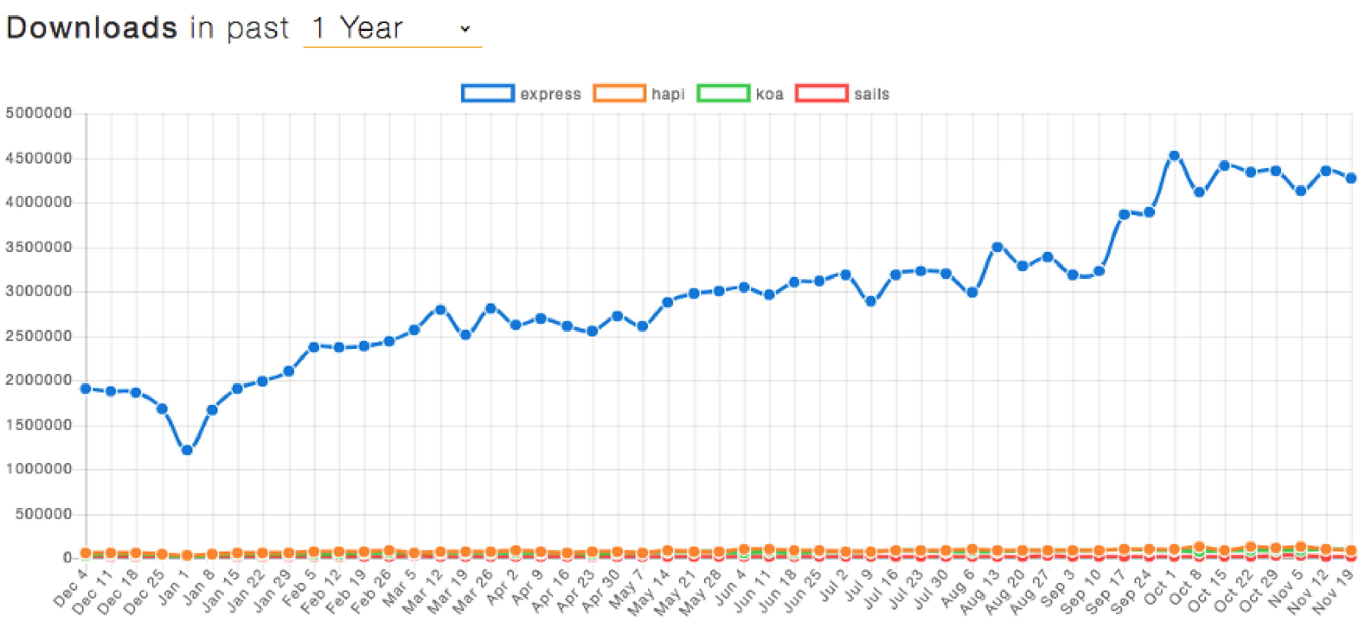
Prisma MENU

November 28, 2017

GraphQL Server Basics: The Network Layer

Nikolas Burk
@nikolasburk

Structure and Implementation of GraphQL Servers (Part II).



In the [previous article](), we covered a lot of ground with respect to the inner workings of GraphQL servers by learning about the [GraphQL schema]() and its fundamental role when it comes to *executing* queries and mutations.

While we learned how a GraphQL server is executing these operations using a *GraphQL engine*, we haven't touched on the aspect of actual *client-server communication:* the question how queries and their responses are *transported* over the network. That's what this article is about!

