

GraphQL

CFP Programador full-stack

Introduction

What is GraphQL?

- Is an API Query Language developed and open-sourced by Facebook that provides a more efficient, powerful and flexible alternative to REST
- Enables declarative data fetching where a client can specify exactly the data that is needed from an API.
- GraphQL servers exposes a single endpoint

History

- Facebook started using GraphQL in 2012 for their native mobile apps
- The first time Facebook publicly spoke about GraphQL was at React.js Conf 2015
- It was believed it was a limited to use only with React
- Netflix implemented Falcor
- Coursera cancelled their project to use GraphQL

GraphQL

CFP
Programador
full-stack

GraphQL vs. REST

A more efficient alternative to REST

- When REST concept was developed, client application were relatively simple
- REST was a good fit for many applications
- API landscape has radically changed over the years
- Three factors that have been challenging the way APIs are designed

1. Increased mobile usage creates need for efficient data loading

- Increased mobile usage, low-powered devices and sloppy networks were the initial reasons
- GraphQL minimizes the amount of data that needs to be transferred
- Majorly improves applications operating under these conditions

2. Variety of different frontend frameworks and platforms

- Difficulty to build and maintain one API that would fit the requirements of all
- GraphQL allows each client to access precisely to the data they need.

3. Fast development & expectation for rapid feature development

- Continuous deployment has become a standard
- Rapid iterations and frequent product updates are indispensable

REST

The way data is exposed often needs to be modified

Scenario

In a blogging application, an app needs to display the titles of the posts of a specific user. The same screen also displays the names of the last 3 followers of that user.

REST

- Has become the standard for designing web APIs
- Stateless servers
- Structured access to resources
- Inflexible to keep up with the rapidly changing requirements of the clients

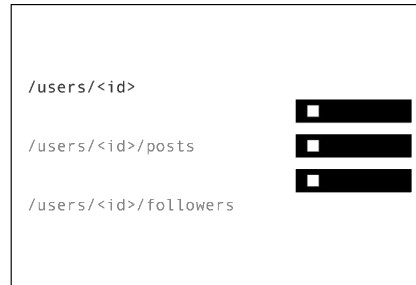
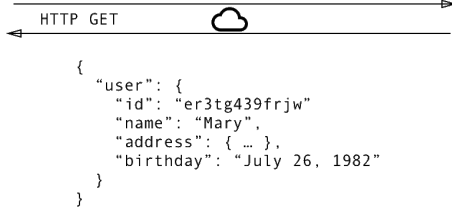
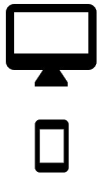
GraphQL

- Stateless
- Solves shortcomings
- More efficient

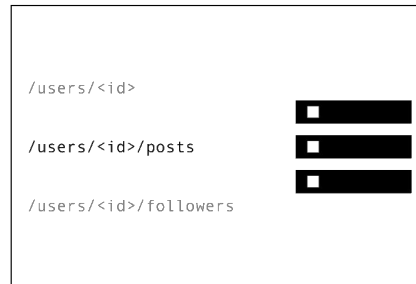
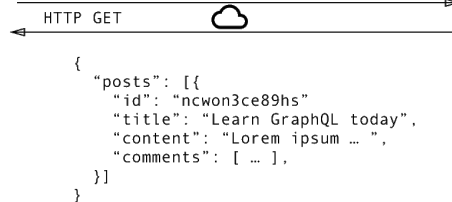
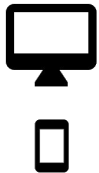
Data Fetching - REST

- Three requests to different endpoints to fetch the required data.
- Overfetching, since the endpoints return additional information that is not needed.

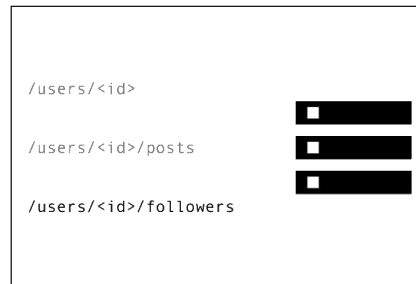
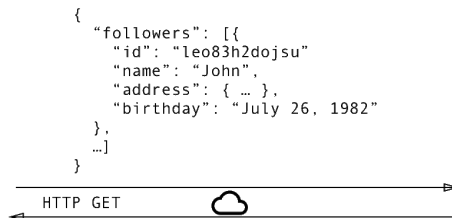
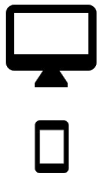
1



