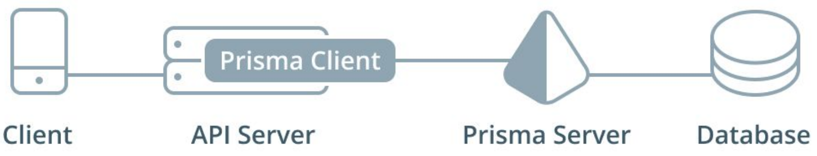
Server

Built using GraphQL, Prisma and Apollo.



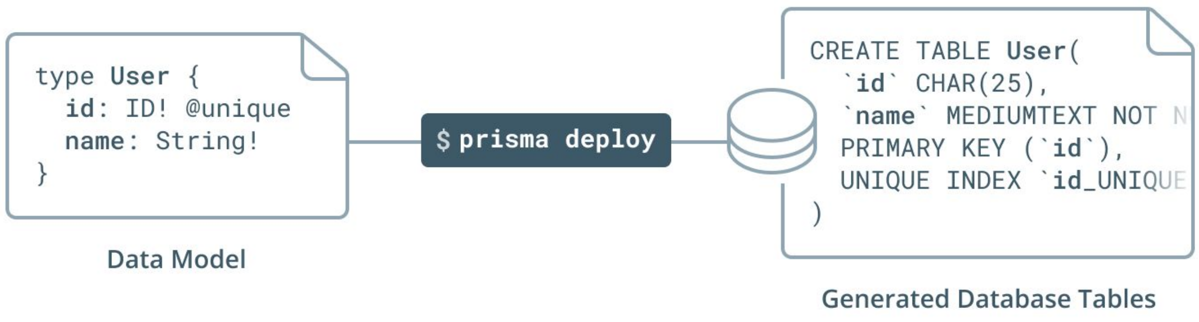
The datamodel is defined in server/database/datamodel.graphql. Every model gets mapped to a table in the database.

The datamodel is written manually by the developers of the Prisma service. It defines the models and structures the developer wants to use in their API.

Strictly speaking, the datamodel is not a valid GraphQL schema because it does not contain any of GraphQL's root types (Query, Mutation, Subscription) and therefore does not define any API operations.

The datamodel only serves as foundation for the generation of the actual GraphQL schema that defines the GraphQL API of your Prisma service.

Going from the Prisma Server to the database.

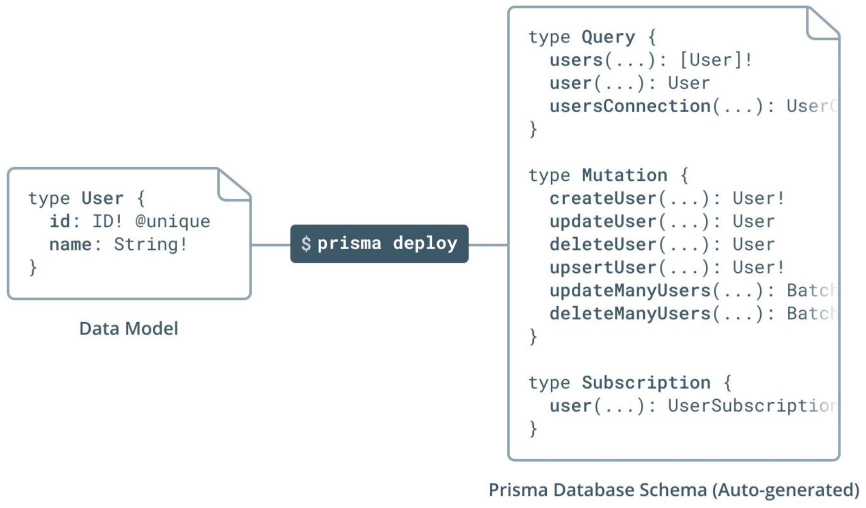


```
type User {
  id: ID! @unique
  name: String!
}
```

```
$ prisma deploy
```

```
CREATE TABLE User(
  `id` CHAR(25),
  `name` MEDIUMTEXT NOT N
  PRIMARY KEY (`id`),
  UNIQUE INDEX `id_UNIQUE
)
```

Going from the Prisma Server to the GraphQL API (API Server).



```
type User {
  id: ID! @unique
  name: String!
}
```

```
type Query {
  users(...): [User]!
  user(...): User
  usersConnection(...): User
}

type Mutation {
  createUser(...): User!
  updateUser(...): User
  deleteUser(...): User
  upsertUser(...): User!
  updateManyUsers(...): Batc
  deleteManyUsers(...): Batc
}

type Subscription {
  user(...): UserSubscriptio
}
```

src: This directory holds the source files for your GraphQL server.

