



```
    {
      taskMainList {
        // .....
        author {
          // .....
        }
        approachList {
      // .....
      author {
        // .....
      }
    }
  }
}
```

Caching and batching

- To analyze a solution to this problem, let's go back to the simpler query

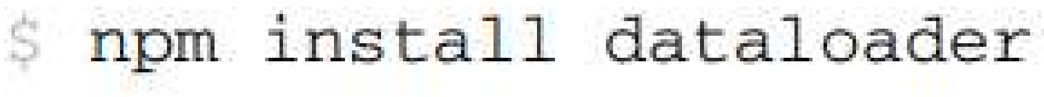


```
{
  taskMainList {
    content
    author {
      id
      username
      name
    }
  }
}
```

Caching and batching

- DataLoader is a generic JavaScript utility library that can be injected into your application's data-fetching layer to manage caching and batching operations on your behalf.

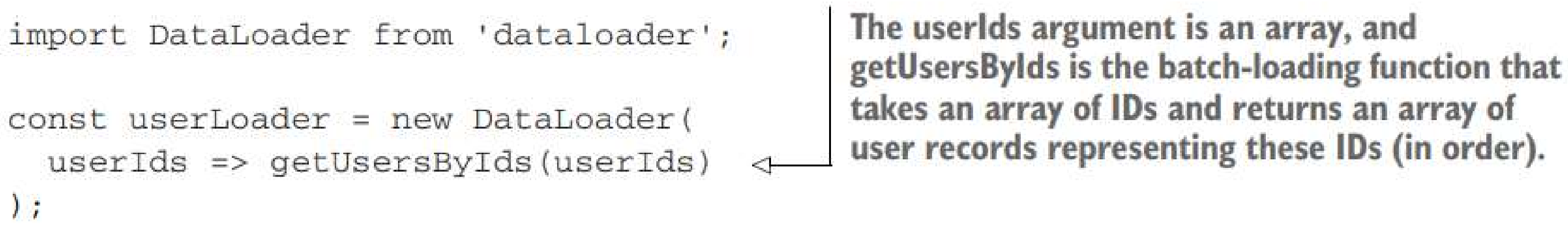
- To use DataLoader in the AZdev API project, we need to install it first.



```
$ npm install dataloader
```

Caching and batching

- For example, here’s one way to create a loader responsible for loading user records.

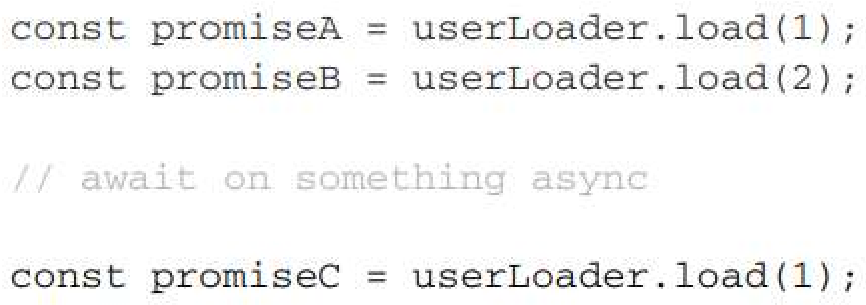


```
import DataLoader from 'dataloader';

const userLoader = new DataLoader(
  userIds => getUsersByIds(userIds)
);
```

The userIds argument is an array, and getUsersByIds is the batch-loading function that takes an array of IDs and returns an array of user records representing these IDs (in order).

Caching and batching

- For example, imagine that a request in your API application needs to load information about users in the following order.



```
const promiseA = userLoader.load(1);
const promiseB = userLoader.load(2);

// await on something async

const promiseC = userLoader.load(1);
```

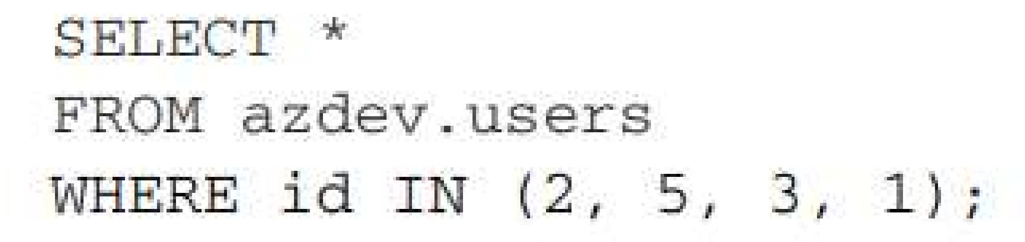
The batch-loading function

- A batch-loading function like getUsersByIds accepts an array of IDs (or generic keys) and should return a promise object that resolves to an array of records.

- To be compatible with DataLoader, the resulting array must be the exact same length as the input array of IDs, and each index in the resulting array of records must correspond to the same index in the input array of IDs.

The batch-loading function

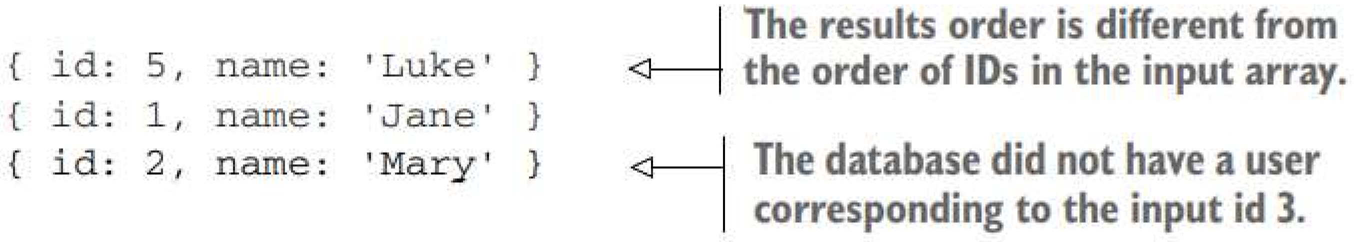
- Here’s one way to do that in PostgreSQL



```
SELECT *
FROM azdev.users
WHERE id IN (2, 5, 3, 1);
```

The batch-loading function

- For the sake of this example, let’s assume that for this SQL statement, the database returned three user records (instead of four) in the following order:



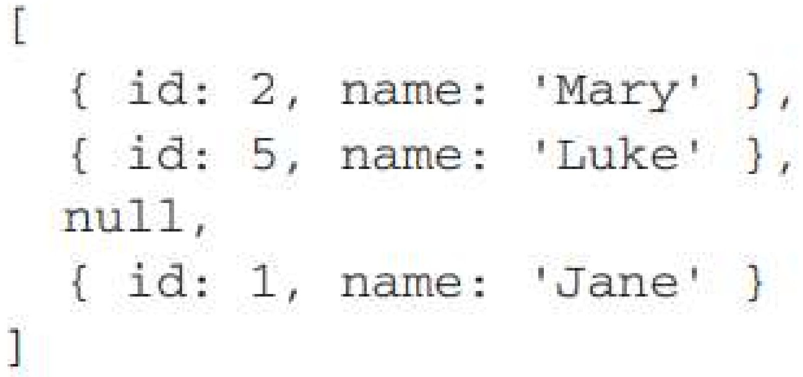
```
{ id: 5, name: 'Luke' }
{ id: 1, name: 'Jane' }
{ id: 2, name: 'Mary' }
```

The results order is different from the order of IDs in the input array.

The database did not have a user corresponding to the input id 3.

The batch-loading function

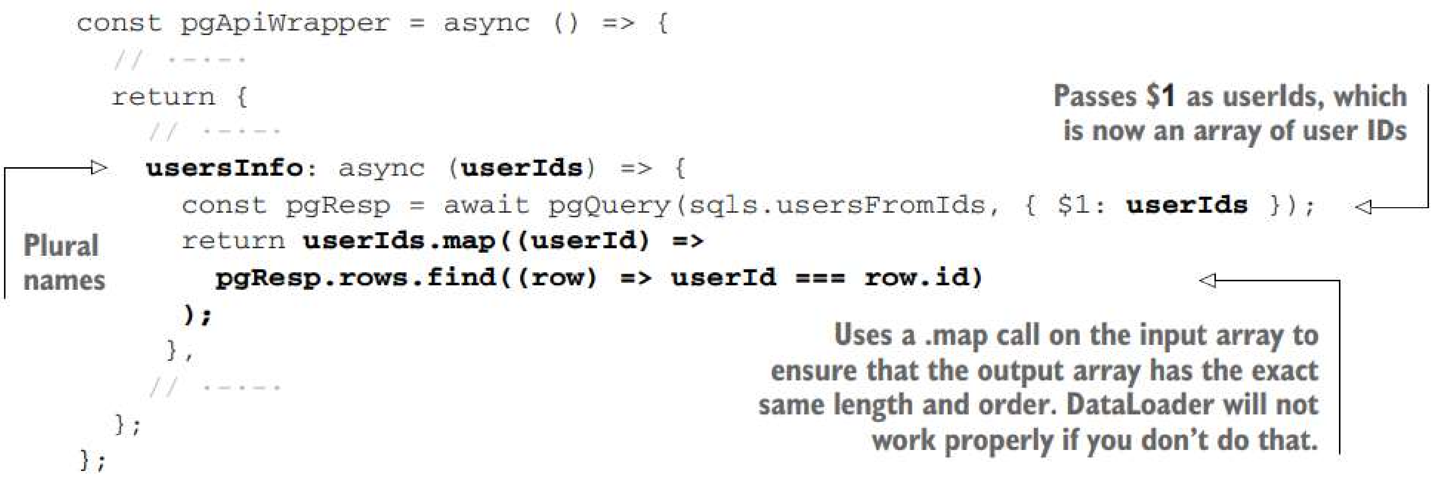
- If an ID has no corresponding record in the
- result, it should be represented with a null value:



```
[
  { id: 2, name: 'Mary' },
  { id: 5, name: 'Luke' },
  null,
  { id: 1, name: 'Jane' }
]
```

The batch-loading function

- We need to make the pgApi.usersInfo a DataLoadercompatible batch-loading function



```
const pgApiWrapper = async () => {
  // -.-.-
  return {
    // -.-.-
    usersInfo: async (userIds) => {
      const pgResp = await pgQuery(sqls.usersFromIds, { $1: userIds });
      return userIds.map((userId) =>
        pgResp.rows.find((row) => userId === row.id)
      );
    },
    // -.-.-
  };
};
```

Plural names

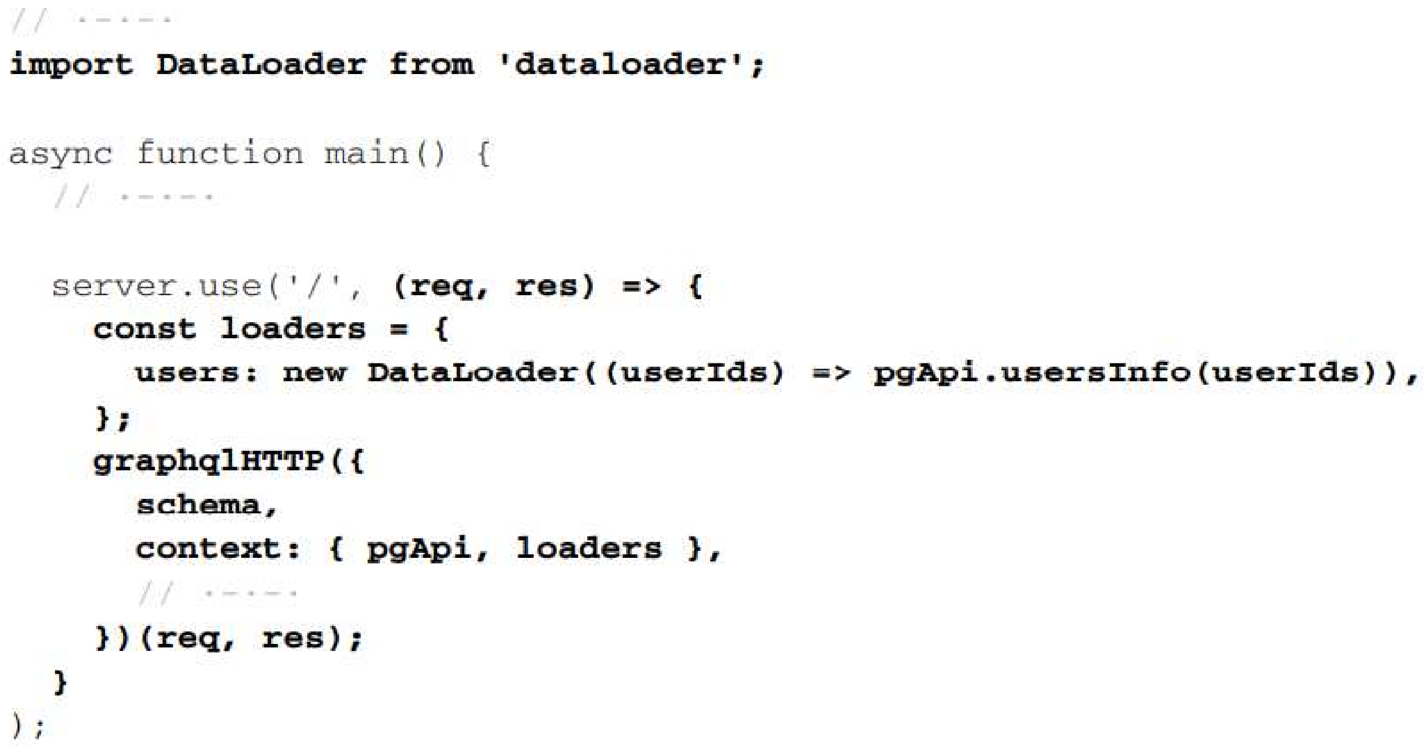
Passes $1 as userIds, which is now an array of user IDs

Uses a .map call on the input array to ensure that the output array has the exact same length and order. DataLoader will not work properly if you don't do that.

Defining and using a DataLoader instance

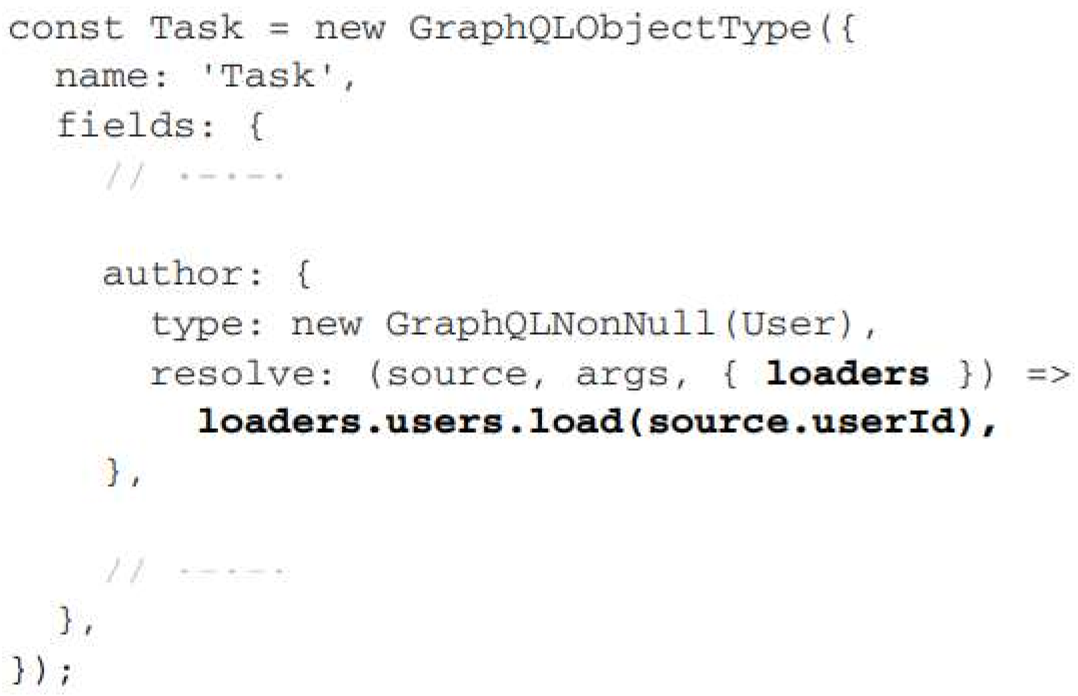
- DataLoader caching is not meant to be part of your application-level caching that’s shared among requests.
- It’s meant to be a simple memoization to avoid repeatedly loading the same data in the context of a single request in your application.
- To do that, you should initialize a loader object for each request in your application and use it only for that request.

