Phoomparin Mano

GraphQL in 3ms

brikl

If you like

slow websites

clap your hand! 👏

brikl

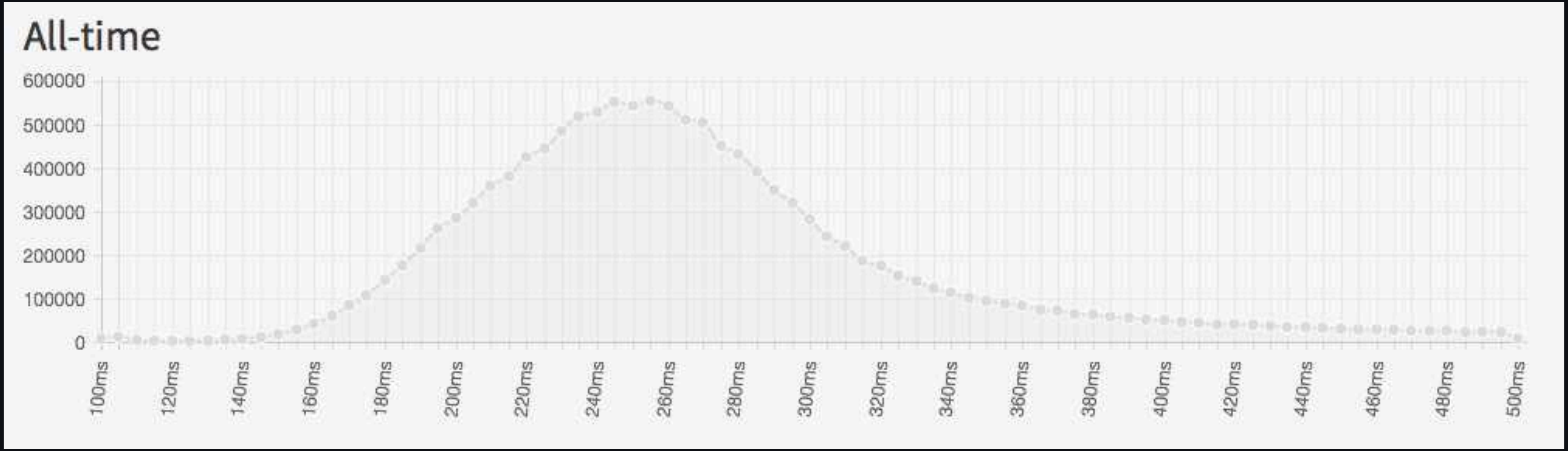
Phoomparin Mano (Poom)

Developer Advocate, BRIKL.

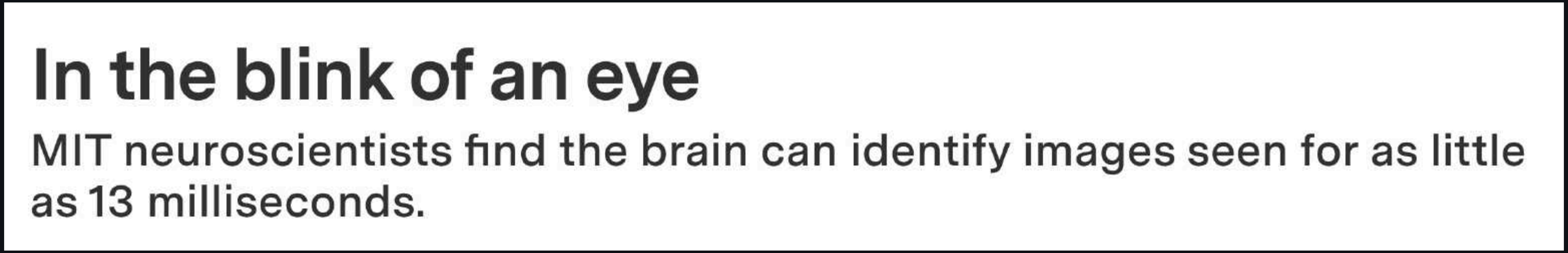
GitHub: @phoomparin

6:38
9
kantiGo
PESSUMAL
3
Zenhom
THE TERRIFIC
THE TERRIFIC
44
25

How *fast* can a *human* react?



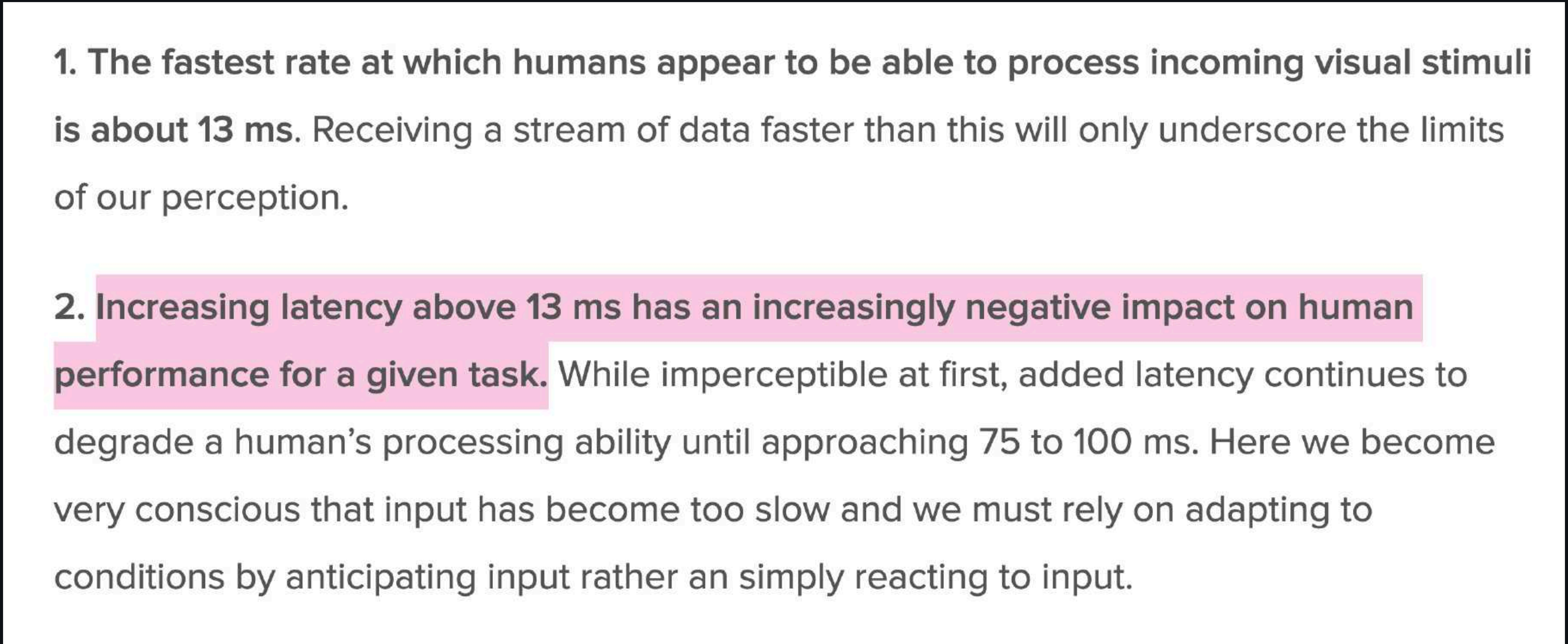
ref: The Human Benchmark Project



In the blink of an eye

MIT neuroscientists find the brain can identify images seen for as little as 13 milliseconds.

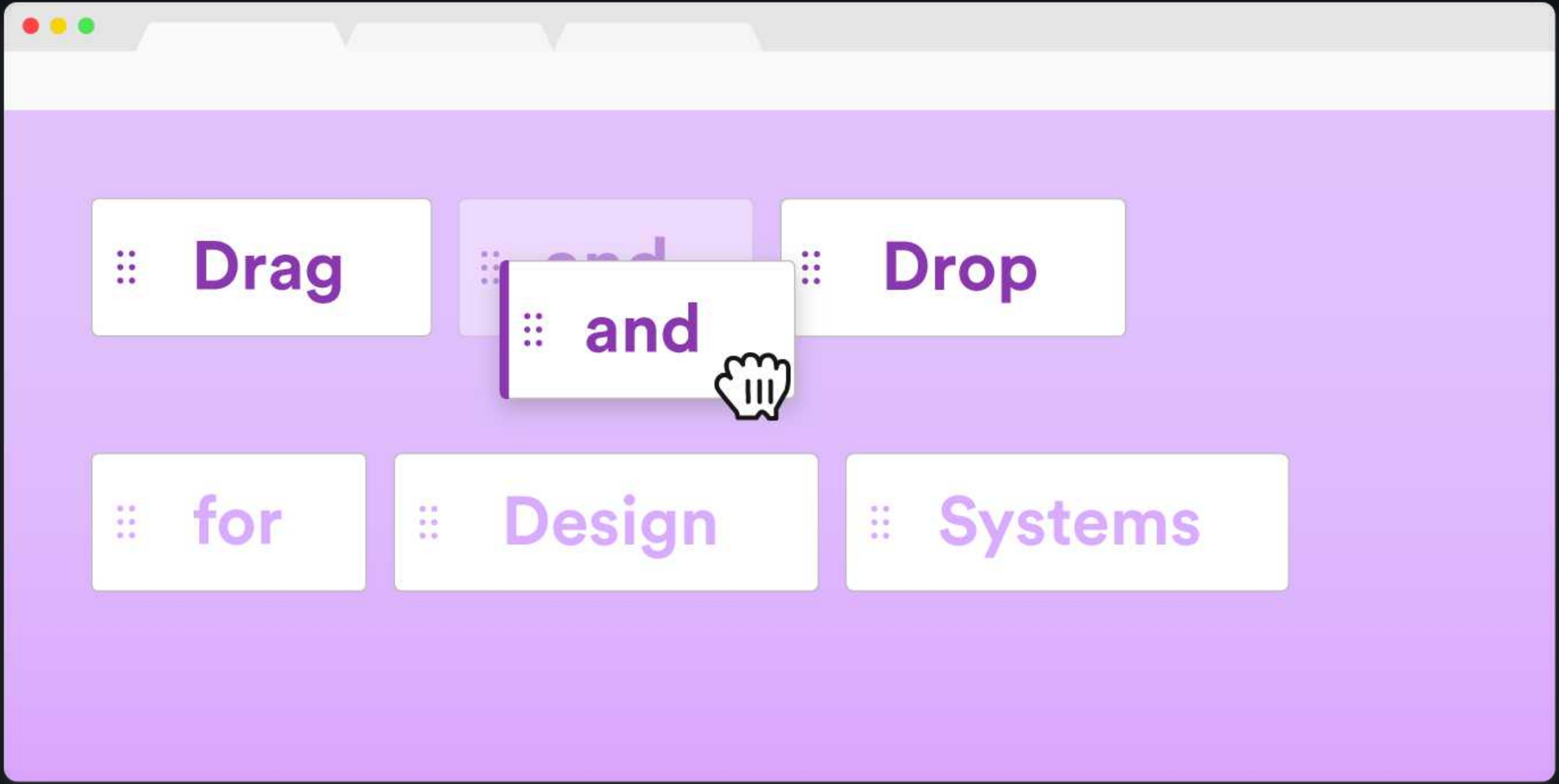
ref: MIT Media Lab



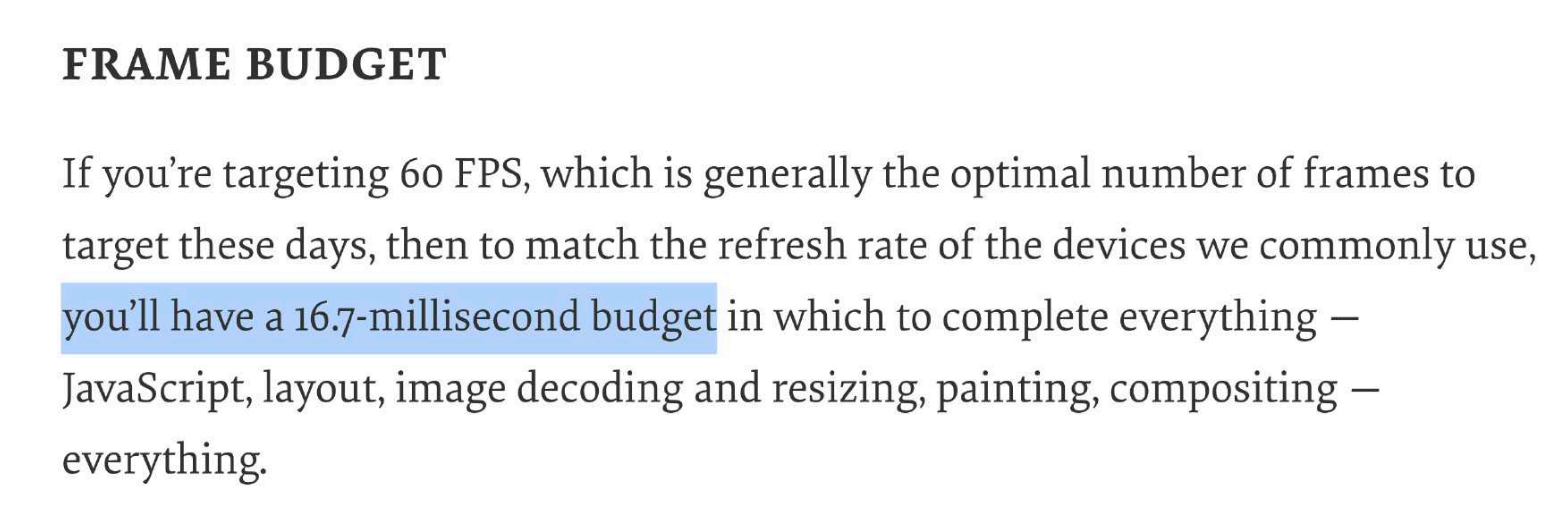
1. **The fastest rate at which humans appear to be able to process incoming visual stimuli is about 13 ms**. Receiving a stream of data faster than this will only underscore the limits of our perception.

2. **Increasing latency above 13 ms has an increasingly negative impact on human performance for a given task.** While imperceptible at first, added latency continues to degrade a human's processing ability until approaching 75 to 100 ms. Here we become very conscious that input has become too slow and we must rely on adapting to conditions by anticipating input rather an simply reacting to input.

ref: PubNub Blog



ref: uxdesign.cc

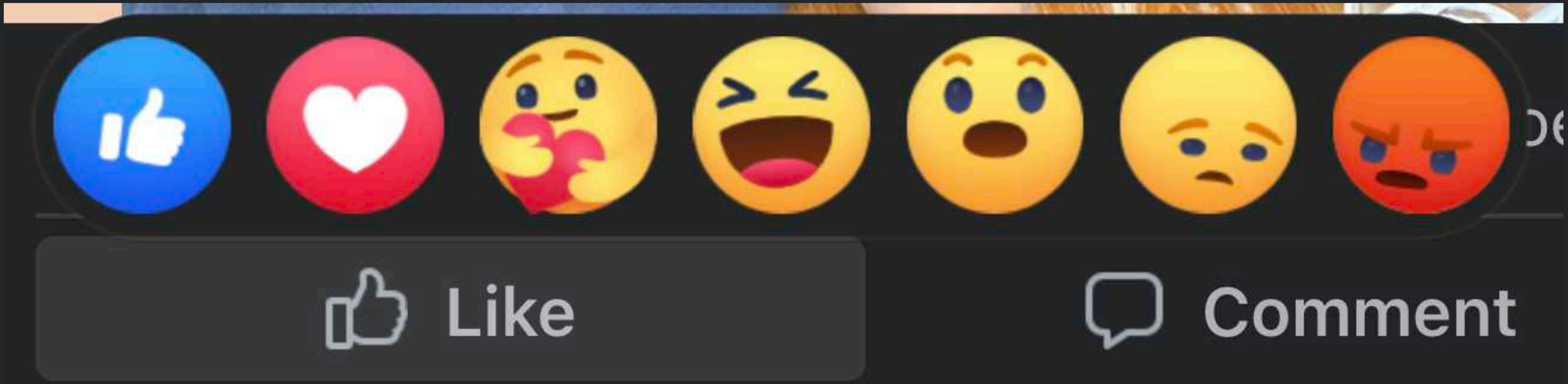


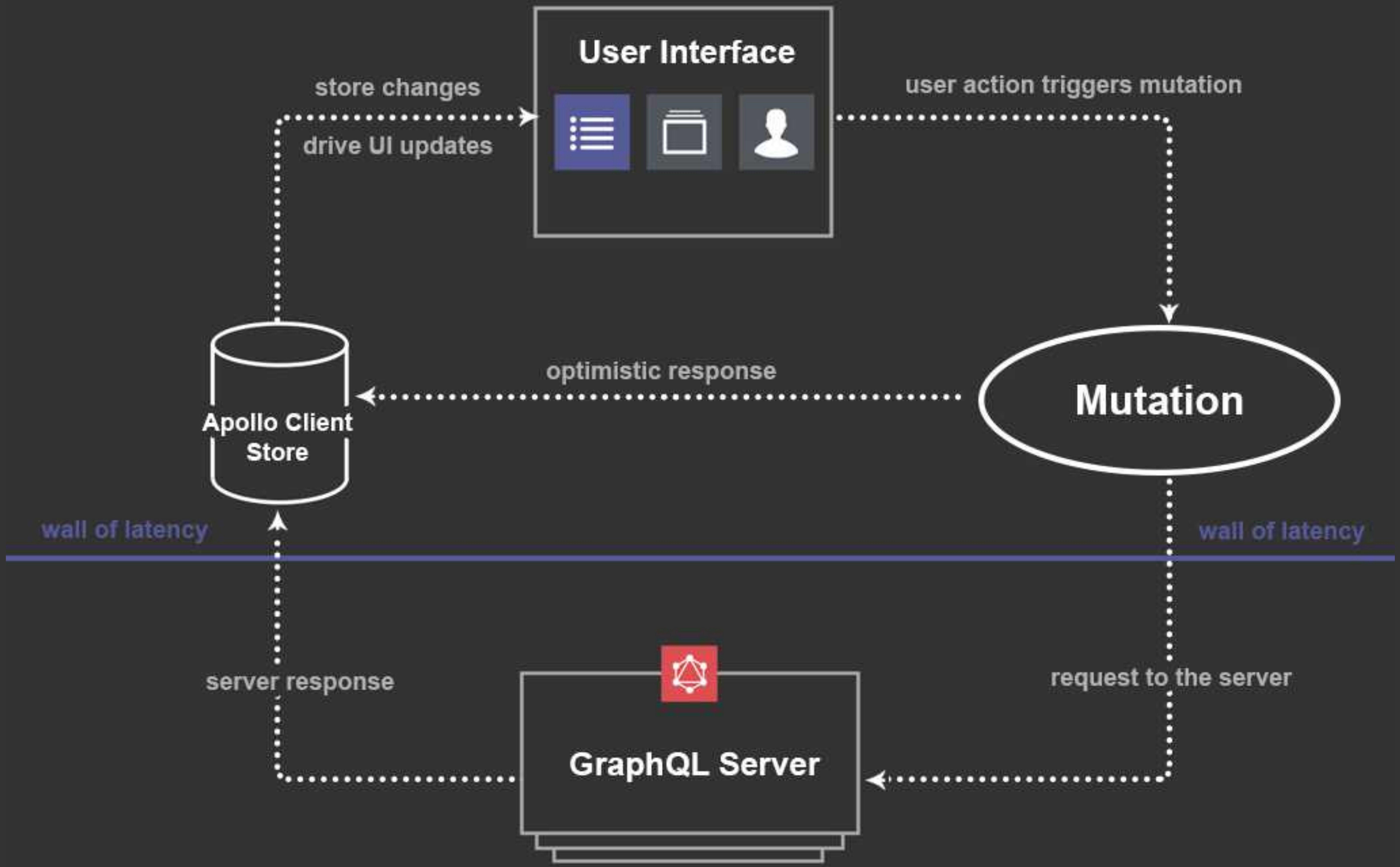
FRAME BUDGET

If you're targeting 60 FPS, which is generally the optimal number of frames to target these days, then to match the refresh rate of the devices we commonly use, you'll have a 16.7-millisecond budget in which to complete everything — JavaScript, layout, image decoding and resizing, painting, compositing — everything.

ref: Smashing Magazine

Optimistic Updates




User Interface
store changes
drive UI updates
user action triggers mutation
optimistic response
Apollo Client Store
Mutation
wall of latency
wall of latency
server response
request to the server
GraphQL Server

WHAT IF

you need the *response* ***quickly***
or *rollback on error* is ***tricky?***

When you can't rely on
optimistic updates, your *API must be fast.*

If *UI* must be *fast*,

What about the *API*?

