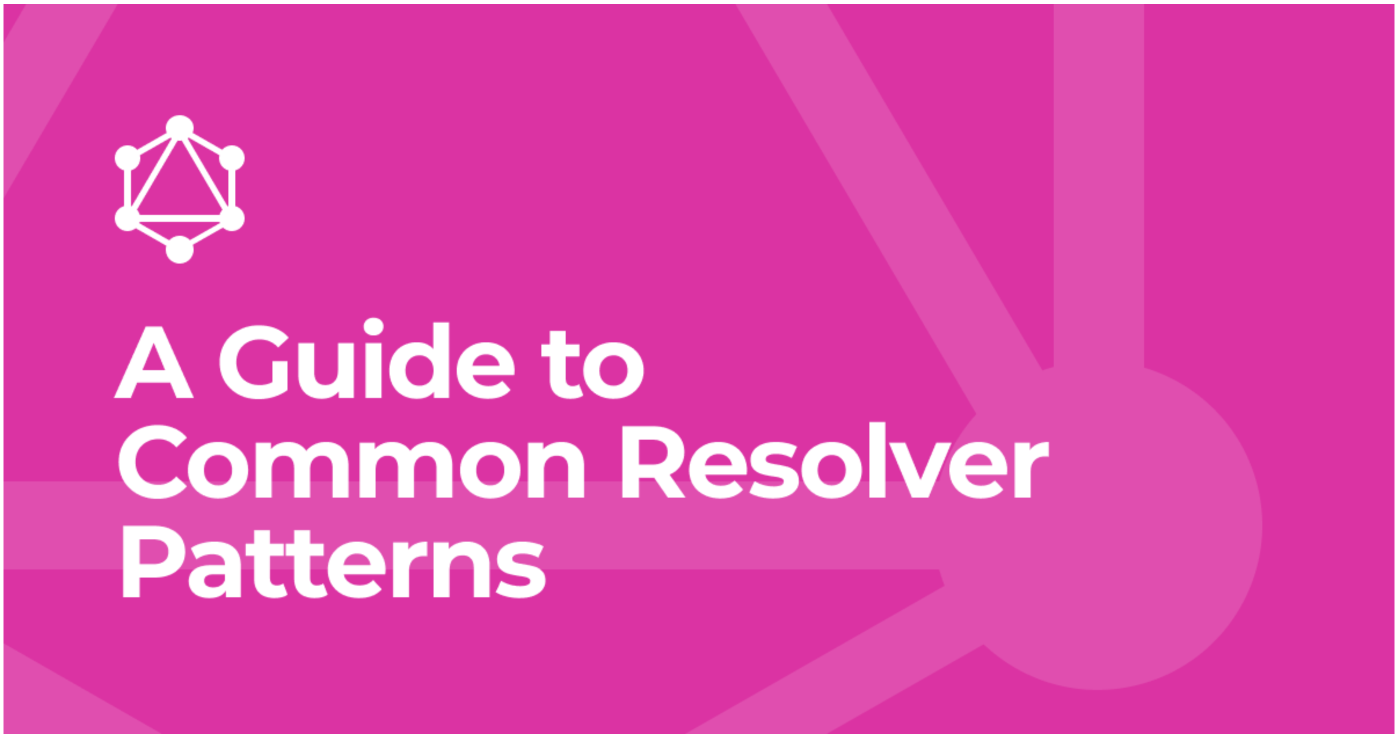
A Guide to Common Resolver Patterns



Introduction

This tutorial gives an overview about common scenarios you might encounter when implementing your GraphQL server with **`graphql-yoga`** and Prisma.

Scenario: Add a custom/computed field to a Prisma model via the application schema (Prisma bindings)

Assume you have the following Prisma datamodel, commonly called **`datamodel.prisma`**:

```
type User {
  id: ID! @id
  firstName: String!
  lastName: String!
}
```

In the GraphQL API of your application layer, you now want to expose a *computed field*, e.g. the **`fullName`** that's composed of the **`firstName`** and **`lastName`**.

In that case, you need to *redefine* the **`User`** type in your application schema, commonly called **`schema.graphql`**:

```
type Query {
  users: [User!]!
}

type User {
  id: ID!
  firstName: String!
  lastName: String!
  fullName: String!
}
```

Now you need to implement the resolvers, here's a naive way to implement it:

```
const resolvers = {
  Query: {
    users: (parent, args, ctx, info) => {
      return ctx.db.query.users({}, `{id firstName lastName}`)
    },
  },
  User: {
    fullName: parent => `${parent.firstName} ${parent.lastName}`,
  },
}
```

This will work but it's very error-prone and especially doesn't scale for more complex use cases. Instead, you should be using *fragment replacements* which will ensure that `firstName` and `lastName` will always be fetched, no matter what the `info` object looks like:

```
const { addFragmentToInfo } = require('graphql-binding')

const resolvers = {
  Query: {
    users: (parent, args, ctx, info) => {
      const fragment = `fragment EnsureFullName on User { firstName lastName }
      return ctx.db.query.users({}, addFragmentToInfo(info, fragment))
    },
  },
  User: {
    fullName: parent => `${parent.firstName} ${parent.lastName}`,
  },
}
```

For more info about what's going on, check out this article on the `info` object.

