

c Palvelimella olevan datan hakeminen



Olemme nyt viipyneet tovin keskittyen pelkkään frontendiin eli selainpuolen toiminnallisuuteen. Rupeamme itse toteuttamaan backendin eli palvelinpuolen toiminnallisuutta vasta kurssin kolmannessa osassa, mutta tutustumme jo nyt siihen, miten selaimessa suoritettava koodi kommunikoi backendin kanssa.

Käytetään nyt palvelimena sovelluskehitykseen tarkoitettua JSON Serveriä.

Tehdään projektin juurihakemistoon tiedosto *db.json*:

```
{
  "notes": [
    {
      "id": 1,
      "content": "HTML is easy",
      "important": true
    },
    {
      "id": 2,
      "content": "Browser can execute only JavaScript",
      "important": false
    },
    {
      "id": 3,
      "content": "GET and POST are the most important methods of HTTP protocol",
      "important": true
    }
  ]
}
```

[copy](#)

JSON Server on mahdollista asentaa koneelle ns. globaalisti komennolla `npm install -g json-server`. Globaali asennus edellyttää kuitenkin pääkäyttäjän oikeuksia eli se ei ole mahdollista laitoksen koneilla tai uusilla fuksiläppäreillä.



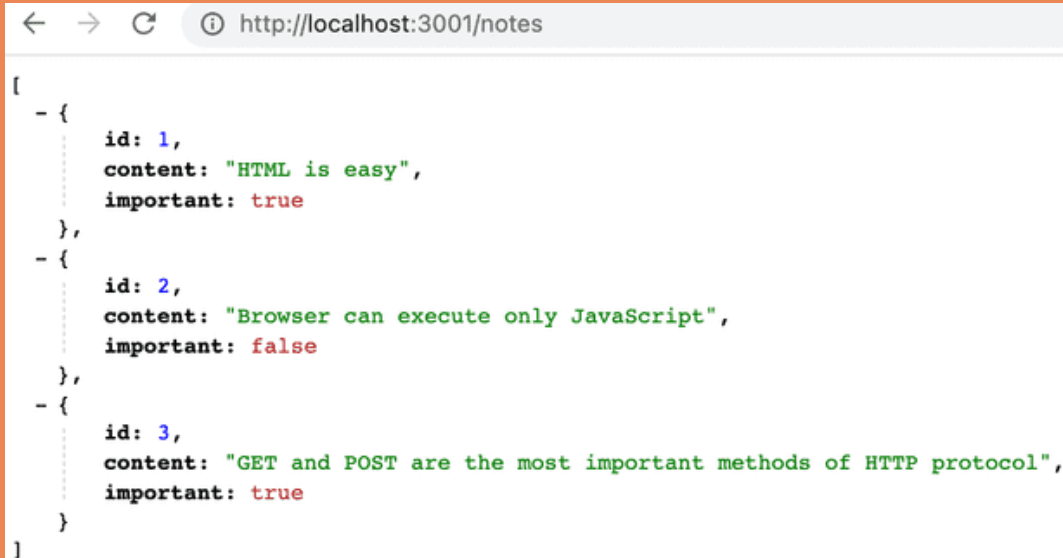
Globaali asennus ei ole kuitenkaan tarpeen, sillä voimme käynnistää JSON Serverin myös `npx` -komennon avulla:

```
npx json-server --port=3001 --watch db.json
```

copy

Oletusarvoisesti JSON Server käynnistyy porttiin 3000. Käytämme nyt kuitenkin porttia 3001.

Mennään selaimella osoitteeseen <http://localhost:3001/notes>. Kuten huomaamme, JSON Server tarjoaa osoitteessa tiedostoon tallentamamme muistiinpanot JSON-muodossa:



```
[
  - {
    id: 1,
    content: "HTML is easy",
    important: true
  },
  - {
    id: 2,
    content: "Browser can execute only JavaScript",
    important: false
  },
  - {
    id: 3,
    content: "GET and POST are the most important methods of HTTP protocol",
    important: true
  }
]
```

Jos selaimesi ei osaa näyttää JSON-muotoista dataa formatoituna, asenna jokin sopiva plugin, esim. [JSONVue](#) helpottamaan elämääsi.