Defining and using a DataLoader instance



```
// ·-·-·
import DataLoader from 'dataloader';

async function main() {
  // ·-·-·

  server.use('/', (req, res) => {
    const loaders = {
      users: new DataLoader((userIds) => pgApi.usersInfo(userIds)),
    };
    graphqlHTTP({
      schema,
      context: { pgApi, loaders },
      // ·-·-·
    })(req, res);
  }
);
```

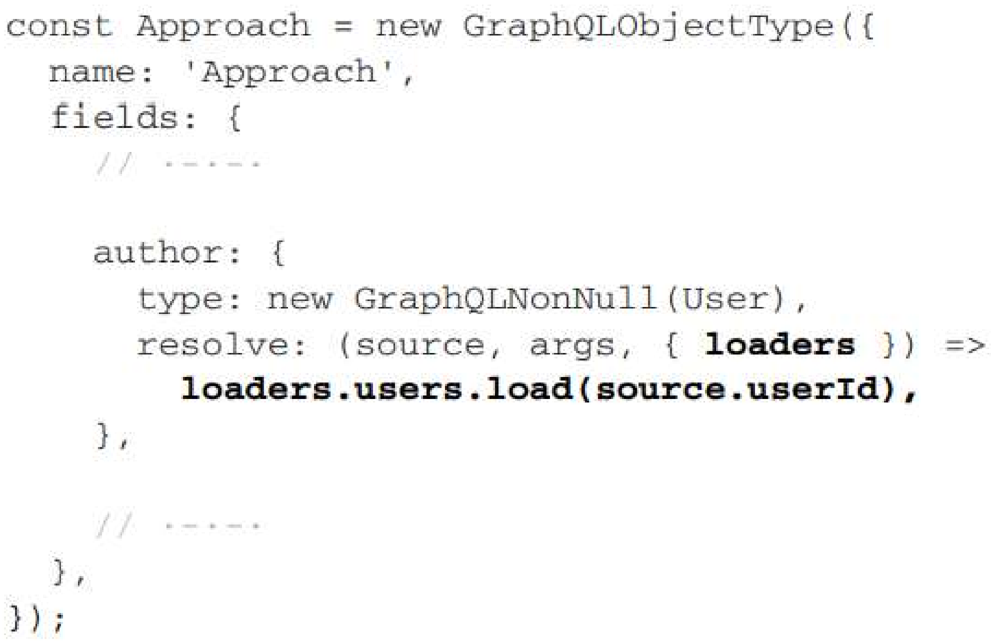
Defining and using a DataLoader instance



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

    author: {
      type: new GraphQLNonNull(User),
      resolve: (source, args, { loaders }) =>
        loaders.users.load(source.userId),
    },

    // ·-·-·
  },
});
```

Defining and using a DataLoader instance

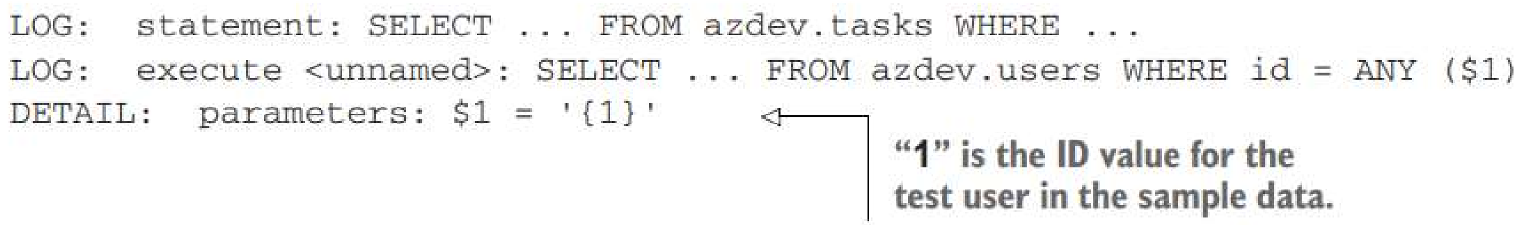
- And here is the Approach type.



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

    author: {
      type: new GraphQLNonNull(User),
      resolve: (source, args, { loaders }) =>
        loaders.users.load(source.userId),
    },

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

Defining and using a DataLoader instance

- If we try the same GraphQL query now while tailing the logs of PostgreSQL, we will see something like the following excerpt from my Postgr



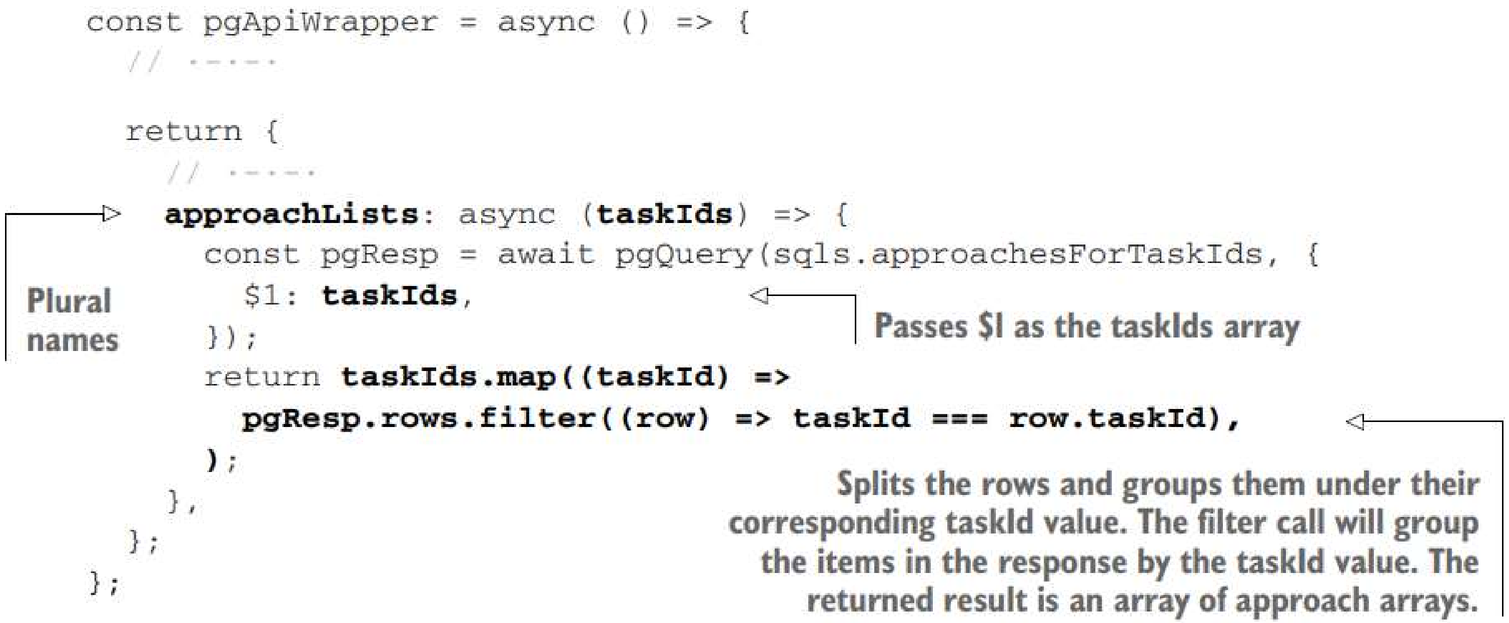
```
LOG:   statement: SELECT ... FROM azdev.tasks WHERE ...
LOG:   execute <unnamed>: SELECT ... FROM azdev.users WHERE id = ANY ($1)
DETAIL:   parameters: $1 = '{1}'
```

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

The loader for the approachList field

- The other ID-based fetching we have done so far is in the pgApi.approachList function in api/src/db/pg-api.js.
- This function is a bit different than the pgApi .usersInfo function as it takes a taskId and returns an array of Approach records.
- This means when we switch it to work with an array of keys instead of a single value, it will take an array of taskIds, and it should return an array of arrays (each array representing the list of Approaches for one Task).

The loader for the approachList field



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

  return {
    // -----
    approachLists: async (taskIds) => {
      const pgResp = await pgQuery(sqls.approachesForTaskIds, {
        $1: taskIds,
      });
      return taskIds.map((taskId) =>
        pgResp.rows.filter((row) => taskId === row.taskId),
      );
    },
  };
};
```

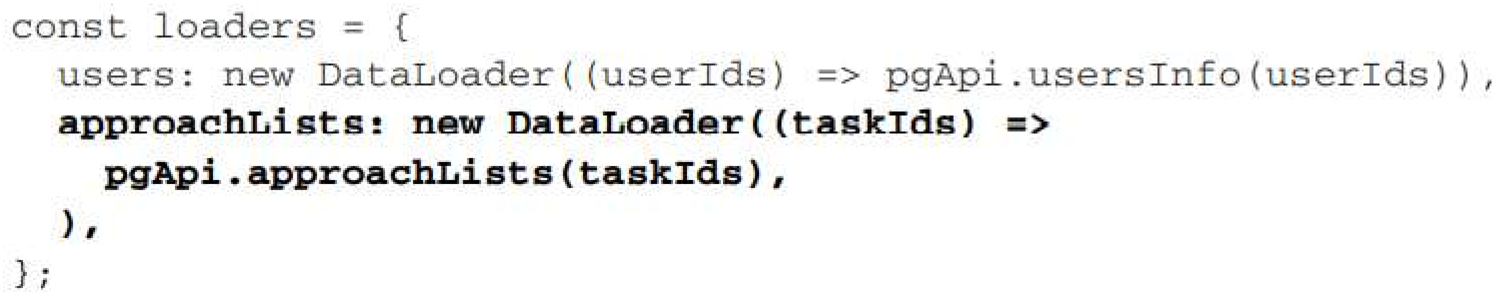
Plural names

Passes $I as the taskIds array

Splits the rows and groups them under their corresponding taskId value. The filter call will group the items in the response by the taskId value. The returned result is an array of approach arrays.

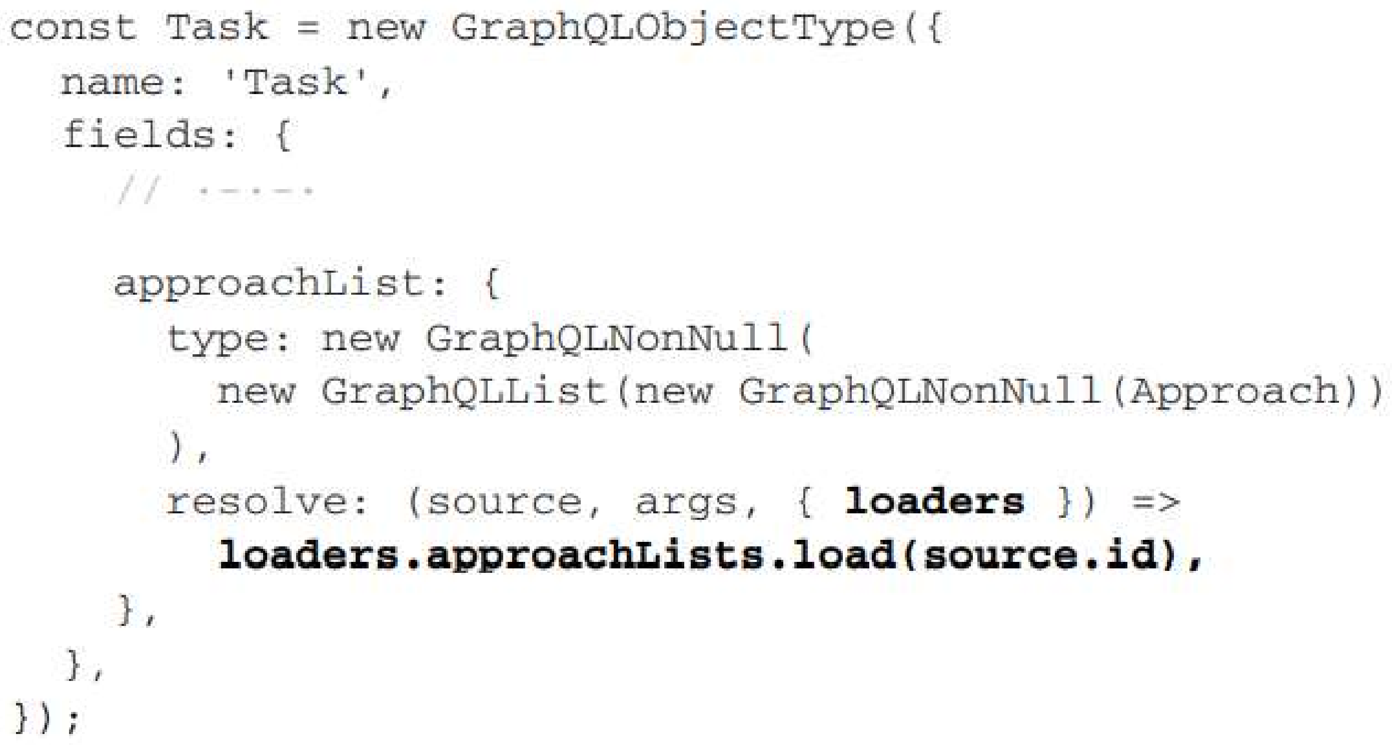
The loader for the approachList field

- The pgApi.approachLists batch-loading function is now compatible with DataLoader.
- To use it, we instantiate a new loader instance in api/src/server.js.



```
const loaders = {
  users: new DataLoader((userIds) => pgApi.usersInfo(userIds)),
  approachLists: new DataLoader((taskIds) =>
    pgApi.approachLists(taskIds),
  ),
};
```

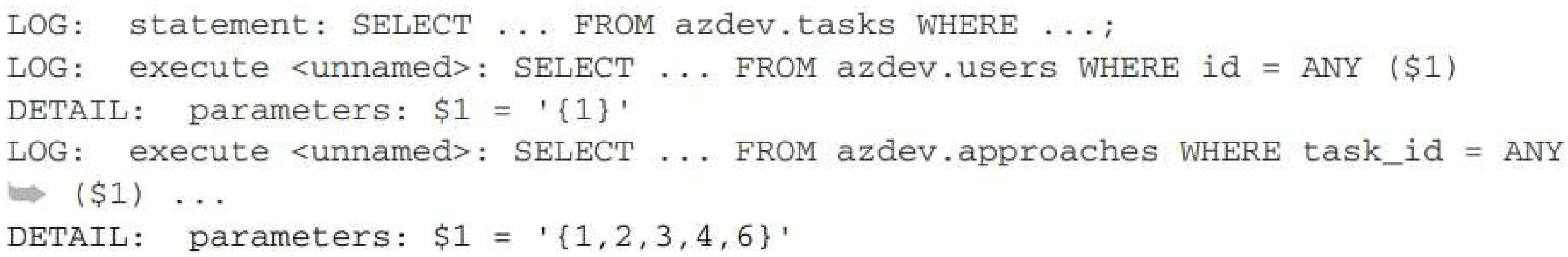
The loader for the approachList field



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

    approachList: {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(Approach))
      ),
      resolve: (source, args, { loaders }) =>
        loaders.approachLists.load(source.id),
    },
  },
});
```

The loader for the approachList field

- That should do it, Go ahead and test the same query we tested at the end of previous lesson while tailing the PostgreSQL logs.
- You should see something like this excerpt from my PostgreSQL logs:

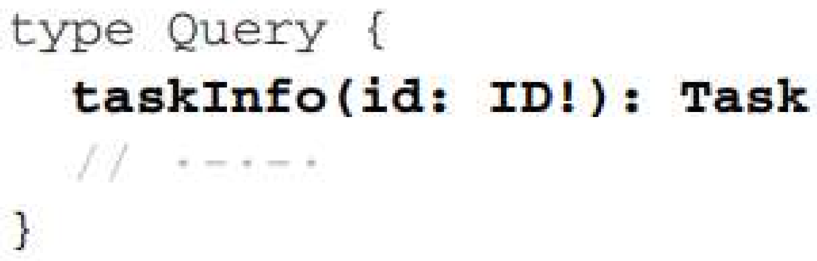


```
LOG:  statement: SELECT ... FROM azdev.tasks WHERE ...;
LOG:  execute <unnamed>: SELECT ... FROM azdev.users WHERE id = ANY ($1)
DETAIL:  parameters: $1 = '{1}'
LOG:  execute <unnamed>: SELECT ... FROM azdev.approaches WHERE task_id = ANY
➥ ($1) ...
DETAIL:  parameters: $1 = '{1,2,3,4,6}'
```

```
{
  taskMainList {
    id
    author {
      id
    }
    a1: approachList {
      id
      author {
        id
      }
    }
    a2: approachList {
      id
      author {
        id
      }
    }
    a3: approachList {
      id
      author {
        id
      }
    }
  }
}
```

Single resource fields

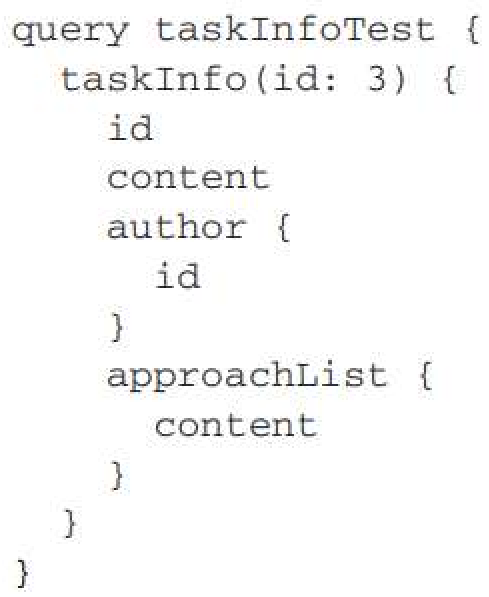
- In our schema plan, the taskInfo root query root field is supposed to fetch the information for a single Task record identified by an ID that the API consumer can send as a field argument.



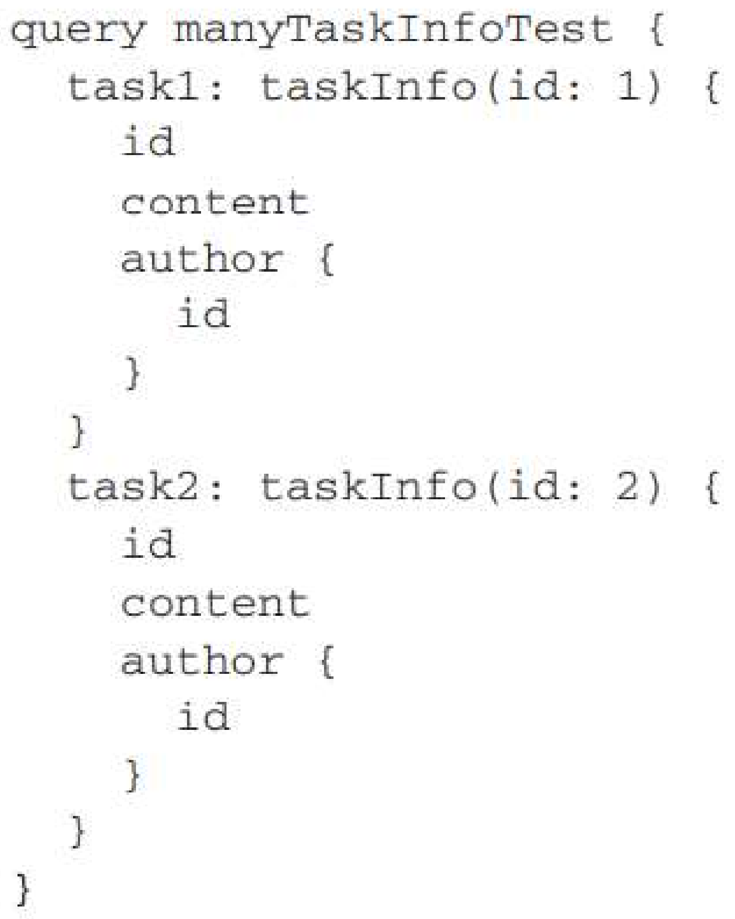
```
type Query {
  taskInfo(id: ID!): Task
  // ·-·-·
}
```

Single resource fields

- A query that we can use to work through this field.



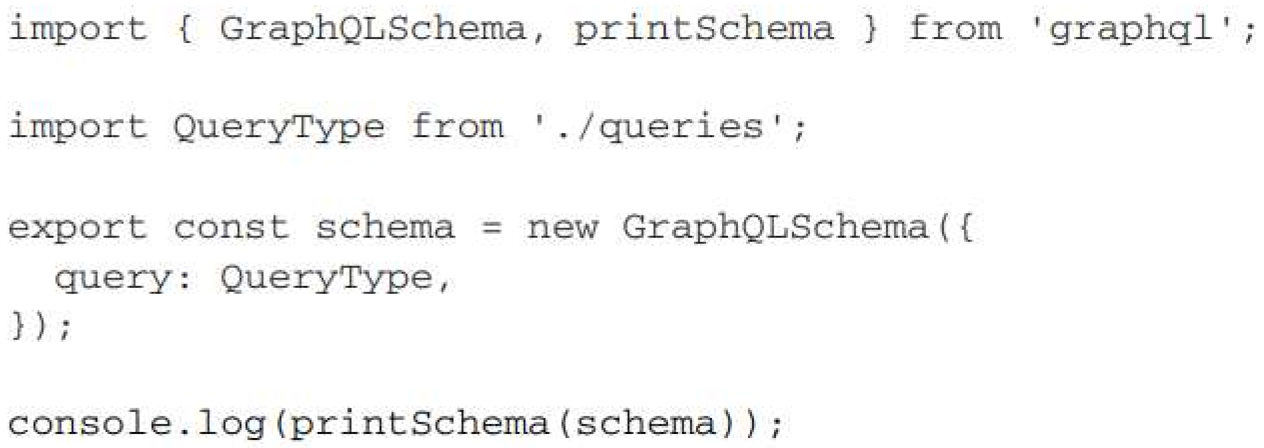
```
query taskInfoTest {
  taskInfo(id: 3) {
    id
    content
    author {
      id
    }
    approachList {
      content
    }
  }
}
```



