1.1 package.json - Server

#nodejs #backend

Scripts

```
"scripts": {
    "watch": "tsc -w",
    "dev": "nodemon dist/index.js",
    "start": "node dist/index.js",
    "start2": "ts-node src/index.ts",
    "dev2": "nodemon --exec ts-node src/index.ts",
    "create:migration": "mikro-orm migration:create"
  },
```

- **tsc -w** - re-compiles .ts to .js **on change**
- **nodemon** - re-executes the .js code on change
- ts-node is slower than node, so we're using node to run the compiled **dist/index.js** file instead of using ts-node to run the index.ts file

Dependencies

- ⚠️ **When installing a package make sure to come back here and look up the version and install the version listed here or it might be "*too new*" and thus different**

Until the end of MikroORM section	At the end of the project

`"dependencies": {`
`"@mikro-orm/cli": "4.5",`
`"@mikro-orm/core": "4.5.10",`
`"@mikro-orm/migrations": "4.5.10",`
`"@mikro-orm/postgresql": "4.5",`
`"apollo-server-express": "2.16.1",`
`"argon2": "^0.40.3",`
`"connect-redis": "5.0.0",`
`"cors": "2.8.5",`
	`"dependencies": {`
	`"apollo-server-express": "2.16.1",`
	`"argon2": "^0.40.3",`
	`"connect-redis": "5.0.0",`
	`"cors": "2.8.5",`
	`"dataloader": "2.0.0",`
	`"express": "^4.19.2",`
	`"express-session": "1.17.1",`
	`"graphql": "15.3.0",`
	`"ioredis": "4.17.3",`
	`"nodemailer": "6.4.11",`

```
    "express": "^4.19.2",
    "express-session": "1.17.1",
    "graphql": "15.3.0",
    "ioredis": "4.17.3",
    "nodemailer": "6.4.11",
    "pg": "^8.12.0",
    "reflect-metadata": "0.1.13",
    "type-graphql": "1.0.0",
    "uuid": "8.3.0"
  },
```

```
    "pg": "^8.12.0",
    "reflect-metadata": "0.1.13",
    "type-graphql": "1.0.0",
    "typeorm": "0.2.25",
    "uuid": "8.3.0"
  },
```

DevDependencies

Until the end of MikroORM section	At the end of the project

```
"devDependencies": {
    "@types/connect-redis": "0.0.13",
    "@types/cors": "2.8.7",
    "@types/express": "^4",
    "@types/express-session": "^1",
    "@types/ioredis": "4.17.3",
    "@types/node": "^20.14.2",
    "@types/nodemailer": "^6",
    "@types/pg": "^8",
    "@types/uuid": "8.3.1",
    "nodemon": "^3.1.3",
    "ts-node": "^10.9.2",
    "typescript": "^5.4.5"
  },
```

```
"devDependencies": {
    "@types/connect-redis": "0.0.13",
    "@types/cors": "2.8.7",
    "@types/express": "^4",
    "@types/express-session": "^1",
    "@types/ioredis": "4.17.3",
    "@types/node": "^20.14.2",
    "@types/nodemailer": "^6",
    "@types/pg": "^8",
    "@types/uuid": "8.3.1",
    "nodemon": "^3.1.3",
    "ts-node": "^10.9.2",
    "typescript": "^5.4.5"
  },
```

1.2 tsconfig.json - Server

#typescript #backend

---

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2017",
    "module": "commonjs",
    "lib": ["dom", "es6", "es2017", "esnext.asynciterable"],
    "skipLibCheck": true,
    "sourceMap": true,
    "outDir": "./dist",
    "moduleResolution": "node",
    "removeComments": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "noImplicitThis": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "allowSyntheticDefaultImports": true,
    "esModuleInterop": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "resolveJsonModule": true,
    "baseUrl": "."
  },
  "exclude": ["node_modules"],
  "include": ["./src/**/*.ts"]
}
```

2. package.json - Web

#reactjs #nextjs

```
"scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "gen": "graphql-codegen --config codegen.yml"
  },
```

Dependencies

- ⚠️ When installing a package make sure to come back here and look up the version and install the version listed here or it might be "*too new*" and thus different

```
"dependencies": {
    "@chakra-ui/icons": "^2.0.2",
    "@chakra-ui/react": "^2.2.1",
    "@chakra-ui/theme-tools": "^2.0.2",
    "@emotion/react": "^11.9.0",
    "@emotion/styled": "^11.9.0",
    "@urql/exchange-graphcache": "3.0.2",
    "formik": "^2.4.6",
    "framer-motion": "^6.3.0",
    "graphql": "15.3.0",
    "graphql-tag": "^2.11.0",
    "isomorphic-unfetch": "3.0.0",
    "next": "latest",
    "next-urql": "1.1.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-is": "16.13.1",
    "urql": "1.10.0"
  },
```

DevDependencies

```
"devDependencies": {
    "@graphql-codegen/cli": "1.17.7",
    "@graphql-codegen/typescript": "1.17.7",
    "@graphql-codegen/typescript-operations": "1.17.7",
    "@graphql-codegen/typescript-urql": "2.0.0",
    "@types/node": "^18.0.0",
    "@types/react": "^18.0.0",
    "@types/react-dom": "^18.0.0",
    "typescript": "^4.7.2"
  },
```

3. MikroORM

#mikroorm #backend

```
yarn add @mikro-orm/cli @mikro-orm/core @mikro-orm/migration @mikro-orm/posgresql
pq
```

- @mikro-orm/posgresql and pq → these are for postgresql, but packages for other DBs can also be installed

---

Initial setup and config

package,json

```
"mikro-orm": {
    "useTsNode": true,
    "configPaths": [
      "./src/mikro-orm.config.ts",
      "./dist/mikro-orm.config.js"
    ]
  }
```

constants.ts

```
export const __prod__ = process.env.NODE_ENV === "production"; // is the env
variable set as "production" ?
export const COOKIE_NAME = "qid";
```

mikro-orm.config.ts

```
import { MikroORM } from "@mikro-orm/core";
import path from "path";
import { __prod__ } from "./constants";
import { Post } from "./entities/Post";
import { User } from "./entities/User";

export default {
        migrations: {
```

```
                path: path.join(__dirname, "./migrations"),
                pattern: /^[\w-]+\d+\.[tj]s$/,
        },
        entities: [Post, User],
        dbName: "lireddit",
        type: "postgresql",
        user: "postgres",
        password: "postgres",
        debug: !__prod__,
} as Parameters<typeof MikroORM.init>[0];
```

- The **entities** are the names of the database tables that mikroorm will interact with (see below)
- as Parameters<typeof MikroORM.init>[0] allows this export to be passed into MikroORM..init() in index.ts
- `entities: [Post, User],` this should be updated everytime we add a new entity
- after creating the entity and updating the above parameter we run `mikro-orm migration:create` to create a new migration that will update the database, add columns and create the new tables if necessary
- the new migrations are automaticlly run with the `await orm.getMigrator().up();` line in index.ts, below.

---

index,ts

```
import { MikroORM } from "@mikro-orm/core";
import microConfig from "./mikro-orm.config";

const main = async () => {
  const orm = await MikroORM.init(microConfig);
  await orm.getMigrator().up(); // run migration
```

- initialize MikroORM using the config and set up migrator to run at startup

---

How to interact with the DB

We will be using these in the Resolvers to interact with the DB

```
const post = orm.em.create(Post, { title: "my first post" }) // the Post class is used to create the missing fields
await orm.em.persistAndFlush(post)
```

```
// or
await orm.em.nativeInsert(Post, { title: 'my first post 2', createdAt: new
Date(), updatedAt: new Date()}) // we must provide all fields

const posts = await orm.em.find(Post, {})
console.log(posts)
```

- **inline** method of creating posts and pushing them into db, or searching for (all) posts with mikroORM we do not use this much but instead use **resolvers**

How to create the migrations

```
npx mikro-orm migration:create
```

- create migrations, i.e. create the DB table according to the entity schemas that are defined
- and then when we start the server the migration is automatically run since we set it up that way in index.ts

4. Entity - Post.ts

#mikroorm #entity #backend

How to implement an Entity

(Later we will Convert the Post entity (class) to a GraphQL type:)

We use the @Entity(), @PrimaryKey() and @Property() atributes imported from @mikro-orm/core to define the columns of the db table (entity) Posts as follows:

entities/Posts.ts

```
import { Entity, PrimaryKey, Property } from "@mikro-orm/core";

@Entity()
export class Post {
  @PrimaryKey()
  id!: number;

  @Property({ type: 'date' }) // explicitly set type for MikroORM otherwise it infers is as "jsonb" type
  createdAt = new Date();

  @Property({ type: 'date', onUpdate: () => new Date() })
  updatedAt = new Date();

  @Property({ type: 'text'})
  title!: string;
}
```

- `entities: [Post, User],` in mikro-orm.config.ts should be updated everytime we add a new entity
- after creating the entity and updating the above parameter we run `mikro-orm migration:create` to update the database create tables and create the new tables if necessary. (see 3. MikroORM)

5. GraphQL

#graphql #apolloserver #express #entity #backend

```
yarn add express apollo-server-express graphql type-graphql
yarn add reflect-metadata
yarn add -D @types/express
```

index.ts

```
import "reflect-metadata"; // needed for GraphQL to infer the types
```

- express is the server that is going to be used
- apollo-server-express allows us to use graphql or create a grapqhl endpoint easily
- graphql and type-graphql for the schema
- @types/express installs the ts types for express servers since they're not in-built
- **GraphQL runs on backend server** - accessed on localhost:4000 (configured in MikroORM step)

Initial setup of Express server

index.ts

```
const app = express();

app.get('/', (req, res) => {
    res.send("hello")
})

app.listen(4000, () => {
    console.log("server started on localhost:4000");
  });
```

- and now that the express server is set up, we should see "hello" at localhost:4000

Adding GraphQL endpoint with Apollo and defining a "hello" resolver

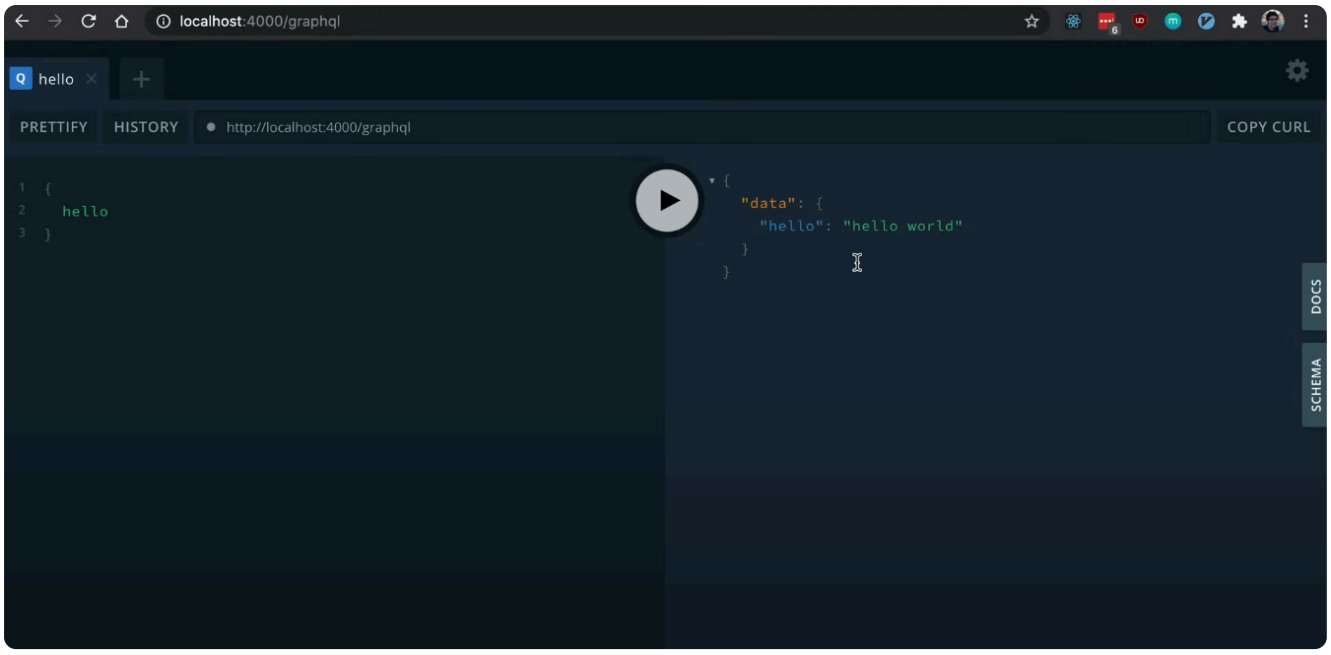
index.ts

```
const app = express();

const apolloServer = new ApolloServer({
  schema: await buildSchema({
    resolvers: [HelloResolver],
    validate: false,
  }),
});

apolloServer.applyMiddleware({ app })

app.listen(4000, () => {
  console.log("server started on localhost:4000");
})
```

resolvers/hello.ts

```
import { Query, Resolver } from "type-graphql";

@Resolver()
export class HelloResolver {
  @Query(() => String) // declare what the query returns (uppercase in
typegreaphql)
  hello() {
    return 'hello world'
  }
}
```

- and now we have the localhost:4000 /graphql endpoint as follows:



Convert the Post entity (class) to a GraphQL type:

- **We use** @ObjectType() and @Field attributes and Types imported from type-graphql package as follows.
- @Field is added to only those columns we want to expose to the GraphQL schema, so the data in them can be retrieved and displayed

entities/Post.ts

```
import { Entity, PrimaryKey, Property } from "@mikro-orm/core";
import { Field, Int, ObjectType } from "type-graphql";

@ObjectType() // graphQL
@Entity() // mikro-orm
export class Post {
  @Field(() => Int)
  @PrimaryKey()
  id!: number;

  @Field(() => String) // explicitly set type for GraphQL
  @Property({ type: 'date' }) // explicitly set type for MikroORM
  createdAt = new Date();

  @Field(() => String)
  @Property({ type: 'date', onUpdate: () => new Date() })
  updatedAt = new Date();
```

```
  @Field()
  @Property({ type: 'text'})
  title!: string;
}
```

- If the type is not explicitly set in @Field(), we might get a NoExplicitTypeError, (e.g., in createdAt and updatedAt)

Adding Context to the Apollo Server

- We create a `MyContext` type
- ExtendedRequest is implemented to extend Request to include { userId: number }

types.ts

```
import { EntityManager, IDatabaseDriver, Connection } from "@mikro-orm/core";
import { Request, Response } from "express";
import { Session, SessionData } from "express-session";
import { Redis } from "ioredis";

interface ExtendedRequest extends Request {
  session: Session &
    Partial<SessionData> &
    Express.Request & { userId: number };
}

export type MyContext = {
  em: EntityManager<IDatabaseDriver<Connection>>;
  req: ExtendedRequest;
  res: Response;
  redis: Redis; // to be added during (11)
};
```

- We modify the apolloServer implementation to include the `context`
- Here the context is providing to the resolvers the `orm.em` code as `em` (to interact with the DB) as well as the `redis` storage, which is defined here:

```
const apolloServer = new ApolloServer({
    schema: await buildSchema({
      resolvers: [HelloResolver, PostResolver, UserResolver],
      validate: false,
```

```
    }),
    context: ({ req, res }: MyContext) => ({ em: orm.em, req, res }), // context
is shared with all resolvers
  });
```

- And now we can implement 6. Resolver - post.ts

6. Resolver - post.ts

#graphql #resolver #mutation #query #mikroorm #backend

How to implement the Post resolver

- First we set up 5. GraphQL
- Then we implement the resolver:

Resolver(), Query(), Mutation(), Arg(), Ctx(), Int, are imported from type-graphql package;

Query() - only retrieves data from the DB, does not make any changes

Mutation() - makes changes to the DB

Arg() - is defined if the operation will have parameters

Ctx() - is the context provided by the Apollo server, accessible by all resolvers - defined here: Adding Context to the Apollo Server

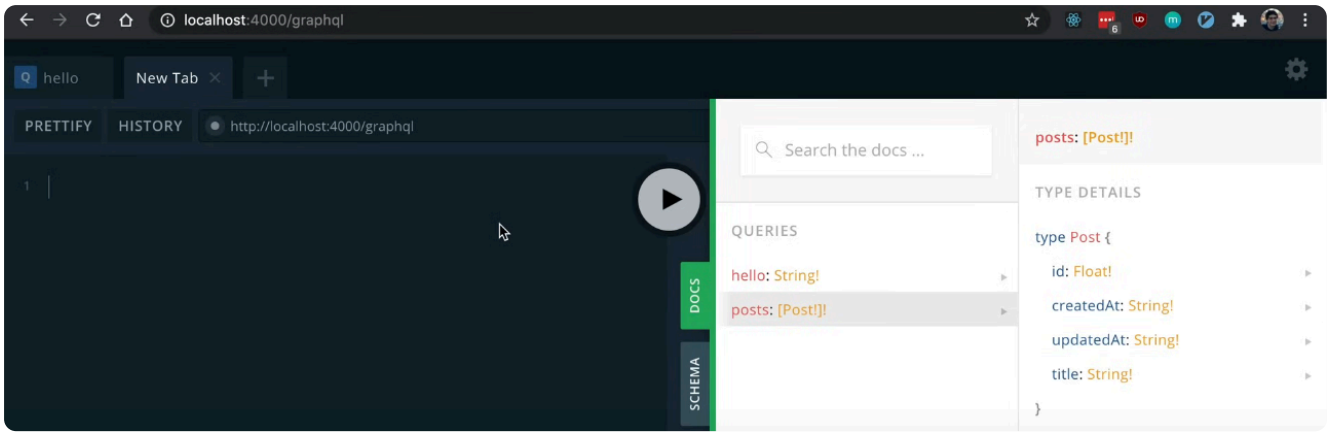
/resolvers/post.ts

```
import { Post } from "../entities/Post";
import { MyContext } from "src/types";
import { Arg, Ctx, Int, Mutation, Query, Resolver } from "type-graphql";
// import { sleep } from "../utils/sleep";

@Resolver()
export class PostResolver {
  @Query(() => [Post]) // [Post] is how we define arrays in return type for the resolver
  async posts(@Ctx() { em }: MyContext): Promise<Post[]> {
    //await sleep(3000); // simulate delay to test csr vs ssr load times
    return em.find(Post, {});
  }

  @Query(() => Post, { nullable: true })
  post(
    @Arg("id", () => Int) id: number, // "id" is the name to use in GraphQL schema, id is the field name and type in DB
    @Ctx() { em }: MyContext
  ): Promise<Post | null> {
    return em.findOne(Post, { id });
  }
```

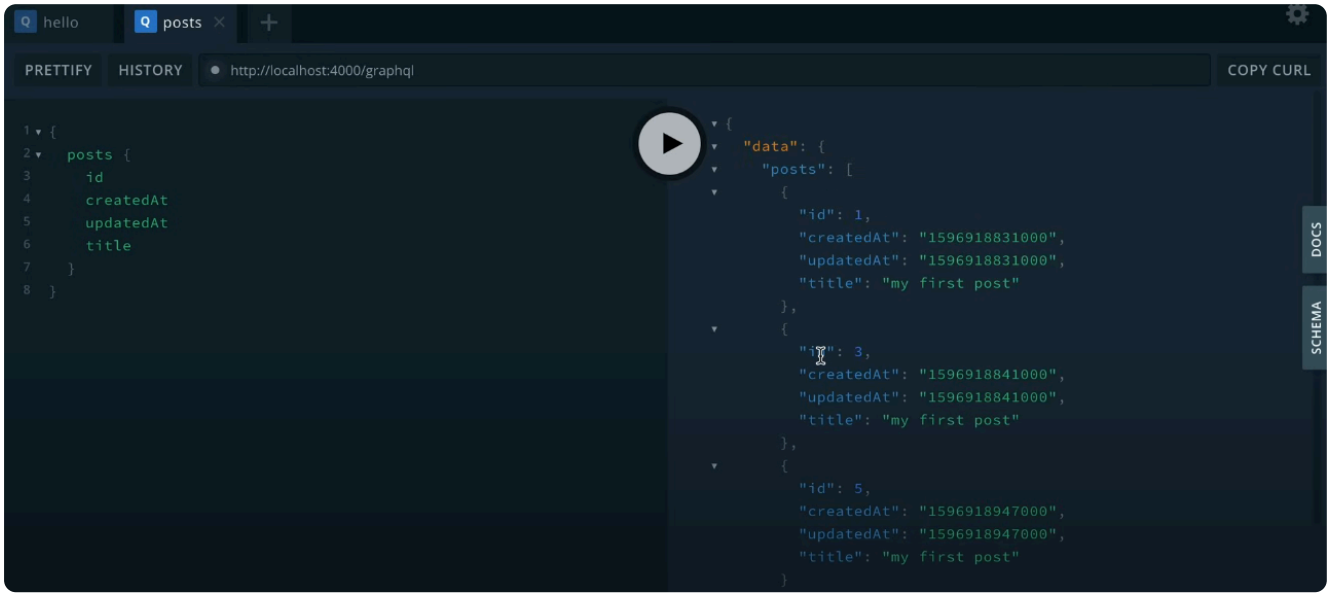
```
  @Mutation(() => Post)
  async createPost(
    @Arg("title", () => String) title: string,
    @Ctx() { em }: MyContext
  ): Promise<Post> {
    const post = em.create(Post, { title });

    await em.persistAndFlush(post);
    return post;
  }

  @Mutation(() => Post, { nullable: true })
  async updatePost(
    @Arg("id") id: number, // here we ommitted type declaration in @Arg - type
inference works for Int and String
    @Arg("title", () => String, { nullable: true }) title: string, // here we
explicitly set type since we want to make it nullable
    @Ctx() { em }: MyContext
  ): Promise<Post | null> {
    const post = await em.findOne(Post, { id });

    if (!post) {
      return null;
    }

    if (typeof title !== "undefined") {
      post.title = title;
      await em.persistAndFlush(post);
    }

    return post;
  }

  @Mutation(() => Boolean)
  async deletePost(
    @Arg("id") id: number,
    @Ctx() { em }: MyContext
  ): Promise<boolean> {
    const post = await em.findOne(Post, { id });
```

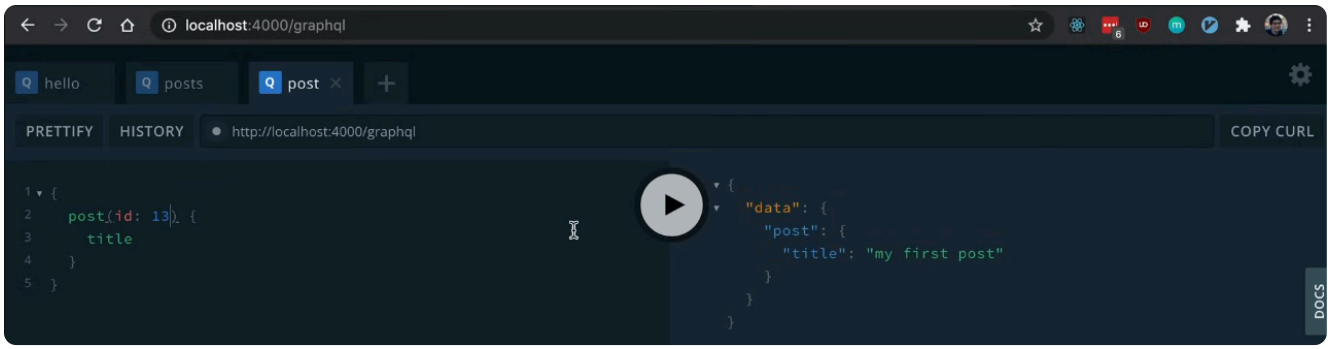
```
    if (!post) {
      return false;
    }

    await em.nativeDelete(Post, { id });
    return true;
  }
}
```

- and now we have the localhost:4000 /graphql endpoint as follows:

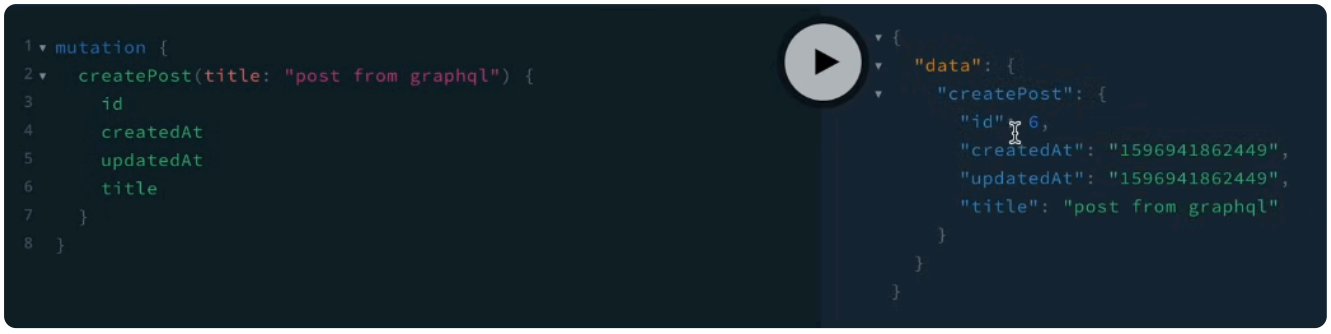


Running the GraphQL queries





- for mutations the syntax is as follows:



7. Entity - User.ts

#mikroorm #graphql #entity #backend

- Note that password does not have the @Field() attribute since we do **not** want to expose it to GraphQL

