

- a Kokoelmien renderöinti ja moduulit
- b Lomakkeiden käsittely
- c Palvelimella olevan datan hakeminen
- d Palvelimella olevan datan muokkaaminen

-REST

-Datan lähetys palvelimelle

-Muistiinpanon tärkeyden muutos

-Palvelimen kanssa tapahtuvan kommunikoinnin eristäminen omaan moduuliin

-Kehittyneempi tapa olioliteraalien määrittelyyn


-Promise ja virheet

-Tehtäviä

- e Tyylien lisääminen React-sovellukseen



Palvelimella olevan datan muokkaaminen

Kun sovelluksella luodaan uusia muistiinpanoja, täytyy ne luonnollisesti tallentaa palvelimelle. 
json-server mainitsee dokumentaatioissaan olevansa ns. REST- tai RESTful-API

Get a full fake REST API with zero coding in less than 30 seconds (seriously)

Ihan alkuperäisen määritelmän mukainen RESTful API json-server ei ole, mutta ei ole kovin moni muukaan itseään REST:iksi kutsuva rajapinta.

Tutustumme REST:iin tarkemmin kurssin seuraavassa osassa, mutta jo nyt on tärkeä ymmärtää minkälaista konventiota json-server ja yleisemminkin REST API:t käyttävät reittien, eli URL:ien ja käytettävien HTTP-pyyntöjen tyyppien suhteen.

REST

REST:issä yksittäisiä asioita esim. meidän tapauksessamme muistiinpanoja kutsutaan *resursseiksi*. Jokaisella resurssilla on yksilöivä osoite eli URL. json-serverin noudattaman yleisen konvention mukaan yksittäistä muistiinpanoa kuvaavan resurssin URL on muotoa *notes/3*, missä 3 on resurssin tunniste. Osoite *notes* taas vastaa kaikkien yksittäisten muistiinpanojen kokoelmaa.

Resursseja haetaan palvelimelta HTTP GET -pyynnöillä. Esim. HTTP GET osoitteeseen *notes/3* palauttaa muistiinpanon, jonka id-kentän arvo on 3. Kun taas HTTP GET -pyyntö osoitteeseen *notes* palauttaa kaikki muistiinpanot.

Uuden muistiinpanoa vastaavan resurssin luominen tapahtuu json-serverin noudattamassa REST-konventiossa tekemällä HTTP POST -pyyntö, joka kohdistuu myös samaan osoitteeseen *notes*. Pyyntöön mukana sen runkona eli *body*nä lähetetään luotavan muistiinpanon tiedot.

json-server vaatii, että tiedot lähetetään JSON-muodossa, eli käytännössä sopivasti muotoiltuna merkkijonona ja asettamalla headerille *Content-Type* arvo *application/json*.

Datan lähetys palvelimelle

Muutetaan nyt uuden muistiinpanon lisäämisestä huolehtivaa tapahtumankäsittelijää seuraavasti:

```
addNote = event => {
  event.preventDefault()
  const noteObject = {
    content: newNote,
    date: new Date(),
    important: Math.random() > 0.5,
  }

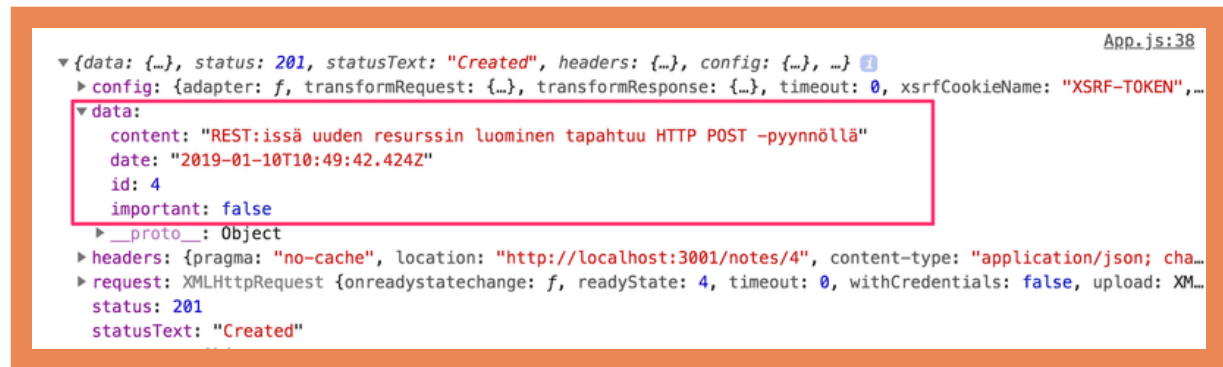
  axios
    .post('http://localhost:3001/notes', noteObject)
    .then(response => {
      console.log(response)
    })
}
```