- schema.graphql contains your GraphQL schema. A GraphQL schema defines the operations of a GraphQL API. It effectively is a collection of types written in SDL (SDL also supports primitives like interfaces, enums, union types and more, you can learn everything about GraphQL's type system here). A GraphQL schema has three special root types: **Query, Mutation and Subscription**. These types define the entry points for the API and define what operations the API will accept
- resolvers contains the resolver functions for the operations defined in the GraphQL schema
- index.js is the entry point for your GraphQL server.

<u>Notes on the datamodel</u>

Notice the two ! type modifiers, here is what they express:

- The first ! type modifier (right after String) means that no item in the list can be null, e.g. this value for tags would not be valid: ["Software", null, "GraphQL"]
- The second ! type modifier (after the closing square bracket) means that the list itself can never be null, it might be empty though. Consequently, null is not a valid value for the tags field but [] is.

The relation is named **ParagraphesByArticle** and the deletion behaviour is as follows:

- When an Article node gets deleted, all its related Paragrah nodes will be deleted as well.
- When a Paragrah node gets deleted, it will simply be removed from the paragraphes list on the related Article node.

Notes on the default queries and mutations generated:
https://www.prisma.io/docs/prisma-graphql-api/reference/queries-qwe1/

- There is a bidirectional relation between Paragraph and Articles
- There is a unidirectional relation from Article to Category

Database

Our database is hosted on Prisma' servers:
https://eu1.prisma.sh/baptiste-debever-bf48d1/paragraphes-iteem-1/dev

Our database contains 3 tables, that are translated in GraphQL types.

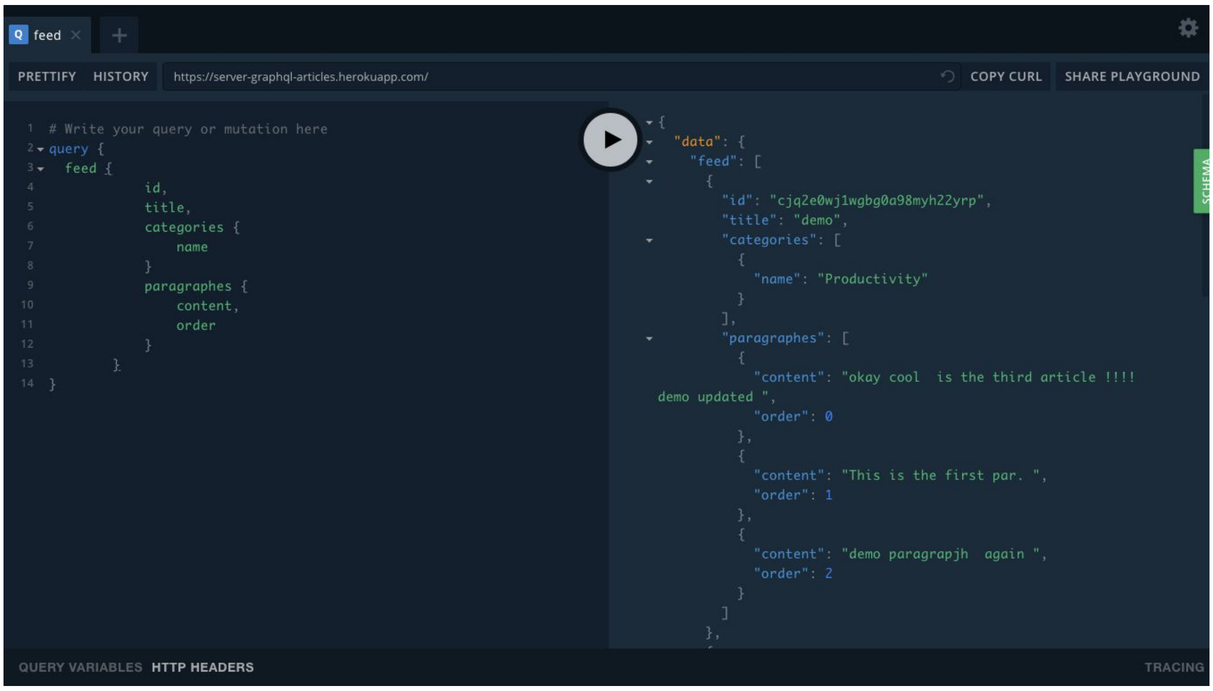
```
type Paragraph {
  id: ID!
  content: String!
  order: Int!
  createdAt: DateTime!
  updatedAt: DateTime!
  article: Article!
}

type Article {
  id: ID!
  title: String!
  categories: [Category!]!
  createdAt: DateTime!
  updatedAt: DateTime!
  paragraphes: [Paragraph!]!
}

type Category {
  id: ID!
  slug: String!
  name: String!
  createdAt: DateTime!
  updatedAt: DateTime!
}
```

