21 Rakenduse andmebaasiga ühendamine

Andmebaasi poole pöördumiseks peab olema kasutusel mingi kliendi-poolne ajur (driver), mis kasutab üht või enamat võrguprotokolli. Sisemise „keelena“ kasutab SQL server TDS (Tabular Data Stream) protokolli. Ajuri eesmärgiks on anda lihtsustatud ligipääs mõnele tehnoloogiale või keelele.

Levinumad:

- ODBC – klassikaline Windowsi andmebaasi liidestus
- .NET Native Client – kaasaegne .NET platvormil kasutatav liides
- JDBC – Java-põhine andmebaasi liides

Tänu standardiseeritud ajuritele saab kasutada andmete kuvamiseks ja analüüsimiseks väga paljusid rakendusi, nagu näiteks Microsoft Excel, Tableau, jpm

Kõik need liidesed annavad meile võimaluse otse andmebaasiga ühenduda (klient-server lahendused). Taoline variant sobib, kui rakendus töötab kohtvõrgus. Mida aga teha siis, kui me tahame anda ligipääsu kaugel asuvatele (üle Interneti kasutatavatele) rakendustele?

Selleks tuleb luua mõni vahekiht, mis transleerib avalikus internetis kasutatava ühenduse andmebaasi enda poolt kasutatavale kujule (ehk siis mõni ülaltoodud liidestustest). Sellisel moel saame omale **mitmekihilise rakenduste arhitektuuri** (*multi-tier architecture*).

Teiste sõnadega saame omale rakendusliidese (API), mis võib olla ellu viidud mitut moodi, muuhulgas:

- Web + AJAX
- XML Web Service
- REST
- OData, GraphQL, gRPC, JSON-RPC vms

Me käsitleme siinkohal REST tüüpi rakendusliidest, kuna seda saab kasutada nii oma veebirakendusest, kui ka suvalistest välistest allikatest (mobiili-äpp, partneri rakendus jne)

REST mudel on muutunud ka de-facto standardiks üheleheliste rajenduste (*Single Page Application (SPA)*) loomise juures, kus andmeid tuuakse AJAX päringute abil peamiselt just läbi REST liidese.

REST ehk REpresentational State Transfer on tarkvara arhitektuuri stiil, mis kasutab ära HTTP protokolli võimalusi ja lihtsustab seeläbi andmetega ümber käimist

HTTP protokolli meetodid (verbid):

- GET – tagastab andmed (CRUD variandis R – Read)
- POST – postitab uusi andmeid (CRUD variandis C – Create)
- PUT – postitab muutusi (CRUD variandis U – Update)
- DELETE – eemaldab andmeid (CRUD variandis D – Delete)
- HEAD – tagastab päised nii, nagu siis, kui oleks andmed ka
- OPTIONS – kirjeldab kommunikatsiooni valikuid. Kasutatakse ka CORS (Cross-Origin Resource Sharing) päringute juures
- PATCH – lubab postitada osalisi andmeid, et neid (kasvõi hulgi) muuta

Kui XML Web Services ja paljud teised on funktsioonipõhised (RPC - *Remote Procedure Call*), siis REST on ressursipõhine, mis tähendab seda, et URL kaudu viidatakse ressurssidele ja HTTP meetodiga väljendatakse soovitud operatsiooni.

Meetod	Ressursi URL	Seletus
GET	/<resource>	tagastab kõik elemendid
POST	/<resource>	postitab uue elemendi
PATCH	/<resource>	värskendab mitut elementi
GET	/<resource>/<identifier>	tagastab ühe elemendi
PUT	/<resource>/<identifier>	värskendab elemendi andmeid
DELETE	/<resource>/<identifier>	kustutab elemendi
PATCH	/<resource>/<identifier>	värskendab üht elementi

Kui näiteks XML WebServices puhul on traditsioonilise RPC (*Remote Procedure Call*) stiilis lähenemisel iga päringu variandi jaoks on oma funktsioon…

```
GetTweets()                    // Kõik postitused
GetRepliesFor(int tweetID)     // Konkreetse postituse vastused
GetRetweetsFor(int tweetID)    // Konkreetse postituse taaspostitused
```

…siis REST'i puhul saame palju lihtsama liidese

```
/tweets                        // Kõik postitused
/tweets?postingType = 2        // Kõik vastused
/tweets/1423                   // konkreetne postitus
/tweets/1423/replies           // konkreetse postituse vastused
```

Mis aga teha siis, kui me sooviks REST tüüpi liidese veelgi paindlikumaks muuta ning URL-i kaudu ära määrata veel mitmeid asju, näiteks:

- tagastavate väljade nimed (et mitte ülearust infot tagastada)
- andmete sorteerimise
- andmete filtreerimise mingite tunnuste alusel (sarnaselt „postingType=2“ variandile)
- andmete tagastamise lehekülgede kaupa
- alam-ressursside kaasamise

Siin tuleb appi Open Data Protocol (OData), mis

- standardiseerib URL kasutuse (mis on pisut erinev tüüpilisest REST kujust)
- lubab URL kaudu muuta tagastatavate andmete projektsiooni ja seotud andmeid kaasata ($select)
- lubab andmeid sorteerida ($orderby)
- lisab võimaluse URL kaudu päringu tingimusi esitada ($filter)
- võimaldab lehekülje kaupa andmeid edastada ($take, $skip)
- alates v4-st võimaldab ka vajadusel RPC stiilis käske välja kutsuda

```
GET http://services.odata.org/v4/TripPinServiceRW/People?$top=2 &
    $select=FirstName, LastName & $filter=Trips/any(d:d/Budget gt 3000)
```

Siit on näha, et me saame mitte ainult eelnevalt paika pandud päringuid käivitada, vaid ka enda jaoks vajalikke ad-hoc kohandusi teha. Samas võivad sellised täiendused serverile nii positiivset kui negatiivset mõju avaldada, kuid API lihtsus ja paindlikus ilmselgelt kasvab.