```
query manyTaskInfoTest {
  task1: taskInfo(id: 1) {
    id
    content
    author {
      id
    }
  }
  task2: taskInfo(id: 2) {
    id
    content
    author {
      id
    }
  }
}
```

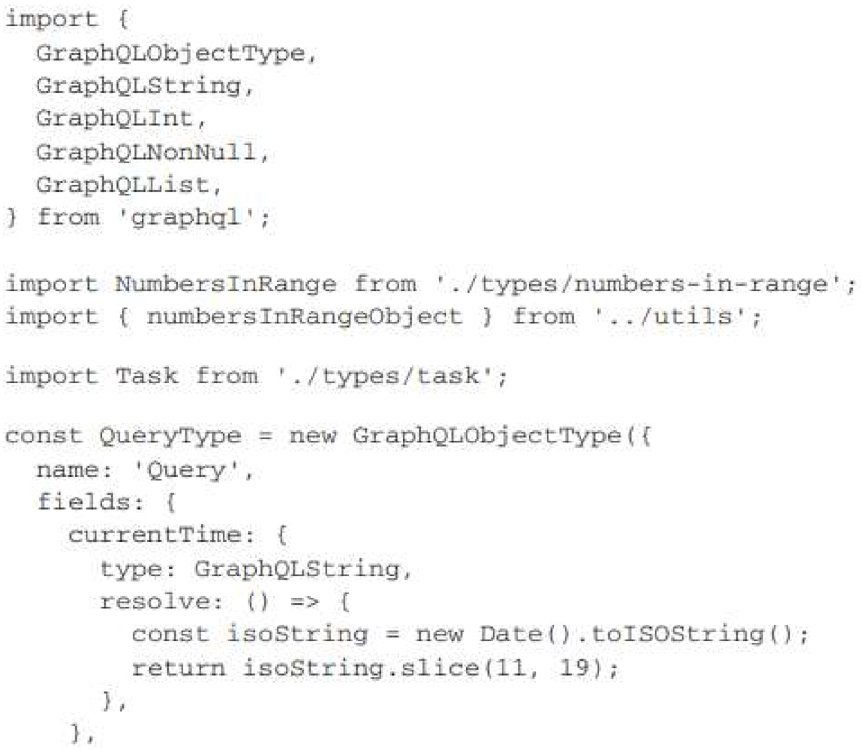
Single resource fields

- Let’s split tasks into two files instead of one.

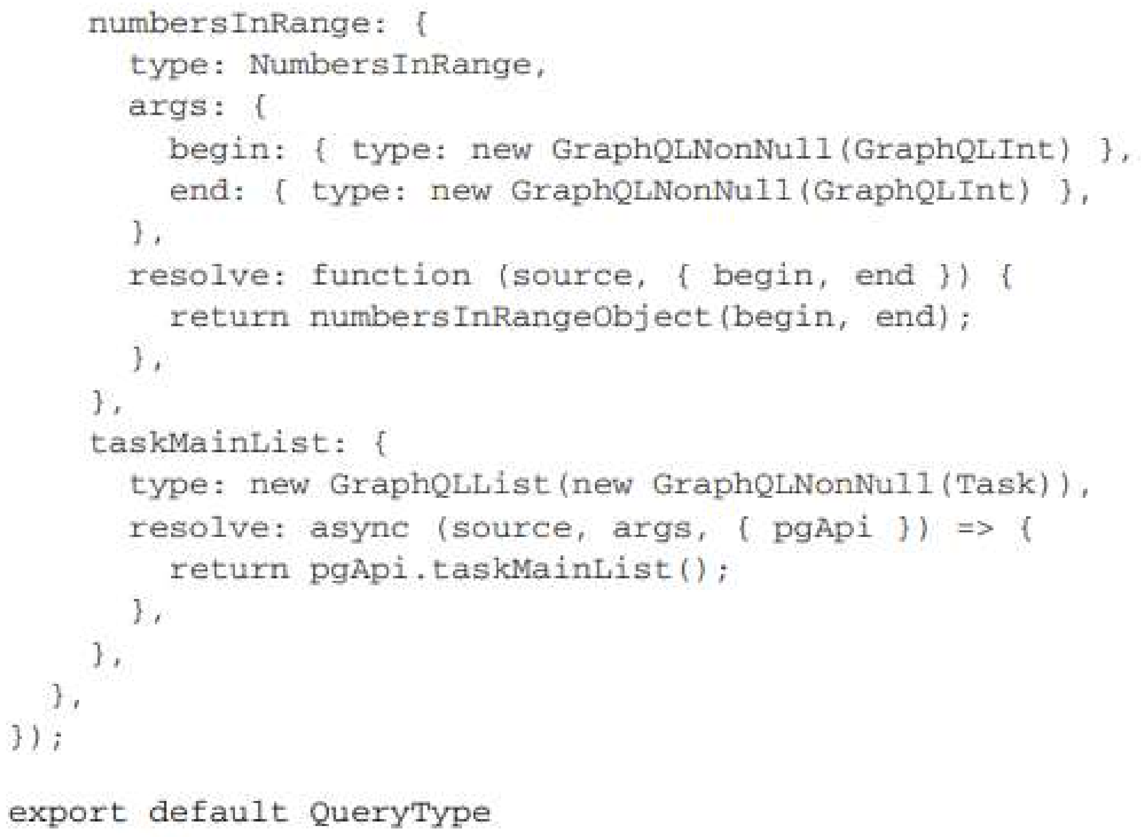


```
import { GraphQLSchema, printSchema } from 'graphql';

import QueryType from './queries';

export const schema = new GraphQLSchema({
  query: QueryType,
});

console.log(printSchema(schema));
```

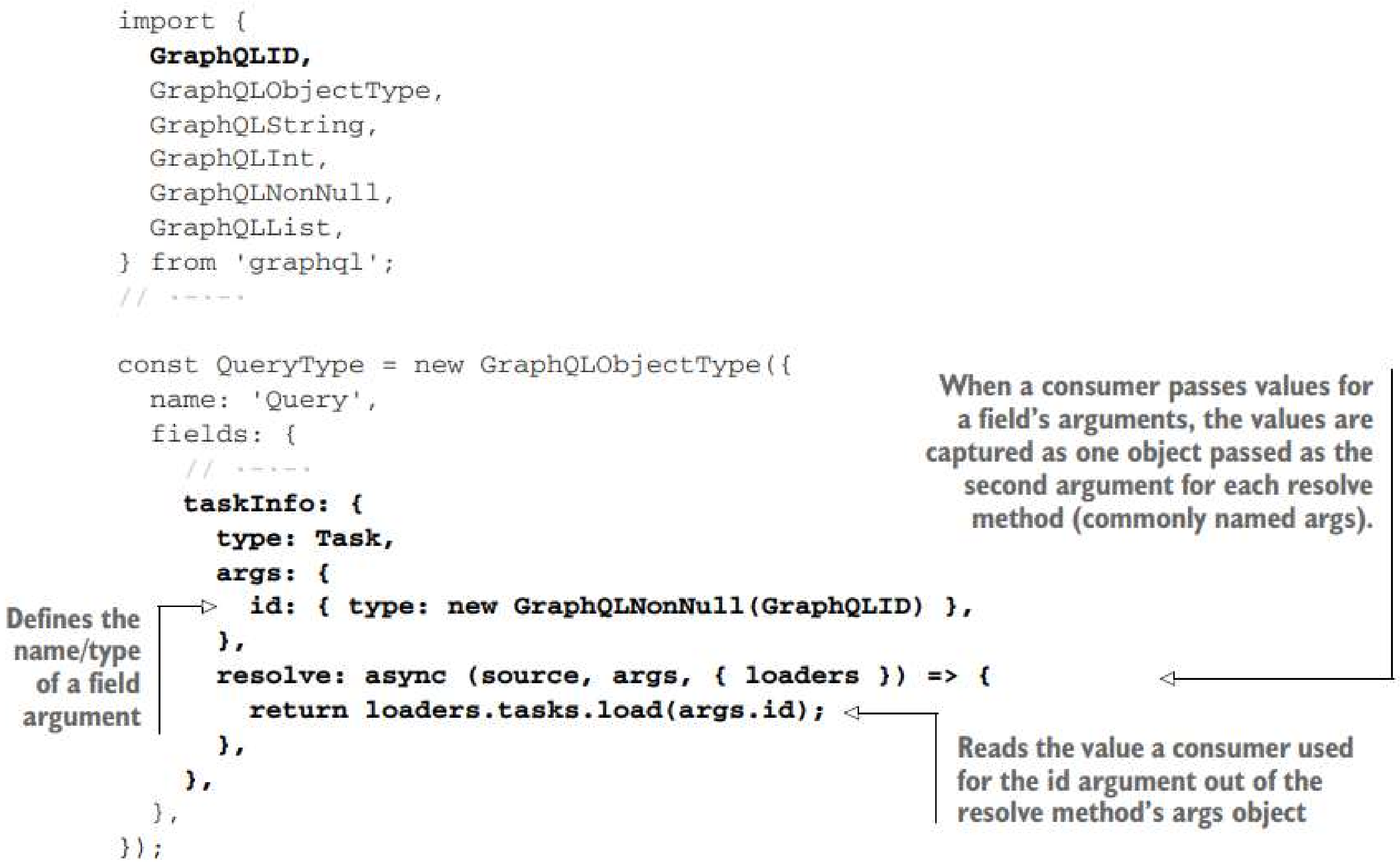
Single resource fields



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

import NumbersInRange from './types/numbers-in-range';
import { numbersInRangeObject } from '../utils';

import Task from './types/task';

const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: {
    currentTime: {
      type: GraphQLString,
      resolve: () => {
        const isoString = new Date().toISOString();
        return isoString.slice(11, 19);
      },
    },
```



```
    numbersInRange: {
      type: NumbersInRange,
      args: {
        begin: { type: new GraphQLNonNull(GraphQLInt) },
        end: { type: new GraphQLNonNull(GraphQLInt) },
      },
      resolve: function (source, { begin, end }) {
        return numbersInRangeObject(begin, end);
      },
    },
    taskMainList: {
      type: new GraphQLList(new GraphQLNonNull(Task)),
      resolve: async (source, args, { pgApi }) => {
        return pgApi.taskMainList();
      },
    },
  },
});

export default QueryType
```



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

const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: {
    // ·-·-·
    taskInfo: {
      type: Task,
      args: {
        id: { type: new GraphQLNonNull(GraphQLID) },
      },
      resolve: async (source, args, { loaders }) => {
        return loaders.tasks.load(args.id);
      },
    },
  },
});
```

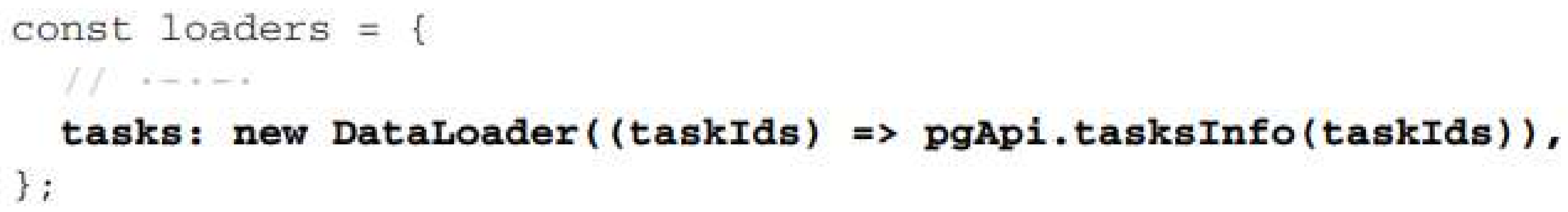
When a consumer passes values for a field's arguments, the values are captured as one object passed as the second argument for each resolve method (commonly named args).

Defines the name/type of a field argument

Reads the value a consumer used for the id argument out of the resolve method's args object

Single resource fields

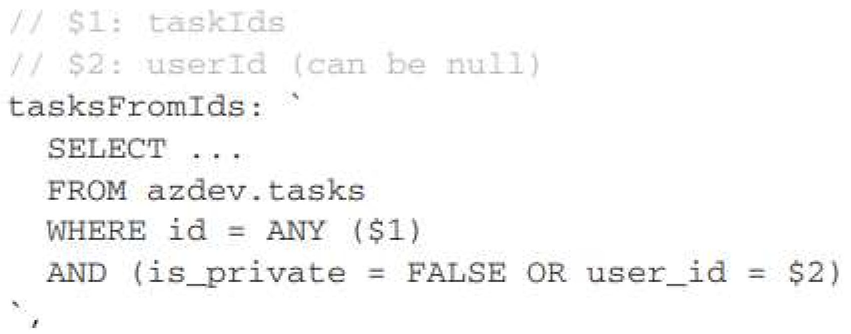
- The new loader function goes in api/src/server.js



```
const loaders = {
  // ·-·-·
  tasks: new DataLoader((taskIds) => pgApi.tasksInfo(taskIds)),
};
```

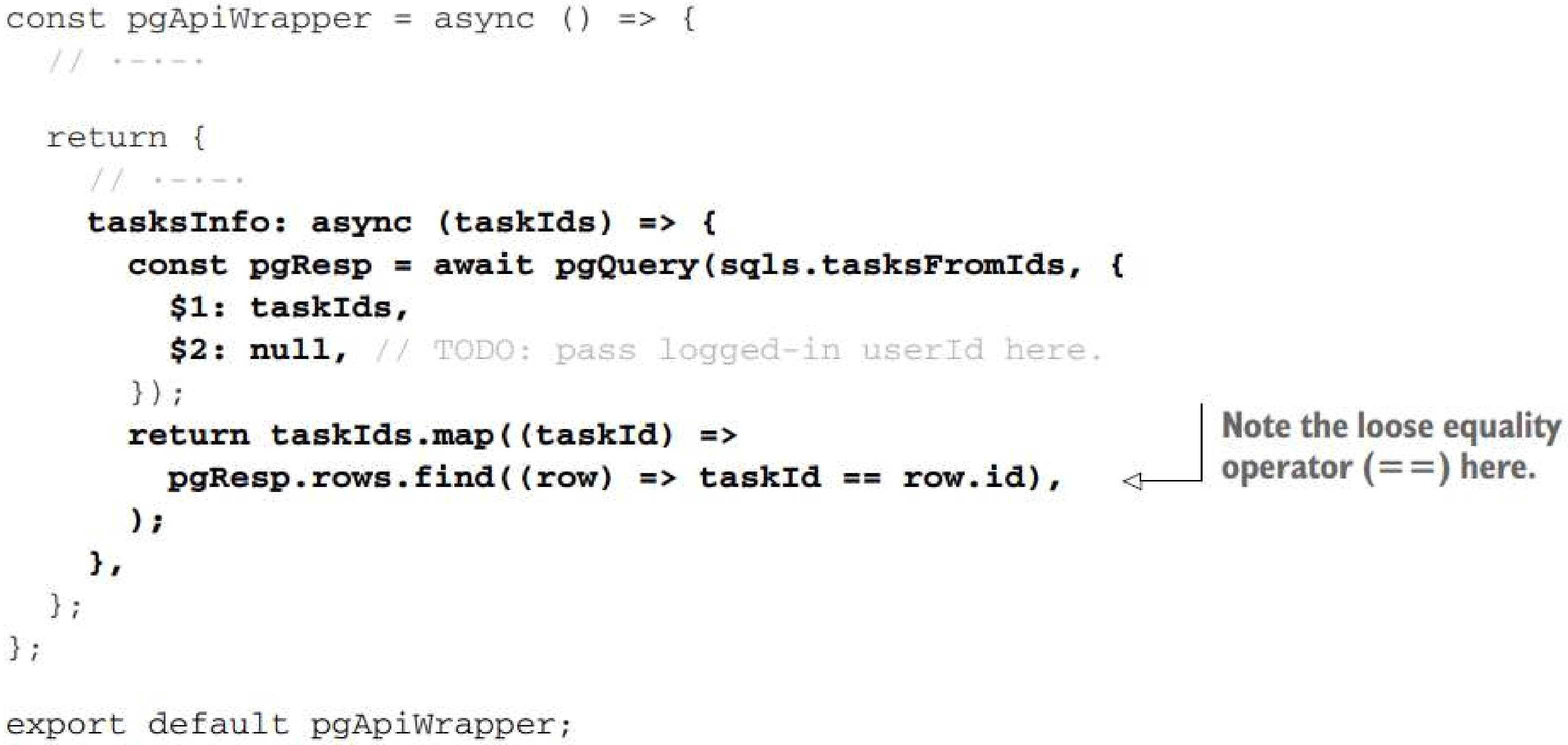
Single resource fields

- Following the top-down analysis, we now need to define the pgApi.tasksInfo function.
- I have prepared a sqls.tasksFromIds statement for it in api/src/db/sqls.j