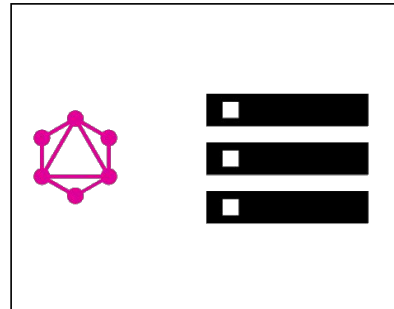
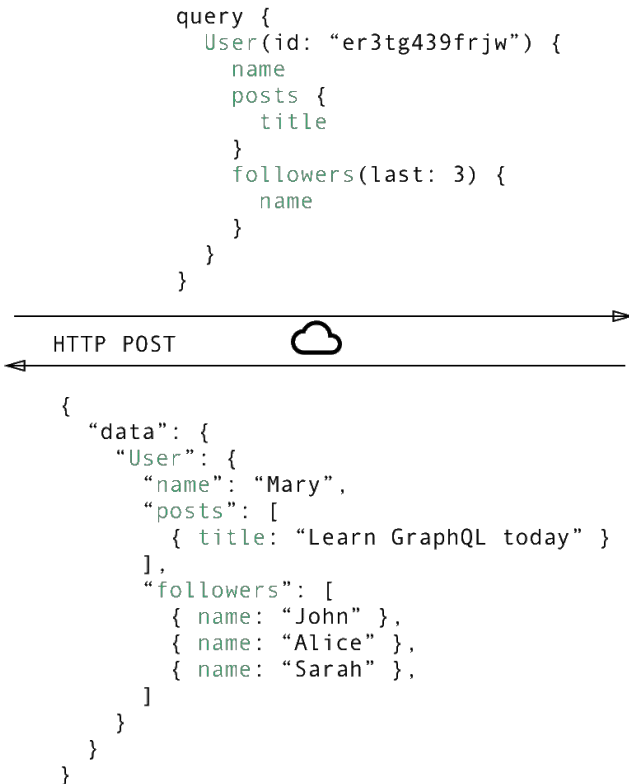
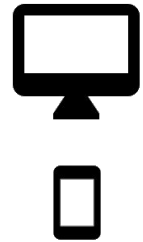
2



3



Data Fetching - GraphQL



- The client can specify exactly the data it needs in a query.
- The structure of the server's response follows the nested structure defined in the query.



No more Over- and Underfetching

Overfetching: when the client downloads more information than is actually required.

Underfetching:

Endpoint does not provides enough of the required information.

The client will have to make additional requests

Situation where a client needs a list of elements and one more request per element to fetch required data

Rapid Product Iterations on the Frontend

- Structure REST APIs endpoint according to the views
 - Does not allow rapid iterations on the frontend
 - High risk that there is more (or less) data required.
-
- With GraphQL, changes on client-side can be done without extra work
 - Clients specify their exact data requirements
 - No backend adjustments

Insightful Analytics on the Backend

- Allows you to have fine-grained insights about the data that's requested on the backend
- It can help in evolving an API and deprecating specific fields that are not requested anymore
- Low-level performance monitoring of requests.
- Instrumenting and measuring performance of resolver functions for bottlenecks.

Benefits of a Schema & Type System

- Uses a strong type system to define the capabilities of an API
- All the types that are exposed in an API are written down in a schema using Schema Definition Language (SDL)
- The Schema serves as the contract between the client and the server.
- Easy to test applications by mocking data structures

GraphQL

CFP Programador full-stack

Architecture

Architecture - Uses Cases

3 different architectures that include a GraphQL server

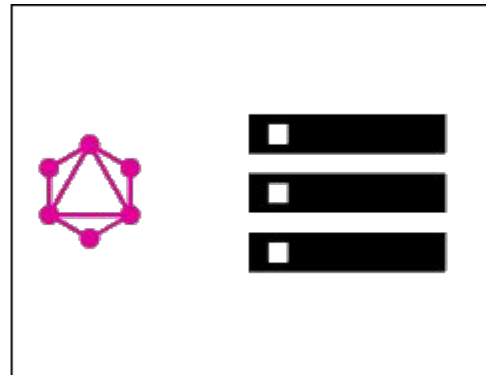
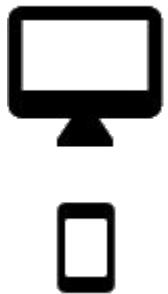
1 - GraphQL server with a
connected database