Jatkossa ideana onkin se, että muistiinpanot talletetaan palvelimelle eli tässä vaiheessa JSON Serverille. React-koodi hakee muistiinpanot palvelimelta ja renderöi ne ruudulle. Kun sovellukseen lisätään uusi muistiinpano, React-koodi lähettää sen myös palvelimelle, jotta uudet muistiinpanot jäävät pysyvästi "muistiin".

JSON Server tallettaa kaiken datan palvelimella sijaitsevaan tiedostoon `db.json`. Todellisuudessa data tullaan tallentamaan johonkin tietokantaan. JSON Server on kuitenkin käyttökelpoinen apuväline, joka mahdollistaa palvelinpuolen toiminnallisuuden käyttämisen kehitysvaiheessa ilman tarvetta itse ohjelmoida mitään.

Tutustumme palvelinpuolen toteuttamisen periaatteisiin tarkemmin kurssin [osassa 3](#).

Selain suoritusympäristönä

Ensimmäisenä tehtävänä on siis hakea React-sovellukseen jo olemassa olevat muistiinpanot osoitteesta <http://localhost:3001/notes>.

Osan 0 [esimerkkiprojektissa](#) nähtiin jo eräs tapa hakea palvelimella olevaa dataa. Esimerkissä data haettiin [XMLHttpRequest](#) - eli XHR-olion avulla muodostetulla HTTP-pyyynnöllä. Kyseessä on vuonna 1999 lanseerattu tekniikka, jota kaikki web-selaimet ovat jo pitkään tukeneet.

Nykyään XHR:ää ei kuitenkaan kannata käyttää, ja selaimet tukevatkin jo laajasti [fetch](#) -metodia, joka perustuu XHR:n käyttämän tapahtumapohjaisen mallin sijaan ns. [promiseihin](#).

Muistutuksena edellisestä osasta (oikeastaan tätä tapaa pitää lähinnä *muistaa olla käyttämättä* ilman painavaa syytä), data haettiin XHR:llä seuraavasti:



```
const xhttp = new XMLHttpRequest()

xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    const data = JSON.parse(this.responseText)
    // käsittele muuttujaan data sijoitettu kyselyn tulos
  }
}

xhttp.open('GET', '/data.json', true)
xhttp.send()
```

copy

Heti alussa HTTP-pyyntöä vastaavalle `xhttp` -oliolle rekisteröidään *tapahtumankäsittelijä*, jota JavaScript Runtime kutsuu `xhttp` -olion tilan muuttuessa. Jos pyynnön vastaus on saapunut, data käsitellään halutulla tavalla.

Huomionarvoista on se, että tapahtumankäsittelijän koodi on määritelty jo ennen kuin itse pyyntö lähetetään palvelimelle. Tapahtumankäsittelijäfunktiolla tullaan kuitenkin suorittamaan vasta jossain myöhemmässä vaiheessa. Koodin suoritus ei siis etene synkronisesti "ylhäältä alas", vaan JavaScript kutsuu sille rekisteröityä tapahtumankäsittelijäfunktiota jossain vaiheessa *asynkronisesti*.

Esim. Java-ohjelmoinnista tuttu synkroninen tapa tehdä kyselyjä etenisi seuraavaan tapaan (huomaa että kyse ei ole oikeasti toimivasta Java-koodista):

```
HttpRequest request = new HttpRequest();

String url = "https://studies.cs.helsinki.fi/exampleapp/data.json";
List<Note> notes = request.get(url);

notes.forEach(m => {
  System.out.println(m.content);
});
```

copy

Javassa koodi etenee nyt rivi riviltä ja koodi pysähtyy odottamaan HTTP-pynnön eli komennon `request.get(...)` valmistumista. Komennon palauttama data eli muistiinpanot talletetaan muuttujaan ja dataa aletaan käsittelemään halutulla tavalla.

JavaScript-enginet eli suoritussympäristöt kuitenkin noudattavat asynkronista mallia, eli periaatteena on, että kaikki IO-operaatiot (poislukien muutama poikkeus) suoritetaan ei-blokkaavana, eli operaatioiden tulosta ei jäädä odottamaan vaan koodin suoritusta jatketaan heti eteenpäin.

