## 

## [**What is Prisma?**](https://www.prisma.io/docs/understand-prisma/prisma-introduction-what-why-how-j9ff/#what-is-prisma)

Prisma replaces traditional ORMs and simplifies database workflows:

* *Access*: Type-safe database access with the auto-generated Prisma client (in JavaScript, TypeScript, Go)
* *Migrate*: Declarative data modelling and migrations (optional)
* *Manage*: Visual data management with [Prisma Admin](https://www.prisma.io/docs/prisma-admin/overview-el3e/)

It is used to build GraphQL, REST, gRPC APIs and a lot more. Prisma [currently supports](https://www.prisma.io/#database-connectors) MySQL, PostgreSQL, MongoDB.

Prisma is a standalone infrastructure component that sits on top of your database. You're then using a Prisma client (which is available in various languages) in your application server to connect to Prisma.

This enables you to talk to your database(s) through a simple and modern API ensuring highly performant and secure database access.

## [**Use cases**](https://www.prisma.io/docs/understand-prisma/prisma-introduction-what-why-how-j9ff/#use-cases)

Prisma is useful in any context where you're working with databases.

### [**Building GraphQL servers**](https://www.prisma.io/docs/understand-prisma/prisma-introduction-what-why-how-j9ff/#building-graphql-servers)

Prisma is the perfect tool for building GraphQL servers. The Prisma client is compatible with the Apollo ecosystem, has default support for GraphQL subscriptions and Relay-style pagination, provides end-to-end type safety and comes with a built-in dataloader to solve the N+1 problem.

## [**Why use Prisma?**](https://www.prisma.io/docs/understand-prisma/prisma-introduction-what-why-how-j9ff/#why-use-prisma)

### [**Simple database workflows**](https://www.prisma.io/docs/understand-prisma/prisma-introduction-what-why-how-j9ff/#simple-database-workflows)

Prisma's overall goal is to remove complexity from common database workflows and simplify data access in your applications:

* Type-safe database access thanks to the custom and auto-generated Prisma client.
* Simple and powerful API for working with relational data and transactions.
* Visual data management with Prisma Admin.
* Prisma unifies access to multiple databases at once (*coming soon*) and therefore drastically reduces complexity in cross-database workflows.
* Realtime streaming & event system for your database ensuring you're getting updates for all important events happening in your database.
* Automatic database migrations (optional) based on a declarative data model expressed using GraphQL's schema definition language (SDL).
* Other database workflows such as [data import, export](https://www.prisma.io/docs/prisma-cli-and-configuration/data-import-and-export-jsw9/) & more.

### [**A realtime layer for your database**](https://www.prisma.io/docs/understand-prisma/prisma-introduction-what-why-how-j9ff/#a-realtime-layer-for-your-database)

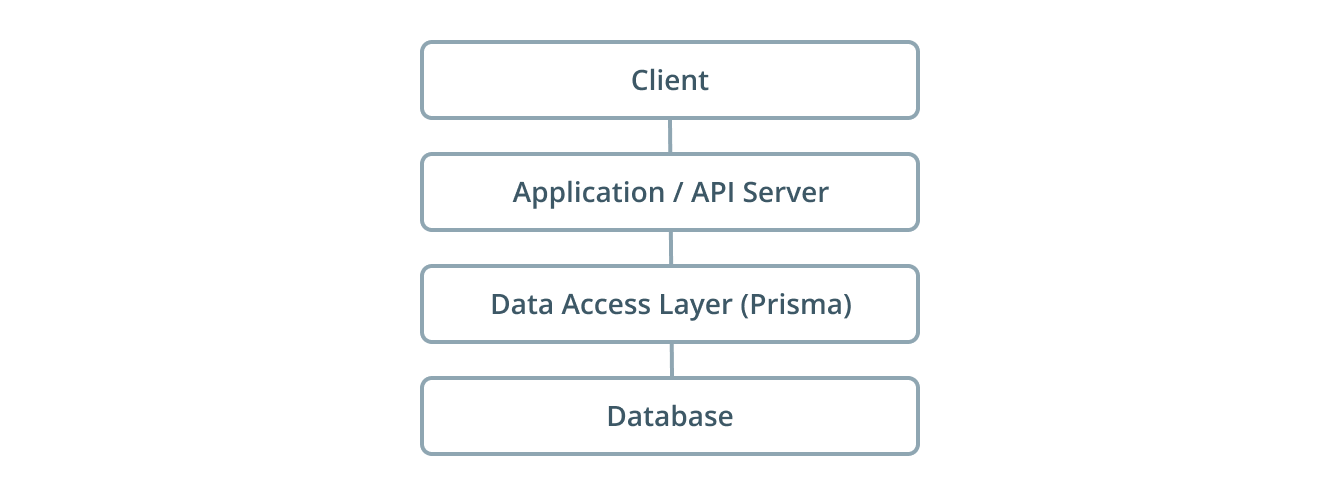
Some databases, such as RethinkDB or DynamoDB provide a realtime API out-of-the box. Such an API lets clients *subscribe* to any changes happening in the database. The vast majority of conventional databases however does not offer such a realtime API, and implementing it manually gets very complex.