```
import { Entity, PrimaryKey, Property } from "@mikro-orm/core";
import { Field, ObjectType } from "type-graphql";

@ObjectType()
@Entity()
export class User {
  @Field()
  @PrimaryKey()
  id!: number;

  @Field(() => String)
  @Property({ type: "date" })
  createdAt = new Date();

  @Field(() => String)
  @Property({ type: "date", onUpdate: () => new Date() })
  updatedAt = new Date();

  @Field()
  @Property({ type: "text", unique: true })
  username!: string;

  @Field()
  @Property({ type: "text", unique: true })
  email!: string;

  @Property({ type: "text" })
  password!: string;
}
```

8. Resolver - user.ts / Mutation - register()

#graphql #resolver #authentication #mutation #mikroorm #backend

---

```
yarn add argon2
```

- argon2 will be used for hashing passwords in the resolver

- We create a **UsernamePasswordInput** class to simplify our code - this will have the InputType() attribute so it can be used in Arg()

/resolvers/UsernamePasswordInput.ts

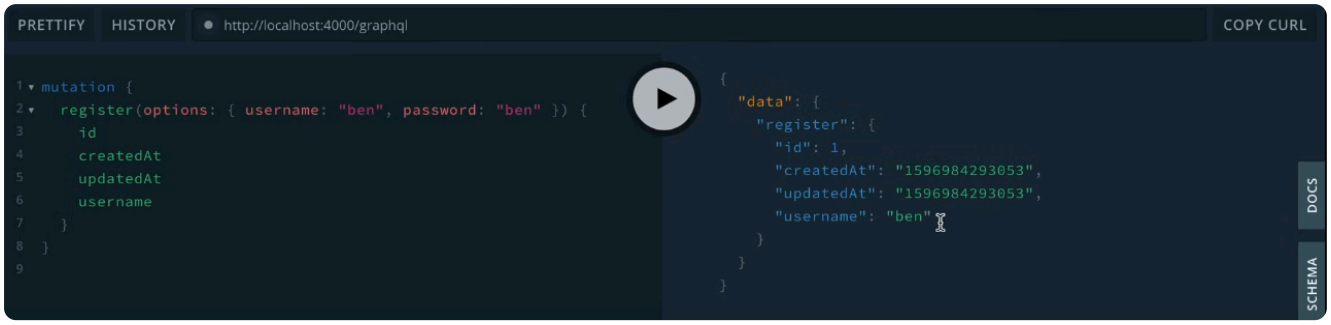
```
import { Field, InputType } from "type-graphql";

@InputType() // InputType are used for arguments
export class UsernamePasswordInput {
  @Field()
  username: string;
  @Field()
  email: string;
  @Field(() => String) // can set type explicitly, or let typescript infer it
  password: string;
}
```

/resolvers/user.ts

```
import { User } from "../entities/User";
import { MyContext } from "src/types";
import {Arg, Ctx, Field, Mutation, Query, Resolver} from "type-graphql";
import argon2 from "argon2";
import { UsernamePasswordInput } from "./UsernamePasswordInput";

@Resolver()
export class UserResolver {
  @Mutation(() => UserResponse)
  async register(
    @Arg("options") options: UsernamePasswordInput, //  let typescript infer type
```

```
UsernamePasswordInput
    @Ctx() { em }: MyContext
  ): Promise<User> {
    const hashedPassword = await argon2.hash(options.password);
    const user = em.create(User, {
     username: options.username,
     password: hashedPassword,
  })
  await em.persistAndFlush(user);
  return user;
  }
}
```

This is the simplest version of this resolver with only register functionality and no error checking etc.



9. Resolver - user.ts / Mutation - login()

#graphql #resolver #authentication #mutation #mikroorm #backend

- Update the user resolver, adding the login() mutation, as shown below:
- An @ObjectType can be defined to be used as return values from mutations and queries
- An @InputType can be defined to be passed into a mutation or query as an input variable

/resolvers/user.ts

```
@ObjectType() // ObjectTypes are returned from Queries and Mutations
class FieldError {
  @Field()
  field: string; // which field the error is about
  @Field()
  message: string; // error message
}

@ObjectType()
class UserResponse {
  @Field(() => [FieldError], { nullable: true })
  errors?: FieldError[];

  @Field(() => User, { nullable: true })
  user?: User;
}



@Resolver()
export class UserResolver {

  @Mutation(() => UserResponse)
  async login(
    @Arg("usernameOrEmail") usernameOrEmail: string,
    @Arg("password") password: string,
    @Ctx() { em, req }: MyContext
  ): Promise<UserResponse> {
    const user = await em.findOne(
      User,
```

```
      usernameOrEmail.includes("@")
        ? { email: usernameOrEmail }
        : { username: usernameOrEmail }
    );
    if (!user) {
      return {
        errors: [
          {
            field: "usernameOrEmail",
            message: "That username or email does not exist",
          },
        ],
      };
    }

    const isPasswordValid = await argon2.verify(user.password, password);
    if (!isPasswordValid) {
      return {
        errors: [
          {
            field: "password",
            message: "Incorrect password",
          },
        ],
      };
    }

    req.session.userId = user.id; // created new type for req in types.ts (5) to
make this work, so the session can store the userId

    return { user };
  }

  return user;
  }
}
```

```
mutation {
  login(options: { username: "ben", password: "ben" }) {
    errors {
      field
      message
    }
    user {
      id
      username
    }
  }
}
```

```
{
  "data": {
    "login": {
      "errors": null,
      "user": {
        "id": 1,
        "username": "ben"
      }
    }
  }
}
```

```
mutation {
  login(options: { username: "ben", password: "bedn" }) {
    errors {
      field
      message
    }
    user {
      id
      username
    }
  }
}
```

```
{
  "data": {
    "login": {
      "errors": [
        {
          "field": "password",
          "message": "incorrect password"
        }
      ],
      "user": null
    }
  }
}
```

```
mutation {
  login(options: { username: "been", password: "bedn" }) {
    errors {
      field
      message
    }
    user {
      id
      username
    }
  }
}
```

```
{
  "data": {
    "login": {
      "errors": [
        {
          "field": "username",
          "message": "that username doesn't exist"
        }
      ],
      "user": null
    }
  }
}
```

10. Resolver - user.ts / Mutation - register()

#graphql #resolver #authentication #mutation #mikroorm #backend

Validating the data

- First we have a utility function for validating the registration username and password :

utils/validateRegister.ts

```
import { UsernamePasswordInput } from "src/resolvers/UsernamePasswordInput";

export const validateRegister = (options: UsernamePasswordInput) => {
  if (!options.email.includes("@")) {
    return [
      {
        field: "email",
        message: "invalid email",
      },
    ];
  }

  if (options.username.length <= 2) {
    return [
      {
        field: "username",
        message: "Length must be greater than 2",
      },
    ];
  }

  if (options.username.includes("@")) {
    return [
      {
        field: "username",
        message: "Username cannot include an '@' symbol",
      },
    ];
  }
```

```
  if (options.password.length <= 3) {
    return [
      {
        field: "password",
        message: "Length must be greater than 3",
      },
    ];
  }

  return null;
};
```

A better register resolver

- Here we first validate the data entered
- And then use `createQueryBuilder()` to insert the data into DB because `persistAndFlush()` causes an error (haha)
- We also catch the `err.code === "23505"` error (duplicate username)

resolvers/user.ts

```
import { validateRegister } from "../utils/validateRegister";
import { EntityManager } from "@mikro-orm/postgresql";

@Mutation(() => UserResponse)
  async register(
    @Arg("options") options: UsernamePasswordInput, //  let typescript infer type
UsernamePasswordInput
    @Ctx() { em, req }: MyContext
  ): Promise<UserResponse> {
    const errors = validateRegister(options);
    if (errors) {
      return { errors };
    }

    const hashedPassword = await argon2.hash(options.password);

    let user;
    try {
```

```
      const result = await (em as EntityManager)
        .createQueryBuilder(User)
        .getKnexQuery()
        .insert({
          username: options.username,
          password: hashedPassword,
          email: options.email,
          created_at: new Date(), // mikroORM adds the underscores in DB so we
must write it like this with Knex
          updated_at: new Date(),
        })
        .returning(["*", "created_at as createdAt", "updated_at as updatedAt"]);
      user = result[0];
    } catch (err) {
      // duplicate username error
      if (err.code === "23505") {
        return {
          errors: [
            {
              field: "username",
              message: "That username is already taken",
            },
          ],
        };
      }
    }

    req.session.userId = user.id; // logs in the user (by sending cookie to
browser)
    return { user };
  }
```

11. Cookie w/ Session and Redis

#express #express-session #redis #cookie #backend

How does it work

- Redis is an in-memory key-value data store.
- When `req.session.userId = user.id` is executed, { userId : 1 } is sent to Redis, and a key is defined, which could look something like this:

sessxasdljhsafliegheginen which maps to { userId: 1 }

- Then **express-session will set a cookie on the browser**: q986hfqkfbjqwl8763487355839
- When user makes a request this cookie q986hfqkfbjqwl8763487355839 is sent to server
- Server decrypts the cookie using the secret defined in session configuration in index.ts, to obtain the redis key

q986hfqkfbjqwl8763487355839 ---decrypt--> sessxasdljhsafliegheginen

- Server makes a request to redis and looks up for the value matching the key sessxasdljhsafliegheginen

sessxasdljhsafliegheginen maps to { userId: 1 }

Install packages

```
yarn add express-session redis connect-redis ioredis
yarn add -D @types/express-session @types/redis @types/connect-redis
@types/ioredis
```

- We will use express-session to keep the user logged in. This stores data (cookie) about user on the server.
- We will store this data in redis, which is a very lightweight and fast in-memory database
- github.com/expressjs/session lists other ways this data can be stored (postgreSQL, MongoDB, etc...)
- We install ioredis library and import Redis from there instead of "redis", since "redis" is crap, does not have Promises built into it etc. (at the time of this tutorial)

Setup session with redis and update apollo-server context

We start by updating index.ts amd setting up redis store and using apollo-server to make the session available in the resolvers via context, by passing the {req, res} provided by express into the context:

- session is accessed via req as req.session

- `req.session.userId = user.id` logs in the user (by sending cookie to browser). It is implemented in register and login queries
- An extended type is defined in types.ts (5) to add session.userId to req
- https://expressjs.com/en/api.html#req
- https://expressjs.com/en/api.html#res
- https://stackoverflow.com/questions/4696283/what-are-res-and-req-parameters-in-express-functions

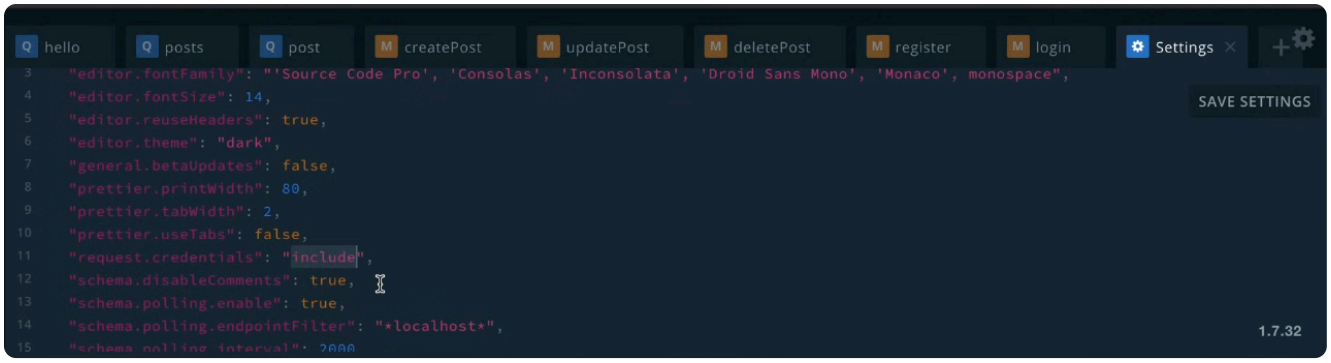
index.ts

```
import "reflect-metadata";
import { MikroORM } from "@mikro-orm/core";
import { COOKIE_NAME, __prod__ } from "./constants";
import microConfig from "./mikro-orm.config";
import express from "express";
import { ApolloServer } from "apollo-server-express";
import { buildSchema } from "type-graphql";
import { HelloResolver } from "./resolvers/hello";
import { PostResolver } from "./resolvers/post";
import { UserResolver } from "./resolvers/user";
import connectRedis from "connect-redis";
import session from "express-session";
import Redis from "ioredis";
import { MyContext } from "./types";

const main = async () => {
  const orm = await MikroORM.init(microConfig);
  await orm.getMigrator().up(); // run migration

  const app = express();

  const RedisStore = connectRedis(session);
  const redis = new Redis();

  // Initialize session storage before Apollo since it will be used from inside
Apollo.
  app.use(
    session({
      name: COOKIE_NAME,
      store: new RedisStore({
        client: redis,
        disableTTL: true, // keep session alive forever
```

```
        disableTouch: true, // disable TTL reset at every touch
      }),
      cookie: {
        maxAge: 1000 * 60 * 60 * 24 * 365 * 10, // 10 years
        httpOnly: true, // prevent accessing the cookie in the JS code in the
frontend
        sameSite: "lax", // csrf
        secure: __prod__, // cookie only works in https
      },
      saveUninitialized: false,
      secret: "asdfasdfasdf", // used to sign cookie - should actually be hidden
in an env variable
      resave: false,
    })
  );

  const apolloServer = new ApolloServer({
    schema: await buildSchema({
      resolvers: [HelloResolver, PostResolver, UserResolver],
      validate: false,
    }),
    context: ({ req, res }: MyContext) => ({ em: orm.em, req, res, redis }), //
context is shared with all resolvers
  });

  apolloServer.applyMiddleware({
    app,
  });

  app.listen(4000, () => {
    console.log("server started on localhost:4000");
  });
};

main().catch((err) => {
  console.log(err);
});
```

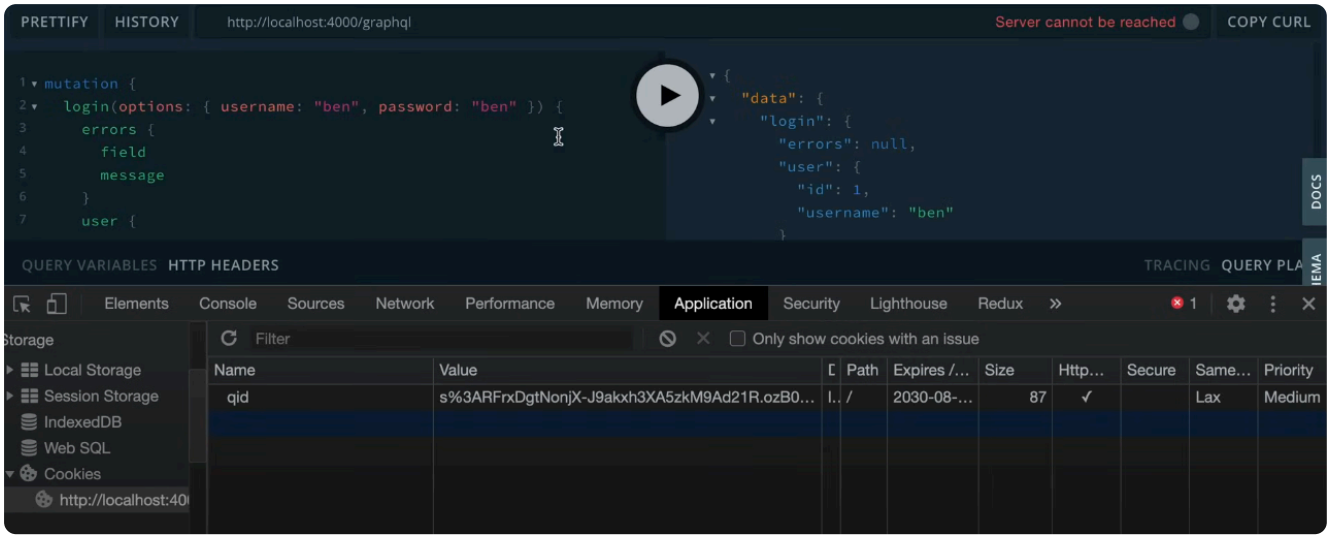
Change GraphQL config

- In GraphQL, change the request.credentials setting from “ommit” to “include”



Result

And now when you register or log in you should see the cookie named “qid” placed in the browser:



12. Resolver - user.ts / Query - me()

#graphql #resolver #authentication #query #mikroorm #backend

This query returns the current user that is logged in and null if you're not logged in

resolvers/user.ts

```
@Query(() => User, { nullable: true })
  async me(@Ctx() { em, req }: MyContext) {
    // you are not logged in
    if (!req.session.userId) {
      return null;
    }

    const user = await em.findOne(User, { id: req.session.userId });
    return user;
  }
```

13. NextJs with ChakraUI

#nextjs #chakraui #frontend

```
yarn create next-app --example with-chakra-ui <folder-name>
```

```
yarn add --dev typescript @types/node
```

- Cleanup _app.tsx and index.tsx to the simplest forms:

pages/_app.tsx

```
import { ChakraProvider } from "@chakra-ui/react";
import { AppProps } from "next/app";
import theme from "../theme";

function MyApp({ Component, pageProps }: AppProps) {
  return (
    <ChakraProvider theme={theme}>
      <Component {...pageProps} />
    </ChakraProvider>
  );
}

export default MyApp;
```

pages/index.tsx

```
const Index = () => {
  return (
      <div>Hello World!</div>
  );
};
```

14. Component - Wrapper

#component #reactjs #chakraui #frontend #component

- This is a <Box> component that will wrap other elements to give the UI a more uniform and tidy look
- The <Box> element in chakraUI is like a but you can style it anyway you want

components/Wrapper.tsx

```
import { Box } from '@chakra-ui/react'
import React from 'react'

interface WrapperProps {
  variant?: 'small' | 'regular',
  children: any
}

export const Wrapper: React.FC<WrapperProps> = ({children, variant='regular'}) =>
{
  return (
    <Box mt={8}
      mx="auto"
      maxW={variant === 'regular' ? "800px" : "400px"}
      w="100%">
      {children}
    </Box>
    )
}
```

15. Component - InputField

#component #reactjs #chakraui #formik #frontend #component

- A reusable React component utilizing chakraUI for text input
- InputFieldProps type is defined to pass props into useField() - which requires { name: string }
- https://formik.org/docs/api/useField
- `field`: An object containing `onChange`, `onBlur`, `name`, and `value` of the field - https://formik.org/docs/api/field
- htmlFor attribute of FormLabel element and id attribute of Input element should be the same

components/InputField.tsx

```
import {
  FormControl,
  FormLabel,
  Input,
  FormErrorMessage,
} from "@chakra-ui/react";
import { useField } from "formik";
import React, { InputHTMLAttributes } from "react";

type InputFieldProps = InputHTMLAttributes<HTMLInputElement> & {
  label: string;
  name: string;
};

export const InputField: React.FC<InputFieldProps> = ({
  size, // <Input> does not want size to be passed into it so we take it out of
props
  label,
  ...props
}) => {
  const [field, { error }] = useField(props);

  return (
    <FormControl isInvalid={!!error}>
      <FormLabel htmlFor={field.name}>{label}</FormLabel>
      <Input {...field} {...props} id={field.name} />
```

```
      {error && <FormErrorMessage>{error}</FormErrorMessage>}
    </FormControl>
  );
};
```

16. Page - register.tsx

#formik #reactjs #nextjs #chakraui #frontend #page

- Formik is an awesome open-source form library for React - https://formik.org/

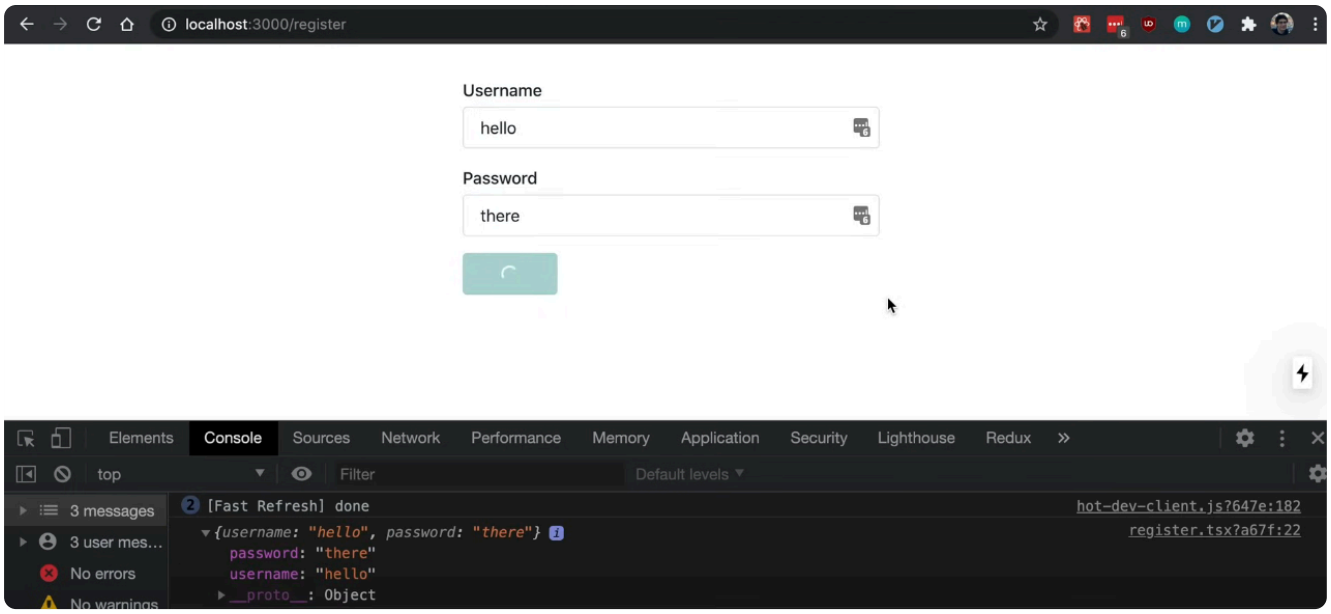
```
yarn add formik
```

pages/register.tsx

```
import React from "react";
import { Form, Formik } from "formik";
import { Button, Box } from "@chakra-ui/react";
import { Wrapper } from "../components/Wrapper";
import { InputField } from "../components/InputField";

interface registerProps {}

const Register: React.FC<registerProps> = ({}) => {

  return (
    <Wrapper variant="small">
      <Formik // initalValues, onsubmit, setErros provided by Formik, values is
inferred from initialValues
        initialValues = {{ username: "", email: "", password: "" }}
        onSubmit={(values) =>  {
          console.log(values)
        }}
      >
        {(
          { isSubmitting } // isSubmitting is provided by Formik
        ) => (
          <Form>
            <InputField
              name="username"
              label="Username"
              placeholder="Username"
            />
```

```
            <Box mt={4}>
              <InputField
                name="email"
                label="Email"
                placeholder="Email"
                type="email"
              />
            </Box>
            <Box mt={4}>
              <InputField
                name="password"
                label="Password"
                placeholder="Password"
                type="password"
              />
            </Box>
            <Button mt={4} type="submit" isLoading={isSubmitting} color="teal">
              Register
            </Button>
          </Form>
        )}
      </Formik>
    </Wrapper>
  );
};

export default Register
```

And we have a register page: (that console.logs the inputs for now)


localhost:3000/register
Username
hello
Password
there
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Redux
top
Filter
Default levels
3 messages
3 user mes...
No errors
No warnings
[Fast Refresh] done
hot-dev-client.js?647e:182
{username: "hello", password: "there"}
register.tsx?a67f:22
password: "there"
username: "hello"
__proto__: Object

17. Urql for GraphQL requests

#graphql #urql #frontend

- To make GraphQL requests we will use urql graphql client
- formidable.com/open-source/urql

Install packages

```
yarn add urql graphql
```

Initial setup

- Wrap the app with the urql client

pages/_app.tsx