Siinä vaiheessa kun operaatio valmistuu tai tarkemmin sanoen jonain valmistumisen jälkeisenä ajanhetkenä, JavaScript-engine kutsuu operaatiolle rekisteröityä tapahtumankäsittelijöitä.

Nykyisellään JavaScript-enginet ovat *yksisäikeisiä* eli ne eivät voi suorittaa rinnakkaista koodia. Tämän takia on käytännössä pakko käyttää ei-blokkaavaa mallia IO-operaatioiden suorittamiseen, sillä muuten selain 'jäätysi' esim. silloin kun palvelimelta haetaan dataa.

JavaScript-engineiden yksisäikeisyydellä on myös sellainen seuraus, että jos koodin suoritus kestää pitkään, selain menee jumiin suorituksen ajaksi. Jos lisätään sovelluksen alkuun seuraava koodi:

```
setTimeout(() => {
  console.log('loop..')
```

copy



```
let i = 0
while (i < 9999999999) {
  i++
}
console.log('end')
}, 5000)
```

Kaikki toimii viiden sekunnin ajan normaalisti. Kun `setTimeout` :in parametrina määritelty funktio suoritetaan, menee selaimen sivu jumiin pitkän loopin suorituksen ajaksi. Ainakaan Chromessa selaimen tabia ei pysty edes sulkemaan loopin suorituksen aikana.

Eli jotta selain säilyy *responsiivisena* eli että se reagoi koko ajan riittävän nopeasti käyttäjän haluamiin toimenpiteisiin, koodin logiikan tulee olla sellainen, että yksittäinen laskenta ei kestä liian kauan.

Aiheesta löytyy paljon lisämateriaalia Internetistä. Philip Robertsin esitelmä [What the heck is the event loop anyway?](#) on varsin havainnollinen esitys.

Nykyään selaimissa on mahdollisuus suorittaa myös rinnakkaista koodia ns. [web workerien](#) avulla. Yksittäisen selainikkunan koodin ns. event loopista huolehtii kuitenkin edelleen [vain yksi säie](#).

npm

Palaamme jälleen asiaan, eli datan hakemiseen palvelimelta.

Voisimme käyttää datan palvelimelta hakemiseen aiemmin mainittua promiseihin perustuvaa funktiota [fetch](#). Fetch on hyvä työkalu, se on standardoitu ja kaikkien modernien selaimien (poislukien IE) tukema.

Käytetään selaimen ja palvelimen väliseen kommunikaatioon kuitenkin [axios](#)-kirjastoa, joka toimii samaan tapaan kuin fetch, mutta on hieman mukavampikäyttöinen. Hyvä syy axios:in käytölle on myös se, että pääsemme tutustumaan siihen miten ulkopuolisia kirjastoja eli *npm-paketteja* liitetään React-projektiin.

Nykyään lähes kaikki JavaScript-projektit määrittellään node "pakkausmanagerin" eli [npm](#):n avulla. Myös Viten avulla generoidut projektit ovat npm-muotoisia projekteja. Varma tuntomerkki siitä on projektin juuressa oleva tiedosto *package.json*:

```
{
  "name": "notes-frontend",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "devDependencies": {
    "@types/react": "^18.2.15",
    "@types/react-dom": "^18.2.7",
    "@vitejs/plugin-react": "^4.0.3",
    "eslint": "^8.45.0",
    "eslint-plugin-react": "^7.32.2",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.4.3",
```

copy



```
"vite": "^4.4.5"
}
}
```

Tässä vaiheessa meitä kiinnostaa osa *dependencies*, joka määrittelee mitä *riippuvuuksia* eli ulkoisia kirjastoja projektilla on.

Voisimme määritellä Axios-kirjaston suoraan tiedostoon *package.json*, mutta on parempi asentaa se komentoriviltä:

```
npm install axios
```

copy

Huomaa, että `npm install` -komennot tulee antaa aina projektin juurihakemistossa eli siinä, jossa tiedosto *package.json* on.

Nyt Axios on mukana riippuvuuksien joukossa:

```
{
  "name": "notes-frontend",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "axios": "^1.4.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  // ...
}
```

copy

Sen lisäksi, että komento `npm install` lisäsi haluamamme kirjaston riippuvuuksien joukkoon, se myös *latasi* kirjaston koodin. Koodi löytyy muiden riippuvuuksien tapaan projektin juuren hakemistosta *node_modules*, mikä kuten huomata saattaa sisältääkin runsaasti kaikenlaista.

