

Node.js: modul mysql + Sessions

modul **mysql**

- komunikácia s SQL serverom
 - napísaný v JS
- objavuje sa aj kompatibilný modul **mysql**
 - rovnaké API, trochu rýchlejší, ...

- inštalácia

```
$ npm install mysql
```

- načítanie modulu

```
const mysql = require ('mysql');
```

Vytvorenie spojenia

- **spojenie na db** server (mariadb, mysql)
 - zvykne sa aj označovať ako session

```
const mysql = require ('mysql');

const connection = mysql.createConnection ({
  host      : 'localhost',
  user      : 'janko',
  password  : 'mojeTajneHeslo',
  database  : 'mojPortal'
});

connection.connect ();
```

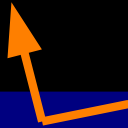
SQL dotaz

- poslanie dotazu, výsledok cez callback

```
connection.query ('SQL dotaz', (err, rows, fields) => {  
  // err      ak nastavené, objekt triedy Error  
  // rows     pole s objektami-riadkami výsledku  
  // fields   info o stĺpcoch výslednej tabuľky  
});
```

- napríklad:

```
connection.query ('SELECT meno FROM Users', (err, rows) => {  
  
  if (err) throw err;  
  
  for (let i = 0; i < rows.length; i++)  
    console.log (i+'.riadok: ', rows[i].meno);  
  
});
```



Riadky sú reprezentované ako objekty, kde názvy položiek zodpovedajú názvom stĺpcov.

Error objekt z dotazu

- inštancia rozšírenej triedy Error

`err.code`

- kód (reťazec) chyby zlyhania

`err.sql`

- reťazec s SQL dotazom, ktorý zlyhal
- dobré na ladenie
- skopírovať si do interaktívneho nástroja
napr. PHPMysqlAdmin-a ...

SQL dotazy

- iba jeden SQL príkaz naraz
 - z bezpečnostných dôvodov
- v rámci spojenia
 - všetky požiadavky sú vykonávané sekvenčne
 - funguje aj zamykanie a odomykanie tabuliek

Ukončenie spojenia

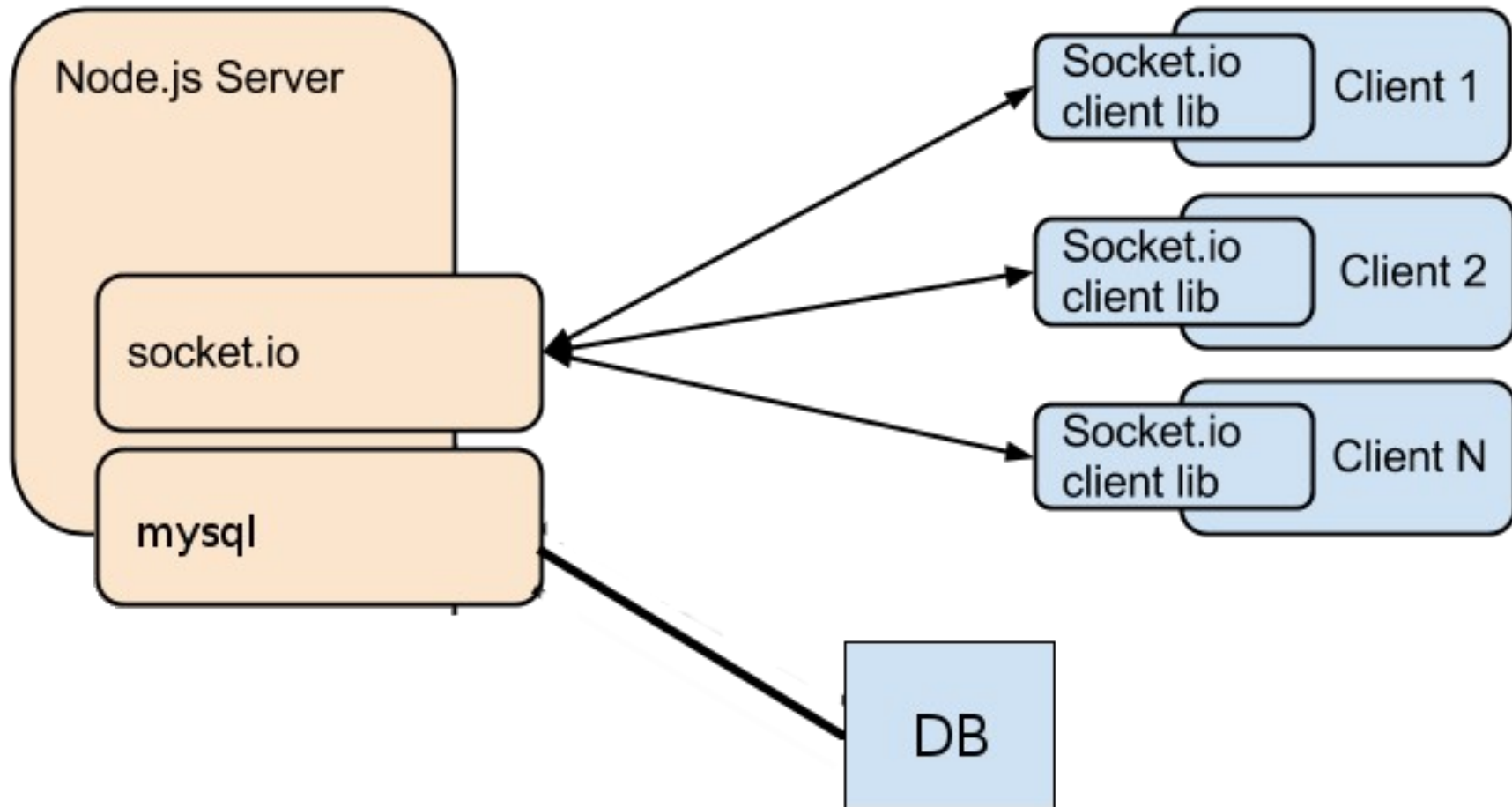
- ukončenie spojenia
 - dokončia sa odoslané požiadavky
 - zrušia sa zámky

```
connection.end ( err => {  
    if (err) throw err;  
    console.log ('Odpojené');  
});
```