Performance-Critical *Apps*

Multiplayer Games 🎮

Stock Trading 📈

Control Systems 🖲️

Mission-Critical Apps 🚀

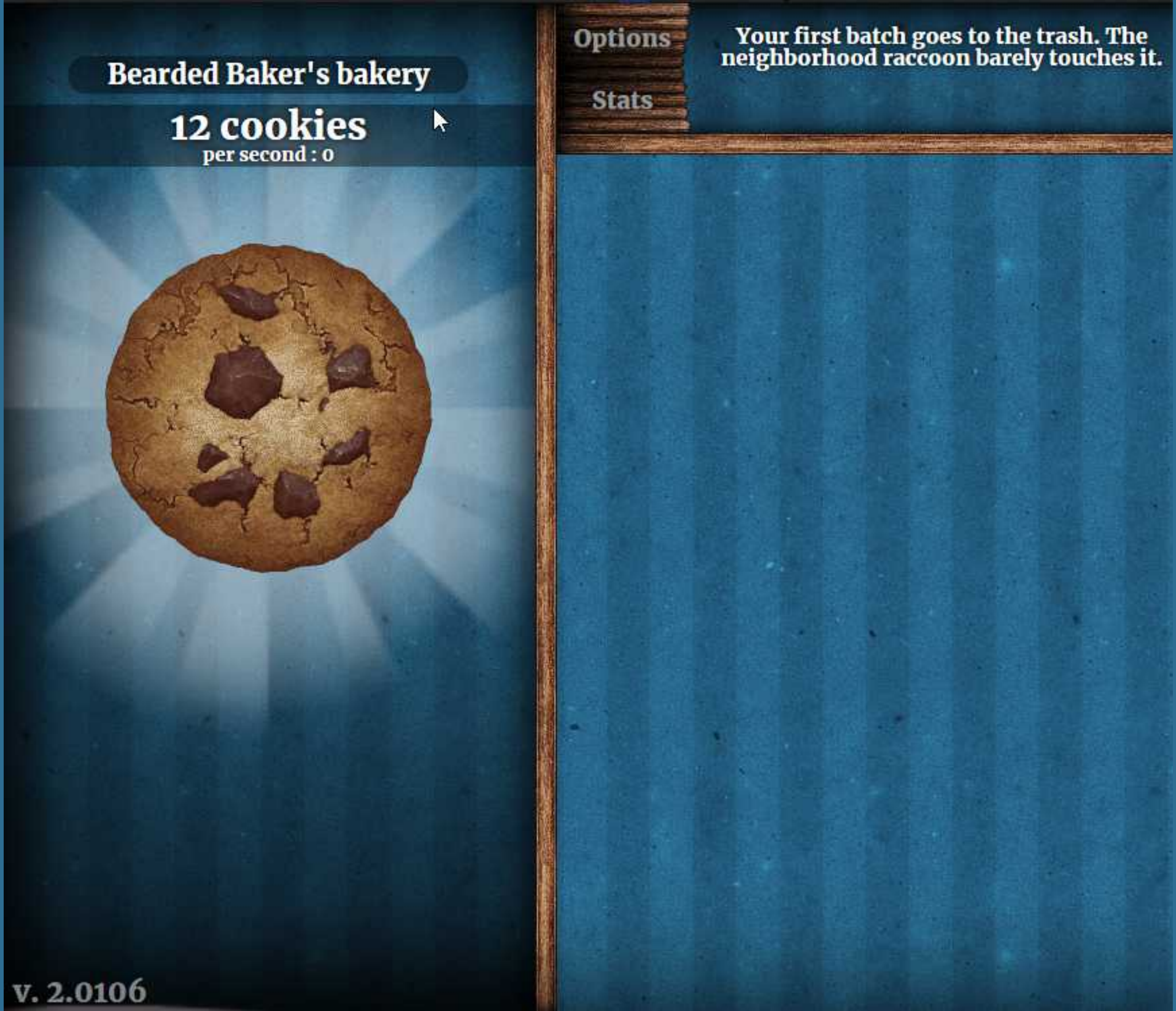
THE BIG QUESTION

Can we use *GraphQL* for *Performance-Critical Apps?*

Let's find out!

Use Case 🌟

Create a GraphQL API for *Crusty Clicker*



Bearded Baker's bakery

12 cookies
per second : 0

Your first batch goes to the trash. The neighborhood raccoon barely touches it.

v. 2.0106

DISCLAIMER

I'm a beginner at Rust.

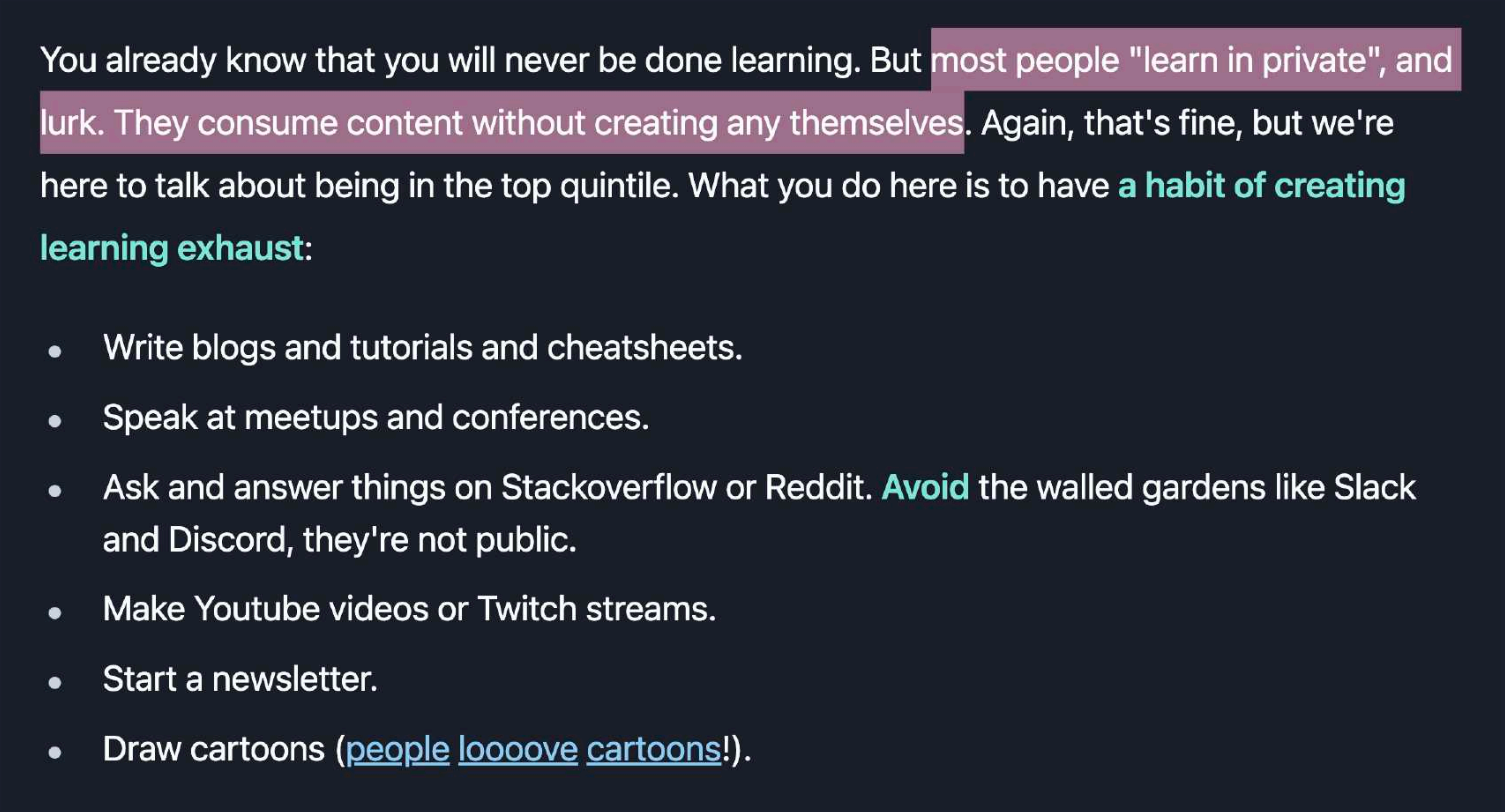
Learn In Public

The fastest way to learn

The fastest way to learn, grow your career, and build your network.

Advice LearnInPublic `Posted: Jun 19 2018`

swyx.io/learn-in-public



You already know that you will never be done learning. But most people "learn in private", and lurk. They consume content without creating any themselves. Again, that's fine, but we're here to talk about being in the top quintile. What you do here is to have **a habit of creating learning exhaust**:

- Write blogs and tutorials and cheatsheets.
- Speak at meetups and conferences.
- Ask and answer things on Stackoverflow or Reddit. **Avoid** the walled gardens like Slack and Discord, they're not public.
- Make Youtube videos or Twitch streams.
- Start a newsletter.
- Draw cartoons (people loooove cartoons!).

Our *Metrics* 𝌹

Request Duration ⏳

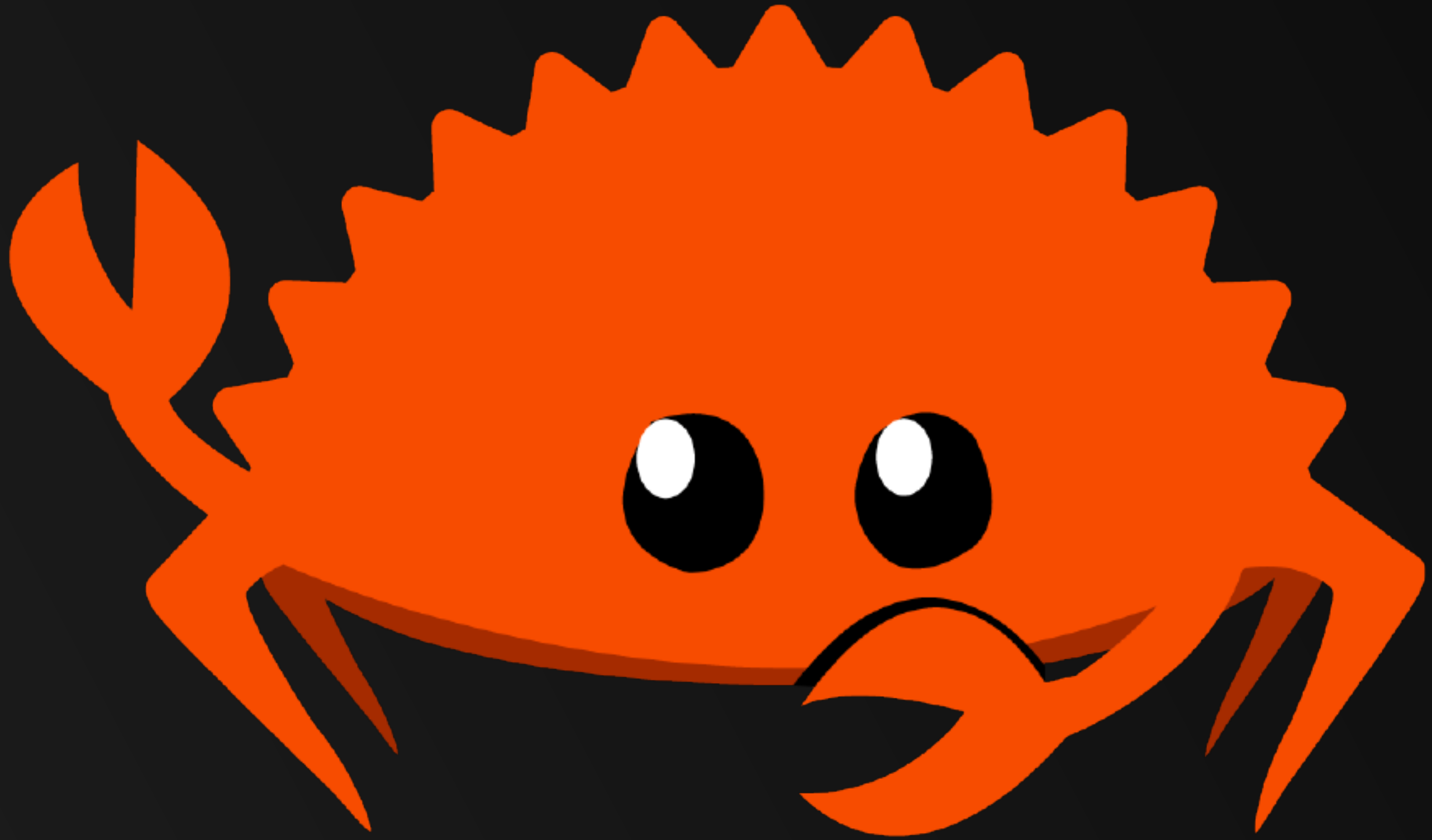
Throughput 🚗

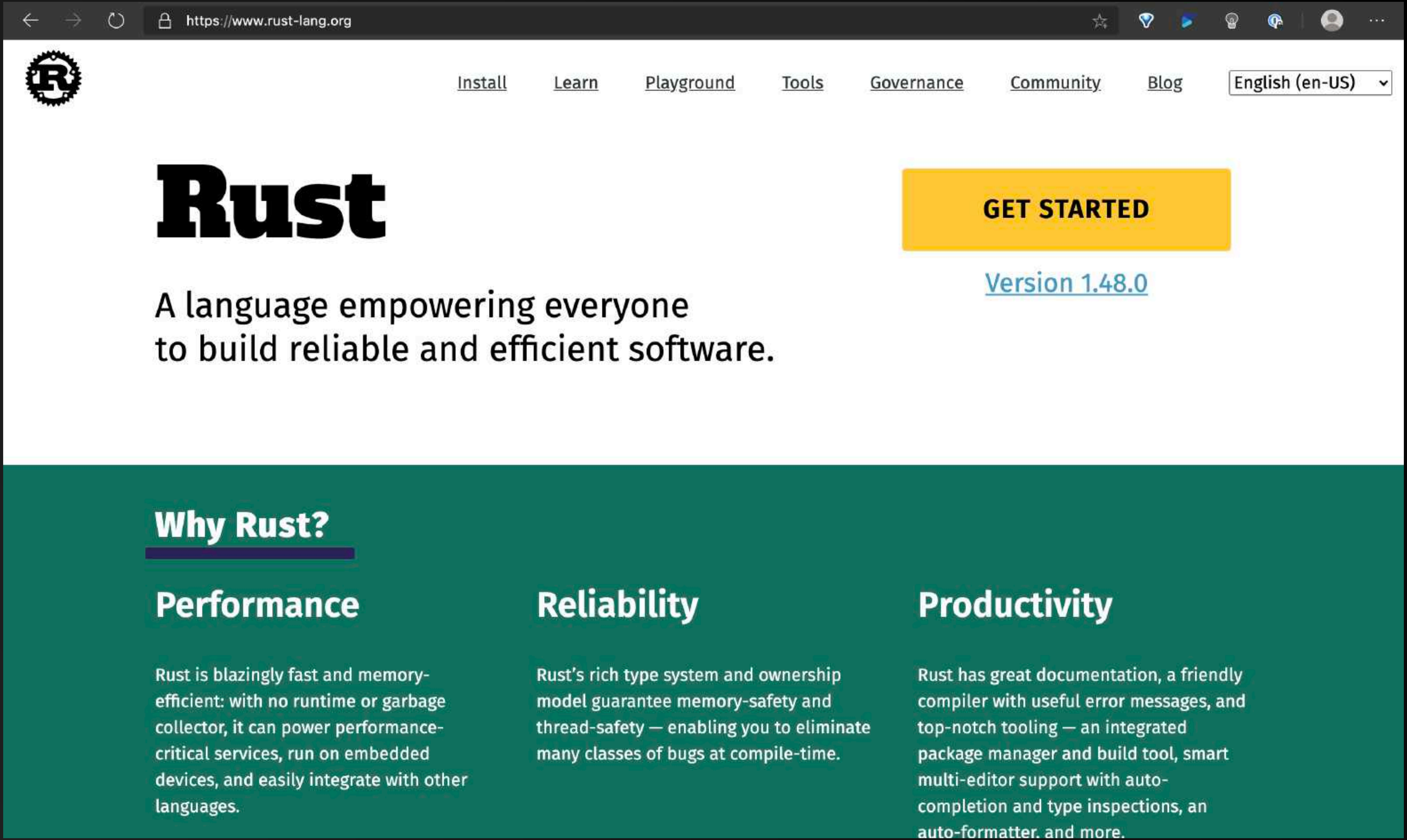
WebSockets? 🏓

I. Hello, Rust!

Why Rust is a viable option for GraphQL

What is Rust?





Rust

A language empowering everyone
to build reliable and efficient software.

Why Rust?

Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

TL;DR: *fast as C* but *modern as TS*

Why Rust?

Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

Rust’s rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

TL;DR: build stuff with performance

Build it in Rust

In 2018, the Rust community decided to improve programming experience for a few distinct domains (see the 2018 roadmap). For these, you can find many high-quality crates and some awesome guides on how to get started.

Command Line

Whip up a CLI tool quickly with Rust's robust ecosystem. Rust helps you maintain your app with confidence and distribute it with ease.

WebAssembly

Use Rust to supercharge your JavaScript, one module at a time. Publish to npm, bundle with webpack, and you're off to the races.

Networking

Predictable performance. Tiny resource footprint. Rock-solid reliability. Rust is great for network services.

Embedded

Targeting low-resource devices? Need low-level control without giving up high-level conveniences? Rust has you covered.

TL;DR: used in production by cool people

Rust in production

Hundreds of companies around the world are using Rust in production today for fast, low-resource, cross-platform solutions. Software you know and love, like Firefox, Dropbox, and Cloudflare, uses Rust. **From startups to large corporations, from embedded devices to scalable web services, Rust is a great fit.**

> “ My biggest compliment to Rust is that it's boring, and this is an amazing compliment.

– Chris Dickinson, Engineer at npm, Inc

II.