2 - GraphQL server that is a thin layer in front of a
number of third party or legacy systems and
integrates them through a single GraphQL API

3 - A hybrid approach of a connected database
and third party or legacy systems that can all be
accessed through the same GraphQL API

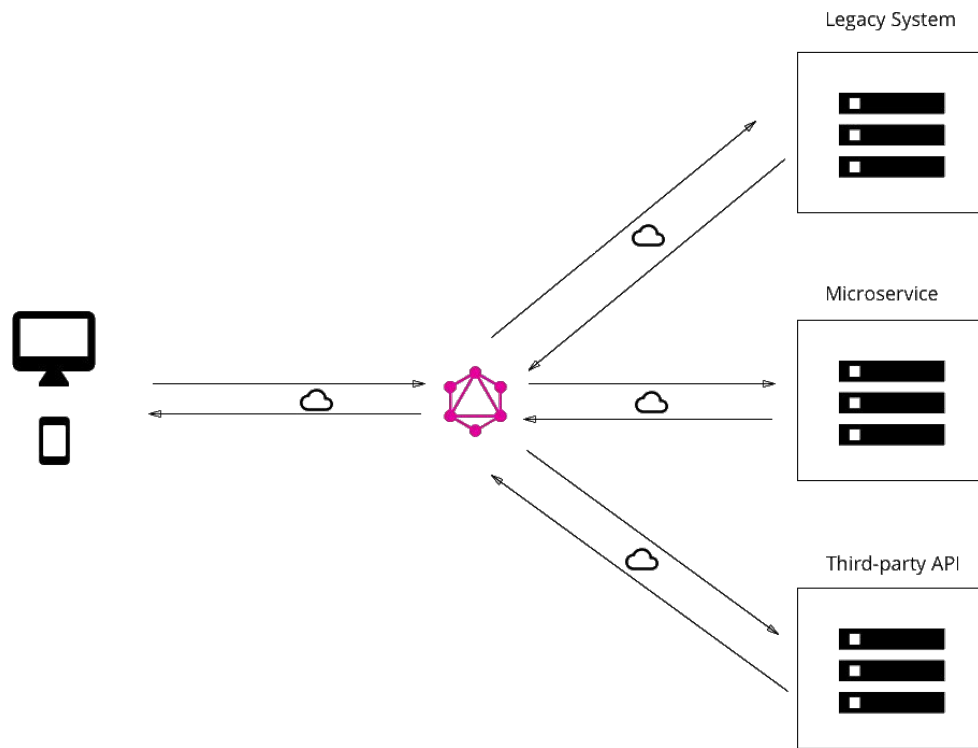
1. GraphQL server with a connected database

- Single server that implements GraphQL specification
- Server read the query's payload and fetches the information from the DB (Resolving)
- Constructs the response object and returns it to the client
- Transport-layer agnostic (TCP, WebSockets)
- Does not care about DB or stored data format