```
import { Provider, createClient } from 'urql'

function MyApp({ Component, pageProps }: AppProps) {
  const client = createClient({ url: "http://localhost:4000/graphql" })

  return (
    <Provider value={client}>
      <ChakraProvider theme={theme}>
        <Component {...pageProps} />
      </ChakraProvider>
    </Provider>
  );
}
```

- Copy the Register mutation from the GraphQL playground (http://localhost:4000/graphql) into useMutation() in the code
- Update onSubmit() inside the Formik form

pages/register.tsx

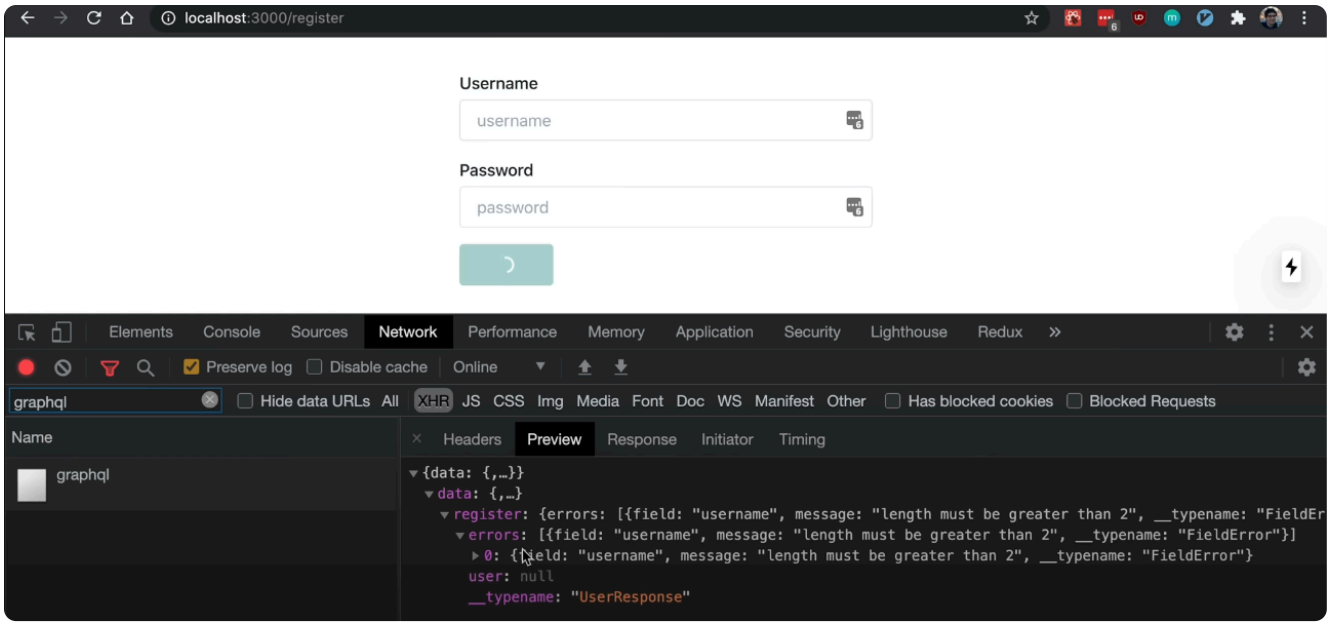
```
import React from "react";
import { Form, Formik } from "formik";
import { Wrapper } from "../components/Wrapper";
import { InputField } from "../components/InputField";
import { useMutation } from "urql";
```

```
const REGISTER_MUT = "mutation Register($username: String!, $email: String!,
$password:String!) {
  register(options: { username: $username, email: $email, password: $password })
{
    errors {
      field
      message
    }
    user {
      id
      username
    }
  }
}"
```

```
const Register: React.FC<registerProps> = ({}) => {

  const [, register] = useMutation(REGISTER_MUT);

  return (
    <Wrapper variant="small">
      <Formik // initalValues, onSubmit, setErrors provided by Formik, values is
inferred from initialValues
        initialValues={{ username: '', email: '', password: '' }}
        onSubmit={(values) => {
          return register(values)
        }}
      >
        {(
          { isSubmitting } // isSubmitting is provided by Formik
        ) => (
          <Form>
            <InputField
              name="username"
              label="Username"
```

```
              placeholder="Username"
            />
            <Box mt={4}>
              <InputField
                name="email"
                label="Email"
                placeholder="Email"
                type="email"
              />
            </Box>
            <Box mt={4}>
              <InputField
                name="password"
                label="Password"
                placeholder="Password"
                type="password"
              />
            </Box>
            <Button mt={4} type="submit" isLoading={isSubmitting} color="teal">
              Register
            </Button>
          </Form>
        )}
      </Formik>
    </Wrapper>
  );
};
```

- Now the register button sends a request to the GraphQL API which in turn executes the register mutation


localhost:3000/register
Username
username
Password
password
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Redux
Preserve log
Disable cache
Online
graphql
Hide data URLs
All
XHR
JS
CSS
Img
Media
Font
Doc
WS
Manifest
Other
Has blocked cookies
Blocked Requests
Name
graphql
Headers
Preview
Response
Initiator
Timing
{data: {,…}}
data: {,…}
register: {errors: [{field: "username", message: "length must be greater than 2", __typename: "FieldEr
errors: [{field: "username", message: "length must be greater than 2", __typename: "FieldError"}]
0: {field: "username", message: "length must be greater than 2", __typename: "FieldError"}
user: null
__typename: "UserResponse"

18. Sending the Cookie and resolving the CORS error

#cookie #cors #frontend

- Add fetchOptions to the urql client

_app.tsx

```
const client = createClient({
  url: 'http://localhost:4000/graphql',
  fetchOptions: 'include' as const
})
```

- Now we get a #cors error "The value of the 'Access-Control-Allow-Origin' header in the response must not be the wildcard '*' when the request's credentials mode is 'include'

```
yarn add cors
yarn add -D @types/cors
```

then update **(server)/index.ts**

```
import cors from "cors";

// ...

const RedisStore = connectRedis(session);
const redis = new Redis();

// define CORS to avoid CORS errors (global solution)
  app.use(
    cors({
      origin: "http://localhost:3000",
      credentials: true,
    })
  );

// Initialize session storage before Apollo since it will be used from inside
Apollo.
```

```
// ...

apolloServer.applyMiddleware({
    app,
    cors: false,
  });
```

- Now cors error disappears

19. GraphQL Mutation - register / graphql-codegen for simpler urql

#urql #graphql #graphql-codegen #mutation #frontend

We set up urql usage and register() mutation in 17. Urql for GraphQL requests but the response for the register() call is <any> and we don't want that

```
<Wrapper variant="small">
  <Formik
    initialV
    onSubmit
      const response = await register(values);
    }}
```

```
const response: OperationResult<any>
'response' is declared but its value is never read. ts(6133)
Quick Fix... (⌘.)
```

Install new packages

We will use `graphql-code-generator` (graphql-code-generator.com)

```
yarn add -D @graphql-codegen/cli
```

```
yarn graphql-codegen init
```

1. The application is of type React
2. The schema is at http://localhost:4000/graphql
3. The operations and fragments will be sotred at src/graphql/**/*.graphql
4. Plugins: TypeScript and TypeScript Operations
5. Output location: The default value should be fine
6. No instrospection file needed
7. Name of the config file codegen.,yml
8. Name of the script to run codegen: gen

After the codegen.,yml config file is created, edit it and add "typescript-urql" to plugins

codegen.yml

```
overwrite: true
schema: "http://localhost:4000/graphql"
```

```
documents: "./src/graphql/**/*.graphql"
generates:
  src/generated/graphql.tsx:
    plugins:
      - "typescript"
      - "typescript-operations"
      - "typescript-urql"
```

- Install the package:

```
yarn add -D @graphql-codegen/typsecsript-urql
```

Add the mutation to generate

- Create the folders src/graphql/mutations and src/graphql/queries
- Install the GraphQLfor VSCode (by Kumar Harsh) extension if it's not already installed to get graphql syntaz highlighting
- And now create register.graphql file in mutations folder and copy our register mutation into it as such:
- Note that instead of giving parameters separately to Register() we make use of the userNamePasswordInput @InputType we defined in 8. Resolver - user.ts / Mutation - register()

/graphql/mutations/register.graphql

```
mutation Register($options: userNamePasswordInput!) {
  register(options: $options) {
    errors {
      field
      message
    }
    user {
      id
      username
    }
  }
}
```

Generate TypeScript code for the Mutation

```
yarn gen
```

- This will run the generator and place the generated TypeScript code in the src/generated/graphql.tsx file
- The most important bit here will be the **useRegisterMutation** custom hook, at the end of the file:

generated/graphql.tsx

```
export function useRegisterMutation() {
  return Urql.useMutation<RegisterMutation, RegisterMutationVariables>
(RegisterDocument);
};
```

Use the custom hook for the request

- Now we can update register.tsx to use this custom hook that was generated

pages/register.tsx

```
import React from "react";
import { Form, Formik } from "formik";
import { Button, Box } from "@chakra-ui/react";
import { Wrapper } from "../components/Wrapper";
import { InputField } from "../components/InputField";
import { useRegisterMutation } from "../generated/graphql";

const Register: React.FC<registerProps> = ({}) => {

  const [, register] = useRegisterMutation();

  return (
    <Wrapper variant="small">
      <Formik // initalValues, onSubmit, setErrors provided by Formik, values is
inferred from initialValues
        initialValues={{ username: '', email: '', password: '' }}
        onSubmit={(values) => {
          return register({ options: values })
        }}
```

```
      >
        {(
          { isSubmitting } // isSubmitting is provided by Formik
        ) => (
          <Form>
            <InputField
              name="username"
              label="Username"
              placeholder="Username"
            />
            <Box mt={4}>
              <InputField
                name="email"
                label="Email"
                placeholder="Email"
                type="email"
              />
            </Box>
            <Box mt={4}>
              <InputField
                name="password"
                label="Password"
                placeholder="Password"
                type="password"
              />
            </Box>
            <Button mt={4} type="submit" isLoading={isSubmitting} color="teal">
              Register
            </Button>
          </Form>
        )}
      </Formik>
    </Wrapper>
  );
};
```

The return type of the custom hook is well-defined

- Now, the type of the response object that is returned from the register() call is not <any> but <RegisterMutation>:

```
const Register: React.FC<registerProps> = ({}) => {
  const [, register] = useRegisterMutation();
  return (
    <Wrapper variant="small">
      <Formik
        initialV
        onSubmit
          const response = await register(values);
        }}
```

```
const response: OperationResult<RegisterMutation>
'response' is declared but its value is never read. ts(6133)
Quick Fix... (⌘.)
```

- And it is possible to look into the response object, since the IDE it knows what's in there:

```
  response.data.register.
  }}
>
  {({ isSubmitting }) => (
```

```
__typename?
errors?          (property) err…
user?
```

Conclusion

- Now everytime we want to add a new graphql query or mutation, we can put the code in the graphql/mutations or graphql/queries folder and easily generate the custom hook for that mutaion or query with `yarn gen`

20. Handling errors during register() call

#formik #graphql-codegen #mutation #frontend

- We have the onSubmit() function as follows:

pages/register.tsx

```
<Formik // initalValues, onsubmit, setErros provded by Formik, values is inferred from initialValues
  initialValues={{ username: "", email: "", password: "" }}
  onSubmit={async (values, { setErrors }) => {
    const response = await register({ options: values });
  }}
```

- Formik provides a setErrors() function that we will use. However it expects a parameter in the following shape:

```
setErrors({
  userName: "hey I'm an error",
})
```

- userName can be any of the initialValues that was defined in <Formik>
- However, if we go deeper into the response objest, the `response.data?.register.errors` has the type FieldError which has the foollowing shape:

```
[{field: "username", message: "hey I'm an error"}]
```

- So we write a utility function to transform this FieldError type into the shape that setErrors() expects
- FieldError type is already defined in the generated GraphQL typescript code so we can easily import it from there

utils/toErrorMap.ts

```
import { FieldError } from "../generated/graphql";

export const toErrorMap = (errors: FieldError[]) => {
  const errorMap: Record<string, string> = {};
```

```
  errors.forEach(({ field, message }) => {
    errorMap[field] = message;
  });

  return errorMap;
};
```

- And we update onSubmit() as follows

pages/register.tsx

```
<Formik // initalValues, onSubmit, setErrors provded by Formik, values is inferred from initialValues
  initialValues={{ username: "", email: "", password: "" }}
  onSubmit={async (values, { setErrors }) => {
    const response = await register({ options: values });
    if (response.data?.register.errors) {
      setErrors(toErrorMap(response.data.register.errors));
    } else if (response.data?.register.user) {
      // register success
      console.log("registration succesful")
    }
  }}
>
```

- Note that we also check `response.data?.register.user` is not undefined to see if registration was successful

21. Routing in NextJS with useRouter()

#nextjs #routing #useRouter #frontend

- Routing with NextJS is very easy with the useRouter() hook:

```
import useRouter from "next/router"

const router = useRouter();
router.push("/");
```

- `router.push("/")` routes to the root `router.push("/pageName")` routes to the pageName

- Using the useRouter() hook, we can update register.tsx to route to homepage on a successful registration:

pages/register.tsx

```
import useRouter from "next/router"

const Register: React.FC<registerProps> = ({}) => {
  const router = useRouter();
  const [, register] = useRegisterMutation();

  return (
    <Wrapper variant="small">
      <Formik // initalValues, onsubmit, setErros provded by Formik, values is
inferred from initialValues
        initialValues={{ username: "", email: "", password: "" }}
        onSubmit={async (values, { setErrors }) => {
          const response = await register({ options: values });
          if (response.data?.register.errors) {
            setErrors(toErrorMap(response.data.register.errors));
          } else if (response.data?.register.user) {
            // register success, route to homepage
            router.push("/");
          }
```

```
        }}
      >
```

22. GraphQL Mutation - login w/ Fragments

#urql #graphql #graphql-codegen #fragment #mutation #frontend

Login mutation shape

- Login mutation looks like below, but we can make it more compact and readable by using fragments

```
mutation Login($usernameOrEmail: String!, $password: String!) {
  login(usernameOrEmail: $usernameOrEmail, password: $password) {
    errors {
      field
      message
    }
    user {
      id
      username
    }
  }
}
```

- Fragment is a GraphQL thing. The idea of fragments is quite simple.
- Here we implement the fragments **based on** ObjectTypes that are already defined in the server code:
  - FieldError and UserResponse are ObjectTypes defined in user.ts resolver (9. Resolver - user.ts / Mutation - login())
  - User is an ObjectType defined in User.ts entity (7. Entity - User.ts)
- We implement the following files:

/graphql/fragments/RegularError.graphql

```
fragment RegularError on FieldError {
  field
  message
}
```

/graphql/fragments/RegularUser.graphql/

```
fragment RegularUser on User {
  id
  username
}
```

- Use these two fragments to implement another fragment that combines them

/graphql/fragments/RegularUserResponse.graphql

```
fragment RegularUserResponse on UserResponse {
  errors {
    ...RegularError
  }
  user {
    ...RegularUser
  }
}
```

Add login mutation

- Now use the RegularUserResponse fragment to implement the Login mutation

/graphql/mutations/login.graphql

```
mutation Login($usernameOrEmail: String!, $password: String!) {
  login(usernameOrEmail: $usernameOrEmail, password: $password) {
    ...RegularUserResponse
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the useLoginMutation() hook in generated/graphql.tsx that we can use

generated/graphql.tsx

```
export function useLoginMutation() {
  return Urql.useMutation<LoginMutation, LoginMutationVariables>(LoginDocument);
};
```

23. Page - login.tsx

#formik #reactjs #nextjs #chakraui #frontend #page

Implement login page

- Now that the useLoginMutation() hook is set up, we can implement the login page itself, which will be very similar to the register page

pages/login.tsx

```
import React from "react";
import { Form, Formik } from "formik";
import { Button, Box } from "@chakra-ui/react";
import { Wrapper } from "../components/Wrapper";
import { InputField } from "../components/InputField";
import { useLoginMutation } from "../generated/graphql";
import { toErrorMap } from "../utils/toErrorMap";
import { useRouter } from "next/router";

const Login: React.FC<{}> = ({}) => {
  const router = useRouter();
  const [, login] = useLoginMutation();

  return (
    <Wrapper variant="small">
      <Formik
        initialValues={{ usernameOrEmail: "", password: "" }}
        onSubmit={async (values, { setErrors }) => {
          const response = await login(values);
          if (response.data?.login.errors) {
            setErrors(toErrorMap(response.data.login.errors));
          } else if (response.data?.login.user) {
            router.push("/");
          }
        }}
      >
        {({ isSubmitting }) => (
          <Form>
```

```
            <InputField
              name="usernameOrEmail"
              label="Username or Email"
              placeholder="Username or Email"
            />
            <Box mt={4}>
              <InputField
                name="password"
                label="Password"
                placeholder="Password"
                type="password"
              />
            </Box>
            <Button type="submit" isLoading={isSubmitting} color="teal">
              Login
            </Button>
          </Form>
        )}
      </Formik>
    </Wrapper>
  );
};

export default Login;
```

24. GraphQL Query - me

#graphql #graphql-codegen #query #frontend

Add me query

- the RegularUser fragment was added in [22. GraphQL Mutation - login_w/ Fragments]() and we use it here again

/graphql/queries/me.graphql

```
query Me {
  me {
    ...RegularUser
  }
}
```

```
yarn gen
```

- and now we have userMeQuery() hook in /generated/graphql.tsx

25. Resolver user.ts / Mutation - logout()

#graphql #resolver #authentication #mutation #backend

- Update the user resolver, adding the logout() mutation, as shown below:

/resolvers/user.ts

```
@Resolver()
export class UserResolver {

  @Mutation(() => Boolean)
  async logout(@Ctx() { req, res }: MyContext): Promise<Boolean> {
    // clear the user's cookie
    res.clearCookie(COOKIE_NAME);

    // clear the redis record
    return new Promise((resolve) =>
      // remove the session from redis
      req.session.destroy((err) => {
        if (err) {
          console.log(err);
          resolve(false);
          return;
        }
        resolve(true);
      })
    );
  }
}
```

26. GraphQL Mutation - logout

#urql #graphql #graphql-codegen #mutation #frontend

Add logout mutation

/graphql/mutations/logout.graphql

```
mutation Logout() {
  logout
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the useLogoutMutation() hook in generated/graphql.tsx that we can use

generated/graphql.tsx

```
export function useLogoutMutation() {
  return Urql.useMutation<LogoutMutation, LogoutMutationVariables>
(LogoutDocument);
};
```

27. Component - Navbar

#reactjs #frontend #chakraui #component

- We will check if user is logged in or not using the me() query .
- If user is not logged in we display login and register buttons which route to their respective pages
- If the user ***is*** logged in, we display a logout button, utilizing the logout mutation
- Read the code, it's pretty self-explanatory

/components/NavBar.tsx

```
import { Box, Button, Flex, Link } from "@chakra-ui/react";
import NextLink from "next/link";
import React from "react";
import { useLogoutMutation, useMeQuery } from "../generated/graphql";

interface NavBarProps {}

export const NavBar: React.FC<NavBarProps> = ({}) => {
  const [{ fetching: logoutFetching }, logout] = useLogoutMutation();

  const [{ data, fetching }] = useMeQuery({});

  let body = null;

  // data is loading
  if (fetching) {
    body = "Loading...";
    // user is not logged in
  } else if (!data?.me) {
    body = (
      <>
        <Link as={NextLink} href="/login" mr={4} color="white">
          Login
        </Link>

        <Link as={NextLink} href="/register" mr={4} color="white">
          Register
        </Link>
```

```
      </>
    );
    // user is logged in
  } else {
    body = (
      <Flex>
        <Box mr={4} color="white">
          {data.me.username}
        </Box>
        <Button
          variant="link"
          isLoading={logoutFetching}
          onClick={() => logout()}
        >
          Logout
        </Button>
      </Flex>
    );
  }

  return (
    <Flex bg="tan" p={4}>
      <Box ml={"auto"} suppressHydrationWarning>
        {body}
      </Box>
    </Flex>
  );
};
```

---

* Add navbar to the homepage

pages/index.tsx

```
const Index = () => {
  const [{ data }] = usePostsQuery();
  return (
    <>
      <NavBar />
      <div>Hello World!</div>
```

```
    </>
  );
};
```

28. Disable urql cache with exchanges

#urql #cache #exchange #mutation #graphql #graphcache #frontend

- urql, by default, caches the data received from GraphQL. This causes problems when we login, logout or register since the page will reload using the data in the cache and will not update properly.
- We will use @urql/exchange-graphcache (formidable.com/open-source/urql/docs/grapchcace/) package to refresh the data everytime a user logins, logouts or registers

```
yarn add @urql/exchange-graphcache
```

- Now we add the exchanges to the createClient() function
- First we need a helper function betterUpdateQuery() because graphcache's updateQuery() function is not very good with types, and is also not very readable

utils/betterUpdateQuery.tsx

```
import { Cache, QueryInput } from "@urql/exchange-graphcache";

export function betterUpdateQuery<Result, Query>( // Query will be updated when
Result mutation is executed
  cache: Cache,
  qi: QueryInput, // input type of Query
  result: any,
  updaterFn: (r: Result, q: Query) => Query
) {
  return cache.updateQuery(qi, (data) => updaterFn(result, data as any) as any);
}
```

- updaterFn() is executed everytime the mutation is executed, in order to update the data of the Query, depending on the Result.
- r is result of the mutation, q is current (cached) state of Query

- The return type of updaterFn() matches the return type of Query. If the Result has errors, the value that is already in the cache is used, if not then the updated data is used (which may or may not be based on the Result)
- betterUpdateQuery() also allows us to properly **cast** the types, as can be seen in the code

update pages/_app.tsx

```
import { Provider, createClient, dedupExchange, fetchExchange } from 'urql'
import {
  LogoutMutation,
  MeQuery,
  MeDocument,
  LoginMutation,
  RegisterMutation,
} from "../generated/graphql";
import { cacheExchange } from "@urql/exchange-graphcache";
import { betterUpdateQuery } from "../utils/betterUpdateQuery";

function MyApp({ Component, pageProps }: AppProps) {
  const client = createClient({
    url: "http://localhost:4000/graphql",
    fetchOptions: {
      credentials: "include" as const,
    },
    exchanges: [
      dedupExchange,
      cacheExchange({
        // this will update the cache everytime the defined mutations are run
        updates: {
          Mutation: {
            logout: (result, args, cache, info) => {
              betterUpdateQuery<LogoutMutation, MeQuery>(
                cache,
                { query: MeDocument }, // e.g. MeQuery's input type is MeDocument
                result,
                () => ({ me: null }) // updaterFn - clear the query
              );
            },
            login: (result, args, cache, info) => {
              // cache.updateQuery({ query: MeDocument }, (data: MeQuery) => { })
              betterUpdateQuery<LoginMutation, MeQuery>(
                cache,
                { query: MeDocument },
                result,
                (r, q) => { // updaterFn
                  if (r.login.errors) {
                    return q; // return the current query if there's error
```

```
                } else {
                  return {
                    me: r.login.user, // return the user info received from
successful login
                  };
                }
              }
            );
          },
          register: (result, args, cache, info) => {
            betterUpdateQuery<RegisterMutation, MeQuery>(
              cache,
              { query: MeDocument },
              result,
              (r, q) => { // updaterFn
                if (r.register.errors) {
                  return q; // return the current query if there's error
                } else {
                  return {
                    me: r.register.user, // return the user info received from
successful register
                  };
                }
              }
            );
          },
        },
      },
    }),
    fetchExchange,
  ],
})

return (
  <Provider value={client}>
    <ChakraProvider theme={theme}>
      <Component {...pageProps} />
    </ChakraProvider>
  </Provider>
```

```
  );
}
```

29. Server Side Rendering with NextJS and urql

#nextjs #urql #ssr #next-urql #frontend

Install packages

- This is also a good time to format the code and take the creation of urql client out of the _app.tsx, and put it into a utility function
- formidable.com/open-source/urql/docs/advanced/server-side-rendering/#nextjs
- What we're doing here is **legacy version** of ssr with urql! Check the docs for the modern implementation

```
yarn add next-urql react-is isomorphic-unfetch
```

Usage of withUrqlClient()

- This is how withUrqlClient() function is used to set up urql provider on the page and the `{ ssr: true }` is added as second parameter to enable SSR for the page:

```
export default withUrqlClient(ssrExchange => ({
  url: 'http://localhost:3000/graphql',
  exchanges: [cacheExchange, ssrExchange, fetchExchange],
}), { ssr: true })(Index);
```

Utility function to create the urql client

- So basically, we copy the code from createClient() in _app.tsx and implement the utility function createUrqlClient() function to pass into wirhUrqlClient() :
- **Note that we add** ssrExchange between cacheExchange and fetchExchange

/utils/createUrqlClient.ts

```
import { dedupExchange, fetchExchange } from "urql";
import {
  LogoutMutation,
  MeQuery,
  MeDocument,
  LoginMutation,
```