Eksisteerib veel üks põnev API loomise viis: **GraphQL** (http://graphql.org/). Selle loomise tingis asjaolu, et kuigi REST on võrreldes RPC stiilis funktsioonidega palju paindlikum, on tema abil raske ja ebaefektiivne pärida graafikujulisi sõltuvusi sisaldavaid andmeid. Seni kuni meid huvitavad ressursid ja nende alamressursid, siis saame REST / OData abil kenasti vastused. Keerukamate vajaduste puhul tuleb aga päringuid kombineerima hakata – ehk tuleb teha mitu ringi (*roundtrip*) server poole.

Kirjelda andmed	Küsi, mida vaja on	Saa tagasi ennustatav tulemus

`type Project {`
`  name: String`
`  tagline: String`
`  contributors:`
`[User]`
`}`
	`{`
	`  project(name:`
	`"GraphQL") {`
	`  tagline`
	`  }`
	`}`
		`{`
		`  "project": {`
		`  "tagline": "A query`
		`language for APIs"`
		`  }`
		`}`

Kasutades taolist lähenemist saame kokku hoida nii päringute arvult (1 päring mitme asemel) kui ka andmemahult (tagastame vaid need andmed, mida tõesti vaja on).

Näitlik rakendusliides: https://graphqlzero.almansi.me/.

Üks paljudest firmadest, mis seda formaati toetab, on GitHub (https://githubengineering.com/the-github-graphql-api/).

Kui API on peamiselt sisekasutuseks, eriti mikroteenuste juures, siis on võimalik valida ka **gRPC** (https://grpc.io/). See on Google poolt loodud protokoll, mis kasutab kliendi ja serveri vahel suhtlemiseks Protocol Buffers formaadis kirjeldatud pakette. gRPC baseerub HTTP/2 protokollil ning lubab eriti efektiivset suhlust. Lisaks pakub gRPC ka garantiisid ajastusele, rikkalikumat veahaldust, andmevooge (stream), mõlemas suunas suhtlust jpm.

22 Rakenduste ligipääsu lubamine andmebaasile

Tüüpiliselt ei asu vahekihi rakendus samas masinas kus andmebaas. See tähendab, et me peame tekitama rakenduse ja andmebaasi vahele turvalise kanali.

Ligipääsu lubamine käib üldiste SQL Serveri turvareeglite alla (https://docs.microsoft.com/en-us/sql/relational-databases/security/security-center-for-sql-server-database-engine-and-azure-sql-database). Täpsemalt käsitleb see võrguprotokollide lubamist, tulemüüri seadistamist, teenuste lubamist jne (https://docs.microsoft.com/en-us/sql/relational-databases/security/securing-sql-server).

Võrguprotokollide seadistamine

MS SQL Server suudab suhelda mitmete võrguprotokollide kaudu:

- Jagatud mälu (*Shared memory*) – vaikimisi lubatud
- *Named Pipes*
- *TCP/IP*

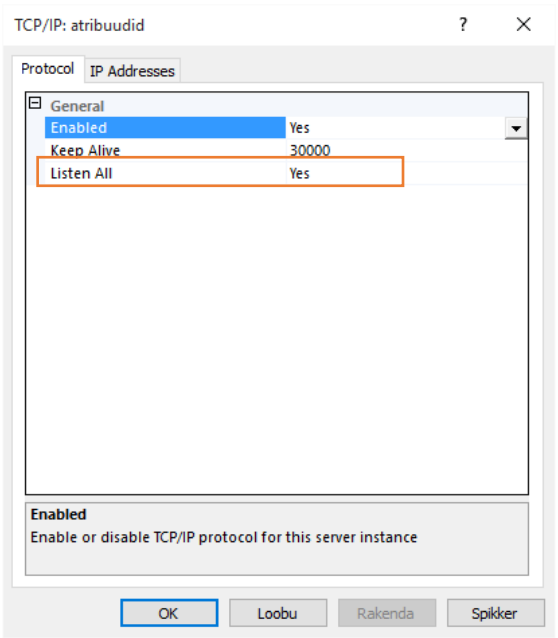
Ainsaks vaikimisi lubatud protokolliks on jagatud mälu, mida ei saa juba definitsiooni kohaselt väljastpoolt üht masinat kasutada.