- okamžité zrušenie spojenia
 - zrušia sa zámky

```
connection.destroy ()
```

Architektúra



Promise namiesto callbacku

- promisifyovanie vybranej funkcie

```
const dbQuery = (db, sql) => {  
  return new Promise ( (resolve, reject) => {  
    db.query (sql, function (err, res) {  
      if (err) reject (err);  
      else resolve (res);  
    });  
  });  
}
```

```
dbQuery (db, "SELECT * FROM Studenti")  
  .then (rows => {fn(rows)},  
        err => {console.log(err)})
```

util.promisify

- modul **util** je z node.js
- promisifyuje funkciu
 - obalí ju do promisu
- obaľovaná funkcia musí mať ako **posledný argument callback** s parametrami (**err, value**)

Promise pomocou modulu **util**

- obalenie do Promisu

```
util = require ('util');  
db.queryAsync = util.promisify (db.query);
```

```
db.queryAsync ("SELECT * FROM Studenti")  
  .then (rows => {fn(rows)},  
        err => {console.log(err)})
```

- objekt db (spojenie) možno aj „zabindovať“

```
util = require ('util');  
const queryAsync = util.promisify (db.query.bind(db));
```

```
queryAsync ("SELECT * FROM Studenti")  
  .then (rows => {fn(rows)},  
        err => {console.log(err)})
```

Teraz nie je potrebné používať
db objekt, lebo je „zabindovaný“
priamo vo funkcii.

insertId

- zistenie autoincrement primary key

```
(async () => {  
  let res;  
  
  res = await db.queryAsync ("INSERT INTO Users SET name='MN'");  
  console.log (res.insertId)  
  
}) ();
```

affectedRows

- pri SQL dotazoch **insert**, **update**, **delete**
- počet prehľadaných/postihnutých záznamov

```
(async () => {  
  let res;  
  
  res = await db.queryAsync ("DELETE FROM Users WHERE age>10");  
  console.log (res.affectedRows)  
  
}) ();
```

changedRows

- pri SQL dotaze **update**
- počet reálne zmenených záznamov

```
(async () => {  
  let res;  
  
  res = await db.queryAsync ("UPDATE Users SET name='Janko'");  
  console.log (res.changedRows)  
  
}) ();
```


„Šablóny“ SQL dotazov

```
let sql = "SELECT * FROM ?? WHERE ?? = ? AND ?";  
sql = mysql.format (sql, ['Users', 'id', 342, {age:3}]);  
console.log (sql);  
// SELECT * FROM `Users` WHERE `id` = 342 AND `age`=3
```

- ? → hodnota
 - ochrana proti injection attacks
 - čísla, reťazce
 - objekty rozbiže → key1='value1', key2='value2', ...
 - polia rozbiže → 'value1', 'value2', 'value3', ...
- ?? → identifikátor

SQL zamykanie tabuliek

- v prípade viacerých spojení na SQL server

- spojenie **zamkne** tabuľku

- **read lock** – všetky spojenia môžu iba čítať

LOCK TABLE Tabulka READ

- **write lock** – ostatné spojenia nemôžu ani zapisovať ani čítať

LOCK TABLE Tabulka WRITE

Chcem z tabuľky niečo prečítať
bez nepredvídateľnej zmeny.