```
  RegisterMutation,
} from "../generated/graphql";
import { cacheExchange } from "@urql/exchange-graphcache";
import { betterUpdateQuery } from "./betterUpdateQuery";

export const createUrqlClient = (ssrExchange: any) => ({
  url: "http://localhost:4000/graphql",
  fetchOptions: {
    credentials: "include" as const,
  },
  exchanges: [
    dedupExchange,
    cacheExchange({
      // this will update the cache everytime the defined mutations are run run
      updates: {
        Mutation: {
          logout: (result, args, cache, info) => {
            betterUpdateQuery<LogoutMutation, MeQuery>(
              cache,
              { query: MeDocument }, // e.g. MeQuery's input type is MeDocument
              result,
              () => ({ me: null }) // clear the query
            );
          },
          login: (result, args, cache, info) => {
            // cache.updateQuery({ query: MeDocument }, (data: MeQuery) => { })
            betterUpdateQuery<LoginMutation, MeQuery>(
              cache,
              { query: MeDocument },
              result,
              (r, q) => {
                if (r.login.errors) {
                  return q; // return the current query if there's error
                } else {
                  return {
                    me: r.login.user, // return the user info received from
successful login
                  };
                }
              }
```

```
            );
          },
          register: (result, args, cache, info) => {
            betterUpdateQuery<RegisterMutation, MeQuery>(
              cache,
              { query: MeDocument },
              result,
              (r, q) => {
                if (r.register.errors) {
                  return q; // return the current query if there's error
                } else {
                  return {
                    me: r.register.user, // return the user info received from
successful register
                  };
                }
              }
            );
          },
        },
      },
    }),
    ssrExchange,
    fetchExchange,
  ],
});
```

Setting a page for urql provider

- Set login.tsx and register.tsx use urql provider (but not for SSR)

pages/login.tsx

```
export default withUrqlClient(createUrqlClient)(Login);
```

pages/register.tsx

```
export default withUrqlClient(createUrqlClient)(Register);
```

Setting a page for SSR

- Set index.tsx to use urql provider and to be rendered with SSR

pages/index.tsx

```
export default withUrqlClient(createUrqlClient, { ssr: true })(Index);
```

30. GraphQL Query - posts

#graphql #graphql-codegen #query #frontend

Add posts query

/graphql/queries/posts.graphql

```
query Posts {
 posts {
    id
    createdAt
    updatedAt
    title
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- and now we have userPostsQuery() hook in /generated/graphql.tsx

31. Sleep() - utlity function

#typescript #sleep #frontend

- Utility function to pause execution of code (e.g. to simulate delay to test csr vs ssr load times)

utils/sleep.ts

```
export const sleep = async (ms: number) =>
  new Promise((res) => setTimeout(res, ms));
```

32. isServer() - utility function

#typescript #sleep #ssr #server #query #graphql #frontend

Fix NavBar - SSR problem

- Since NavBar component is displayed on the index.tsx page and SSR is enabled on that page , it is actually going to make a request via useMeQuery(), on the NextJS server., to get the current user. However NextJS server does not have a cookie in this implementation. So we want to prevent it from running useMeQuery() when the code is executed on the NextJS server via SSR.
- We implement a utility function isServer() to check if the code is being executed on the server side or on the client side

utils/isServer.ts

```
export const isServer = () => typeof window === "undefined";
```

- and update useMeQuery() implementation as follows:

components/NavBar.tsx

```
const [{ data, fetching }] = useMeQuery({
    pause: isServer(), // this will prevent the query from running on the server
(there's no cookie on the server to look for)
  });
```

33. Resolver user.ts / Mutation - forgotPassword()

#backend #graphql #resolver #authentication #mutation #email #nodemailer

Install nodemailer on the server

- nodemailer is a package for sending test emails in development, which can be set up to also use other email providers, e.g. gmail, etc.
- nodemailer.com

on the server

```
yarn add nodemailer
yarn add -D @types/nodemailer
```

Implement sendEmail() utility function

- **Implementeed based on the example on nodemailer.com**

server/src/utils/sendEmail.ts

```
import nodemailer from "nodemailer";

export async function sendEmail(to: string, subject: string, text: string) {
  // run with createTestAccount() once, console.log the account, get the password
and use the same account afterwards
  //let testAccount = await nodemailer.createTestAccount();
  //console.log("testAccount: ", testAccount);

  const transporter = nodemailer.createTransport({
    host: "smtp.ethereal.email",
    port: 587,
    secure: false, // Use `true` for port 465, `false` for all other ports
    auth: {
      user: "i3sabqkaflvfyrhh@ethereal.email", //testAccount.user,
      pass: "BS85g71pdwCQDEz5Bz", //testAccount.pass,
    },
  });
```

```
  // send mail with defined transport object
  const info = await transporter.sendMail({
    from: '"Fred Foo 👻" <foo@ethereal.email>',
    to,
    subject,
    html: text,
  });

  console.log("Message sent: %s", info.messageId);
  console.log("PreviewURL: %s", nodemailer.getTestMessageUrl(info));
}
```

Install uuid on the server for token creation

- uuid will be used for creating unique identifiers to be used as a token
- we will store the token in the redis store with expiration time 1 day

```
yarn add uuid
```

/resolvers/user.ts

```
import v4 from "uuid"
```

- Note that it's a good practice to put a prefix in front of the tokens/keys when saving them into redis store so that during development they can be identified easily

constants.tsx

```
export const FORGOT_PASSWORD_PREFIX = "forgot-password:";
```

Add forgotPassword mutation

- Update the user resolver, adding the forgotPassword() mutation, as shown below:
- We basically send the token to user in an email, and when user sends it back to us we look it up in the redis store and if it's valid we let them change password

/resolvers/user.ts

```
@Mutation(() => Boolean)
  async forgotPassword(
    @Arg("email") email: string,
```

```
    @Ctx() { em, redis }: MyContext
  ): Promise<Boolean> {
    const user = await em.findOne(User, { email });
    if (!user) {
      // the email is not in the db
      return true; //  don't let the person know that the email is not in the db
    }

    const token = v4(); // token for resetting pw

    // save token to redis with value userId, expires in 1 day
    await redis.set(
      FORGOT_PASSWORD_PREFIX  + token, // redis key
      user.id,                         // value
      "ex",                            // expiry mode
      1000 * 60 * 60 * 24              // expiration duration - 24 hours
    );
    const resetLink = `<a href="http://localhost:3000/change-
password/${token}">Reset password</a>`;

    sendEmail(email, "Reset Password", resetLink);
    return true;
  }
```

34. GraphQL Mutation - forgotPassword

#urql #graphql #graphql-codegen #mutation #frontend

Add forgotPassword mutation

/graphql/mutations/forgotPassword.**graphql**

```
mutation ForgotPassword() {
  forgotPassword
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the useForgotPasswordMutation() hook in generated/graphql.tsx that we can use

generated/graphql.tsx

```
export function useForgotPasswordMutation() {
  return Urql.useMutation<ForgotPasswordMutation,
ForgotPasswordMutationVariables>(ForgotPasswordDocument);
};
```

35. Page - forgot-password

#formik #reactjs #nextjs #chakraui #frontend #page

Implement forgot-password page

/forgot-password.tsx

```
import { Box, Button, Flex, Text } from "@chakra-ui/react";
import { Form, Formik } from "formik";
import { withUrqlClient } from "next-urql";
import React, { useState } from "react";
import { InputField } from "../components/InputField";
import { Wrapper } from "../components/Wrapper";
import { useForgotPasswordMutation } from "../generated/graphql";
import { createUrqlClient } from "../utils/createUrqlClient";

const ForgotPassword: React.FC<{}> = ({}) => {
  const [complete, setComplete] = useState(false);
  const [, forgotPassword] = useForgotPasswordMutation();
  return (
    <Wrapper variant="small">
      <Formik
        initialValues={{ email: "" }}
        onSubmit={async (values, { setErrors }) => {
          if (!values.email || !values.email.includes("@")) {
            setErrors({ email: "Provide a valid email address" });
          } else {
            await forgotPassword(values);
            setComplete(true);
          }
        }}
      >
        {({ isSubmitting, values }) =>
          complete ? (
            <Flex flexDirection={"column"} alignItems={"center"}>
              <Text>Please check your email address</Text>
              <Text fontWeight={"bold"}>{values.email}</Text>
              <Text>for the password reset link</Text>
```

```
            </Flex>
          ) : (
            <Form>
              <InputField
                name="email"
                label="Email Address"
                placeholder="Email Address"
              />
              <Box mt={4}>
                <Button type="submit" isLoading={isSubmitting} color="teal">
                  Send password reset email
                </Button>
              </Box>
            </Form>
          )
        }
      </Formik>
    </Wrapper>
  );
};

export default withUrqlClient(createUrqlClient)(ForgotPassword);
```

Update login page

- Let's also add a "Forgot Password" button to the login page, right before the login button, and wrap them in Flex

/pages/login.tsx

```
import { Box, Button, Flex, Link } from "@chakra-ui/react";
import NextLink from "next/link";

<Flex mt={4} justify={"space-between"}>
  <Button type="submit" isLoading={isSubmitting} color="teal">
    Login
  </Button>
  <Button type="button" color="red">
    <Link as={NextLink} href="/forgot-password">
```

```
      Forgot Password
    </Link>
  </Button>
</Flex>
```

36. Resolver user.ts / Mutation - changePassword()

#backend #graphql #resolver #authentication #mutation #mikroorm

---

Add changePassword mutation

- Update the user resolver, adding the changePassword() mutation, as shown below:
- We will also log the users in when they change their password
- token is sent to the user in the email in forgotPassword mutation, as the query scetion of the change password URL:
  localhost:3000/change-password/05329837b4-deuh6-rben34-293874yt
- token was also saved in to the Redis store in the forgotPassword mutation
- this token will be retrieved by the front-end (change-password page) and sent to the backend (changePassword mutation), where we will use it to authenticate the password change attempt

/resolvers/user.ts

```
@Mutation(() => UserResponse)
  async changePassword(
    @Arg("token") token: string,
    @Arg("newPassword") newPassword: string,
    @Ctx() { em, redis, req }: MyContext
  ): Promise<UserResponse> {
    if (newPassword.length <= 2) {
      return {
        errors: [
          {
            field: "newPassword", // must match the name of the field on front-
end
            message: "Length must be greater than 3",
          },
        ],
      };
    }

    const tokenKey = FORGOT_PASSWORD_PREFIX + token;
    const userId = await redis.get(tokenKey); // retrieve value for token from
redis
    if (!userId) {
      return {
```

```
        errors: [
          {
            field: "token",
            message: "Token expired",
          },
        ],
      };
    }

    const user = await em.findOne(User, { id: parseInt(userId) });

    if (!user) {
      return {
        errors: [
          {
            field: "token",
            message: "User no longer exists",
          },
        ],
      };
    }

    user.password = await argon2.hash(newPassword); // hash and set the pw in
user
    await em.persistAndFlush(user); // change pw in db
    await redis.del(tokenKey); // delete token so it can't be reused
    req.session.userId = user.id; // log the user in
    return { user };
  }
```

37. GraphQL Mutation - changePassword

#urql #graphql #graphql-codegen #mutation #frontend

Add changePassword mutation

/graphql/mutations/change Password.**graphql**

```
mutation ChangePassword($token: String!, $newPassword: String!) {
  changePassword(token: $token, newPassword: $newPassword) {
    ...RegularUserResponse
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the useChangePasswordMutation() hook in generated/graphql.tsx that we can use

generated/graphql.tsx

```
export function useChangePasswordMutation() {
  return Urql.useMutation<ChangePasswordMutation,
ChangePasswordMutationVariables>(ChangePasswordDocument);
};
```

38. Page - change-password

#formik #reactjs #nextjs #chakraui #frontend #page

- We need a variable (token) in the URL of this page, so that only a user with the token can come here to change password
  i.e. **localhost:3000/change-password/05329837b4-deuh6-rben34-293874yt**
- In NextJS the convention for such pages is to create a folder with the page name and place the file inside this folder
- The file name should be `[variableName].tsx` - in our case it will be [token].tsx
- Note that setErrors() of Formik works automatically and displays the error to user for the newPassword field, but when we have a token error we have to display it to user manually

/change-password/[token].tsx

```
import { Box, Button, Flex, Link } from "@chakra-ui/react";
import { Formik, Form } from "formik";
import { NextPage } from "next";
import { useRouter } from "next/router";
import { InputField } from "../../components/InputField";
import { Wrapper } from "../../components/Wrapper";
import { toErrorMap } from "../../utils/toErrorMap";
import { useChangePasswordMutation } from "../../generated/graphql";
import { useState } from "react";
import { withUrqlClient } from "next-urql";
import { createUrqlClient } from "../../utils/createUrqlClient";
import NextLink from "next/link";

export const ChangePassword: NextPage<{ token: string }> = ({ token }) => {
  const router = useRouter();
  const [, changePassword] = useChangePasswordMutation();
  const [tokenError, setTokenError] = useState("");

  return (
    <Wrapper variant="small">
      <Formik
        initialValues={{ newPassword: "" }}
        onSubmit={async (values, { setErrors }) => {
          const response = await changePassword({
```

```
          newPassword: values.newPassword,
          token,
        });
        if (response.data?.changePassword.errors) {
          const errorMap = toErrorMap(response.data.changePassword.errors);
          if ("token" in errorMap) {
            setTokenError(errorMap.token);
          }
          setErrors(errorMap);
        } else if (response.data?.changePassword.user) {
          router.push("/");
        }
      }}
    >
      {({ isSubmitting }) => (
        <Form>
          <InputField
            name="newPassword"
            label="New Password"
            placeholder="Enter your new password"
            type="password"
          />
          {tokenError && (
            <Flex>
              <Box mr={2} color="red">
                {tokenError}
              </Box>
              <Link as={NextLink} href="/forgot-password">
                click here to get new token
              </Link>
            </Flex>
          )}
          <Button mt={4} type="submit" isLoading={isSubmitting} color="teal">
            Change Password
          </Button>
        </Form>
      )}
    </Formik>
  </Wrapper>
);
```

```
};

ChangePassword.getInitialProps = ({ query }) => {
  return {
    token: query.token as string, // take the token from the query string section
of URL and pass it to ResetPassword page as props
  };
};

export default withUrqlClient(createUrqlClient)(ChangePassword);
```

- NextPage,getInitialProps() allows us to retrieve props from the URL (as well as other props) before rendering the page
- Here we retrieve the token from the query string section of the URL:
  localhost:3000/change-password/05329837b4-deuh6-rben34-293874yt
- ⚠️ **Note that** for #optimization purpose, it's better to retrieve the token using `router.query.token` instead of getInitialProps() since pages without getInitialProps() are optimized as static pages by NextJs

39. Switching to TypeORM from MikroORM

#backend #typeorm #mikroorm #entity #resolver #query #mutation

---

Install TypeORM, uninstall MikroORM

- Due to MikroORM being too abstracted from the database and also not very user-friendly when creating many-to-one relations, we're switching to TypeORM (⚠️ **Note that** TypeORM version 0.2.25 is used in this tutorial. Many things have changed since then and these TypeORM implementations will not work for versions >= 0.3.0)

```
yarn add typeorm
yarn remove @mikro-orm/cli @mikro-orm/core @mikro-orm/migration @mikro-orm/posgresql
```

Initialize TypeORM Connection

- **Note that** TypeORM requires reflect-metadata to work, so we have to import it !

index.ts

```
import "reflect-metadata";
```

- similar to how we set up the connecttion with MikroORM, we will set up connection with TypeORM
- also create a **/src/migrations** folder, to put the custom migrations in later 📄 and point TypeOrm to look in there for migrations
- **Note that** we do not need to pass orm.em to the context anymore

index.ts

```
import "reflect-metadata";
import { COOKIE_NAME, __prod__ } from "./constants";
import { ApolloServer } from "apollo-server-express";
import connectRedis from "connect-redis";
import cors from "cors";
import express from "express";
import session from "express-session";
import Redis from "ioredis";
import { buildSchema } from "type-graphql";
import { createConnection } from "typeorm";
import { Post } from "./entities/Post";
import { User } from "./entities/User";
```

```
import { HelloResolver } from "./resolvers/hello";
import { PostResolver } from "./resolvers/post";
import { UserResolver } from "./resolvers/user";
import { MyContext } from "./types";
import path from "path";
import { Updoot } from "./entities/Updoot";

const main = async () => {
  const conn = await createConnection({
    type: "postgres",
    database: "lireddit2",
    username: "postgres",
    password: "postgres",
    logging: true,
    synchronize: true, // automatically syncs the DB so no need to run migrations
- very useful in development
    migrations: [path.join(__dirname, "./migrations/*")],
    entities: [Post, User],
  });

  await conn.runMigrations();
  const app = express();

  const RedisStore = connectRedis(session);
  const redis = new Redis();

  // define CORS to avoid CORS errors
  app.use(
    cors({
      origin: "http://localhost:3000",
      credentials: true,
    })
  );

  // Initialize session storage before Apollo since it will be used from inside
Apollo.
  app.use(
    session({
      name: COOKIE_NAME,
      store: new RedisStore({
```

```
        client: redis,
        disableTTL: true, // keep session alive forever
        disableTouch: true, // disable TTL reset at every touch
      }),
      cookie: {
        maxAge: 1000 * 60 * 60 * 24 * 365 * 10, // 10 years
        httpOnly: true, // prevent accessing the cookie in the JS code in the
frontend
        sameSite: "lax",
        secure: __prod__, // cookie only works in https
      },
      saveUninitialized: false,
      secret: "asdfasdfasdf", // used to sign cookie - should actually be hidden
in an env variable
      resave: false,
    })
  );

  const apolloServer = new ApolloServer({
    schema: await buildSchema({
      resolvers: [HelloResolver, PostResolver, UserResolver],
      validate: false,
    }),
    context: ({ req, res }: MyContext) => ({ req, res, redis }), // context is
shared with all resolvers
  });

  apolloServer.applyMiddleware({
    app,
    cors: false,
  });

  app.listen(4000, () => {
    console.log("server started on localhost:4000");
  });

main().catch((err) => {
  console.log(err);
});
```

Update Entities from MikroORM to TypeORM

- typeorm.io/#entities
- User and Post entities were tagged with the MikroOrm's @ attributes. We update them to TypeORM as follows
- **Note that** there are specific attributes @CreateDateColumn() and @UpdateDateColum() for date management
- BaseEntity allows Post.find(), Post.insert(), some easy command to be used in SQL
- With TypeOrm we don't need to specify `{ type: "text" }` for string types

/entities/Post.ts

MikroORM	TypeORM

`import { Field, ObjectType } from "type-graphql";`
`import { Entity, PrimaryKey, Property } from "@mikro-orm/core";`
 	
`@ObjectType() // graphQL`
`@Entity() // mikro-orm`
`export class Post {`
`  @Field()`
`  @PrimaryKey()`
`  id!: number;`
 	
`  @Field(() => String) // explicitly set type for GraphQL`
`  @Property({ type: 'date' }) // explicitly set type for MikroORM`
`  createdAt = new Date();`
 	
`  @Field(() => String)`
`  @Property({ type: 'date', onUpdate: () => new Date() })`
`  updatedAt = new Date();`
	`import { Field, ObjectType } from "type-graphql";`
	`import { BaseEntity, Column, CreateDateColumn, Entity, PrimaryGeneratedColumn, UpdateDateColumn } from "typeorm";`
	
	`@ObjectType() // graphQL`
	`@Entity() // typeorm`
	`export class Post extends BaseEntity {`
	`  @Field()`
	`  @PrimaryGeneratedColumn()`
	`  id!: number;`
	
	`  @Field(() => String) // explicitly set type for GraphQL`
	`  @CreateDateColumn()`
	`  createdAt: Date;`
	
	`  @Field(() => String)`
	`  @UpdateDateColumn()`
	`  updatedAt: Date;`
	
	`  @Field()`
	`  @Column()`

```
  @Field()
  @Property({ type: 'text'})
  title!: string;
}
```

```
  title!: string;
}
```

/entities/User.ts

MikroORM	TypeORM

```
import { Field, ObjectType }
from "type-graphql";
import { Entity, PrimaryKey,
Property } from "@mikro-
orm/core";


@ObjectType()
@Entity()
export class User {
  @Field()
  @PrimaryKey()
  id!: number;

  @Field(() => String)
  @Property({ type: "date" })
  createdAt = new Date();

  @Field(() => String)
  @Property({ type: "date",
onUpdate: () => new Date() })
  updatedAt = new Date();

  @Field()
  @Property({ type: "text",
unique: true })
  username!: string;

  @Field()
  @Property({ type: "text",
```

```
import { Field, ObjectType } from
"type-graphql";
import { BaseEntity, Column,
CreateDateColumn, Entity,
PrimaryGeneratedColumn,
UpdateDateColumn } from "typeorm";

@ObjectType()
@Entity()
export class User extends BaseEntity
{
  @Field()
  @PrimaryGeneratedColumn()
  id!: number;

  @Field(() => String)
  @CreateDateColumn()
  createdAt: Date;

  @Field(() => String)
  @UpdateDateColumn()
  updatedAt: Date;

  @Field()
  @Column({ unique: true })
  username!: string;

  @Field()
  @Column({ unique: true })
  email!: string;
```

```
unique: true })
  email!: string;

  @Property({ type: "text" })
  password!: string;
}
```

```
  @Column()
  password!: string;
}
```

Update Context

- since we do not need to pass orm.em to the context anymore, we delete it from MyContext

types.ts

```
export type MyContext = {
  // Not needed anymore, we delete this --->  em:
EntityManager<IDatabaseDriver<Connection>>;
  req: ExtendedRequest;
  res: Response;
  redis: Redis; // to be added during (11)
};
```

Update Post Resolver

- Since we do not use em.orm anymore, we update the Resolvers accordingly

/resolvers/post.ts

```
import { Post } from "../entities/Post";
import { MyContext } from "src/types";
import { Arg, Ctx, Int, Mutation, Query, Resolver } from "type-graphql";

@Resolver()
export class PostResolver {
  @Query(() => [Post]) // [Post] is how we define arrays in return type for the
resolver
  async posts(): Promise<Post[]> {
    return Post.find()
  }

  @Query(() => Post, { nullable: true })
  post(@Arg("id") id: number): Promise<Post | undefined> {
    return Post.findOne(id)y;
```

```
  }

  @Mutation(() => Post)
  async createPost(@Arg("title") title: string): Promise<Post> {
    return Post.create({title}).save();
  }

  @Mutation(() => Post, { nullable: true })
  async updatePost(
    @Arg("id") id: number, // here we ommitted type declaration in @Arg - type
inference works for Int and String
    @Arg("title", () => String, { nullable: true }) title: string // here we
explicitly set type since we want to make it nullable
  ): Promise<Post | null> {
    const post = await Post.findOne(id);

    if (!post) {
      return null;
    }

    if (typeof title !== "undefined") {
      post.title = title;
      await Post.update({id}, {title});
    }

    return post; // this is actually wrong and returns the unmodified post. we'll
fix it later
  }

  @Mutation(() => Boolean)
  async deletePost(@Arg("id") id: number): Promise<boolean> {
    const post = await em.findOne(Post, { id });

    if (!post) {
      return false;
    }

    await Post.delete(id);
    return true;
```

```
  }
}
```

Update User Resolver

- **Note that** we can use User.findOne(id) since id is the primary key
- When searching with a key that is ***not*** the primary key we use { where : key : value }

  e.g. `User.findOne({ where: email })` or `User.findOne({ where: { email : userNameOrEmail })`