eli luodaan muistiinpanoa vastaava olio, ei kuitenkaan lisätä sille kenttää *id*, sillä on parempi jättää id:n generointi palvelimen vastuulle!



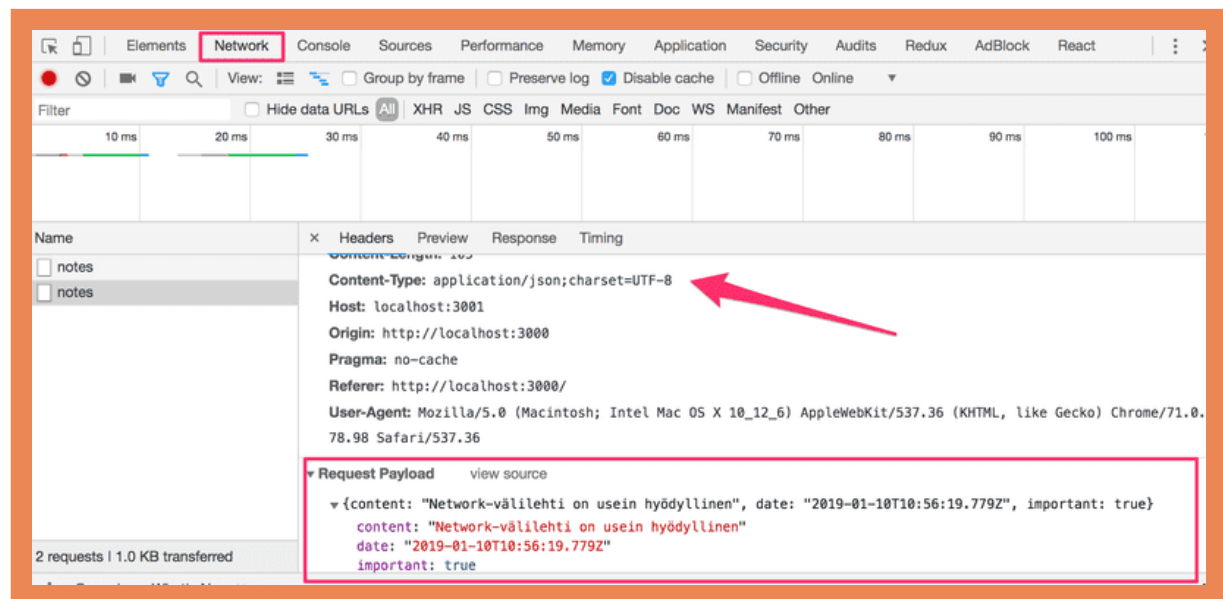
Olio lähetetään palvelimelle käyttämällä axiosin metodia `post`. Rekisteröity tapahtumankäsittelijä tulostaa konsoliin palvelimen vastauksen.

Kun nyt kokeillaan luoda uusi muistiinpano, konsoliin tulostus näyttää seuraavalta:



Uusi muistiinpano on siis `response` -olion kentän `data` arvona. Palvelin on lisännyt muistiinpanolle tunnusteen, eli `id`-kentän.

