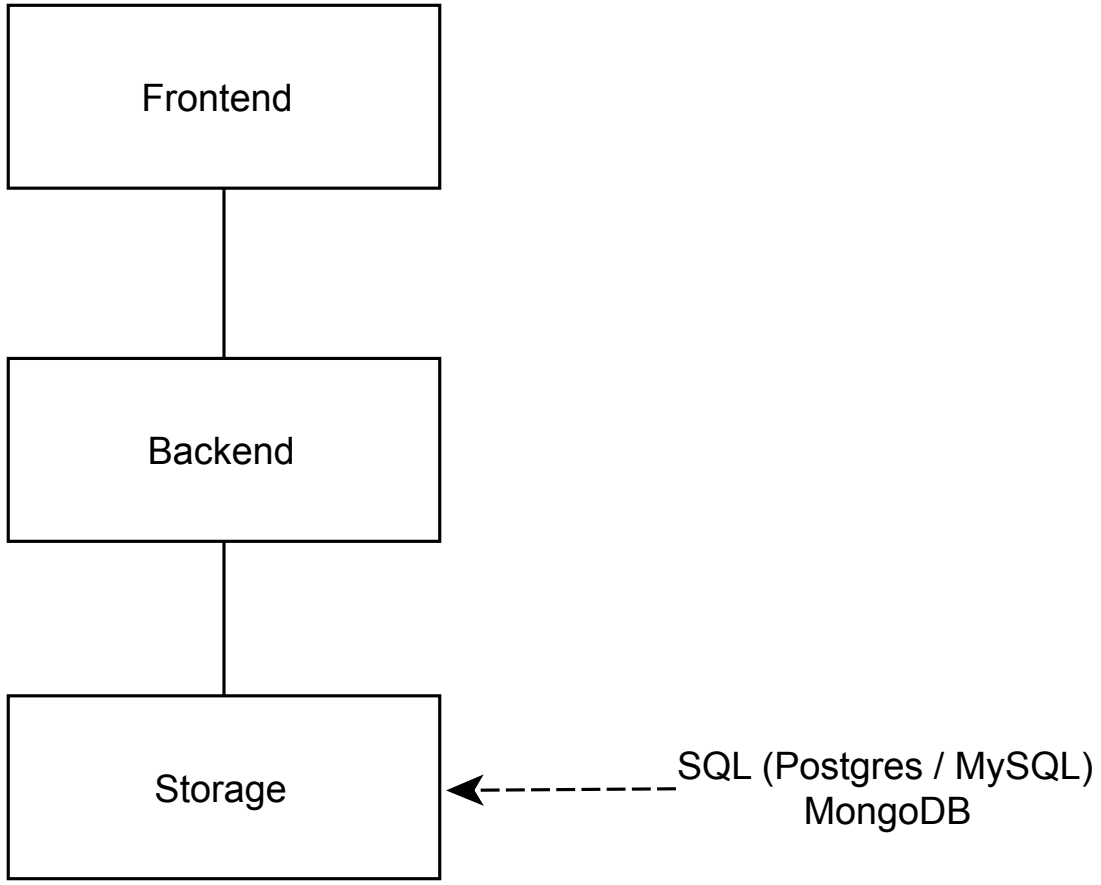
Programowanie App Internetowych

Bazy danych

Bazy danych



Na dziś

- bazy danych - co mamy do wyboru?
- plain SQL vs DSL vs ORM
- SQL DB, express.io i knex
- ORM na przykładzie primsa.io