This is pretty straightforward. Note that the category table could be used further.

Test the queries

Thanks to GraphQL Playground, it is possible to test the queries/mutations on the server: https://server-graphql-articles.herokuapp.com/.

For example, we could test the following query that is run in-app in order to fetch all articles.

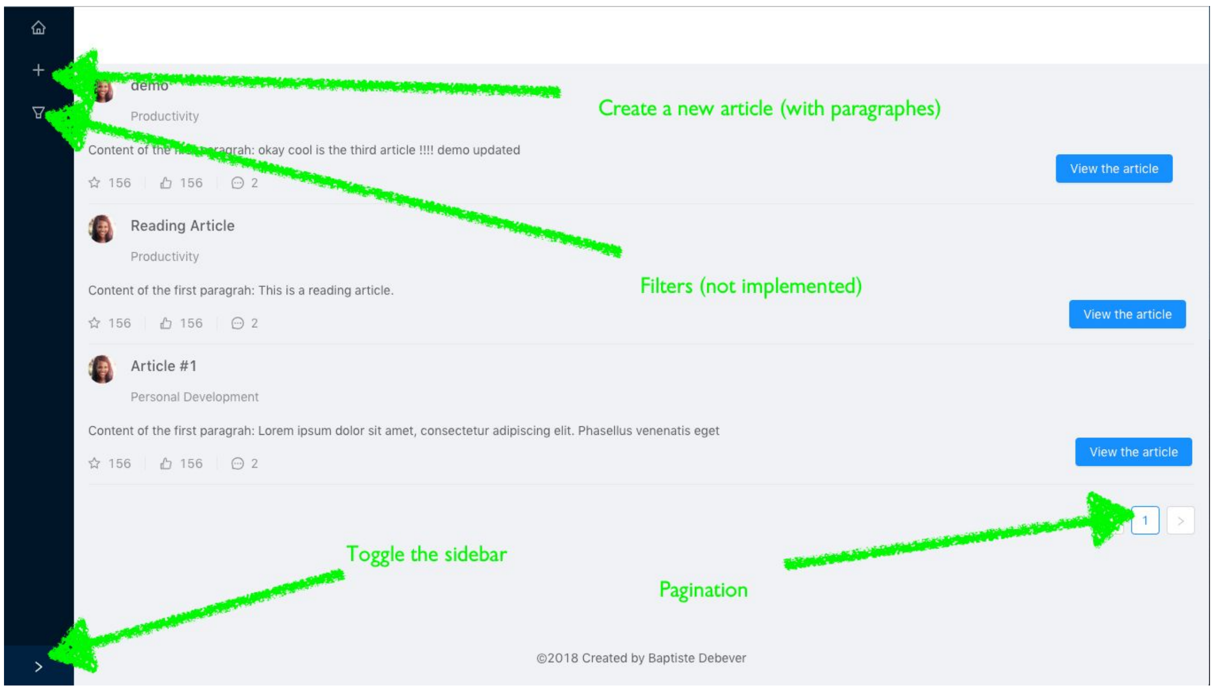


```
# Write your query or mutation here
query {
  feed {
            id,
            title,
            categories {
                name
            }
            paragraphes {
                content,
                order
            }
        }
}
```

```
{
  "data": {
    "feed": [
      {
        "id": "cjq2e0wj1wgbg0a98myh22yrp",
        "title": "demo",
        "categories": [
          {
            "name": "Productivity"
          }
        ],
        "paragraphes": [
          {
            "content": "okay cool  is the third article !!!!
demo updated ",
            "order": 0
          },
          {
            "content": "This is the first par. ",
            "order": 1
          },
          {
            "content": "demo paragrapjh  again ",
            "order": 2
          }
        ]
      },
```