Tehdään toinenkin pieni lisäys. Asennetaan myös JSON Server projektin *sovelluskehityksen aikaiseksi* riippuvuudeksi komennolla

```
npm install json-server --save-dev
```

copy

ja tehdään tiedoston *package.json* osaan *scripts* pieni lisäys

```
{
  // ...
  "scripts": {
    "dev": "vite",
```

copy ↑

```
"build": "vite build",
"lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
"preview": "vite preview",
"server": "json-server -p3001 --watch db.json"
},
}
```

Nyt voimme käynnistää JSON Serverin projektin hakemistosta mukavasti ilman tarvetta parametrien määrittelylle:

```
npm run server
```

[copy](#)

Tutustumme `npm` -työkaluun tarkemmin kurssin kolmannessa osassa.

Huomaa, että aiemmin käynnistetty JSON Server tulee olla sammutettuna, muuten seuraa ongelmia:

```
Cannot bind to the port 3001. Please specify another port number either through --port argument or through the j
ration file
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! notes@0.1.0 server: `json-server -p3001 db.json`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the notes@0.1.0 server script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.
```

Virheilmoituksen punaisella oleva teksti kertoo mistä on kyse:

Cannot bind to the port 3001. Please specify another port number either through --port argument or through the json-server.json configuration file

Sovellus ei onnistu käynnistyessään kytkemään itseään porttiin 3001, koska kyseinen portti on jo aiemmin käynnistetyn JSON Serverin varaama.

Käytimme komentoa `npm install` kahteen kertaan hieman eri tavalla:

```
npm install axios
npm install json-server --save-dev
```

[copy](#)

Parametrissa oli siis hienoinen ero. Axios tallennettiin sovelluksen suoritusaikaiseksi riippuvuudeksi, sillä ohjelman suoritus edellyttää kirjaston olemassaoloa. JSON Server taas asennettiin sovelluskehityksen aikaiseksi riippuvuudeksi (`--save-dev`). JSON Server on ainoastaan apuna sovelluskehityksen aikana eikä varsinainen sovelluksemme tarvitse sitä. Erilaisista riippuvuuksista kerrotaan lisää kurssin seuraavassa osassa.

Axios ja promiset

Axios on nyt valmis käyttöömmme. Jatkossa oletetaan, että JSON Server on käynnissä portissa 3001. Lisäksi varsinainen React-sovellus tulee käynnistää erikseen erilliseen komentorivi-ikkunaan:

```
npm run dev
```

[copy](#)

Kirjaston voi ottaa käyttöön samaan tapaan kuin esim. React otetaan käyttöön eli sopivalla `import` -lauseella.

Lisätään seuraava tiedostoon *main.jsx*:

```
import axios from 'axios'

const promise = axios.get('http://localhost:3001/notes')
console.log(promise)

const promise2 = axios.get('http://localhost:3001/foobar')
console.log(promise2)
```

copy

Konsoliin tulostuu:



The screenshot shows a browser console with the following output:

- ▼ Promise {<pending>} ⓘ
 - ▶ [[Prototype]]: Promise
 - ▶ [[PromiseState]]: "fulfilled"
 - ▶ [[PromiseResult]]: Object
- ▼ Promise {<pending>} ⓘ
 - ▶ [[Prototype]]: Promise
 - ▶ [[PromiseState]]: "rejected"
 - ▶ [[PromiseResult]]: AxiosError
- ▶ GET http://localhost:3001/foobar 404 (Not Found)
- ▶ Uncaught (in promise)
- ▶ AxiosError {message: 'Request failed with status code 404', name: 'AxiosError', code: 'ERR_BAD_REQUEST', config: {...}}

Axiosin metodi `get` palauttaa promisen.

Mozillan dokumentaatio kertoo promisesta seuraavaa:

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

Promise siis edustaa asynkronista operaatiota. Promise voi olla kolmessa eri tilassa:

- Aluksi promise on *pending*, eli promisea vastaava asynkroninen operaatio ei ole vielä tapahtunut.
- Jos operaatio päättyy onnistuneesti, promise menee tilaan *fulfilled*, josta joskus käytetään myös nimitystä *resolved*.
- Kolmas mahdollinen tila on *rejected*, ja se edustaa epäonnistunutta operaatiota.