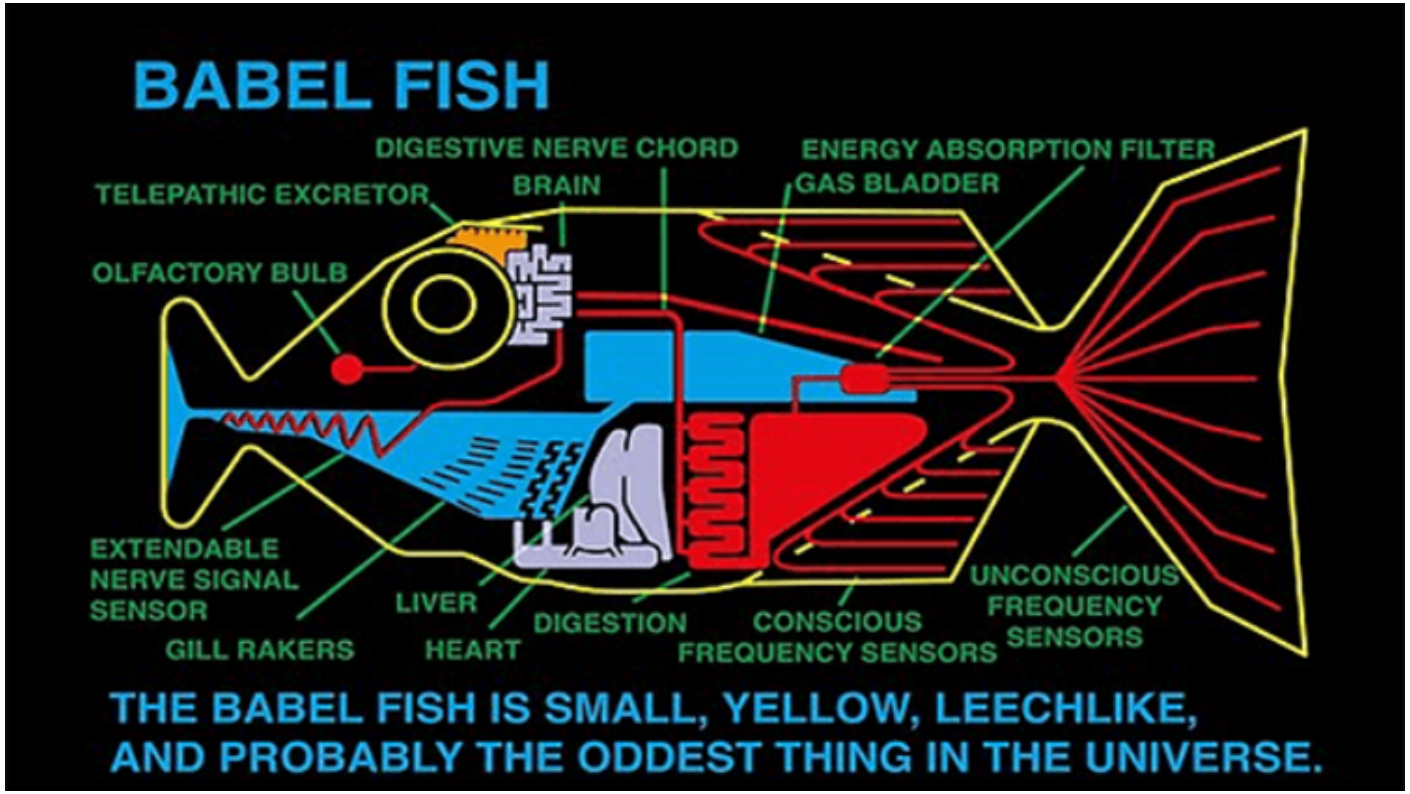
Bazy danych

- SQL: PSQL && MySQL,
- noSQLs: Amazon DynamoDB, mongodb,
- cache: od edge i web cache po redis/memcache po stronie backendu.

Nie zapomnijmy o

- Object storage, np.: Amazon S3 czy Google Storage
- kolejki:
  - Amazon SQS, RabbitMQ, Kafka.
- Backend-as-a-Serive, np., Google Firebase.

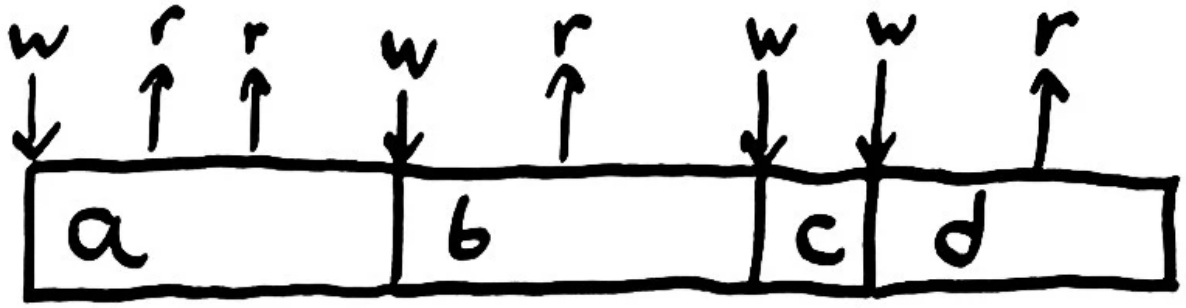
SQL


BABEL FISH
DIGESTIVE NERVE CHORD
ENERGY ABSORPTION FILTER
TELEPATHIC EXCRETOR
BRAIN
GAS BLADDER
OLFACTORY BULB
EXTENDABLE
NERVE SIGNAL
SENSOR
LIVER
DIGESTION
CONSCIOUS
FREQUENCY SENSORS
UNCONSCIOUS
FREQUENCY
SENSORS
GILL RAKERS
HEART
THE BABEL FISH IS SMALL, YELLOW, LEECHLIKE,
AND PROBABLY THE ODDEST THING IN THE UNIVERSE.

