Implementing mutations

Implementing mutations

This lesson covers

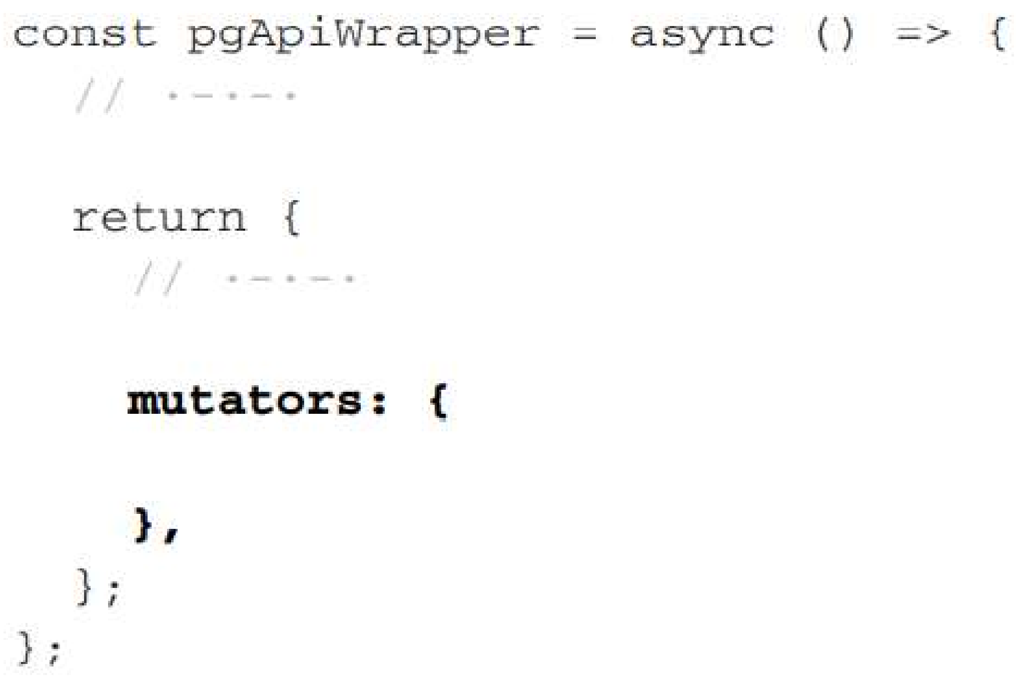
- Implementing GraphQL’s mutation fields
- Authenticating users for mutation and query operations
- Creating custom, user-friendly error messages
- Using powerful database features to optimize mutations

The mutators context object

- We’ve abstracted all database READ operations to go through DataLoader instances using the loaders object we passed to each resolver as part of the global GraphQL context, It’s time to think about the WRITE operations.

- Every mutation operation will perform an INSERT, UPDATE, or DELETE SQL statement or a MongoDB operation (or a combination of them).

The mutators context object



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

  return {
    // ·-·-·

    mutators: {

    },
  };
};
```

The mutators context object

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

  return {
    // ·-·-·

    mutators: {

    },
  };
};
```

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

  server.use('/', (req, res) => {
    // ·-·-·

    const mutators = {
      ...pgApi.mutators,
      ...mongoApi.mutators,
    };

    graphqlHTTP({
      schema,
      context: { loaders, mutators },
      graphiql: true,
    })(req, res);
  });

  // ·-·-·
 };
```

The Mutation type

- We do that in api/src/schema/index.js, mirroring what we already have there for QueryType.

```
import QueryType from './queries';
import MutationType from './mutations';

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

The Mutation type

- The new mutations.js file will be under api/src/schema, and it will use a regular GraphQLObjectType object.

```
import { GraphQLObjectType } from 'graphql';
const MutationType = new GraphQLObjectType({
  name: 'Mutation',
  fields: () => ({
    // ...
  }),
});

export default MutationType;
```

User mutations

- Now that we have the skeleton to support mutation operations, let’s tackle the first two mutations that will allow a consumer of this API to create an account (userCreate) and then obtain an authentication token to use other mutations (userLogin).

- Remember that for each mutation, in addition to the main mutation field, we need to define two types: an input type and a payload type.

The userCreate mutation

```
  input UserInput {
    username: String!
    password: String!

  firstName: String
  lastName: String
}

type UserError {
  message: String!
}

type UserPayload {
  errors: [UserError!]!
  user: User
  authToken: String!
}

type Mutation {
  userCreate(input: UserInput!): UserPayload!

  # More mutations
}
```

The userCreate mutation

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

const UserError = new GraphQLObjectType({
  name: 'UserError',
  fields: () => ({
    message: {
      type: new GraphQLNonNull(GraphQLString),
    },
  }),
});

export default UserError;
```

```
import {
  GraphQLObjectType,
  GraphQLString,
  GraphQLNonNull,
  GraphQLList,

  } from 'graphql';

  import User from './user';
  import UserError from './user-error';

  const UserPayload = new GraphQLObjectType({
    name: 'UserPayload',
    fields: () => ({
      errors: {
        type: new GraphQLNonNull(
          new GraphQLList(new GraphQLNonNull(UserError)),
        ),
      },
      user: { type: User },
      authToken: { type: GraphQLString },
    }),
  });

  export default UserPayload;
```

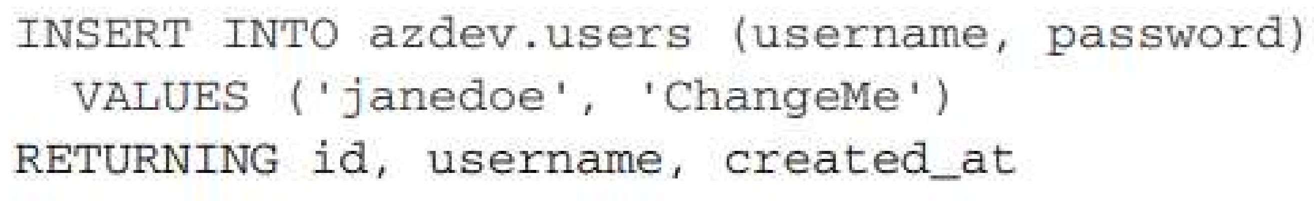
```
import {
  GraphQLInputObjectType,
  GraphQLString,
  GraphQLNonNull,
} from 'graphql';

const UserInput = new GraphQLInputObjectType({
  name: 'UserInput',
  fields: () => ({
    username: { type: new GraphQLNonNull(GraphQLString) },
    password: { type: new GraphQLNonNull(GraphQLString) },
    firstName: { type: GraphQLString },
    lastName: { type: GraphQLString },
  }),
});

export default UserInput;
```

```
import { GraphQLObjectType, GraphQLNonNull } from 'graphql';
import UserPayload from './types/payload-user';
import UserInput from './types/input-user';

const MutationType = new GraphQLObjectType({
  name: 'Mutation',
  fields: () => ({
    userCreate: {
      type: new GraphQLNonNull(UserPayload),
      args: {
        input: { type: new GraphQLNonNull(UserInput) },
      },
      resolve: async (source, { input }, { mutators }) => {
        return mutators.userCreate({ input });
      },
    },
  }),
});

export default MutationType;
```

The mutators.userCreate method does not exist yet.

The userCreate mutation

- That’s when PostgreSQL will surprise you.
- We can actually tell PostgreSQL to return a newly created record using the same INSERT statement.



```
INSERT INTO azdev.users (username, password)
  VALUES ('janedoe', 'ChangeMe')
RETURNING id, username, created_at
```

```
// ·-·-·

import { randomString } from '../utils';

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

  return {
    // ·-·-·

    mutators: {
      // ·-·-·

      userCreate: async ({ input }) => {
        const payload = { errors: [] };
        if (input.password.length < 6) {
          payload.errors.push({
            message: 'Use a stronger password',
          });
        }
        if (payload.errors.length === 0) {
          const authToken = randomString();
          const pgResp = await pgQuery(sqls.userInsert, {
            $1: input.username.toLowerCase(),
            $2: input.password,
            $3: input.firstName,
            $4: input.lastName,
            $5: authToken,
          });
          if (pgResp.rows[0]) {
            payload.user = pgResp.rows[0];
            payload.authToken = authToken;
          }

        }
        return payload;
      },
    },
  };
};
```

The randomString function returns a random string. It's already implemented in api/src/utils.js.

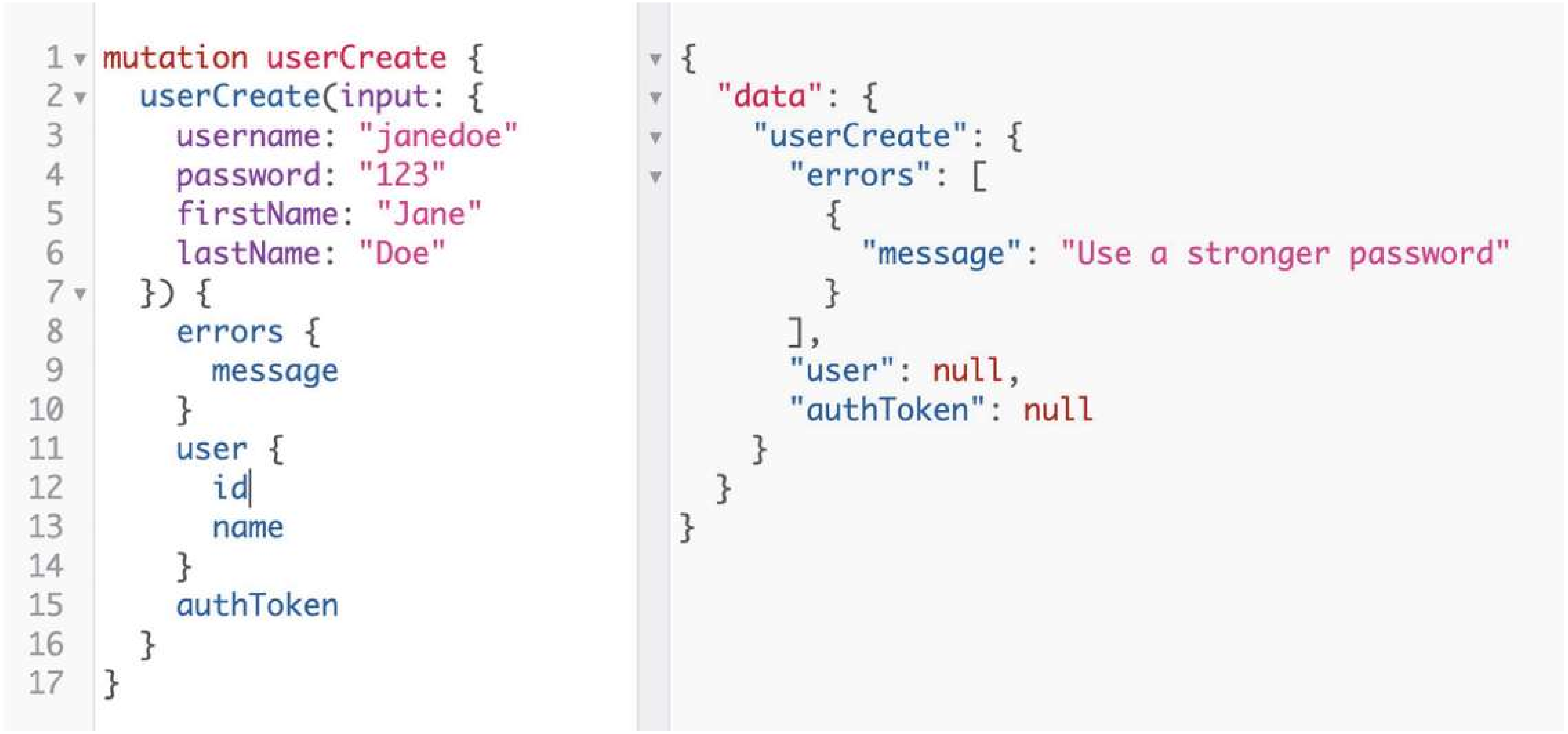
The userInsert SQL statement inserts a row into the azdev.users table. It's already implemented in api/src/db/sqls.js.

  };
};

- To test the userCreate mutation, here's a request you can use in GraphiQL.