Esimerkkimme ensimmäinen promise on *fulfilled*, eli vastaa onnistunutta

`axios.get('http://localhost:3001/notes')` pyyntöä. Promiseista toinen taas on *rejected*. Syy selviää konsolista, eli yritimme tehdä HTTP GET -pyyntöä osoitteeseen, jota ei ole olemassa.

Jos ja kun haluamme tietoon promisea vastaavan operaation tuloksen, tulee promiselle rekisteröidä tapahtumankuuntelija. Tämä tapahtuu metodilla `then` :

```
const promise = axios.get('http://localhost:3001/notes')

promise.then(response => {
  console.log(response)
})
```

copy



Konsoliin tulostuu:

```

▼ {data: Array(3), status: 200, statusText: 'OK', headers: AxiosHeaders, config: {...}, ...} ⓘ
  ► config: {transitional: {...}, adapter: Array(2), transformRequest: Array(1), transformResponse: Array(1), timeout: 0, ...}
  ▼ data: Array(3)
    ► 0: {id: 1, content: 'HTML is easy', important: true}
    ► 1: {id: 2, content: 'Browser can execute only JavaScript', important: false}
    ► 2: {id: 3, content: 'GET and POST are the most important methods of HTTP protocol', important: true}
      length: 3
    ► [[Prototype]]: Array(0)
  ► headers: AxiosHeaders {cache-control: 'no-cache', content-length: '299', content-type: 'application/json; charset=utf-8', ...}
  ► request: XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequestUpload, ...}
    status: 200
    statusText: "OK"
  ► [[Prototype]]: Object

```

JavaScriptin suoritussympäristö kutsuu `then` -metodin avulla rekisteröityä takaisinkutsufunktiota antaen sille parametriksi olion `response`, joka sisältää kaiken oleellisen HTTP GET -pyynnön vastaukseen liittyvän, eli palautetun *datan*, *statuskoodin* ja *headerit*.

Promise-oliota ei ole yleensä tarvetta tallettaa muuttujaan, ja onkin tapana ketjuttaa metodin `then` kutsu suoraan Axiosin metodin kutsun perään:

```

axios.get('http://localhost:3001/notes').then(response => {
  const notes = response.data
  console.log(notes)
})

```

copy

Takaisinkutsufunktio ottaa nyt vastauksen sisällä olevan datan muuttujaan ja tulostaa muistiinpanot konsoliin.

Luettavampi tapa formatoida *ketjutettuja* metodikutsuja on sijoittaa jokainen kutsu omalle rivilleen:

```

axios
  .get('http://localhost:3001/notes')
  .then(response => {
    const notes = response.data
    console.log(notes)
  })

```

copy

Palvelimen palauttama data on pelkkää tekstiä, käytännössä yksi iso merkkijono. Axios osaa kuitenkin parsia datan JavaScript-taulukoksi, sillä palvelin on kertonut headerin *content-type* avulla että datan muoto on *application/json; charset=utf-8* (ks. edellinen kuva).

Voimme vihdoin siirtyä käyttämään sovelluksessamme palvelimelta haettavaa dataa.

Tehdään se aluksi "huonosti", eli lisätään sovellusta vastaavan komponentin *App* renderöinti takaisinkutsufunktion sisälle muuttamalla *main.jsx* seuraavaan muotoon:

```

import ReactDOM from 'react-dom/client'
import axios from 'axios'
import App from './App'

axios.get('http://localhost:3001/notes').then(response => {
  const notes = response.data
  ReactDOM.createRoot(document.getElementById('root')).render(<App notes={notes} />)
})

```

copy



Joissain tilanteissa tämäkin tapa voisi olla ok, mutta se on hieman ongelmallinen ja on parempi siirtää datan hakeminen komponenttiin *App*.

Ei ole kuitenkaan ihan selvää, mihin kohtaan komponentin koodia komento `axios.get` olisi hyvä sijoittaa.