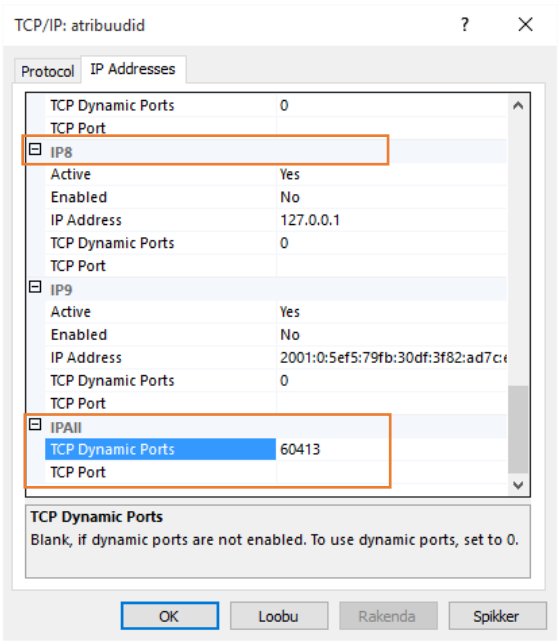
Named Pipes võib kasulikuks osutuda nii ühe masina kui kohtvõrkude puhul ja ainus seadistatav väärtus on toru nimi, näiteks:

```
\\.\pipe\MSSQL$SQLEXPRESS\sql\query
```

Kõikidel teistel puhkudel on hea kasutada TCP/IP protokolli

Listen All – määrab ära kas server kuulab ühendusi ja päringuid kõikide arvutis olevate või üksnes valitud võrguliideste pealt





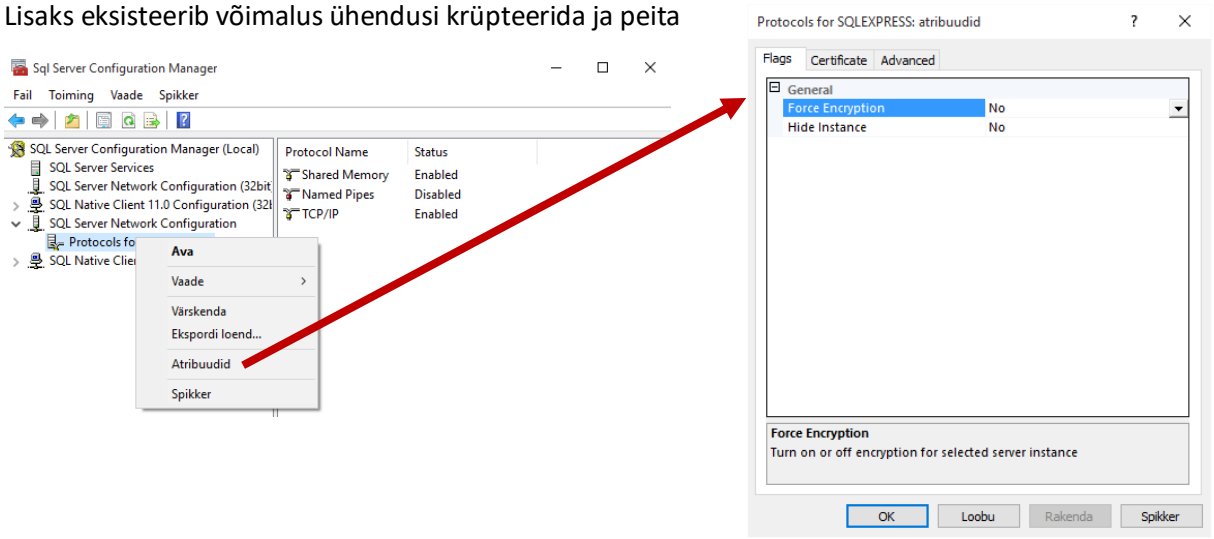
IP1…IPn – konkreetsed võrguliidesed.

IPAll – seadistused, mis kehtivad üle kõigi.

TCP Dynamic Ports – kui 0, siis määrab ise juhusliku pordi numbri; kui tühi, siis fikseetitud vaadatakse porti. *TCP Port* –

Kui määratud, kasutab seda porti eeldusel, et dünaamiline kasutusel pole (vaikimisi 1433).

Lisaks eksisteerib võimalus ühendusi krüpteerida ja peita

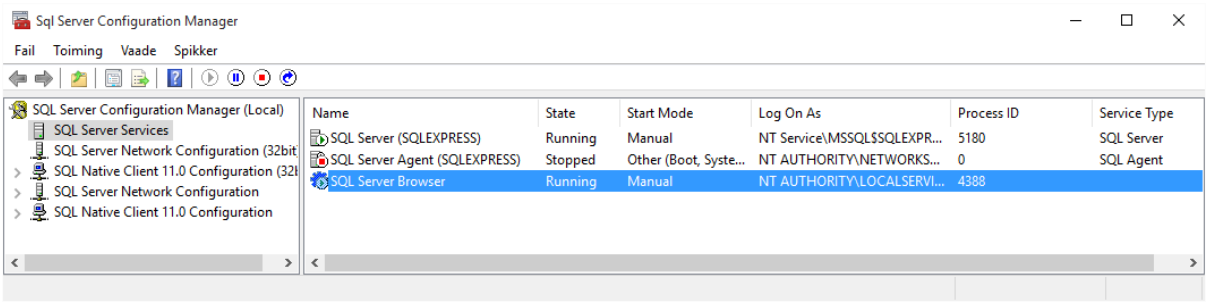


SQL Server Browser teenuse lubamine

Nagu me juba näinud oleme, suudab SQL Server edukalt töötada ka ilma selle teenuseta. Samas on teisest masinast temale ligi saamine raskendatud kuna

- ei tea, millist võrguprotokolli kasutada
- ei tea, millise pordi all ta kättesaadav on
- ei tea, millise instantsi poole me pöörduda tahame