```
mutation userCreate {
  userCreate(input: {
    username: "janedoe"
    password: "123"
    firstName: "Jane"
    lastName: "Doe"
  }) {
    errors {
      message
    }
    user {
      id
      name
    }
    authToken
  }
}
```

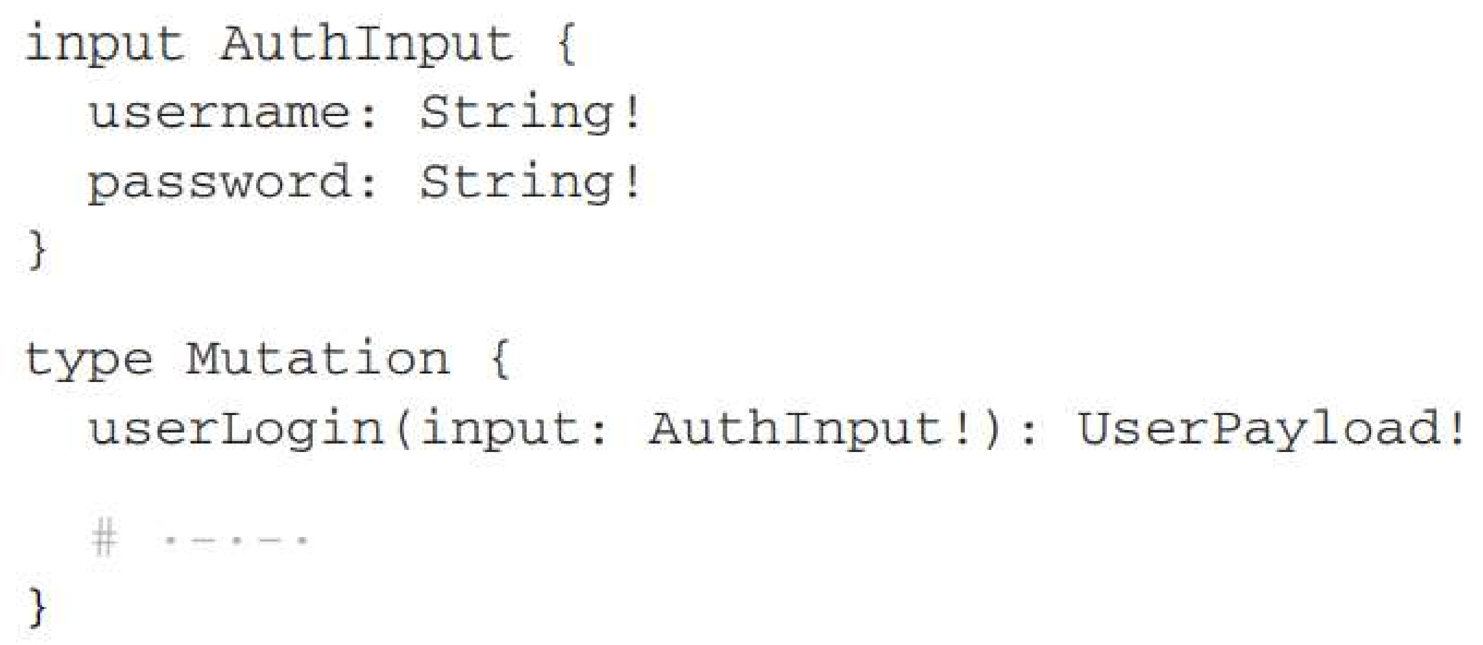
Try it first with a short password to see the UserError response, and then try it with a valid password.



```graphql
mutation userCreate {
  userCreate(input: {
    username: "janedoe"
    password: "123"
    firstName: "Jane"
    lastName: "Doe"
  }) {
    errors {
      message
    }
    user {
      id
      name
    }
    authToken
  }
}
```

```json
{
  "data": {
    "userCreate": {
      "errors": [
        {
          "message": "Use a stronger password"
        }
      ],
      "user": null,
      "authToken": null
    }
  }
}
```

The userLogin mutation



```
input AuthInput {
  username: String!
  password: String!
}

type Mutation {
  userLogin(input: AuthInput!): UserPayload!

  # ....
}
```

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

const AuthInput = new GraphQLInputObjectType({
  name: 'AuthInput',
  fields: () => ({
    username: { type: new GraphQLNonNull(GraphQLString) },
    password: { type: new GraphQLNonNull(GraphQLString) },
  }),
});

export default AuthInput;
```

```
// ·-·-·

import AuthInput from './types/input-auth';

const MutationType = new GraphQLObjectType({
  name: 'Mutation',
  fields: () => ({
    // ·-·-·
    userLogin: {
      type: new GraphQLNonNull(UserPayload),
      args: {
        input: { type: new GraphQLNonNull(AuthInput) },
      },
      resolve: async (source, { input }, { mutators }) => {
        return mutators.userLogin({ input });
      },
    },
  }),
});

export default MutationType;
```

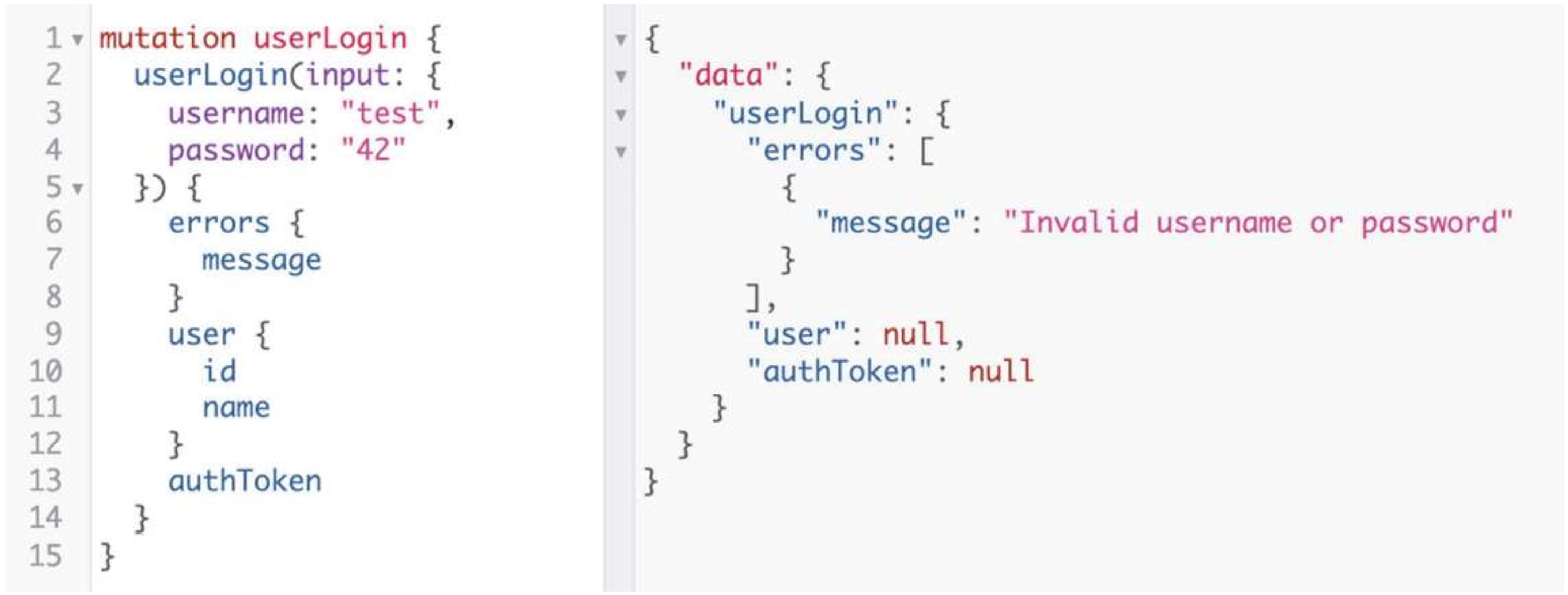
```
const pgApiWrapper = async () => {
  // ·····

  return {
    // ·····

    mutators: {
      // ·····
      userLogin: async ({ input }) => {
        const payload = { errors: [] };
        if (!input.username || !input.password) {
          payload.errors.push({
            message: 'Invalid username or password',
          });
        }
        if (payload.errors.length === 0) {
          const pgResp = await pgQuery(sqls.userFromCredentials, {
            $1: input.username.toLowerCase(),
            $2: input.password,
          });
          const user = pgResp.rows[0];
          if (user) {
            const authToken = randomString();
            await pgQuery(sqls.userUpdateAuthToken, {
              $1: user.id,
              $2: authToken,
            });
            payload.user = user;
            payload.authToken = authToken;
          } else {
            payload.errors.push({
              message: 'Invalid username or password'
            });
          }
        }
        return payload;
      },
    },
  };
};
```

The userLogin mutation

```
mutation userLogin {
  userLogin(input: {
    username: "test",
    password: "123456"
  }) {
    errors {
      message
    }
    user {
      id
      name
    }
    authToken
  }
}
```

The "test/123456" credentials are valid (from the sample development data).