Joskus on hyödyllistä tarkastella HTTP-pyyntöjä osan 0 alussa paljon käytetyn konsolin *Network*-välilehden kautta:



Voimme esim. tarkastaa onko POST-pyyntöön mukana menevä data juuri se mitä oletimme, onko headerit asetettu oikein ym.

Koska POST-pyyntöissä lähettämämme data oli Javascript-olio, osasi axios automaattisesti asettaa pyynnön *Content-type* headerille oikean arvon eli *application/json*.

Uusi muistiinpano ei vielä renderöidy ruudulle, sillä emme aseta komponentille *App* uutta tilaa muistiinpanon luomisen yhteydessä. Viimeistellään sovellus vielä tältä osin:

```

addNote = event => {
  event.preventDefault()
  const noteObject = {
    content: newNote,
    date: new Date(),
    important: Math.random() > 0.5,
  }
}
  
```

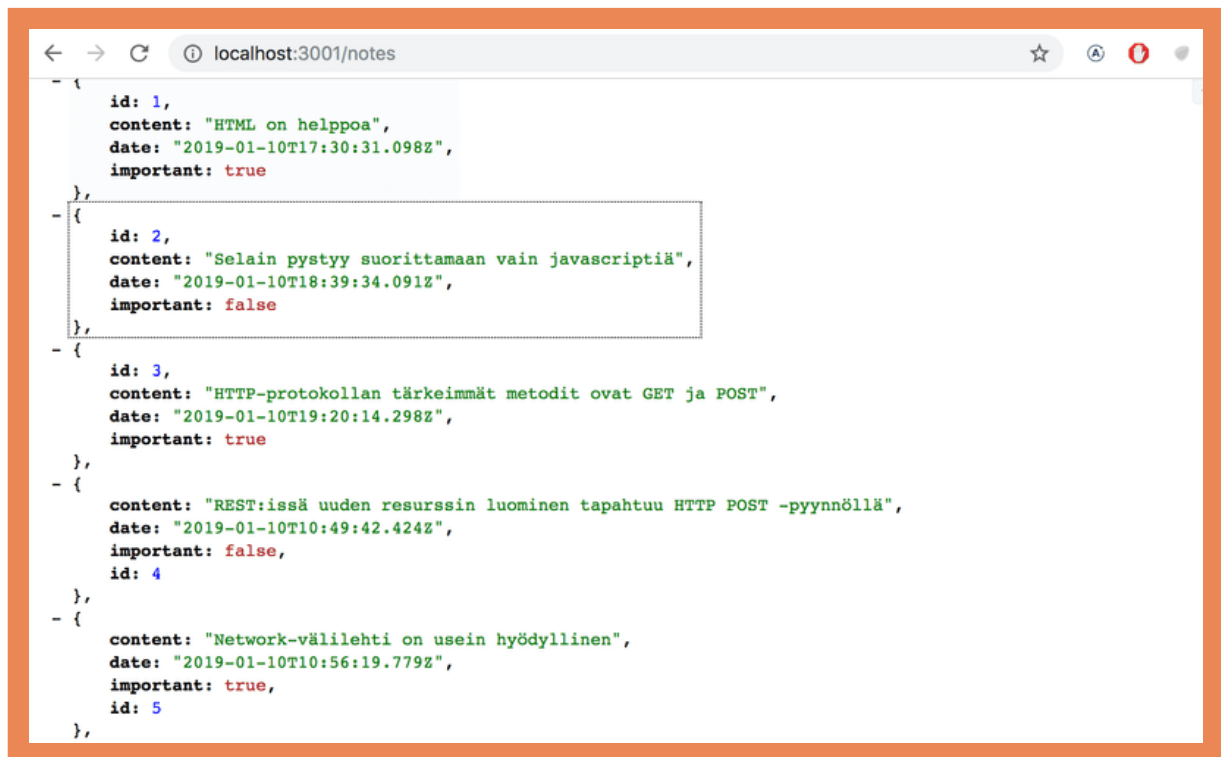


```
}  
  
axios  
  .post('http://localhost:3001/notes', noteObject)  
  .then(response => {  
    setNotes(notes.concat(response.data))  
    setNewNote('')  
  })  
}
```