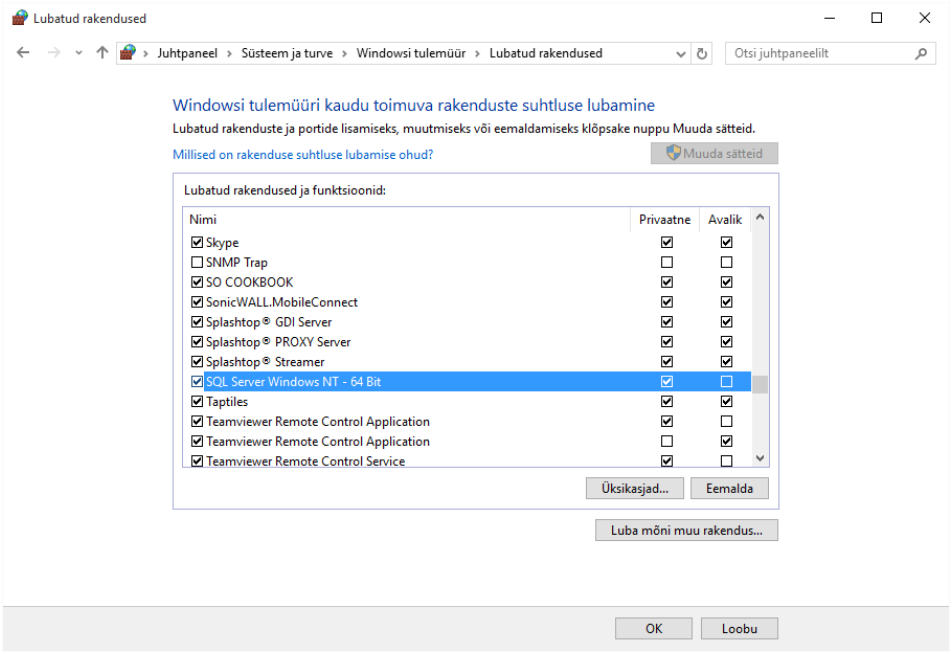
Neist probleemidest aitab välja *SQL Server Browser* teenus. Kui me seda teenust kasutame, siis serveris peaks muidugi mõlemad „Auto Start“ seadistuses olema.



Üks märkus kontode kohta, milliste all teenused jooksevad: nagu pildilt näha, jookseb *SQL Server* ise spetsiaalse konto all ja *SQL Server Browser* lokaalse süsteemiteenuse konto all. Põhjus, miks SQL server ise eraldi konto all jookseb, on selleks, et vähendada võimalikust serveri häkkimisest tulenevat turvariski – sellel kontol puuduvad süsteemiadministraatori õigused.

Tulemüüri reeglite lisamine

Kuigi tüüpiliselt kannab SQL Serveri paigaldamise protsess hoolt ka selle eest, et sobilikud tulemüüri reeglid paika saaks, võib neid teinekord olla vaja juurde lisada. Kui kasutusel on *SQL Server Browser*, siis võib olla vaja lubada ligipääs ka sellele.



23 Node.js rakenduskihi loomine

Selleks, et katsetada anndmebaasi toimimist pisut realistlikumas situatsioonis, tuleb meil vaadelda andmete liikumist läbi terve ahela: andmebaas→vahekiht→klient (brauser vms).

Selleks loome vahekihi Node.js abil REST-stiilis Web API-na ning kliendi rolli täidab kas PostMan rakendus (või mõni muu sarnane) või Chrome brauser.

Kokkuvõttes on meil vaja järgmisi asju:

- Node.js, mille saab alla laadida www.nodejs.org. Üldjuhul võib võtta kõige uuema versiooni, kui just mõni muu rakendus teisiti ei sätesta
- Koodi kirjutamiseks on vaja sobivat tekstiredaktorit – näiteks Brackets (www.brackets.io), Visual Studio Code (code.visualstudio.com) või mõnda muud sarnast.
- Node juhtimiseks on vaja käsurida. See võib olla kas Command Prompt või Windows PowerShell
- Kliendi rollis on brauser, PostMan (www.getpostman.com), CURL või mõni muu sarnane rakendus, millega saab salvestada, jooksutada ja testida REST rakendusliideseid.

Testimaks Node töökorras olekut, saab käsurealt anda käsu:

```
>node --version
```

Selleks, et me saaks oma vaheliidest looma hakata, läheme mõnda sobivasse kataloogi ning loome sinna alamkataloogi „miniinsta“. Seejärel liigume sinna sisse ning anname käsu „npm init“:

```
>mkdir miniinsta
>cd miniinsta
>npm init
```

NPM `Init` käsk küsib meie käest terve rea küsimusi, enamikule milledest võib vaikimisi väärtuse jätta. Kirjelduse ja autori koha peale võite ka soovi korral midagi kirjutada. Üks oluline küsimus on rakenduse sisendpunkti (entry point) kohta – vaikimisi on selleks „index.js“ fail, kuigi vahest kasutatakse selles rollis ka „app.js“-nimelist faili. Tulemuseks on jooksvasse kataloogi lisatud `package.json` fail:

```
{
  "name": "miniinsta",
  "version": "1.0.0",
  "description": "Pisike vaheliides Mini-Insta andmebaasile",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "PP",
  "license": "ISC"
}
```

Järgmine samm on avada rakenduse kataloog valitud tekstiredaktoris ning lisada laiend Node rakenduste lihtsamaks kasutamiseks.