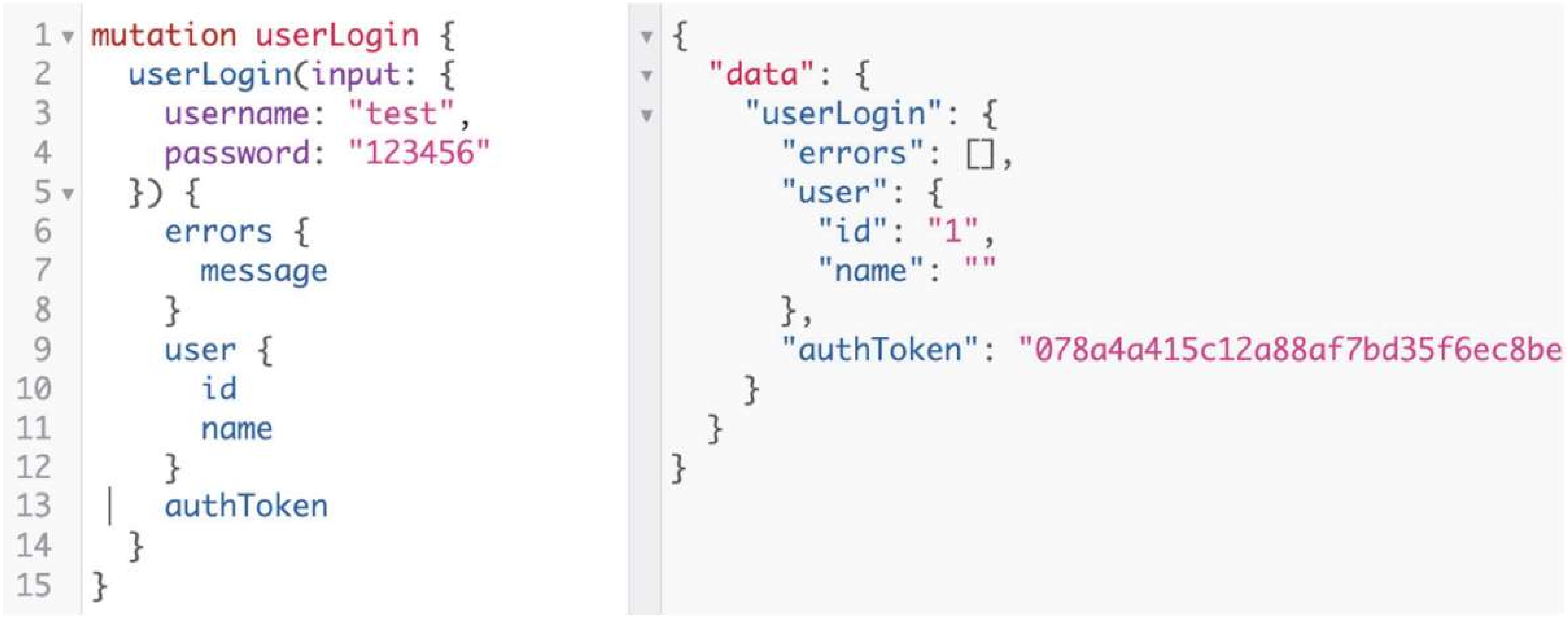
The userLogin mutation



```
mutation userLogin {
  userLogin(input: {
    username: "test",
    password: "42"
  }) {
    errors {
      message
    }
    user {
      id
      name
    }
    authToken
  }
}
```

```
{
  "data": {
    "userLogin": {
      "errors": [
        {
          "message": "Invalid username or password"
        }
      ],
      "user": null,
      "authToken": null
    }
  }
}
```

The userLogin mutation


```
mutation userLogin {
  userLogin(input: {
    username: "test",
    password: "123456"
  }) {
    errors {
      message
    }
    user {
      id
      name
    }
    authToken
  }
}
```

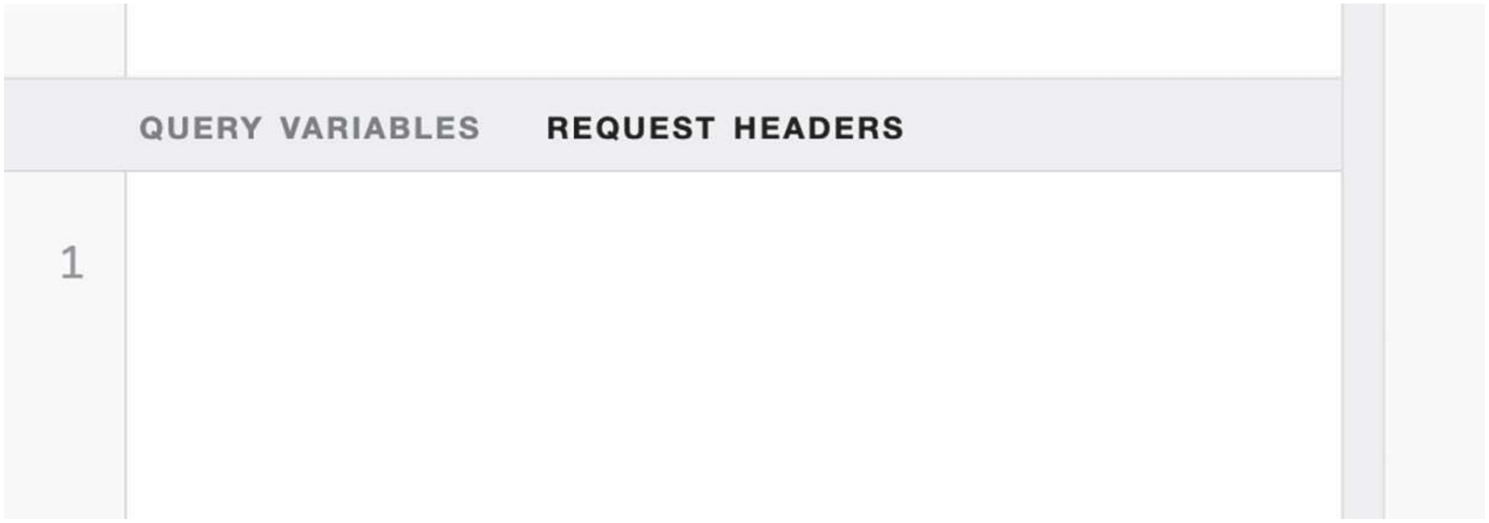
```
{
  "data": {
    "userLogin": {
      "errors": [],
      "user": {
        "id": "1",
        "name": ""
      },
      "authToken": "078a4a415c12a88af7bd35f6ec8be
    }
  }
}
```

Authenticating API consumers

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

  server.use('/', (req, res) => {
    // ·-·-·
    graphqlHTTP({
      schema,
      context: { loaders, mutators },
      graphiql: { headerEditorEnabled: true },
      // ·-·-·
    })(req, res);
  });

  // ·-·-·
}
```

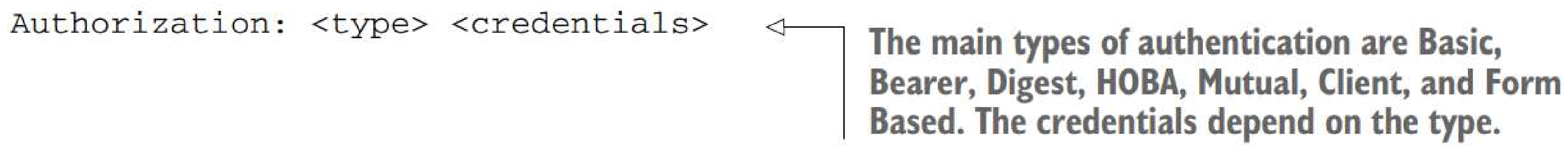
Authenticating API consumers

- The GraphiQL editor should now show the REQUEST HEADERS editor



Authenticating API consumers

- We can use the Authorization request header to include the authToken value with every request made by GraphiQL.
- The syntax for that request header is shown next.



```
Authorization: <type> <credentials>
```

The main types of authentication are Basic, Bearer, Digest, HOBA, Mutual, Client, and Form Based. The credentials depend on the type.

Authenticating API consumers

```
{
  search(term: "babel") {
    content
  }
}
```

To include the `authToken` value, put this in the request headers editor:

```
{
  "Authorization": "Bearer AUTH_TOKEN_VALUE_HERE"
}
```

Replace this with the valid authToken value you get from the userLogin mutation.

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

  return {
    userFromAuthToken: async (authToken) => {
      if (!authToken) {
        return null;
      }
      const pgResp = await pgQuery(sqls.userFromAuthToken, {
        $1: authToken,
      });
      return pgResp.rows[0];
    },

    // ·-·-·
  };
};
```