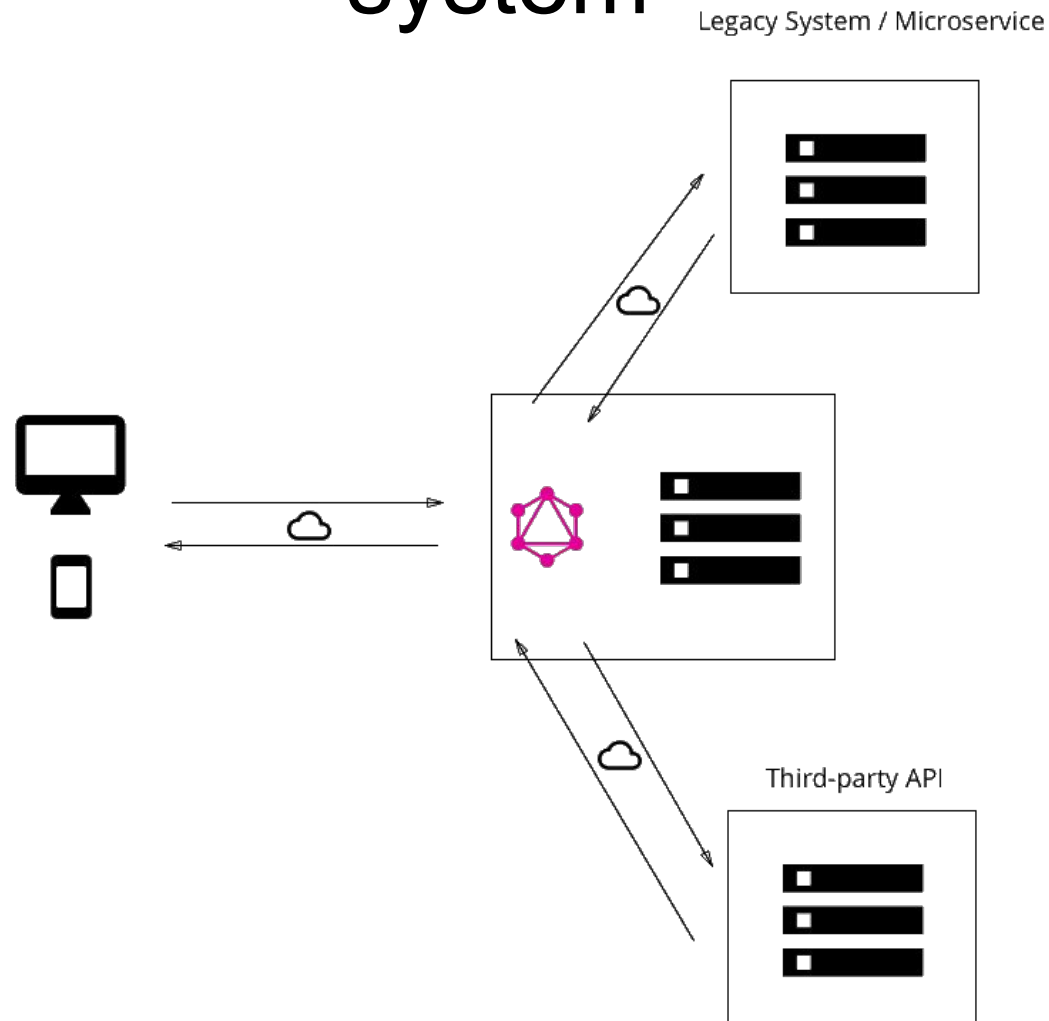
A standard greenfield architecture with one GraphQL server that connects to a single database.

2. GraphQL layer that integrates existing systems



- Legacy systems make practically impossible to build products that need access to multiple systems
- Unify systems and hide complexity behind a GraphQL API
- GraphQL does not care about the data sources to fetch needed data

3. Hybrid approach with connected database and integration of existing system



How do we gain this flexibility with GraphQL?

The payload of a GraphQL query (or mutation) consists of a set of fields

In the GraphQL server implementation each of these fields actually corresponds to exactly one function that's called a resolver

The purpose of a resolver function is to fetch the data for its field

Resolver Functions



```
query {  
  User(id: "er3tg4sa9frj") {  
    name  
    followers(first: 5) {  
      name  
      age  
    }  
  }  
}
```

Resolvers



```
User(id: String!): User  
name(user: User!): String  
age(user: User!): Int  
friends(first: Int, user: User!): [User!]!  
...
```

GraphQL

CFP Programador full-stack

Core Concepts

The Schema Definition Language (SDL)

GraphQL has its own type system that's used to define the schema of an API

```
type Person {  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}
```

```
type Post {  
  title: String!  
  author: Person!  
}
```



Basic Queries

Request

```
query{  
  allPeople {  
    name  
  }  
}
```

Response

```
{  
  "allPeople": [  
    { "name": "Johnny" },  
    { "name": "Sarah" },  
    { "name": "Alice" }]  
}
```

Basic Queries

GraphQL also allows for naturally querying nested information

For example, loading all the posts that a Person has written:

Request

```
{
  allPeople {
    name
    age
    posts {
      title
    }
  }
}
```

Response

```
"allPeople": [
  {
    "name": "Johnny",
    "age": 23,
    "posts": [{
      "title": "GraphQL is awesome"
    }]
  },
  ...
]
```

Basic Queries

In GraphQL, each field can have zero or more arguments if that is specified in the schema.