Effect-hookit

Olemme jo käyttäneet Reactin version 16.8.0 mukanaan tuomia state hookeja tuomaan funktioina määritettyihin React-komponentteihin tilan. Versio 16.8.0 tarjoaa kokonaan uutena ominaisuutena myös effect-hookit, joista dokumentaatio kertoo:

*The Effect Hook lets you perform side effects in function components. **Data fetching**, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.*

Eli effect-hookit ovat juuri oikea tapa hakea dataa palvelimelta.

Poistetaan nyt datan hakeminen tiedostosta *main.jsx*. Komponentille *App* ei ole enää tarvetta välittää dataa propseina. Eli *main.jsx* pelkistyy seuraavaan muotoon:

```
ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

copy

Komponentti *App* muuttuu seuraavasti:

```
import { useState, useEffect } from 'react'
import axios from 'axios'
import Note from './components/Note'

const App = () => {
  const [notes, setNotes] = useState([])
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)

  useEffect(() => {
    console.log('effect')
    axios
      .get('http://localhost:3001/notes')
      .then(response => {
        console.log('promise fulfilled')
        setNotes(response.data)
      })
  }, [])
  console.log('render', notes.length, 'notes')

  // ...
}
```

copy

Koodiin on myös lisätty muutamia aputulostuksia, jotka auttavat hahmottamaan miten suoritus etenee.

Konsoliin tulostuu:

```
render 0 notes
effect
promise fulfilled
```

copy



Ensin siis suoritetaan komponentin määrittelevän funktion runko ja renderöidään komponentti ensimmäistä kertaa. Tässä vaiheessa tulostuu *render 0 notes* eli dataa ei ole vielä haettu palvelimelta.

Efekti, eli funktio

```
() => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}
```

[copy](#)

suoritetaan heti komponentin renderöinnin jälkeen. Funktion suoritus saa aikaan sen, että konsoliin tulostuu *effect* ja että komento `axios.get` aloittaa datan hakemisen palvelimelta sekä rekisteröi operaatiolle *tapahtumankäsittelijäksi* funktion

```
response => {
  console.log('promise fulfilled')
  setNotes(response.data)
})
```

[copy](#)

Siinä vaiheessa kun data saapuu palvelimelta, JavaScript Runtime kutsuu rekisteröityä tapahtumankäsittelijäfunktiota, joka tulostaa konsoliin *promise fulfilled* sekä tallettaa tilaan palvelimen palauttavat muistiinpanot funktiolla `setNotes(response.data)`.

Kuten aina, *tilan päivittävän funktion kutsu aiheuttaa komponentin uudelleen renderöitymisen*. Tämän seurauksena konsoliin tulostuu *render 3 notes* ja palvelimelta haetut muistiinpanot renderöityvät ruudulle.

Tarkastellaan vielä efektihookin määrittelyä kokonaisuudessaan

```
useEffect(() => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes').then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}, [])
```

[copy](#)

Kirjoitetaan koodi hieman toisella tavalla:

```
const hook = () => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
```

[copy](#)


```

    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
  }
}

```

```
useEffect(hook, [])
```

Nyt huomaamme selvemmin, että funktiolle `useEffect` (<https://react.dev/reference/react/useEffect>) annetaan *kaksi parametria*. Näistä ensimmäinen on funktio eli itse *efekti*. Dokumentaation mukaan

By default, effects run after every completed render, but you can choose to fire it only when certain values have changed.

Eli oletusarvoisesti efekti suoritetaan *aina* sen jälkeen, kun komponentti renderöidään. Meidän tapauksessamme haluamme suorittaa efektin vain ensimmäisen renderöinnin yhteydessä.

Funktion `useEffect` toista parametria käytetään tarkentamaan sitä, miten usein efekti suoritetaan. Jos toisena parametrina on tyhjä taulukko `[]`, suoritetaan efekti ainoastaan komponentin ensimmäisen renderöinnin jälkeen.

Effect hookien avulla on mahdollisuus tehdä paljon muutakin kuin hakea dataa palvelimelta, mutta tämä riittää meille tässä vaiheessa.

Mieti vielä tarkasti äsken läpikäytyä tapahtumasarjaa eli sitä, mitä kaikkea koodista suoritetaan, missä järjetyksessä ja kuinka monta kertaa. Tapahtumien järjestyksen ymmärtäminen on erittäin tärkeää!

Huomaa, että olisimme voineet kirjoittaa efektifunktion koodin myös seuraavasti:

```

useEffect(() => {
  console.log('effect')

  const eventHandler = response => {
    console.log('promise fulfilled')
    setNotes(response.data)
  }

  const promise = axios.get('http://localhost:3001/notes')
  promise.then(eventHandler)
}, [])

```

copy

Muuttujaan `eventHandler` on sijoitettu viite tapahtumankäsittelijäfunktioon. Axiosin metodin `get` palauttama promise on talletettu muuttujaan `promise`. Takaisinkutsun rekisteröinti tapahtuu antamalla promisen `then`-metodin parametrina muuttuja `eventHandler`, joka viittaa käsittelijäfunktioon. Useimmiten funktioiden ja promisejen sijoittaminen muuttujiin ei ole tarpeen, ja ylempänä käyttämämme kompaktimpi esitystapa riittää:

```

useEffect(() => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}

```

copy



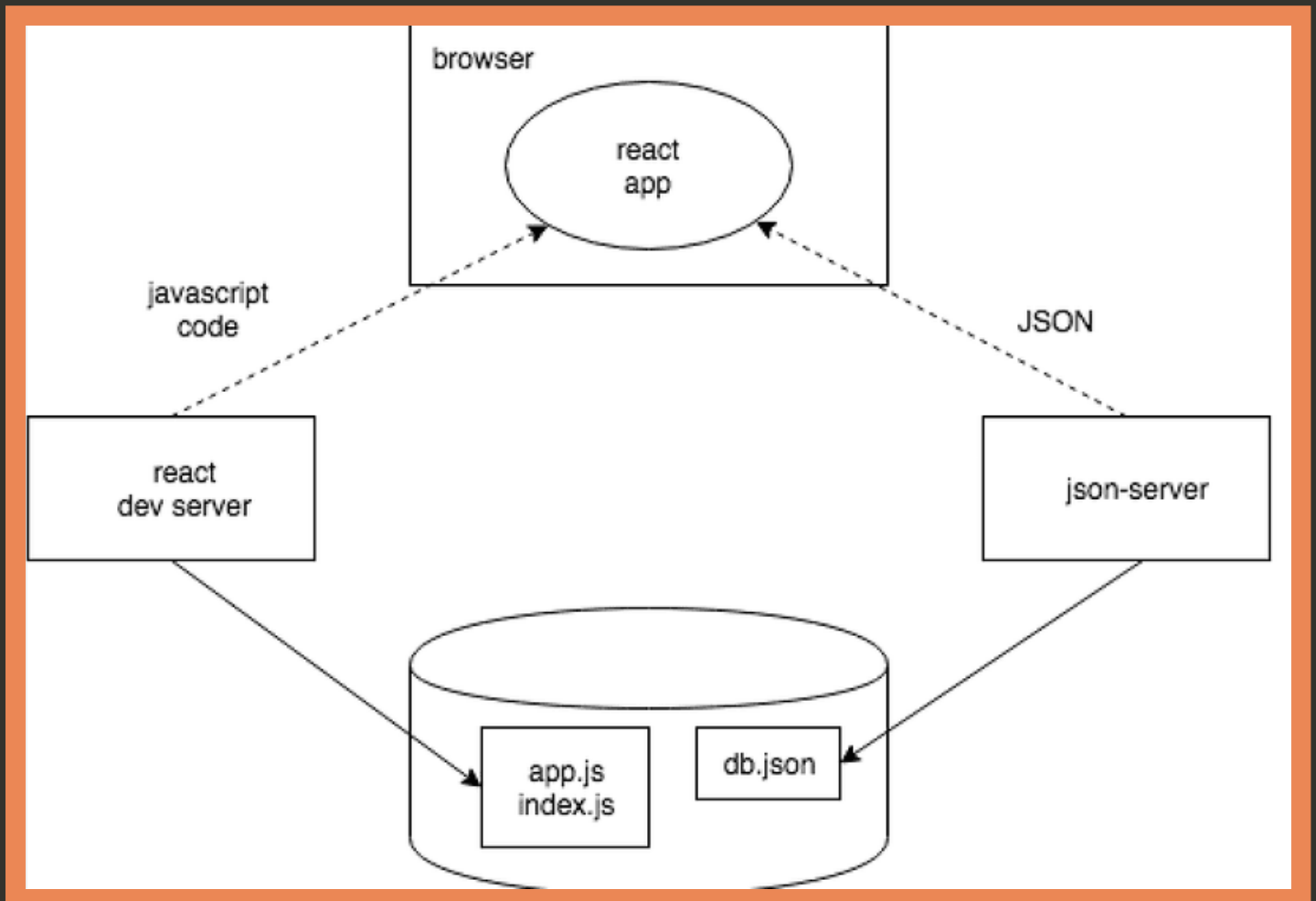
```
}  
}, [])
```

Sovelluksessa on tällä hetkellä vielä se ongelma, että jos lisäämme uusia muistiinpanoja, ne eivät tallennu palvelimelle asti. Eli kun lataamme sovelluksen uudelleen, kaikki lisäykset katoavat. Korjaus asiaan tulee pian.

Sovelluksen tämänhetkinen koodi on kokonaisuudessaan [GitHubissa](#), branchissa *part2-4*.

Sovelluskehityksen suoritusympäristö

Sovelluksemme kokonaisuuden konfiguraatiosta on pikkuhiljaa muodostunut melko monimutkainen. Käydään vielä läpi mitä tapahtuu missäkin. Seuraava diagrammi kuvaa asetelmaa:



React-sovelluksen muodostavaa JavaScript-koodia siis suoritetaan selaimessa. Selain hakee JavaScriptin *React Development Serveriltä*, joka on se ohjelma, joka käynnistyy kun suoritetaan komento `npm start`. Development Server muokkaa sovelluksen JavaScriptin selainta varten sopivaan muotoon, se mm. yhdistelee eri tiedostoissa olevan JavaScript-koodin yhdeksi tiedostoksi. Puhumme enemmän Development Serveristä kurssin [osassa 7](#).

JSON-muodossa olevan datan selaimessa pyörivä React-sovellus hakee siis koneella portissa 3001 käynnissä olevalta JSON Serveriltä, joka taas saa JSON-datan tiedostosta *db.json*.

Kaikki sovelluksen osat ovat sovelluskehitysvaiheessa siis ohjelmoijan koneella eli *localhostissa*. Tilanne muuttuu, kun sovellus viedään Internetiin. Teemme näin [osassa 3](#).



Tehtävä 2.11.

2.11: puhelinluettelo step6

Jatketaan puhelinluettelon kehittämistä. Talleta sovelluksen alkutila projektin juureen sijoitettavaan tiedostoon *db.json*:

```
{
  "persons": [
    {
      "name": "Arto Hellas",
      "number": "040-123456",
      "id": 1
    },
    {
      "name": "Ada Lovelace",
      "number": "39-44-5323523",
      "id": 2
    },
    {
      "name": "Dan Abramov",
      "number": "12-43-234345",
      "id": 3
    },
    {
      "name": "Mary Poppendieck",
      "number": "39-23-6423122",
      "id": 4
    }
  ]
}
```

copy

Käynnistä JSON Server porttiin 3001 ja varmista selaimella osoitteesta <http://localhost:3001/persons>, että palvelin palauttaa henkilölistan.

Jos saat virheilmoituksen

```
events.js:182
    throw er; // Unhandled 'error' event
    ^

Error: listen EADDRINUSE 0.0.0.0:3001
    at Object._errnoException (util.js:1019:11)
    at _exceptionWithHostPort (util.js:1041:20)
```

copy

on portti 3001 jo jonkin muun sovelluksen, esim. jo käynnissä olevan JSON Serverin käytössä. Sulje toinen sovellus tai jos se ei onnistu, vaihda porttia.

Muuta sovellusta siten, että alkutila haetaan Axios-kirjaston avulla palvelimelta. Hoida datan hakeminen Effect hookilla.



Osa 2b
Edellinen osa

Osa 2d
Seuraava osa

Kurssista

Kurssin sisältö

FAQ

Kurssilla mukana

Haaste



UNIVERSITY OF HELSINKI



HOUSTON