Prisma offers a realtime API for every supported database, letting you subscribe to any database event, such as *creating*, *updating* or *deleting* data.

### [**Clean and layered architecture**](https://www.prisma.io/docs/understand-prisma/prisma-introduction-what-why-how-j9ff/#clean-and-layered-architecture)

**When developing application servers, most complexity lies in implementing a safe and well-organized database access with respect to *synchronization*, *query optimization*/*performance* and *security*. This becomes even more complicated when *multiple* databases are involved.**

**One common solution to this problem is the introduction of a dedicated *data access layer* (DAL) that abstracts away the complexities of database access. The DAL's API is consumed by the application server, allowing API developers to simply think about *what* data they need instead of worrying about *how* to securely and performantly retrieve it from the database.**

****

**Using a DAL ensures a clear separation of concerns and therefore improves maintainability and reusability of your code. Having some sort of database abstraction (be it a simple ORM library or a standalone infrastructure component) is best practice for smaller sized applications as well as for applications running at scale. It ensures the application server can talk to your database(s) in a secure and performant way.**

**Prisma is an auto-generated DAL following the same principles as industry-leading DALs (such as Twitter's Strato or Facebook's TAO) while still being accessible for smaller applications.**

**Prisma lets you start your project with a clean architecture from the beginning and saves you from writing the boilerplate that is otherwise required to glue together database and application server.**

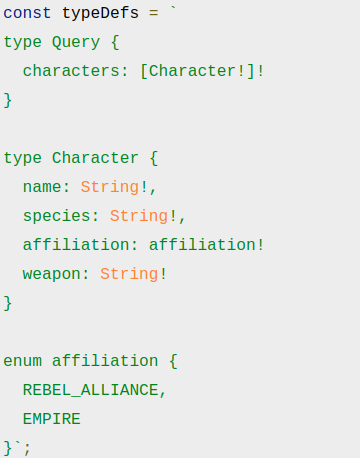
**Demo setup for Javascript-graphql-prisma**

**A common issue with GraphQl is connecting it’s server to our database. Prisma is an excellent tool that greatly simplifies interacting with our database via a GraphQL API.**

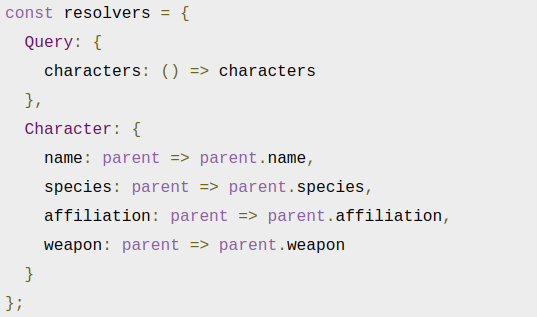
**Prisma provides a database client for your GraphQL API. This client makes your life as a developer significantly easier because it auto-generates the basic database commands for you. The Prisma CLI will even spin up a database within a docker container with just a few commands.**