- VS Code puhul on põhiline debugger juba sisse ehitatud, kuid soovi korral saab lisada ka muid laiendusi (näiteks „VS Code for Node.js - Development Quickstart Pack“, „Code Runner“), kuigi peamine asi – debugger – on juba uuema VS code-ga kaasa pandud
- Kui me kasutame Brackets redaktorit, siis tuleks lisada „Node.js bindings“ laiendus (File→Extension Manager menüüst)
- Käsurealt kirjutades: „`node index.js`“

Esimene samm peale neid on `index.js` faili loomine samasse kataloogi. Kuna Node on üldotstarbeline JavaScripti skriptide jooksutamise keskkond, siis millegi kasuliku tegemiseks on tarvis talle lisada mooduleid. Selleks, et moodulit koodis kasutada, tuleb see esmalt faili importida:

```
var muutuja = require('mooduli_nimi');
```

Oma veebiliidese loomiseks kasutame „Express“-nimelist moodulit (https://expressjs.com/) ning tema kõige lihtsam kood on ainult 5-realine.

```
// Module requires
let express = require('express');

// Instatiate application instance
let app = express();

// Serveri initsialiseerimine
let server = app.listen(3000, function() {
    console.log('Listening on port 3000');
});
```

Me saame seda kohe katsetada, kui redaktoris koodi käivitame (Brackets'i puhul Alt+N; VS Code puhul F5 ja valida Node.js keskkond). Kui me seda praegu teeme, siis saame vea:

```
module.js:472
    throw err;
    ^
Error: Cannot find module 'express'
    at Function.Module._resolveFilename (module.js:470:15)
    at Function.Module._load (module.js:418:25)
    at Module.require (module.js:498:17)
    at require (internal/module.js:20:19)
    at Object.<anonymous> (C:\Dev\DBCourse\miniinsta\index.js:2:15)
    at Module._compile (module.js:571:32)
    at Object.Module._extensions..js (module.js:580:10)
    at Module.load (module.js:488:32)
    at tryModuleLoad (module.js:447:12)
    at Function.Module._load (module.js:439:3)
Program exited with status code of 1.
```

VS Code väljund on pisut erinev, kuid tulemus on sama. Mis küll viga võiks olla?

Viga tuleb sellest, et me oleme küll avaldatud soovi Express moodulit kasutada, kuid me pole seda oma rakendusele veel lisanud. Moodulite lisamiseks saame kasutada „`npm install`" käsku, mille üldkuju on järgmine:

```
>npm install [-g] <package> [--save]
```

Selleks, et me saaks ikkagi Express mooduli rakendusele lisada, anname järgmise käsu:

```
>npm install express --save
```

Peale seda ilmub `packages.json` faili uus osa nimega „dependencies". Samuti tekitatakse meile kataloog nimega „`node_packages`", kuhu kõik rakendusega seotud moodulid paigutatakse. Seda viimast ei ole vaja koodihaldusesse lisada kuna me saame igal ajal selle sisu taastada kasutades käsku

```
>npm install
```

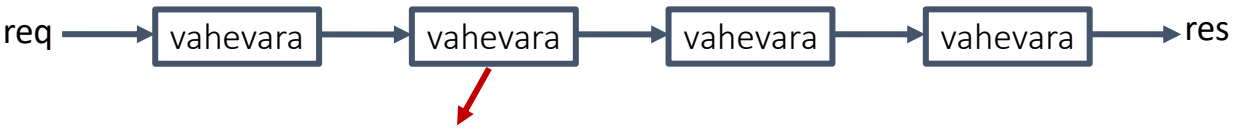
Kui me oma rakenduse praegu käima laseme, siis käivitatakse meile veebiserver pordi 3000 peal (http://localhost:3000), ning tema poole võime pöörduda tavalise brauseriga. Kui me seda teeme, siis saame vastuseks:

```
Cannot GET /
```

Vaatamata sellele teatele näeme, et server siiski vastab meile.

Rakenduse sisu loomine

Katsume oma rakendust nüüd veidi sisukamaks muuta. Selleks aga on vaja esmalt tutvustada vahevara (*middleware*) olemust. Express mooduli seisukohast on tegu päringuid töötlevate plokkide ahelaga.



```
// req – request, res – response, next – ahela järgmine samm
function (req, res, next) {
  // töö, mis kasutab req objekti
  if (!ok)
      return res; // Kui ei õnnestunud või vastus käes, tagastame tulemuse

  next(); // Muudel puhkudel jätkame vahevara ahelat
}
```

Sisuliselt tähendab see seda, et kui päring tuleb ühest otsast sisse, siis püütakse erinevate funktsioonide poolt vaadata, milline seda teenindada oskab. Kui sobivat ei leita, saame veateate, mida juba korra nägime.

Express mooduli kasutamine päringutele vastamiseks

Tuues selle loogika üle veebi- või REST maailma, on meil vaja tegeleda URI teekondade/ressurssidega:

```
...
var app = express();

// Juur-kataloogi haldamine
app.get('/', function(req, res) {
    res.send('<h1>Hello</h1>');
});

// Vaikimisi vastus, kui muid teekondi ei leitud
app.get('*', function(req, res) {
    res.status(404).send('Invalid route');
});

// Serveri initsialiseerimine
var server = app.listen(3000, function() {
    console.log('Listening on port 3000');
});
```

Teekonna http://localhost:3000 puhul saame vastuseks tervituse ja http://localhost:3000/suva peale saame „Invalid route“ – ehk tulemuse annab meie vaikimisi haldur, kuna „suva“ teekond polnud kaetud.