Palvelimen palauttama uusi muistiinpano siis lisätään tuttuun tapaan funktiolla `setNotes` tilassa olevien muiden muistiinpanojen joukkoon (kannattaa muistaa tärkeä detalji siitä, että metodi `concat` ei muuta komponentin alkuperäistä tilaa, vaan luo uuden taulukon) ja tyhjennetään lomakkeen teksti.

Kun palvelimella oleva data alkaa vaikuttaa web-sovelluksen toimintalogiikkaan, tulee sovelluskehitykseen heti iso joukko uusia haasteita, joita tuo mukanaan mm. kommunikoinnin asynkronisuus. Debuggaamiseenkin tarvitaan uusia strategioita, debug-printtaukset ym. muuttuvat vain tärkeämmäksi, myös Javascriptin runtimen periaatteita ja React-komponenttien toimintaa on pakko tuntea riittävällä tasolla, arvaileminen ei riitä.

Palvelimen tilaa kannattaa tarkastella myös suoraan, esim. selaimella:



näin on mahdollista varmistua, mm. siirtyykö kaikki oletettu data palvelimelle.

Kurssin seuraavassa osassa alamme toteuttaa itse myös palvelimella olevan sovelluslogiikan, tutustumme silloin tarkemmin palvelimen debuggausta auttaviin työkaluihin, mm. postmaniin. Tässä vaiheessa json-server-palvelimen tilan tarkkailuun riittänee selain.

HUOM: sovelluksen nykyisessä versiossa selain lisää uudelle muistiinpanolle sen luomishetkeä kuvaavan kentän. Koska koneen oma kello voi näyttää periaatteessa mitä



sattuu, on aikaleimojen generointi todellisuudessa viisaampaa hoitaa palvelimella ja tulemmekin tekemään tämän muutoksen kurssin seuraavassa osassa.

Sovelluksen tämän hetkinen koodi on kokonaisuudessaan [githubissa](#), branchissa *part2-5*.

Muistiinpanon tärkeyden muutos

Lisätään muistiinpanojen yhteyteen painike, millä niiden tärkeyttä voi muuttaa.

Muistiinpanon määrittelevän komponentin muutos on seuraavat:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important' : 'make important'

  return (
    <li>
      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Komponentissa on nappi, jolle on rekisteröity klikkaustapahtuman käsittelijäksi propsien avulla välitetty funktio `toggleImportance`.

Komponentti *App* määrittelee alustavan version tapahtumankäsittelijästä `toggleImportanceOf` ja välittää sen jokaiselle *Note*-komponentille:

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

  // ...

  const toggleImportanceOf = id => {
    console.log(
      'importance of ' + id + ' needs to be toggled'
    )
  }

  const rows = () => notesToShow.map(note =>
    <Note
      key={note.id}
      note={note}
      toggleImportance={() => toggleImportanceOf(note.id)}
    />
  )

  return (
    // ...
  )
}
```



```
)
}
```

Huomaa, että jokaisen muistiinpanon tapahtumankäsittelijäksi tulee nyt *yksilöllinen* funktio, sillä kunkin muistiinpanon *id* on uniikki.

Esim. jos *node.id* on 3, tulee tapahtumankäsittelijäksi `toggleImportance(note.id)` eli käytännössä:

```
() => { console.log('importance of 3 needs to be toggled') }
```

Pieni muistutus tähän väliin. Tapahtumankäsittelijän koodin tulostuksessa muodostetaan tulostettava merkkijono Javan tyyliin plussaamalla stringejä:

```
console.log('importance of ' + id + ' needs to be toggled')
```

ES6:n template string -ominaisuuden ansiosta Javascriptissa vastaavat merkkijonot voidaan kirjoittaa hieman mukavammin:

```
console.log(`importance of ${id} needs to be toggled`)
```

Merkkijonon sisälle voi nyt määritellä "dollari-aaltosulku"-syntaksilla kohtia, minkä sisälle evaluoidaan javascript-lausekkeita, esim. muuttujan arvo. Huomaa, että template stringien hipsutyyppi poikkeaa Javascriptin normaaleista merkkijonojen käyttämisestä hipsuista.

Yksittäistä json-serverillä olevaa muistiinpanoa voi muuttaa kahdella tavalla, joko *korvaamalla* sen tekemällä HTTP PUT -pyyntö muistiinpanon yksilöivään osoitteeseen tai muuttamalla ainoastaan joidenkin muistiinpanon kenttien arvoja HTTP PATCH -pyynnöllä.

Korvaamme nyt muistiinpanon kokonaan, sillä samalla tulee esille muutama tärkeä React:iin ja Javascriptiin liittyvä seikka.

Tapahtumankäsittelijäfunktion lopullinen muoto on seuraavassa:

```
const toggleImportanceOf = id => {
  const url = `http://localhost:3001/notes/${id}`
  const note = notes.find(n => n.id === id)
  const changedNote = { ...note, important: !note.important }

  axios.put(url, changedNote).then(response => {
    setNotes(notes.map(note => note.id !== id ? note : response.data))
  })
}
```



Melkein jokaiselle riville sisältyy tärkeitä yksityiskohtia. Ensimmäinen rivi määrittelee jokaiselle muistiinpanolle id-kenttään perustuvan yksilöivän url:in.

Taulukon metodilla `find` etsitään muutettava muistiinpano ja talletetaan muuttujaan `note` viite siihen.

Sen jälkeen luodaan *uusi olio*, jonka sisältö on sama kuin vanhan olion sisältö poislukien kenttä `important`. Luominen näyttää hieman erikoiselta:

```
const changedNote = { ...note, important: !note.important }
```

Kyseessä on vielä standardoimattoman `object spread` -operaation soveltaminen.

Käytännössä `{ ... note }` luo olion, jolla on kenttinaan kopiot olion `note` kenttien arvoista. Kun aaltosulkeisiin lisätään asioita, esim. `{ ...note, important: true }`, tulee uuden olion kenttä `important` saamaan arvon `true`. Eli esimerkissämme `important` saa uudessa oliossa vanhan arvonsa käänteisarvon.

Pari huomioita. Miksi teimme muutettavasta oliosta kopion vaikka myös seuraava koodi näyttää toimivan:

```
const note = notes.find(n => n.id === id)
note.important = !note.important

axios.put(url, note).then(response => {
  // ...
})
```

Näin ei ole suositettavaa tehdä, sillä muuttuja `note` on viite komponentin tilassa, eli `notes` -taulukossa olevaan olioon, ja kuten muistamme tilaa ei Reactissa saa muuttaa suoraan!

Kannattaa myös huomata, että uusi olio `changedNote` on ainoastaan ns. shallow copy, eli uuden olion kenttien arvoina on vanhan olion kenttien arvot. Jos vanhan olion kentät olisivat itsessään olioita, viittaisivat uuden olion kentät samoihin olioihin.

Uusi muistiinpano lähetetään sitten PUT-pyyntönä mukana palvelimelle, jossa se korvaa aiemman muistiinpanon.

Takaisinkutsufunktiossa asetetaan komponentin `App` tilaan `notes` kaikki vanhat muistiinpanot paitsi muuttuneen, josta tilaan asetetaan palvelimen palauttama versio:

```
axios.put(url, changedNote).then(response => {
  setNotes(notes.map(note => note.id !== id ? note : response.data))
})
```



Tämä saadaan aikaan metodilla `map` :