For example, `allPeople` field could have a “last” parameter to return a specific number of people:

Request

```
{  
  allPeople(last: 2) {  
    name  
  }  
}
```

Response

```
"allPeople": [  
  {  
    "name": "Sarah"  
  },  
  {  
    "name": "Alice"  
  }  
]
```

Writing data with Mutations

Three kinds of mutations:

- Create
- Update
- Delete

Mutations follow the same syntactical structure as queries, but they always need to start with the mutation keyword

Writing data with Mutations

Request

```
mutation{  
  createPerson(name: "Bob", age: 36) {  
    name  
    age  
  }  
}
```

Response

```
"createPerson": {  
  "name": "Bob",  
  "age": 36  
}
```

Writing data with Mutations

GraphQL types have unique IDs generated by the server.

```
type Person {  
  Id: ID!  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}
```

Request

```
mutation{  
  createPerson (name: "Alice", age: 36)  
  {  
    id  
  }  
}
```

Response

```
"createPerson": {  
  "id": "ck1o9omiv0c2f0"  
}
```

Realtime Updates with Subscriptions

Realtime connection to the server to get informed about important events

When a client subscribes to an event, it will initiate a steady connection to the server

Unlike queries and mutations that follow a typical “request-response-cycle” subscriptions represent a stream of data sent over to the client

Realtime Updates with Subscriptions

Subscribing to an event

```
subscription{  
  newPerson{  
    name  
    age  
  }  
}
```

Opens a connection



Client listening

...

Server sends information

```
"newPerson": {  
  "name": "Jane",  
  "age": 23  
}
```



New person created data

Defining a Schema

One of the most important concepts when working with GraphQL API.

It specifies the capabilities of the API and defines how clients can request data.

A schema is a collection of GraphQL types.

Root types:

```
type Query { ... }  
type Mutation { ... }  
type Subscription { ... }
```

Defining a Schema

```
type Query {  
  allPeople(last: Int!): [Person!]!  
}
```

```
type Mutation {  
  createPerson(name: String!, age:  
Int!): Person!  
}
```

```
type Subscription {  
  newPerson: Person!  
}
```

```
type Person {  
  name: String!  
  age: Int!  
  posts: [Post!]!  
}
```

```
type Post {  
  title: String!  
  author: Person!  
}
```

GraphQL

CFP
Programador
full-stack

Backend

Technologies

- GraphQL-yoga: Fully-featured GraphQL server built on top of Express, apollo-server, graphql-js and more.
- Prisma: replaces traditional ORMs. Use the Prisma client to implement your GraphQL resolvers and simplify database access
- GraphQL Playground: “GraphQL IDE” that allows to interactively explore the functionality of a GraphQL API

Creating a new project

- Install [Yarn](#)
- Open your terminal and run the following commands:

```
mkdir hackernews-node
```

```
cd hackernews-node
```

```
yarn init -y
```

This will create a new directory called “hackernews-node” and initializes it with configuration files.

Creating a raw GraphQL server

- Open the project in Visual Studio
- Create a folder called src
- Inside src folder create index.js

Start building GraphQL server

Let's add a dependency to the project

```
yarn add graphql-yoga
```

Graphql-yoga y a fully featured GraphQL server. Based on Express.js

GraphQL-yoga features

- GraphQL spec-compliant
- Supports file upload
- Realtime functionality with GraphQL subscriptions
- Works with TypeScript typings
- Out-of-the-box support for GraphQL Playground
- Extensible via Express middlewares
- Resolves custom directives in your GraphQL schema
- Query performance tracing
- Accepts both application/json and application/graphql content-types
- Runs everywhere: Can be deployed via now, up, AWS Lambda, Heroku etc.

Start building GraphQL server

Open index.js and type the following:

```
const { GraphQLServer } = require('graphql-yoga')
```

```
const typeDefs = `  
type Query {  
  info: String!  
}`
```

typeDefs constant defines your GraphQL schema.

Start building GraphQL server

```
// 2
const resolvers = {
  Query: {
    info: () => `This is the API of a Hackernews Clone`
  }
}
```

Resolvers object is the implementation of the GraphQL schema

Start building GraphQL server

```
// 3
const server = new GraphQLServer({
  typeDefs,
  resolvers,
})
server.start(() => console.log(`Server is running on
http://localhost:4000`))
```

The schema and resolvers are bundled and passed to the GraphQL server.

This tells the server, what API operations are accepted and how they should be resolved.

Testing GraphQL server

In the root directory of your project run:

```
node src/index.js
```

The server will be running on `http://localhost:4000`

Open your browser and you'll see a [GraphQL Playground](#)

Send your first query

```
query {  
  info  
}
```

Try updating resolver function to return null
instead of ``This is the API of a Hackernews Clone``

Extending the schema definition

Add a new root type called “Link” with the following fields:

- id - mandatory ID type.
- description - mandatory String.
- url - mandatory String.

Add the following in “Query”:

- Add a field “feed” to retrieve a list of all stored links, the list and Link cannot be null.