Authenticating API consumers

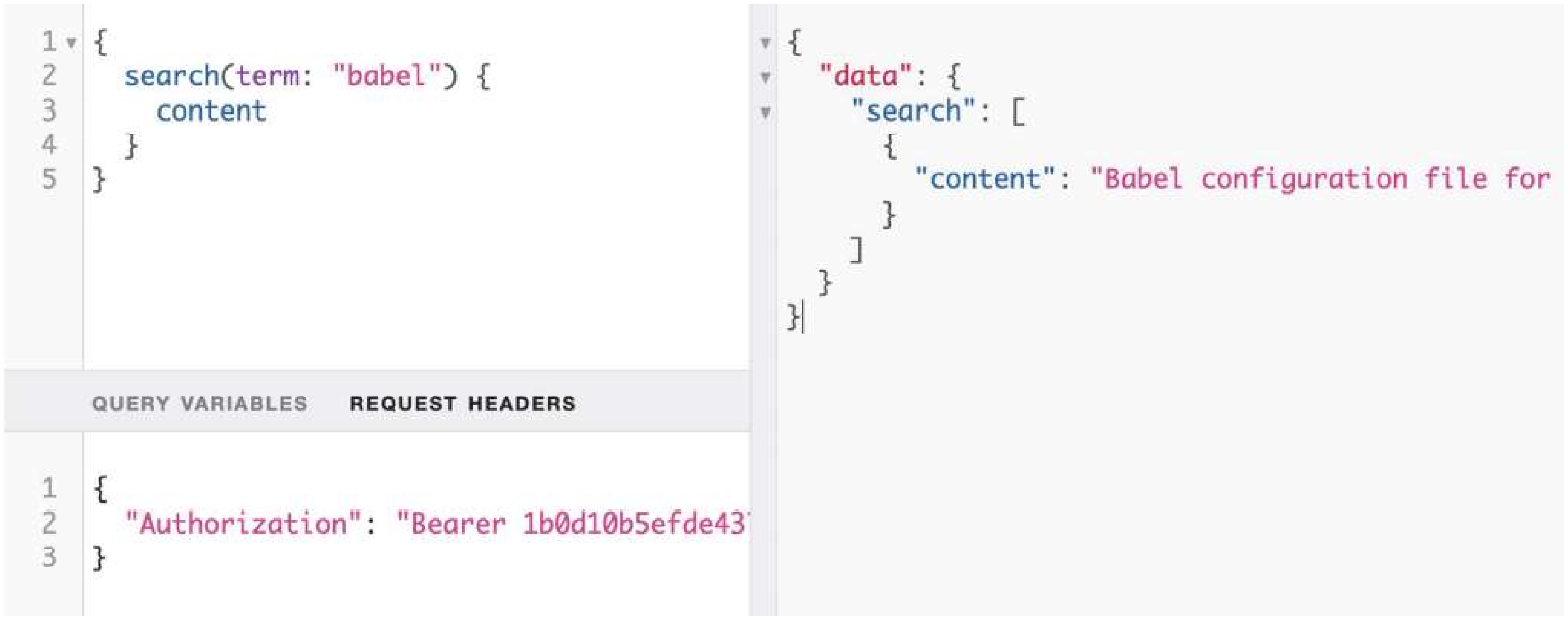
```
async function main() {
  // ·-·-·

  server.use('/', async (req, res) => {
    const authToken =
      req && req.headers && req.headers.authorization
        ? req.headers.authorization.slice(7) // "Bearer "
        : null;
    const currentUser = await pgApi.userFromAuthToken(authToken);
    if (authToken && !currentUser) {
      return res.status(401).send({
        errors: [{ message: 'Invalid access token' }],
      });
    }

    // ·-·-·
  });
```

Note the new async keyword, which is needed since the new code uses the await keyword on the userFromAuthToken async function.

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

  server.use('/', async (req, res) => {
    // ·-·-·

    const loaders = {
      users: new DataLoader((userIds) => pgApi.usersInfo(userIds)),
      approachLists: new DataLoader((taskIds) =>
        pgApi.approachLists(taskIds),
      ),
      tasks: new DataLoader((taskIds) =>
        pgApi.tasksInfo({ taskIds, currentUser }),
      ),
      tasksByTypes: new DataLoader((types) =>
        pgApi.tasksByTypes(types),
      ),
      searchResults: new DataLoader((searchTerms) =>
        pgApi.searchResults({ searchTerms, currentUser }),
      ),
      detailLists: new DataLoader((approachIds) =>
        mongoApi.detailLists(approachIds),
      ),
    };

    // ·-·-·
  });

  // ·-·-·
}
```

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

  return {
    // ·-·-·
    tasksInfo: async ({ taskIds, currentUser }) => {
      const pgResp = await pgQuery(sqls.tasksFromIds, {
        $1: taskIds,
        $2: currentUser ? currentUser.id : null,
      });
      return taskIds.map((taskId) =>
        pgResp.rows.find((row) => taskId == row.id),
      );
    },
    searchResults: async ({ searchTerms, currentUser }) => {
      const results = searchTerms.map(async (searchTerm) => {
        const pgResp = await pgQuery(sqls.searchResults, {
          $1: searchTerm,
          $2: currentUser ? currentUser.id : null,
        });
        return pgResp.rows;
      });
      return Promise.all(results);
    },

    // ·-·-·
  };
};
```

Authenticating API consumers



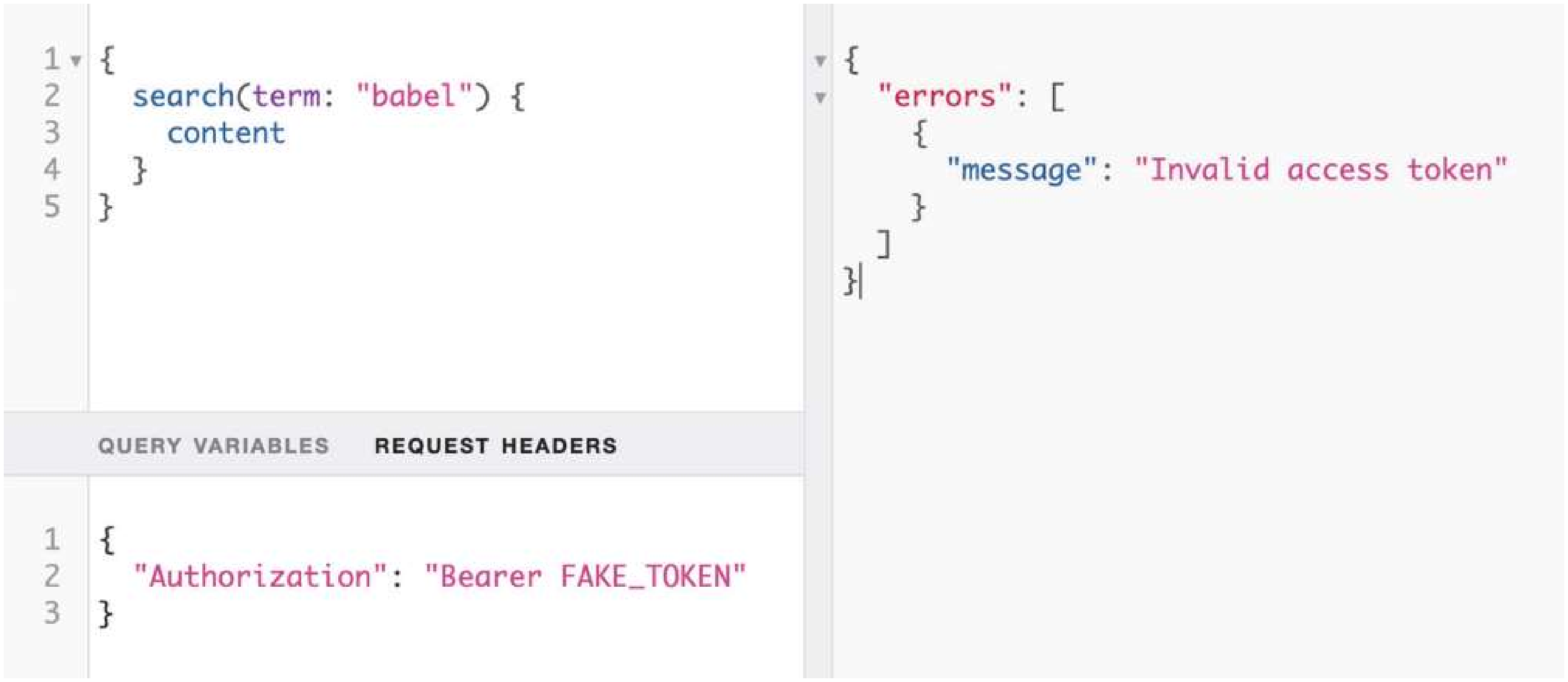
```
{
  search(term: "babel") {
    content
  }
}
```

QUERY VARIABLES   REQUEST HEADERS

```
{
  "Authorization": "Bearer 1b0d10b5efde43
}
```

```
{
  "data": {
    "search": [
      {
        "content": "Babel configuration file for
      }
    ]
  }
}
```

Authenticating API consumers



```
{
  search(term: "babel") {
    content
  }
}
```

```
{
  "Authorization": "Bearer FAKE_TOKEN"
}
```

```
{
  "errors": [
    {
      "message": "Invalid access token"
    }
  ]
}
```