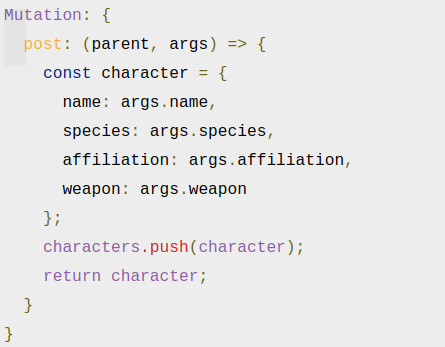
**First we need a demo Graphql project without prisma.With GraphQL you build a data schema and resolvers to control what data is returned and how you interact with it.**

**Data schema looks something like this initially**



**And the resolvers are as such**



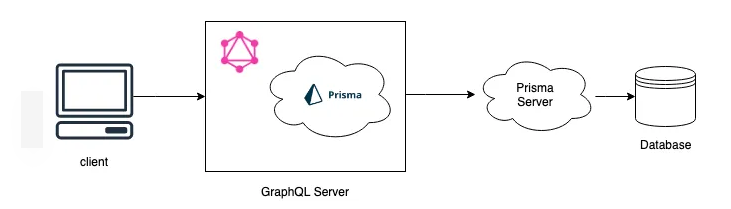


**What this is saying is that for a mutation, create a character with the args provided and push it to the characters array.**

## 

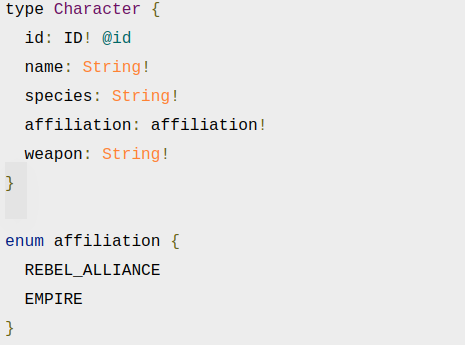
## **Creating a Prisma client**

**Prisma takes care of the challenge of connecting your server to your database through (1) a generated client and (2) a server that translates your GraphQL calls into commands for your database.**



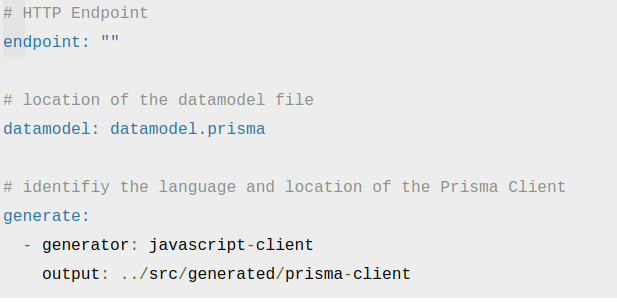
**We need two files**

1. **Datamodel.prisma : This defines the Character object for Prisma.the @id value here, this is so that every record created with Prisma is unique. Prisma will automatically generate the ID value for us, with each new record.**



1. **Prisma.yml : This file does the following:**

* **identifies the HTTP endpoint for your client (note this will be filled out by the CLI when we run the deploy command)**
* **defines the location of the datamodel file**
* **defines the application language**
* **defines the location of the (generated) Prisma Client**



**Now we need to actually build the prisma client, for this we need to install the prisma CLI globally then we’ll need to generate the Prisma Client code. The command for this is**

****

**Now with the infrastructure deployed into the cloud, you can generate the Prisma Client which will be used by your GraphQL API by running the following**

****

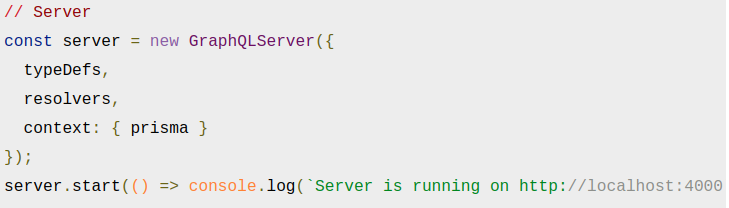
**This will setup our prisma client in a “generated/prisma-client” directory**