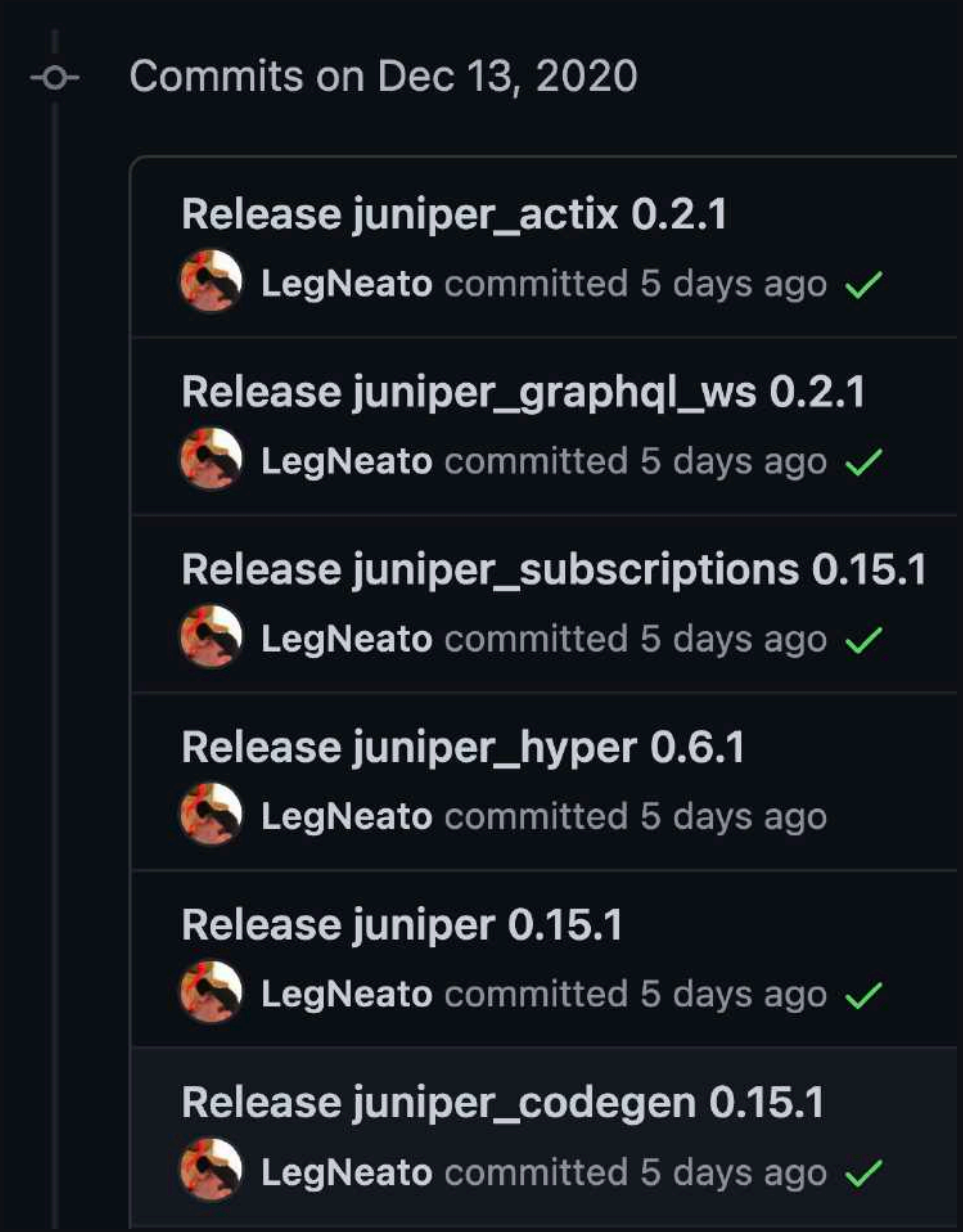
Hey, Juniper!

Introduction to GraphQL on Rust

Juniper

> GraphQL server library for Rust

Juniper 0.15.1 is awesome!



Commits on Dec 13, 2020

Release juniper_actix 0.2.1
LegNeato committed 5 days ago ✓

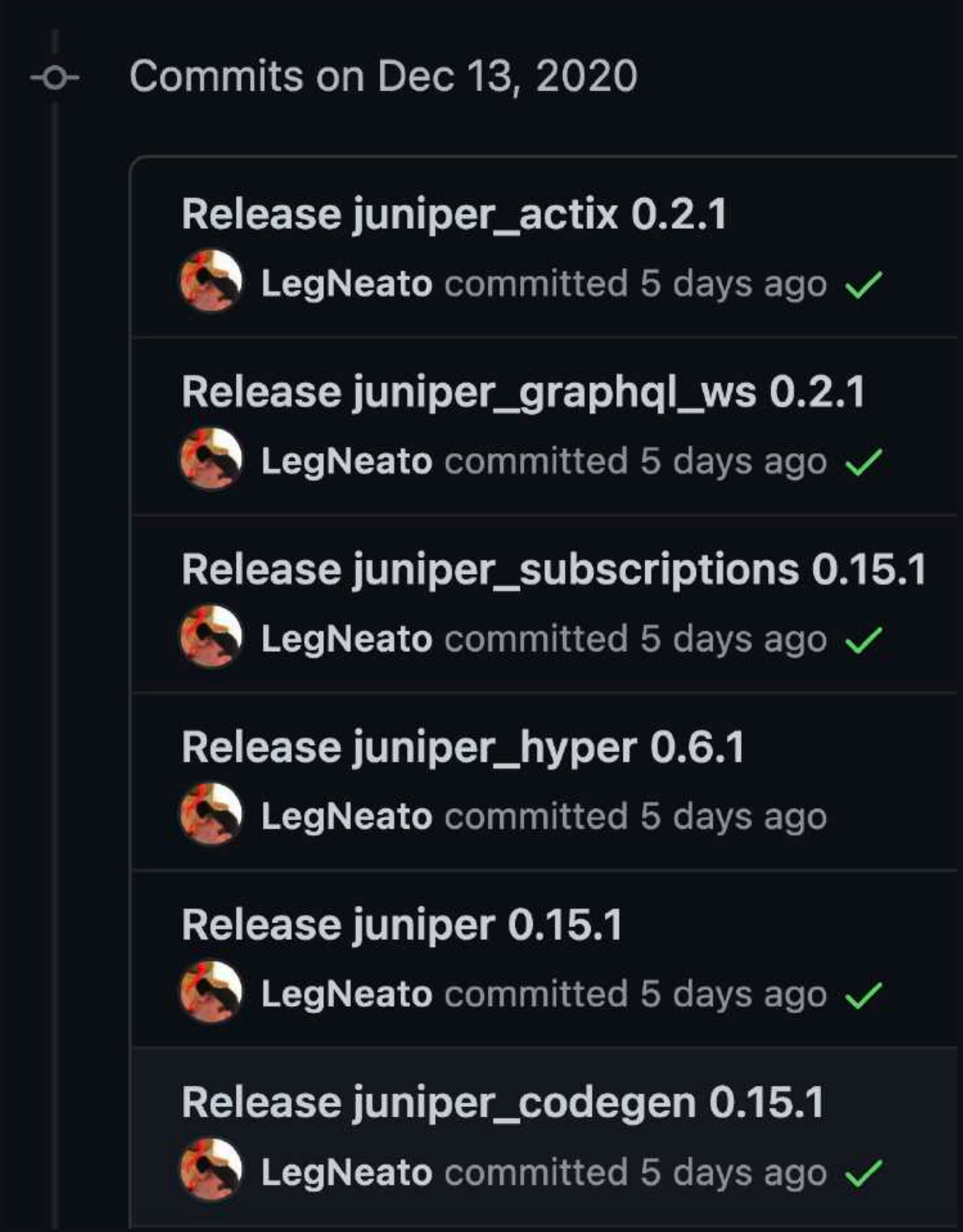
Release juniper_graphql_ws 0.2.1
LegNeato committed 5 days ago ✓

Release juniper_subscriptions 0.15.1
LegNeato committed 5 days ago ✓

Release juniper_hyper 0.6.1
LegNeato committed 5 days ago

Release juniper 0.15.1
LegNeato committed 5 days ago ✓

Release juniper_codegen 0.15.1
LegNeato committed 5 days ago ✓



Commits on Dec 13, 2020

Release juniper_actix 0.2.1
LegNeato committed 5 days ago ✓

Release juniper_graphql_ws 0.2.1
LegNeato committed 5 days ago ✓

Release juniper_subscriptions 0.15.1
LegNeato committed 5 days ago ✓

Release juniper_hyper 0.6.1
LegNeato committed 5 days ago

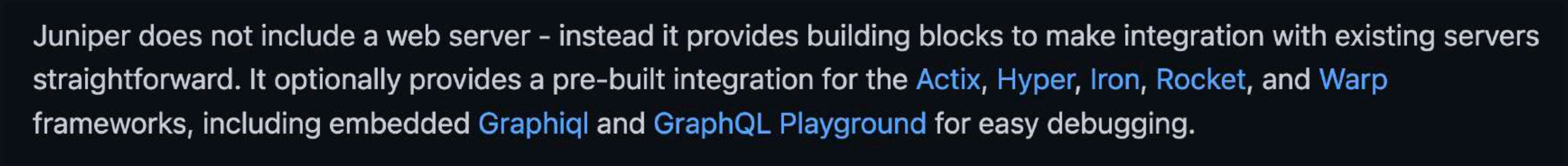
Release juniper 0.15.1
LegNeato committed 5 days ago ✓

Release juniper_codegen 0.15.1
LegNeato committed 5 days ago ✓

Asynchronous
Non-Blocking IO

Built-in
WebSocket
Subscriptions

Choose your own *web server*



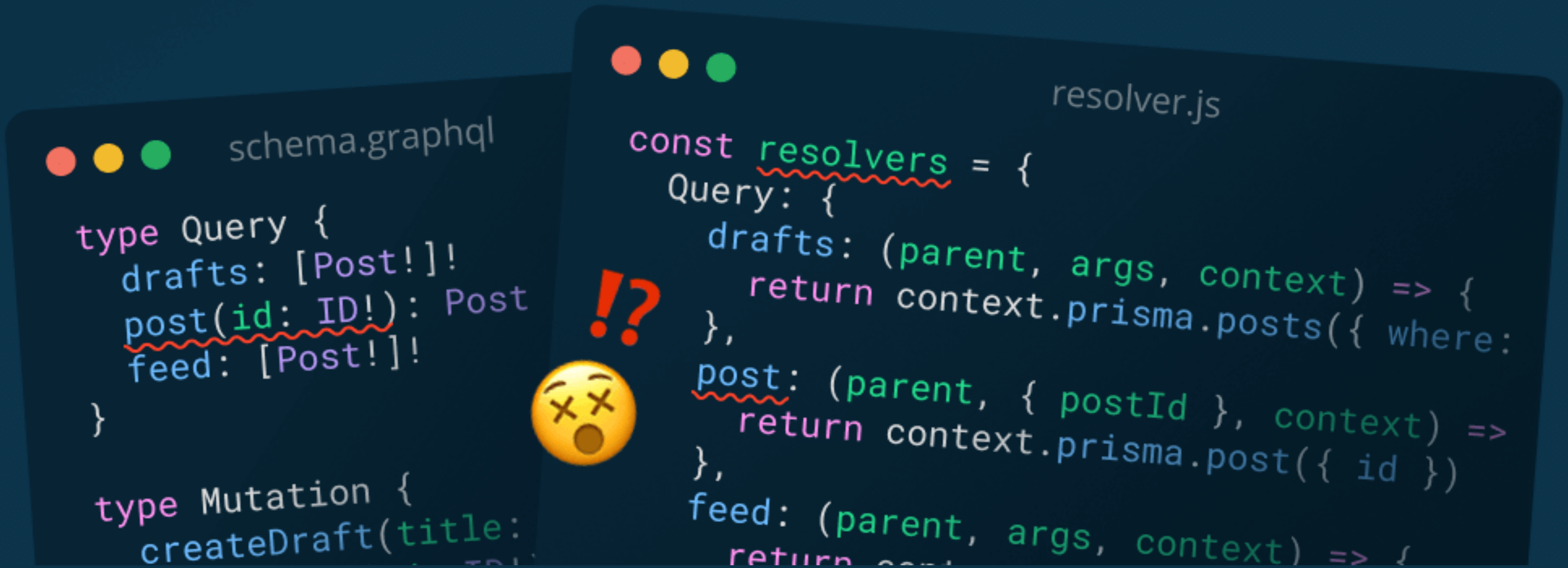
Juniper does not include a web server - instead it provides building blocks to make integration with existing servers straightforward. It optionally provides a pre-built integration for the Actix, Hyper, Iron, Rocket, and Warp frameworks, including embedded Graphiql and GraphQL Playground for easy debugging.

Let's begin!

```
~/Projects/graphql-in-3ms  master
```

III. Code-First vs Schema-First

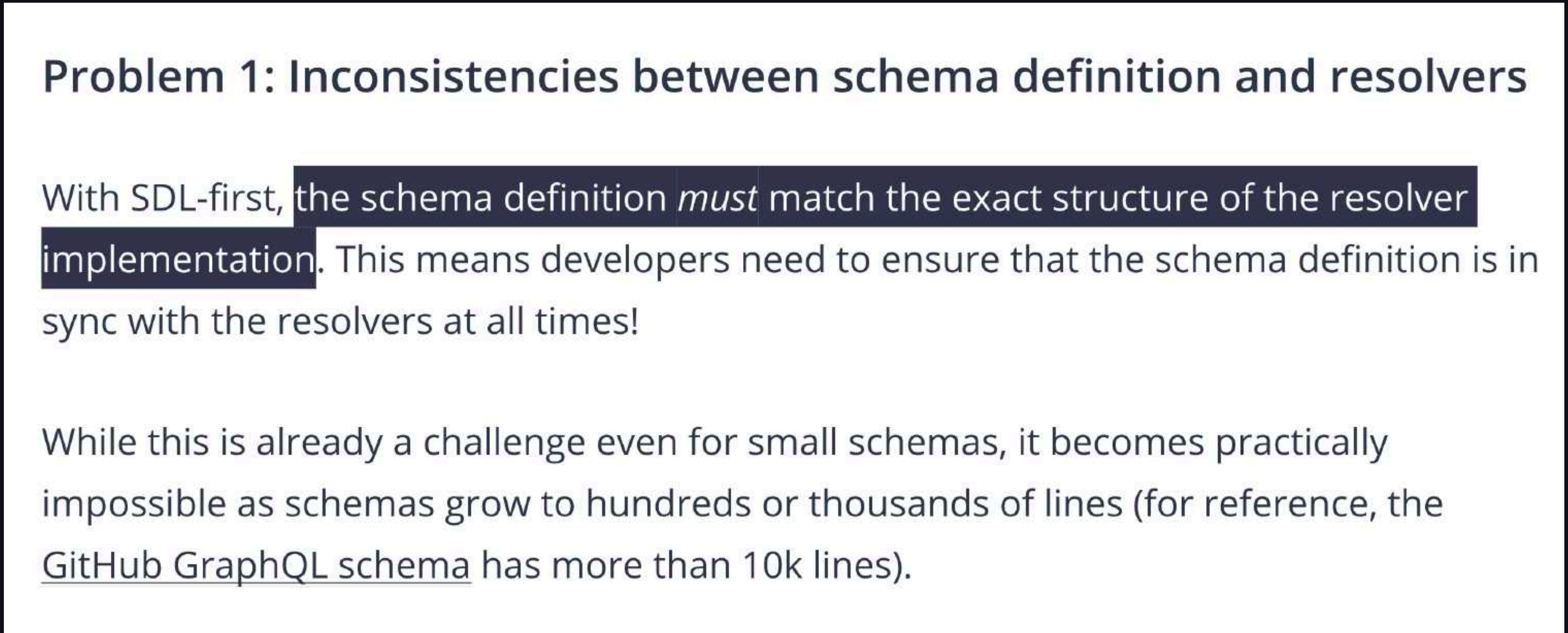
The Problems with ~~Schema~~ SDL-First GraphQL Server Development



```
type Query {
  drafts: [Post!]!
  post(id: ID!): Post
  feed: [Post!]!
}

type Mutation {
  createDraft(title:
```

```
const resolvers = {
  Query: {
    drafts: (parent, args, context) => {
      return context.prisma.posts({ where:
    },
    post: (parent, { postId }, context) =>
      return context.prisma.post({ id })
    },
    feed: (parent, args, context) => {
      return con
```

https://www.prisma.io/blog/the-problems-of-schema-first-graphql-development-x1mn4cb0tyl3

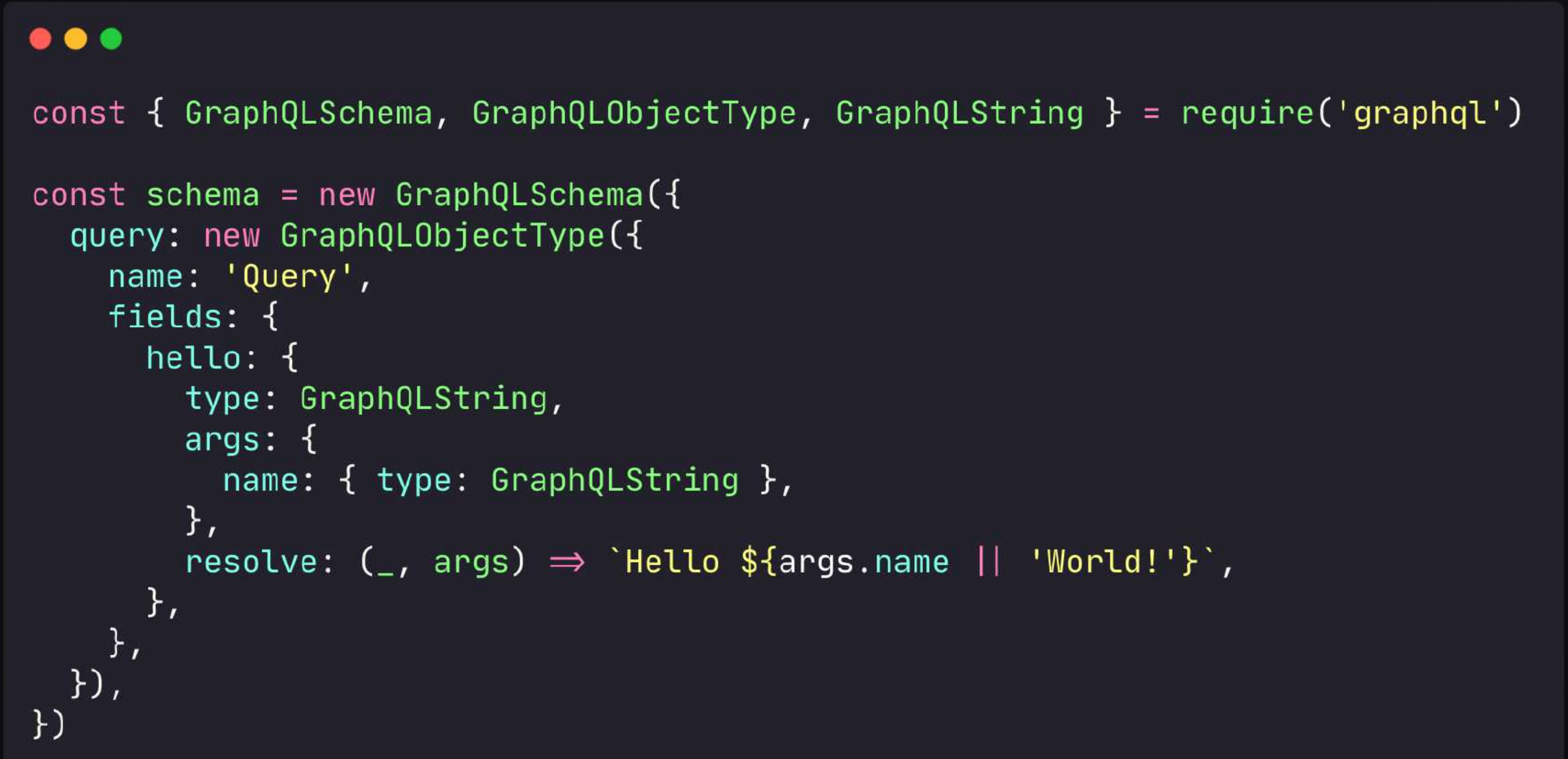


Problem 1: Inconsistencies between schema definition and resolvers

With SDL-first, the schema definition *must* match the exact structure of the resolver implementation. This means developers need to ensure that the schema definition is in sync with the resolvers at all times!