Scenario: Implementing relations with Prisma client

Assume you want to implement the following application schema using Prisma client:

```
type User {
  id: ID!
  name: String!
  posts: [Post!]!
}
```

```
type Post {
  id: ID!
  title: String!
  published: Boolean!
  author: User!
}

type Query {
  posts: [Post!]!
}

type Mutation {
  createDraft(title: String!, authorId: ID!): Post!
  publish(id: ID!): Post
}
```

Assume your Prisma datamodel is defined as follows:

```
type User {
  id: ID! @id
  name: String!
  posts: [Post!]!
}

type Post {
  id: ID! @id
  title: String!
  published: Boolean! @default(value: false)
  author: User!
}
```

The implementation of the root resolvers **`posts`**, **`createDraft`** and **`publish`** can be done as follows (assuming a Prisma client instance is available as **`prisma`**):

```
const resolvers = {
  Query: {
    posts() {
      return prisma.posts()
    },
  },
  Mutation: {
    createDraft(_, args) {
      return prisma.createPost({
        title: args.title,
        author: {
          connect: {
            id: args.authorId,
          },
        },
      })
    },
    publish(_, args) {
      return prisma.updatePost({
        data: {
          published: true,
        },
        where: {
          id: args.id,
        },
      })
    },
  },
}
```

This implementation seems fairly straightforward, but it has a subtle bug that's related to the relation between **`User`** and **`Post`**. For example, assume you send the following query to the API:

```
query {
  posts {
    title
    author {
      name
    }
  }
}
```

With the current resolver implementation, the `author` relation of a `Post` object can not be resolved! This is because the `posts` resolver returns a list of `Post` objects where each `Post` object only contains *scalar* values - no relations! The `author` fields are not fetched by the client when invoking the `posts` method as is done in the `posts` resolver.

The way to resolve this situation is to introduce "type resolvers" for the `Post` and `User` types with explicit resolvers for the relation fields.

Here is how you need to adjut the `resolvers` object from before to add the type resolvers:

```
const resolvers = {
  Query: {


  },
  Mutation: {


  },
  User: {
    posts(parent) {
      return prisma.user({ id: parent.id }).posts()
    },
  },
  Post: {
    author(parent) {
      return prisma.post({ id: parent.id }).author()
    },
  },
}
```

When the GraphQL server now receives the nested query from before, it will not only invoke the `posts` resolver, but now it can also invoke the `author` resolver to fetch the `author` for each `Post` object. Note that the `parent` argument that's passed into each resolver is the return value of the previous resolver execution level. You can learn more about the query resolution process and the resolver arguments in this article.

Scenario: Add a new field to the data model and expose it in the GraphQL API via Prisma bindings

This scenario is based on the `typescript-basic` GraphQL boilerplate project.

Adding a new address field to the `User` type in the database, with the purpose of exposing it in the application API as well.

Instructions

1. Adding the field to the data model

in `database/datamodel.graphql`:

```
type User {
  id: ID! @id
  email: String! @unique
  password: String!
  name: String!
  posts: [Post!]!
+ address: String
}
```

2. Deploying the updated data model

```
prisma deploy
```

This will...

- ... deploy the new database structure to the local service
- ... download the new GraphQL schema for the database to `database/schema.graphql`

3. Adding the field to the application schema

In `src/schema.graphql`:

```
type User {
  id: ID!
  email: String!
  name: String!
  posts: [Post!]!
+ address: String
}
```

Scenario: Adding a new resolver

Suppose we want to add a custom resolver to delete a `Post`.

Instructions

Add a new `delete` field to the Mutation type in `src/schema.graphql`

```
type Mutation {
  createDraft(title: String!, text: String): Post
  publish(id: ID!): Post
+ delete(id: ID!): Post
}
```

Add a **`delete`** resolver to Mutation part of **`src/index.js`**

```
delete(parent, { id }, ctx, info) {
  return ctx.db.mutation.deletePost(
  {
    where: { id }
  },
    info
  );
}
```

Start the server with **`yarn start`**.

Then you can run the following mutation to delete a post:

```
mutation {
  delete(id: "__POST_ID__") {
    id
  }
}
```