Idem niečo
do tabuľky zapisovať.

- spojenie **odomkne** tabuľku

UNLOCK TABLES

- odomkne všetky zámky

```
(async () => {  
  try {  
    await db.queryAsync ("LOCK TABLE Users WRITE");  
  
    res = await db.queryAsync ("SELECT COUNT(*) AS c FROM Users");  
    if (res[0].c < 10)  
      await db.queryAsync ("INSERT INTO Users SET name='Janko'");  
  }  
  catch (err) {  
    console.log ('err:', err.sql);  
    throw new Error ('Problém vložit uživateľa!');  
  }  
  finally {  
    await db.queryAsync ("UNLOCK TABLES");  
  }  
}) ();
```




Pozor, ak sa
neodomkne, ostatní
klienti sa k údajom
nedostanú!

Pool - zlúčená sada spojení

- vytvorenie **banky spojení** na **ten istý server**
 - spojenia sa vytvoria, ak je to potrebné
- na SQL požiadavku sa nájde voľné spojenie
 - klient môže vykonávať **požiadavky paralelne**

```
const pool = mysql.createPool({  
  connectionLimit : 10,  
  host            : 'localhost',  
  user            : 'ferko',  
  password        : 'mojeHeslo',  
  database        : 'MojPortal'  
});
```



Určenie maximálneho
počtu spojení na SQL
server

SQL dotazy cez pool

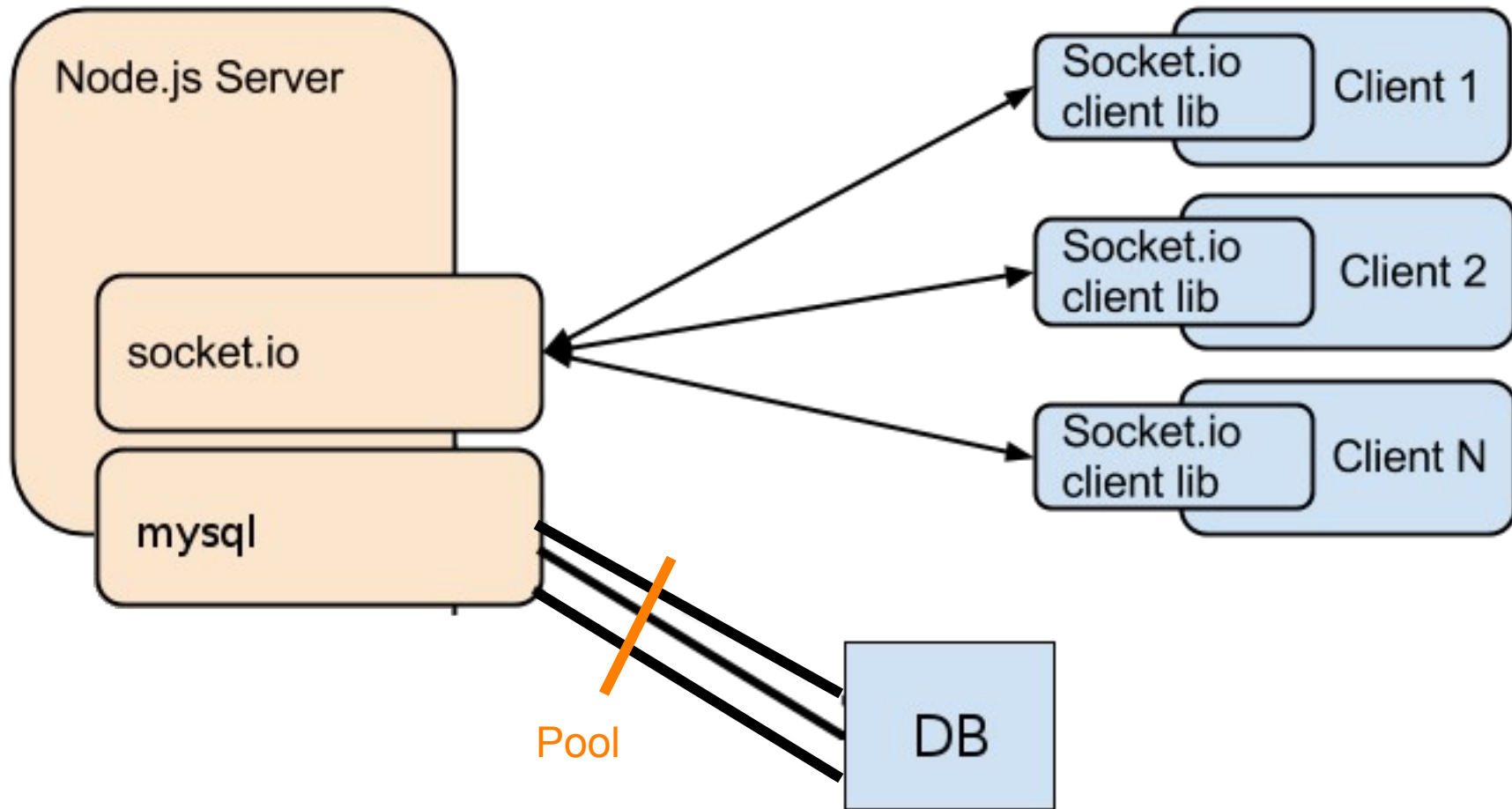
- cez metódu **query**
 - rovnaká syntax ako pri jednom spojení
- urobiť si z nej Promise verziu

```
util = require ('util');  
pool.queryAsync = util.promisify (pool.query);  
  
(async () => {  
  let rows = await pool.queryAsync ('SELECT ....');  
}) ();
```