Implement resolver functions

Let's use a links variable (at least for now) to simulate DB

```
let links = [{  
  id: 'link-0',  
  url: 'www.howtographql.com',  
  description: 'Fullstack tutorial for GraphQL'  
}]
```

Add feed in Query resolver to return links variable.

Add Link resolver

As Link has its own fields, resolver function will return them using parent argument.

I.e:

Id: (parent) => parent.id

Implement the same line for other fields

Is parent argument needed?

- Is the result of the previous resolver execution level.
- On the first level, it invokes the **feed** resolver and return data stored in links variable
- For the second level, graphql is smart enough to invoke the resolvers of **Link** type.
- The incoming parent object is the element inside **links** list.

As the implementation is trivial, resolvers for Link can be omitted

Mutations

Let's add a new post Mutation in the schema

```
type Mutation {  
  post(url: String!, description: String!): Link!  
}
```

Update resolver

```
Mutation: {  
  // 2  
  post: (parent, args) => {  
    const link = {  
      id: `link-${idCount++}`,  
      description: args.description,  
      url: args.url,  
    }  
    links.push(link)  
    return link  
  }  
},
```

Testing the mutation

```
mutation {  
  post(  
    url: "www.prisma.io"  
    description: "Prisma replaces traditional ORMs"  
  ) {  
    id  
  }  
}
```

Add the following in the Schema:

```
type Query {  
  # Fetch a single link by its `id`  
  link(id: ID!): Link  
}  
  
type Mutation {  
  # Update a link  
  updateLink(id: ID!, url: String, description: String):  
  Link  
  
  # Delete a link  
  deleteLink(id: ID!): Link  
}
```

Implement the resolvers for each one

Adding a Database

Access the
Database Directly

Dealing with SQL in
resolvers is complex

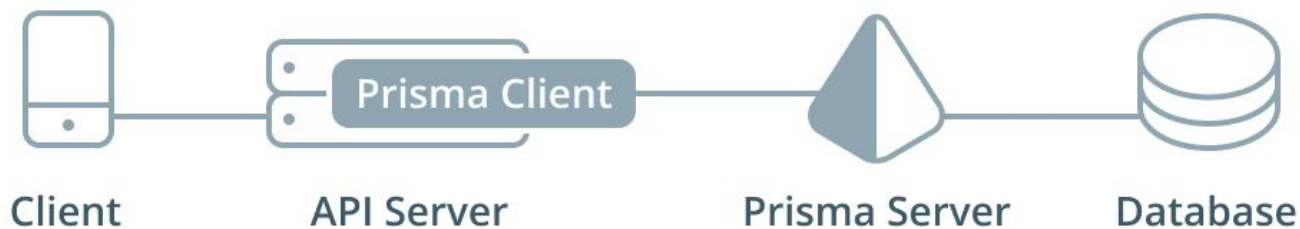
Tooling won't be able to
help finding issues

Use ORM for
abstraction for DB

Usually implement
simple solutions for DB
access

Prisma

Data access layer, takes care of resolving queries.



Setting up prisma

- Create a new directory called *prisma* in the root folder of the project
- Create 2 files inside prisma directory:
 - *prisma.yml* (main configuration file)
 - *datamodel.prisma* (definition of datamodel)

Setting up prisma

- Each model will be mapped to a table in the underlying database.
- As Prisma uses GraphQL SDL for model definitions you can copy Link definition from schema into *datamodel.prisma*

```
type Link {  
  id: ID! @id  
  createdAt: DateTime! @createdAt  
  description: String!  
  url: String!  
}
```

Setting up prisma

Add the following contents to *prisma.yml*

```
# The HTTP endpoint for your Prisma API
```

```
endpoint: ''
```

```
# Points to the file that contains your datamodel
```

```
datamodel: datamodel.prisma
```

```
# Specifies language & location for the generated  
Prisma client
```

```
generate:
```

```
- generator: javascript-client
```

```
  output: ../src/generated/prisma-client
```

Setting up prisma

- **endpoint:** HTTP endpoint for the Prisma API
- **datamodel:** points to the datamodel file.
- **generate:** specifies in which language the Prisma client should be generated and where it will be located

Setting up prisma

Before deploying the service, we need to install the Prisma CLI

```
yarn global add prisma
```

```
prisma deploy
```

1. Select the Demo server, register with Prisma Cloud.
2. Select the region for Demo server. Hit enter twice to use suggested values for service and stage

Once the command has finished running, the CLI writes the endpoint for the Prisma API to `prisma.yml`