The me root query field

- Let’s implement the me root query field next.
- The following is the part of the SDL related to that field.

```
type Query {
  // ...
  me: User
}

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

The me root query field

- Let's start with the me field itself, Here's a query we can use to test it when it's finished.

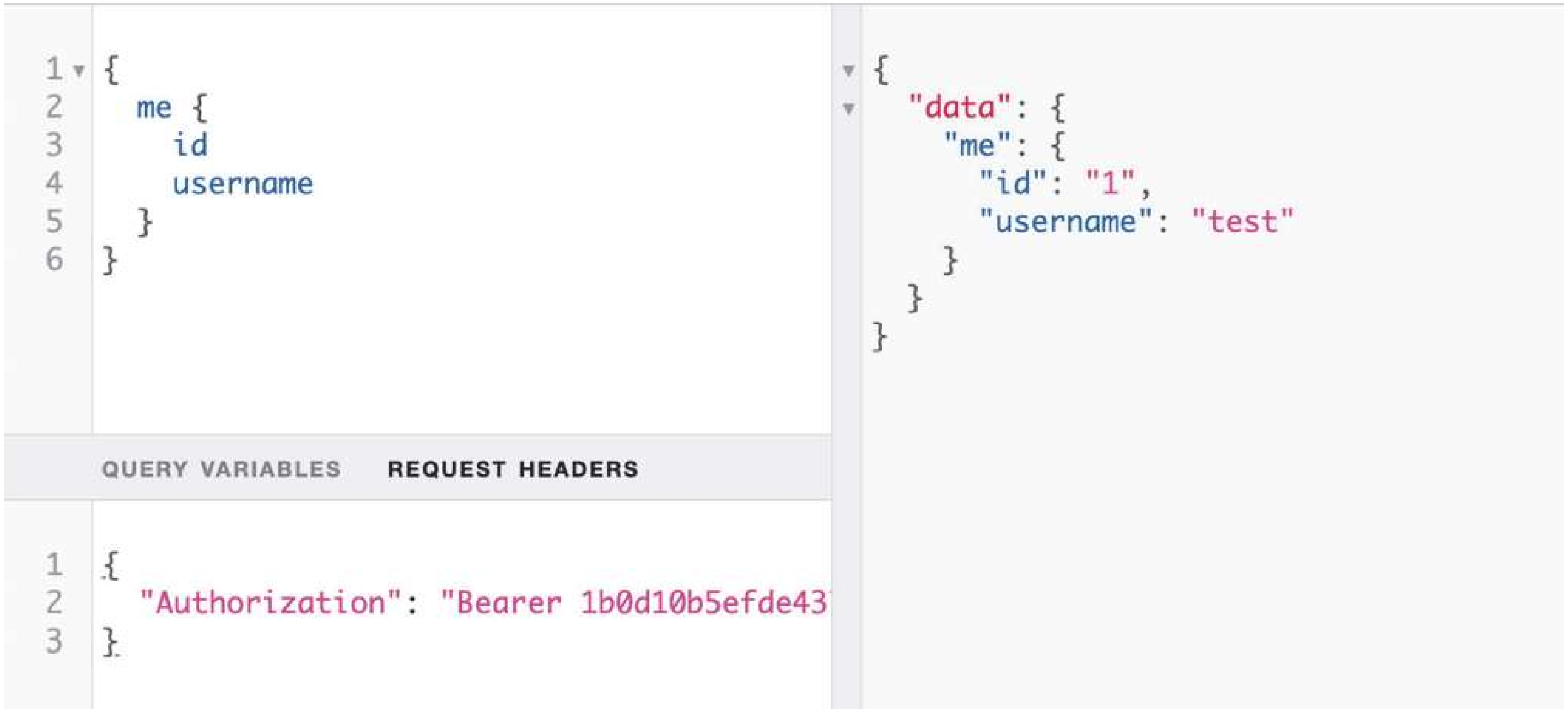
```
{
  me {
    id
    username
  }
}
```

The me root query field

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

  server.use('/', async (req, res) => {
    // ·-·-·
    graphqlHTTP({
      schema,
      context: { loaders, mutators, currentUser },
      graphiql: { headerEditorEnabled: true },
      // ·-·-·
    })(req, res);
  });

  // ·-·-·
}
```

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

const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    // ...
    me: {
      type: User,
      resolve: async (source, args, { currentUser }) => {
        return currentUser;
      },
    },
  }),
});
```

The me root query field



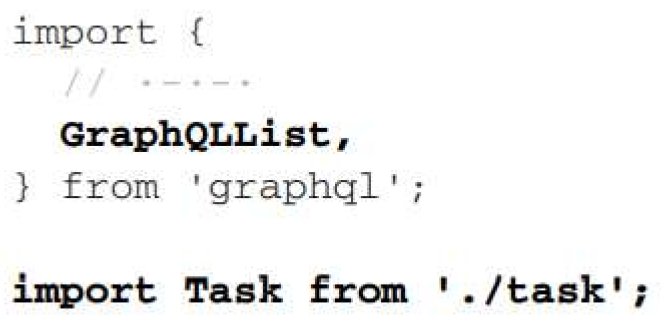
```
{
  me {
    id
    username
  }
}
```

```
{
  "Authorization": "Bearer 1b0d10b5efde43
}
```

```
{
  "data": {
    "me": {
      "id": "1",
      "username": "test"
    }
  }
}
```

The me root query field

- The new type gets the taskList field, To implement that without duplicating the User type fields, we can make its fields configuration property a function that returns the fields object with or without a taskList field
- Here’s what I came up with



```
import {
  // ·-·-·
  GraphQLList,
} from 'graphql';

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

```
const fieldsWrapper = ({ meScope }) => {
  const userFields = {
    id: { type: new GraphQLNonNull(GraphQLID) },
    username: { type: GraphQLString },
    name: {
      type: GraphQLString,
      resolve: ({ firstName, lastName }) =>
        [firstName, lastName].filter(Boolean).join(' '),
    },
  };

  if (meScope) {
    userFields.taskList = {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(Task)),
      ),
      resolve: (source, args, { loaders, currentUser }) => {
        return loaders.tasksForUsers.load(currentUser.id);
      },
    };
  }

  return userFields;
};
const User = new GraphQLObjectType({
  name: 'User',
  fields: () => fieldsWrapper({ meScope: false }),
});

export const Me = new GraphQLObjectType({
  name: 'Me',
  fields: () => fieldsWrapper({ meScope: true }),
});

export default User;
```

The loaders.tasksForUsers function does not exist yet.

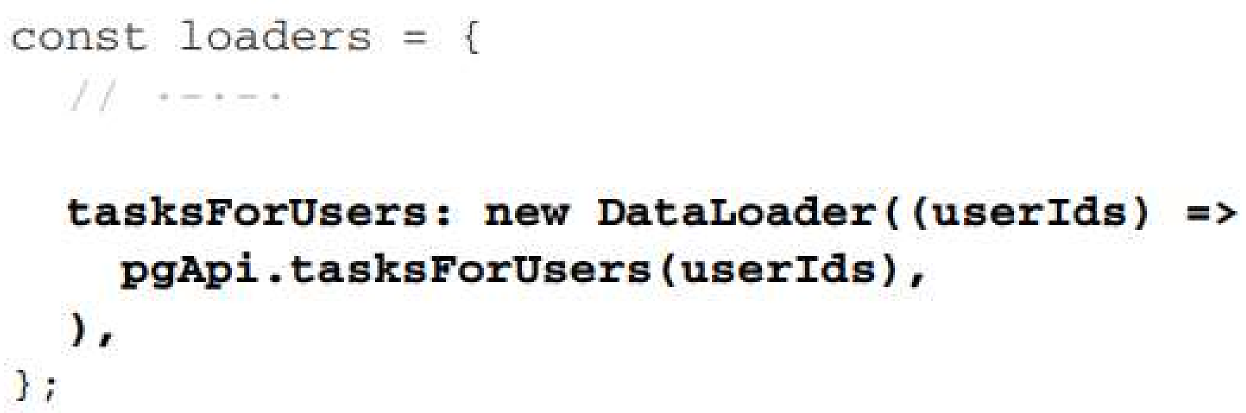
The me root query field

```
  // ·-·-·
  import { Me } from './types/user';

  const QueryType = new GraphQLObjectType({
    name: 'Query',
    fields: () => ({
      // ·-·-·
       me: {
        type: Me,
        resolve: async (source, args, { currentUser }) => {
        return currentUser;
      },
    },
  }),
});
```

Replaces the default User import line

The me root query field

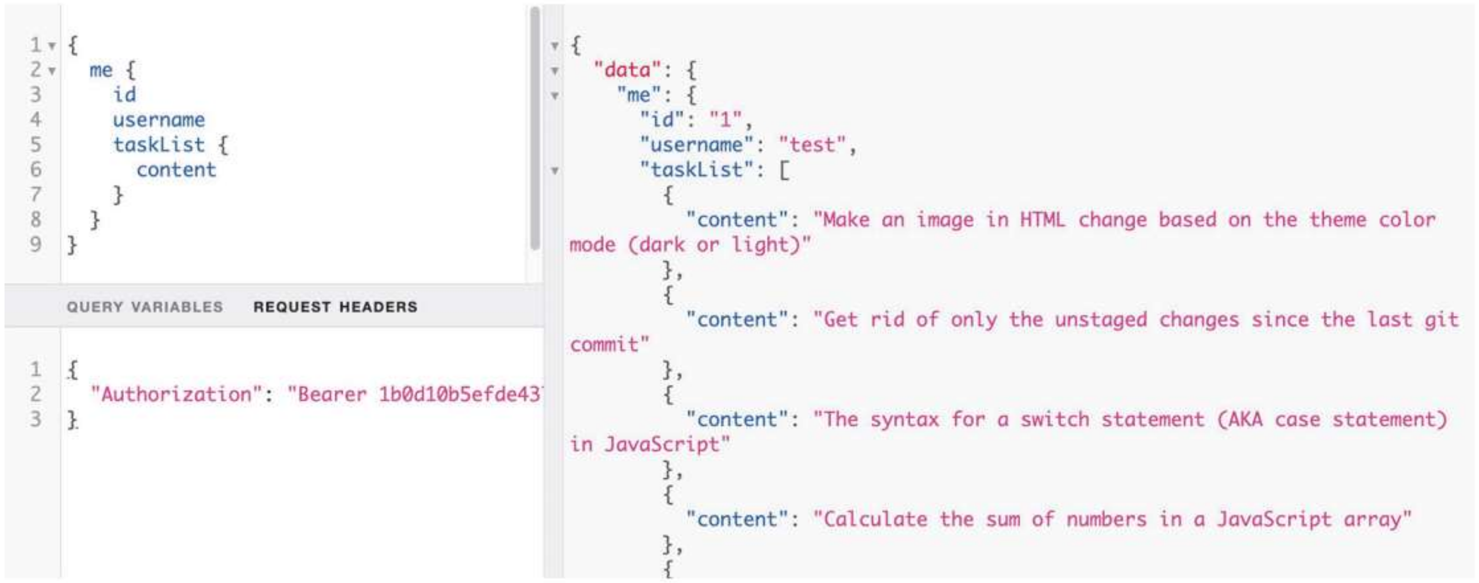
- Let's define the loaders.tasksForUsers function and its batch-loading pgApi function.



```
const loaders = {
  // ·····

  tasksForUsers: new DataLoader((userIds) =>
    pgApi.tasksForUsers(userIds),
  ),
};
```

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

  return {
    // ·-·-·

    tasksForUsers: async (userIds) => {
      const pgResp = await pgQuery(sqls.tasksForUsers, {
        $1: userIds,
      });
      return userIds.map((userId) =>
        pgResp.rows.filter((row) => userId === row.userId),
      );
    },

    // ·-·-·
  };
};
```

```
{
  me {
    id
    username
    taskList {
      content
    }
  }
}
```



```
{
  me {
    id
    username
    taskList {
      content
    }
  }
}
```

QUERY VARIABLES REQUEST HEADERS