- ukončenie pool-u (všetkých spojení)

```
pool.end ( err => {  
  if (!err) console.log ('Ukončené');  
});
```

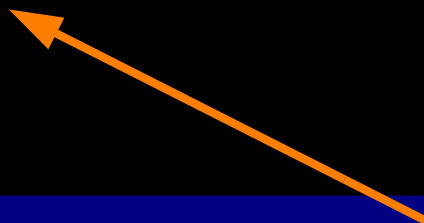

Architektúra



Vyňatie jedného spojenia z pool-u

- spojenie už **nebude** všeobecne **k dispozícii**
- vhodné pri LOCK / UNLOCK operáciách
- vhodné tiež promisifikovať

```
pool.getConnection( (err, con) => {  
  // Vybrané spojenie z pool-u  
  
  con.query ('SELECT .....', (err, rows) => {  
    con.release (); // Vrátené spojenie do pool-u  
  });  
});
```



Spojenie možno vrátiť späť, ale ho aj možno ukončiť **con.end()**.

Promisifikovanie metód tried **Pool** a **Connection**

```
util = require ('util');


const pl = require('mysql/lib/Pool');
pl.prototype.getConnectionAsync =
    util.promisify (pl.prototype.getConnection);
pl.prototype.queryAsync =
    util.promisify (pl.prototype.query);

const cn = require('mysql/lib/Connection');
cn.prototype.queryAsync =
    util.promisify (cn.prototype.query);

(async () => {

    let con = await pool.getConnectionAsync ();
    await con.queryAsync ('SELECT .....');
    con.release ();

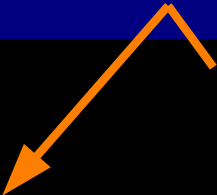
    let zs = await pool.queryAsync ('SELECT.....');
})();
```



Klasické rozšírenie tried o nové metódy. Je to univerzálnejšie pre všetky spojenia a pooly.

„Súčasné“ vykonanie SQL dotazov


```
async function Moja (a, b) {  
  try {  
    let p1 = pool.queryAsync ('SELECT 1.....');  
    let p2 = pool.queryAsync ('SELECT 2.....');  
    let p3 = pool.queryAsync ('SELECT 3.....');  
  
    let [p1, p2, p3] = await Promise.all ([p1, p2, p3]);  
  
    return p1[0].name + p2[0].school + p3[0].grade;  
  }  
  catch (err) {  
    console.log (err);  
    throw new Error ('Chyba v Moja');  
  }  
}
```



Každý dotaz pôjde do iného spojenia v banke spojení. Samozrejme, že banka musí mať limit aspoň 3.