/resolvers/user.ts

```
import { User } from "../entities/User";
import { MyContext } from "src/types";
import {Arg, Ctx, Field, Mutation, Query, Resolver} from "type-graphql";
import argon2 from "argon2";
import { UsernamePasswordInput } from "./UsernamePasswordInput";
import { validateRegister } from "../utils/validateRegister";
import v4 from "uuid"
import { getConnection } from "typeorm";

@ObjectType() // ObjectTypes are returned from Queries and Mutations
class FieldError {
  @Field()
  field: string; // which field the error is about
  @Field()
  message: string; // error message
}

@ObjectType()
class UserResponse {
  @Field(() => [FieldError], { nullable: true })
  errors?: FieldError[];

  @Field(() => User, { nullable: true })
  user?: User;
}

@Resolver()
```

```
export class UserResolver {
  @Mutation(() => UserResponse)
  async changePassword(
    @Arg("token") token: string,
    @Arg("newPassword") newPassword: string,
    @Ctx() { redis, req }: MyContext
  ): Promise<UserResponse> {
    if (newPassword.length <= 2) {
      return {
        errors: [
          {
            field: "newPassword", // must match the name of the field on front-
end
            message: "Length must be greater than 3",
          },
        ],
      };
    }

    const tokenKey = FORGOT_PASSWORD_PREFIX + token;
    const userId = await redis.get(tokenKey); // retrieve value for token from
redis
    if (!userId) {
      return {
        errors: [
          {
            field: "token",
            message: "Token expired",
          },
        ],
      };
    }

    const userIdNum = parseInt(userId);
    const user = await User.findOne(parseInt(userIdNum));

    if (!user) {
      return {
        errors: [
          {
```

```
            field: "token",
            message: "User no longer exists",
          },
        ],
      };
    }

    await User.update(
      { id: userIdNum },
      { password: await argon2.hash(newPassword) }
    ); // change pw in db

    await redis.del(tokenKey); // delete token so it can't be reused
    req.session.userId = user.id; // log the user in
    return { user };
  }

  @Mutation(() => Boolean)
  async forgotPassword(
    @Arg("email") email: string,
    @Ctx() { redis }: MyContext
  ): Promise<Boolean> {
    const user = await User.findOne({ where: email }); // email not primary key,
so we have to use "where"
    if (!user) {
      // the email is not in the db
      return true; //  don't let the person know that the email is not in the db
    }

    const token = v4(); // token for resetting pw

    // save token to redis with value userId, expires in 1 day
    await redis.set(
      FORGOT_PASSWORD_PREFIX + token, // redis key
      user.id, // value
      "ex", // expiry mode
      1000 * 60 * 60 * 24 // expiration duration - 24 hours
    );
    const resetLink = `<a href="http://localhost:3000/change-
password/${token}">Reset password</a>`;
```

```
  sendEmail(email, "Reset Password", resetLink);
  return true;
}

@Query(() => User, { nullable: true })
me(@Ctx() { req }: MyContext) {
  // you are not logged in
  if (!req.session.userId) {
    return null;
  }

  return User.findOne(req.session.userId);
}

@Mutation(() => Boolean)
async logout(@Ctx() { req, res }: MyContext): Promise<Boolean> {
  // clear the user's cookie
  res.clearCookie(COOKIE_NAME);

  // clear the redis record
  return new Promise(
    (
      resolve // remove the session from redis
    ) =>
      req.session.destroy((err) => {
        if (err) {
          console.log(err);
          resolve(false);
          return;
        }
        resolve(true);
      })
  );
}

@Mutation(() => UserResponse)
async login(
  @Arg("usernameOrEmail") usernameOrEmail: string,
  @Arg("password") password: string,
```

```
    @Ctx() { req }: MyContext
  ): Promise<UserResponse> {
    const user = await User.findOne(
      usernameOrEmail.includes("@")
        ? { where: { email: usernameOrEmail } }
        : { where: { username: usernameOrEmail } }
    );
    if (!user) {
      return {
        errors: [
          {
            field: "usernameOrEmail",
            message: "That username or email does not exist",
          },
        ],
      };
    }

    const isPasswordValid = await argon2.verify(user.password, password);
    if (!isPasswordValid) {
      return {
        errors: [
          {
            field: "password",
            message: "Incorrect password",
          },
        ],
      };
    }

    req.session.userId = user.id; // created new type for req in types.ts to make
this work, so the session can store the userId

    return { user };
  }

  async register(
    @Arg("options") options: UsernamePasswordInput, //  let typescript infer type
UsernamePasswordInput
    @Ctx() { req }: MyContext
```

```
  ): Promise<UserResponse> {
    const errors = validateRegister(options);
    if (errors) {
      return { errors };
    }

    const hashedPassword = await argon2.hash(options.password);

    let user;
    try {
      /* Same opeartion Using .create - but may return undefined */
      // user = await User.create({
      //   username: options.username,
      //   password: hashedPassword,
      //   email: options.email,
      // }).save();

      const result = await getConnection()
        .createQueryBuilder()
        .insert()
        .into(User)
        .values({
          username: options.username,
          password: hashedPassword,
          email: options.email,
        })
        .returning("*")
        .execute();
      user = result.raw[0];
    } catch (err) {
      // duplicate username error
      if (err.code === "23505") {
        return {
          errors: [
            {
              field: "username",
              message: "That username is already taken",
            },
          ],
        };
```

```
      }
    }

    req.session.userId = user.id; // logs in the user (by sending cookie to
browser)
    return { user };
  }
}
```

40. Implement Post.creator and User.posts columns

#typeorm #entity #graphql #onetomany #manytoone #backend

Add creator column to Post entity

- We will add a creator column to the Post entity so that we know who created the post
- This column will define a ManyToOne relationship between the Post entity and the User entity
- The foreignKey for the creator will be stored in creatorId column
- Note that after defining this relationship we also have to add the posts column to the User entity with a OneToMany attribute
- Note that we're not exposing the creator column to the client (there's no @Field() attribute).

/entities/Post.ts

```
@Field()
@Column()
creatorId!: number;

@ManyToOne(() => User, (user) => user.posts)
creator: User;
```

Add posts column to User entity

- Note that we're not exposing the posts column to the client (there's no @Field() attribute).

/entities/User.ts

```
@OneToMany(() => Post, (post) => post.creator)
  posts: Post[];
```

41. Implement Post.text and Post.points columns

#typeorm #entity #graphql #backend

Add text column to Post entity

- This column will hold the body text of the post

/entities/Post.ts

```
@Field()
@Column()
text!: string;
```

Add points column to Post entity

- This column will hold the upvotes of the post

/entities/Post.ts

```
@Field()
@Column({ type: int, default: 0 })
points!: number;
```

Delete old posts

- You might need to delete old posts to update the DB, otherwise it can give errors
- To do that
  - Disable synchronisation in createConnection() in index,ts
  - And again, in index.ts, after `await conn.runMigrations();`, we add `await Post.delete({}); //delete all posts` and restart the server

42. Resolver - post.ts / update Mutation - createPost()

#graphql #resolver #mutation #authorization #backend #authentication #typeorm

- The createPost() mutation needs to be updated since we added a text field, a points field and a creatorId to it
- points field will default to zero so we don't need to set it at post creation
- Define an @InputType for passing into createPost() as an input parameter

post.ts

```
@InputType()
class PostInput {
  @Field()
  title: string;
  @Field()
  text: string;
}
```

- We can take the creatorId from our context

post.ts

```
@Mutation(() => Post)
  async createPost(
    @Arg("input") input: PostInput,
    @Ctx() { req }: MyContext
  ): Promise<Post> {
    return Post.create({
      ...input,
      creatorId: req.session.userId,
    }).save();
  }
```

Allow only logged in users to crete posts

- Simplest way to do this would be to check if a user is logged in. We will implement a better way in next section

post.ts

```
@Mutation(() => Post)
  async createPost(
```

```
    @Arg("input") input: PostInput,
    @Ctx() { req }: MyContext
  ): Promise<Post> {
    if (!req.session.userId) {
      // if user not logged in
      throw new Error("not authenticated");
    }

    return Post.create({
      ...input,
      creatorId: req.session.userId,
    }).save();
  }
```

43. Middleware authentication check - isAuth()

#middleware #authentication #typescript #graphql #backend #error-handling

---

- We want only authenticated users to be able to create posts. We will write a middleware function isAuth() to check if the user is logged on or not
- The middleware function runs before the resolver. It has access to args, context, info and root
- We can pass MyContext to it so that it knows the type of the context object
- **Note that** the error that is thrown here does **NOT** end up in response.data.createPost.errors like the errors returned from within the mutation. It ends up in response.error
- See 47. Page - create-post for receiving the errors (`const { error } = await createPost({ input: values })`)

/middleware/isAuth.ts

```
import { MyContext } from "src/types";
import { MiddlewareFn } from "type-graphql";

// MiddlewareFn runs before the resolver
export const isAuth: MiddlewareFn<MyContext> = ({ context }, next) => {
  if (!context.req.session.userId) {
    // if user is not logged in
    throw new Error("not authenticated");
  }

  // if user is logged in continue with resolver
  return next();
};
```

- Then we wrap the createPost() mutation with this middleware as follows:

/resolvers/post.ts

```
@UseMiddleware(isAuth)
@Mutation(() => Post)
async createPost(
  @Arg("input") input: PostInput,
  @Ctx() { req }: MyContext
): Promise<Post> {
   return Post.create({
```

```
    ...input,
    creatorId: req.session.userId,
  }).save();
}
```

- We want to handle this case on the front-end so we will first implement a create-post page and then handle this error there

44. GraphQL Mutation - createPost

#urql #graphql #graphql-codegen #mutation #frontend

Add createPost mutation

/graphql/mutations/createPost.graphql

```
mutation CreatePost($input: PostInput!) {
  createPost(input: $input) {
    id
    createdAt
    updatedAt
    title
    text
    points
    creatorId
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the useCreatePostMutation() hook in generated/graphql.tsx that we can use

generated/graphql.tsx

```
export function createPostMutation() {
  return Urql.useMutation<CreatePostdMutation, CreatePostdMutationVariables>
(CreatePostDocument);
};
```

45. Component - TextAreaField

#reactjs #frontend #chakraui #formik #component

- We implement a TextAreaField component to be used in the create-post page

/components/TextAreaField.tsx

```
import {
  FormControl,
  FormLabel,
  FormErrorMessage,
  Textarea,
} from "@chakra-ui/react";
import { useField } from "formik";
import React, { TextareaHTMLAttributes } from "react";

type TextAreaFieldProps = TextareaHTMLAttributes<HTMLTextAreaElement> & {
  label: string;
  name: string;
};

export const TextAreaField: React.FC<TextAreaFieldProps> = ({
  label,
  ...props
}) => {
  const [field, { error }] = useField(props);

  return (
    <FormControl isInvalid={!!error}>
      <FormLabel htmlFor={field.name}>{label}</FormLabel>
      <Textarea {...field} {...props} id={field.name} />
      {error && <FormErrorMessage>{error}</FormErrorMessage>}
    </FormControl>
  );
};
```

46. Component - Layout

#reactjs #frontend #component

- We will implement a simple Layout component to wrap pages where we want the NavBar to be displayed, simplifying the structures of our pages.

Update the Wrapper component

- We add a WrapperVariant type in the Wrapper component and export it to be used in the Layout component:

/components/Wrapper.tsx

```
import { Box } from "@chakra-ui/react";
import React from "react";

export type WrapperVariant = "small" | "regular";

interface WrapperProps {
  variant?: WrapperVariant;
  children: any;
}
```

Implement the Layout component

/components/Layout.tsx

```
import React from "react";
import { Wrapper, WrapperVariant } from "./Wrapper";
import { NavBar } from "./NavBar";

interface LayoutProps {
  variant?: WrapperVariant;
  children: any;
}

export const Layout: React.FC<LayoutProps> = ({ children, variant }) => {
```

```
  return (
    <>
      <NavBar />
      <Wrapper variant={variant}>{children}</Wrapper>
    </>
  );
};
```

Sticky the NavBar

- The NavBar moves up and disappears when we scroll down, so we want to stick it to the top and put it "above" the content on the z-index.
- We update the NavBar component as such:

/components/NavBar.tsx

```
return (
  <Flex zIndex={1} position="sticky" top={0} bg="tan" p={4}>
    <Box ml={"auto"} suppressHydrationWarning>
      {body}
    </Box>
  </Flex>
);
```

47. Page - create-post

#formik #reactjs #nextjs #chakraui #frontend #page

- Here we use the Layout component instead of the Wrapper, so that the NavBar is also displayed
- We *could* handle the "not authenticated" error, here with something like

```
if (error?.message.includes('not authenticated')) {
  router.push("/login");
```

But then we would have to do it like this for *every* global error (i.e. from the middleware) that is returned from GraphQL. There's a bit more complicated but much better way to do it which we implement through an errorExchange at [48. Global Error Handling with urql]

/pages/create-post.tsx

```
import { Box, Button } from "@chakra-ui/react";
import { Form, Formik } from "formik";
import React, { useEffect } from "react";
import { InputField } from "../components/InputField";
import { TextAreaField } from "../components/TextAreaField";
import { useCreatePostMutation } from "../generated/graphql";
import { useRouter } from "next/router";
import { withUrqlClient } from "next-urql";
import { createUrqlClient } from "../utils/createUrqlClient";
import { Layout } from "../components/Layout";

const CreatePost: React.FC<{}> = ({}) => {
  const router = useRouter();
  const [, createPost] = useCreatePostMutation();
  return (
    <Layout variant="small">
      <Formik
        initialValues={{ title: "", text: "" }}
        onSubmit={async (values) => {
          const { error } = await createPost({ input: values });
          if (!error) {
            router.push("/");
```

```
          }
        }}
      >
        {({ isSubmitting }) => (
          <Form>
            <InputField
              name="title"
              label="Title"
              placeholder="Title of the post"
            />
            <Box mt={4}>
              <TextAreaField
                name="text"
                label="Body"
                placeholder="Enter your text"
              />
            </Box>
            <Button type="submit" isLoading={isSubmitting} mt={4} color="teal">
              Create Post
            </Button>
          </Form>
        )}
      </Formik>
    </Layout>
  );
};

export default withUrqlClient(createUrqlClient)(CreatePost);
```

48. Global Error Handling with urql

#reactjs #nextjs #urql #exchange #graphql #frontend #error-handling #authentication

---

- As it was mentioned in 47. Page - create-post , we could handle the "not authenticated" error, within the create-post page with something like

```
if (error?.message.includes('not authenticated')) {
  router.push("/login");
```

But then we would have to do it like this for every global error (e.g. from the middleware) that is returned from GraphQL. There's a bit more complicated but much better way to do it which we implement through an errorExchange

---

Implement errorExchange

/utils/createUrqlClient.ts

```
import { pipe, tap } from "wonka";
import router from "next/router";

const errorExchange: Exchange =
  ({ forward }) =>
  (ops$) => {
    return pipe(
      forward(ops$),
      tap(({ error }) => {
        if (error) {
          console.log(error);
          if (error.message.includes("not authenticated")) {
            router.replace("/login");
          }
        }
      })
    );
  };
```

Add errorExchange to the urqlClient

- We insert the errorExchange between cacheExchange and ssrExchange in createUrqlClient(). So the order is as follows:

```
exchanges: [ dedupExchange, cacheExchange, errorExchange, ssrExchange,
fetchExchange, ]
```

Conclusion

- Now this errorExchange catches the global errors and handles them accordingly:

```
if (error.message.includes("not authenticated")) {
            router.replace("/login");
```

- More info at github.com/FormidableLabs/urql/issues/225 - global error handling

49. Custom Hook - useIsAuth()

#reactjs #hook #graphql #query #frontend #authentication #routing #url

Implement useIsAuth()

- We implement a custom hook to check if the user is logged in or not on the frontend
- This simple hook routes the user to login page is he's not logged in
- **Note that** we wait for fetching to end before we check if there's a user that's logged in or not
- We also use the next query in the URL to tell the login page where it should send the user after succesfully logging in

/hooks/useIsAuth.ts

```
import { useRouter } from "next/router";
import { useMeQuery } from "../generated/graphql";
import { useEffect } from "react";

export const useIsAuth = () => {
  const [{ data, fetching }] = useMeQuery();
  const router = useRouter();
  useEffect(() => {
    // if user is not logged in, re-route to login page
    if (!fetching && !data?.me) {
      router.replace("/login?next=" + router.pathname);
    }
  }, [fetching, data, router]);
};
```

Update create-post page to use useIsAuth() hook

- Simply insert the hook at the top of the page so it gets executed first

/pages/create=page.tsx

```
const CreatePost: React.FC<{}> = ({}) => {
  const router = useRouter();
  useIsAuth(); // reroute user to login if not logged in
  const [, createPost] = useCreatePostMutation();
```

Update login page to direct to “next” query in the URL

- If the URL includes a next query, we direct the user to that page

/pages/login.tsx

```
onSubmit={async (values, { setErrors }) => {
  const response = await login(values);
  if (response.data?.login.errors) {
    setErrors(toErrorMap(response.data.login.errors));
  } else if (response.data?.login.user) {
    if (typeof router.query.next === "string") {
      router.push(router.query.next);
    } else {
      router.push("/");
    }
  }
}}
```

50. Display Posts on Homepage

#formik #reactjs #nextjs #chakraui #frontend #page

- We set up index.tsx (the homepage) to display all of the posts that have been created
- We also add a button so the user can create post

/index.tsx

```
const Index = () => {
  const [{ data }] = usePostsQuery();
  return (
    <Layout>
      <Flex mb={4} align="center">
        <Heading>LiReddit</Heading>
        <Button ml="auto" type="button" color="teal">
          <Link as={NextLink} href="/create-post">
            Create Post
          </Link>
        </Button>
      </Flex>

      <br />
      {!data ? (
        <div>Loading...</div>
      ) : (
        data.posts.map((p) => <div key={p.id}>{p.title}</div>)
      )}
    </Layout>
  );
};

export default withUrqlClient(createUrqlClient, { ssr: true })(Index);
```

51. Pagination for Posts - Resolver post.ts / Query posts()

#pagination #resolver #query #graphql #backend #typeorm

- Right now the posts query retrieves all of the posts from the server
- We have to add pagination to limit the number of posts retrieved at a time

Define a PaginatedPosts type

- This will hold the posts that are fetched as well as a boolean value that tells whether there are more posts to fetch or not

/resolvers/post.ts

```
@ObjectType()
class PaginatedPosts {
  @Field(() => [Post])
  posts: Post[];
  @Field()
  hasMore: boolean;
}
```

Implement the query

- We will use typeorm.io/#/select-query-builder
- There's offset based pagination and cursor based pagination. We will implement cursor based since offset based can cause performance issues as well as refresh issues when new posts are being added frequently,
- The limit determines how many posts should be fetched and put into the list (we set max. as 50)
- The cursor determines a specific location (in our case a date) that the list will start from
- We fetch realLimitPlusOne posts to see if there's more posts than realLimit

/resolvers/post.ts

```
import { getConnection } from "typeorm";

@Query(() => PaginatedPosts)
  async posts(
```

```
    @Arg("limit", () => Int) limit: number,
    // the first fetch will not have a cursor so cursor should be nullable
    @Arg("cursor", () => String, { nullable: true }) cursor: string | null
  ): Promise<PaginatedPosts> {

    const realLimit = Math.min(50, limit);
    const realLimitPlusOne = Math.min(50, limit) + 1;

    const qb = getConnection()
      .getRepository(Post)
      .createQueryBuilder("p")
      .orderBy('"createdAt"', "DESC") // mind the double quotes '" ... "'
      .take(realLimitPlusOne);

    if (cursor) {
      qb.where("createdAt < :cursor", { cursor: new Date(parseInt(cursor)) });
    }

    const posts = await qb.getMany();

    return {
      posts: posts.slice(0, realLimit),
      hasMore: posts.length === realLimitPlusOne,
    }; // see if there's more posts to retrieve
  }
```

52. Pagination for Posts - GraphQL

Query - posts

#pagination #query #urql #graphql #graphql-codegen #frontend

Update GraphQL query

- Now we update the posts query in graphQL based on the changes we made on the backend

/graphql/queries/posts.graphql

```
query Posts($limit: Int!, $cursor: String) {
  posts(limit: $limit, cursor: $cursor) {
    hasMore
    posts {
      id
      createdAt
      updatedAt
      title
      text
    }
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- and now we have the updated usePostsQuery() hook in /generated/graphql.tsx

53. Pagination for Posts - Homepage

#pagination #typescript #reactjs #chakraui #frontend

Update the homepage to display paginated posts

- We will use the stack component of chakraui to style the posts - chakra-ui.com/stack
- So we will put a "load more" button at the bottom of the list to trigger loading more posts
- Note that we're using useState() to store the current values for limit and cursor
- The limit value is hardcoded and does not change in this implementation but we could also easily implement a user-selectable limit
- **Note that** this will pull and display the next batch of posts, but **the new batch will replace the previous batch.** We'll fix it in the urql client

/pages/index.tsx

```
import { withUrqlClient } from "next-urql";
import { Layout } from "../components/Layout";
import { usePostsQuery } from "../generated/graphql";
import { createUrqlClient } from "../utils/createUrqlClient";
import NextLink from "next/link";
import {
  Box,
  Button,
  Flex,
  Heading,
  Link,
  Stack,
  Text,
} from "@chakra-ui/react";
import { useState } from "react";

const Index = () => {
  const [postsQueryVariables, setPostsQueryVariables] = useState({
    limit: 10,
    cursor: null as string | null,
  });

  const [{ data, fetching }] = usePostsQuery({
    variables: postsQueryVariables,
```

```
});

if (!data && !fetching) {
  return <div>No posts loaded for some reason...</div>;
}

return (
  <>
    <Layout>
      <Flex mb={4} align="center">
        <Heading>LiReddit</Heading>
        <Button ml="auto" type="button" color="teal">
          <Link as={NextLink} href="/create-post">
            Create Post
          </Link>
        </Button>
      </Flex>
      <br />
      {!data ? (
        <div>Loading...</div>
      ) : (
        <Stack spacing={8}>
          {data!.posts.posts.map((p) => (
            <Box key={p.id} p={5} shadow="md" borderWidth="1px">
              <Flex>
                <Heading fontSize="xl">{p.title}</Heading>
                <Flex ml="auto">
                  <Text>posted by:</Text>
                  <Text ml={2} fontWeight="bold">
                    {p.creator.username}
                  </Text>
                </Flex>
              </Flex>
              <Text>{p.text}...</Text>
            </Box>
          ))}
        </Stack>
      )}
      {data && data.posts.hasMore ? (
        <Flex>
```

```
            <Button
              onClick={() =>
                setPostsQueryVariables({
                  limit: postsQueryVariables.limit,
                  cursor:
                    data.posts.posts[data.posts.posts.length - 1].createdAt,
                })
              }
              isLoading={fetching}
              m="auto"
              my={8}
            >
              Load more...
            </Button>
          </Flex>
        ) : null}
      </Layout>
    </>
  );
};

export default withUrqlClient(createUrqlClient, { ssr: true })(Index);
```

- Note that every time we load the posts, we set the cursor to the createdAt value of the last post in the list, so the next batch starts loading from there:

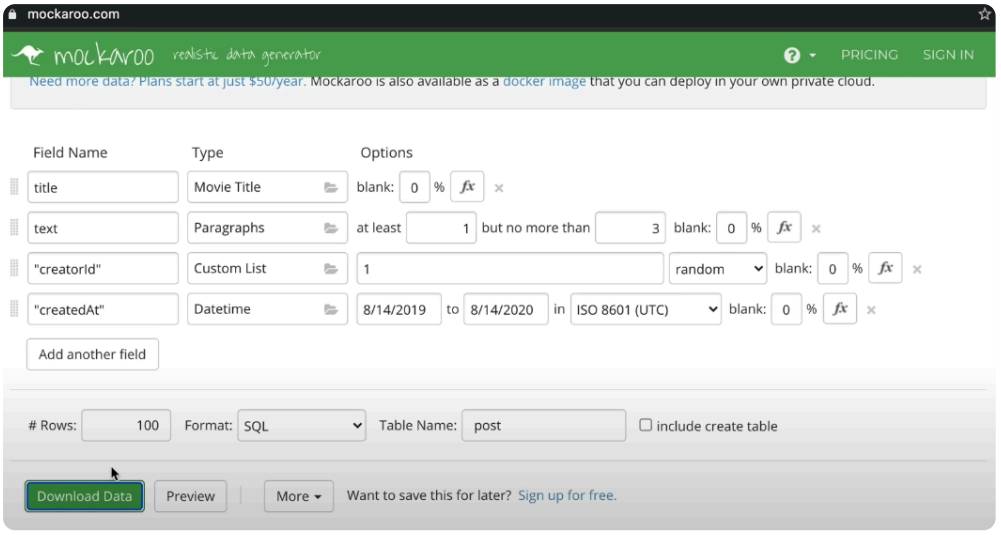
```
setPostsQueryVariables({
  limit: postsQueryVariables.limit,
  cursor:
    data.posts.posts[data.posts.posts.length - 1].createdAt,
})
```

54. Add Mock Data to the Database

#mockdata #database #typeorm #migration #sql

Generate the mock data

- mockaroo.com
- With mockaroo, you can fill out the columns of your database table that you want to insert data into and it will generate fake data for you
- We click "download" to download the data as a .sql file
- **Note that** we wrap createdAt with quotes, becasue it contains an uppercase letter, and if we don't, it will be converted to lowercase
- **Note that** we use a date range for createdAt because we don't want all posts to have same createdAt value



Migrate the mock data into database

- Create a new migration. the -n flag allows us to give it a name

```
npx typeorm migration:create -n FakeData
```

- Create a **/src/migrations** folder and move the new migration file there
- Now copy the contents of the .sql file and paste it into this migration file as such:

/migrations/1718394969560-FakePosts.ts

```
import { MigrationInterface, QueryRunner } from "typeorm";
```

```
export class FakePosts1718394969560 implements MigrationInterface {
  public async up(queryRunner: QueryRunner): Promise<void> {
    // query created with mockaroo.com
    await queryRunner.query(`PASTE THE SQL FILE CONTENTS HERE`); // mind the
back-tick instead of single quotes
  }

  public async down(_: /*queryRunner*/ QueryRunner): Promise<void> {}
}
```

- And when we restart the server, the `await conn.runMigrations();` line in index.tsx will automatically run this migration and load this mockdata in to the database

55. Resolver - post.ts / FieldResolver - textSnippet()

#graphql #resolver #fieldresolver #query #backend #frontend #graphql-codegen

Implement textSnippet FieldResolver

- Assuming the text field of a post is very large, we wouldn't want to download and display all of the texts on the homepage
- We could display only a snippet of it on the frontend like this: `<Text>{p.text.slice(0,50}... </Text>` but still, all of the data would be downloaded to the client needlessly
- Instead we define a FieldResolver that acts on Post objects and can be accessed with `post.textSnippet` from the frontend
- **Note that** we changed `@Resolver()` to `@Resolver(Post)` to let GraphQL know that this Resolver is for Post objects. Only then we can define a FieldResolver that will act on Post objects

/resolvers/post.ts

```
@Resolver(Post)
export class PostResolver {
  @FieldResolver(() => String)
  textSnippet(
    @Root() root: Post // get called for Post objects
  ) {
    return root.text.slice(0, 150);
  }
```

Update GraphQL and generate TypeScript code

- Remove text and add textSnipet to the returned vakues of the posts query
- Even though we do not have a textSnippet column in the database, the FieldResolver will provide this field as a return value, so only the snippets will be sent to to the client from the server

/grapqhql/posts.graphql

```
query Posts($limit: Int!, $cursor: String) {
  posts(limit: $limit, cursor: $cursor) {
    hasMore
    posts {
      id
      createdAt
```

```
      updatedAt
      title
      textSnippet
    }
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- and now we have the updated usePostsQuery() hook in /generated/graphql.tsx

---

Update homepage to use post.textSnippet

- We update the homepage to use this textSnippet

web/src//index.tsx

```
{data!.posts.posts.map((p) => (
  <Box key={p.id} p={5} shadow="md" borderWidth="1px">
    <Flex>
      <Heading fontSize="xl">{p.title}</Heading>
      <Flex ml="auto">
        <Text>posted by:</Text>
        <Text ml={2} fontWeight="bold">
          p.creator.username}
        </Text>
      </Flex>
    </Flex>
    <Text>{p.textSnippet}...</Text>
  </Box>
))}
```

56. Pagination for Posts - Urql Client

#pagination #urql #typescript #frontend #exchange #graphql #graphcache #cache #resolver #error

---

Implement cursorPagination() resolver to pass into cacheExchange

- Right now, frontend is pulling and displaying the next batch of posts, but **the new batch replaces the previous batch.** We'll fix it so new batches are appended to the previous batches
- urql has a simplePagination() function that can be used when doing pagination with limit and offset, as well as a relay pagination version.

formidable.com/open-source/urql/docs/graphcache/computed-queries/#simple-pagination

- However we're using limit and cursor for our pagination, so we'll implement cursorPagination() function by altering the simplePagination()

github.com/urql-graphql/urql/blob/a7d2b21f5c1d456709ac9c520e9132ba6e2e857e/exchanges/graphcache/src/extras/simplePagination.ts

- More info on `cache.resolve()` → formidable.com/open-source/urql/docs/graphcache/computed-queries/

/utils/createUrqlClient.ts

```
import { Resolver, cacheExchange } from "@urql/exchange-graphcache";
import { Exchange, dedupExchange, fetchExchange, stringifyVariables } from
"urql";

const cursorPagination = (): Resolver => {
  return (_parent, fieldArgs, cache, info) => {
    const { parentKey: entityKey, fieldName } = info;
    // entityKey = Query, fieldName = posts, since we plug this Resolver into
cacheExchange like that (see below)

    const allFields = cache.inspectFields(entityKey);
    // Retrieves the fields of the cached queries - cache can contain different
queries so we will filter them
    // allFields: [
    //   {
    //     fieldKey: 'posts({"limit":10})',
    //     fieldName: 'posts',
    //     arguments: { limit: 10 }
```

```
    //   }
    // ]

    // filter allFields to get only the field infos related to the query we want
to work on
    const fieldInfos = allFields.filter((info) => info.fieldName === fieldName);

    const size = fieldInfos.length;
    if (size === 0) {
      return undefined;
    }

    // create a new fieldKey to check if the data is in the cache and return it
from cache, updating cache if needed
    // fieldArgs is the arguments passed into the current query, e.g. { limit:
10, cursor: "159734454958" }
    // fieldKey will have the form 'posts({limit:10,cursor:"159734454958"})' as
seen in allFields
    // so we use fieldName and fieldArgs to construct the most recent fieldKey
and check if it is in the cache
    const fieldKey = `${fieldName}(${stringifyVariables(fieldArgs)})`;

    const isItInTheCache = cache.resolve(
      cache.resolveFieldByKey(entityKey, fieldKey) as string,
      "posts"
    );

    info.partial = !isItInTheCache; // reload if new results are not in the cache

    // cache.readQuery() --> This will call the resolver again and enter an
infinite loop
    // so we use this:
    const results: string[] = [];
    let hasMore = true;
    fieldInfos.forEach((fi) => {
      const key = cache.resolveFieldByKey(entityKey, fi.fieldKey) as string;
      const data = cache.resolve(key, "posts") as string[];
      if (!(cache.resolve(key, "hasMore") as boolean)) {
        hasMore = false;
      }
```

```
      results.push(...data);
    });

    return {
      __typename: "PaginatedPosts", // NOT PUTTING THIS WAS CAUSING AN ERROR
graphql.tsx:374 Invalid resolver value: The field at `Query.posts({"limit":10})`
is a scalar (number, boolean, etc), but the GraphQL query expects a selection set
for this field.
      hasMore,
      posts: results,
    };
  };
};
```

Insert the cursorPagination() function into cacheExchange

- Here we add cursorPagination() as a client-side resolver to the cacheExchange so that it will be executed everytime the posts query is run

/utils/createUrqlClient.ts

```
cacheExchange({
  // to circumvent the error: Invalid key: The GraphQL query at the field at
`Query.posts({"limit":10})` has a
  // selection set, but no key could be generated
  keys: {
    PaginatedPosts: () => null,
  },
  resolvers: {
    Query: {
      // this will run whenever the posts query is run
      // name of it matches what we used in posts.graphql
      posts: cursorPagination(),
    },
  },
  updates: { ... }
```

57. Retrieve the creator of a post

#typeorm #entity #graphql #onetomany #manytoone #backend #frontend #query #resolver #sql #postgresql

---

Expose the creator column to the client

- We already added creator column to the Post entity in 40. Implement Post.creator and User.posts columns
- We were not exposing the creator column to the client (there's no @Field() attribute). We add the @Field() attribute to it
- Since creator of type User and that's an @ObjectType() GraphQL automatically knows which fields creator has

/entities/Post.ts

```
@Field()
@Column()
creatorId!: number;

@Field()
@ManyToOne(() => User, (user) => user.posts)
creator: User;
```

Update posts query in Resolver

- We implemented an SQL query to pull posts from the database in 51. Pagination for Posts - Resolver post.ts / Query posts()
- We update this SQL query with an inner join, to provide the creator field by pulling a user from user table via creatorId
- typeorm.io/#/select-query-builder/joining-relations

/resvolvers/post.ts

```
const qb = getConnection()
  .getRepository(Post)
  .createQueryBuilder("p")
  .innerJoinAndSelect("p.creator", "u", "u.id = p.creatorId")
```

```
  .orderBy("p.createdAt", "DESC") // mind the double quotes '" ... "'
  .take(realLimitPlusOne);

if (cursor) {
  qb.where("p.createdAt < :cursor", { cursor: new Date(parseInt(cursor)) });
}
```

Alternative raw SQL query implementation

- We could also do this with raw sql query as follows
- The drawback of this method is that the sql query will always return the creator, even if the end user doesn't need it and the graphql query does not ask for it
- Note that `json_build_object()` is a PostreSQL featurer

/resvolvers/post.ts

```
const replacements: any[] = [realLimitPlusOne];

if (cursor) {
  replacements.push(new Date(parseInt(cursor)));
}

const posts = await getConnection().query(
  `
  select p.*,
  json_build_object(
  'id', u.id,
  'username', u.username,
  'email', u.email
  ) creator
    from post p
    inner join public.user u on u.id = p."creatorId"
    ${cursor ? `where p."createdAt" < $2` : ""}
    order by p."createdAt" DESC
    limit $1
    `,
  replacements
);
```

Update GraphQL query

- Now we update the posts query in graphQL based on the changes we made on the backend

/graphql/queries/posts.graphql

```
query Posts($limit: Int!, $cursor: String) {
  posts(limit: $limit, cursor: $cursor) {
    hasMore
    posts {
      id
      createdAt
      updatedAt
      title
      textSnippet
      creator {
        id
        username
        email
      }
    }
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- and now we have the updated usePostsQuery() hook in /generated/graphql.tsx

58. Resolver - user.ts / FieldResolver - email()

#graphql #resolver #fieldresolver #backend

Implement email FieldResolver

- We don't want users to be able to see other users' emails
- We define a FieldResolver that acts on User objects and filters the fields based on the logged in user
- **Note that** we changed `@Resolver()` to `@Resolver(User)` to let GraphQL know that this Resolver is for User objects. Only then we can define a FieldResolver that will act on User objects
- **Note that** textSnippet created a new field that could be retrieved, however here, since the name of the FieldResolver matches an actual field (email) it replaces the value of the email field

/resolvers/user.ts

```
@Resolver(User)
export class UserResolver {
  @FieldResolver(() => String)
  email(@Root() user: User, @Ctx() { req }: MyContext) {
    // this is the current user and can see own email
    if (req.session.userId === user.id) {
      return user.email;
    }
    // users can't see email of other users
    return "";
  }
```

59. Entity - Updoot.ts

#typeorm #graphql #entity #manytomany #backend

Implement Updoot entity

- Now we implement the Updoot entity which will store the upvotes of a post and define a many-to-many relationship between User and Post entities
- Many users can upvote a post AND a user can upvote many posts, therefore many-to-many
- userID and postId are the primary keys, so each updoot is unique based on these two keys
- **Note that** we're not exposing any of the fields to graphql since client does not need to see this entity and will *only* need the points field in the Post entity

/entities/Updoot.ts

```
import { Field, ObjectType } from "type-graphql";
import { BaseEntity, Column, Entity, ManyToOne, PrimaryColumn } from "typeorm";
import { Post } from "./Post";
import { User } from "./User";

// many to many relationship
// user <-> post
// user -> join table <- post
// user -> updoot <- post

@Entity() // typeorm
export class Updoot extends BaseEntity {
  @Column({ type: "int" })
  value: number;

  @PrimaryColumn()
  userId: number;

  @ManyToOne(() => User, (user) => user.updoots)
  user: User;

  @PrimaryColumn()
  postId: number;

  @ManyToOne(() => Post, (post) => post.updoots)
```

```
  post: Post;
}
```

Update index.ts

- We have to tell typeorm to use this Updoot entity , by adding it to `entities: [Post, User],` in the typeorm config passed into createConnection()

/index.ts

```
const conn = await createConnection({
  type: "postgres",
  database: "lireddit2",
  username: "postgres",
  password: "postgres",
  logging: true,
  synchronize: true,
  migrations: [path.join(__dirname, "./migrations/*")],
  entities: [Post, User, Updoot],
});
```

Update User entity

- Add tthe Updoots to User entity
- Note that we're not exposing the updoots to the client with a @Field() attribute

/entities/User.ts

```
@OneToMany(() => Updoot, (updoot) => updoot.user)
  updoots: Updoot[];
```

Update Post entity

- Add tthe Updoots to Post entity
- Also add a points field that will hold the total upvoates of this post
- Note that we're not exposing the updoots to the client with a @Field() attribute

/entities/Post.ts

```
@Field()
@Column({ type: "int", default: 0 })
points!: number;
```

```
@OneToMany(() => Updoot, (updoot) => updoot.post)
updoots: Updoot[];
```

60. Resolver - post.ts / Mutation - vote()

#graphql #resolver #authentication #mutation #backend #typeorm

Implement the mutation for voting

- It will just be an upvote or downvote no matter how many points user gives

/resolvers/post.ts

```
@Mutation(() => Boolean)
@UseMiddleware(isAuth)
async vote(
  @Arg("value", () => Int) value: number,
  @Arg("postId", () => Int) postId: number,
  @Ctx() { req }: MyContext
) {
  const isUpdoot = value !== -1;
  const realValue = isUpdoot ? 1 : -1;
  const userId = req.session.userId;

  // This also works but we can also do it in the sql query as below
  // await Updoot.insert({
  //   userId,
  //   postId,
  //   value: realValue,
  // });

  await getConnection().query(
    `
    START TRANSACTION;

    insert into updoot("userId", "postId", "value") //quotes are needed to
preserve capital letters
    values (${userId}, ${postId}, ${realValue});

    update post
    set points = points + ${realValue}
    where id = ${postId};
```

```
    COMMIT;
    `
  );

  return true;
}
```

- Now we can vote up or down on posts, however we can’t cancel or change our vote afterwards.

61. Invalidating the cache after createPost() - Urql Client

#urql #cache #exchange #mutation #graphql #graphcache #frontend

Invalidating part of the list

- When we create a new post it is not immediately displayed on homepage. We can invalidate cache so that data is reloaded and new the post appears
- We *could* simply add the new post to the top of the list, but that's more error-prone since there could be race conditions between different clients etc
- We use the cacheExchange in Urql Client config and add the createPost mutation so that everytime it is executed the entire posts query in the cache will be invalidated (as opposed to invalidating a single post as in [76. Invalidating the cache after deletePost() - Urql Client]( )
- We also get the previousLimit (should never be null anyway) from the Query to pass it back to the new posts query that will be executed, so the same number of results are displayed.

/utils/createUrqlClient.ts

```
updates: {
  Mutation: {
    createPost: (result, args, cache, info) => {
      var previousLimit = cache
        .inspectFields("Query")
        .find((f) => f.fieldName === "posts")?.arguments?.limit as number;
      cache.invalidate("Query", "posts", {
        limit: previousLimit,
      });
    },

    logout: (result, args, cache, info) => { ...
```

- Note that adding `cursor: null` to the third parameter of cache.invalidate() results in a new query being created instead of replacing the previous query since the previous query does not have a `cursor` field in its fieldKey (see screenshot below)

⚠️ **so this is wrong:**

```
createPost: (result, args, cache, info) => {
  console.log("start");
  console.log(cache.inspectFields("Query"));
  cache.invalidate("Query", "posts", {
```

```
    limit: 10,
    cursor: null
  });
  console.log(cache.inspectFields("Query"));
  console.log("end");
},
```

and results in this:

```
start                                                          createUrqlClient.ts:105
                                                               createUrqlClient.ts:106
▼ (2) [{…}, {…}]
  ▶ 0: {fieldKey: 'posts({"limit":10})', fieldName: 'posts', arguments: {…}}
  ▶ 1: {fieldKey: 'me', fieldName: 'me', arguments: null}
    length: 2
  ▶ [[Prototype]]: Array(0)
                                                               createUrqlClient.ts:111
▼ (3) [{…}, {…}, {…}]
  ▶ 0: {fieldKey: 'posts({"limit":10})', fieldName: 'posts', arguments: {…}}
  ▶ 1: {fieldKey: 'me', fieldName: 'me', arguments: null}
  ▶ 2: {fieldKey: 'posts({"cursor":null,"limit":10})', fieldName: 'posts', arguments: {…}}
    length: 3
  ▶ [[Prototype]]: Array(0)
end                                                            createUrqlClient.ts:112
```

Invalidating the entire list

- But we could have a big paginated list created by clicking “load more” a couple of times, , how do we know which part of it to invalidate? : Answer is we don’t, so we will actually have to invalidate the entire list, otherwise the other parts will remain in the cache and could cause errors / unexpected behaviour
- We will do it in a similar way we did the cursorPagination as follows
- Similar to the initial implementation, fi.arguments should never be null anyway

/utils/createUrqlClient.ts

```
createPost: (result, args, cache, info) => {
  const allFields = cache.inspectFields("Query");
  // filter allFields to get only the field infos related to the field we want to
work on
  const fieldInfos = allFields.filter(
    (info) => info.fieldName === "posts"
  );
  fieldInfos.forEach((fi) => {
    cache.invalidate("Query", "posts", fi.arguments || {});
  });
},
```

62. Display Creator of a Post

#reactjs #chakraui #frontend

- We are already retrieving the creator field from backend via the graphql usePostsQuery()
- We just update the Stack component on homepage to display the creator

/index.tsx

```
<Stack spacing={8}>
  {data!.posts.posts.map((p) => (
    <Box key={p.id} p={5} shadow="md" borderWidth="1px">
      <Flex>
        <Heading fontSize="xl">{p.title}</Heading>
        <Flex ml="auto">
          <Text>posted by:</Text>
          <Text ml={2} fontWeight="bold">
            {p.creator.username}
          </Text>
        </Flex>
      </Flex>
      <Text>{p.textSnippet}...</Text>
    </Box>
  ))}
</Stack>
```

63. GraphQL Mutation - vote

#urql #graphql #graphql-codegen #mutation #frontend

Add vote mutation

/graphql/mutations/vote.graphql

```
mutation Vote($value: Int!, $postId: Int!) {
  vote(value: $value, postId: $postId)
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the useVoteMutation() hook in generated/graphql.tsx that we can use

64. Display Updoot Buttons and Points of a Post

#reactjs #chakraui #frontend #styling

Update graphql query

- First we update our graphql file so we retrieve the points on a post from the database

/graphql/queries/posts.graphql

```
query Posts($limit: Int!, $cursor: String) {
  posts(limit: $limit, cursor: $cursor) {
    hasMore
    posts {
      id
      createdAt
      updatedAt
      title
      textSnippet
      points
      creator {
        id
        username
      }
    }
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- and now we have the updated usePostsQuery() hook in /generated/graphql.tsx

Implement the buttons and the points

- We will add upvote and downvote buttons as well as the current points a post has
- We will also add some styling

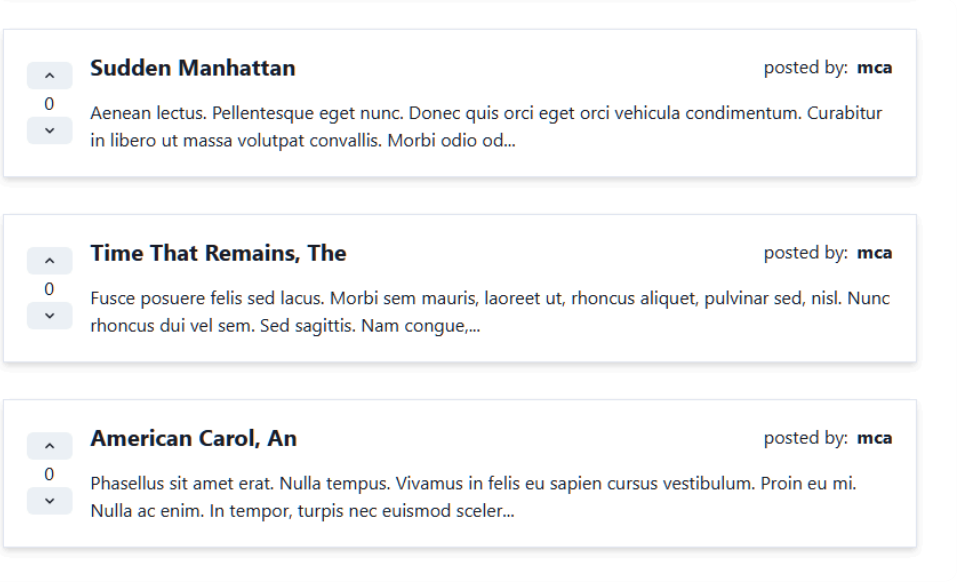
index.tsx

```
<Stack spacing={8}>
  {data!.posts.posts.map((p) => (
    <Flex key={p.id} p={5} shadow="md" borderWidth="1px">
      <Flex
        direction="column"
        justifyContent="center"
        alignItems="center"
        mr={4}
      >
        <IconButton
          boxSize={6}
          icon={<ChevronUpIcon />}
          aria-label={"Updoot post"}
        />
        {p.points}
        <IconButton
          boxSize={6}
          icon={<ChevronDownIcon />}
          aria-label={"Downvdoot post"}
        />
      </Flex>

      <Box width="100%">
        <Flex justifyContent="space-between">
          <Heading marginEnd="auto" fontSize="xl">
            {p.title}
          </Heading>

          <Flex>
            <Text>posted by:</Text>
            <Text ml={2} fontWeight="bold">
              {p.creator.username}
            </Text>
          </Flex>
        </Flex>
        <Text mt={4}>{p.textSnippet}</Text>
      </Box>
    </Flex>
```

```
  ))}
</Stack>
```

Conclusion

- The end result is like this:



65. Component - UpdootSection

#reactjs #frontend #component #graphql #graphql-codegen #fragment

Implement PostSnippet Fragment

- First we define a PostSnippet fragment to have graphql-codegen create us a PostSnippet type that we can use in the Updoot component
- As we saw in 22. GraphQL Mutation - login w/ Fragments , we implement the fragments **based on** ObjectTypes that are already defined in the server code:
  - Post is an ObjectType defined in Post.ts entity (4. Entity - Post.ts)
- We create the following file:

/fragments/PostSnippet.graphql

```
fragment PostSnippet on Post {
  id
  createdAt
  updatedAt
  title
  textSnippet
  points
  creator {
    id
    username
  }
}
```

- We also insert this new fragment into posts.graphql to simplify it

/graphql/queries/posts.graphql

```
query Posts($limit: Int!, $cursor: String) {
  posts(limit: $limit, cursor: $cursor) {
    hasMore
    posts {
      ...PostSnippet
    }
  }
}
```

Obtain the TypeScript Type

- Run codegen to generate the TypeScript types

```
yarn gen
```

- and now we have the updated new PostSnippet type that we can use:

/generated/graphql.tsx

```
export type PostSnippetFragment = (
  { __typename?: 'Post' }
  & Pick<Post, 'id' | 'createdAt' | 'updatedAt' | 'title' | 'textSnippet' |
'points'>
  & { creator: (
    { __typename?: 'User' }
    & Pick<User, 'id' | 'username'>
  ) }
);
```

Implement UpdootSection Component

- Now we extract the Flex component that contains the buttons from the Stack component in index.ts into an Updoot component and implement the voting logic

- **Note that** in the UpdootSectionProps object we could just import the points

```
interface UpdootProps {
  points: number;
}
```

and this would be enough for this particular implementation, however we want to be able to access all fields of a post, so that if in the future this component needs to access more than the points, it will be able to do that.