```
// $1: taskIds
// $2: userId (can be null)
tasksFromIds: `
  SELECT ...
  FROM azdev.tasks
  WHERE id = ANY ($1)
  AND (is_private = FALSE OR user_id = $2)
`,
```

- The pgApi DataLoader-compatible function to execute the SQL statement



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

  return {
    // ·-·-·
    tasksInfo: async (taskIds) => {
      const pgResp = await pgQuery(sqls.tasksFromIds, {
        $1: taskIds,
        $2: null, // TODO: pass logged-in userId here.
      });
      return taskIds.map((taskId) =>
        pgResp.rows.find((row) => taskId == row.id),
      );
    },
  };
};

export default pgApiWrapper;
```

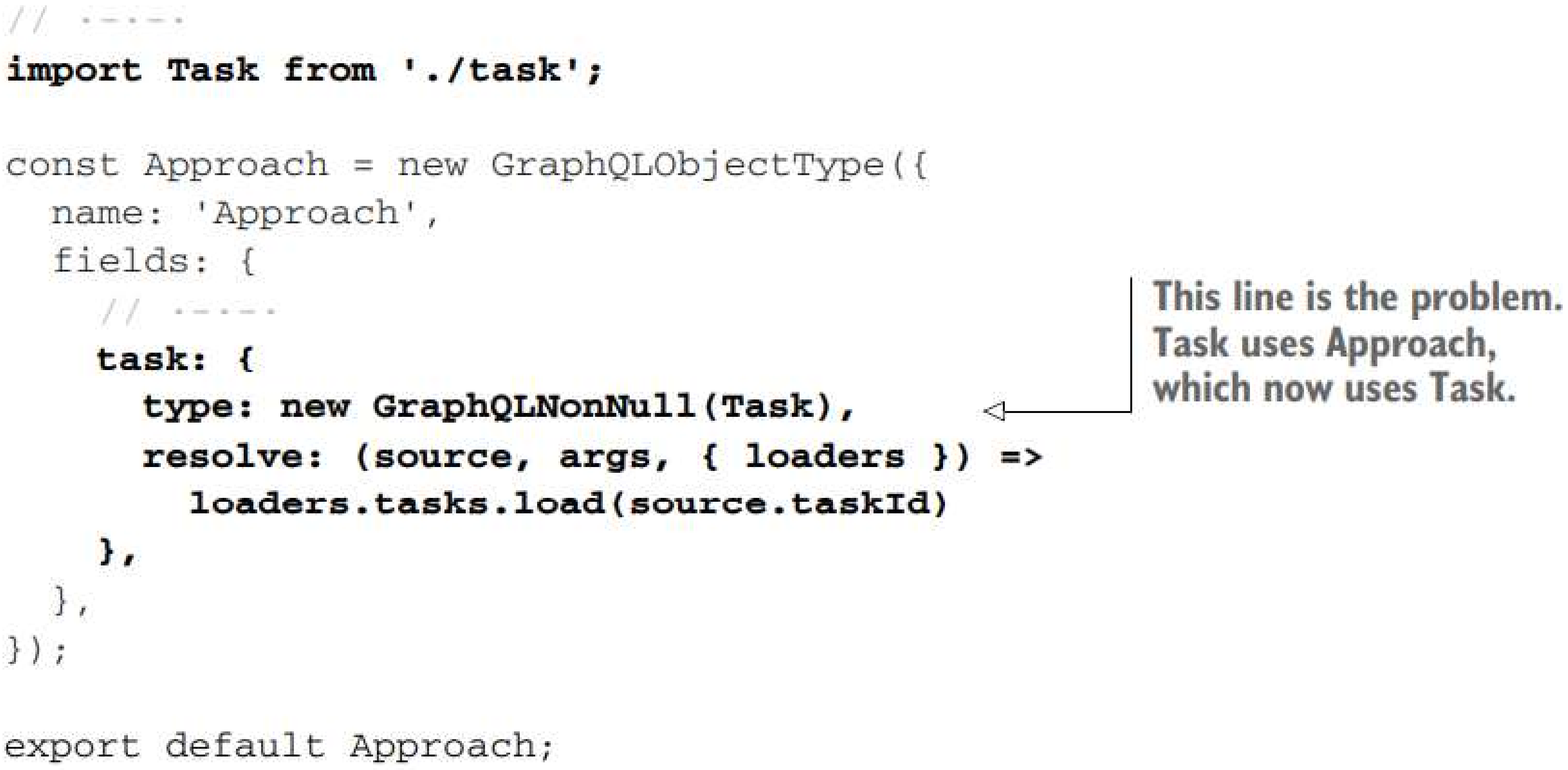
Note the loose equality operator (==) here.

Circular dependencies in GraphQL types

- We designed the Approach type to have a task field so that we can display the parent Task information when a search result item is an Approach record.

- To implement this relation, we can reuse the loaders and pgApi function we wrote for the taskInfo root field.

- However, this relation is the inverse of the Task –> Approach relation we implemented for the approachList field.



```
// ...
import Task from './task';

const Approach = new GraphQLObjectType({
  name: 'Approach',
  fields: {
    // ...
    task: {
      type: new GraphQLNonNull(Task),
      resolve: (source, args, { loaders }) =>
        loaders.tasks.load(source.taskId)
    },
  },
});

export default Approach;
```

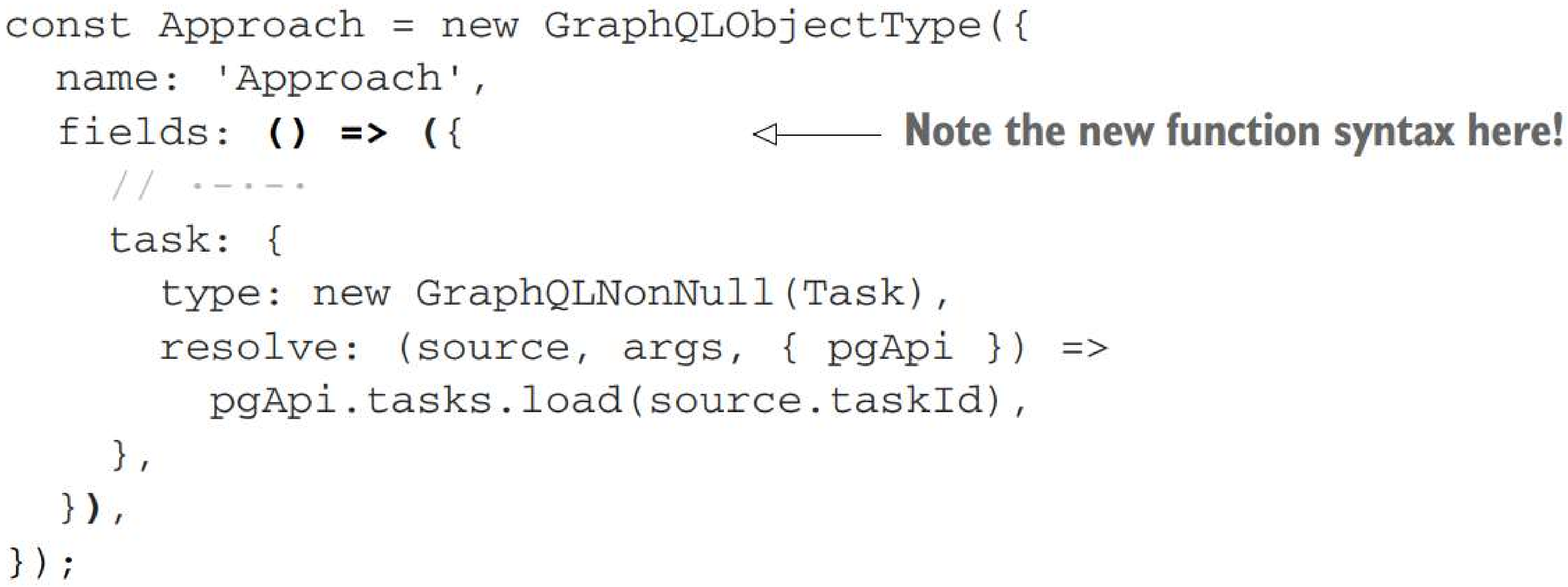
This line is the problem. Task uses Approach, which now uses Task.

Circular dependencies in GraphQL types

- The server logs will report this problem:

```
ReferenceError: Task is not defined
```

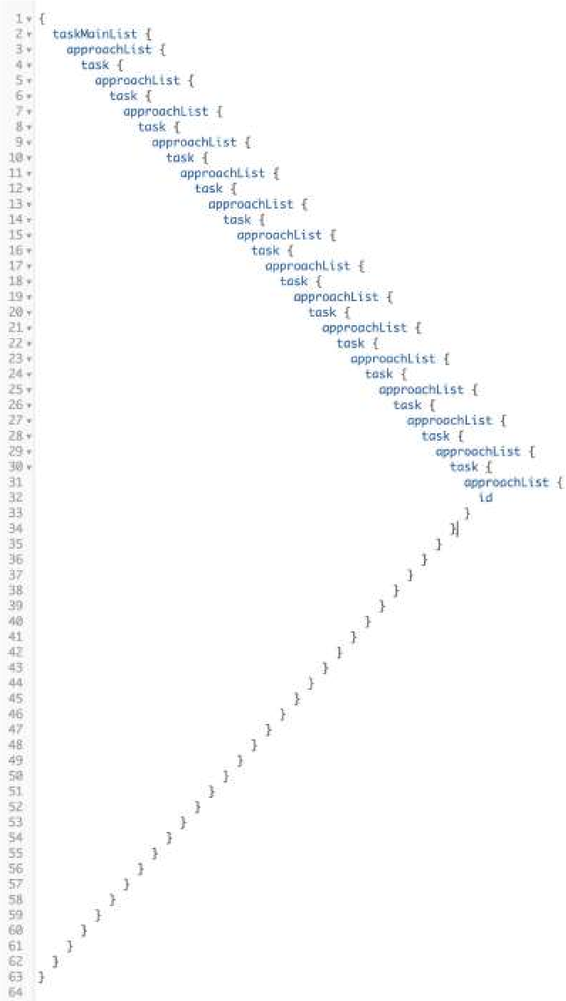
Circular dependencies in GraphQL types



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

Note the new function syntax here!

Deeply nested field attacks



```
{
  taskMainList {
    approachList {
      task {
        approachList {
          task {
            approachList {
              task {
                approachList {
                  task {
                    approachList {
                      task {
                        approachList {
                          task {
                            approachList {
                              task {
                                approachList {
                                  task {
                                    approachList {
                                      task {
                                        approachList {
                                          task {
                                            approachList {
                                              task {
                                                approachList {
                                                  task {
                                                    approachList {
                                                      task {
                                                        approachList {
                                                          task {
                                                            approachList {
                                                              id
                                                            }
                                                          }
                                                        }
                                                      }
                                                    }
                                                  }
                                                }
                                              }
                                            }
                                          }
                                        }
                                      }
                                    }
                                  }
                                }
                              }
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

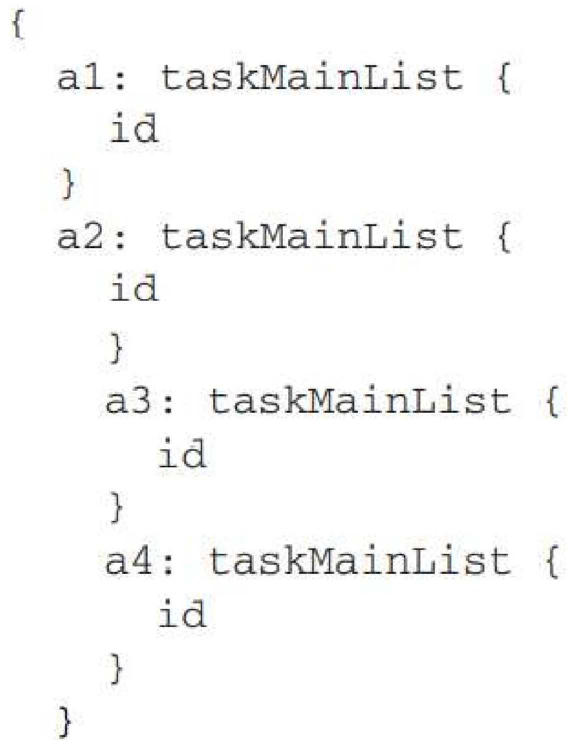
Using DataLoader with custom IDs for caching

- Although a DataLoader batch-loading function is often associated with a list of input IDs, you don't need actual IDs coming from primary fields in the database.
- You can come up with your own ID-to-result association and use DataLoader with the custom map you designed.
- This is usually helpful when you are using the caching aspect of DataLoader.

The taskMainList field

- Let’s test how many SQL statements the following GraphQL query will currently issue.

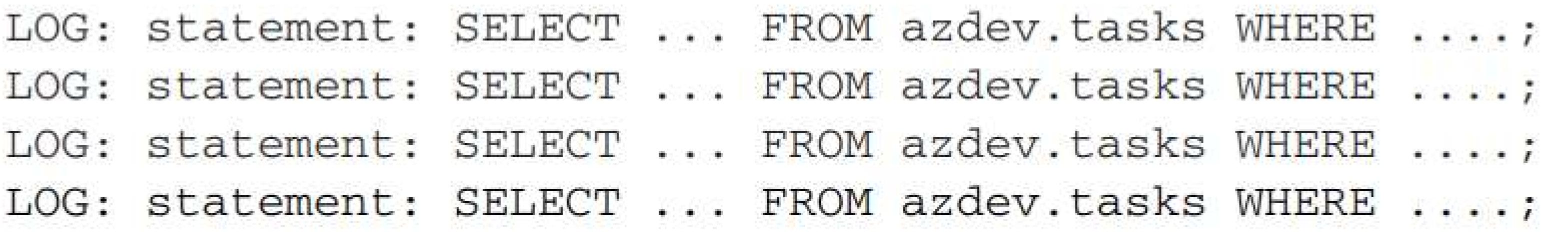
Can you guess?