```
notes.map(note => note.id !== id ? note : response.data)
```

Operaatio siis luo uuden taulukon vanhan taulukon perusteella. Jokainen uuden taulukon alkio luodaan mahdollisesti siten, että jos ehto `note.id !== id` on tosi, otetaan uuteen taulukkoon suoraan vanhan taulukon kyseinen alkio. Jos ehto on epätosi, eli kyseessä on muutettu muistiinpano, otetaan uuteen taulukkoon palvelimen palauttama olio.

Käytetty `map` -kikka saattaa olla aluksi hieman hämmentävä. Asiaa kannattaakin miettiä tovi. Tapaa tullaan käyttämään kurssilla vielä kymmeniä kertoja.

Palvelimen kanssa tapahtuvan kommunikoinnin eristäminen omaan moduuliin

App-komponentti alkaa kasvaa uhkaavasti kun myös palvelimen kanssa kommunikointi tapahtuu komponentissa. Single responsibility -periaatteen hengessä kommunikointi onkin viisainta eristää omaan moduuliinsa.

Luodaan hakemisto `src/services` ja sinne tiedosto `notes.js`:

```
import axios from 'axios'
const baseUrl = 'http://localhost:3001/notes'

const getAll = () => {
  return axios.get(baseUrl)
}

const create = newObject => {
  return axios.post(baseUrl, newObject)
}

const update = (id, newObject) => {
  return axios.put(`${baseUrl}/${id}`, newObject)
}

export default {
  getAll: getAll,
  create: create,
  update: update
}
```

Moduuli palauttaa nyt olion, jonka kenttinä (*getAll*, *create* ja *update*) on kolme muistiinpanojen käsittelyä hoitavaa funktiota. Funktiot palauttavat suoraan axiosin metodien palauttaman promisen.

Komponentti *App* saa moduulin käyttöön `import` -lauseella




```
import noteService from './services/notes'
```

```
const App = () => {
```

moduulin funktioita käytetään importatun muuttujan `noteService` kautta seuraavasti:

```
const App = () => {
  // ...

  useEffect(() => {
    noteService
      .getAll()
      .then(response => {
        setNotes(response.data)
      })
  }, [])

  const toggleImportanceOf = id => {
    const note = notes.find(n => n.id === id)
    const changedNote = { ...note, important: !note.important }

    noteService
      .update(id, changedNote)
      .then(response => {
        setNotes(notes.map(note => note.id !== id ? note : response.data))
      })
  }

  const addNote = (event) => {
    event.preventDefault()
    const noteObject = {
      content: newNote,
      date: new Date().toISOString(),
      important: Math.random() > 0.5
    }

    noteService
      .create(noteObject)
      .then(response => {
        setNotes(notes.concat(response.data))
        setNewNote('')
      })
  }

  // ...
}

export default App
```

Voisimme viedä ratkaisua vielä askeleen pidemmälle, sillä käyttäessään moduulin funktioita komponentti `App` saa olion, joka sisältää koko HTTP-pyyntöön vastauksen:



```
noteService
  .getAll()
  .then(response => {
    setNotes(response.data)
  })
```

Eli asia mistä *App* on kiinnostunut on parametrin kentässä *response.data*.

Moduulia olisi miellyttävämpi käyttää, jos se HTTP-pyynnön vastauksen sijaan palauttaisi suoraan muistiinpanot sisältävän taulukon. Tällöin moduulin käyttö näyttäisi seuraavalta

```
noteService
  .getAll()
  .then(initialNotes => {
    setNotes(initialNotes)
  })
```

Tämä onnistuu muuttamalla moduulin koodia seuraavasti (koodiin jää ikävästi copy-pastea, emme kuitenkaan nyt välitä siitä):