Overall Workflow

List of all the articles


Create a new article (with paragraphes)
Filters (not implemented)
Toggle the sidebar
Pagination
View the article

demo
Productivity
Content of the first paragrah: okay cool is the third article !!!! demo updated
156 | 156 | 2

Reading Article
Productivity
Content of the first paragrah: This is a reading article.
156 | 156 | 2

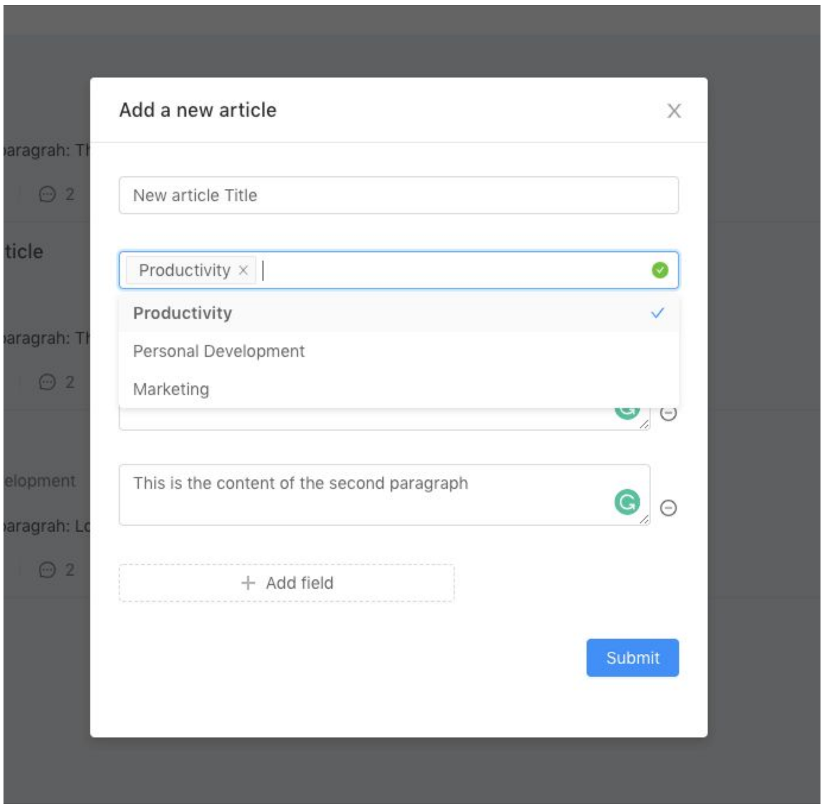
Article #1
Personal Development
Content of the first paragrah: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus venenatis eget
156 | 156 | 2

©2018 Created by Baptiste Debever

The routing is handled by React Router. We have 2 routes:

- The home route: “/” returning the list of articles
- The article route: /:articleIe returning an individual article