```
{
  a1: taskMainList {
    id
  }
  a2: taskMainList {
    id
    }
    a3: taskMainList {
      id
    }
    a4: taskMainList {
      id
    }
  }
```

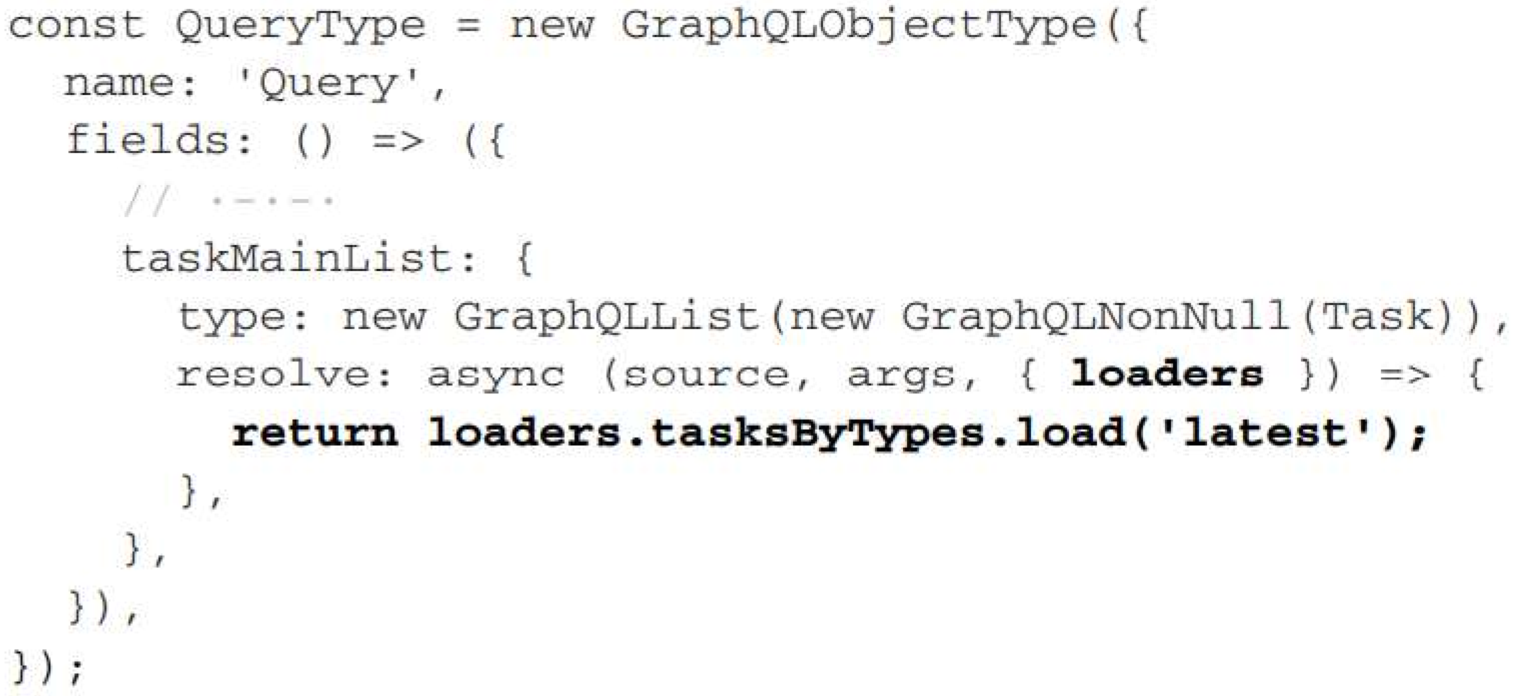
The taskMainList field

- Here’s the related excerpt from my PostgreSQL logs:



```
LOG: statement: SELECT ... FROM azdev.tasks WHERE ....;
LOG: statement: SELECT ... FROM azdev.tasks WHERE ....;
LOG: statement: SELECT ... FROM azdev.tasks WHERE ....;
LOG: statement: SELECT ... FROM azdev.tasks WHERE ....;
```

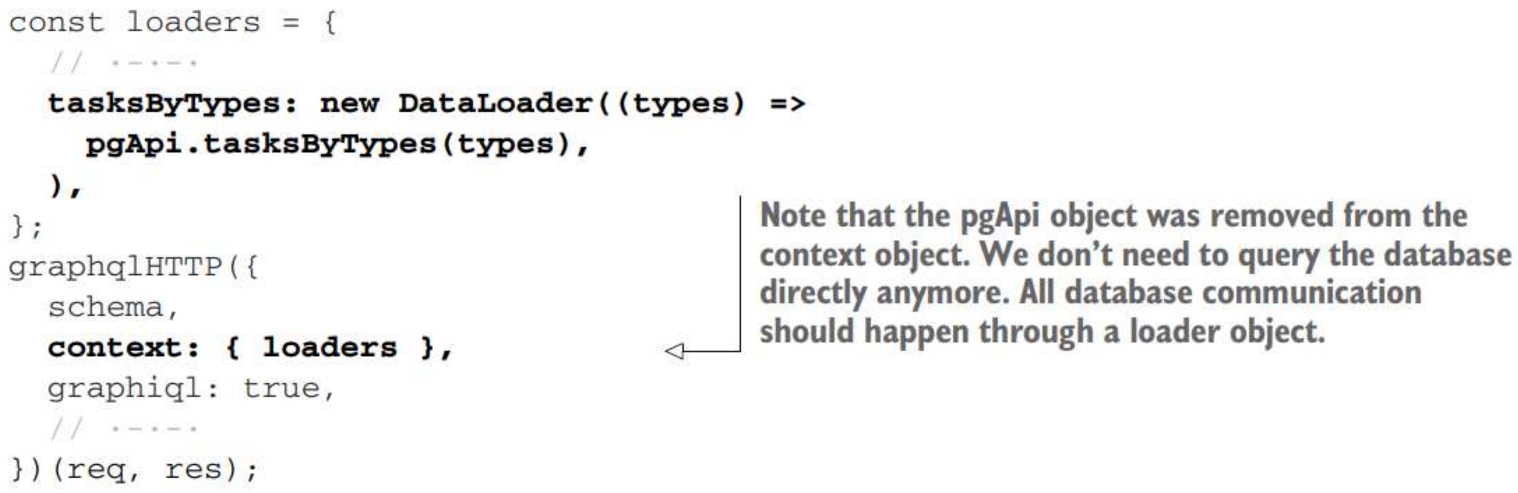
The taskMainList field



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

The taskMainList field

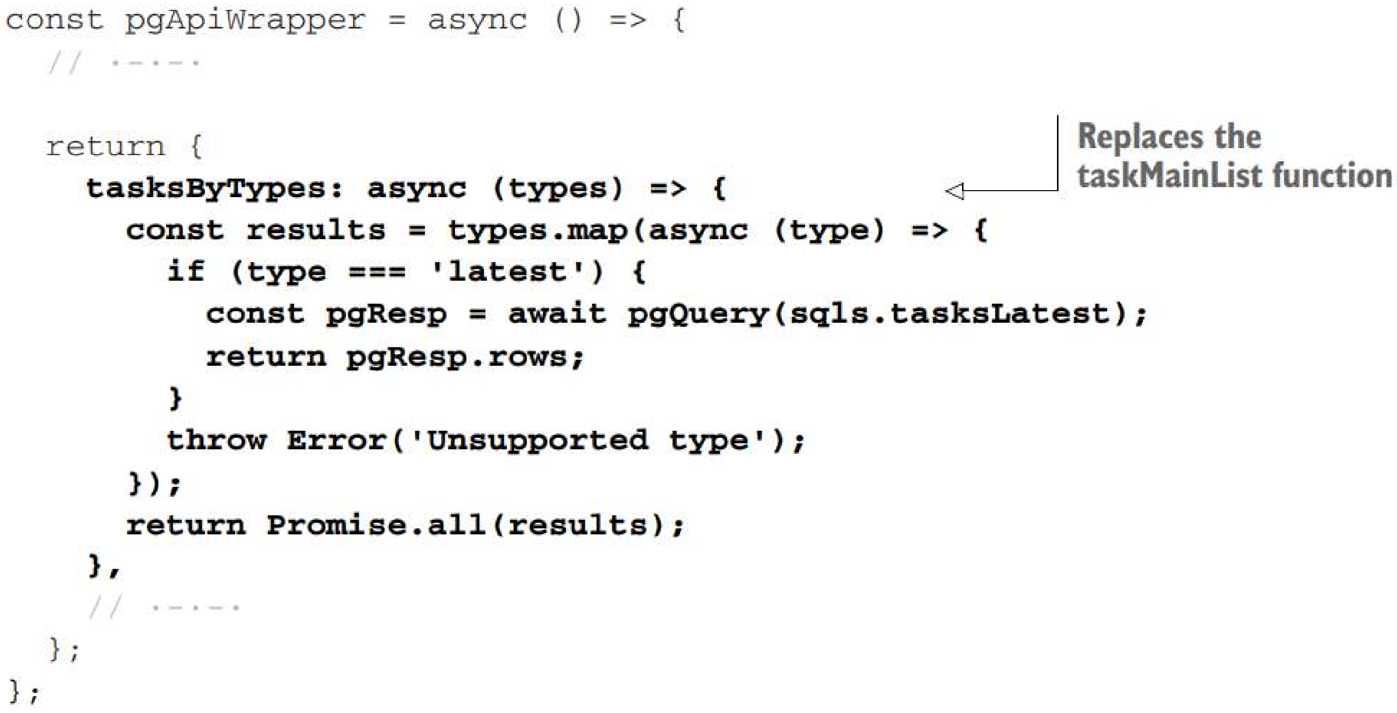
- Let’s now write the tasksByTypes loader.
- We’ll need to add it to the listener function (in api/src/server.js)



```
const loaders = {
  // ·-·-·
  tasksByTypes: new DataLoader((types) =>
    pgApi.tasksByTypes(types),
  ),
};
graphqlHTTP({
  schema,
  context: { loaders },
  graphiql: true,
  // ·-·-·
})(req, res);
```

Note that the pgApi object was removed from the context object. We don’t need to query the database directly anymore. All database communication should happen through a loader object.

The taskMainList field

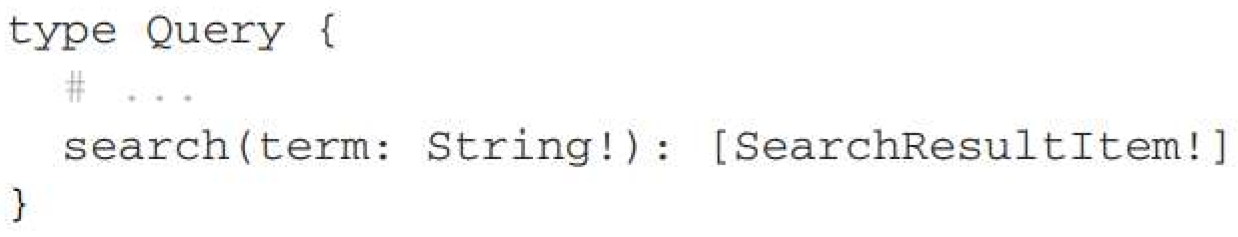


```
const pgApiWrapper = async () => {
  // ·-·-·

  return {
    tasksByTypes: async (types) => {
      const results = types.map(async (type) => {
        if (type === 'latest') {
          const pgResp = await pgQuery(sqls.tasksLatest);
          return pgResp.rows;
        }
        throw Error('Unsupported type');
      });
      return Promise.all(results);
    },
    // ·-·-·
  };
};
```

Replaces the taskMainList function

The search field

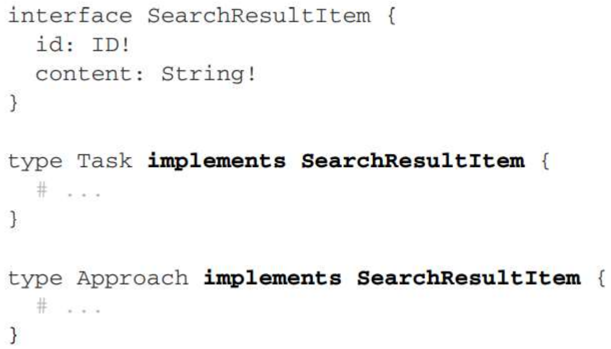
- The search field takes an argument—the search term—and returns a list of matching records from both the Task and Approach models through the interface type they implement: SearchResultItem.



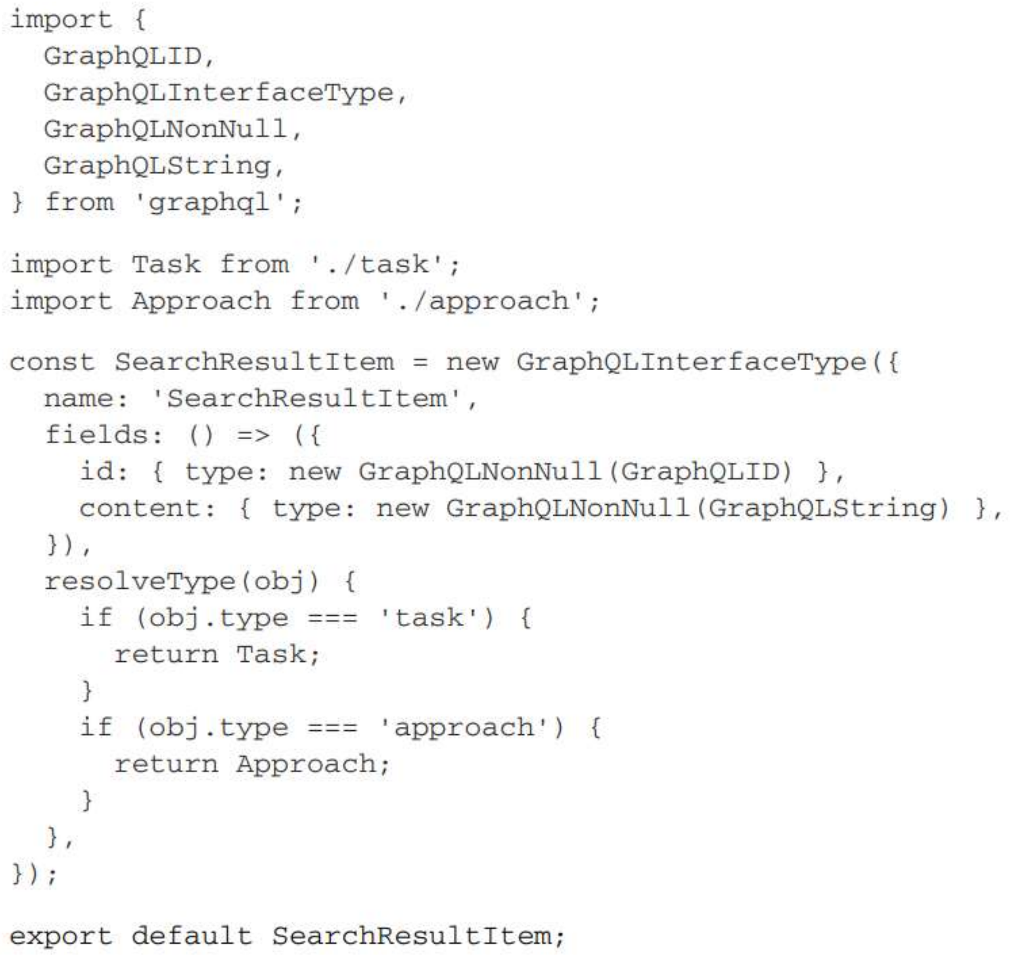
```
type Query {
  # ...
  search(term: String!): [SearchResultItem!]
}
```

The search field

- The search feature has a new concept that we’re going to implement for the first time: the GraphQL interface type.
- Here are the parts of the schema related to it.

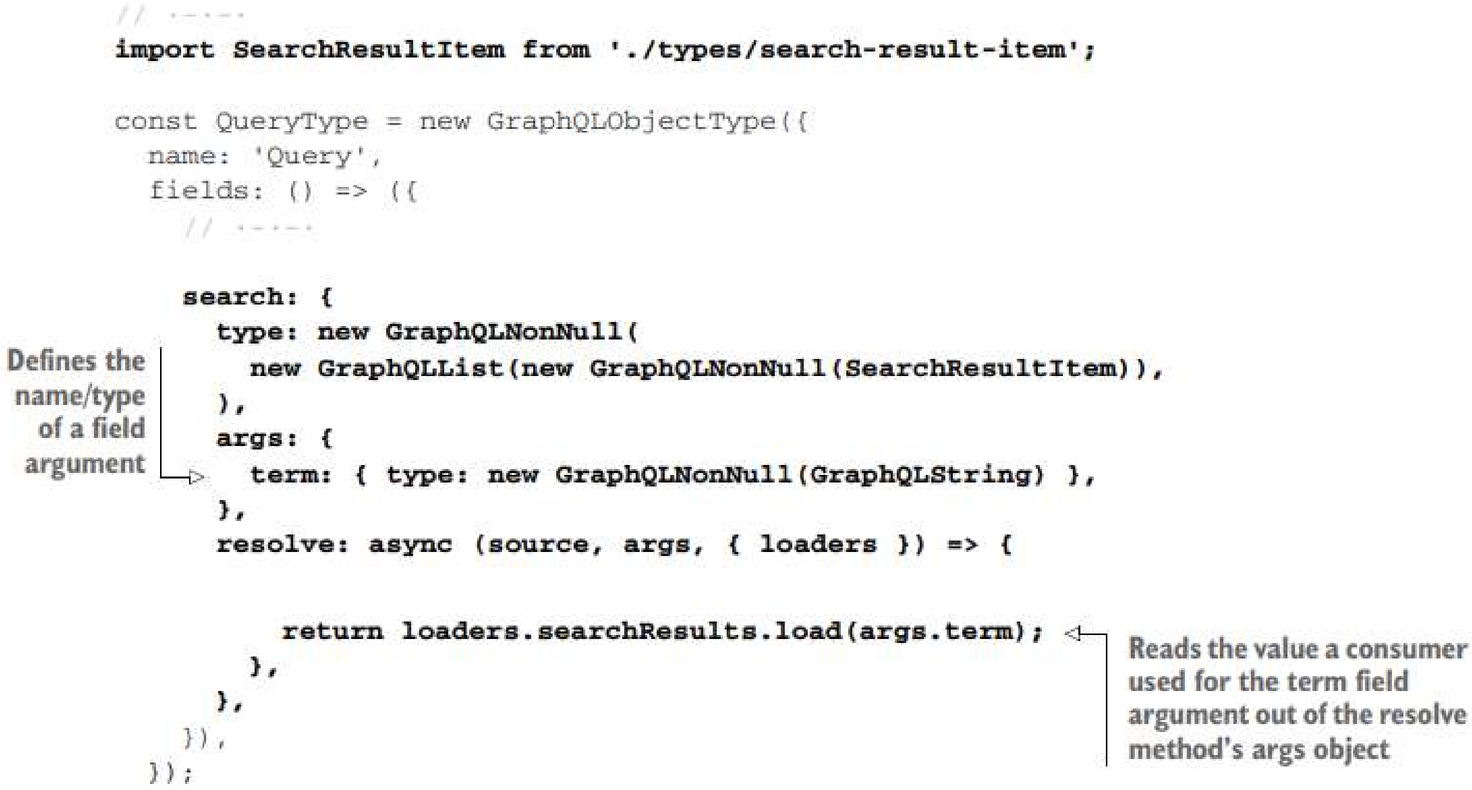


```
interface SearchResultItem {
  id: ID!
  content: String!
}

type Task implements SearchResultItem {
  # ...
}

type Approach implements SearchResultItem {
  # ...
}
```



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

import Task from './task';
import Approach from './approach';

const SearchResultItem = new GraphQLInterfaceType({
  name: 'SearchResultItem',
  fields: () => ({
    id: { type: new GraphQLNonNull(GraphQLID) },
    content: { type: new GraphQLNonNull(GraphQLString) },
  }),
  resolveType(obj) {
    if (obj.type === 'task') {
      return Task;
    }
    if (obj.type === 'approach') {
      return Approach;
    }
  },
});

export default SearchResultItem;
```