Setting up prisma

The last step is to generate the Prisma Client for the datamodel

```
prisma generate
```

The prisma client is now generated and located in `hackernews-node/src/generated/prisma-client`

Setting up prisma

To use the client, import prisma instance as the following example:

```
const { prisma } = require('./generated/prisma-client')
async function main() {
  // Create a new link
  const newLink = await prisma.createLink({
    url: 'www.prisma.io',
    description: 'Prisma replaces traditional ORMs',
  })
  console.log(`Created new link: ${newLink.url} (ID: ${newLink.id})`)
  // Read all links from the database and print them to the console
  const allLinks = await prisma.links()
  console.log(allLinks)
}
main().catch(e => console.error(e))
```

Connecting the Prisma Client

Updating the resolver functions to use Prisma client - refactor

- Open index.js and entirely remove the links array as well as the idCount variable.
- Update the implementation of the resolver functions

Connecting the Prisma Client

```
const resolvers = {
  Query: {
    info: () => `This is the API of a Hackernews Clone`,
    feed: (root, args, context, info) => {
      return context.prisma.links()
    },
  },
  Mutation: {
    post: (root, args, context) => {
      return context.prisma.createLink({
        url: args.url,
        description: args.description,
      })
    },
  },
}
```


Connecting the Prisma Client

```
feed: (root, args, context, info)
```

Context Argument

- Plain JS object that every resolver can read from or write to
- A way to pass arbitrary data or functions to the resolvers
- Attach prisma client instance to the **context**

Connecting the Prisma Client

Understanding the **feed** resolver

```
feed: (root, args, context, info) => {  
  return context.prisma.links()  
},
```

- Accesses a prisma object on **context**
- Prisma client instance lets you access the DB through the Prisma API
- It exposes a number of methods (CRUD)

Connecting the Prisma Client

Understanding the **post** resolver

```
post: (root, args, context) => {  
  return context.prisma.createLink({  
    url: args.url,  
    description: args.description, })),
```

- Invoke a function on the prisma client
- **createLink** method from the Prisma client API.
- **args** data as arguments

Attaching Prisma client to **context**

1. Add a new dependency to the project

```
yarn add prisma-client-lib
```

2. Import the prisma client instance into index.js

```
const { prisma } =  
require('./generated/prisma-client')
```

3. Update the instantiation of the **GraphQLServer**

```
const server = new GraphQLServer({  
  typeDefs: './src/schema.graphql',  
  resolvers,  
  context: { prisma },  
})
```



1- Add a User model in datamodel.prisma and the corresponding User type in the schema

- id
- name
- email (@unique)
- password
- links

2- Add a postedBy (type User) field in Link type in datamodel and Schema

Remember to run generate and deploy!

Hook

- Create a post-deployment hook in prisma.yml

```
hooks:  
  post-deploy:  
    - prisma generate
```

Authentication

Let's implement signup and login functionality that allows a user to authenticate against the GraphQL server.

Add the following **mutations**

```
signup(email: String!, password: String!, name: String!): AuthPayload
```

```
login(email: String!, password: String!): AuthPayload
```

Authentication

Add a new type for authentication

```
type AuthPayload {  
  token: String  
  user: User  
}
```


Authentication

Before implementing resolver functions, let's refactor

1. First, create a new directory called resolvers and add four files to it: Query.js, Mutation.js, User.js and Link.js.
2. Move the implementation from index to each file as follows (query.js example):

```
function feed(parent, args, context, info) {  
  return context.prisma.links()  
}  
  
module.exports = {  
  Feed,  
}
```

Authentication

Let's add resolvers for signup and login

```
async function signup(parent, args, context, info) {  
  // 1  
  const password = await bcrypt.hash(args.password, 10)  
  // 2  
  const user = await context.prisma.createUser({ ...args, password })  
  // 3  
  const token = jwt.sign({ userId: user.id }, APP_SECRET)  
  // 4  
  return {  
    token,  
    user,  
  }  
}
```

Authentication

```
async function login(parent, args, context, info) {  
  // 1  
  const user = await context.prisma.user({ email: args.email })  
  if (!user) {  
    throw new Error('No such user found')  
  }  
  // 2  
  const valid = await bcrypt.compare(args.password, user.password)  
  if (!valid) {  
    throw new Error('Invalid password')  
  }  
  const token = jwt.sign({ userId: user.id }, APP_SECRET)  
  // 3  
  return {  
    token,  
    user,  
  }  
}
```