SQL

Nie będziemy omawiać:

- CAP/PACELC;
- Modele spójności i ACID;
- Eventual consistency.



Którą bazę danych?

Pragmatycznie:

1. SQL - zgodny z Postgres lub MySQL API;
2. Hostowany przez cloud providers (aka życie za krótkie jest, aby zarządzać bazą danych i backupem);
3. Nie szedłbym w NoSQL czy grafowe, jeśli nie ma bardzo mocnych przesłanek.

Którą bazę danych?

Side note - ważna decyzja techniczna

1. Napisz design doc,
2. Warto dodać sekcję pre-mortem.

Którą bazę danych?

New global scale DBs:

- cockroach DB;
- Yugabyte;
- ...

Fascynująca dziedzina w inżynierii!

Baza danych vs App

1. Plain SQL
2. Lightweight DSL over SQL (**tutaj zacznij**)
3. ORM

Zobacz też: post na blogu logrocketa.

Plain SQL with Database driver

```
const mysql = require('mysql')
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'dbuser',
  password: 's3kreee7',
  database: 'my_db'
})

connection.connect()

connection.query('SELECT 1 + 1 AS solution', (err, rows, fields) => {
  if (err) throw err

  console.log('The solution is: ', rows[0].solution)
})

connection.end()
```

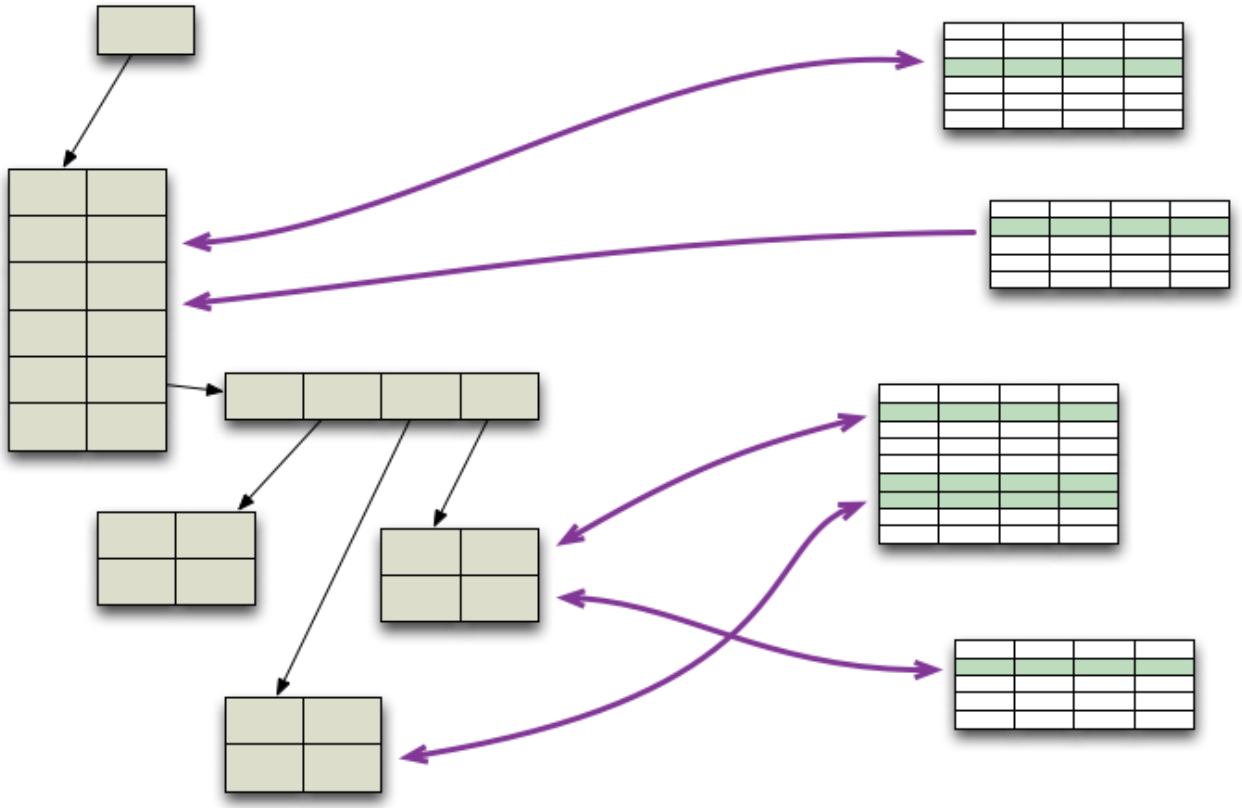
Lightweight DSL

knex - query builder:

```
const pg = require('knex')({client: 'pg'});

knex('table')
  .insert({a: 'b'})
  .returning('*')
  .toString();
```

ORM



ORM

Famous example (not a recommendation):

- hibernate - Java
- SQLAlchemy - Python

ORM

Na plus:

- Avoid redundant code
- Easily switch from one database to another
- Query for multiple tables (ORM converts the object-oriented query approach to SQL)
- Focus more on business logic and less on writing interfaces

ORM

