

# CS188

# Scalable Internet Services

---

John Rothfels, 11/17/20

# Recall

Where we left off last week...

(code on the [jnr/perf branch](#) of the course website)

- Use k6 load testing script
- Look at data in Honeycomb
- Identify performance problems
- Fix
- Re-measure

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?
- Too many computations? Memory overload?
- Poor API design / implementation? Tight coupling?  $n+1$ ?

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?

MySQL is designed to work with high concurrency. Why is this important?

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?

MySQL is designed to work with high concurrency. Why is this important?

- One query may be reading data from disk; another query can start processing
- Multi-core machines are mainstream

# Investigation

 **Hypothesis? What's going on to create the latency?**


- Connecting to (or querying) the DB?

Your appserver connects to MySQL, but that doesn't mean it can execute queries in parallel! A single connection to MySQL can do one query at a time. If you want to do more than one query at a time, you need to have multiple connections open to the database.

# Investigation

 Hypothesis? What's going on to create the latency?


- Connecting to (or querying) the DB?

Creating a database connection has some overhead associated with it. Given how frequently we query the database, we would **not** want to have to open a new connection for each query.  Why?

# Investigation

 Hypothesis? What's going on to create the latency?

- Connecting to (or querying) the DB?

Creating a database connection has some overhead associated with it. Given how frequently we query the database, we would **not** want to have to open a new connection for each query.  Why?

- What if we open too many connections?
- What if a query finishes on a connection? Can we reuse the connection?



# Investigation

 Hypothesis? What's going on to create the latency?


- Connecting to (or querying) the DB?

```
export async function initORM() {  
  return await createConnection({  
    ...  
    extra: {  
      connectionLimit: 5, // this number is possibly too small  
    },  
  })  
}
```

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?

 Make sure your database can support the number of connections that you will attempt to open.

```
SHOW VARIABLES LIKE 'max_connections';
```

```
SET GLOBAL max_connections = 1024;
```

## Aside: resource pools

A common scaling pattern is to create **pools** of resources. Your pool should both:

- reduce the overhead of creating new resources (one may already be allocated in the pool)
- put limit on resource allocation (by limiting the size of the pool)



The most common resource pools you find on web/application servers are **thread pools** and **database connection pools**.

💡 Use (or tune) your pools to help scale your application.

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Too many computations? Not enough RAM?

Usually this is easy to see, but harder to identify the root cause of.

- CPU: look for trace spans that are long, events with high `duration_ms` or `SUM(duration_ms)`
- RAM: look for long lists, too much data being read out of your database

## Aside: know your runtime

Recall: application servers running Node.js have a ***single thread of execution*** running the “event loop”. Using only non-blocking system calls for IO makes this efficient.

**!?** What happens if your server is handling a request, or doing something in the background, that requires CPU? How does this differ from a multi-threaded environment?

## Aside: background “process”

Here's a background “process” (or sometimes called a “tick”, it's not an actual process) you can run on your appserver which will take CPU time every 100ms:

```
setInterval(() => {  
  const data = 'hash-me'  
  require('crypto').createHash('sha256').update(data).digest('base64')  
}, 100)
```

!/? What happens to your server performance if you add such a background process? What if it has to compute 100 hashes every tick? 1000?

## Aside: background “process”

You are **\*not\*** required to have a background process on your server. (I have updated the Project requirements on the course website.)

But you can get credit for testing the effects of different kinds of background processes on your server! Feel free to use the one from the previous slide and modify it to fit your needs. (As-is, it probably wouldn't have a huge effect.)



In a production setting, to scale we would very likely need to run background / CPU heavy tasks on a separate computer (or Node process). **Why?**

# Recall

You are creating multiple type systems in your application:

- `schema.graphql` defines the ***shape of your API***. It is an interface implemented on the server by `api.ts`.
- MySQL schema defines the ***shape of your tables***. It is created for you by TypeORM (unless you manually modify the db via migrations).
- TypeORM entities define the ***shape of your objects stored & read from DB***.



# Recall

You are creating multiple type systems in your application:

- `schema.graphql` defines the ***shape of your API***. It is an interface implemented on the server by `api.ts`.
- MySQL schema defines the ***shape of your tables***. It is created for you by TypeORM (unless you manually modify the db via migrations).
- TypeORM entities define the ***shape of your objects stored & read from DB***.



**They do not always play nicely together!**

# Recall

Recall the benefits of GraphQL:

- Strictly defined data
- Codegen (`npm run gen`) can create types for you 🎉
- Fewer round trips to server
- No tight coupling between frontend and backend
- Non-verbose API definition
- No over- or under-fetching

# Recall

Recall the benefits of GraphQL:

- Strictly defined data
- Codegen (`npm run gen`) can create types for you 🎉
- Fewer round trips to server
- **No tight coupling between frontend and backend \*\***
- **Non-verbose API definition \*\***
- **No over- or under-fetching \*\***

Unfortunately, there's no free lunch. 😭

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?

Conflating our TypeORM entities (types) with our GraphQL types is tight coupling!  
When should we load relations?

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Poor API design / implementation? Tight coupling?  $n+1$ ?

Conflating our TypeORM entities (types) with our GraphQL types is tight coupling!

When should we load relations?

- Eagerly?
- Only when requested by the client?

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?


The way GraphQL works, you have a bunch of **resolver** (functions). They have to return scalar values (primitives), objects, or Promise of either.

 *Any fields you put on your objects are visible to the GraphQL execution engine.*

# Investigation

 **Hypothesis? What's going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?


 We can decouple the resolver functions for fields on our GraphQL schema from the TypeORM objects returned by the resolvers. The GraphQL machinery can decide to call the resolver function if and only if the query requests that particular data.