Enne, kui meie rakendus väga keerukaks ja haldamatuks muutub, viime haldur-funktsioonid eraldi faili nimega `routes.js`. *Siinkohal tasub mainida, et kuigi Express moodul omab omaenda teekondade haldamise süsteemi, kasutame hetkel parema mõistmise mõttes iseenda tehtud haldureid!*

Sellesse uude faili lisame hetkel 2 funktsioni sellisel kujul (exports muudab funktsioonid mooduli kaudu kättesaadavaks):

```
exports.index = function(req, res) {
    res.send('<h1>Hello</h1>');
}
```

```
exports.default = function(req, res) {
    res.status(404).send('Invalid route');
}
```

Selleks, et me saaks nüüd nendele funktsioonidele ka peafailist viidata, tuleb `index.js` faili lisada uus `require()` ning muuta praegusete funktsioonide definitsioonid teises failis asuvate viidete vastu.

```
let routes = require('./routes');

app.get('/', routes.index);
app.get('*', routes.default);
```

Soovides oma rakendusele Web API liidest luua, siis tüüpiliselt on selleks kaks lähenemist:

- Lisades ülejäänud URI ette /api
  http://localhost:3000/api
- Kasutades eraldi alamdomeeni nime
  http://api.domeen.com

Seda viimast varianti kasutab näiteks ka Instagram ise – `api.instagram.com` (https://www.instagram.com/developer/endpoints/).

Näiteks saaksime siis kasutajate nimekirja tagastamiseks luua sellise teekonna:

```
// Meie spetsiifiline juur-kataloogi haldamine
app.get('/', routes.index);

// Rakenduskesksed teekonnad
app.get('/api/users, routes.users);

// Vaikimisi vastus, kui muid teekondi ei leitud
app.get('*', routes.default);
```

ja muidugi peame `routes.js` faili lisama ka vastava halduri

```
exports.users = function(req, res) {
    res.send('<h1>Kasutajad</h1>');
}
```

Side loomine MSSQL andmebaasiga

Võimaldamaks pärida andmeid andmebaasist, tuleb meil kasutusele võtta mõni sobiv moodul. Üks taolistest on „mssql“ (https://www.npmjs.com/package/mssql):

```
>npm install mssql --save
```

Selle asemel, et kogu andmebaasiga seotud koodi `routes.js` faili kaasata, loome uue faili nimega `sql.js`, kuhu hakkame lisama andmebaasiga ümber käimise abifunktsioone. Esmalt tuleb meil seal ära kirjeldada ühenduse seadistus – kasutaja, serveri nimi, port ja andmebaas.

Seejärel on meil vaja andmebaasi külge ühenduda. Antud juhul kasutame ise-käivituvat funktsiooni ühenduse tegemiseks isegi siis, kui kliendi poolt serverile veel ühtegi päringut tehtud pole. Eks seda

saab organiseerida ka muud moodi, kuid hetkel siis nii. Edasi tulevad funktsioonid päringute tegemiseks ja kõige lõpuks registreerime halduri vigade kinni püüdmiseks.

```
let mssql = require('mssql');

let config = {
    user: 'testapp',
    password: 'testapp',
    server: '127.0.0.1\\sqlexpress',
    //port: 1433, // Should not set this when connecting to a named instance
    database: 'MiniInsta',
    connectionTimeout: 5000,
    options: {
        encrypt: false // needed when connecting to azure managed database
    }
}

let pool; // Koht, mis salvestab ühenduse info

(async function() {
    try {
        pool = await mssql.connect(config);

        console.log('Connected to DB');
    } catch (err) {
        // Log errors
        console.log('ERROR: ' + err);
    }
})()

exports.querySql = function(query, onData, onError) {
    try {
        pool.request()
            .query(query)
            .then(result => {
                // data returns:
                //   data.recordsets.length
                //   data.recordsets[0].length
                //   data.recordset
                //   data.returnValue
                //   data.output
                //   data.rowsAffected

                if (onData !== undefined)
                    onData(result);
            })
            .catch(error => {
```

```
                if (onError !== undefined)
                    onError(error);
            });
    } catch (err) {
        // Log errors
        if (onError !== undefined)
            onError(err);
    }
}

mssql.on('error', err => {
    console.log('Error with MSSQL: ' + err);
})
```

Edasi asendame teekondade failis kasutajate funktsiooni järgneva koodiga:

```
let sql = require('./sql');

exports.users = function(req, res) {
    let query = 'select * from dbo.[User];

    let result = sql.querySql(query, function(data) {
        if (data !== undefined)
        {
            console.log('DATA rowsAffected: ' + data.rowsAffected);
            res.send(data.recordset);
        }
    }, function(err) {
        console.log('ERROR: ' + err);
        res.status(500).send('ERROR: ' + err);

    });
}
```

Kui me nüüd teeme päringu aadressile http://localhost:3000/api/users, siis saame tagasi järgnevad andmed:


```
[{"ID":1,"Kasutajanimi":"jcardon0","Nimi":"Jacinda Cardon","Website":"http://tamu.edu","SuguID":3,"Kirjeldus":"Quisque porta volutpat erat.","Email":"jcardon0@delicious.com","LisamiseAeg":"2017-11-13T19:42:55.883Z","Parool":"xKBvAAtIwz","PildiUrl":"http://dummyimage.com/134x204.jpg/cc0000/ffffff"},
{"ID":2,"Kasutajanimi":"gblaxill1","Nimi":"Giff Blaxill","Website":"https://google.com.br","SuguID":2,"Kirjeldus":"Vestibulum ac est lacinia nisi venenatis tristique. Fusce congue, diam id ornare imperdiet, sapien urna pretium nisl, ut volutpat sapien arcu sed augue. Aliquam erat volutpat.","Email":"gblaxill1@census.gov","LisamiseAeg":"2017-11-13T19:42:55.888Z","Parool":"Yt82Kr","PildiUrl":"http://dummyimage.com/135x211.bmp/dddddd/000000"},
{"ID":3,"Kasutajanimi":"gkrollmann2","Nimi":"Garnette Krollmann","Website":"http://slashdot.org","SuguID":3,"Kirjeldus":"Praesent blandit lacinia erat. Vestibulum sed magna at nunc commodo placerat. Praesent blandit.","Email":"gkrollmann2@nyu.edu","LisamiseAeg":"2017-11-13T19:42:55.894Z","Parool":"xvTjvxVB","PildiUrl":"http://dummyimage.com/173x180.bmp/dddddd/000000"},
```