Praktyka:

- niska wydajność;
- daje fałszywe poczucie, że nie trzeba myśleć o bazie danych;
- dużo magi, trudne w debugingu.

Rekomendacja

- Unikaj ORMów;
- Query buildery lub DSL z lekką abstrakcją OK;
- Zacznij od query builderów, np., knex;
- Weryfikuj zapytania z `explain` i `analyze` !!
- Monitoruj zapytania w produkcji.

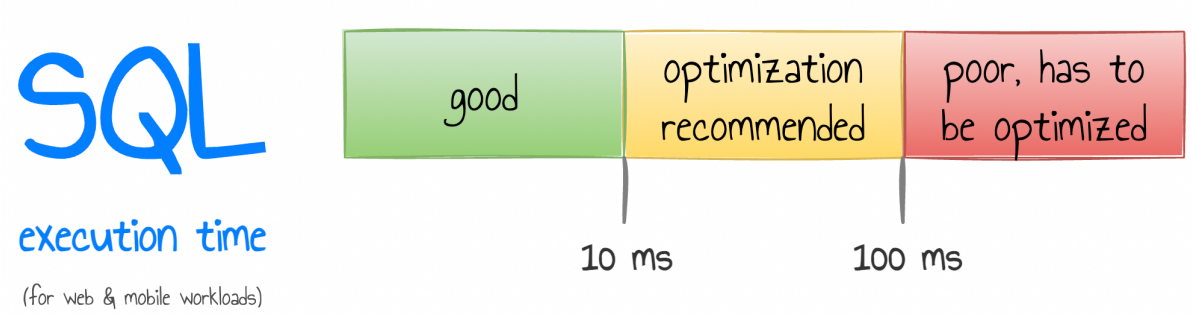
Na przykład - PSQL

- Explain,
- Query Plan Visualization
- Types of Indexes

Good read: Debugging PSQL the hardway.

Performance

- Powyżej 200ms = **za wolno**!!
- Generalnie (src):



PSQL + Expressjs

Expressjs + Knex


Prisma

Prisma

- ORM for Node.js and Typescript

Prisma

Model (`prisma/schema.prisma`):

```
model User {
  id    Int     @id @default(autoincrement())
  email String  @unique
  name  String?
  posts Post[]
}

model Post {
  id        Int     @id @default(autoincrement())
  title     String
  content   String?
  published Boolean @default(false)
  author    User    @relation(fields: [authorId], references: [id])
  authorId  Int
}
```

Prisma

Create a tuple:

```
const user = await prisma.user.create({
  data: {
    name: 'Alice',
    email: 'alice@prisma.io',
  },
})
```

Prisma

Query:

```
const users = await prisma.user.findMany()
console.log(users)
```

Prisma

Create with the related tuple:

```
const user = await prisma.user.create({
  data: {
    name: 'Bob',
    email: 'bob@prisma.io',
    posts: {
      create: {
        title: 'Hello World',
      },
    },
  },
})
```

Prisma

- Quickstart
- Next (after learning *knex*):
  - prisma + express
  - prisma + Next.js

Dziękuję

Czy są jakieś pytania?