```
{
  "Authorization": "Bearer 1b0d10b5efde43
}
```

```
{
  "data": {
    "me": {
      "id": "1",
      "username": "test",
      "taskList": [
        {
          "content": "Make an image in HTML change based on the theme color mode (dark or light)"
        },
        {
          "content": "Get rid of only the unstaged changes since the last git commit"
        },
        {
          "content": "The syntax for a switch statement (AKA case statement) in JavaScript"
        },
        {
          "content": "Calculate the sum of numbers in a JavaScript array"
        },
        {
```

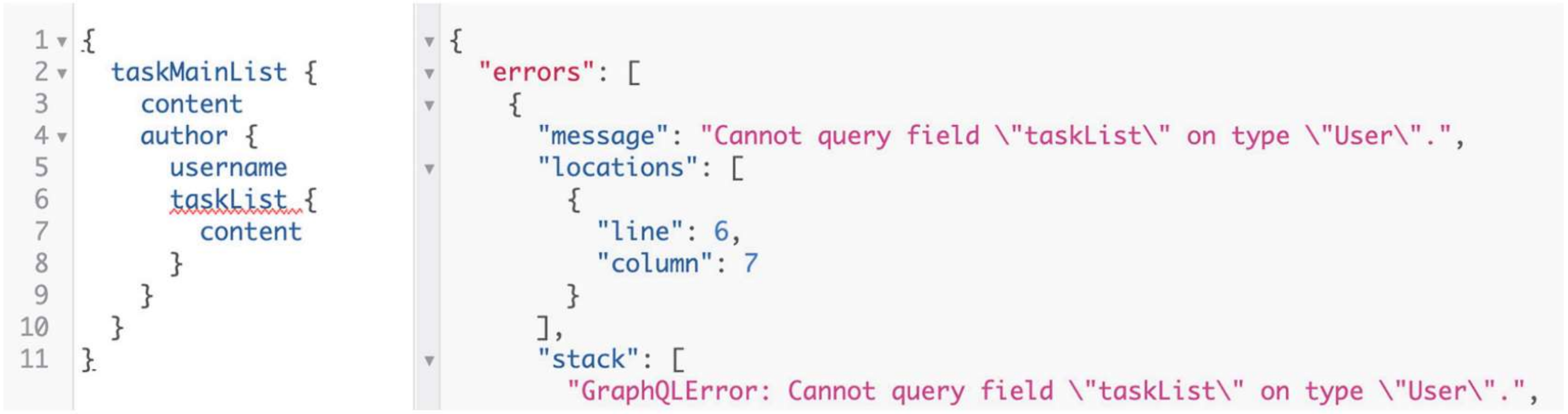
The me root query field

- You can also make sure that the taskList field is not available under the author field using this query.

```
{
  taskMainList {
    content
    author {
      username
      taskList {
        content
      }
    }
  }
}
```

The taskList field should not be available under the author field scope.

The me root query field



```
{
  taskMainList {
    content
    author {
      username
      taskList {
        content
      }
    }
  }
}
```

```
{
  "errors": [
    {
      "message": "Cannot query field \"taskList\" on type \"User\".",
      "locations": [
        {
          "line": 6,
          "column": 7
        }
      ],
      "stack": [
        "GraphQLError: Cannot query field \"taskList\" on type \"User\".",
```

Mutations for the Task model

```
input TaskInput {
  content: String!
  tags: [String!]!
  isPrivate: Boolean!
}

type TaskPayload {
  errors: [UserError!]!
  task: Task
}

type Mutation {
  taskCreate(input: TaskInput!): TaskPayload!

  # ·-·-·
}
```

```
import {
  GraphQLInputObjectType,
  GraphQLString,
  GraphQLNonNull,
  GraphQLBoolean,
  GraphQLList,
} from 'graphql';

const TaskInput = new GraphQLInputObjectType({
  name: 'TaskInput',

    fields: () => ({
      content: { type: new GraphQLNonNull(GraphQLString) },
      tags: {
        type: new GraphQLNonNull(
          new GraphQLList(new GraphQLNonNull(GraphQLString)),
        ),
      },
      isPrivate: { type: new GraphQLNonNull(GraphQLBoolean) },
    }),
  });

  export default TaskInput;
```

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

import Task from './task';
import UserError from './user-error';

const TaskPayload = new GraphQLObjectType({
  name: 'TaskPayload',
  fields: () => ({
    errors: {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(UserError)),
      ),
    },
    task: { type: Task },
  }),
});

export default TaskPayload;
```

```
// -----
import TaskPayload from './types/payload-task';
import TaskInput from './types/input-task';

const MutationType = new GraphQLObjectType({
  name: 'Mutation',
  fields: () => ({
    // -----

    taskCreate: {
      type: TaskPayload,
      args: {
        input: { type: new GraphQLNonNull(TaskInput) },
      },
      resolve: async (
        source,
        { input },
        { mutators, currentUser },
      ) => {
        return mutators.taskCreate({ input, currentUser });
      },
    },
  }),
});
```

The mutators.taskCreate method does not exist yet.

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

  return {
    // ·-·-·
    mutators: {
      // ·-·-·
      taskCreate: async ({ input, currentUser }) => {
        const payload = { errors: [] };
        if (input.content.length < 15) {
          payload.errors.push({
            message: 'Text is too short',
          });
        }
        if (payload.errors.length === 0) {
          const pgResp = await pgQuery(sqls.taskInsert, {
            $1: currentUser.id,
            $2: input.content,
            $3: input.tags.join(','),
            $4: input.isPrivate,
          });

          if (pgResp.rows[0]) {
            payload.task = pgResp.rows[0];
          }
        }

        return payload;
      },
    },
  };
};
```

Remember that tags are stored as comma-separated values in the database, but the API consumer sends them as an array of strings. That's why we needed a join call here.

- Here’s a request you can use in GraphiQL to test the taskCreate mutation.

```
mutation taskCreate {
  taskCreate (
    input: {
      content: "Use INSERT/SELECT together in PostgreSQL",
      tags: ["sql", "postgresql"]
      isPrivate: false,
   }
  ) {
    errors {
      message
    }
    task {
      id
      content
      tags
      author {
        id
      }
      createdAt
    }
  }
}
```

Mutations for the Approach model

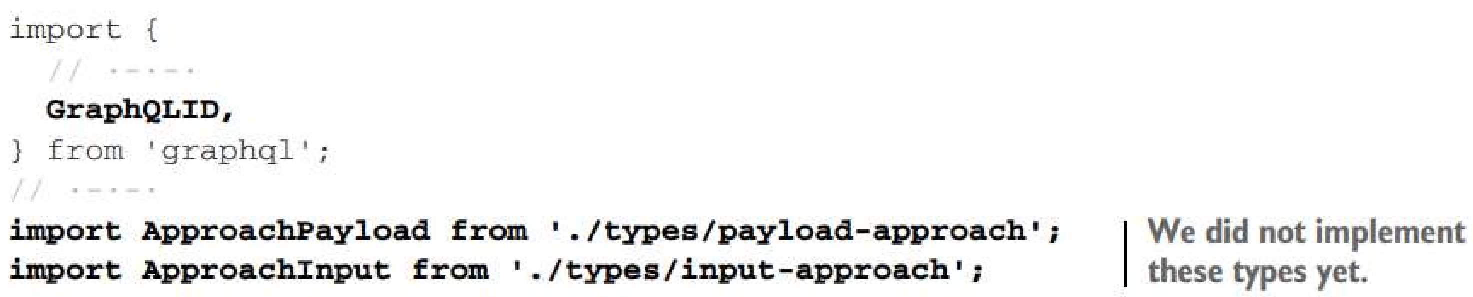
- Let’s now implement the two mutations to add an Approach to a Task (approachCreate) and vote on existing Approaches (approachVote).

The approachCreate mutation

```
input ApproachDetailInput {
  content: String!
  category: ApproachDetailCategory!
}

input ApproachInput {
  content: String!
  detailList: [ApproachDetailInput!]!
}

type ApproachPayload {
  errors: [UserError!]!
  approach: Approach
}

type Mutation {
  approachCreate(
    taskId: ID!
    input: ApproachInput!
  ): ApproachPayload!

  # ...
}
```

Mutations for the Task model

- One way to do that is to pass the contextlevel mutators object to the main mutator function as an argument.



```
import {
  // ·-·-·
  GraphQLID,
} from 'graphql';
// ·-·-·
import ApproachPayload from './types/payload-approach';
import ApproachInput from './types/input-approach';
```

We did not implement these types yet.

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

    approachCreate: {
      type: ApproachPayload,
      args: {
        taskId: { type: new GraphQLNonNull(GraphQLID) },
        input: { type: new GraphQLNonNull(ApproachInput) },
      },
      resolve: async (
        source,
        { taskId, input },
        { mutators, currentUser },
      ) => {
        return mutators.approachCreate({
          taskId,
          input,
          currentUser,
          mutators,
        });
      },
    },
  }),
});
```

The main mutator (not implemented yet)

Note that the mutators object is passed here.