- For this, we could import a PostsQuery object as follows, and it would contain all fields of a post at the deepest level

```
import { PostsQuery } from "../generated/graphql";

interface UpdootProps {
```

```
  post: PostsQuery["posts"]["posts"][0];
}
```

- Instead we simply use the PostSnippet type that we obtained
- We also use the useVoteMutation() hook we created with graphql-codegen
- **Note that** here, instead of using the fetching state we could obtain with `[{fetching}, vote] = useVotingMutation()` we instead implement the loading indicators on the buttons separately, since the fetching variable would not allow us to know which button to display as "loading". In fact the operation variable shows us what value was passed to the vote() function, but that is updated only after the function is completed so it is also not useful for us

/components/UpdootSection.tsx

```
import { ChevronUpIcon, ChevronDownIcon } from "@chakra-ui/icons";
import { Flex, IconButton } from "@chakra-ui/react";
import React, { useState } from "react";
import { PostSnippetFragment, useVoteMutation } from "../generated/graphql";

interface UpdootSectionProps {
  post: PostSnippetFragment;
}

export const UpdootSection: React.FC<UpdootSectionProps> = ({ post }) => {
  const [loadingState, setLoadingState] = useState<
    "updoot-loading" | "downdoot-loading" | "not-loading"
  >("not-loading");
  const [, vote] = useVoteMutation();
  return (
    <Flex direction="column" justifyContent="center" alignItems="center" mr={4}>
      <IconButton
        onClick={async () => {
          setLoadingState("updoot-loading");
          await vote({
            postId: post.id,
            value: 1,
          });
          setLoadingState("not-loading");
        }}
        isLoading={loadingState === "updoot-loading"}
        boxSize={6}
```

```
        icon={<ChevronUpIcon />}
        aria-label={"Upvote post"}
      />
      {post.points}
      <IconButton
        onClick={async () => {
          setLoadingState("downdoot-loading");
          await vote({
            postId: post.id,
            value: -1,
          });
          setLoadingState("not-loading");
        }}
        isLoading={loadingState === "downdoot-loading"}
        boxSize={6}
        icon={<ChevronDownIcon />}
        aria-label={"Downvote post"}
      />
    </Flex>
  );
};
```

66. Resolver - post.ts / Mutation - vote()

#graphql #resolver #authentication #mutation #backend #typeorm

Update the mutation for voting

- We update the mutation so that user can change his vote on a post
- We change the method of creating SQL queries so that typeorm will create a transaction using the transaction manager (tm)

/resolvers/post.ts

```
@Mutation(() => Boolean)
  @UseMiddleware(isAuth)
  async vote(
    @Arg("value", () => Int) value: number,
    @Arg("postId", () => Int) postId: number,
    @Ctx() { req }: MyContext
  ) {
    const isUpdoot = value !== -1;
    const realValue = isUpdoot ? 1 : -1;
    const userId = req.session.userId;

    // check to see if the user has voted before
    const updoot = await Updoot.findOne({ where: { postId, userId } });

    // user has voted before and is changing the vote
    if (updoot && updoot.value !== realValue) {
      await getConnection().transaction(async (tm) => {
        // update the updoot table
        await tm.query(
          `
            update updoot
            set value = $1
            where "postId" = $2 and "userId" = $3
          `,
          [realValue, postId, userId]
        );

        // update the post
```

```
        await tm.query(
          `
            update post
            set points = points + $1
            where id = $2
          `,
          [2 * realValue, postId] // 2*realValue so that 1 changes to -1 and vice
versa
        );
      });
    } // use has not voted before
    else if (!updoot) {
      await getConnection().transaction(async (tm) => {
        // update the updoot table
        await tm.query(
          `
            insert into updoot("userId", "postId", "value")
            values ($1, $2, $3)
          `,
          [userId, postId, realValue]
        );

        // update the post
        await tm.query(
          `
            update post
            set points = points + $1
            where id = $2
          `,
          [realValue, postId]
        );
      });
    }

    return true;
  }
```

67. Updating the Displayed Points

#urql #graphql #exchange #cache #frontend

- One way is to return the new value from vote() instead of a boolean value, and display it on the frontend
- Here we will use a method that involves reading and updating fragments, in the cacheExchange

Implement the UpdateResolver in cacheExchange in urqlClient

- We have the readFragment() and writeFragment() function from urql
  - formidable.com/open-source/urql/docs/api/graphcache/#readfragment
  - formidable.com/open-source/urql/docs/api/graphcache/#writefragment
- We don't care about pagination or anything here. No matter where the fragment is located, it will be updated in cache. We're searching for them by postId.

utils/createUrqlClient

```
updates: {
  Mutation: {
    vote: (result, args, cache, info) => {
      const { postId, value } = args as VoteMutationVariables;

      const data = cache.readFragment(
        gql`
          fragment _ on Post {
            id
            points
          }
        `,
        { id: postId } as any
      );
      console.log("data: ", data);
      if (data) {
        const newPoints = (data.points as number) + value;

        cache.writeFragment(
          gql`
            fragment __ on Post {
              points
            }
```

```
          `,
          { id: postId, points: newPoints } as any
        );
      }
    },
```

68. Preventing Too Many Votes

#graphql #urql #resolver #query #sql #backend #graphql-codegen #frontend #ssr #nextjs #cookie

---

- We don't want the user to be able to upvote or downvote more than once. Currently, even though the backend does not allow more thatn 1 up or downvote, the frontend updates the points everytime user clicks,. We will prevent that.

Add voteStatus Field to Post Entity

- We add a new @Field in the Post entity that will indicate whether the current user upvoted or downvoted on that post before
- **Note that** we don't define this as a database column with the typeORM's @Column attribute, but instead will use the posts query in the Post resolver to resolve it by pulling it from the updoot table's value column

/entities/Post.ts

```
@Field()
@Column({ type: "int", default: 0 })
points!: number;

//---- Added here:
@Field(() => Int, { nullable: true })
voteStatus!: number | null; // 1 or -1 or null
//-----

@Field()
@Column()
creatorId!: number;
```

Update posts() Query in post.ts Resolver to Return this New Field

- Ben used the SQL query to do this so I'm also switching to using the raw SQL query instead of typeORM's query builder
- For the voteStatus field, if there's an entry in updoot table, we return that value and we return null if not

resolvers/post.ts

```
@Query(() => PaginatedPosts)
async posts(
  @Arg("limit", () => Int) limit: number,
  @Arg("cursor", () => String, { nullable: true }) cursor: string | null
  @Ctx() { req } : MyContext
): Promise<PaginatedPosts> {
  //await sleep(3000); // simulate delay to test csr vs ssr load times

  const realLimit = Math.min(50, limit);
  const realLimitPlusOne = Math.min(50, limit) + 1;

  // Use SQL query to get the data from DB:

  const replacements: any[] = [realLimitPlusOne];

  if (req.session.userId) {
    replacements.push(req.session.userId);
  }

  let cursorIdx = 3
  if (cursor) {
    replacements.push(new Date(parseInt(cursor)));
    cursorIdx = replacements.length
  }

  const posts = await getConnection().query(
    `
  select p.*,
  json_build_object(
    'id', u.id,
    'username', u.username,
    'email', u.email
    ) creator,
  ${
    req.session.userId
      ? '(select value from updoot where "userId" = $2 and "postId" = p.id)
"voteStatus"'
      : 'null as "voteStatus"'
  }
      from post p
```

```
      inner join public.user u on u.id = p."creatorId"
      ${cursor ? `where p."createdAt" < $${cursorIdx}` : ""}
      order by p."createdAt" DESC
      limit $1
      `,
    replacements
  );

  return {
    posts: posts.slice(0, realLimit),
    hasMore: posts.length === realLimitPlusOne,
  }; // see if there's more posts to retrieve
}
```

Update PostSnippet fragment to retrieve this new voteStatus field from GraphQL

/fragments/PostSnippet.graphql

```
fragment PostSnippet on Post {
  id
  createdAt
  updatedAt
  title
  textSnippet
  points
  voteStatus
  creator {
    id
    username
  }
}
```

- Run codegen to regenerate the TypeScript types

```
yarn gen
```

Update UpdootSection Component

- We want to implement two features:
  - Prevent voting twice in the same direction if a vote was cast
  - Change button color if a vote was cast
- We update the UpdootSection component as follows, to implement these feautres using the voteStatus information:

/components/UpdootSection.tsx

```
<Flex direction="column" justifyContent="center" alignItems="center" mr={4}>
  <IconButton
    onClick={async () => {
      if (post.voteStatus === 1) {
       return;
      }
      setLoadingState("updoot-loading");
      await vote({
        postId: post.id,
        value: 1,
      });
      setLoadingState("not-loading");
    }}
    isLoading={loadingState === "updoot-loading"}
    boxSize={6}
    backgroundColor={post.voteStatus === 1 ? "teal.100" : ""}
    icon={<ChevronUpIcon />}
    aria-label={"Upvote post"}
  />
  {post.points}
  <IconButton
    onClick={async () => {
      if (post.voteStatus === -1) {
        return;
      }
      setLoadingState("downdoot-loading");
      await vote({
        postId: post.id,
        value: -1,
      });
      setLoadingState("not-loading");
    }}
    isLoading={loadingState === "downdoot-loading"}
```

```
    boxSize={6}
    backgroundColor={post.voteStatus === -1 ? "red.100" : ""}
    icon={<ChevronDownIcon />}
    aria-label={"Downvote post"}
  />
</Flex>
```

Update the urqlClient to utilize voteStatus

- So that we don't update the frontend points value correctly, and prevent updating it when voting is not allowed
- There's a `#bug` here that causes wrong points to be displayed if post already has votes from other people and then the user changes his vote Perhaps I'll come back to it later. It looks like it's related to new user login after a logout does not refresh the homepage (cast votes and button colors stay for old user)

utils/createUrqlClient.ts

```
vote: (result, args, cache, info) => {
  const { postId, value } = args as VoteMutationVariables;

  const data = cache.readFragment(
    gql`
      fragment _ on Post {
        id
        points
        voteStatus
      }
    `,
    { id: postId } as any
  );

  console.log("data: ", data);

  if (data) {
    if (data.voteStatus === value) {
      return;
    }

    const newPoints =
```

```
      (data.points as number) + (!data.voteStatus ? 1 : 2) * value;

    cache.writeFragment(
      gql`
        fragment __ on Post {
          points
          voteStatus
        }
      `,
      { id: postId, points: newPoints, voteStatus: value } as any
    );
  }
},
```

Why is it not working?

⚠️ **This is still not working correctly**. When we refresh the page the voteStatus value is not retrieved from the updoot table so the frontend logic does not work, votes can be cast multiple times, and the button colors do not work.

When we do a client-side request it automatically sends the cookie to the server. In our posts query in the post resolver we have this code:

```
${
  req.session.userId
    ? '(select value from updoot where "userId" = $2 and "postId" = p.id)
"voteStatus"'
    : 'null as "voteStatus"'
}
```

which needs the userId to pull the voteStatus from updoot table.

The reason is that when we have a server side rendering, the request is first sent to NextJS server which in turn sends it to graphql api:

SSR: browser → next.js → graphql ip

When we have client side rendering the request is directly sent to the graphql api

CSR: browser→ graphql api

The browser sends the cookie with the request. So when it is SSR, the cookie is sent to next.js server which doesn't know what to do with it so the cookie is lost and then when the request is passed to graphql there's no cookie, and no userId information that can be read

When we create a new post, however, the request (and the cookie) is directly sent to graphql and then the posts query is executed again. This time the cookie is available in graphql api so the userId can be read and the posts query works as intended

69. SSR Cookie forwarding

#ssr #cookie #nextjs #urql #graphql #fronent #backend #context

- See Why is it not working? for how we arrived here

Find the cookie in the next.js server

- In the createUrqlClient function, we can pass a ctx context object. So if we update the code as such:

/utils/createUrqlClient

```
// this code runs both on the browser and the server
export const createUrqlClient = (ssrExchange: any, ctx: any) => {
  if (isServer()) { // we don't have the ctx object on the browser
     console.log(ctx)
   }
   return {
      // .... all the code remains the same .....
   }
}
```

We see that there's A LOT of stuff in the ctx object:

```
PROBLEMS 11   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   GITLENS                                   3: node (web)

⚠ Fast Refresh had to perform a full reload when ./src/utils/createUrqlClient.ts changed. Read more: https://nextjs.org/docs/messages/fast-refresh-reload
✓ Compiled in 319ms (1708 modules)
ctx:  {
  err: undefined,
  req: IncomingMessage {
    _events: {
      close: undefined,
      error: [Array],
      data: undefined,
      end: [Array],
      readable: undefined
    },
    _readableState: ReadableState {
      highWaterMark: 16384,
      buffer: [],
      bufferIndex: 0,
      length: 0,
      pipes: [],
      awaitDrainWriters: null,
      [Symbol(kState)]: 1315596
    },
    _maxListeners: undefined,
    socket: Socket {
      connecting: false,
      _hadError: false,
      _parent: null,
      _host: null,
      _closeAfterHandlingError: false,
      _events: [Object],
      _readableState: [ReadableState],
      _writableState: [WritableState],
      allowHalfOpen: true,
      _maxListeners: undefined,
      _eventsCount: 8,
      _sockname: null,
      _pendingData: null,
      _pendingEncoding: '',
      server: [Server],
      _server: [Server],
      parser: [HTTPParser],
      on: [Function: socketListenerWrap],
      addListener: [Function: socketListenerWrap],
      prependListener: [Function: socketListenerWrap],
      setEncoding: [Function: socketSetEncoding],
      _paused: false,
      _httpMessage: [ServerResponse],
      timeout: 0,
```

- And if we scroll through this we see that there's a req object and a res object in the ctx
- The cookie can be seen in req.headers (as well as req.rawHeaders and req,cookies.cookie)
- **Note that** in my implementation req.headers was not displayed when I `console.log` 'ged ctx. But when I `console.log` 'ged ctx.req.headers it displayed:

```
PROBLEMS 11   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   GITLENS                                   3: node (web)
 GET / 200 in 285ms                                                                ctx.req.headers:   1 of 1
 ⚠ Fast Refresh had to perform a full reload when ./src/utils/createUrqlClient.ts changed. Read more: https://nextjs.org/docs/m
 ✓ Compiled in 268ms (1581 modules)
ctx.req.headers:  {
  host: 'localhost:3000',
  connection: 'keep-alive',
  'cache-control': 'max-age=0',
  'sec-ch-ua': '"Google Chrome";v="129", "Not=A?Brand";v="8", "Chromium";v="129"',
  'sec-ch-ua-mobile': '?0',
  'sec-ch-ua-platform': '"Windows"',
  'upgrade-insecure-requests': '1',
  dnt: '1',
  'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/129.0.0.0 Safari/537.36',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
  'sec-fetch-site': 'same-origin',
  'sec-fetch-mode': 'navigate',
  'sec-fetch-dest': 'document',
  referer: 'http://localhost:3000/',
  'accept-encoding': 'gzip, deflate, br, zstd',
  'accept-language': 'en-US,en;q=0.9,tr;q=0.8',
  cookie: 'qid=s%3AxKYTRiDKpZDlvndcPxXJR4c67Axzofro.jbJWjTqWfX%2BecvUWG99I8fSVOHrwnfC0wwU%2FADan1DY',
  'x-forwarded-host': 'localhost:3000',
  'x-forwarded-port': '3000',
  'x-forwarded-proto': 'http',
  'x-forwarded-for': '::1'
}
 GET / 200 in 166ms
```

- And more specifically, `console.log("cookie: ", ctx.req.headers.cookie)` gives:

cookie: 'qid=s%3AxKYTRiDKpZDlvndcPxXJR4c67Axzofro.jbJWjTqWfX%2BecvUWG99I8fSVOHrwnfC0wwU%2FADan1DY'

- So all we want to do is to send this cookie from next.js to graphql

---

Find headers field in fetchOptions()

- First ctrl+click `"urql"` import

```
import
import
  Excha
  dedup
  fetch
  strin
} from "urql";
```

```
module "c:/CodeBase/Courses/React_Redis_GraphQL/web/node_modules/urql/dist/types/index"
export * from '@urql/core';
export * from './context';
export * from './components';
export * from './hooks';
You, 4 months ago • implement cursorPaginatin resolver to urql
```

- Then ctrl+click `"@urql/core"` import

```
node_modules > urql > dist > types > index.d.ts
1  export * from '@urql/core';
2  export * from
3  export * from
4  export * from
5
```

```
module "c:/CodeBase/Courses/React_Redis_GraphQL/web/node_modules/@urql/core/dist/types/index"
export * from './client';
export * from './exchanges';
export * from './types';
export { CombinedError, stringifyVariables, createRequest, makeResult, makeErrorResult,
formatDocument, maskTypename, } from './utils';
```

- Then ctrl+click `"./client"` import

```
web > node_modules > @urql > core > dist > types > index.d.ts
1  export * from './client';
2  export * from
3  export * from
4  export { Combi
5
```

```
module "c:/CodeBase/Courses/React_Redis_GraphQL/web/node_modules/@urql/core/dist/types/client"
import { Source, Subscription } from 'wonka';
import { Exchange, GraphQLRequest, Operation, OperationContext, OperationResult, OperationType,
RequestPolicy, PromisifiedSource, DebugEvent } from './types';
```

- we see that in client.d.ts we have fetchOptions in ClientOptions

```
web > node_modules > @urql > core > dist > types > client.d.ts > ClientOptions
1  import { Source, Subscription } from 'wonka';
2  import { Exchange, GraphQLRequest, Operation, OperationContext, OperationRe
3  import { DocumentNode } from 'graphql';
4  /** Options for configuring the URQL [client]{@link Client}. */
5  export interface ClientOptions {
6      /** Target endpoint URL such as `https://my-target:8080/graphql`. */
7      url: string;
8      /** Any additional options to pass to fetch. */
9      fetchOptions?: RequestInit | (() => RequestInit);
10     /** An alternative fetch implementation. */
11     fetch?: typeof fetch;
12     /** An ordered array of Exchanges. */
13     exchanges?: Exchange[];
14     /** Activates support for Suspense. */
```

- ctrl+click on RequestInit to see that the definition in lib.dom.d.ts has the headers field in it

```
9      fetchOptions?: RequestInit | (() => RequestInit);
lib.dom.d.ts C:\Users\muratcan\AppData\Local\Programs\Microsoft VS Code\resources\app\extensions\node_modules\typescript\lib - Definitions (3)
1688     types?: string[];
1689 }
1690
1691 interface RequestInit {
1692     /** A BodyInit object or null to set request's body. */
1693     body?: BodyInit | null;
1694     /** A string indicating how the request will interact with the browser's cache t
1695     cache?: RequestCache;
1696     /** A string indicating whether credentials will be sent with the request always
1697     credentials?: RequestCredentials;
1698     /** A Headers object, an object literal, or an array of two-item arrays to set r
1699     headers?: HeadersInit;
1700     /** A cryptographic hash of the resource to be fetched by request. Sets request'
1701     integrity?: string;
1702     /** A boolean to set request's keepalive. */
1703     keepalive?: boolean;
```

```
globals.d.ts web\node_modules\@types\node
  interface RequestInit extends _RequestInit {}
global.d.ts web\node_modules\next\types
lib.dom.d.ts C:\Users\muratcan\AppData\Local\Programs\Micr...
  interface RequestInit {
```

- And if we ctrl+click HeadersInit type we also see it's definition:

```
28385  type HTMLOrSVGScriptElement = HTMLScriptElement | SVGScriptElement;
28386  type HashAlgorithmIdentifier = AlgorithmIdentifier;
28387  type HeadersInit = [string, string][] | Record<string, string> | Headers;
28388  type IDBValidKey = number | string | Date | BufferSource | IDBValidKey[];
28389  type ImageBitmapSource = CanvasImageSource | Blob | ImageData;
```

- Note that if it didn't have the headers field in it, we could've added it ourselves

---

Send cookie to backend via fetchOptions()

- Now we just send the cookie to the backend in the headers field of fetchOptions()
- Note that we send it as an object

/utils/createUrqlClient.ts

```
// this code runs both on the browser and the server
export const createUrqlClient = (ssrExchange: any, ctx: any) => {
  let cookie = "";
  if (isServer()) {
    // we don't have the ctx object on the browser
    cookie = ctx.req.headers.cookie;
  }

  return {
    url: "http://localhost:4000/graphql",
    fetchOptions: {
      credentials: "include" as const,
      headers: cookie ? { cookie } : undefined,
    },
```

Conclusion

- And now everything is working. The next.js server sends the cookie in the header and the graphql api receives the cookie retrieves the session.userId from it and even when we refresh the page the the userVote value is sent to frontend

---

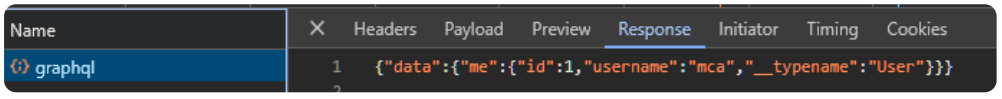
(Optional) Enable server-side meQuery()

- We were preventing the execution of meQuery() on the server through these lines. So a request was made from the client for it,

/components/NavBar.tsx

```
const [{ data, fetching }] = useMeQuery({
  pause: isServer(), // this will prevent the query from running on the server
(there's no cookie on the server to look for)
});
```

- Request after a refresh:



- We can actually enable it now since the server also receives the cookie. This way everything is done on the server side and we are not making any requests to the server from client side when we refresh the page

/components/NavBar.tsx

```
const [{ data, fetching }] = useMeQuery();
```

- No more requests after a refresh:

Name	Method	Status	Type	Initiator

- In the tutorial Ben leaves it client-side with `pause: isServer()`

70. GraphQL Query - post

#urql #graphql #graphql-codegen #mutation #frontend

Add post query

/graphql/queries/post.graphql

```
query post($id: Int!) {
  post(id: $id) {
    id
    createdAt
    updatedAt
    title
    text
    points
    voteStatus
    creator {
      id
      username
    }
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the usePostQuery() hook in generated/graphql.tsx that we can use

---

71. Resolver - post.ts / Query - post()

#graphql #resolver #authentication #query #backend #typeorm

Update post() Query in post Resolver

- The graphql schema in post,graphql expects a creator object, but right now our post() query is not returning that, it's just returning the post:

```
@Query(() => Post, { nullable: true })
  post(
    @Arg("id", () => Int) id: number
  ): Promise<Post | undefined> {
    return Post.findOne(id);
  }
```

- So we update the query to return the creator:
- In TypeOrm we can use `{ relations }` object to have it create the join for us
- We named the many-to-one field in the Post entity "creator", so we use that in the `{ relations }`

/resolvers/post.ts

```
@Query(() => Post, { nullable: true })
  post(
    @Arg("id", () => Int) id: number // 'id' is just a name for using in GraphQL
schema, id is the actual field in database
  ): Promise<Post | undefined> {
    return Post.findOne(id, { relations: ["creator"] });
  }
```

- Now the SQL query looks like this:

```
SELECT "Post"."id" AS "Post_id", "Post"."title" AS "Post_title", "Post"."text" AS
"Post_text", "Post"."points" AS "Post_points", "Post"."creatorId" AS
"Post_creatorId", "Post"."createdAt" AS "Post_createdAt", "Post"."updatedAt" AS
"Post_updatedAt", "Post__creator"."id" AS "Post__creator_id",
"Post__creator"."username" AS "Post__creator_username", "Post__creator"."email"
AS "Post__creator_email", "Post__creator"."password" AS "Post__creator_password",
"Post__creator"."createdAt" AS "Post__creator_createdAt",
```

```
"Post__creator"."updatedAt" AS "Post__creator_updatedAt" FROM "post" "Post" LEFT
JOIN "user" "Post__creator" ON "Post__creator"."id"="Post"."creatorId" WHERE
"Post"."id" IN ($1) -- PARAMETERS: [300]
```

72. Page - post

#reactjs #nextjs #chakraui #frontend #page

Implement post Page

- We need a variable (postId) in the URL of this page, so that the post with that id is shown
  i.e. **localhost:3000/post/123**
- In NextJS the convention for such pages is to create a folder with the page name and place the file inside this folder
- The file name should be `[variableName].tsx` - in our case it will be [id].tsx

/pages/post/[id].tsx

```
import { withUrqlClient } from "next-urql";
import React from "react";
import { createUrqlClient } from "../../utils/createUrqlClient";
import { useRouter } from "next/router";
import { usePostQuery } from "../../generated/graphql";
import { Layout } from "../../components/Layout";
import { Box, Heading } from "@chakra-ui/react";

const Post = ({}) => {
  const router = useRouter();
  const postId =
    typeof router.query.id === "string" ? parseInt(router.query.id) : -1;
  const [{ data, error, fetching }] = usePostQuery({
    pause: postId === -1,
    variables: {
      id: postId,
    },
  });

  if (fetching) {
    return (
      <Layout>
        <Box>Loading...</Box>
      </Layout>
    );
  }
```

```
  if (error) {
    return <div>error</div>;
  }

  if (!data?.post) {
    return (
      <Layout>
        <Box>Could not find post...</Box>
      </Layout>
    );
  }

  return (
    <Layout>
      <Heading mb={4}>{data.post.title}</Heading>
      {data.post.text}
    </Layout>
  );
};

export default withUrqlClient(createUrqlClient, { ssr: true })(Post);
```

Link to post Page from the posts on Home Page

- Update index.tsx to link to individual post pages

/pages/index.tsx

```
<Link as={NextLink} href={`/post/${p.id}`}>
  <Heading marginEnd="auto" fontSize="xl">
    {p.title}
  </Heading>
</Link>
```

Link to HomePage from Nav Bar

- Update NavBarto link to homepage
- Move the “Create Post” button to Navbar
- Add styling

/pages/index.tsx - *Delete the following*

```
<Flex mb={4} align="center">
  <Heading>LiReddit</Heading>
  <Button ml="auto" type="button" color="teal">
    <Link as={NextLink} href="/create-post">
      Create Post
    </Link>
  </Button>
</Flex>
<br />
```

/components/NavBar.tsx

```
} else {
  body = (
    <Flex align="center">
      <Button mr={4} type="button" color="teal">
        <Link as={NextLink} href="/create-post">
          Create Post
        </Link>
      </Button>
      <Box mr={4} color="white">
        {data.me.username}
      </Box>
      <Button
        variant="link"
        isLoading={logoutFetching}
        onClick={() => logout()}
      >
        Logout
      </Button>
    </Flex>
  );
}

return (
  <Flex zIndex={1} position="sticky" top={0} bg="tan" p={4}>
    <Flex flex={1} m="auto" align="center" maxW={800}>
      <Link as={NextLink} href="/">
```

