
Optimizing data fetching
LEARNING VOYAGE

Table of Contents

Caching and batching

```
{
  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)
);
```

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

- For example, if the getUsersByIds batch function is given the input array of IDs [2, 5, 3, 1], the function needs to issue one SQL statement to fetch all user records for those IDs.
- 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 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

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

Defining and using a DataLoader instance

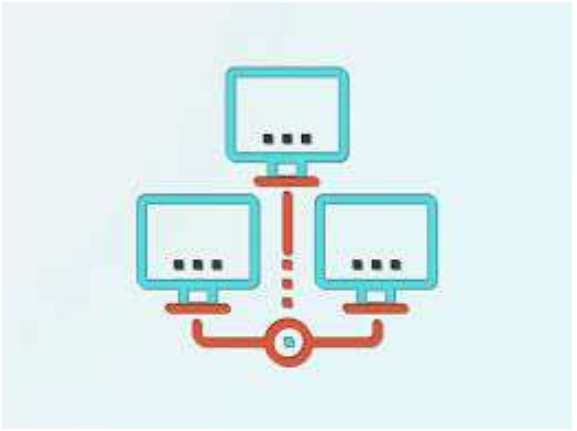
```
// ·-·-·
import DataLoader from 'dataloader';

async function main() {
  // ·-·-·

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

```
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

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 PostgreSQL logs:

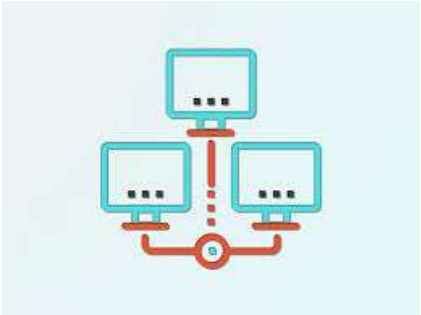
```
LOG:  statement: SELECT ... FROM azdev.tasks
WHERE ...
LOG:  execute <unnamed>: SELECT ... FROM
azdev.users WHERE id = ANY ($1)
DETAIL:  parameters: $1 = '{1}'
```

The loader for the approachList field

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

  return {
    // ....
    approachLists: async (taskIds) => {                          1
      const pgResp = await pgQuery(sqls.approachesForTaskIds, {
        $1: taskIds,                                             2
      });
      return taskIds.map((taskId) =>
        pgResp.rows.filter((row) => taskId === row.taskId),      3
      );
    },
  };
};
```

The loader for the approachList field

```
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

```
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}'
```

The loader for the approachList field

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

Table of Contents

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
  // ·-·-·
}
```

- Query that we can use to work through this field.

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

Single resource fields

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

Single resource fields

Single resource fields

```
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);
      },
    }
```

Continued code

```
    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);
      },
    },
  },
});
```

Single resource fields

Single resource fields

- I find it helpful to think about the new objects and functions I need and use them before I write them.
- This approach helps me come up with better, more practical designs. 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.js.

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

```
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;
```

Single resource fields

Table of Contents

```
// ....
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;
```

Circular dependencies in GraphQL types

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),
    },
  }),
});
```

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
                                                            }
```

Table of Contents

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

Using DataLoader with custom IDs for caching

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

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

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

The taskMainList field

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

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

type Task implements SearchResultItem {
  # ...
}

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

The search field

```
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;
```

The search 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) },          1
      },
      resolve: async (source, args, { loaders }) => {
        return loaders.searchResults.load(args.term);               2
      },
    },
  }),
});
```

The search field

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

The search field

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

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

  // ·-·-·
};
```

The search field

```
// $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
`,
```

```
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

The search field

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

The search field

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


```
{
  "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
      }
    ]
  }
}
```

Table of Contents

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

  // ·-·-·
};
```

Using DataLoader with MongoDB

```
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;
```

Using DataLoader with MongoDB

```
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;

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

Using DataLoader with MongoDB

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

- Each of these properties holds an array of text values. However, remember that we designed the ApproachDetail type to have a category field and a content field.

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

- We convert the object to the following:

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

Using DataLoader with MongoDB

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

Using DataLoader with MongoDB

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

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

export default ApproachDetailCategory;
```

Using DataLoader with MongoDB

```
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;
```

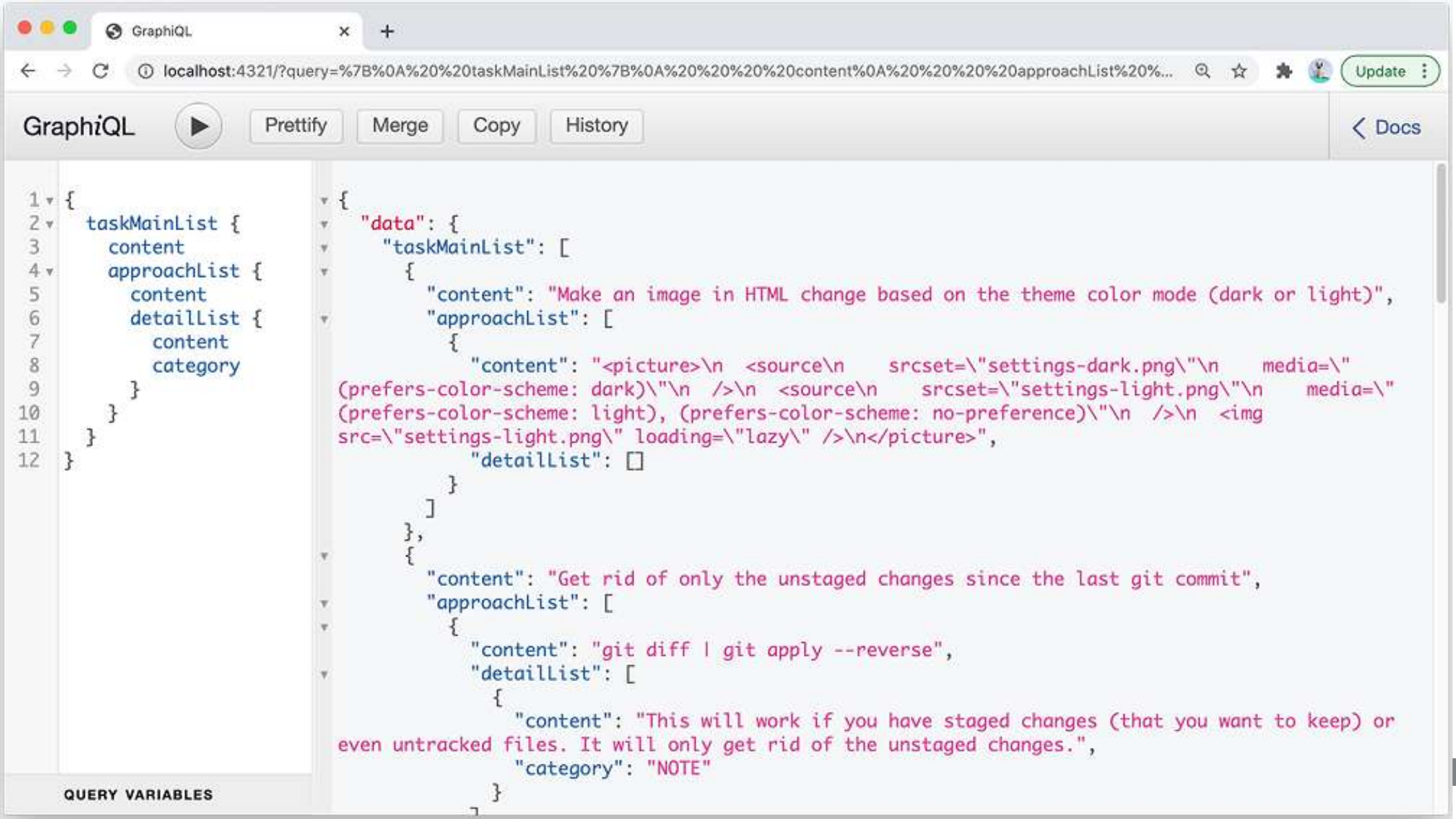
Using DataLoader with MongoDB

```
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),
    },
  },
});
```

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

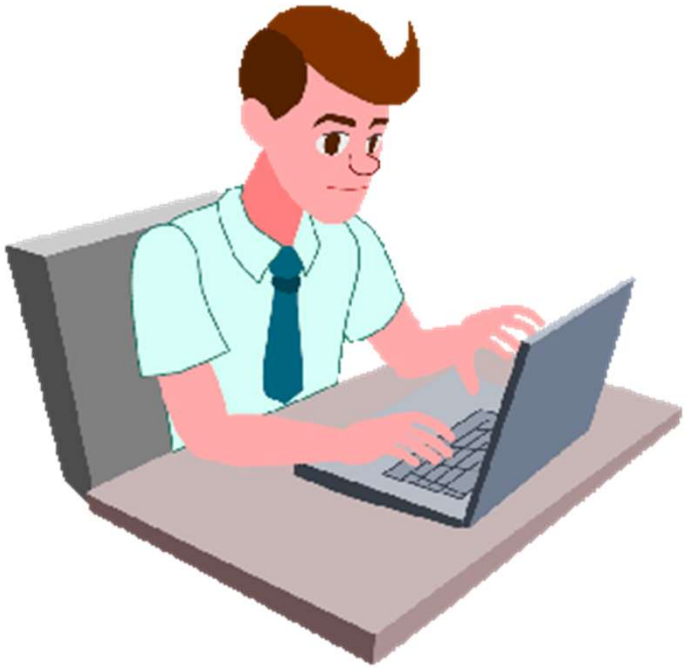
```
{
  taskMainList {
    content
    approachList {
      content
      detailList {
        content
        category
      }
    }
  }
}
```

Using DataLoader with MongoDB


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
{
taskMainList {
content
approachList {
content
detailList {
content
category
}
}
}
}
QUERY VARIABLES
{
"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.



"Complete Lab"