> *GraphQL servers can be implemented in any of your preferred programming languages. **This article is focussing on JavaScript** and the available libraries helping you to build your server, most notably: `express-graphql`, `apollo-server` and `graphql-yoga`.*

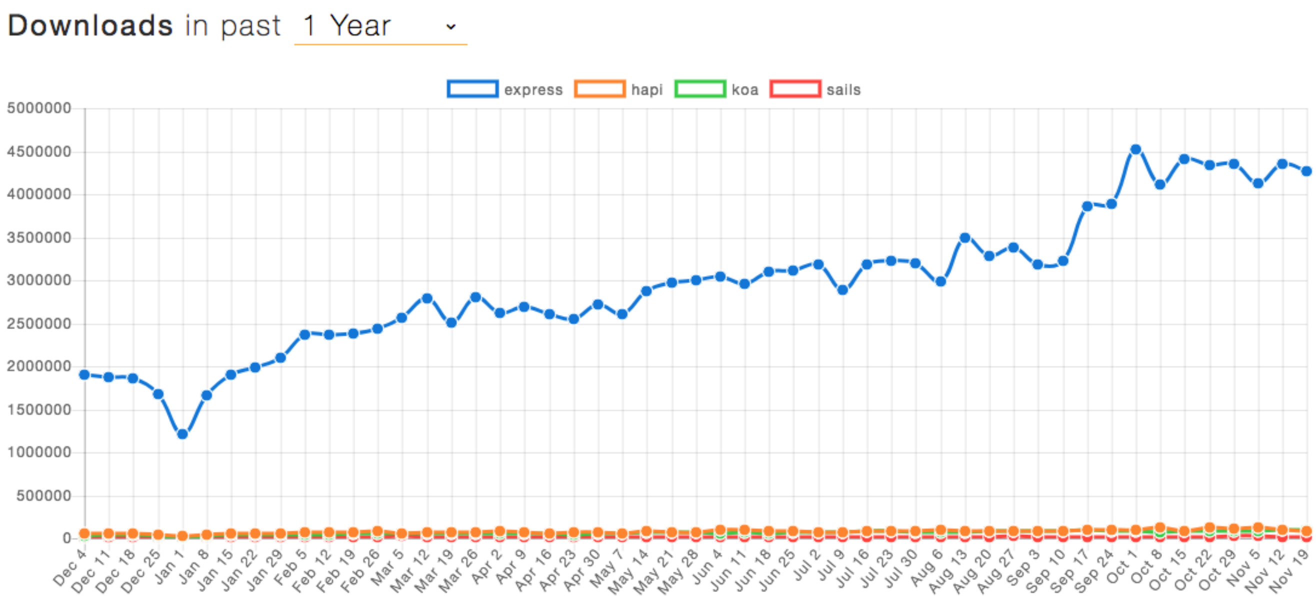
Serving GraphQL over HTTP

GraphQL is transport-layer agnostic

A key thing to understand about GraphQL is that it's actually agnostic to the way how data is transferred over the network. This means a GraphQL server potentially could work based on protocols other than HTTP, like WebSockets or the lower-level TCP. However, this article focusses on the most common way to implement GraphQL servers today, which is indeed based on HTTP.

Express.js is used as a strong & flexible foundation

> *The following section is mainly about Express.js and its concept of middleware that's used for GraphQL libraries like `express-graphql` and `apollo-server`. **If you're already familiar with Express, you can skip ahead to the next section.***



Comparison of [express](https://github.com/expressjs/), [hapi](https://hapijs.com/), [koa](http://koajs.com/) and [sail](https://sailsjs.com/) on [npm trends](http://www.npmtrends.com/)