Ja neid andmeid on palju – täpsemalt kõik meie 200 kasutajat. Mis siis, kui me soovime tagasi saada vaid ühe kasutaja infot? Sellisel juhul oleks loogiline lisada teekonnale ID väärtus: http://localhost:3000/api/users/12. Selleks, et taolist asja saavutada, tuleb meil täiendada nii teekonna kirjeldust, kui ka teekonna halduri funktsiooni. Esmalt siis teekonna kirjeldus:

```
app.get('/api/users/:id?', routes.users);
```

Siin on näha teekonnale lisatud "`:id?`" osa. Mida see tähendab?

- `:id` – koolonile järgnevat teksti käsitletakse parameetrina, mille väärtus URList välja loetakse ning Node Expressi rakenduses kättesaadavaks tehakse
- `?` – märgib, et parameeter on valikuline – ehk võib ka määramata olla

Haldurisse tuleb lisada kontroll `req.params.id` olemasolu kohta. `req.params` on Express'i päringu omadus, mis annab ligipääsu teekonna kirjeldusse lisatud parameetritele. „`id`" on seesama nimi, mille me eelnevalt sinna kirjutasime.

```
exports.users = function(req, res) {
    let query = 'select * from dbo.[User];

    // If there's an ID passed along
    if (typeof(req.params.id) !== 'undefined') {
        query = query.concat(' where id=' + req.params.id);
    }

    var result = sql.querySql(query, function(data) {
        if (data !== undefined)
        {
            console.log('DATA rowsAffected: ' + data.rowsAffected);
            res.send(data.recordset);
        }
    }, function(err) {
        console.log('ERROR: ' + err);
    });
}
```

See lisatud kood kontrollib, kas `req.params.id` parameeter on defineeritud või mitte. Kui ta on olemas, siis lisatakse algsele `query` muutujas toodud päringule ka vastav where tingimus. Tulemus on just see, mida me lootsime näha:


```
[{"ID":12,"Kasutajanimi":"bparamoreb","Nimi":"Binnie Paramore","Website":"https://zdnet.com","SuguID":2,"Kirjeldus":"Curabitur
convallis. Duis consequat dui nec nisi volutpat eleifend. Donec ut dolor. Morbi vel lectus in quam fringilla
rhoncus.","Email":"bparamoreb@tamu.edu","LisamiseAeg":"2017-11-
13T19:42:55.935Z","Parool":"a2u2VpAgVbtG","PildiUrl":"http://dummyimage.com/151x104.jpg/dddddd/000000"}]
```


Mida aga peaks tegema siis, kui me tahame vastu tulla SEO soovitustele, kus URL peaks olema inimloetav. See tähendab, et me peaks näiteks püüdma ID=12 asemel kasutada hoopis kasutajanime – samamoodi nagu ka Instagram seda teeb (https://www.instagram.com/vakufoto/). Seega püüame oma APIs samamoodi teha (http://localhost:3000/api/users/bparamoreb). Kui me aga seda URL-i praegu välja kutsume, saame veateate:

```
RequestError: Invalid column name 'bparamoreb'.
```

Eks see ole ka mõistetav, sest praegu püütakse lisada päringule „where ID = bparamoreb", aga kuna puuduvad jutumärgid, arvab andmebaas, et ju siis peab tegu olema tabeli veeruga, kuid sellist asja ta ei leia. Isegi kui ta tuvastaks, et tegu on stringiga, võrdleme me ikkagi valet veergu – ehk praeguse ID asemel peaks me võrdlema seda „username" veeruga. See tähendab, et me peame kasutajanime jaoks looma teistsuguse päringu.

Kuidas me seda aga teekonda kirjeldama peaks? Kas teekonna definitsiooni võib jätta samale kujule või peaks seda ka muutma?

Põhimõtteliselt saab minna mõlemat teed pidi. Püüdes teekonna kirjeldusi täiendada, saame:

```
app.get('/api/users/:id([0-9]{1,9})?', routes.usersByID);
app.get('/api/users/:username?', routes.usersByUsername);
```

Siin defineerime esmalt kitsendatud variandi, kus määrame ära, millised sümbolid meil esineda tohivad. Kuna numbreid on lihtsam kirjeldada, siis juhul kui numbrid on vahemikus 0..999999999, valitakse haldur „kasutajadIDJargi“. Kui tegu on mingite muude sümbolitega, siis „kasutajadKasutajanimeKaudu“.

Teine võimalus, mida me hetkel ka kasutama hakkame, on muuta üksnes halduri id parameetri kontrolli koodi:

```
// If there's an ID passed along
if (typeof(req.params.id) !== 'undefined') {
    if (isNumber(req.params.id)) {
        query = query.concat(' where id=' + req.params.id);
    } else {
        query = query.concat(' where Username=\'' + req.params.id + '\'');
    }
}
```

Kui meil on id parameeter kaasa antud, siis kontrollime, kas tegu on numbriga. Kui jah, kasutame varasemat where tingimust, kui aga mitte, siis kasutajanime. Selleks, et meie test töötaks, on vaja defineerida ka isNumber funktsioon:

```
function isNumber(n) {
    return !isNaN(parseFloat(n)) && isFinite(n);
}
```

Kui me nüüd oma päringu käivitame, saame ka soovitud tulemuse.