While this is already a challenge even for small schemas, it becomes practically impossible as schemas grow to hundreds or thousands of lines (for reference, the GitHub GraphQL schema has more than 10k lines).

Code-First in JS can be *very verbose* 😢



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

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        args: {
          name: { type: GraphQLString },
        },
        resolve: (_, args) => `Hello ${args.name || 'World!'}`,
      },
    },
  }),
})
```

Surprisingly,

Not with **Rust!**

IV.

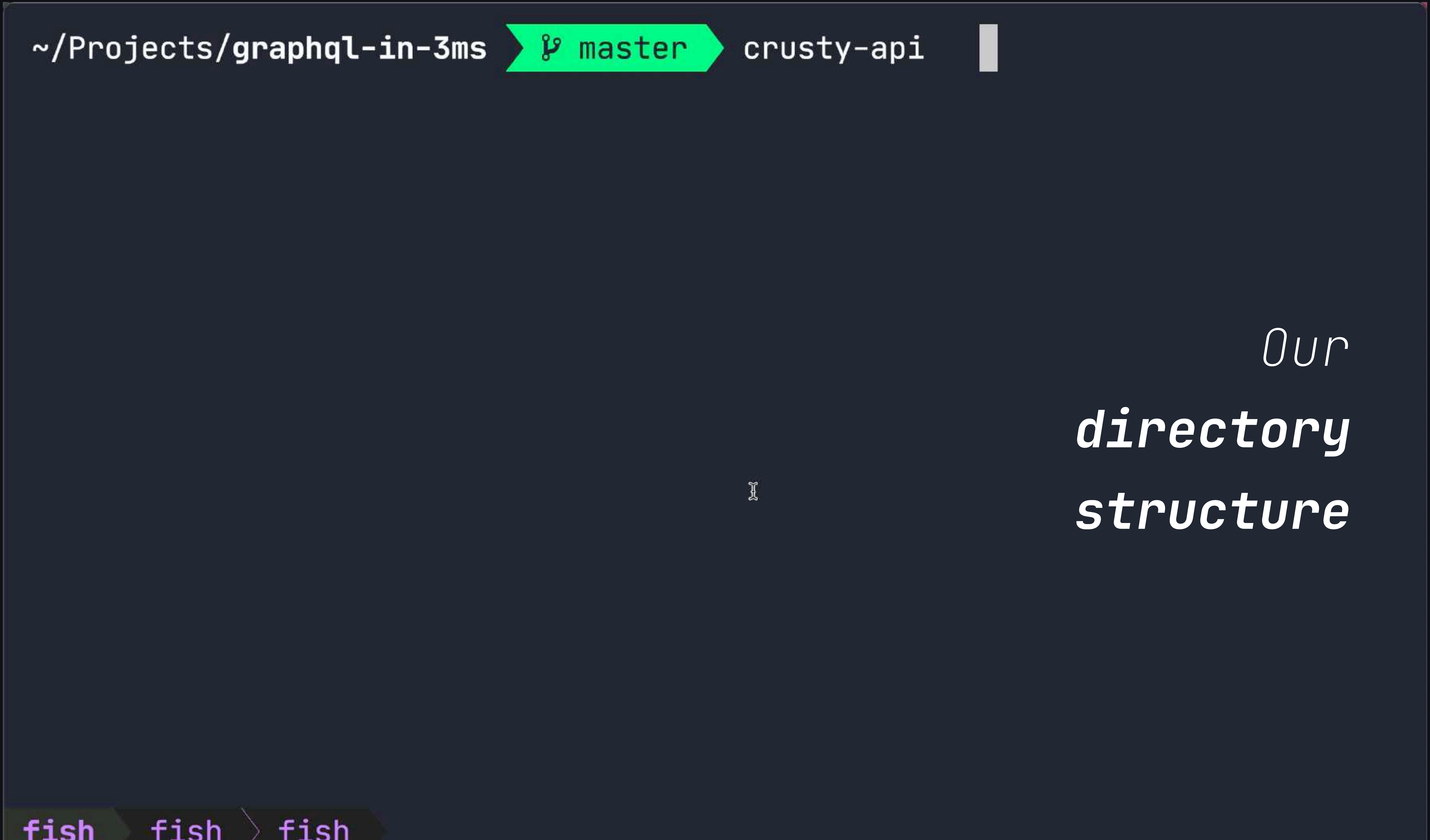
Building a **GraphQL API** with **Rust**

3 Steps for Code-First GraphQL

TODO *Data Model*

TODO *Query Resolver*

TODO *Root Schema*



```
~/Projects/graphql-in-3ms  master  crusty-api
```

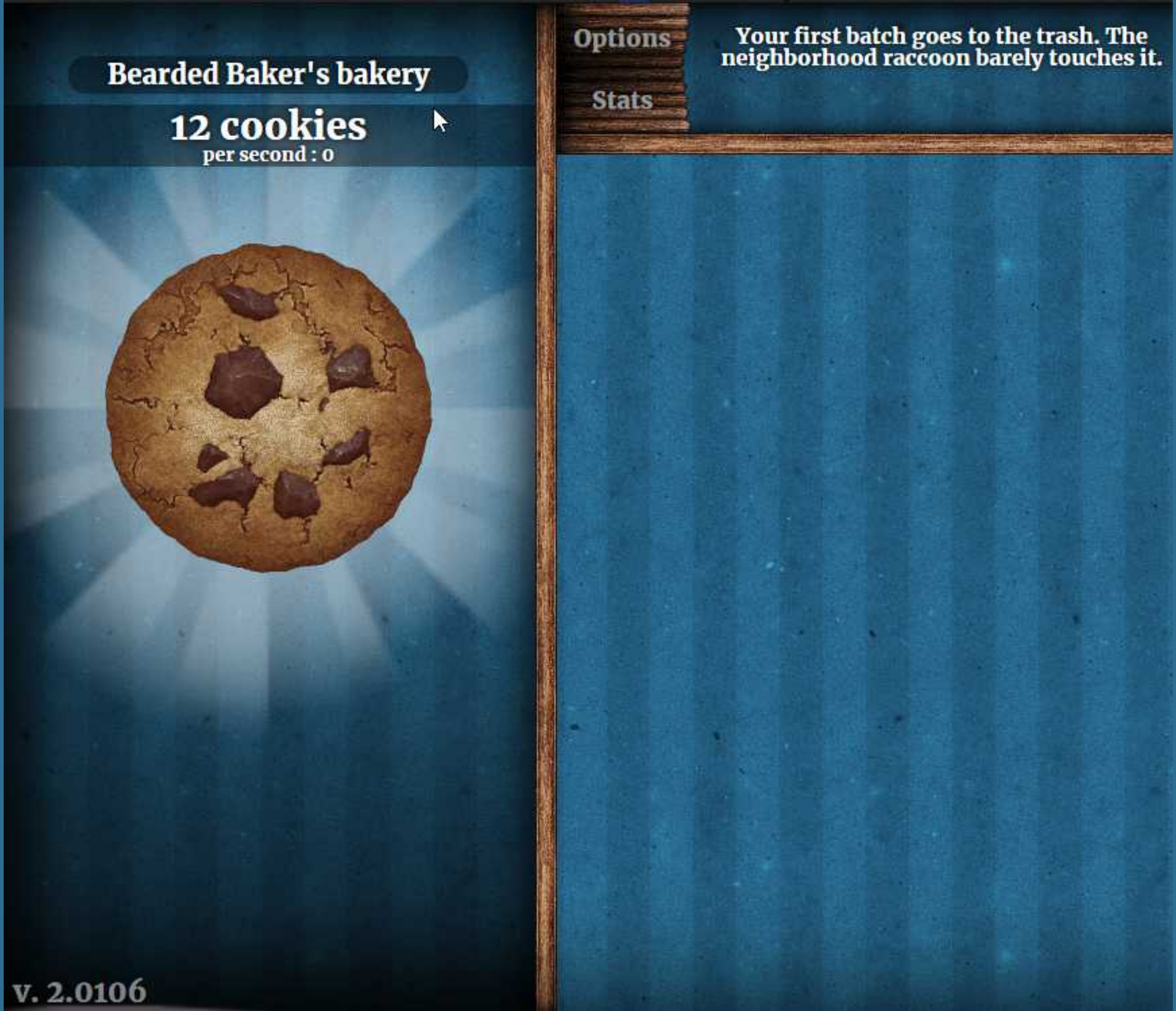
Our

directory structure

TODO Data Model

TODO Query Resolver

TODO Root Schema

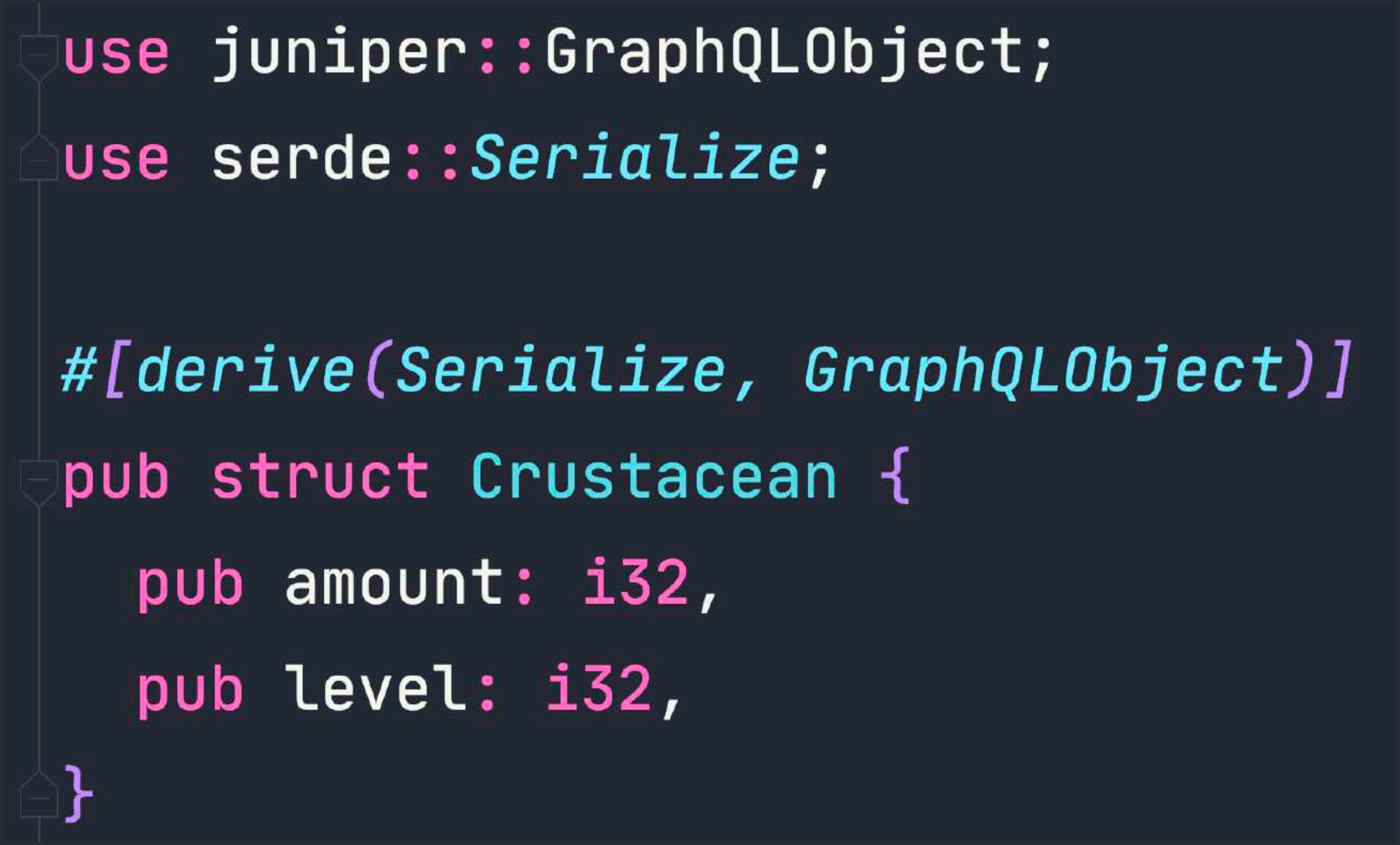


Bearded Baker's bakery

12 cookies
per second : 0

Your first batch goes to the trash. The neighborhood raccoon barely touches it.

Let's define a data model.



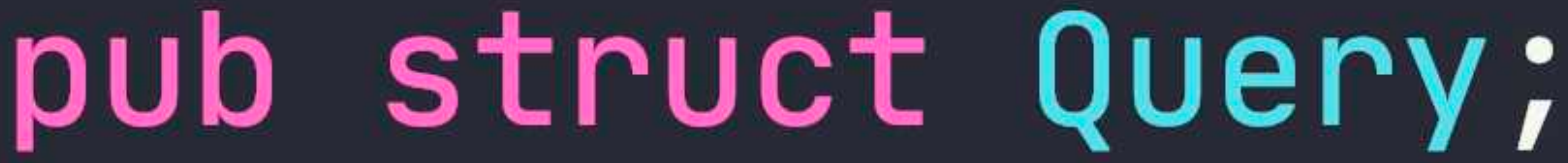
```rust
use juniper::GraphQLObject;
use serde::Serialize;

#[derive(Serialize, GraphQLObject)]
pub struct Crustacean {
  pub amount: i32,
  pub level: i32,
}
```

src/models/game.rs

DONE ~~Data Model~~

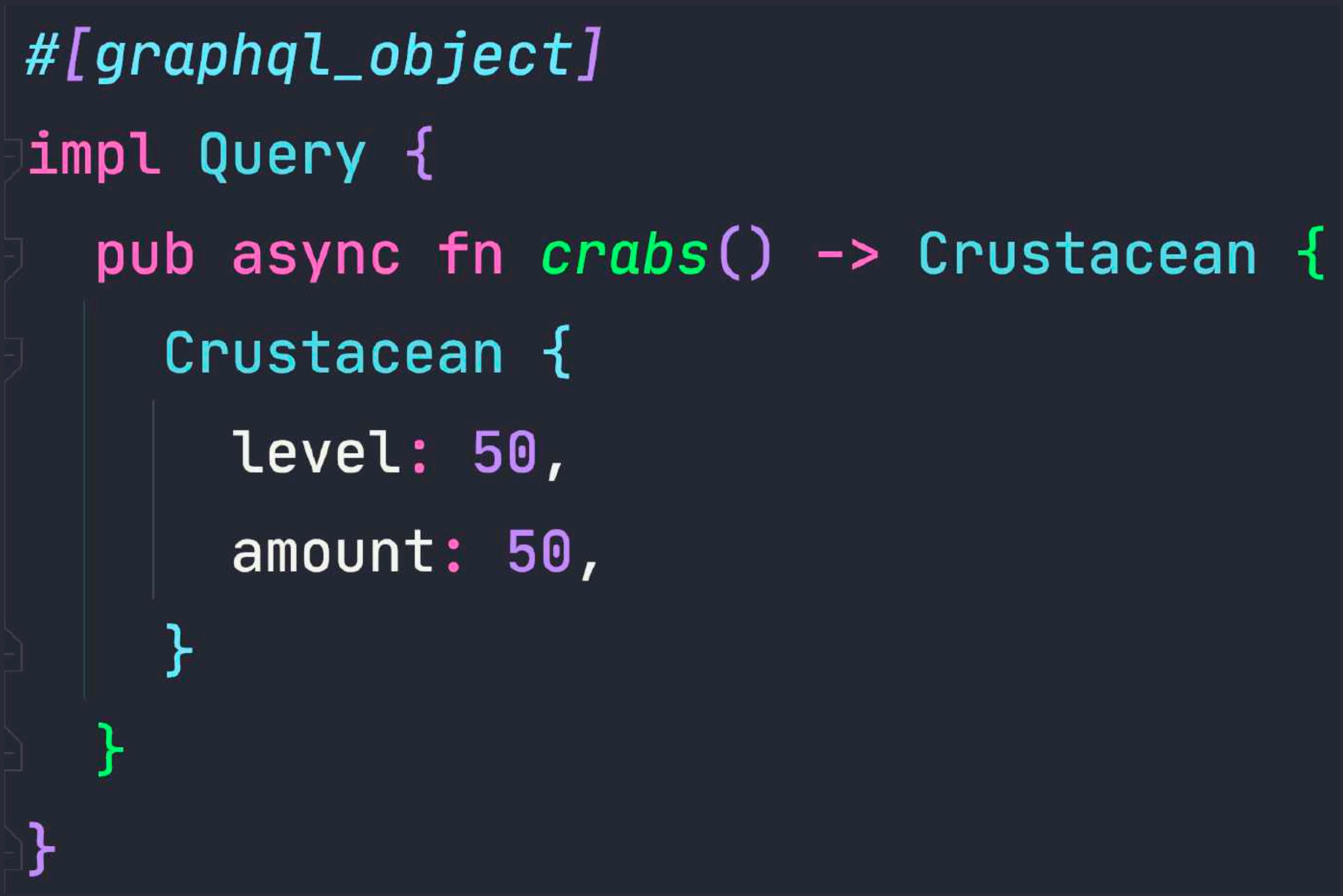
TODO **Query Resolver**

TODO Root Schema

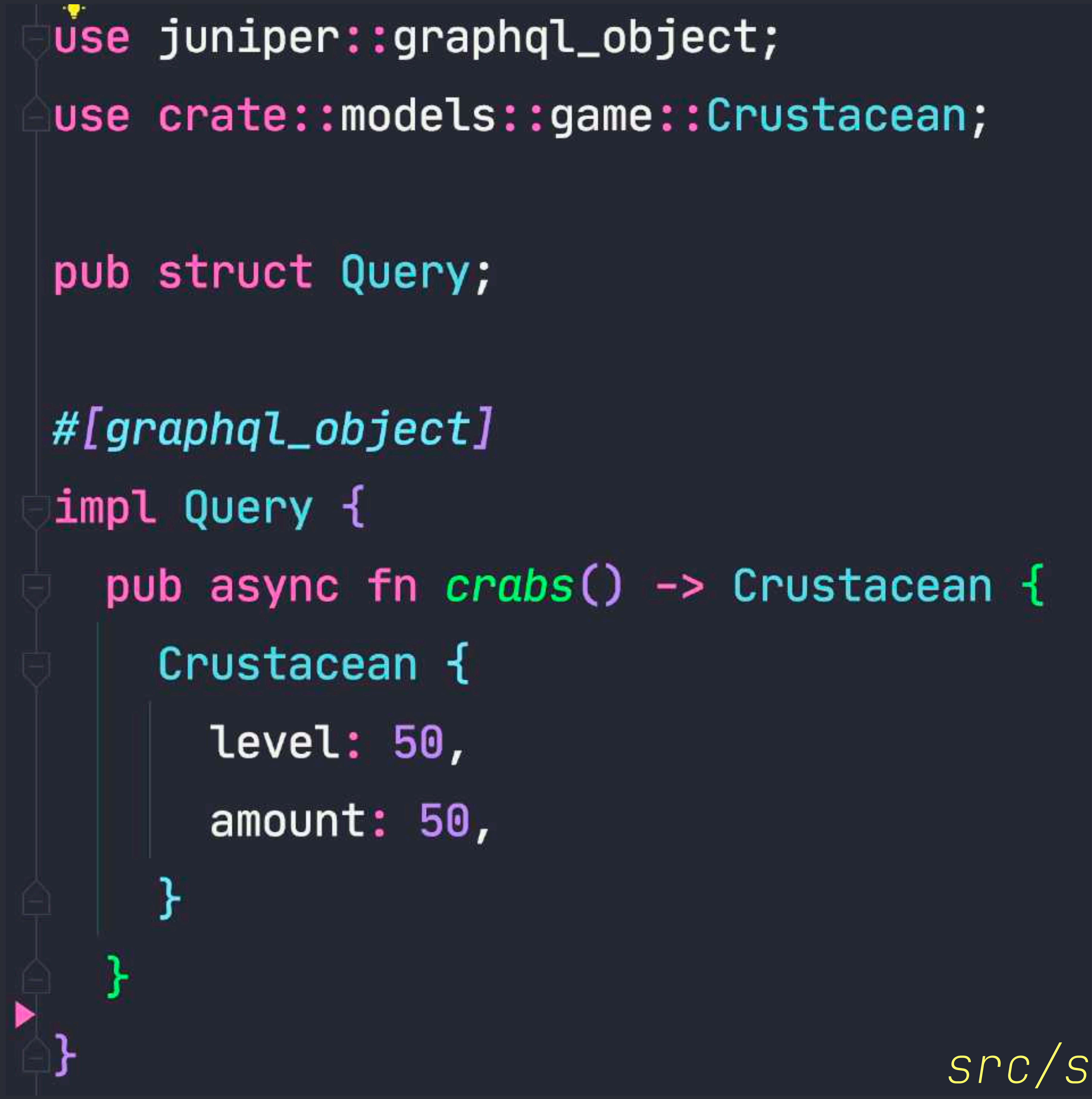