# Investigation

## Hypothesis? What's going on to create the latency?

- Poor API design / implementation? Tight coupling?  $n+1$ ?

Decoupling resolver functions can lead to the dreaded  $n+1$  problem:

- Your GraphQL client requests a field on a query that returns a list (of objects)
- For each object returned, you want an edge (relation)
- Each edge calls a resolver function that requests a single row from the database
- For a list of  $n$  edges, you make  $n$  SQL calls to fetch  $n$  rows! 



# Fetching from the DB

It's *very* common for latency to come from how we are reading data from the DB:

- Are we reading more data (columns or rows) than we need?
- Are we reading rows one-by-one in a loop? Could we make a single request instead to read all the rows in a batch?
- Are we reading rows over and over again that never (or infrequently) change?
- Are we looking for rows without using an index? Or doing a complicated join?
- Is our database schema forcing us to make inefficient queries?

# Fetching from the DB

It's very common for latency to come from how we are reading data from the DB:

- Are we reading more data (columns or rows) than we need?
- **Are we reading rows one-by-one in a loop? Could we make a single request instead to read all the rows in a batch?**
- **Are we reading rows over and over again that never (or infrequently) change?**
- Are we looking for rows without using an index? Or doing a complicated join?
- Is our database schema forcing us to make inefficient queries?



Can we automatically batch & cache our queries? How would it work?

# Investigation

 Hypothesis? What's going on to create the latency?

- Poor API design / implementation? Tight coupling?  $n+1$ ?

At scale you want to use batching and caching *liberally* to avoid unnecessary DB lookups, especially within a single request. The [dataloader library](#) provides automatic batching and caching for your resolver functions.

You define a `DataLoader`, which knows how to read a batch of rows from the DB by ID. You ask the `DataLoader` to load single rows, and it transparently creates does batching and caching under the hood.

# Fetching from the server

**!?** What if we serve the app to the browser and then some data changes on the server while the user is in the app? How can we update the app with the new data from the server?

# Fetching from the server

**!?** What if we serve the app to the browser and then some data changes on the server while the user is in the app? How can we update the app with the new data from the server?

- Re-request the data (**polling**)
- Have the server tell us about the data (**subscription**)

# Fetching from the server

Re-request the data (**polling**):

- If the client thinks the data it fetched could be stale, it can request it again.

```
const { loading, data, refetch } = useQuery<FetchCandies>(fetchCandies)
```

- Apollo lets you specify a `pollInterval` to repeat the query automatically

```
useQuery<FetchCandies>(fetchCandies, { pollInterval: 10000 })
```

**!?** Pros / cons?

# Fetching from the server

Re-request the data (**polling**):


- Pros:
  - Simple
  - Easy to implement
  - Easy to load test (set up your load script to simulate the polling)
- Cons:
  - Server has to handle more requests
  - Data may not have changed, clients request new data anyway
  - Fresh data not immediately delivered to the client
  - You have to poll more to get fresh data faster

**!?** Does this approach scale? What happens if we have hundreds of components, each with data requirements, each polling on an interval?

# Fetching from the server

Have the server tell us about the data (**subscription**):

- The HTTP protocol provides uni-directional data flow over a TCP connection: client requests data, server responds with data
- The **websocket** protocol provides bi-directional data flow over a TCP connection: client opens a websocket, client can deliver messages to the server, server can deliver messages to the client

 We may think of there being “requests” and “responses” over a websocket, but this need not be the case. We can have a websocket where *\*only\** the client sends messages, or *\*only\** the server.



# Fetching from the server

Have the server tell us about the data (**subscription**):

- Open a websocket
- Let the server know “we want to hear about \$TOPIC”
- Server keeps a list of websocket connections interested in every topic
- When there's new information about a topic, server publishes data to the subscribed clients (by sending a message down the websocket)

# Fetching from the server

Have the server tell us about the data (**subscription**):

- GraphQL support! Define `type Subscription`, and resolvers which return `AsyncIterator`

```
type Subscription {  
  surveyUpdates (surveyId: Int!): Survey  
  candyUpdates: UserCandy  
}
```

# Fetching from the server

## api.ts

```
Subscription: {  
  surveyUpdates: {  
    subscribe: (_, { surveyId }, ctx) => ctx.pubsub.asyncIterator('SURVEY_UPDATE_' + surveyId),  
    resolve: (payload: any) => payload,  
  },  
  candyUpdates: {  
    subscribe: (_, arg, ctx) => ctx.pubsub.asyncIterator('CANDY_UPDATE'),  
    resolve: (payload: any) => payload,  
  },  
},  
  
ctx.pubsub.publish('SURVEY_UPDATE_' + surveyId, survey)
```

# Fetching from the server

Have the server tell us about the data (**subscription**):

- Pros:
  - Fewer requests for server to handle than polling
  - Fresh data immediately delivered to client
- Cons:
  - Much more complicated to implement / test
  - Stateful! Your backend has to keep track of which websocket connections are subscribed to what topics

# Fetching from the server

For your final project, you should try adding polling to your application. It takes very little work, and is easy to simulate polling behavior in your load testing script.

Ambitious students can try using GraphQL subscriptions in their project. It is [possible](#) but more difficult to load test; you can get credit for just implementing them.

# What else can go wrong?

! ? What else should we be thinking about if we want a more performant server?

# What else can go wrong?

! ? What else should we be thinking about if we want a more performant server?

- ORM-generated SQL queries
  - Can you do better writing SQL yourself? Consider moving away from ORM entirely; use a [query builder library](#) instead.
- Dependencies!!!
  - [Are we using a good GraphQL server?](#)
  - How do we know our dependencies are good?
- GraphQL
  - Do we need it? Are we better off with ordinary REST?