- Here’s one possible implementation of the field



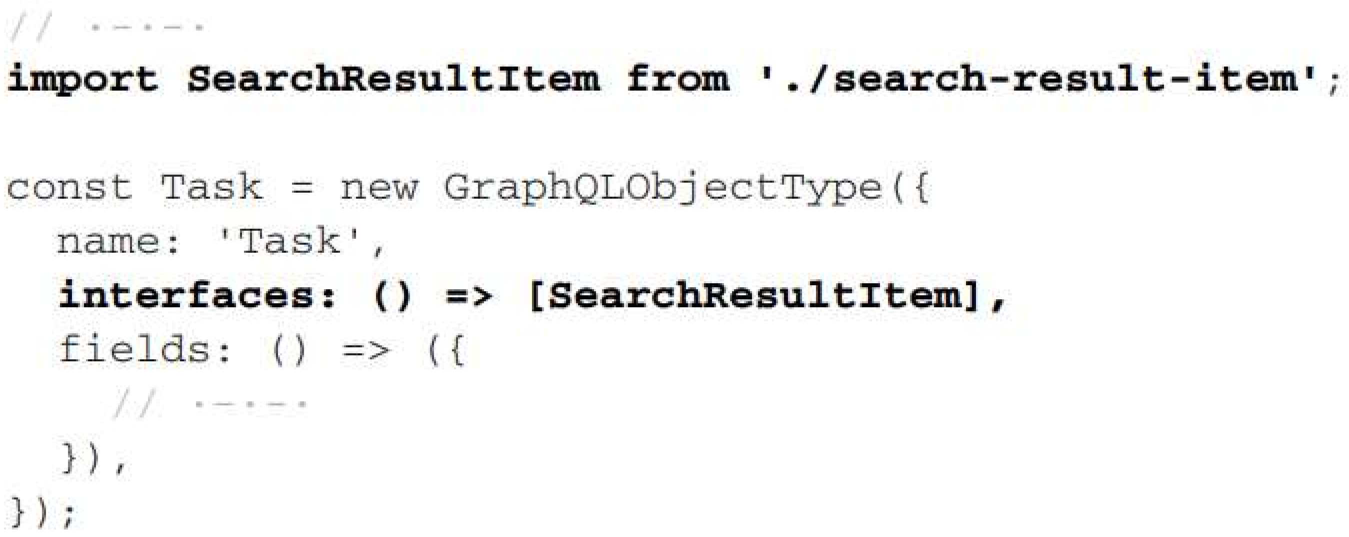
```
// ·-·-·
import SearchResultItem from './types/search-result-item';

const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    // ·-·-·

    search: {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(SearchResultItem)),
      ),
      args: {
        term: { type: new GraphQLNonNull(GraphQLString) },
      },
      resolve: async (source, args, { loaders }) => {

          return loaders.searchResults.load(args.term);
        },
      },
    }),
  });
```

Defines the name/type of a field argument

Reads the value a consumer used for the term field argument out of the resolve method’s args object

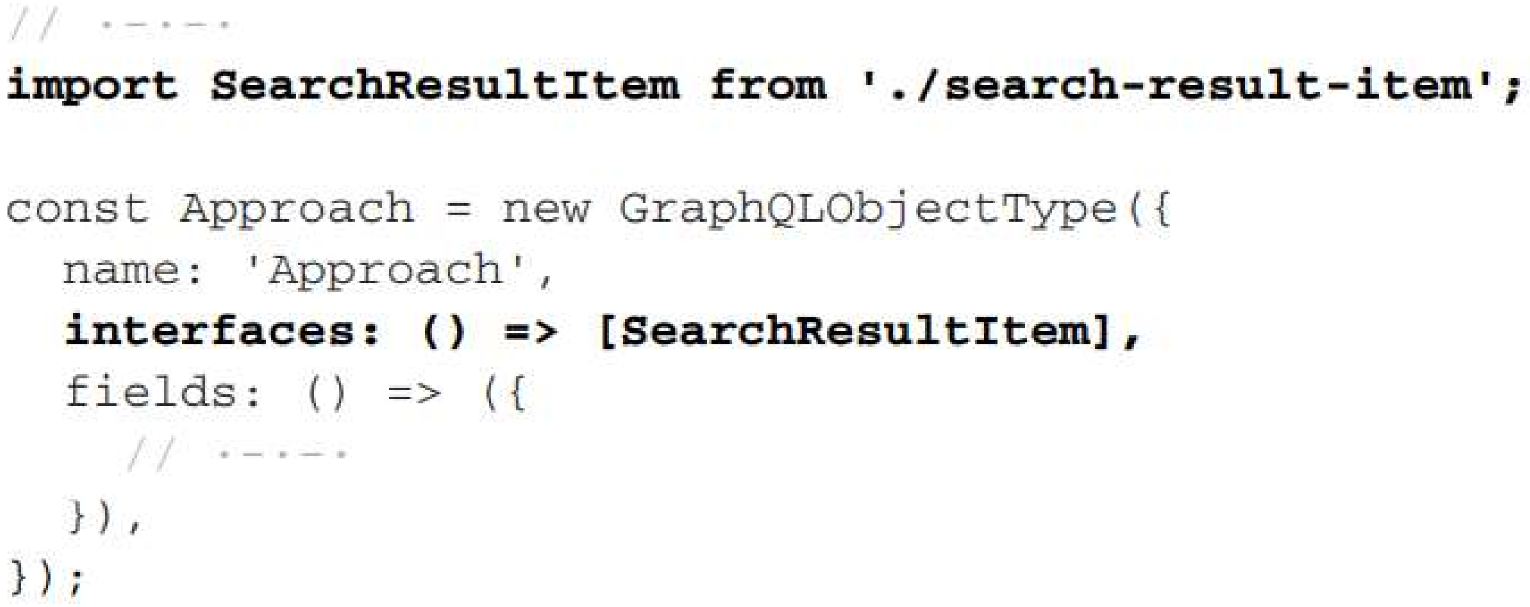
The search field

- The value of this property is an array of all the interface types an object type implements.



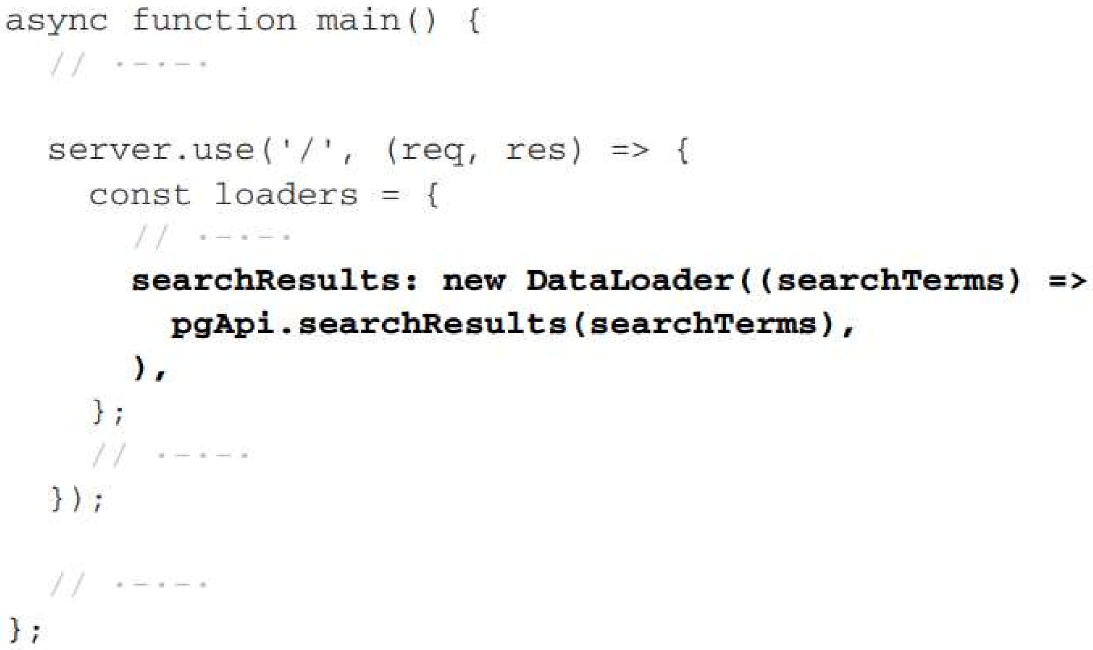
```
// ...
import SearchResultItem from './search-result-item';

const Task = new GraphQLObjectType({
  name: 'Task',
  interfaces: () => [SearchResultItem],
  fields: () => ({
    // ...
  }),
});
```

The search field

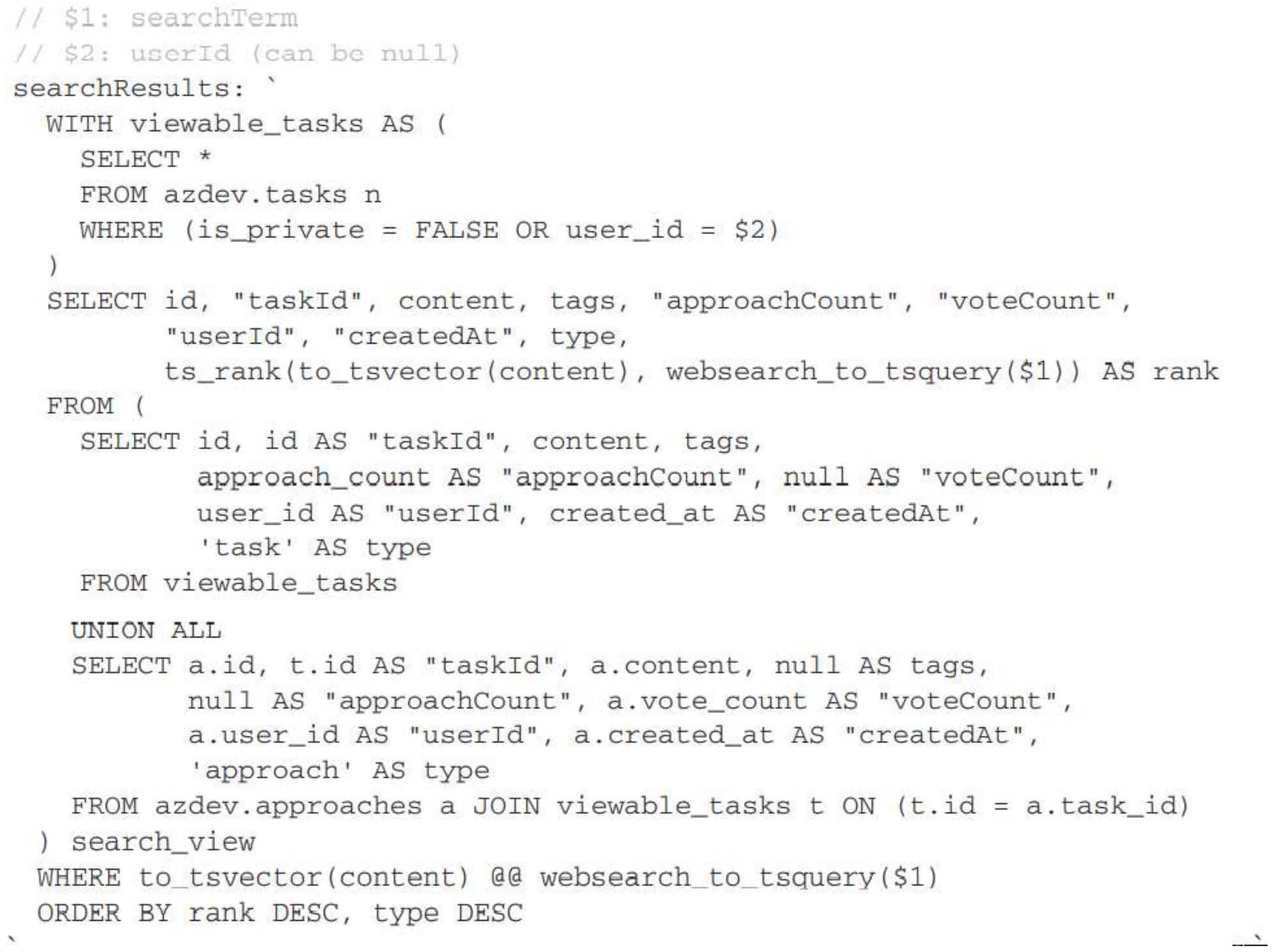


```
// ...
import SearchResultItem from './search-result-item';

const Approach = new GraphQLObjectType({
  name: 'Approach',
  interfaces: () => [SearchResultItem],
  fields: () => ({
    // ...
  }),
});
```

- Let’s assume that the pgApi module has a searchResults method to do the SQL communication.
- Here’s what I came up with for the loader definition.



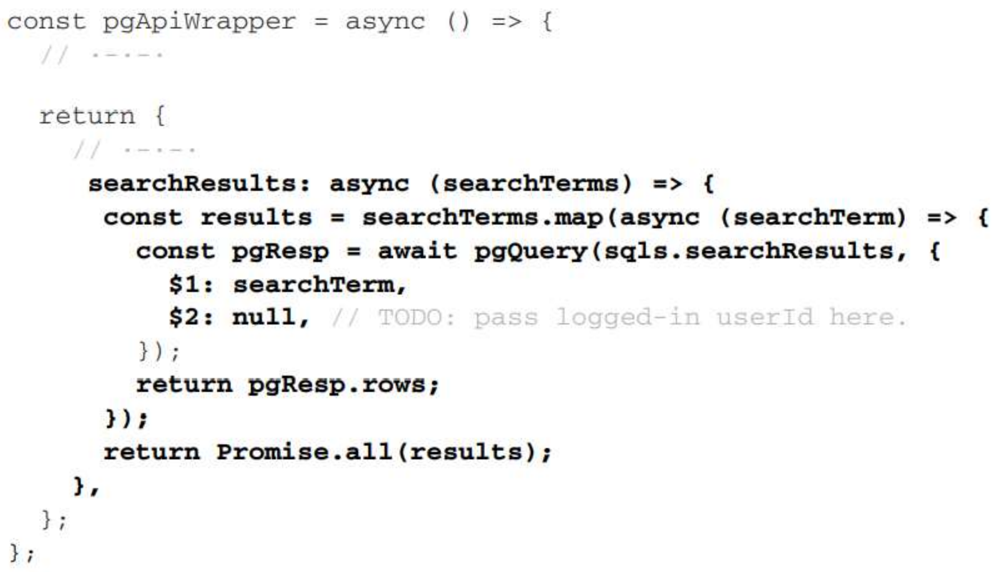
```
async function main() {
  // ·-·-·

  server.use('/', (req, res) => {
    const loaders = {
      // ·-·-·
      searchResults: new DataLoader((searchTerms) =>
        pgApi.searchResults(searchTerms),
      ),
    };
    // ·-·-·
  });

  // ·-·-·
};
```



```
// $1: searchTerm
// $2: userId (can be null)
searchResults: `
  WITH viewable_tasks AS (
    SELECT *
    FROM azdev.tasks n
    WHERE (is_private = FALSE OR user_id = $2)
  )
  SELECT id, "taskId", content, tags, "approachCount", "voteCount",
         "userId", "createdAt", type,
         ts_rank(to_tsvector(content), websearch_to_tsquery($1)) AS rank
  FROM (
    SELECT id, id AS "taskId", content, tags,
           approach_count AS "approachCount", null AS "voteCount",
           user_id AS "userId", created_at AS "createdAt",
           'task' AS type
    FROM viewable_tasks
   UNION ALL
   SELECT a.id, t.id AS "taskId", a.content, null AS tags,
          null AS "approachCount", a.vote_count AS "voteCount",
          a.user_id AS "userId", a.created_at AS "createdAt",
          'approach' AS type
   FROM azdev.approaches a JOIN viewable_tasks t ON (t.id = a.task_id)
 ) search_view
 WHERE to_tsvector(content) @@ websearch_to_tsquery($1)
 ORDER BY rank DESC, type DESC
`
```

The search field

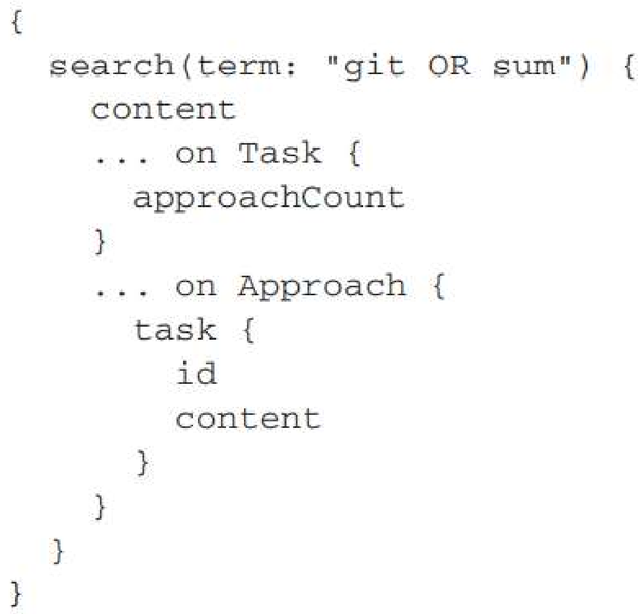
- Here’s how we can use this SQL statement in the pgApi module.