Sessions v Node.js

- riešené cez **Cookies**
 - „malé premenné“ uložené v prehliadači
 - sú vždy v hlavičke „**Cookie:**“ HTTP požiadavky
 - v HTTP odpovedi možno dať nastaviť „premennú“ cez hlavičku „**Set-Cookie:**“
 - možno im nastaviť životnosť (dočasné, permanentné, ...)
 - nezávislé sady pre každú doménu-server
- **session identifikátor** uložený v dohodnutej cookie premennej
 - identifikuje klienta – prehliadač
- server
 - prideluje jednoznačné **sessionID**
 - podľa **sessionID** ukladá dáta klienta na serveri
 - po obnovení spojenia opäť identifikuje klienta cez **sessionID**



Server sa tak dozvie aktuálnu hodnotu pri každej HTTP požiadavke. Pozor pri WebSocket protokole už nie!

modul **express-session**

- určené pre express, t.j. pre HTTP server
- dáta pre sessionId sú v nejakom úložisku
 - napr.: súbor, databáza
 - napr.: modul **express-mysql-session**
- využije sa middleware funkcia
- inštalovanie modulov

```
$ npm install express-session  
$ npm install express-mysql-session
```

```
const express = require('express');
const app = express ();
const server = require('http').createServer (app);

const session = require('express-session');

// Vytvorenie úložiska session-ov
const MySQLStore = require('express-mysql-session') (session);
const sessionStore = new MySQLStore({
  host: 'localhost',
  port: 3306,
  user: 'ferko',
  password: 'heslo',
  database: 'Sessions',
  createDatabaseTable: true // Ak nie je, vyrobí tabuľku v DB.
});

// Funkcia vyparsuje cookie a pohľadá dané session v úložisku
sessionMiddlewareFunc = session ({
  key: 'session_cookie_name',
  secret: 'session_cookie_secret',
  store: sessionStore,
  resave: true,
  saveUninitialized: true
});

app.use ('/', sessionMiddlewareFunc);
server.listen (9000);
```

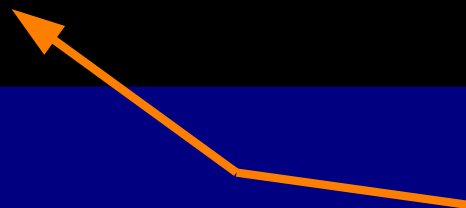
Vyrobí sessionID a aj záznam v databáze pre každého nového klienta. Klientovi odošle set-cookie s patričným sessionID.

Toto je dôležité pri kooperácii so **socket.io**, ktoré už nevie poslať set-cookie klientovi v prípade, že klient ešte nemá pridelené sessionID a záznam v databáze od express-u.

Použitie v express-e

- v ľubovoľných pravidlách
 - pribudne vlastnosť `req.session`

```
app.get('/', (req, res) => {  
  let sess = req.session;  
  sess.user = 'Janko';  
  
  res.sendFile (__dirname+'/index.html');  
  
  // sess.save(); // Automaticky sa zavolá pri HTTP odpovedi.  
});
```



Aktualizuje sa hodnota v úložisku. Je to dôležité, lebo pri ďalšej požiadavke sa opäť načítava hodnota z úložiska.

Využitie sessions v socket.io

- aby aj socket.io mohol **pristupovať** ku sessions
 - zmena sa automaticky neprenesie do úložiska

```
// Použitie middleware funkcie na sessions v socket.io
io.use ((socket, next) => {
  sessionMiddlewareFunc(socket.request, socket.request.res, next);
});

io.on ('connect', socket => {
  let req = socket.request;

  socket.on ('login', usr => {
    req.session.reload (err => {

      let sess = req.session;
      sess.user = usr;
      sess.save ();

    });
  });
});
```

Vytvorenie **socket.io** spojenia predchádza jedna výmena podľa HTTP protokolu. T.j. možno využiť tú istú middleware funkciu, aby vyparsovala cookie a pohľadala údaje z úložiska podľa sessionId.

Po zmene je nevyhnutné dať pokyn na uloženie údajov do úložiska. Tu by bol problém, ak by ešte nebolo pridelené sessionId a neexistoval by záznam v úložisku. Klienta už nemožno informovať cez Set-Cookie!

V callbackoch udalostí treba znovu načítať údaje, lebo sa medzičasom mohli zmeniť cez HTTP požiadavky!

modul `express-socket.io-session`

- **automatické** prenesenie zmeny do úložiska
 - možno určiť explicitné uloženie alebo `autoSave`

```
$ npm install express-socket.io-session --save
```

```
const sharedsession = require("express-socket.io-session");  
  
io.use (sharedsession(sessionMiddlewareFunc, {autoSave: true}));  
  
io.on ('connect', socket => {  
  let hs = socket.handshake;  
  let sess = hs.session;  
  sess.user = 'Janko';  
  
  // sess.save();  
});
```

Trochu upraví-obalí
middleware
funkciu.

Na konci udalosti sa automaticky
realizuje uloženie.

Problém je ak sa aktuálny obsah
načítava cez `sess.reload()`.
Callback sa spustí až po skončení
udalosti. Vtedy treba `sess.save()`.

Ďakujem za pozornosť