Mutations for the Task model

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

 import Approach from './approach';
 import UserError from './user-error';

const ApproachPayload = new GraphQLObjectType({
  name: 'ApproachPayload',
  fields: () => ({
    errors: {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(UserError)),
      ),
    },
    approach: { type: Approach },
  }),
});

export default ApproachPayload;
```

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

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

const ApproachDetailInput = new GraphQLInputObjectType({
  name: 'ApproachDetailInput',
  fields: () => ({
    content: { type: new GraphQLNonNull(GraphQLString) },
    category: {
      type: new GraphQLNonNull(ApproachDetailCategory),
    },
  }),
});

export default ApproachDetailInput;
```

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

import ApproachDetailInput from './input-approach-detail';

const ApproachInput = new GraphQLInputObjectType({
  name: 'ApproachInput',
  fields: () => ({

      content: { type: new GraphQLNonNull(GraphQLString) },
      detailList: {
        type: new GraphQLNonNull(
          new GraphQLList(new GraphQLNonNull(ApproachDetailInput)),
        ),
      },
    }),
  });

  export default ApproachInput;
```

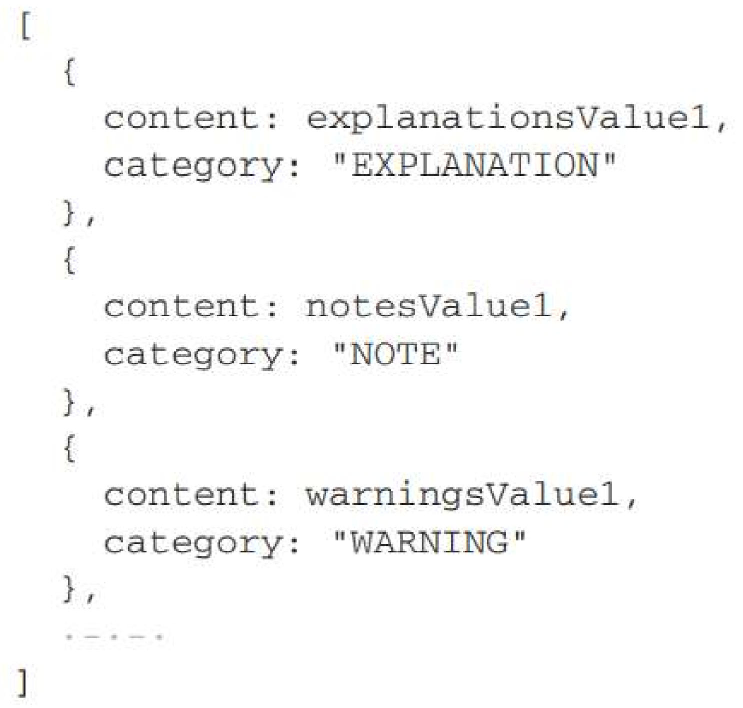
```
const pgApiWrapper = async () => {
  // ·-·-·

  return {
    // ·-·-·

    mutators: {
      // ·-·-·

      approachCreate: async ({
        taskId,
        input,
        currentUser,
        mutators,
      }) => {
        const payload = { errors: [] };
          if (payload.errors.length === 0) {
            const pgResp = await pgQuery(sqls.approachInsert, {
            $1: currentUser.id,
            $2: input.content,
            $3: taskId,
          });
          if (pgResp.rows[0]) {
            payload.approach = pgResp.rows[0];
            await pgQuery(sqls.approachCountIncrement, {
              $1: taskId,
            });
            await mutators.approachDetailCreate(
              payload.approach.id,
              input.detailList,
            );
          }
        }
         return payload;
       },
    },
  };
};
```

Invokes the PostgreSQL operation to create the Approach record

The Approach record is created. Increment the Task's approachCount.

Continue to add its details in MongoDB.

- We need to convert this format (as we designed it for the ApproachDetailInput type):



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

Mutations for the Task model

- Here is the format we will convert it into, which is expected by the approachDetails MongoDB collection:

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

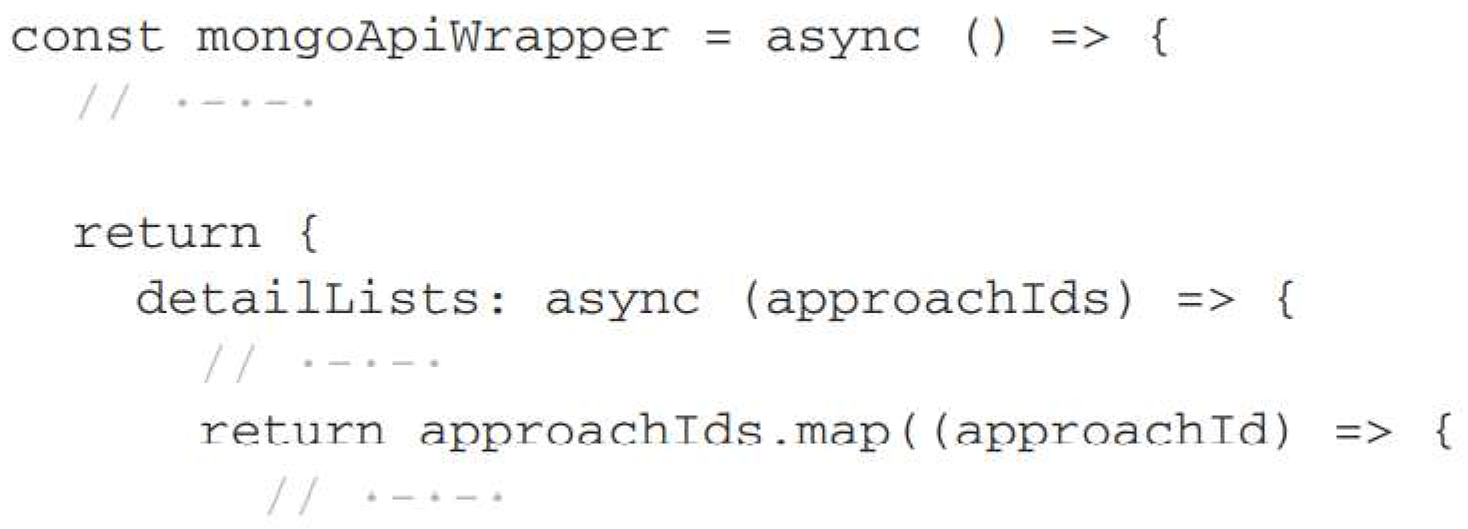
```
const mongoApiWrapper = async () => {
  // -----

  return {
    // -----

    mutators: {
      approachDetailCreate: async (approachId, detailsInput) => {

          const details = {};
          detailsInput.forEach(({ content, category }) => {
            details[category] = details[category] || [];
            details[category].push(content);
          });
          return mdb.collection('approachDetails').insertOne({
            pgId: approachId,
            ...details,
          });
        },
      },
    };
  };
```

Mutations for the Task model

```
const ApproachDetailCategory = new GraphQLEnumType({
  name: 'ApproachDetailCategory',
  values: {
    NOTE: { value: 'notes' },
    EXPLANATION: { value: 'explanations' },
    WARNING: { value: 'warnings' },
  },
});
```

Mutations for the Task model

- We now need to change the first conversion we made to work with this ENUM type change.



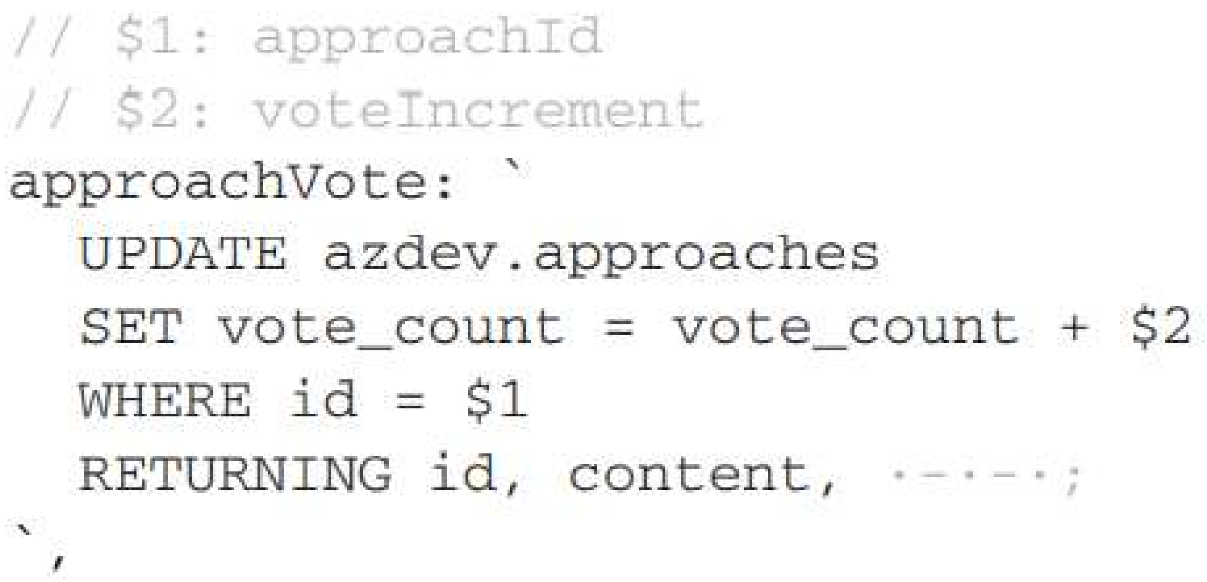
```
const mongoApiWrapper = async () => {
  // ·-·-·

  return {
    detailLists: async (approachIds) => {
      // ·-·-·
      return approachIds.map((approachId) => {
        // ·-·-·
      if (explanations) {
        approachDetails.push(
          ...explanations.map((explanationText) => ({
            content: explanationText,
            category: 'explanations',
          }))
        );
      }
      if (notes) {
        approachDetails.push(
          ...notes.map((noteText) => ({
            content: noteText,
            category: 'notes',
          }))
        );
      }
      if (warnings) {
        approachDetails.push(
          ...warnings.map((warningText) => ({
            content: warningText,
            category: 'warnings',
          }))
        );
      }
      return approachDetails;
    });
  },
mutation approachCreate {
  approachCreate(
    taskId: 42 # Get this value from a taskCreate mutation call
    input: {
      content: "INSERT INTO tableName ·-·-·] ) ] SELECT-STATEMENT",
      detailList: [
        {
          content: "You can still use a RETURNING clause after that",
          category: NOTE,
        },
        {
          content: "The INSERT statement only works if the SELECT statement
          ➡ does",
          category: EXPLANATION,
        },
      ],
    }
  ) {
    errors {
      message
    }
    approach {
      id
      content
      voteCount
      author {
        username
      }
      detailList {
        content
        category
      }
    }
  }
}
```

The approachVote mutation

- The part of the SDL text that we need to focus on for the approachVote mutation

```
input ApproachVoteInput {
  """true for up-vote and false for down-vote"""
  up: Boolean!
}

type Mutation {
  approachVote(
    approachId: ID!
    input: ApproachVoteInput!
  ): ApproachPayload!