```rust
pub struct Query;
```

src/services/query.rs

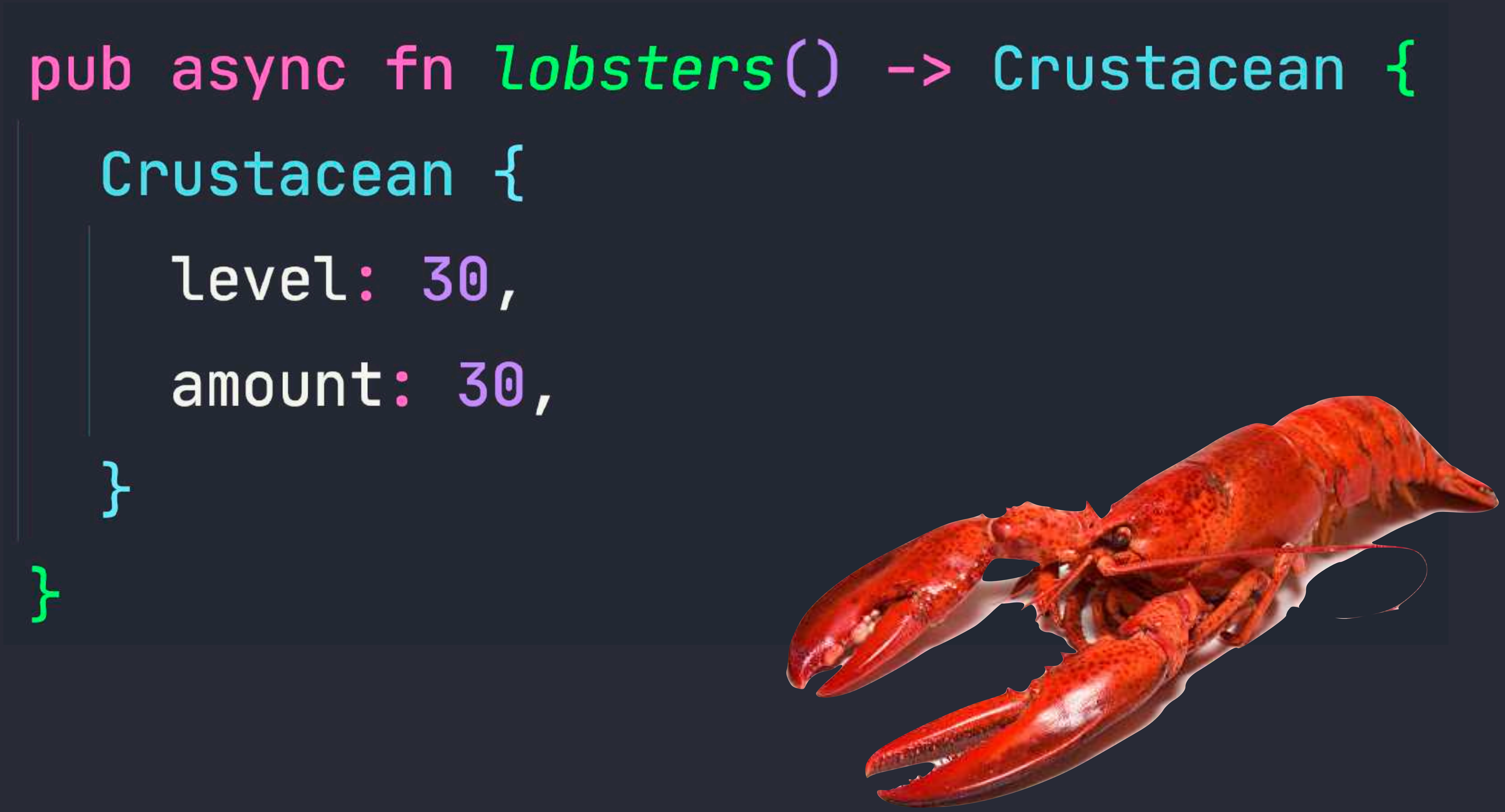


```rust
#[graphql_object]
impl Query {
  pub async fn crabs() -> Crustacean {
    Crustacean {
      level: 50,
      amount: 50,
    }
  }
}
```



```rust
use juniper::graphql_object;
use crate::models::game::Crustacean;

pub struct Query;

#[graphql_object]
impl Query {
  pub async fn crabs() -> Crustacean {
    Crustacean {
      level: 50,
      amount: 50,
    }
  }
}
```

src/services/query.rs

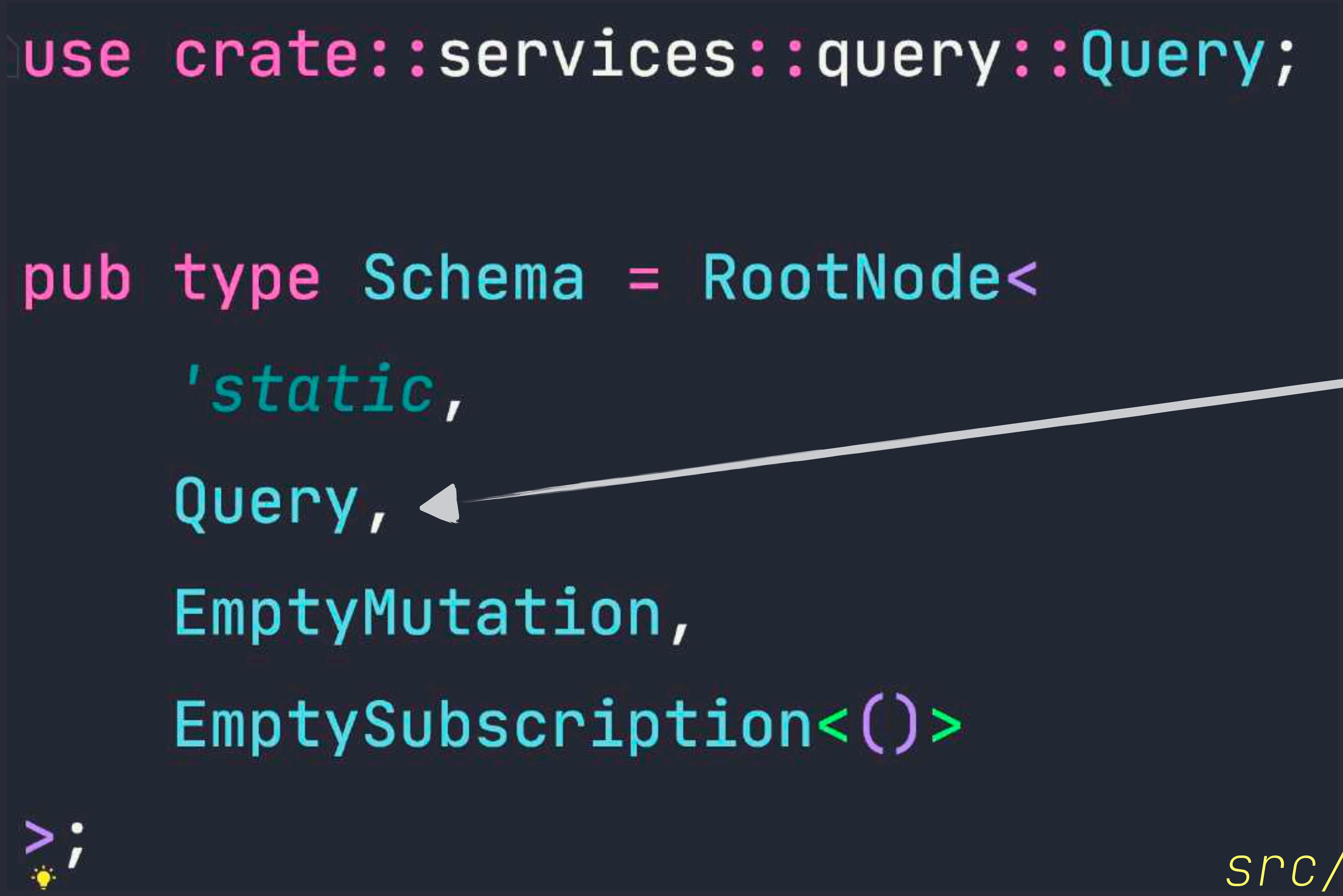


```rust
pub async fn lobsters() -> Crustacean {
    Crustacean {
        level: 30,
        amount: 30,
    }
}
```

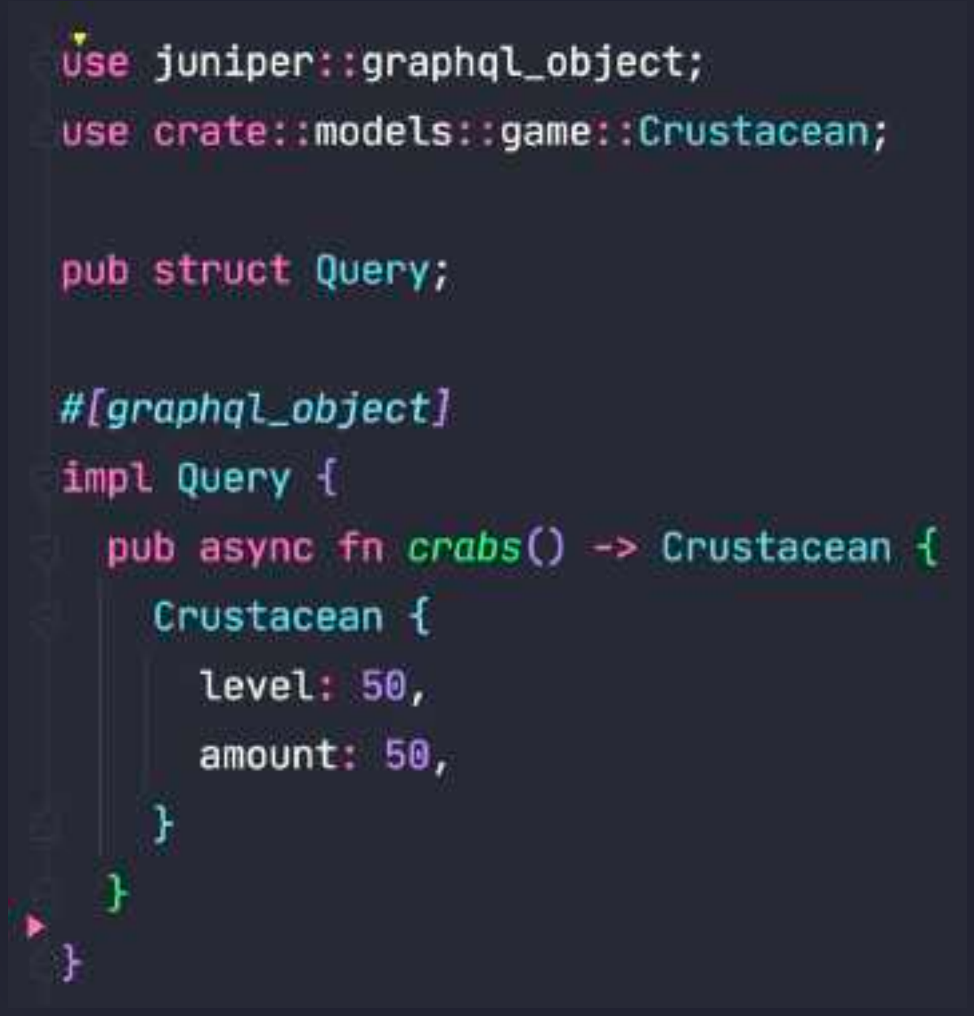
DONE ~~Data Model~~

DONE ~~Query Resolver~~

TODO Root Schema

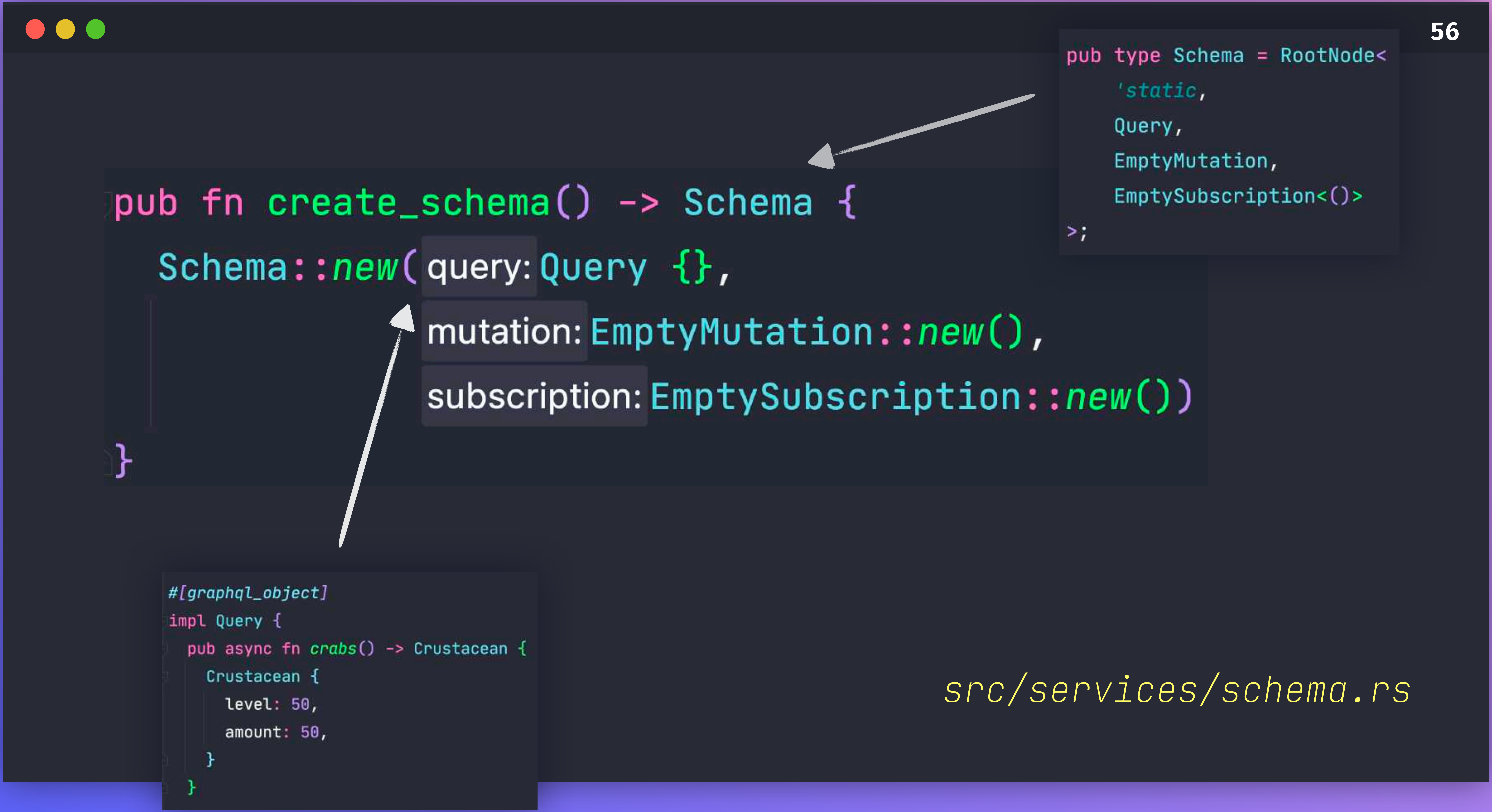


```rust
use crate::services::query::Query;

pub type Schema = RootNode<
    'static,
    Query,
    EmptyMutation,
    EmptySubscription<()>
>;
```



```rust
use juniper::graphql_object;
use crate::models::game::Crustacean;

pub struct Query;