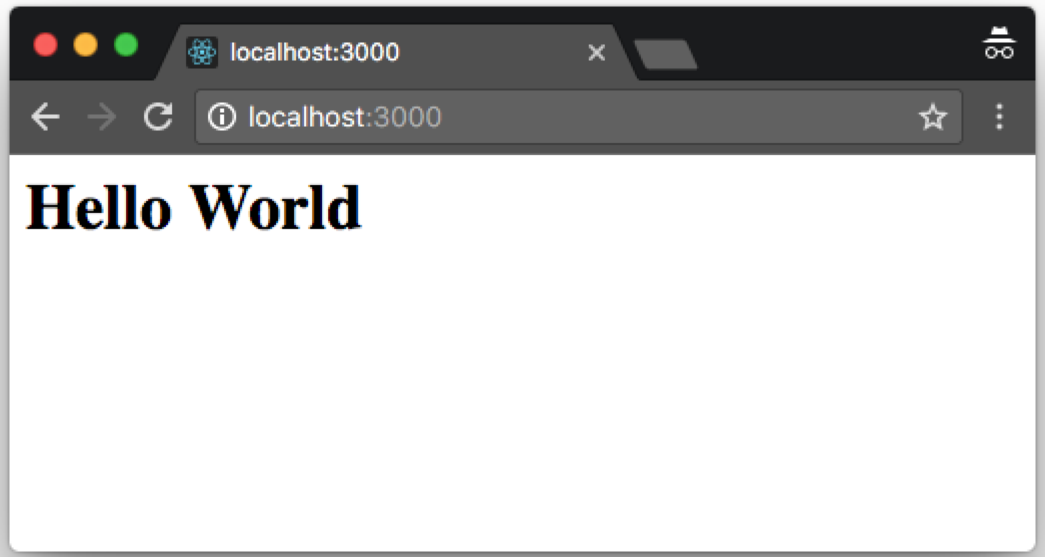
Comparison of express, hapi, koa and sail on npm trends

Express.js is by far the most popular JavaScript web framework. It shines thanks to its simplicity, flexibility and performance.

All you need to get started with your own web server is code looking as follows:

```
const express = require('express')
const app = express()

// respond with "hello world" when a GET request is received
app.get('/', function(req, res) {
  res.send('<h1>Hello World</h1>')
})
app.listen(3000)
```

After executing this script with Node.js, you can access the website on `http://localhost:3000` in your browser:



Hello World

You can easily add more *endpoints* (also called *routes*) to your server's API:

```
app.get('/goodbye', function(req, res) {
  res.send('<h1>Goodbye</h1>')
})
```

Or use another HTTP method, for example POST instead of GET:

```
app.post('/', function(req, res) {
  res.send('<h1>You just made a POST request</h1>')
})
```

Express provides great flexibility around implementing your server, allowing you to easily add functionality using the concept of *middleware*.

The key to flexibility and modularity in Express: Middleware

A middleware allows to *intercept* an incoming a request and perform dedicated tasks, while the request is processed or before the response is returned.

In essence, a middleware is nothing but a *function* taking three arguments:

- `req`: The incoming request from the client
- `res`: The response to be returned to the client
- `next`: A function to invoke the next piece of middleware

Since middleware functions have (write-)access to the incoming request objects as well as to the outgoing response objects, they are a very powerful concept that can shape the requests and responses according to a specific purpose.

Middleware can be used for many use cases, such as *authentication*, *caching*, *data transformation* and *validation*, *execution of custom business logic* and a lot more. Here is a simple example for *logging* that will print the time at which a request was received:

```
function loggingMiddleware(req, res, next) {
  console.log(`Received a request at: ${Date.now()}`)
  next()
}
app.use(loggingMiddleware)
```

The flexibility gained through this middleware approach is leveraged by frameworks like `graphql-express`, `apollo-server` or `graphql-yoga`, which are all based on Express!

Express & GraphQL

With everything we learned in the last article about the `graphql` function and GraphQL execution engines in general, we can already anticipate how an Express-based GraphQL server could work.

> *As Express offers everything we need to process HTTP requests, and GraphQL.js provides functionality for resolving queries, all we still need is the glue between them.*

This glue is provided by libraries like `express-graphql` and `apollo-server` which are nothing but middleware functions for Express!

GraphQL middleware glues together HTTP and GraphQL.js

`express-graphql`: Facebook's version for GraphQL middleware

`express-graphql` is Facebook's version for GraphQL middleware that can be used with Express and GraphQL.js. If you take a look at its source code, you'll notice that its core functionality is implemented in only a few lines of code.

