# 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:

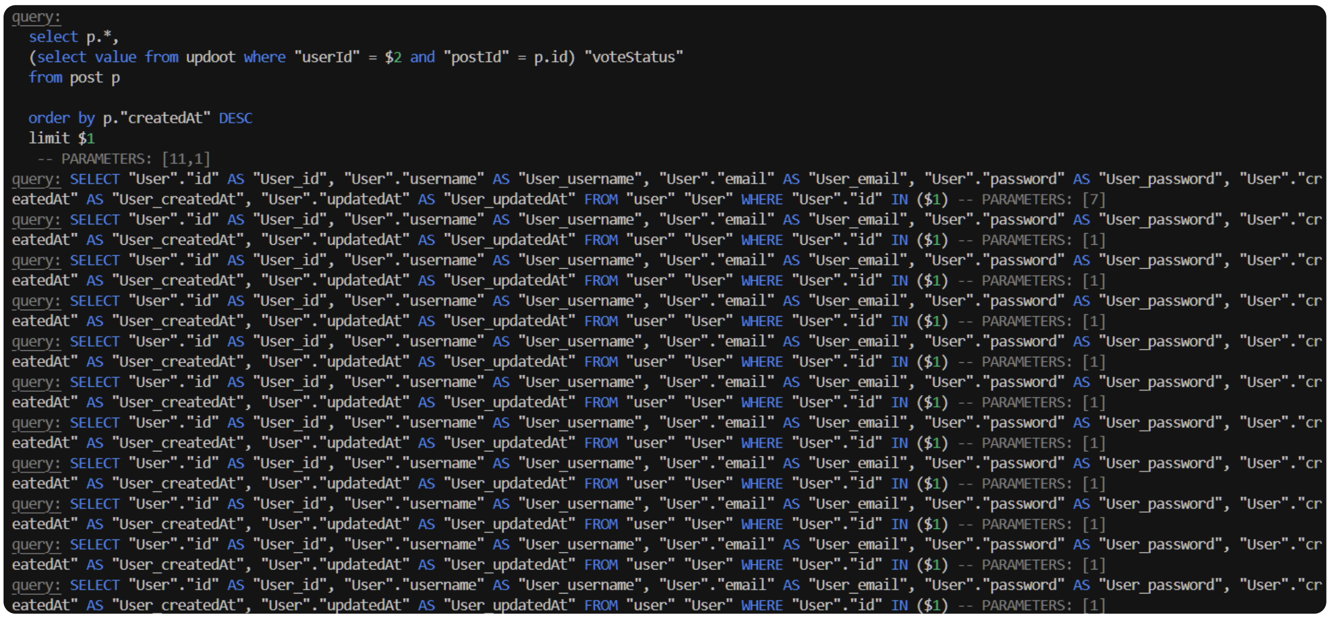
/resolvers/post.ts

```
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>;
};
```

---

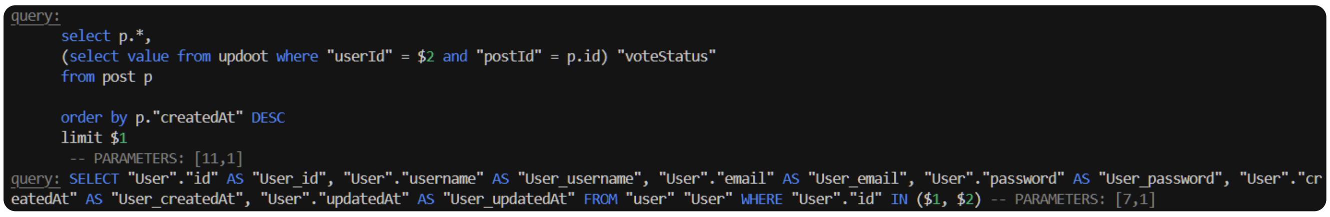
## Use userLoader() in the creator() FieldResolver

- 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]
```