#[graphql_object]
impl Query {
  pub async fn crabs() -> Crustacean {
    Crustacean {
      level: 50,
      amount: 50,
    }
  }
}
```

src/services/schema.rs

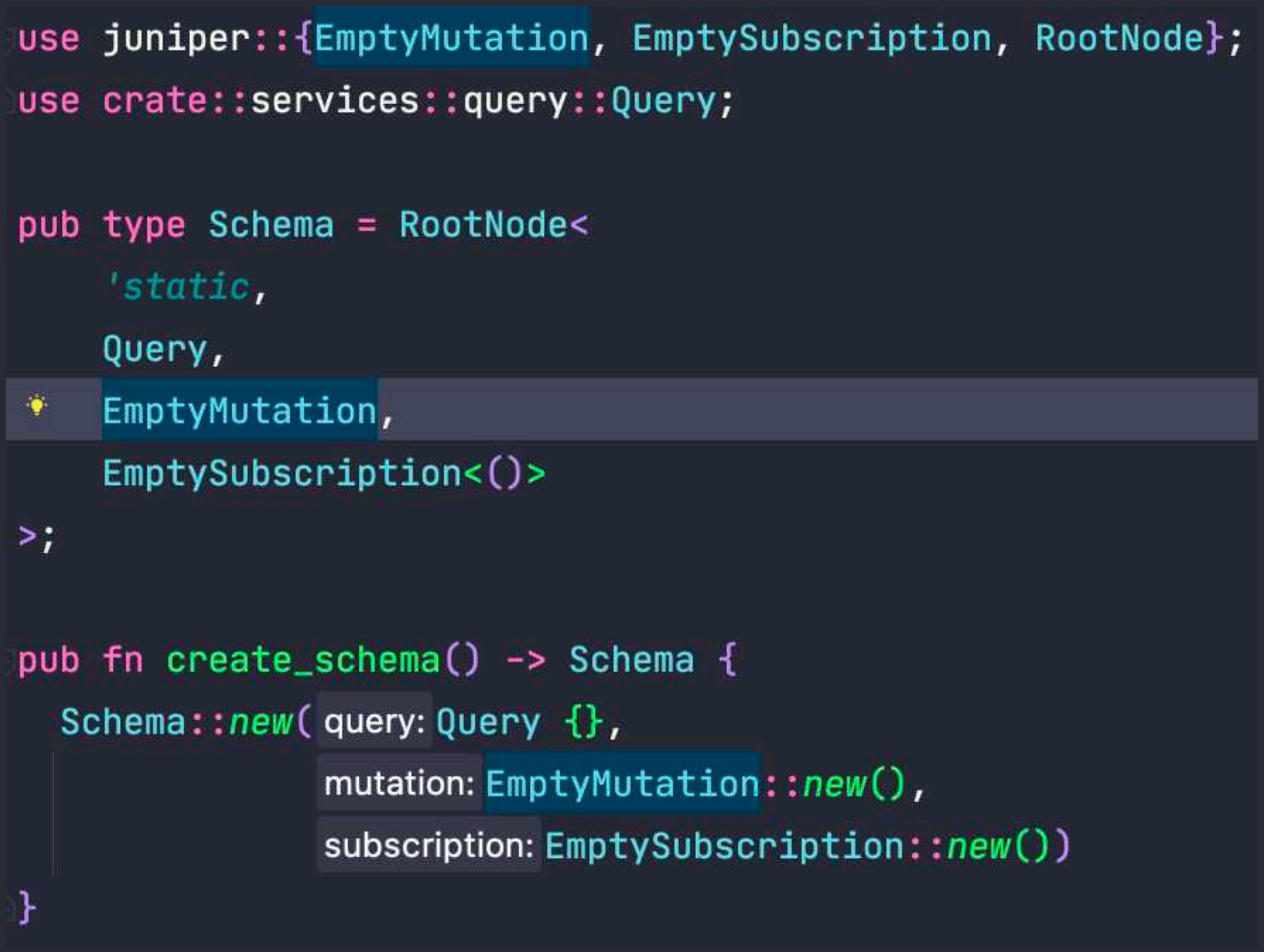


```
pub fn create_schema() -> Schema {
    Schema::new( query: Query {},
                 mutation: EmptyMutation::new(),
                 subscription: EmptySubscription::new())
}
```

```
pub type Schema = RootNode<
    'static,
    Query,
    EmptyMutation,
    EmptySubscription<()>
>;
```

```
#[graphql_object]
impl Query {
  pub async fn crabs() -> Crustacean {
    Crustacean {
      level: 50,
      amount: 50,
    }
  }
```

src/services/schema.rs



```rust
use juniper::{EmptyMutation, EmptySubscription, RootNode};
use crate::services::query::Query;

pub type Schema = RootNode<
    'static,
    Query,
    EmptyMutation,
    EmptySubscription<()>
>;

pub fn create_schema() -> Schema {
  Schema::new(query: Query {},
              mutation: EmptyMutation::new(),
              subscription: EmptySubscription::new())
}
```

src/services/schema.rs

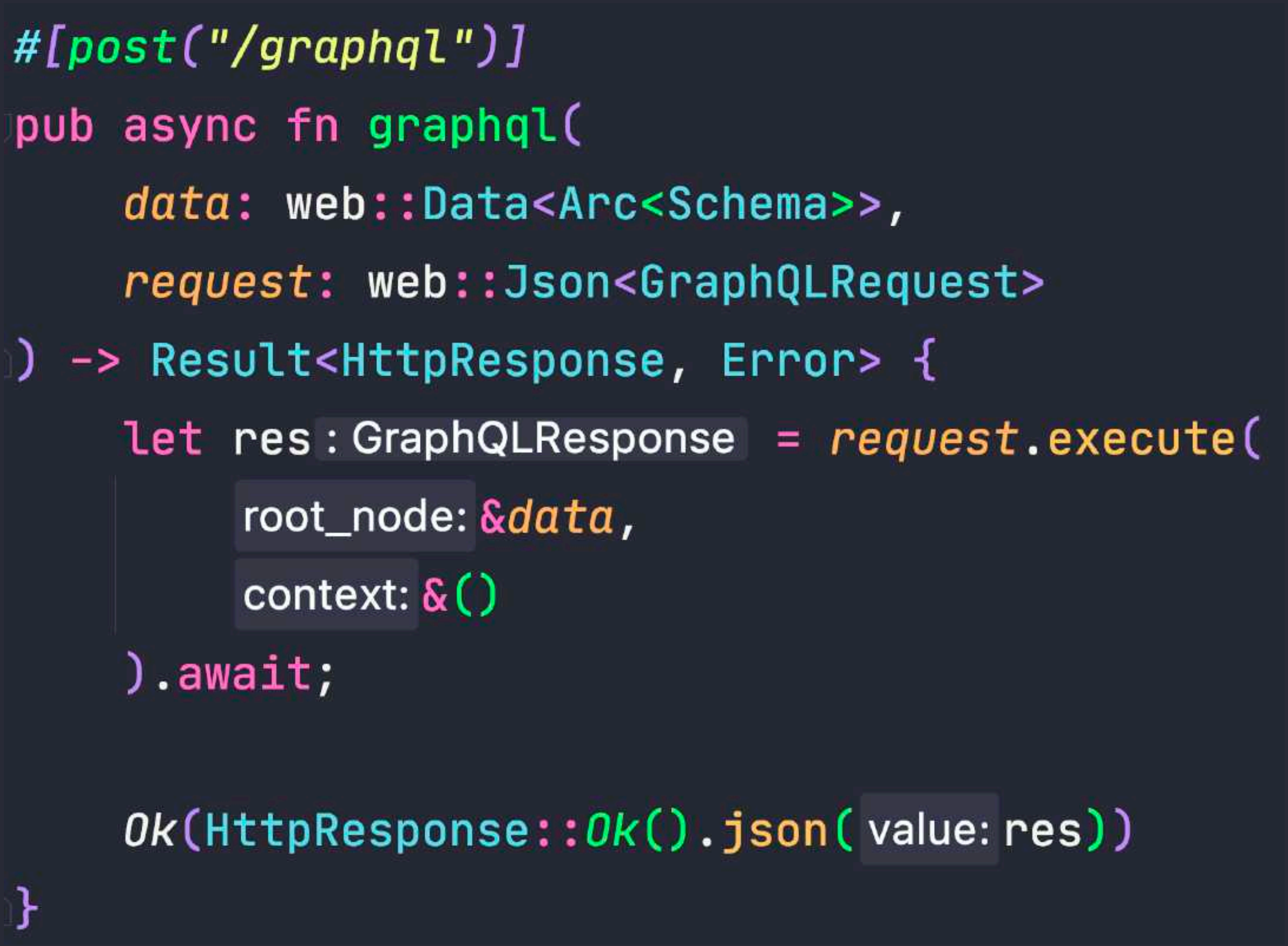
~~DONE Data Model~~

~~DONE Query Resolver~~

~~DONE Root Schema~~

TODO Add GQL route

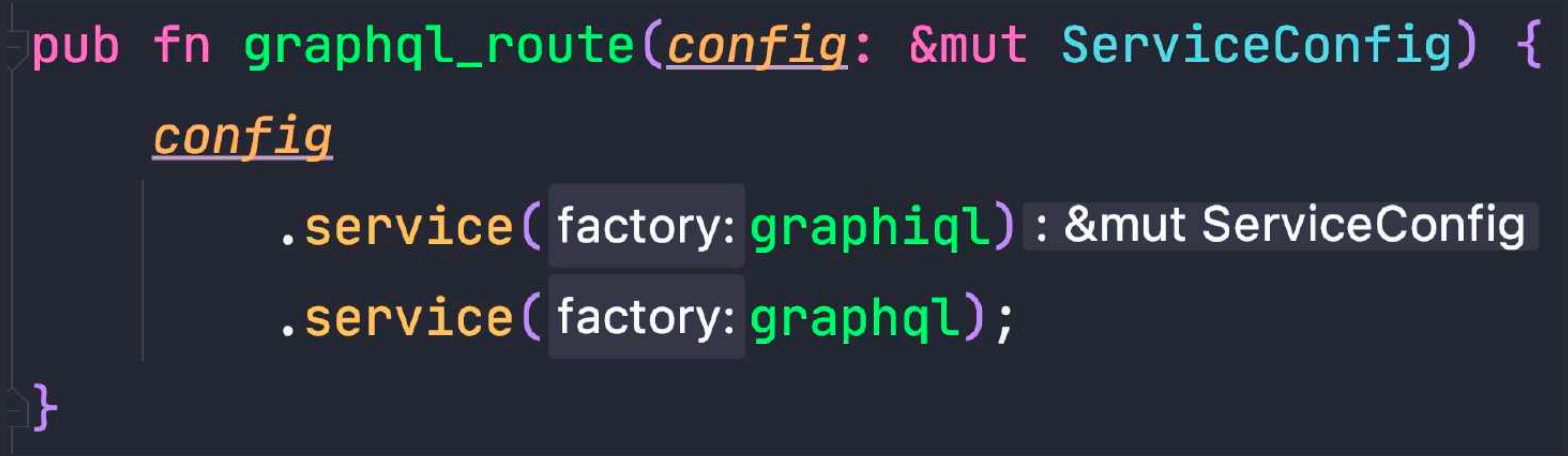
TODO *Setup Actix*



```
#[post("/graphql")]
pub async fn graphql(
    data: web::Data<Arc<Schema>>,
    request: web::Json<GraphQLRequest>
) -> Result<HttpResponse, Error> {
    let res : GraphQLResponse = request.execute(
        root_node: &data,
        context: &()
    ).await;

    Ok(HttpResponse::Ok().json( value: res))
}
```

Register a /graphql endpoint

Setup our /graphql endpoint,
as well as GraphiQL

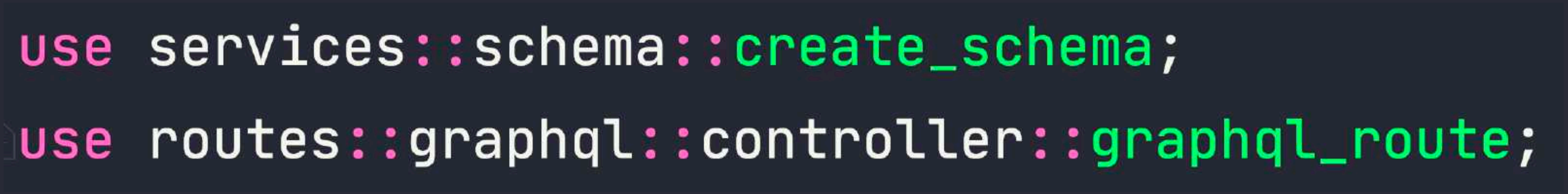


```rust
pub fn graphql_route(config: &mut ServiceConfig) {
    config
        .service(factory: graphiql) : &mut ServiceConfig
        .service(factory: graphql);
}
```

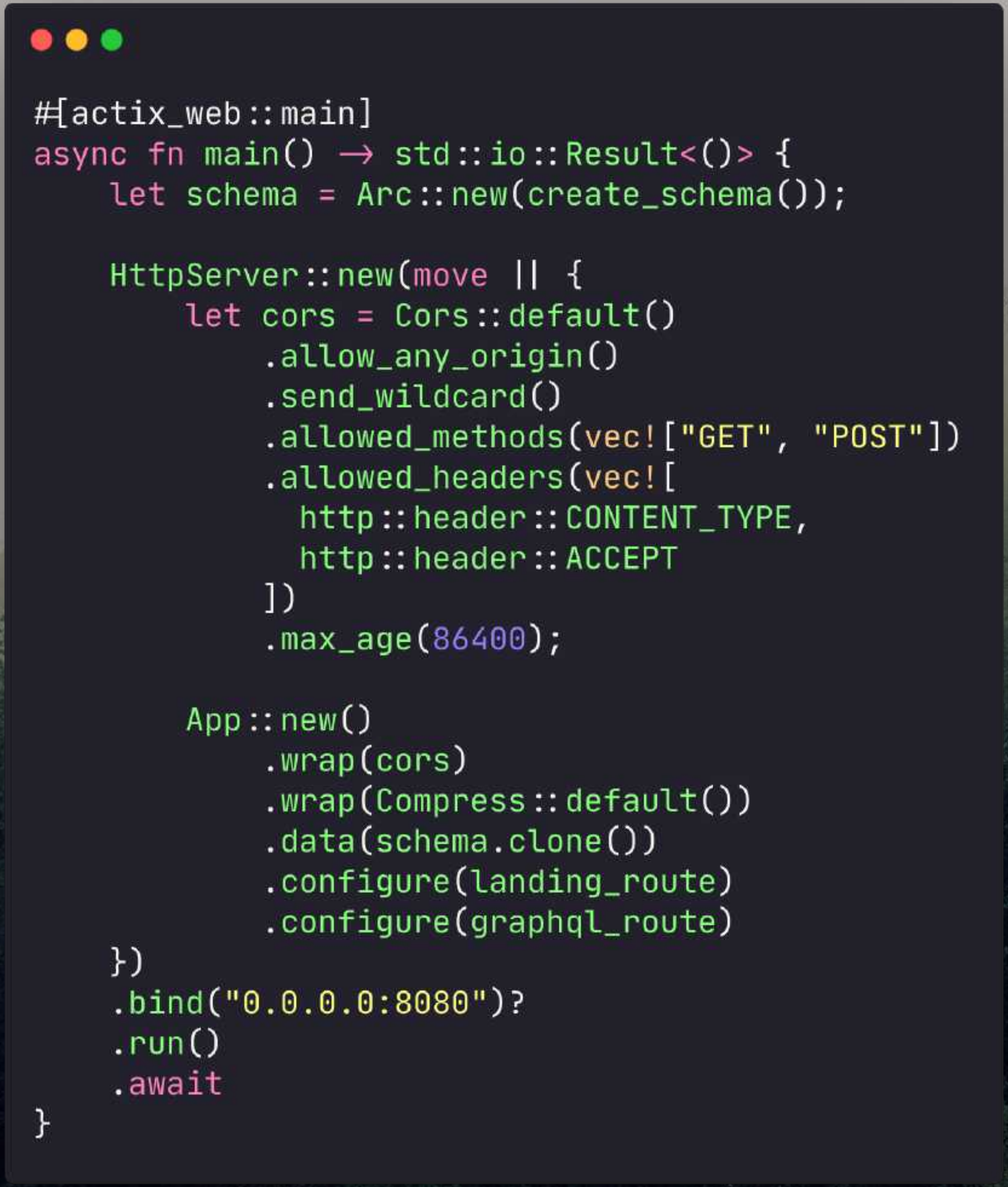
DONE ~~Add GQL route~~

TODO Setup Actix

Import our schema and endpoint



```rust
use services::schema::create_schema;
use routes::graphql::controller::graphql_route;
```



```rust
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let schema = Arc::new(create_schema());

    HttpServer::new(move || {
        let cors = Cors::default()
            .allow_any_origin()
            .send_wildcard()
            .allowed_methods(vec!["GET", "POST"])
            .allowed_headers(vec![
              http::header::CONTENT_TYPE,
              http::header::ACCEPT
            ])
            .max_age(86400);

        App::new()
            .wrap(cors)
            .wrap(Compress::default())
            .data(schema.clone())
            .configure(landing_route)
            .configure(graphql_route)
    })
    .bind("0.0.0.0:8080")?
    .run()
    .await
}
```

Create the Schema

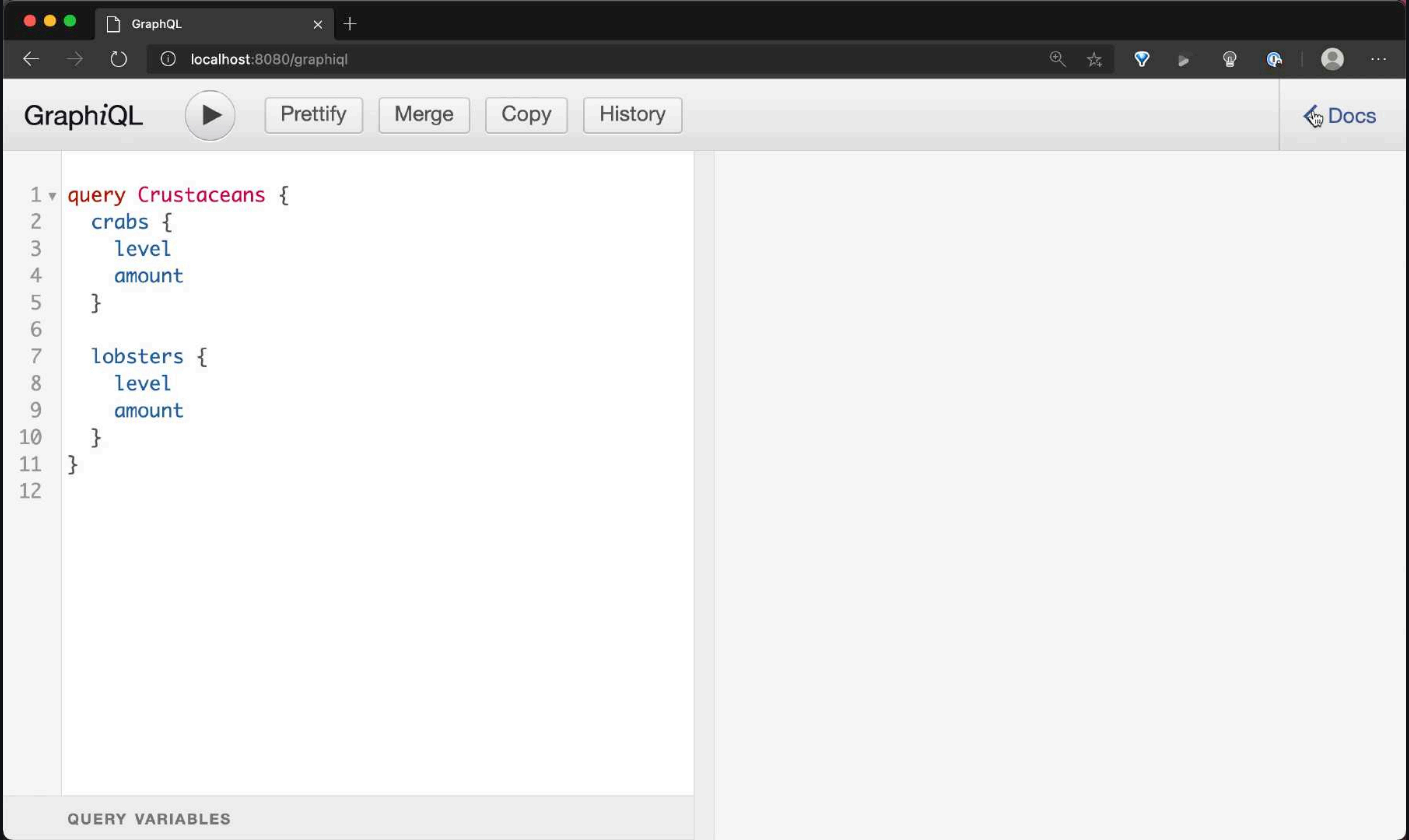
Setup CORS

Setup Compression

Create an Actix App

Configure Routes