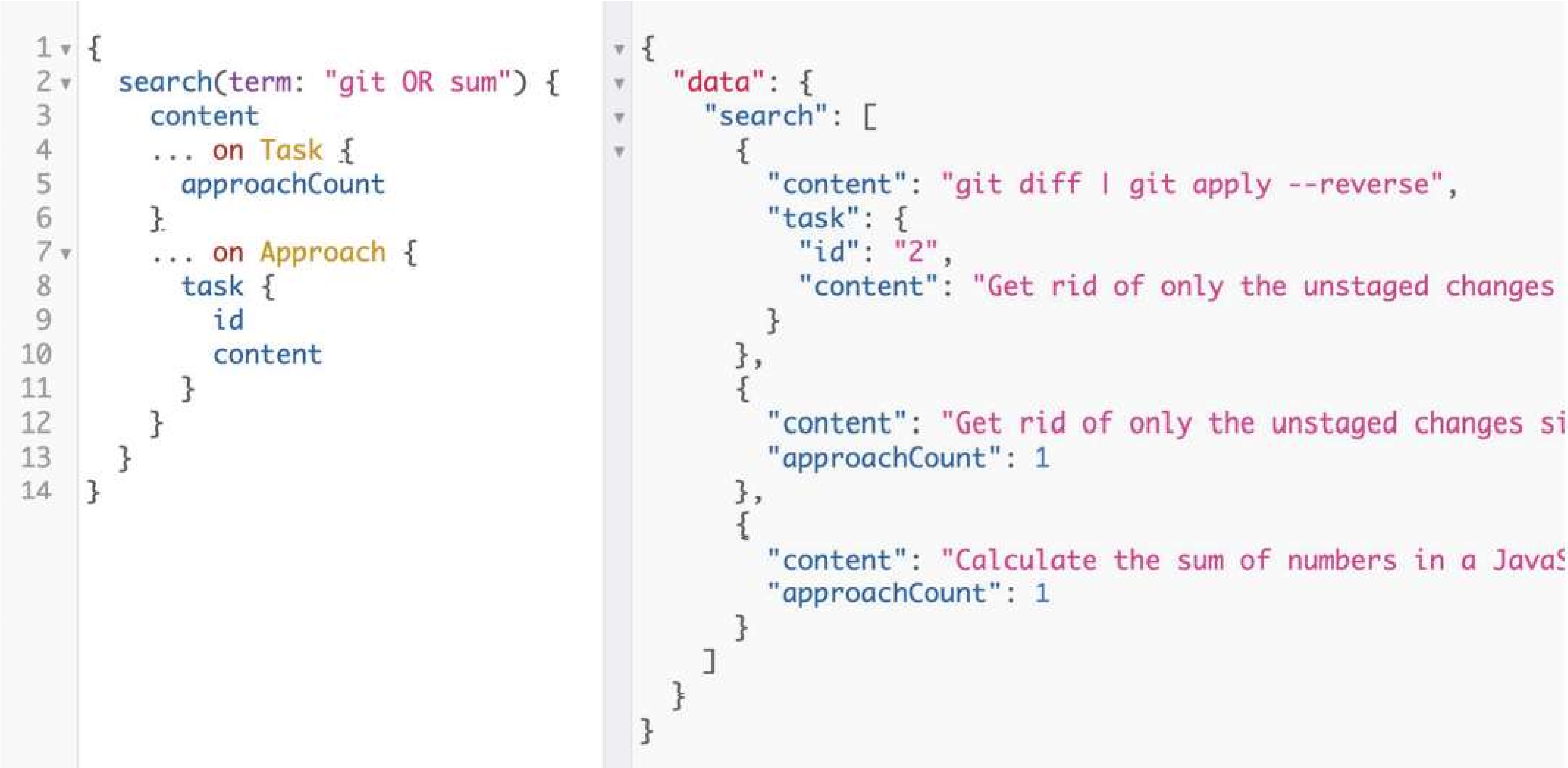
```
const pgApiWrapper = async () => {
  // ·-·-·

  return {
    // ·-·-·
     searchResults: async (searchTerms) => {
      const results = searchTerms.map(async (searchTerm) => {
        const pgResp = await pgQuery(sqls.searchResults, {
          $1: searchTerm,
          $2: null, // TODO: pass logged-in userId here.
        });
        return pgResp.rows;
      });
      return Promise.all(results);
    },
  };
};
```

The search field

- We can test now! Here’s an example of how to query the new search field in GraphQL.



```
{
  search(term: "git OR sum") {
    content
    ... on Task {
      approachCount
    }
    ... on Approach {
      task {
        id
        content
      }
    }
  }
}
```



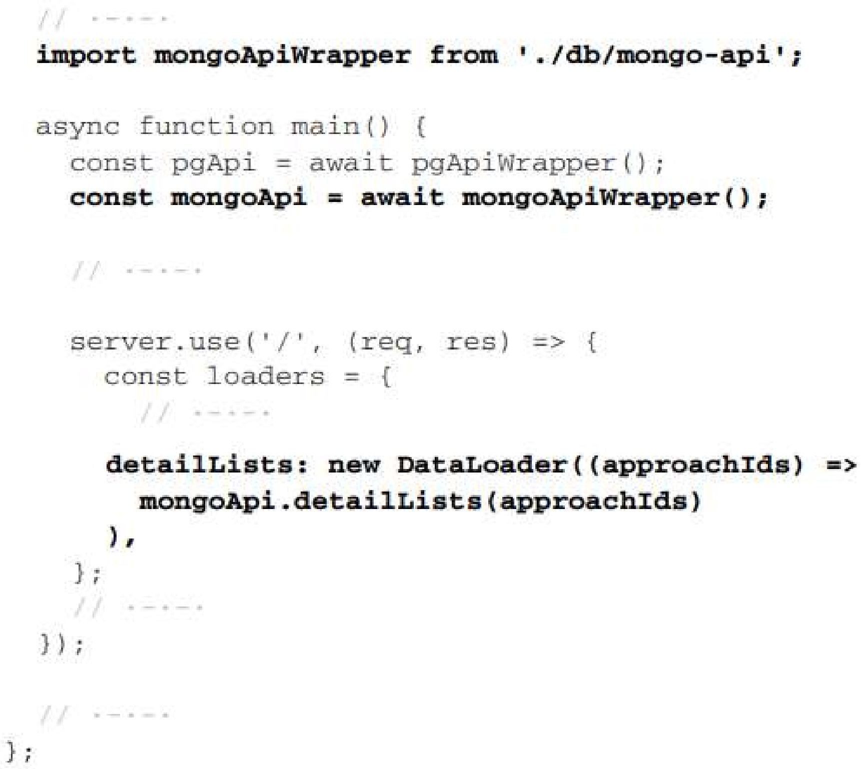
```graphql
{
  search(term: "git OR sum") {
    content
    ... on Task {
      approachCount
    }
    ... on Approach {
      task {
        id
        content
      }
    }
  }
}
```

```json
{
  "data": {
    "search": [
      {
        "content": "git diff | git apply --reverse",
        "task": {
          "id": "2",
          "content": "Get rid of only the unstaged changes
        }
      },
      {
        "content": "Get rid of only the unstaged changes si
        "approachCount": 1
      },
      {
        "content": "Calculate the sum of numbers in a JavaS
        "approachCount": 1
      }
    ]
  }
}
```

Using DataLoader with MongoDB

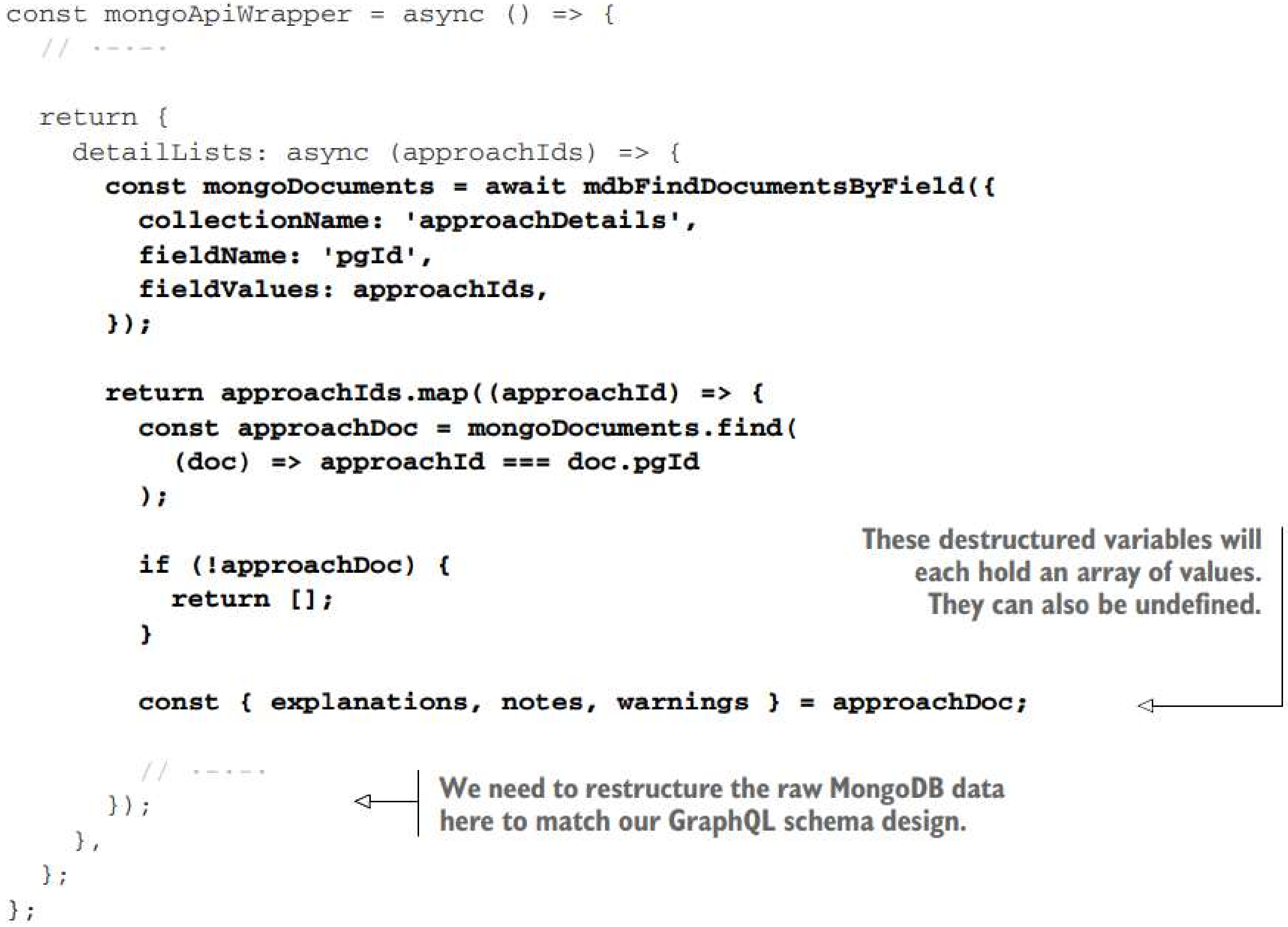
- Let’s do this one with a top-down approach as well.
- Similar to how we named objects for PostgreSQL and where we stored its modules
- let’s come up with a mongoApi module and assume that it has a batch-loading function named detailLists to load a list of Detail objects given a list of Approach IDs.

Using DataLoader with MongoDB



```
 // ·-·-·
 import mongoApiWrapper from './db/mongo-api';

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

   // ·-·-·

   server.use('/', (req, res) => {
     const loaders = {
       // ·-·-·
     detailLists: new DataLoader((approachIds) =>
       mongoApi.detailLists(approachIds)
     ),
   };
   // ·-·-·
 });

 // ·-·-·
};
```

```
import mongoClient from './mongo-client';

const mongoApiWrapper = async () => {
  const { mdb } = await mongoClient();

  const mdbFindDocumentsByField = ({
    collectionName,
    fieldName,
    fieldValues,
  }) =>
    mdb
      .collection(collectionName)
      .find({ [fieldName]: { $in: fieldValues } })
      .toArray();

  return {
    detailLists: async (approachIds) => {
      // TODO: Use mdbFindDocumentsByField to
      // implement the batch-loading logic here
    },
  };
};

export default mongoApiWrapper;
```



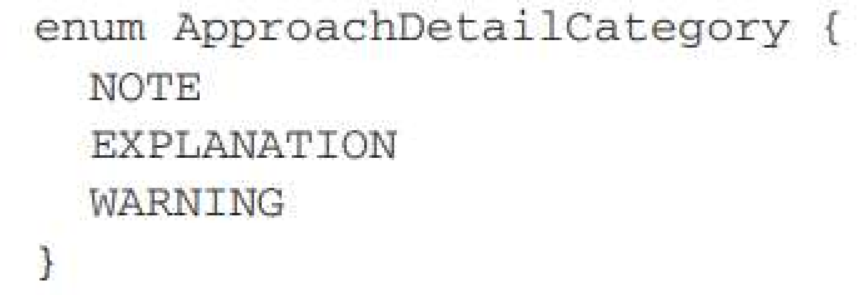
```
const mongoApiWrapper = async () => {
  // ·-·-·

  return {
    detailLists: async (approachIds) => {
      const mongoDocuments = await mdbFindDocumentsByField({
        collectionName: 'approachDetails',
        fieldName: 'pgId',
        fieldValues: approachIds,
      });

      return approachIds.map((approachId) => {
        const approachDoc = mongoDocuments.find(
          (doc) => approachId === doc.pgId
        );

        if (!approachDoc) {
          return [];
        }

        const { explanations, notes, warnings } = approachDoc;

        // ·-·-·
      });
    },
  };
};
```

These destructured variables will each hold an array of values. They can also be undefined.

We need to restructure the raw MongoDB data here to match our GraphQL schema design.

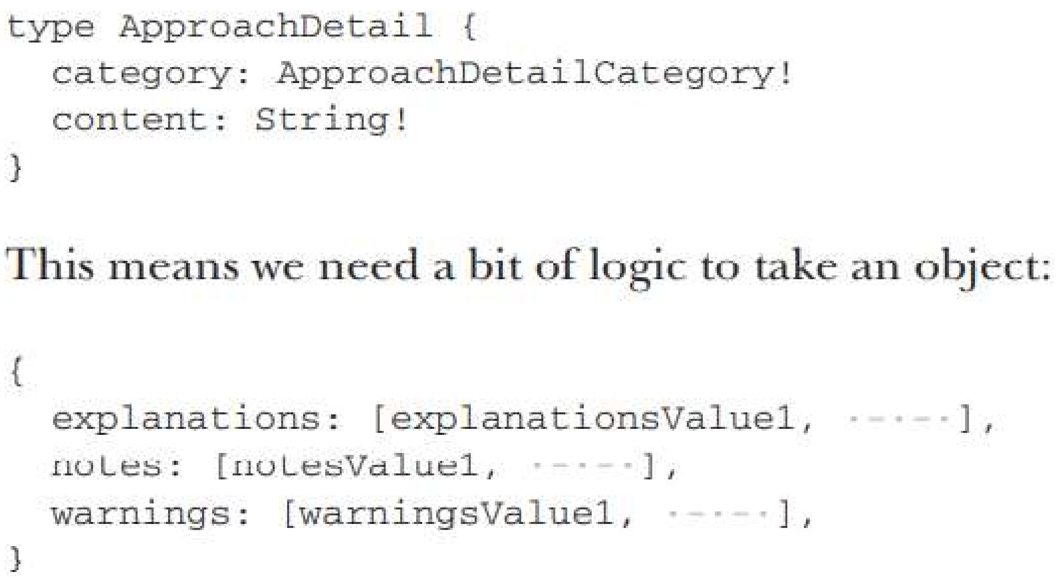
Using DataLoader with MongoDB

- Once the ID-to-document map is finished, each approachDetails document in MongoDB is an object whose properties represent the three content categories that we designed for the ApproachDetail ENUM type.



```
enum ApproachDetailCategory {
  NOTE
  EXPLANATION
  WARNING
}
```

Using DataLoader with MongoDB



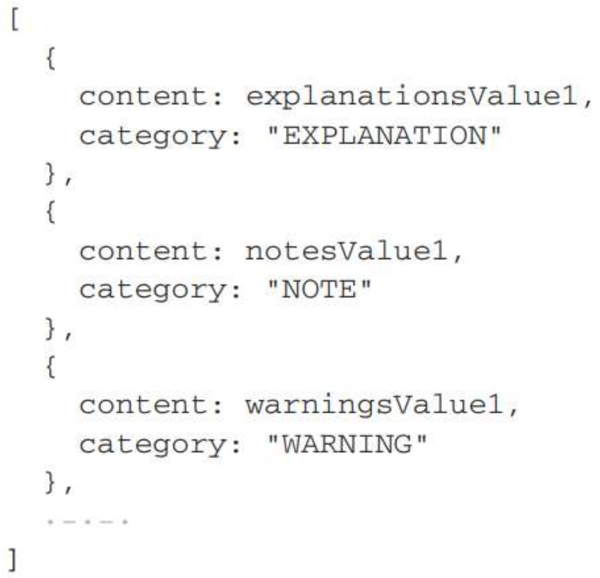
```
type ApproachDetail {
  category: ApproachDetailCategory!
  content: String!
}
```

This means we need a bit of logic to take an object:

```
{
  explanations: [explanationsValue1, ·-·-·],
  notes: [notesValue1, ·-·-·],
  warnings: [warningsValue1, ·-·-·],
}
```

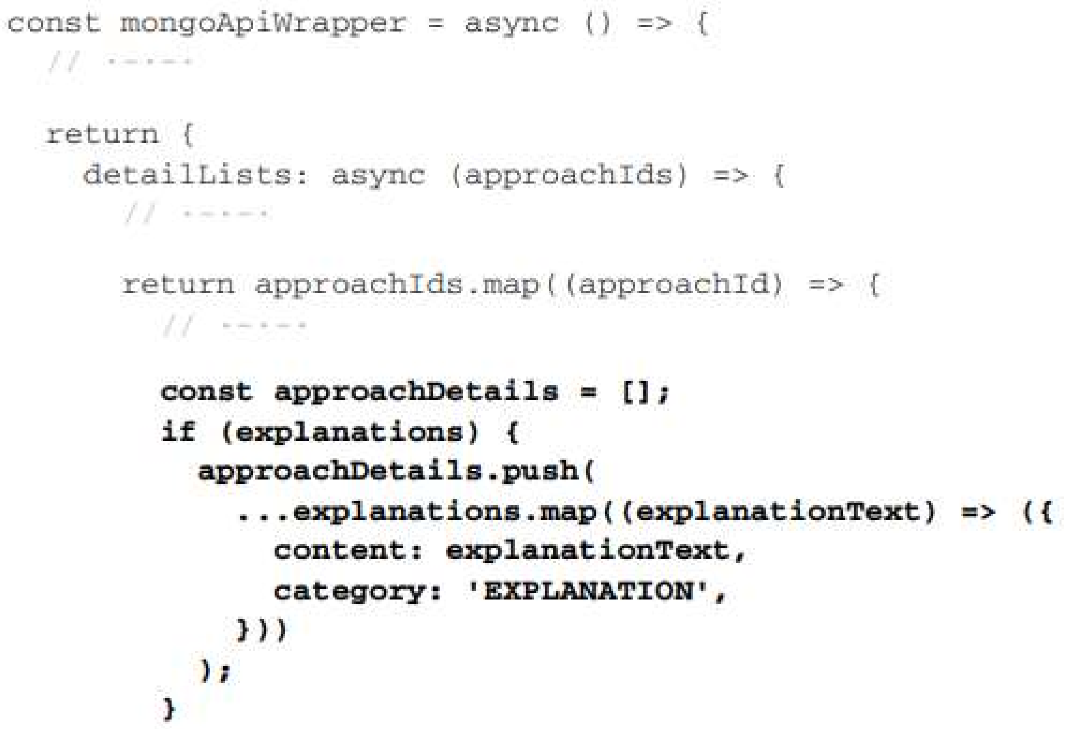
Using DataLoader with MongoDB

- And we convert the object to the following:

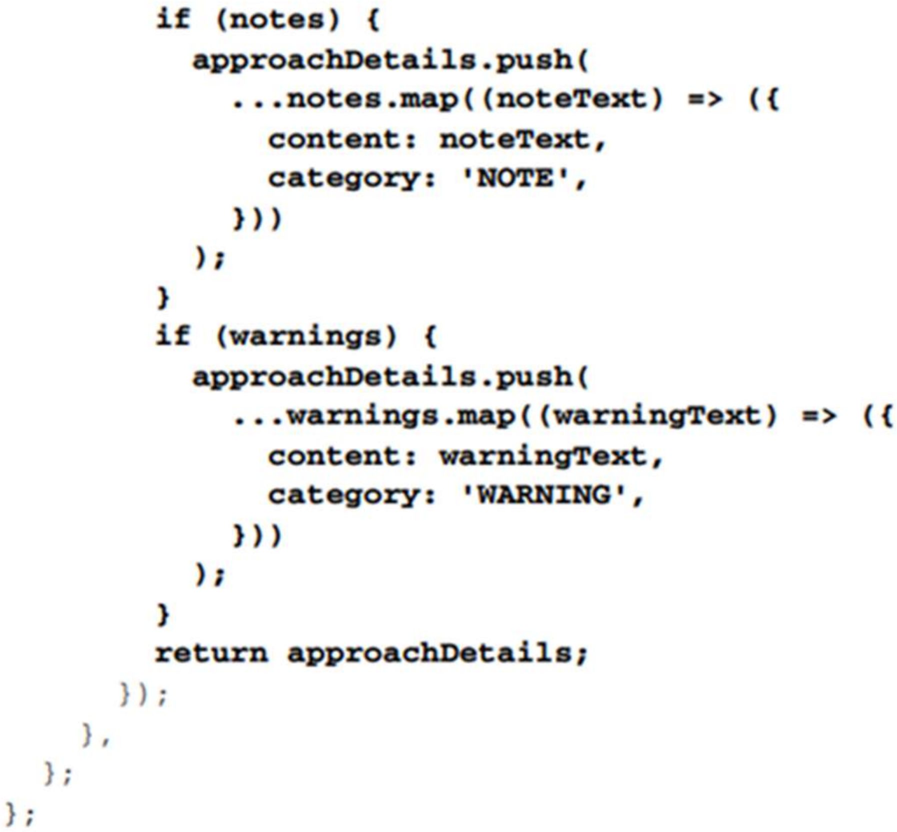


```
[
  {
    content: explanationsValue1,
    category: "EXPLANATION"
  },
  {
    content: notesValue1,
    category: "NOTE"
  },
  {
    content: warningsValue1,
    category: "WARNING"
  },
  .-.-.
]
```

Using DataLoader with MongoDB

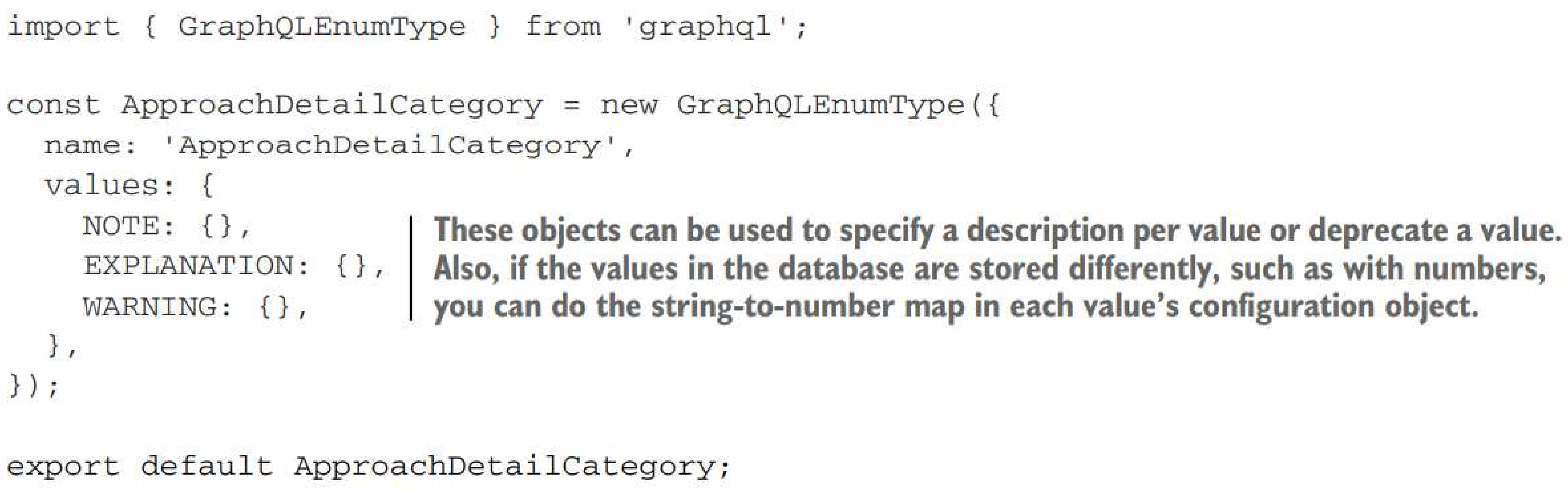


```
const mongoApiWrapper = async () => {
  // ·-·-·

  return {
    detailLists: async (approachIds) => {
      // ·-·-·

      return approachIds.map((approachId) => {
        // ·-·-·

        const approachDetails = [];
        if (explanations) {
          approachDetails.push(
            ...explanations.map((explanationText) => ({
              content: explanationText,
              category: 'EXPLANATION',
            }))
          );
        }
```



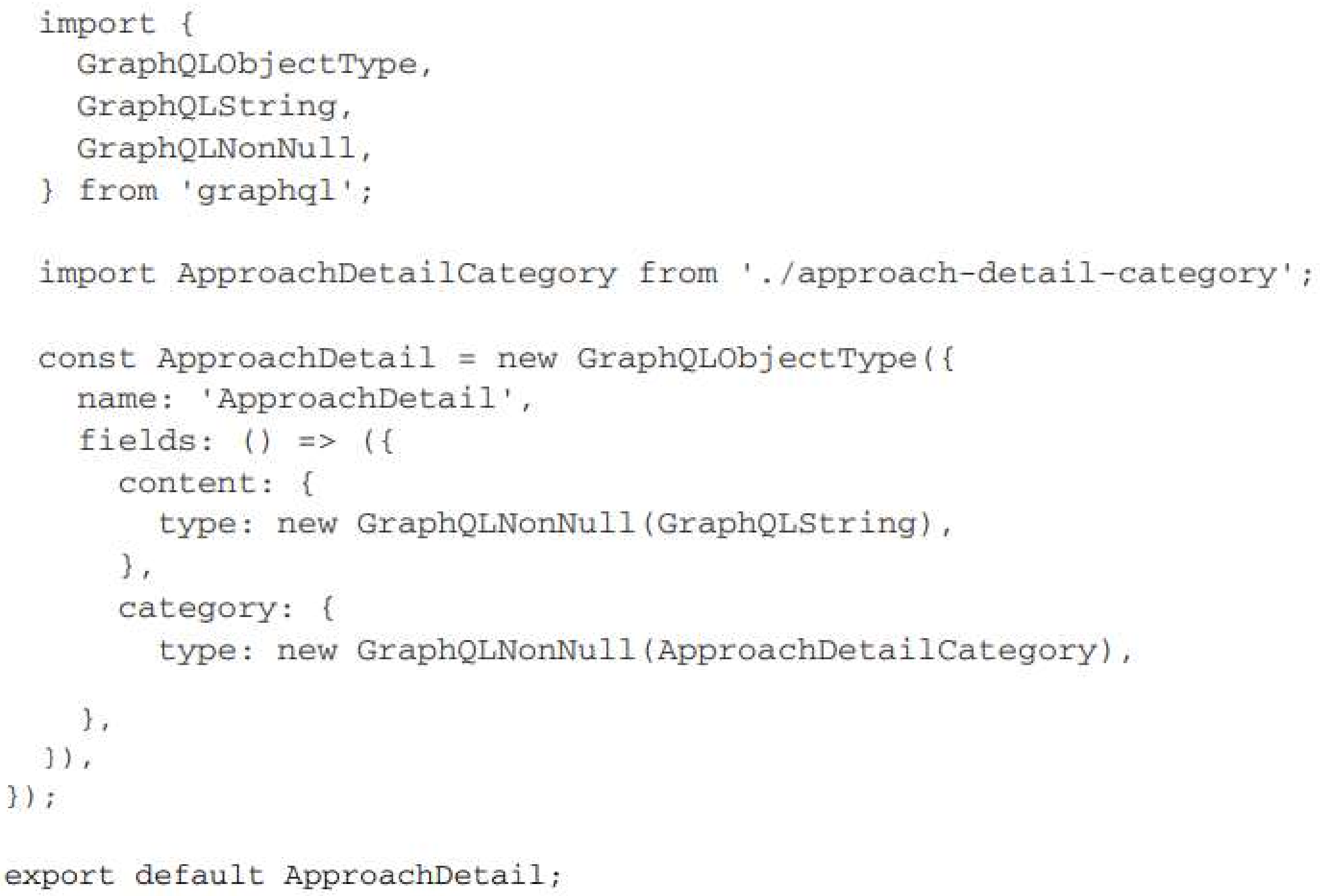
```
        if (notes) {
          approachDetails.push(
            ...notes.map((noteText) => ({
              content: noteText,
              category: 'NOTE',
            }))
          );
        }
        if (warnings) {
          approachDetails.push(
            ...warnings.map((warningText) => ({
              content: warningText,
              category: 'WARNING',
            }))
          );
        }
        return approachDetails;
      });
    },
  };
};
```

Using DataLoader with MongoDB



```
import { GraphQLEnumType } from 'graphql';

const ApproachDetailCategory = new GraphQLEnumType({
  name: 'ApproachDetailCategory',
  values: {
    NOTE: {},
    EXPLANATION: {},
    WARNING: {},
  },
});

export default ApproachDetailCategory;
```

These objects can be used to specify a description per value or deprecate a value. Also, if the values in the database are stored differently, such as with numbers, you can do the string-to-number map in each value's configuration object.

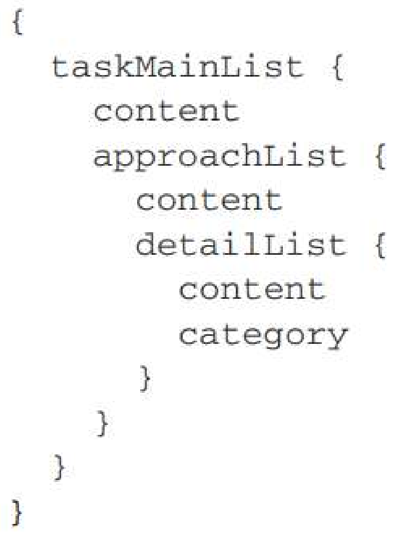


```
  import {
    GraphQLObjectType,
    GraphQLString,
    GraphQLNonNull,
  } from 'graphql';

  import ApproachDetailCategory from './approach-detail-category';

  const ApproachDetail = new GraphQLObjectType({
    name: 'ApproachDetail',
    fields: () => ({
      content: {
        type: new GraphQLNonNull(GraphQLString),
      },
      category: {
        type: new GraphQLNonNull(ApproachDetailCategory),

    },
  }),
});

export default ApproachDetail;
```

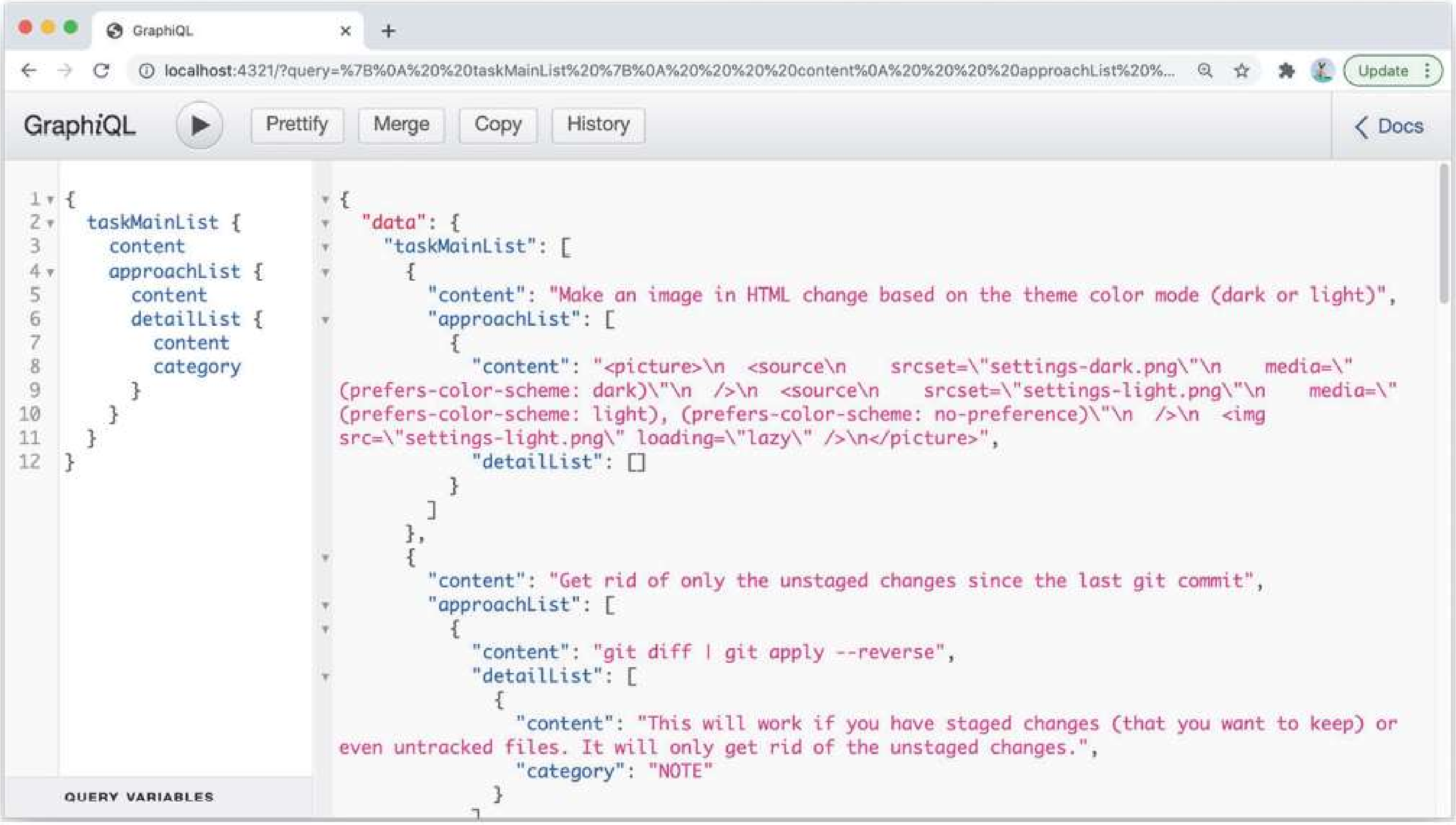
```
import {
  // ·-·-·
  GraphQLList,
} from 'graphql';
// ·-·-·
import ApproachDetail from './approach-detail';

const Approach = new GraphQLObjectType({
  name: 'Approach',
  fields: () => ({
    // ·-·-·

    detailList: {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(ApproachDetail))
      ),
      resolve: (source, args, { loaders }) =>
        loaders.detailLists.load(source.id),
    },
  },
});
```

Using DataLoader with MongoDB

- You can test this new feature using the following query(next slide figure)



```
{
  taskMainList {
    content
    approachList {
      content
      detailList {
        content
        category
      }
    }
  }
}
```


GraphiQL
localhost:4321/?query=%7B%0A%20%20taskMainList%20%7B%0A%20%20%20%20content%0A%20%20%20%20approachList%20%...
Update
GraphiQL
Prettify
Merge
Copy
History
Docs
QUERY VARIABLES

```
{
  taskMainList {
    content
    approachList {
      content
      detailList {
        content
        category
      }
    }
  }
}
```

```
{
  "data": {
    "taskMainList": [
      {
        "content": "Make an image in HTML change based on the theme color mode (dark or light)",
        "approachList": [
          {
            "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>",
            "detailList": []
          }
        ]
      },
      {
        "content": "Get rid of only the unstaged changes since the last git commit",
        "approachList": [
          {
            "content": "git diff | git apply --reverse",
            "detailList": [
              {
                "content": "This will work if you have staged changes (that you want to keep) or even untracked files. It will only get rid of the unstaged changes.",
                "category": "NOTE"
              }
            ]
```

Summary

- To optimize data-fetching operations in a generic, scalable way, you can use the concepts of caching and batching.
- You can cache SQL responses based on unique values like IDs or any other custom unique values you design in your API service.
- You can also delay asking the database about a specific resource until you figure out all the unique IDs of all the records needed from that resource and then send a single request to the database to include all the records based on all the IDs.

"Complete Lab"