```
import axios from 'axios'
const baseUrl = 'http://localhost:3001/notes'

const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}
```

```
{() => fs}
```

```
}

const update = (id, newObject) => {
  const request = axios.put(`${baseUrl}/${id}`, newObject)
  return request.then(response => response.data)
}

export default {
  getAll: getAll,
  create: create,
  update: update
}
```

eli enää ei palautetakaan suoraan axiosin palauttamaa promisea, vaan otetaan promise ensin muuttuun `request` ja kutsutaan sille metodia `then` :



```
const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}
```

Täydellisessä muodossa kirjoitettuna viimeinen rivi olisi:

```
const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => {
    return response.data
  })
}
```

Myös nyt funktio `getAll` palauttaa promisen, sillä promisen metodi `then` palauttaa promisen.

Koska `then` :in parametri palauttaa suoraan arvon `response.data`, on funktion `getAll` palauttama promise sellainen, että jos HTTP-kutsu onnistuu, antaa promise takaisinkutsulleen HTTP-pyyntöön mukana olleen datan, eli se toimii juuri niin kuin haluamme.

Moduulin muutoksen jälkeen täytyy komponentti `App` muokata `noteService` :n metodien takaisinkutsujen osalta ottamaan huomioon, että ne palauttavat datan suoraan:

```
const App = () => {
  // ...

  useEffect(() => {
    noteService
      .getAll()
      .then(initialNotes => {
        setNotes(initialNotes)
      })
  }, [])

  const toggleImportanceOf = id => {
    const note = notes.find(n => n.id === id)
    const changedNote = { ...note, important: !note.important }

    noteService
      .update(id, changedNote)
      .then(returnedNote => {
        setNotes(notes.map(note => note.id !== id ? note : returnedNote))
      })
  }

  const addNote = (event) => {
    event.preventDefault()
    const noteObject = {
      content: newNote,
      date: new Date().toISOString(),
    }
  }
}
```



```
    important: Math.random() > 0.5
  }

  noteService
    .create(noteObject)
    .then(returnedNote => {
      setNotes(notes.concat(returnedNote))
      setNewNote('')
    })
  }

  // ...
}
```

Tämä kaikki on hieman monimutkaista ja asian selittäminen varmaan vain vaikeuttaa sen ymmärtämistä. Internetistä löytyy paljon vaihtelevatasoista materiaalia aiheesta, esim. tämä.

You do not know JS sarjan kirja "Async and performance" selittää asian hyvin mutta tarvitsee selitykseen kohtuullisen määrän sivuja.

Promisejen ymmärtäminen on erittäin keskeistä modernissa Javascript-sovelluskehityksessä, joten asiaan kannattaa uhrata kohtuullisessa määrin aikaa.

Kehittyneempi tapa olioliteraalien määrittelyyn

Muistiinpanopalvelut määrittelevä moduuli siis eksporttaa olion, jonka kenttinä *getAll*, *create* ja *update* ovat muistiinpanojen käsittelyyn tarkoitetut funktiot.

Moduulin määrittely tapahtui seuraavasti:

```
import axios from 'axios'
const baseUrl = 'http://localhost:3001/notes'

const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

const create = newObject => {
  const request = axios.post(baseUrl, newObject)
  return request.then(response => response.data)
}

const update = (id, newObject) => {
  const request = axios.put(`${baseUrl}/${id}`, newObject)
  return request.then(response => response.data)
}

export default {
  getAll: getAll,
  create: create,
  update: update
}
```



Exportattava asia on siis seuraava, hieman erikoiselta näyttävä olio:

```
{
  getAll: getAll,
  create: create,
  update: update
}
```

Olion määrittelyssä vasemmalla puolella kaksoispistettä olevat nimet tarkoittavat eksportoitavan olion *kenttiä*, kun taas oikealla puolella olevat nimet ovat moduulin sisällä *määriteltyjä muuttujia*.

Koska olion kenttien nimet ovat samat kuin niiden arvon määrittelevien muuttujien nimet, voidaan olion määrittely kirjoittaa tiivimmässä muodossa:

```
{
  getAll,
  create,
  update
}
```

Eli moduulin määrittely yksinkertaisuus seuraavaan muotoon