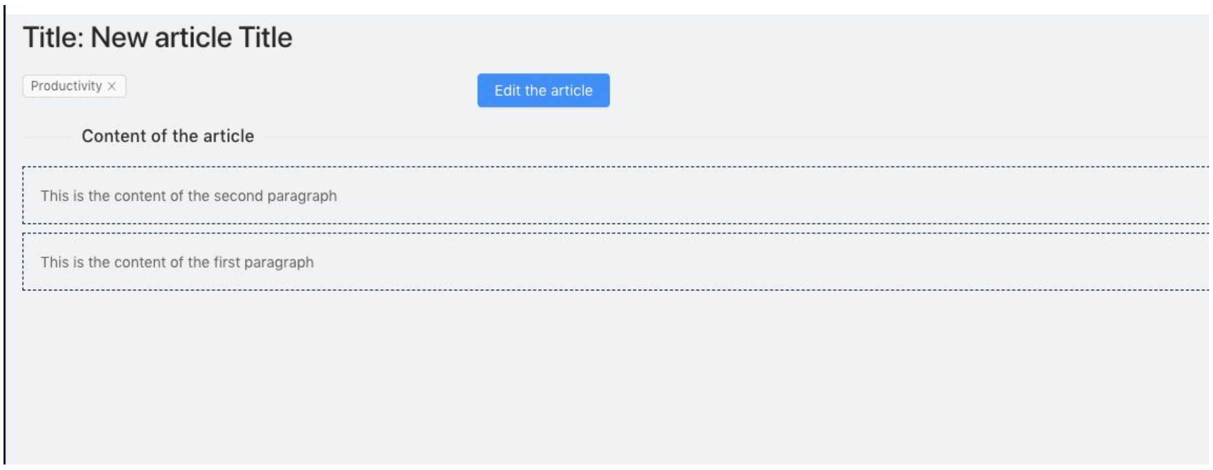
Create an article



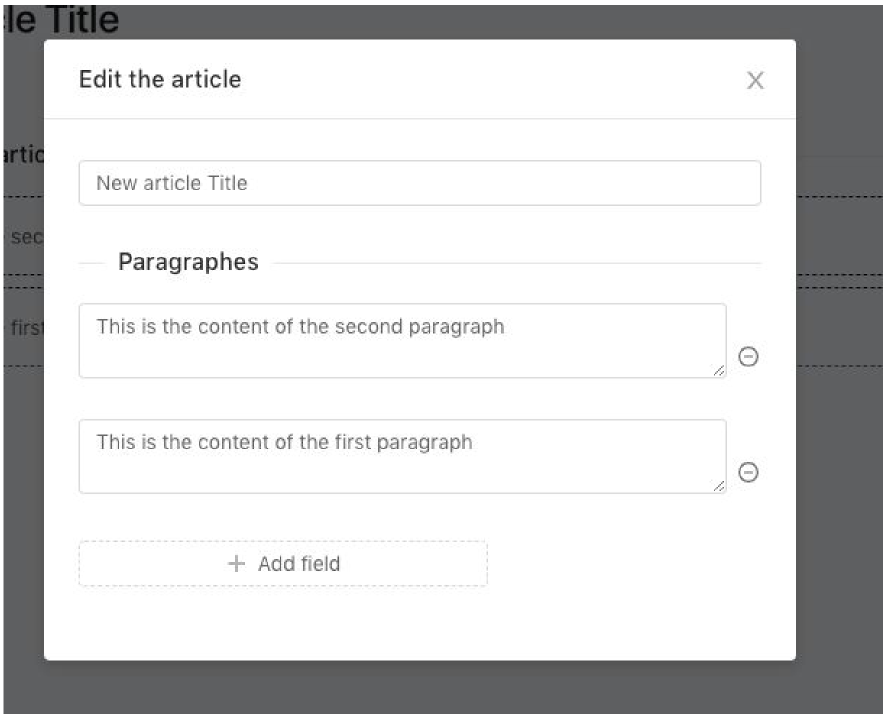
- We fill in the title of the article
- We have also added a field called categories, with all the categories stored in the database
- Finally, we add as many paragraphs as we want; we can reorder them later.
- And we submit! 👋

Update an article

From an individual article page, we can drag-and-drop the paragraphs and change their order. When we hover an article, the hand icon appears.


Title: New article Title
Productivity
Edit the article
Content of the article
This is the content of the second paragraph
This is the content of the first paragraph

Then, we can also click on the edit button in order to edit the content of the paragraphs, add a new paragraph or delete a paragraph.


Edit the article
New article Title
Paragraphes
This is the content of the second paragraph
This is the content of the first paragraph
+ Add field

As soon as we make a change to a paragraph and blur out of the text area, the change is committed to the database. Same for the delete.