```
~/Projects/graphql-in-3ms  master  crusty-api
```



```graphql
query Crustaceans {
  crabs {
    level
    amount
  }

  lobsters {
    level
    amount
  }
}
```

But is it fast?

V.

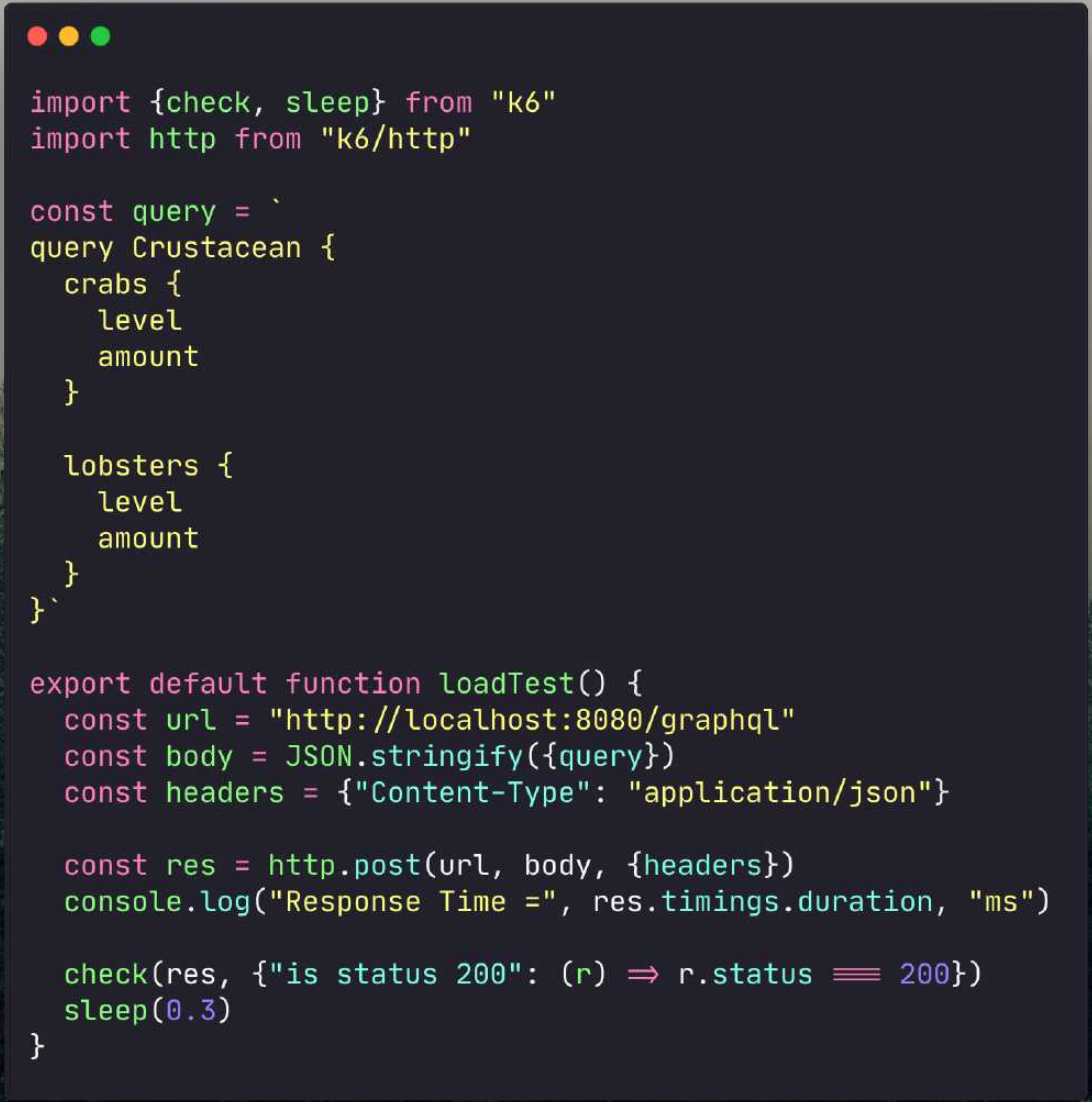
Load Testing GraphQL with **K6**

Our *Metrics* 🧮

Request Duration ⌛

Throughput 🚗

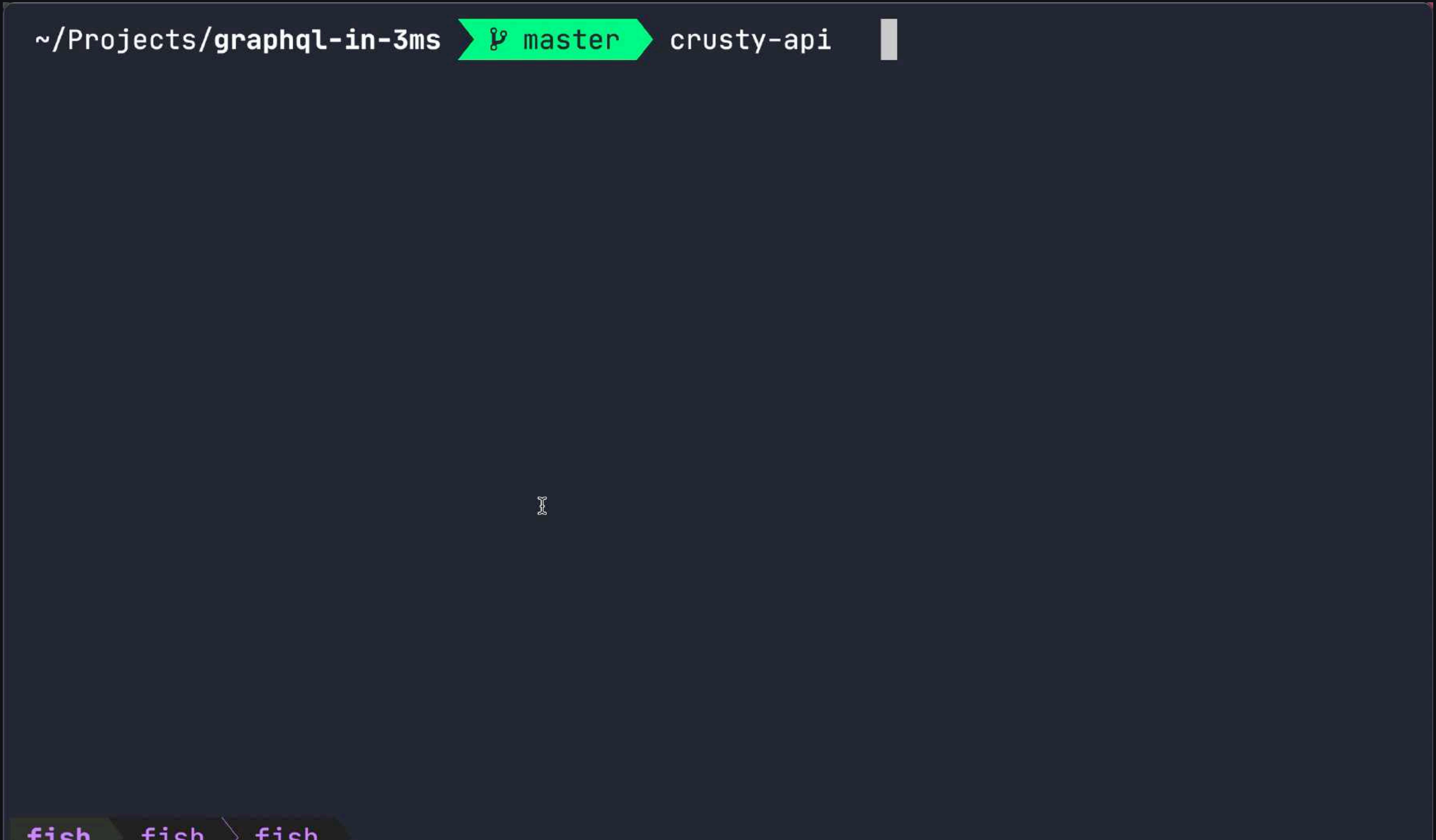
~~WebSockets? 🏓~~



```
import {check, sleep} from "k6"
import http from "k6/http"

const query = `
query Crustacean {
  crabs {
    level
    amount
  }

  lobsters {
    level
    amount
  }
}`

export default function loadTest() {
  const url = "http://localhost:8080/graphql"
  const body = JSON.stringify({query})
  const headers = {"Content-Type": "application/json"}

  const res = http.post(url, body, {headers})
  console.log("Response Time =", res.timings.duration, "ms")

  check(res, {"is status 200": (r) => r.status === 200})
  sleep(0.3)
}
```

TAKE A GUESS

What will be the ***response time*** *be for* ***10 Concurrent Users?***



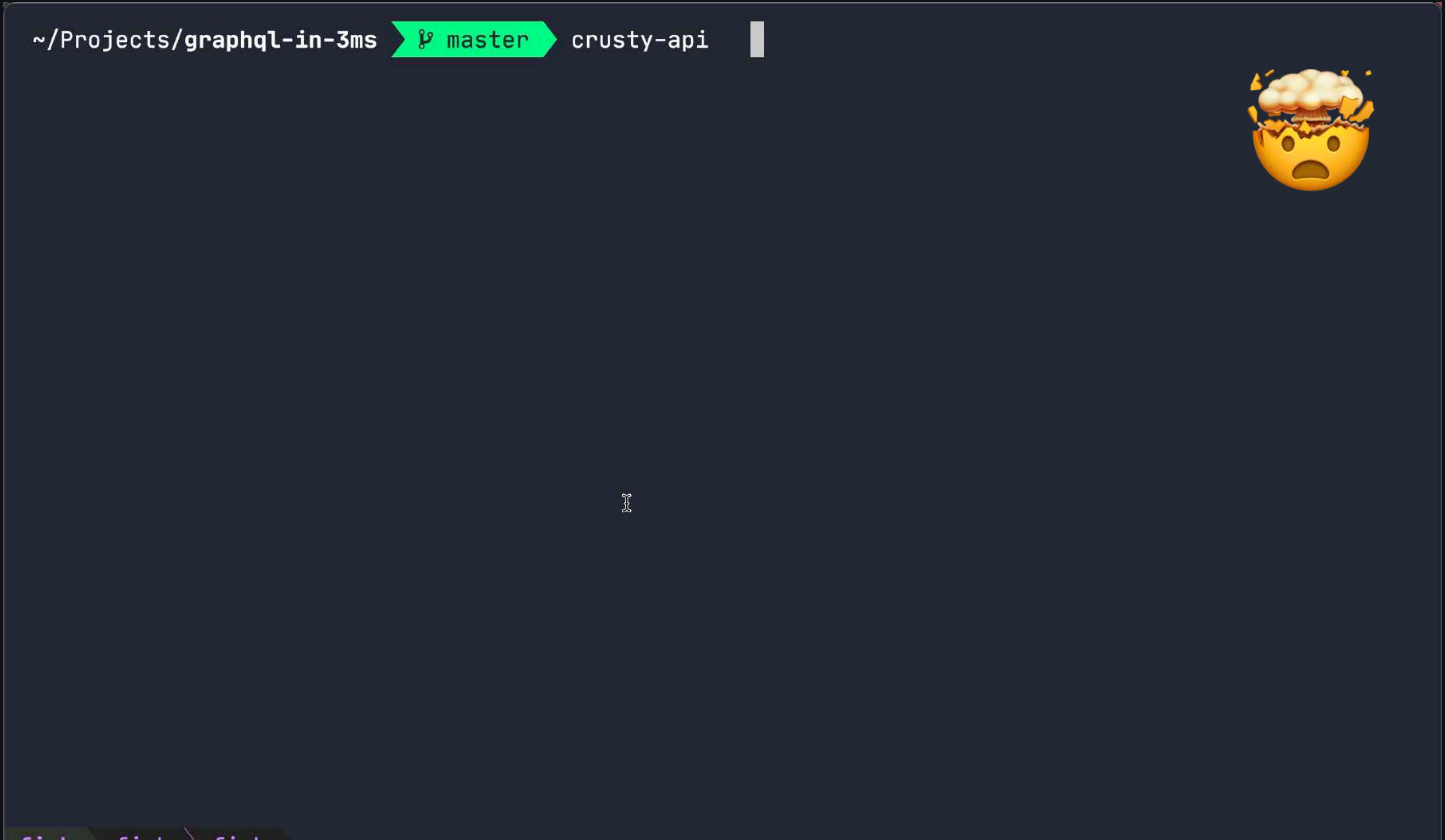
```
~/Projects/graphql-in-3ms  master  crusty-api
```

```
running (10.3s), 00/10 VUs, 331 complete and 0 interrupted iterations
default ✓ [======================================] 10 VUs  10s

    ✓ is status 200

    checks.....................: 100.00% ✓ 331  ✗ 0
    data_received..............: 62 kB   6.0 kB/s
    data_sent..................: 86 kB   8.3 kB/s
    http_req_blocked...........: avg=113.24µs min=2µs      med=4µs      max=4.22ms   p(90)=9µs      p(95)=23.5µs
    http_req_connecting........: avg=34.82µs  min=0s       med=0s       max=1.64ms   p
(90)=0s      p(95)=0s
    http_req_duration..........: avg=1.18ms   min=400µs    med=1.04ms   max=4.53ms   p(90)=1.83ms   p(95)=2.5ms
    http_req_receiving.........: avg=51.47µs  min=15µs     med=33µs     max=715µs    p(90)=81µs     p(95)=131.5µs
    http_req_sending...........: avg=74.34µs  min=10µs     med=28µs     max=861µs    p(90)=189µs    p(95)=336.5µs
    http_req_tls_handshaking...: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
    http_req_waiting...........: avg=1.05ms   min=227µs    med=918µs    max=4.49ms   p(90)=1.64ms   p(95)=2.36ms
    http_reqs..................: 331     32.006387/s
    iteration_duration.........: avg=303.05ms min=300.83ms med=302.49ms max=309.81ms p(90)=305.57ms p(95)=306.38ms
    iterations.................: 331     32.006387/s
    vus........................: 10      min=10 max=10
    vus_max....................: 10      min=10 max=10
```

TAKE A GUESS

What about

200 Concurrent Users?