Authentication

Add a new dependency to the project

```
yarn add jsonwebtoken bcryptjs
```

Create a new file inside src called utils.js, it will be used in some places of our code

Authentication

Add the following code to utils.js

```
const jwt = require('jsonwebtoken')
const APP_SECRET = 'GraphQL-is-aw3some'

function getUserId(context) {
  const Authorization = context.request.get('Authorization')
  if (Authorization) {
    const token = Authorization.replace('Bearer ', '')
    const { userId } = jwt.verify(token, APP_SECRET)
    return userId
  }
  throw new Error('Not authenticated')
}

module.exports = {
  APP_SECRET,
  getUserId,
}
```

Authentication

APP_SECRET is used to sign the JWTs

getUserId function is a helper for authentication. It retrieves authorization header from the context, verifies the jwt and retrieves the User's ID.

Add the following to Mutation.js

```
const bcrypt = require('bcryptjs')
const jwt = require('jsonwebtoken')
const { APP_SECRET, getUserId } = require('../utils')
```

Authentication

```
const Authorization = context.request.get('Authorization')
```

When initializing the context, just prisma is attached, adjust the instantiation of the GraphQLServer:

```
const server = new GraphQLServer({  
  typeDefs: './src/schema.graphql',  
  resolvers,  
  context: request => {  
    return {  
      ...request,  
      prisma,  
    }  
  },  
})
```

Authentication

Requiring authentication for the post mutation

```
function post(parent, args, context, info) {  
  const userId = getUserId(context)  
  return context.prisma.createLink({  
    url: args.url,  
    description: args.description,  
    postedBy: { connect: { id: userId } },  
  })  
}
```


Authentication

To resolve the `postedBy` relation, add the following code into `Link.js`:

```
function postedBy(parent, args, context) {  
  return context.prisma.link({ id: parent.id  
}).postedBy()  
}
```

```
module.exports = {  
  postedBy,  
}
```

Authentication

To resolve the links relation, add the following code into User.js:

```
function links(parent, args, context) {  
  return context.prisma.user({ id: parent.id }).links()  
}
```

```
module.exports = {  
  links,  
}
```

Authentication

Let's modify the resolvers const.

1. Import all resolvers files

resolvers **const** should be

```
const resolvers = {  
  Query,  
  Mutation,  
  User,  
  Link  
}
```

Testing Authentication Flow

1. Create a new user with name, email and password of your choice. The mutation must return a token and the user id

Testing Authentication Flow

Copy the authentication token and open another tab in the Playground.

Inside that new tab, open the HTTP HEADERS pane in the bottom-left corner and specify the Authorization header.

Replace the `__TOKEN__` placeholder in the following snippet with the copied token:

```
{  
  "Authorization": "Bearer __TOKEN__"  
}
```

1. Create 2 posts
2. Try login functionality and return the following data:
 - a. Token
 - b. User email
 - c. Url and descriptions of user's links

Subscriptions

Luckily, Prisma comes with out-of-the-box support for subscriptions.

For each model in your Prisma datamodel, Prisma lets you subscribe to the following events:

- A new model is created
- An existing model updated
- An existing model is deleted

You can subscribe to these events using the **\$subscribe** method of the Prisma client.

Subscriptions

Subscribing to new **Link** elements

Add the type Subscription in the schema

```
type Subscription {  
  newLink: Link  
}
```

Let's implement the resolver for newLink field

Subscriptions

Resolvers for subscriptions are slightly different than the ones for queries and mutations:

1. Rather than returning any data directly, they **return an AsyncIterator** which subsequently is used by the GraphQL server to push the event data to the client.
2. Subscription resolvers are wrapped inside an object and need to be provided as the value for a subscribe field. You also need to provide another field called resolve that actually returns the data from the data emitted by the AsyncIterator.

Subscriptions

Create a new file called Subscription.js inside resolvers folder.

```
function newLinkSubscribe(parent, args, context, info) {  
  return context.prisma.$subscribe.link({ mutation_in: [ 'CREATED' ]  
}).node()  
}  
  
const newLink = {  
  subscribe: newLinkSubscribe,  
  resolve: payload => {  
    return payload  
  },  
}  
  
module.exports = {  
  newLink,  
}
```

Subscriptions

1. Add an import statement for the Subscription module
2. Update the definition of the resolvers

Testing Subscriptions

Send the following subscription:

```
subscription {  
  newLink {  
    id  
    url  
    description  
    postedBy {  
      id  
      name  
      email  
    }  
  }  
}
```