  # ·-·-·
}
```

The approachVote mutation



```
// $1: approachId
// $2: voteIncrement
approachVote: `
  UPDATE azdev.approaches
  SET vote_count = vote_count + $2
  WHERE id = $1
  RETURNING id, content, ·-·-·;
`,
```

- Here’s a request you can use in GraphiQL to test your implementation.

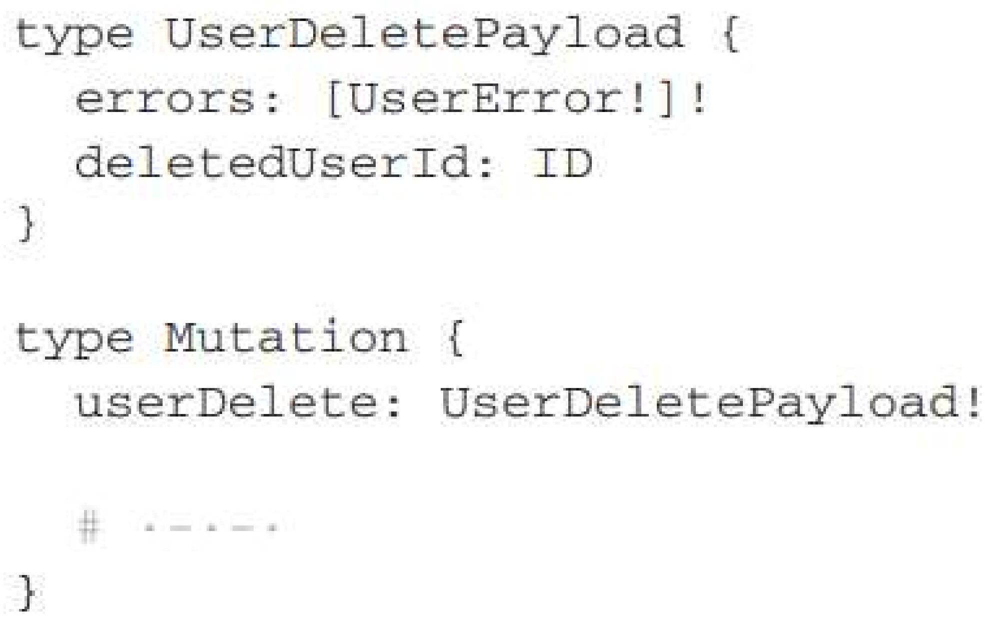
```
mutation approachVote {
  approachVote(
    approachId: 42 # Get this value from approachCreate

      input: { up: false }
    ) {
      errors {
        message
      }
      approach {
        content
        voteCount
      }
    }
  }
```

```
import {
  GraphQLInputObjectType,
  GraphQLBoolean,
  GraphQLNonNull,
} from 'graphql';

const ApproachVoteInputType = new GraphQLInputObjectType({
  name: 'ApproachVoteInput',
  description: "true for up-vote and false for down-vote",
  fields: () => ({
    up: { type: new GraphQLNonNull(GraphQLBoolean) },
  }),
});

export default ApproachVoteInputType;
```

```
  // -.-.-.
  import ApproachVoteInput from './types/input-approach-vote';

  const MutationType = new GraphQLObjectType({
    name: 'Mutation',
    fields: () => ({
      // -.-.-.
      approachVote: {
        type: ApproachPayload,
        args: {
          approachId: { type: new GraphQLNonNull(GraphQLID) },
          input: { type: new GraphQLNonNull(ApproachVoteInput) },
        },
        resolve: async (
          source,
          { approachId, input },
          { mutators },
        ) => {
          return mutators.approachVote({ approachId, input });

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

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

  return {
    // ·-·-·
    mutators: {
      // ·-·-·
      approachVote: async ({ approachId, input }) => {
        const payload = { errors: [] };
        const pgResp = await pgQuery(sqls.approachVote, {
          $1: approachId,
          $2: input.up ? 1 : -1,
        });

        if (pgResp.rows[0]) {
          payload.approach = pgResp.rows[0];
        }

        return payload;
      },
    },
  };
};
```

The userDelete mutation

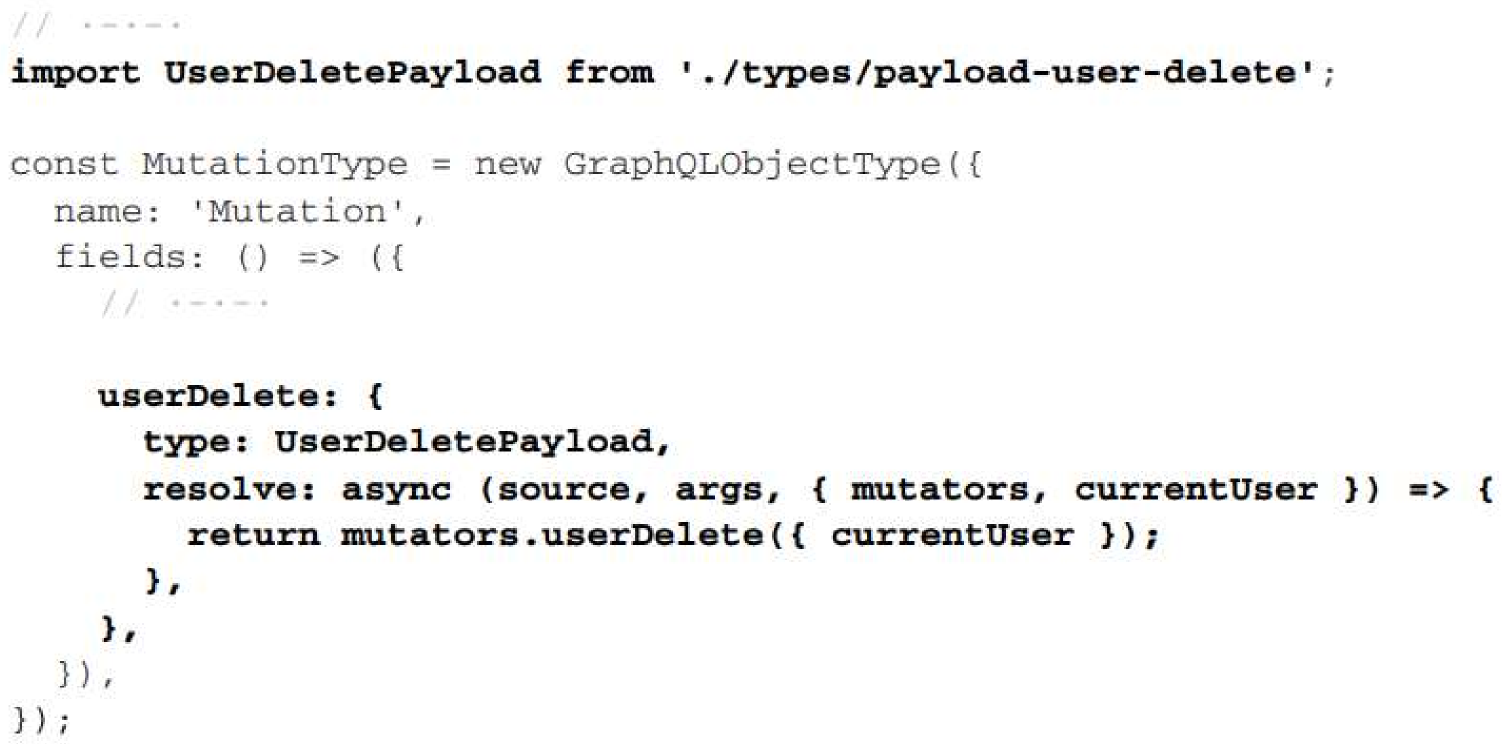


```
type UserDeletePayload {
  errors: [UserError!]!
  deletedUserId: ID
}

type Mutation {
  userDelete: UserDeletePayload!

  # ·-·-·
}
```

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

import UserError from './user-error';

const UserDeletePayload = new GraphQLObjectType({
  name: 'UserDeletePayload',
  fields: () => ({
    errors: {
      type: new GraphQLNonNull(
        new GraphQLList(new GraphQLNonNull(UserError)),
      ),
    },
    deletedUserId: { type: GraphQLID },
  }),
});

export default UserDeletePayload;
```

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

   return {
     // ·-·-·

     mutators: {

     // ·-·-·

     userDelete: async ({ currentUser }) => {
       const payload = { errors: [] };
       try {
         await pgQuery(sqls.userDelete, {
           $1: currentUser.id,
         });
         payload.deletedUserId = currentUser.id;
       } catch (err) {
         payload.errors.push({
           message: 'We were not able to delete this account',
         });
       }

       return payload;
     },
   },
  };
};
```

The userDelete mutation

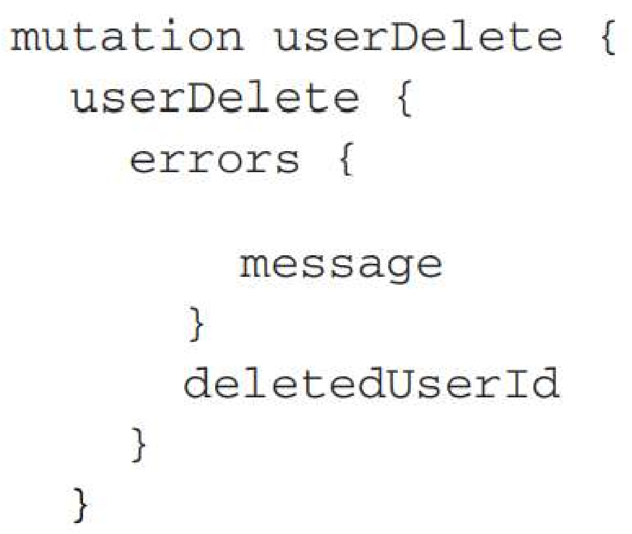


```
// ....
import UserDeletePayload from './types/payload-user-delete';

const MutationType = new GraphQLObjectType({
  name: 'Mutation',
  fields: () => ({
    // ....

    userDelete: {
      type: UserDeletePayload,
      resolve: async (source, args, { mutators, currentUser }) => {
        return mutators.userDelete({ currentUser });
      },
    },
  }),
});
```

The userDelete mutation

- To test this mutation, create a new user account using the userCreate mutation example

- Use that account’s authToken value in the request headers editor and send the following mutation request.



```
mutation userDelete {
  userDelete {
    errors {

        message
      }
      deletedUserId
    }
  }
```

Summary

- To host mutations, a GraphQL schema must define a root mutation type.
- To organize database operations for mutations, you can group them on a single object that you expose as part of the global context for resolvers.
- User-friendly error messages can and should be included as part of any mutation response

"Complete Lab"