```
~/Projects/graphql-in-3ms  master  crusty-api
```

Pretty much the same response time!

(and there's 200 ways to optimize it)

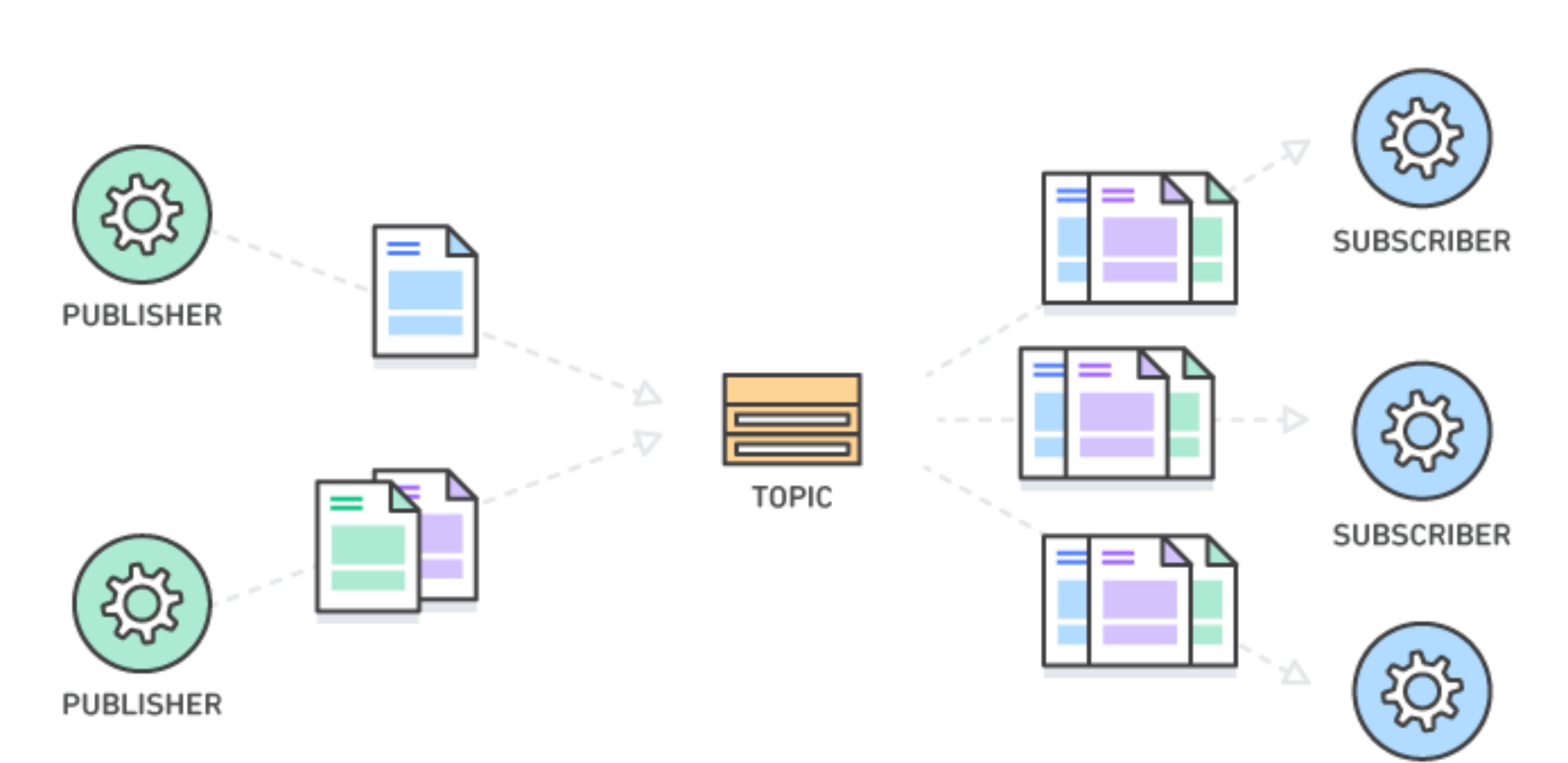
VI.

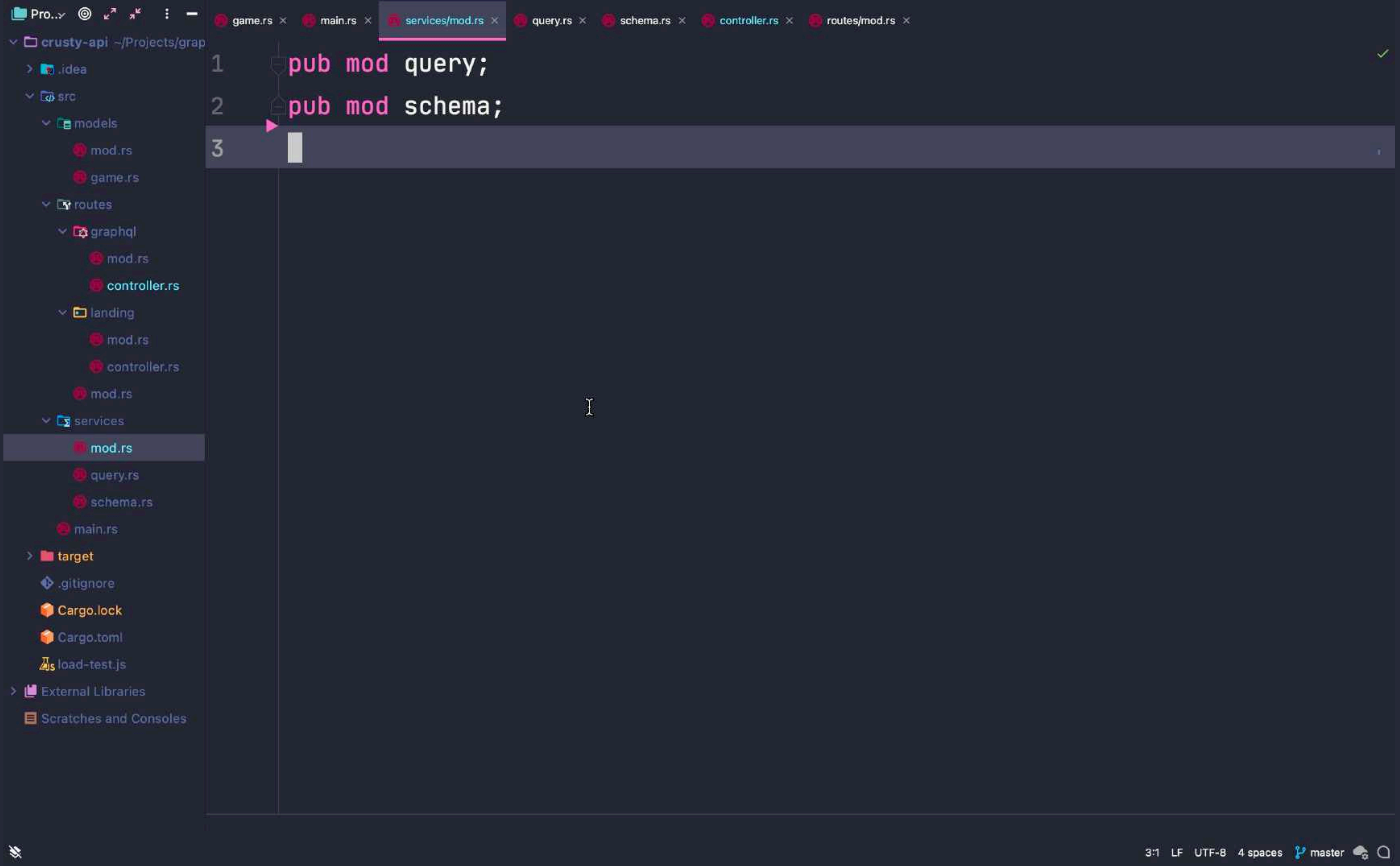
Adding **Subscriptions** *to Juniper*

Who here have used ***subscriptions*** *in GraphQL before?*

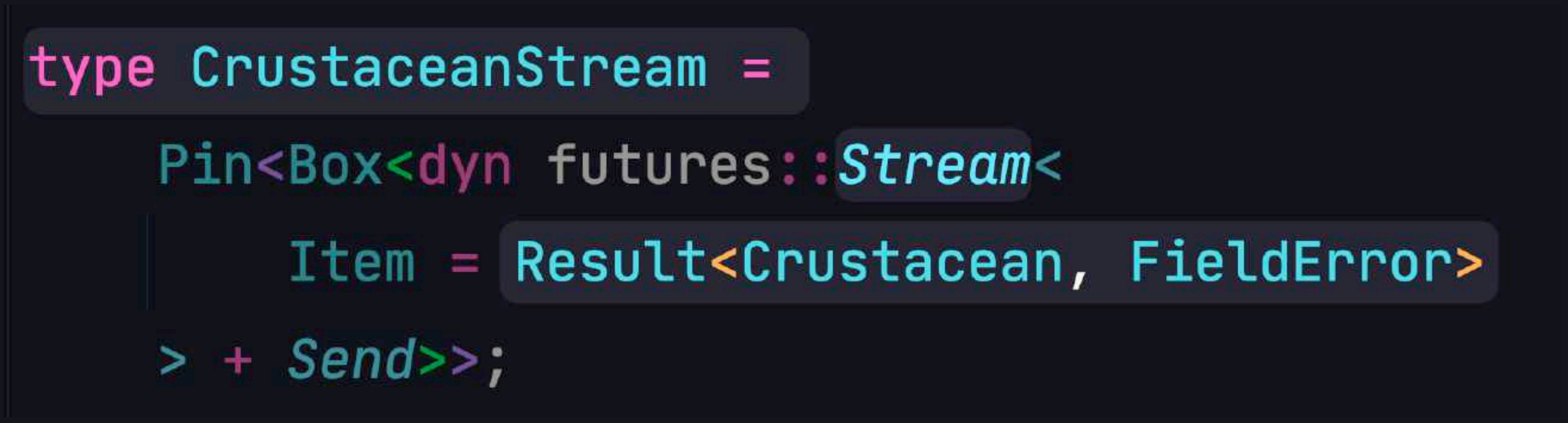
Server **Publish** new events.

Client **Subscribe** to events



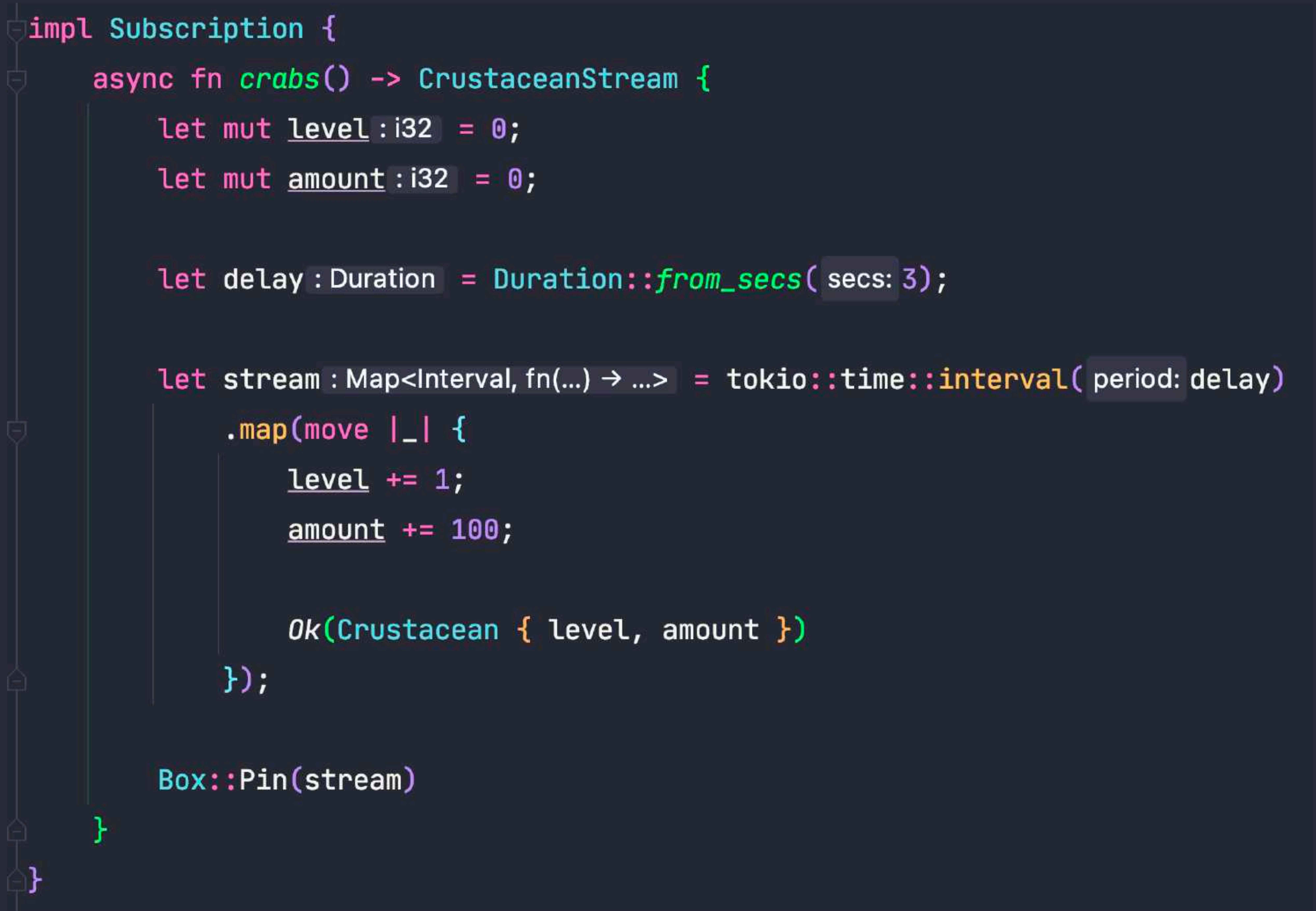


```rust
pub mod query;
pub mod schema;

```

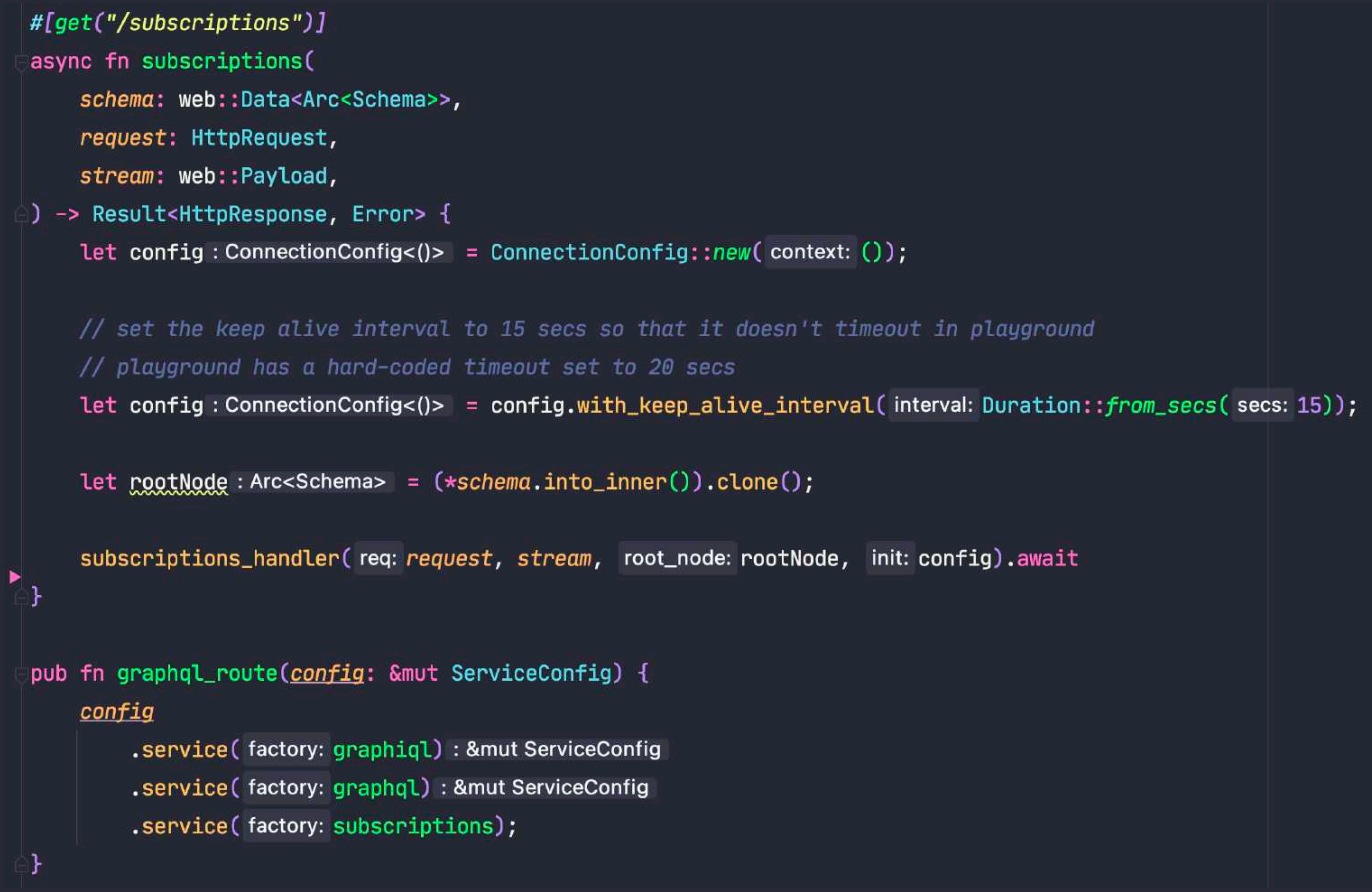


```rust
type CrustaceanStream =
    Pin<Box<dyn futures::Stream<
        Item = Result<Crustacean, FieldError>
    > + Send>>;
```

```rust
pub struct Subscription;
```

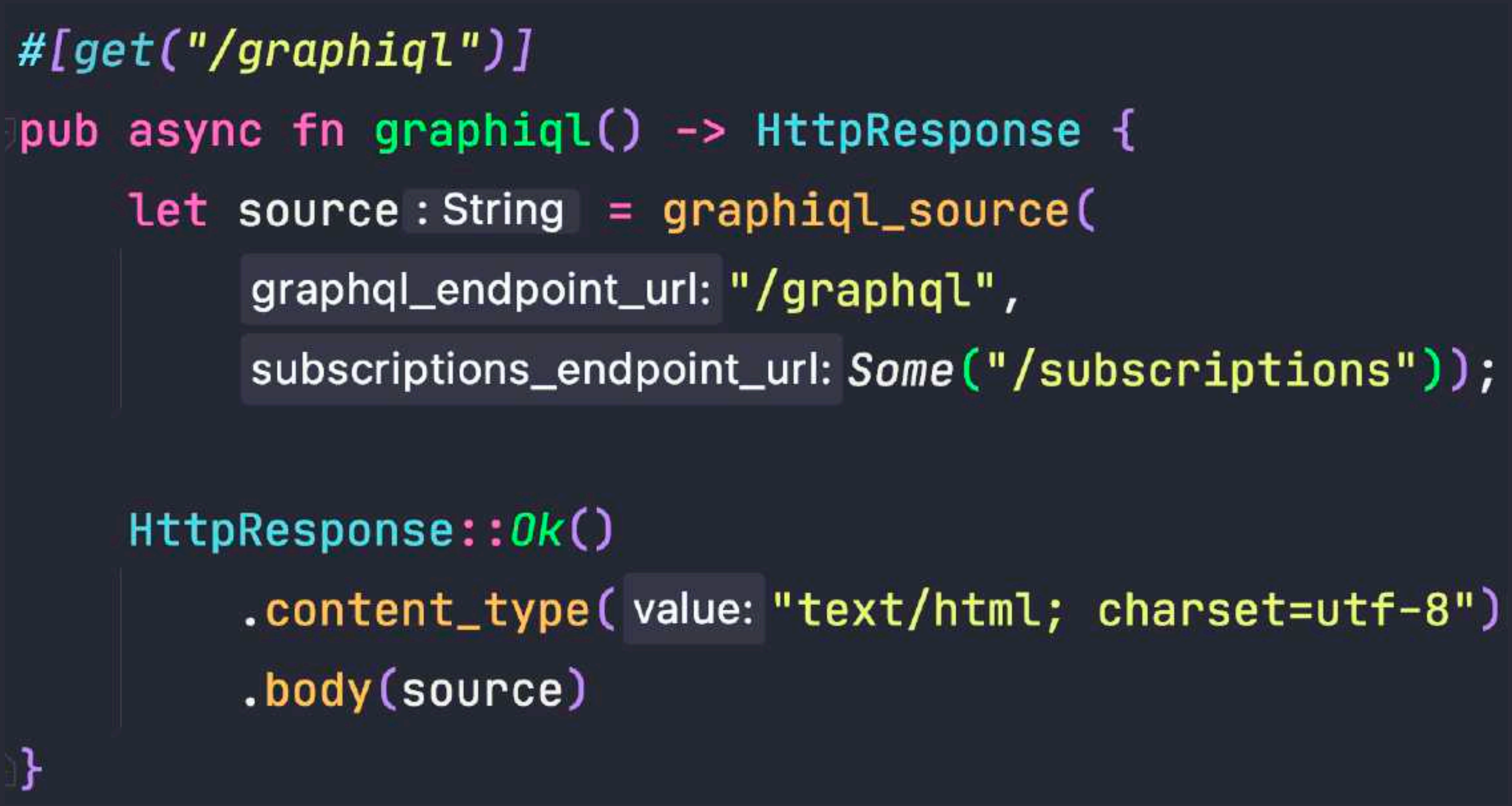


```
impl Subscription {
    async fn crabs() -> CrustaceanStream {
        let mut level : i32 = 0;
        let mut amount : i32 = 0;

        let delay : Duration = Duration::from_secs( secs: 3);

        let stream : Map<Interval, fn(...) → ...> = tokio::time::interval( period: delay)
            .map(move |_| {
                level += 1;
                amount += 100;

                Ok(Crustacean { level, amount })
            });

        Box::Pin(stream)
    }
}
```



```
#[get("/subscriptions")]
async fn subscriptions(
    schema: web::Data<Arc<Schema>>,
    request: HttpRequest,
    stream: web::Payload,
) -> Result<HttpResponse, Error> {
    let config : ConnectionConfig<()> = ConnectionConfig::new( context: ());

    // set the keep alive interval to 15 secs so that it doesn't timeout in playground
    // playground has a hard-coded timeout set to 20 secs
    let config : ConnectionConfig<()> = config.with_keep_alive_interval( interval: Duration::from_secs( secs: 15));

    let rootNode : Arc<Schema> = (*schema.into_inner()).clone();

    subscriptions_handler( req: request, stream, root_node: rootNode, init: config).await
}

pub fn graphql_route(config: &mut ServiceConfig) {
    config
        .service( factory: graphiql) : &mut ServiceConfig
        .service( factory: graphql) : &mut ServiceConfig
        .service( factory: subscriptions);
}
```

Setup **subscriptions URL** in **GraphiQL**



```
#[get("/graphiql")]
pub async fn graphiql() -> HttpResponse {
    let source : String  = graphiql_source(
        graphql_endpoint_url: "/graphql",
        subscriptions_endpoint_url: Some("/subscriptions"));

    HttpResponse::Ok()
        .content_type( value: "text/html; charset=utf-8")
        .body(source)
}
```

Our *Metrics* 🧮

Request Duration ⏳

Throughput 🚗

WebSockets? 🏓

```
subscription {
  crabs {
    level
    amount
  }
}
```

```json
{
  "crabs": {
    "level": 92,
    "amount": 9200
  }
}
```

How fast do you expect this will be?

Let's see the demo.

THE FLASH
THE CW

KEY POINTS 🤪

Rust has great ***performance*** like C 🚀
with ***modern facilities*** like TypeScript 🍣

Code-first GQL is **clean** with **Rust *Macros*** 🎲

React to **real-time events** with ***Subscriptions*** 🚤

Use ***K6*** to write JS to do **load testing** 😵

THINGS TO TRY NEXT 😋

Run Serverless Rust on **Google Cloud Run** ⛅

Build our code to **WebAssembly** target ⚡

Run it on the **Edge** with **Cloudflare Workers**
Persist **WebSocket** with **Durable Objects**
Store data on **Workers KV**

one last thing.

brikl

we're hiring.

brikl.com/careers

Let's go *AFTER PARTY*

@ ADMIRAL PUB

(Sukhumvit)

brikl

Senior Software Engineer

Frontend || Backend

Site Reliability Engineer

QA Engineer

brikl.com/career

That's it.

Thank you!

Any Questions?