```
        <Heading>LiReddit</Heading>
      </Link>
      <Box ml={"auto"} suppressHydrationWarning>
        {body}
      </Box>
    </Flex>
  </Flex>
);
```

73. Resolver - post.ts / Mutation - delete()

#graphql #resolver #authentication #mutation #backend #typeorm

---

Update the mutation for deleting a post

- We want only users who are logged in to be able to delete posts, and only their own posts
- So we update the delete() mutation

/resolvers/post.ts

```
@UseMiddleware(isAuth)
@Mutation(() => Boolean)
async deletePost(
  @Arg("id", ()=> Int) id: number,
  @Ctx() { req }: MyContext
): Promise<boolean> {
  const post = await Post.findOne(id);

  if (!post) {
    return false;
  }

  await Post.delete({ id, creatorId: req.session.userId });
  return true;
}
```

74. GraphQL Mutation - deletePost

#urql #graphql #graphql-codegen #mutation #frontend

Add deletePost mutation

/graphql/mutations/deletePost.graphql

```
mutation DeletePost($id:Int!) {
  deletePost (id:$id)
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the useDeletePostMutation() hook in generated/graphql.tsx that we can use

75. Display Delete Button on Post Snippets

#reactjs #chakraui #frontend #styling

- Update the post snippet code to display a delete button
- We use the DeletePostMutation() to setup delete action
- **Note that:** as usual, the cache is not reset when we delete, so we will need to fix that. Also we cannot delete posts which we have voted on due to foreign key violation error on updoot table. We will also fix that

/pages/index.tsx

```
const [{ data, fetching }] = usePostsQuery({
  variables: postsQueryVariables,
});

const [, deletePost] = useDeletePostMutation();

if (!data && !fetching) {
  return <div>No posts loaded for some reason...</div>;
}

/* ......... */

<Box width="100%">
  <Flex justifyContent="space-between">
    <Link as={NextLink} href={`/post/${p.id}`}>
      <Heading marginEnd="auto" fontSize="xl">
        {p.title}
      </Heading>
    </Link>
    <Flex>
      <Text>posted by:</Text>
      <Text ml={2} fontWeight="bold">
        {p.creator.username}
      </Text>
    </Flex>
  </Flex>

  <Flex mt={4} flex={1} align="center">
```

```
    <Text>{p.textSnippet}</Text>
    <IconButton
      ml="auto"
      icon={<DeleteIcon />}
      aria-label="Delete Post"
    />
  </Flex>
</Box>
```

76. Invalidating the cache after deletePost() - Urql Client

#urql #cache #exchange #mutation #graphql #graphcache #frontend

Add a new update in createUrqlClient for deletePost()

- We use cache.invalidate() as we did in 61. Invalidating the cache after createPost() - Urql Client
- By default, invalidate() will make the post that we're deleting null. So we modify the code in index.tsx to prevent trying to read null posts

/utils/createUrqlClient.ts

```
updates: {
  Mutation: {
    deletePost: (result, args, cache, info) => {
      const { id } = args as DeletePostMutationVariables
      cache.invalidate({__typename: "Post", id})

    },
```

/pages/index.tsx

```
return (
    <>
      <Layout>
        {!data ? (
          <div>Loading...</div>
        ) : (
          <Stack spacing={8}>
            {data!.posts.posts.map((p) =>
              !p ? null : (
                <Flex key={p.id} p={5} shadow="md" borderWidth="1px">

/* ...... */
```

77. Delete post with foreign key from Updoot table

#graphql #resolver #authentication #mutation #backend #typeorm #error #foreignkey #cascade

---

- Currently when we try to delete a post that has an upvote or downvote we get a foreign key constraint error:
  `update or delete on table \"post\" violates foreign key constraint \"FK_fd6b...bb5\" on table \"updoot\"`
- Here we will see two ways to resolve this issue:

Manually delete the corresponding entry in the updoots table

/resolver/post.ts

```
@UseMiddleware(isAuth)
@Mutation(() => Boolean)
async deletePost(
  @Arg("id", () => Int) id: number,
  @Ctx() { req }: MyContext
): Promise<boolean> {
  const post = await Post.findOne(id);

  if (!post) {
    return false;
  }

  if (post.creatorId !== req.session.userId) {
    throw new Error("Not authorized");
  }

  await Updoot.delete({ postId: id });
  await Post.delete({ id });
  return true;
}
```

Cascade delete the corresponding entry from the updoot table

- We define in the Updoot entity, that when a post is deleted it should cascade to the updoot entry and delete that entry as well

/entities/Updoot.ts

```
@ManyToOne(() => Post, (post) => post.updoots, { onDelete: "CASCADE" })
post: Post;
```

78. Resolver - post.ts / Mutation - updatePost()

#graphql #resolver #authentication #mutation #backend #typeorm

---

Update updatePost() Mutation in post Resolver

- We will update using the TypeORM Query Builder - typeorm.io/#/update-query-builder
- We `console.log` the result object to see what exactly we want to return from it (in this case result.raw[0])

/resolvers/post.ts

```
@UseMiddleware(isAuth)
@Mutation(() => Post, { nullable: true })
async updatePost(
  @Arg("id", () => Int) id: number,
  @Arg("title") title: string,
  @Arg("text") text: string,
  @Ctx() { req }: MyContext
): Promise<Post | null> {
  const result = await getConnection()
    .createQueryBuilder()
    .update(Post)
    .set({ title, text })
    .where('id = :id and "creatorId" = :creatorId', {
      id,
      creatorId: req.session.userId,
    })
    .returning("*")
    .execute();

  return result.raw[0];
}
```

79. GraphQL Mutation - updatePost

#urql #graphql #graphql-codegen #mutation #frontend

Add updatePost mutation

/graphql/mutations/updatePost.graphql

```
mutation UpdatePost($id: Int!, $title: String!, $text: String!) {
  updatePost(id: $id, title: $title, text: $text) {
    id
    title
    text
    textSnippet
  }
}
```

- Run codegen to generate the TypeScript code for graphql

```
yarn gen
```

- Now we have the useUpdatePostMutation() hook in generated/graphql.tsx that we can use

80. Custom Hook - useGetPostFromUrl()

#reactjs #nextjs #hook #fronent

- In the edit page we will use the same method we used in post page to retrieve the id of the post from the url, and pass it into usePostQuery() to retrieve the post data :

/pages/post/[id].tsx

```
const router = useRouter();
const postId =
  typeof router.query.id === "string" ? parseInt(router.query.id) : -1;
const [{ data, error, fetching }] = usePostQuery({
  pause: postId === -1,
  variables: {
    id: postId,
  },
});
```

- So based on the DIY principle, we extract this code into a custom hook:

/utils/useGetPostFromUrl.ts

```
import { useRouter } from "next/router";
import { usePostQuery } from "../generated/graphql";

export const useGetPostFromUrl = () => {
  const router = useRouter();
  const postId =
    typeof router.query.id === "string" ? parseInt(router.query.id) : -1;
  const [{ data, error, fetching }] = usePostQuery({
    pause: postId === -1,
    variables: {
      id: postId,
    },
  });
  return [{ data, error, fetching, postId }];
};
```

- And we use this custom hook both in post page and edit page

- We also use postId in the edit page

/pages/post/[id].tsx

```
const [{ data, fetching, error }] = useGetPostFromUrl();
```

/pages/post/edit/[id].tsx

```
const [{ data, fetching, error, postId }] = useGetPostFromUrl();
```

81. Page - edit

#reactjs #nextjs #chakraui #frontend #page #error #server #backend #urql

Implement edit Page

- We need a variable (postId) in the URL of this page, so that the post with that id is shown to edit
  i.e. **localhost:3000/edit/123**
- In NextJS the convention for such pages is to create a folder with the page name post and place the file inside this folder
- The file name should be `[variableName].tsx` - in our case it will be [id].tsx
- **Note that** this page is almost 100% the same as the post page. We could create a component that *could* be used in both pages to reduce code repetition (but we won't do it here)

/pages/post/edit/[id].tsx

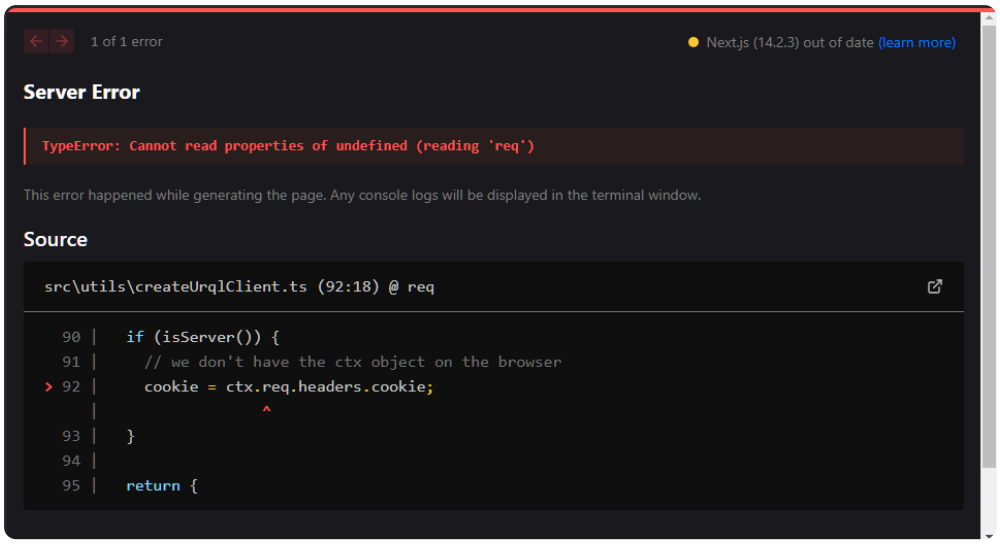
```
import { withUrqlClient } from "next-urql";
import { createUrqlClient } from "../../../utils/createUrqlClient";
import { Box, Button } from "@chakra-ui/react";
import { Formik, Form } from "formik";
import { InputField } from "../../../components/InputField";
import { Layout } from "../../../components/Layout";
import { TextAreaField } from "../../../components/TextAreaField";
import { useGetPostFromUrl } from "../../../utils/useGetPostFromUrl";
import { useUpdatePostMutation } from "../../../generated/graphql";
import { useRouter } from "next/router";

const EditPost = ({}) => {
  const router = useRouter();
  const [{ data, fetching, error, postId }] = useGetPostFromUrl();
  const [, updatePost] = useUpdatePostMutation();

  if (fetching) {
    return (
      <Layout>
        <Box>Loading...</Box>
      </Layout>
    );
  }
```

```
if (error) {
  return <div>error</div>;
}

if (!data?.post) {
  return (
    <Layout>
      <Box>Could not find post...</Box>
    </Layout>
  );
}

return (
  <Layout variant="small">
    <Formik
      initialValues={{ title: data.post.title, text: data.post.text }}
      onSubmit={async (values) => {
        await updatePost({ id: postId, ...values });
        router.back(); // takes back to the page that routed here
      }}
    >
      {({ isSubmitting }) => (
        <Form>
          <InputField
            name="title"
            label="Title"
            placeholder="Title of the post"
          />
          <Box mt={4}>
            <TextAreaField
              name="text"
              label="Body"
              placeholder="Enter your text"
            />
          </Box>
          <Button type="submit" isLoading={isSubmitting} mt={4} color="teal">
            Update Post
          </Button>
        </Form>
      )}
```

```
      </Formik>
    </Layout>
  );
};

export default withUrqlClient(createUrqlClient, { ssr: true })(EditPost);
```

Fix Server Error

- A server error occurs if we refresh the edit page:



- We can fix this simplt by using an optional chain as follows:

/utils/createUrqlClient.ts

```
if (isServer()) {
  // we don't have the ctx object on the browser
  cookie = ctx?.req?.headers?.cookie;
  }
```

82. Display Update Button on Post Snippets

#reactjs #chakraui #frontend #styling

- Update the post snippet code to display an update button
- When clicked we just navigate the user to the post edit page

/pages/index.tsx

```
<Flex mt={4} flex={1} align="center">
  <Text>{p.textSnippet}</Text>
  <Flex ml="auto">
    <NextLink
      href="/post/edit/[id]"
      as={`/post/edit/${p.id}`}
    >
      <IconButton
        mr={1}
        icon={<EditIcon />}
        aria-label="Edit Post"
      />
    </NextLink>
    <IconButton
      onClick={() => deletePost({ id: p.id })}
      icon={<DeleteIcon />}
      aria-label="Delete Post"
    />
  </Flex>
</Flex>
```

83. Display Delete and Update Buttons only to Post Creator

#reactjs #chakraui #frontend #styling #hydration #error

- We should show delete and update buttons only to the creator of the post
- We use useMeQuery() to check that

/pages/index.tsx

```
const [{ data: meData }] = useMeQuery({
  pause: isServer(), // when I run the query on the server I get a hydration
error on browser
});

/* .... */

<Flex mt={4} flex={1} align="center">
  <Text>{p.textSnippet}</Text>
  {meData?.me?.id !== p.creator.id ? null : (
    <Flex ml="auto">
      <NextLink
        href="/post/edit/[id]"
        as={`/post/edit/${p.id}`}
      >
        <IconButton
          mr={1}
          icon={<EditIcon />}
          aria-label="Edit Post"
        />
      </NextLink>
      <IconButton
        onClick={() => deletePost({ id: p.id })}
        icon={<DeleteIcon />}
        aria-label="Delete Post"
      />
    </Flex>
  )}
</Flex>
```

84. Component - EditDeletePostButtons

#reactjs #frontend #component #graphql #graphql-codegen #fragment

- We extract these buttons from index.tsx and put them in a component so we can reuse the component in the post page and the edit page

Implement the component

/components/EditDe;etePostButtons.tsx

```
import { EditIcon, DeleteIcon } from "@chakra-ui/icons";
import { Flex, IconButton } from "@chakra-ui/react";
import NextLink from "next/link";
import React from "react";
import { useDeletePostMutation, useMeQuery } from "../generated/graphql";
import { isServer } from "../utils/isServer";

interface EditDeletePostButtonsProps {
  id: number;
  creatorId: number;
}

export const EditDeletePostButtons: React.FC<EditDeletePostButtonsProps> = ({
  id,
  creatorId,
}) => {
  const [{ data: meData }] = useMeQuery({
    pause: isServer(), // when I run the query on the server I get a hydration
error
  });
  const [, deletePost] = useDeletePostMutation();

  if (meData?.me?.id !== creatorId) {
    return null;
  }

  return (
    <Flex>
      <NextLink href="/post/edit/[id]" as={`/post/edit/${id}`}>
```

```
        <IconButton mr={1} icon={<EditIcon />} aria-label="Edit Post" />
      </NextLink>
      <IconButton
        onClick={() => deletePost({ id })}
        icon={<DeleteIcon />}
        aria-label="Delete Post"
      />
    </Flex>
  );
};
```

Add component to homepage post snippets

/pages/index.tsx

```
<Flex mt={4} flex={1} align="center">
  <Text mr="auto" >{p.textSnippet}</Text>
  <EditDeletePostButtons
    id={p.id}
    creatorId={p.creator.id}
  />
</Flex>
```

Add component to edit post page

/pages/post/edit/[id].tsx

```
return (
  <Layout>
    <Flex flex={1}>
      <Heading mr="auto" mb={4}>
        {data.post.title}
      </Heading>
      <EditDeletePostButtons
        id={data.post.id}
        creatorId={data.post.creator.id}
      />
    </Flex>
    {data.post.text}
  </Layout>
);
```

85. Simpler Data Load - FieldResolver creator()

#typeorm #graphql #resolver #fieldresolver #sql #query #apolloserver #context #dataloader #backend

---

- The posts() query is fetching the creator of the post. This may not be always needed
- It is better to sometimes split up a big query into smaller queries
- Currently we have this:

/resolvers/post.ts

```
const posts = await getConnection().query(
  `
select p.*,
json_build_object(
  'id', u.id,
  'username', u.username,
  'email', u.email
  ) creator,
${
  req.session.userId
    ? '(select value from updoot where "userId" = $2 and "postId" = p.id)
"voteStatus"'
    : 'null as "voteStatus"'
}
    from post p
    inner join public.user u on u.id = p."creatorId"
    ${cursor ? `where p."createdAt" < $${cursorIdx}` : ""}
    order by p."createdAt" DESC
    limit $1
    `,
  replacements
);
```

- We can implement a fieldResolver to simplify this query

FieldResolver() for Creator of a Post

- We already had a @FieldResolver() textSnippet() that gets called for Post objects and returns a String
- This new @FieldResolver() creator() gets called for Post objects and returns a User (entity)

/resolvers/post.ts

```
import { User } from "../entities/User";



@Resolver(Post)
export class PostResolver {
  @FieldResolver(() => String)
  textSnippet(
    @Root() post: Post // get called for Post objects
  ) {
    return post.text.slice(0, 150) + (post.text.length > 150 ? "..." : "");
  }

  @FieldResolver(() => User)
  creator(
    @Root() post: Post // get called for Post objects
  ) {
    return User.findOne(post.creatorId);
  }
```

- And we can remove the bits where we query for the creator in posts() and post() queries:

/resolvers/post.ts

```
const posts = await getConnection().query(
  `
  select p.*,
  ${
    req.session.userId
      ? '(select value from updoot where "userId" = $2 and "postId" = p.id)
"voteStatus"'
      : 'null as "voteStatus"'
  }
  from post p
  ${cursor ? `where p."createdAt" < $${cursorIdx}` : ""}
  order by p."createdAt" DESC
  limit $1
  `,
```

```
  replacements
);
```

/resolvers/post.ts

```
@Query(() => Post, { nullable: true })
post(
  @Arg("id", () => Int) id: number // 'id' is just a name for using in GraphQL schema, id is the actual field in database
): Promise<Post | undefined> {
  return Post.findOne(id);
}
```

- Now everything still works **HOWEVER** when we look at the SQL queries that are being executed we see this:

```
query:
  select p.*,
  (select value from updoot where "userId" = $2 and "postId" = p.id) "voteStatus"
  from post p

  order by p."createdAt" DESC
  limit $1
   -- PARAMETERS: [11,1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [7]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1) -- PARAMETERS: [1]
```

- We are making a **separate SQL query** to the DB **for each Post** to get the creator. This is **NOT** efficient at all
- This is called an (N+1) problem https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping

---

Install dataloader Library

- dataloader will help us patch multiple queries into a single query so we make only one request to the server

```
yarn add dataloader@2.0.0
```

Implement new utility function createUserLoader()

- This function will take an array of userIds and return an array of User objects that match those ids.

/utils/createUserLoader.**ts**

```
import DataLoader from "dataloader";
import { User } from "../entities/User";

// [1, 5, 6, 9] ==> [{user with id 1}, {user with id 5}, {user with id 6}, {user
with id 9}]
export const createUserLoader = () =>
  new DataLoader<number, User>(async (userIds) => {
    const users = await User.findByIds(userIds as number[]);
    // we don't directly return this since it could be out of order, and order
matters here

    const userIdToUser: Record<number, User> = {};
    users.forEach((user) => {
      userIdToUser[user.id] = user;
    });

    return userIds.map((userId) => userIdToUser[userId]);
  });
```

Create a userLoader in the apolloServer context

- Note that context will be run on every request, so a new userLoader will be created on every request
- This userLoader batches and caches the loading of creatores into a single DB query

/index.ts

```
const apolloServer = new ApolloServer({
  schema: await buildSchema({
    resolvers: [HelloResolver, PostResolver, UserResolver],
    validate: false,
  }),
  context: ({ req, res }: MyContext) => ({
    req,
    res,
    redis,
    userLoader: createUserLoader(),
```

```
  }), // context is shared with all resolvers
});
```

- Also update the MyContext type to include userLoader

/types.ts

```
export type MyContext = {
  req: ExtendedRequest;
  res: Response;
  redis: Redis;
  userLoader: ReturnType<typeof createUserLoader>;
};
```

Use userLoader() in the creator() FieldResolver

- Now we load the creatorId into the userLoader() and it will return the correct User for that creatorId after executing a batch query

/resolvers/post.ts

```
@FieldResolver(() => User)
  async creator(
    @Root() post: Post, // get called for Post objects
    @Ctx() { userLoader }: MyContext
  ) {
    return await userLoader.load(post.creatorId);
  }
```

- There are two users in the first 10 posts so the batch query is made for 2 creatorIds:

```
query:
      select p.*,
      (select value from updoot where "userId" = $2 and "postId" = p.id) "voteStatus"
      from post p

      order by p."createdAt" DESC
      limit $1
       -- PARAMETERS: [11,1]
query: SELECT "User"."id" AS "User_id", "User"."username" AS "User_username", "User"."email" AS "User_email", "User"."password" AS "User_password", "User"."cr
eatedAt" AS "User_createdAt", "User"."updatedAt" AS "User_updatedAt" FROM "user" "User" WHERE "User"."id" IN ($1, $2) -- PARAMETERS: [7,1]
```

86. Simpler Data Load - FieldResolver voteStatus()

#typeorm #graphql #resolver #fieldresolver #sql #query #apolloserver #context #dataloader #backend

---

Implement voteStatus FieldResolver

- Let's implement a FieldResolver for voteStatus much like we did with creator

/resolvers/post.ts

```
@FieldResolver(() => Int, { nullable: true })
async voteStatus(
  @Root() post: Post, // get called for Post objects
  @Ctx() { req, updootLoader }: MyContext
) {
  if (!req.session.userId) {
    return null;
  }

  const updoot = await updootLoader.load({
    postId: post.id,
    userId: req.session.userId,
  });

  return updoot ? updoot.value : null;
}
```

Implement new utility function createUpdootLoader()

- This function will take an array of `{ postId, userId }` objects and return an array of Updoot objects that match those ids, or null if not object is found

/utils/createUpdootLoader.**ts**

```
import DataLoader from "dataloader";
import { Updoot } from "../entities/Updoot";

// [{postId: 5, userId: 10}] ==> [voteStatus for that postId and userId]
export const createUpdootLoader = () =>
  new DataLoader<{ postId: number; userId: number }, Updoot | null>(
```

```
    async (keys) => {
      const updoots = await Updoot.findByIds(keys as any);
      // we don't directly return this since it could be out of order, and order
matters here

      const updootIdsToUpdoot: Record<string, Updoot> = {};
      updoots.forEach((updoot) => {
        updootIdsToUpdoot[`${updoot.userId}|${updoot.postId}`] = updoot;
      });

      return keys.map(
        (key) => updootIdsToUpdoot[`${key.userId}|${key.postId}`]
      );
    }
  );
```

Create a updootLoader in the apolloServer context

- Note that context will be run on every request, so a new updootLoader will be created on every request
- This updootLoader batches and caches the loading of voteStatuses into a single DB query

/index.ts

```
const apolloServer = new ApolloServer({
  schema: await buildSchema({
    resolvers: [HelloResolver, PostResolver, UserResolver],
    validate: false,
  }),
  context: ({ req, res }: MyContext) => ({
    req,
    res,
    redis,
    userLoader: createUserLoader(),
    updootLoader: createUpdootLoader(),
  }), // context is shared with all resolvers
});
```

- Also update the MyContext type to include updootLoader

/types.ts

```typescript
export type MyContext = {
  req: ExtendedRequest;
  res: Response;
  redis: Redis;
  userLoader: ReturnType<typeof createUserLoader>;
  updootLoader: ReturnType<typeof createUpdootLoader>;
};
```

87. Refresh cache on login/logout

#nextjs #reactjs #frontend

Refresh page on logout

- To refresh the cache on logout we can simply refresh the page with `router.reload()`

/components/NavBar.tsx

```
import { useRouter } from "next/router";

export const NavBar: React.FC<NavBarProps> = ({}) => {
  const router = useRouter();

/* .... */

<Button
  variant="link"
  isLoading={logoutFetching}
  onClick={async () => {
    await logout();
    router.reload();
  }}
>
```

Invalidate posts in the cache on login

- We already are doing this in createPost(). So we extract that logic as a function and use it in both createPost() and login()

/utils/createUrqlClient.ts

```
function invalidateAllPosts(cache: Cache) {
  var previousLimit = cache
    .inspectFields("Query")
    .find((f) => f.fieldName === "posts")?.arguments?.limit as number;
  cache.invalidate("Query", "posts", {
    limit: previousLimit,
  });
}
```

- And then plug it into both `createPost()` and `login()`

/utils/createUrqlClient.ts

```
createPost: (result, args, cache, info) => {
  invalidateAllPosts(cache);
},

/* .... */

login: (result, args, cache, info) => {
  // cache.updateQuery({ query: MeDocument }, (data: MeQuery) => { })
  betterUpdateQuery<LoginMutation, MeQuery>(
    cache,
    { query: MeDocument },
    result,
    (r, q) => {
      if (r.login.errors) {
        return q; // return the current query if there's error
      } else {
        return {
          me: r.login.user, // return the user info received from successful
login
        };
      }
    }
  );
  invalidateAllPosts(cache);
},
```