Edaspidi katsume kõikides kohtades, kus midagi kasutaja kaudu soovitakse saada, teha seda nii ID kui Username kaudu. Näiteks:

- /api/users/:id/followees
- /api/users/:id/followers
- /api/users/:id/posts
- /api/users/:id/posts/:post_id
- /api/users/:id/posts/:post_id/comments

Teised päringud

Nüüd, kus oleme kasutajate päringu kaudu erinevad stsenaariumid läbi käinud, on aeg lisada ka ülejäänud päringud. Selleks võtame need, mis said eelnevalt ülesandena valmis tehtud:

- Esilehe päring
  /api/frontpage
- Profiili lehe päring
  /api/profile/:id

- Postituse detailid
  /api/post/:id
- Analüütilised arvud ühe andmehulgana
  /api/stats
- Top 10 kasutajat, kellel on kõige rohkem jälgijaid
  /api/stats/top10/followedusers
- Kasutajaks registreerimiste arv päevade kaupa
  /api/stats/registrations
- Kasutajate jagunemine sooliselt
  /api/stats/genderdivision

Loome esmalt teekondade definitsioonid:

```
app.get('/api/frontpage, routes.frontpage);
app.get('/api/profile/:id', routes.profilePage);
app.get('/api/posts/:id', routes.postDetails);
app.get('/api/stats', routes.statistics);
app.get('/api/stats/top10/followedusers',
          routes.top10FollowedUsers);
app.get('/api/stats/registrations', routes.userRegistrations);
app.get('/api/stats/genderdivision', routes.genderDivision);
```

Nendele teekondadele vastavad haldurid tulevad pisut hiljem, kuid seni palun ma teil päringud valmis luua ning proovida ise luua esilehe jaoks vajalik haldur.

Kasutajaliidese lisamine

Pisikese kõrvalepõikena proovime lisada ka pisut visuaalsema külje, mis tagastab meile nimekirja API funktsioonidest HTML kujul. Expressi puhul on võimalik kasutada mitmeid vaadete mootoreid (*view engine*), nagu näiteks Jade, eds, handlebars (hbs) jne. Antud juhul kasutame just viimast (http://handlebarsjs.com/). Handlebars kasutab MVC lähenemist, kus mudel, vaade ja kontroller hoitakse teineteisest lahus. Vaade defineerib visuaalse külje, mis kuvab mudeli omadusi. Mudel on andmekandjaks ning kontrolleri ülesandeks on andmete saamine, mudeli moodustamine, sobiva vaate valimine ning andmete sellele ette söötmine.

Selle kasutamiseks peame esmalt handlebars toe lisama:

```
>npm install hbs --save
```

Seejärel lisame Expressi rakendusele viite vastavale vaate mootorile:

```
// Instatiate application instance
let app = express();

// Let's add a View Engine - Handlebars
app.set('view engine', 'hbs');
```

Edasi peame lisama vaate enda kirjelduse. Selleks loome eraldi kataloogi nimega “views” ning lisame sinna faili nimega “api-index.hbs”.

```
<html>
    <head>
        <title>{{title}}</title>
    </head>
    <body>
        <h1>{{title}}</h1>
        <ul>
            {{#each api}}
            <li><a href="{{{url}}}">{{name}}</a></li>
            {{/each}}
        </ul>
    </body>
</html>
```

Vaate faili paneme kirja üksnes selle osa, mida ja kuidas me kuvada soovime. Topelt loogeliste sulgude vahele kirjutatakse mudeli omadused ja juhtkäsud (nagu näiteks `#each`). Mudeli enda defineerime hiljem – praegu huvitab meid vaid see, et sellel objektil on pealkiri ning api omaduse kaudu funktsioonide nimekiri läbi nime ja url paaride. Vaikimisi URL-kodeeritakse kõik omadused, mistõttu tuleb meil `url` välja jaoks kasutada kolmekordset loogelist sulgu.

Nüüd seadistame teekonna, mille alt seda vaadet näha saaks (peale juurkataloogi kirjeldust):

```
app.get('/', routes.index);
app.get('/api', routes.apiIndex);
```

Lõpuks loome halduri enda teekonna:

```
exports.apiIndex = function(req, res) {
    let vm = {                          // vm = View Model
        title: 'API Functions',
        api: [
            { name: 'Users', url: '/api/users' },
            { name: 'Front Page', url: '/api/frontpage' },
            { name: 'Profile Page', url: '/api/profile/cbaccup3b' },
            { name: 'Post', url: '/api/post/19' },
            { name: 'General Statistics', url: '/api/stats' },
            { name: 'TOP 10 Most Followed Users', url:
'/api/stats/top10/followedusers' },
            { name: 'Registrations', url: '/api/stats/registrations' },
            { name: 'Gender Division', url: '/api/stats/genderdivision' }
        ]
    }

    res.render('api-index', vm);
}
```

Esimene samm on siin vaate mudeli (*view model*) loomine. Kui mudel on olemas, siis kutsume välja `render()` funktsiooni, öeldes millist vaadet me kuvada tahame ning andes kaasa äsja loodud mudeli.