**In order to work with our Prisma Client, we will need to install the prisma-client-lib package**

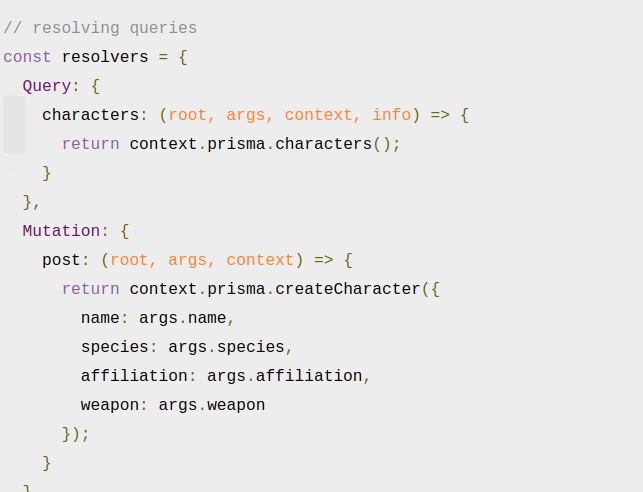
## **Connecting to the Prisma Client**

**the last step is to connect our GraphQL server to our generated Client. First we need to add out prisma client in our server file.**





**Finally our resolvers and queries will change accordingly**



**This modified the query and mutation to point to the Prisma Client and server in lieu of our local array.**

**Everything else like querying the api on GraphIql will remain the same.**

**ADVANTAGES**

**Database access requires building direct SQL queries or using a traditional Object-Relational Mapping (ORM) to interact with the database. SQL Queries are error-prone as they have to be sent over as strings. ORMs typically are very simplistic and difficult to scale to properly work with complex applications. Prisma took care of everything.**

**LIMITATION**

**The major restraints we encountered so far were**

1. **Currently, no support for Java. Only for Javascript and GO**
2. **Only supports Mysql, mongo and postgreSQL.**

**Connect Presto Running on docker using Prisma with node js**

1. Setup Presto using Docker.
2. Setup node js prisma server using following.

mkdir appname

cd appname

yarn init -y

mkdir src

touch src/index.js

yarn add graphql-yoga

1. Add following Content in index.js

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

// Type Definition

const typeDefs = `

type Query {

character: String!

}

`

// Resolvers

const resolvers = {

Query: {

character: () => `The force is strong with this API!`

}

}

// Server

const server = new GraphQLServer({

typeDefs,

resolvers,

})

server.start(() => console.log(`Server is running on http://localhost:4000`))

1. node src/index.js -> this will run and create graphql playground
2. Now Setup for client generation

mkdir prisma

touch prisma/prisma.yml

touch prisma/datamodel.prisma

1. Content of datamodel.prisma

type Character {

id: ID! @id

name: String!

species: String!

affiliation: affiliation!

weapon: String!

}

enum affiliation {

REBEL\_ALLIANCE

EMPIRE

}

1. Content of prisma.yml

# HTTP Endpoint

endpoint: ""

# location of the datamodel file

datamodel: datamodel.prisma

# identifiy the language and location of the Prisma Client

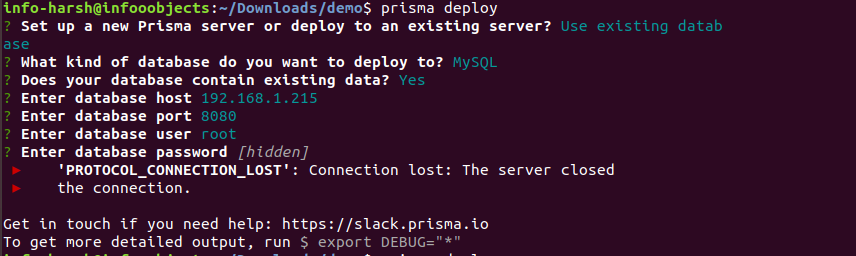
generate:

- generator: javascript-client

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

1. yarn global add prisma -> install the Prisma CLI globally
2. Now we are ready to deploy prisma give your presto configuration.

prisma deploy



1. We are unable to connect presto using prisma.