Really, its main responsibility is twofold:

- Ensure that the GraphQL query (or mutation) contained in the body of an incoming POST request can be executed by GraphQL.js. So, it needs to parse out the query and forward it to the `graphql` function for execution.
- Attach the result of the execution to the response object so it can be returned to the client.

Using `express-graphql`, you can quickly start your GraphQL server as follows:

```
const express = require('express')
const graphqlHTTP = require('express-graphql')
const { GraphQLSchema, GraphQLObjectType, GraphQLString } = require('graphql')

const app = express()

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
  name: 'Query',
  fields: {
    hello: {
      type: GraphQLString,
      resolve: (root, args, context, info) => {
        return 'Hello World'
      }
    }
  }
})

app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true // enable GraphiQL
}))

app.listen(4000)
```

> *Executing this code with Node.js starts a GraphQL server on `http://localhost:4000/graphql`*

If you've read the previous article on the GraphQL schema, you'll have a pretty good understanding what the *lines 7 to 18* are being used for: We build a `GraphQLSchema` that can execute the following query:

```
query {
  hello
} # responds:  { "data": { "hello": "Hello World" } }
```

The new part about this code snippet however is the integrated network layer. Rather than writing a query inline and executing it directly with GraphQL.js (as demonstrated here), this time we're just setting up the server to wait for incoming queries which can then be executed against the `GraphQLSchema`.

You really don't need a lot more to get started with GraphQL on the server-side.

`apollo-server`: Better compatibility outside the Express ecosystem

At its essence, `apollo-server` is very similar to `express-graphql`, with a few minor differences. The main difference between the two is that `apollo-server` also allows for integrations with lots of other frameworks, like `koa` and `hapi` as well as for FaaS provides like AWS Lambda or Azure Functions. Each integration can be installed by appending the corresponding suffix for the package name, e.g. `apollo-server-express`, `apollo-server-koa` or `apollo-server-lambda`.

However, at the core it also simply is a middleware bridging the HTTP layer with the GraphQL engine provided by GraphQL.js. Here is what an equivalent implementation of the above `express-graphql`-based example looks like with `apollo-server-express`:

```
const express = require('express')
const bodyParser = require('body-parser')
const { graphqlExpress, graphiqlExpress } = require('apollo-server-express')
const { GraphQLSchema, GraphQLObjectType, GraphQLString } = require('graphql')

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        resolve: (root, args, context, info) => {
          return 'Hello World'
        },
      },
    },
  }),
})

const app = express()
```

```
app.use('/graphql', bodyParser.json(), graphqlExpress({ schema }))
app.get('/graphiql', graphiqlExpress({ endpointURL: '/graphql' })) // enable GraphiQL

app.listen(4000)
```

`graphql-yoga` : The easiest way to build a GraphQL server

Removing friction when building GraphQL servers

Even when using `express-graphql` or `apollo-server`, there are various points of friction:

- Requires installation of multiple dependencies
- Assumes prior knowledge of Express
- Complicated setup for using GraphQL subscriptions

This friction is removed by `graphql-yoga`, a simple library for building GraphQL servers. It essentially is a convenience layer on top of Express, `apollo-server` and a few other libraries to provide a quick way for creating GraphQL servers. (Think of it like create-react-app for GraphQL servers.)

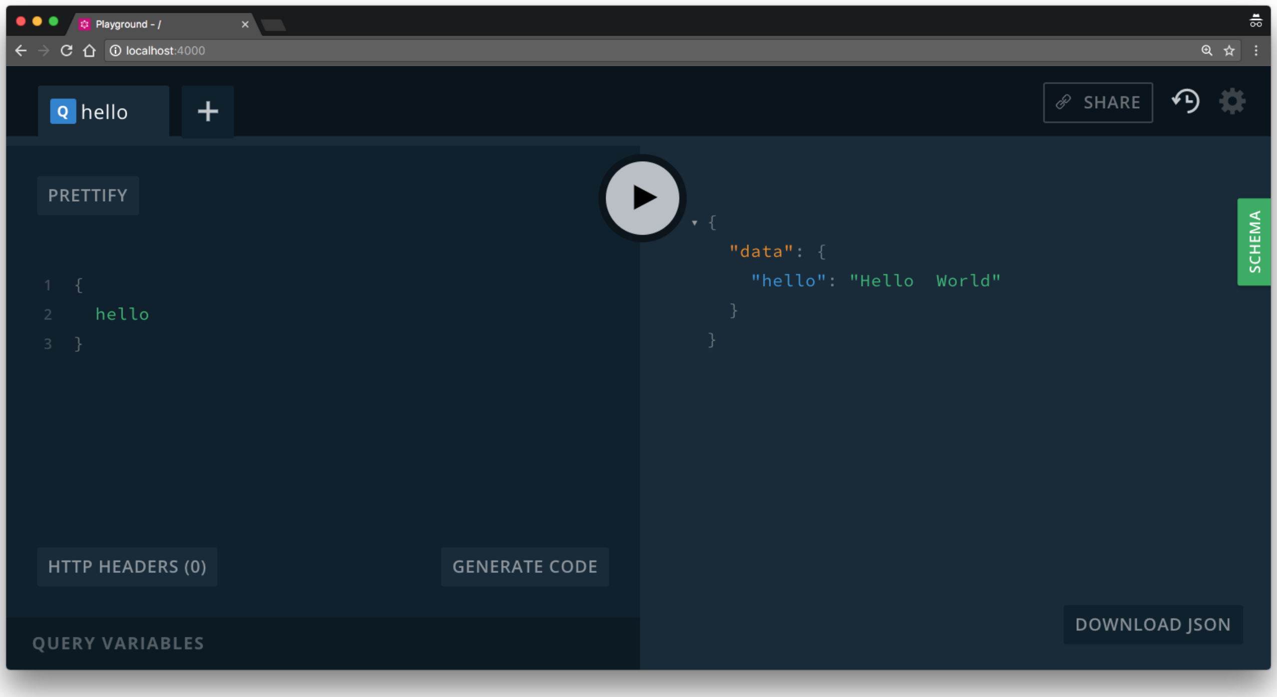
Here is what the same GraphQL server we already saw with `express-graphql` and `apollo-server` looks like:

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

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

const resolvers = {
  Query: {
    hello: (root, args, context, info) => 'Hello  World',
  },
}

const server = new GraphQLServer({ typeDefs, resolvers })
server.start() // defaults to port 4000
```

Note that a `GraphQLServer` can either be instantiated using a ready instance of `GraphQLSchema` or by using the convenience API (based on `makeExecutableSchema` from `graphql-tools`) as shown in the snippet above.

Built-in support for GraphQL Playgrounds, Subscriptions & Tracing

Note that `graphql-yoga` also has built-in support for `graphql-playground`. With the code above you can open the Playground at `http://localhost:4000`:



```
{
  hello
}
```

```
{
  "data": {
    "hello": "Hello  World"
  }
}
```

`graphql-yoga` also features a simple API for GraphQL subscriptions out-of-the-box, built on top of the `graphql-subscriptions` and `ws-subscriptions-transport` package.

You can check out how it works in this straightforward example.

To enable field-level analytics for your GraphQL operations executed with `graphql-yoga`, there also is built-in support for Apollo Tracing.

Conclusion

After having discussed the GraphQL execution process based on the `GraphQLSchema` and the concept of a GraphQL engine (such as GraphQL.js) in the last article, this time we focussed on the network layer. In particular, how a GraphQL server responds to HTTP requests by processing the queries (or mutations) with the execution engine.

In the Node ecosystem, Express is by far the most popular framework to build web servers thanks to its simplicity and flexibility. Consequently, the most common implementations for GraphQL servers are based on Express, most notably `express-graphql` and `apollo-server`. Both libraries are very similar with a few minor differences, the most important one being that `apollo-server` is also compatible with other web frameworks like `koa` and `hapi`.

`graphql-yoga` is a convenience layer on top of a number of other libraries (such as `graphql-tools`, `express`, `graphql-subscriptions` and `graphql-playground`) and is the easiest way for building GraphQL servers.

In the next article, we'll discuss the internals of the `info` argument that gets passed into your GraphQL resolvers.

Join the discussion

Follow @prisma on Twitter

Twitter

Don't miss the next post!

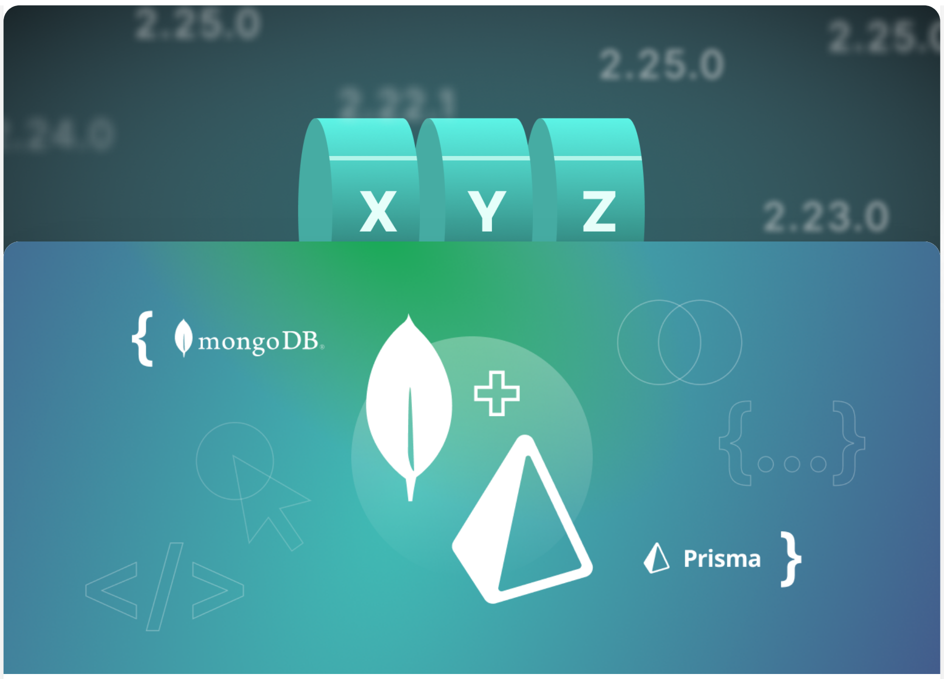
Sign up for the Prisma newsletter

your@email.com Join

Fullstack App With TypeScript, PostgreSQL, Next.js, Prisma & GraphQL: Data Modeling

August 18, 2021

This article is the first part of a course where we build a fullstack app with Next.js, GraphQL, TypeScript,Prisma and PostgreSQL. In this article, we'll create the data model and explore the different components of Prisma.



Prisma 2.27 Adds Preview Support for MongoDB

July 14, 2021

Today we are excited to expand the range of supported databases with preview support of **MongoDB**. Try it out and let us know what you think!

Prisma

PRODUCTS

Prisma Client

Prisma Migrate

Prisma Studio

Prisma 1 Cloud

Prisma Data Platform

Product Roadmap

RESOURCES

Docs

Get Started

API Reference
Examples

How to GraphQL

Data Guide

Enterprise Event

PRISMA WITH

Prisma with Next.js

Prisma with GraphQL

Prisma with Apollo

Prisma with NestJS

Prisma with Express

Prisma with hapi

COMMUNITY

Prisma Ambassador

Meet the Community

Prisma Day

Slack

GitHub

Discussions

GraphQL Meetup

TypeScript Meetup

Advanced TypeScript Trickery

Connect Dev Africa

COMPANY

About

Jobs We're hiring!