```
import axios from 'axios'
const baseUrl = 'http://localhost:3001/notes'

const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

const create = newObject => {
  const request = axios.post(baseUrl, newObject)
  return request.then(response => response.data)
}

const update = (id, newObject) => {
  const request = axios.put(`${baseUrl}/${id}`, newObject)
  return request.then(response => response.data)
}

export default { getAll, create, update }
```

Tässä tiivimmässä olioiden määrittelytavassa hyödynnetään ES6:n myötä Javascriptiin tullutta uutta ominaisuutta, joka mahdollistaa hieman tiiviimmän tavan muuttujien avulla tapahtuvaan olioiden määrittelyyn.



Havainnollistaaksemme asiaa tarkastellaan tilannetta, jossa meillä on muuttujissa arvoja

```
const name = 'Leevi'  
const age = 0
```

Vanhassa Javascriptissä olio täytyi määritellä seuraavaan tyyliin

```
const person = {  
  name: name,  
  age: age  
}
```

koska muuttujien ja luotavan olion kenttien nimi nyt on sama, riittää ES6:ssa kirjoittaa:

```
const person = { name, age }
```

lopputulos molemmissa on täsmälleen sama, eli ne luovat olion jonka kentän *name* arvo on *Leevi* ja kentän *age* arvo *0*.

Promise ja virheet

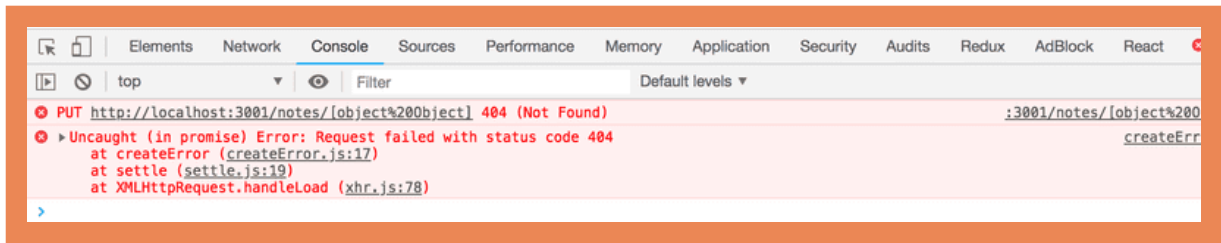
Jos sovelluksemme mahdollistaisi muistiinpanojen poistamisen, voisi syntyä tilanne, missä käyttäjä yrittää muuttaa sellaisen muistiinpanon tärkeyttä, joka on jo poistettu järjestelmästä.

Simuloidaan tällaista tilannetta "kovakoodaamalla" noteServiceen funktioon `getAll` muistiinpano, jota ei ole todellisuudessa (eli palvelimella) olemassa:

```
const getAll = () => {  
  const request = axios.get(baseUrl)  
  const nonExisting = {  
    id: 10000,  
    content: 'Tätä muistiinpanoa ei ole palvelimelta',  
    date: '2017-12-10T17:30:31.098Z',  
    important: true,  
  }  
  return request.then(response => response.data.concat(nonExisting))  
}
```

Kun valemuistiinpanon tärkeyttä yritetään muuttaa, konsoliin tulee virheilmoitus, joka kertoo palvelimen vastanneen urliin `/notes/10000` tehtyyn HTTP PUT -pyyntöön statuskoodilla 404 *not found*:





Sovelluksen tulisi pystyä käsittelemään tilanne hallitusti. Jos konsoli ei ole auki, ei käyttäjä huomaa mitään muuta kuin sen, että muistiinpanon tärkeys ei vaihdu napin painelusta huolimatta.

Jo aiemmin mainittiin, että promisella voi olla kolme tilaa. Kun HTTP-pyyntö epäonnistuu, menee pyyntöä vastaava promise tilaan *rejected*. Emme tällä hetkellä käsittele koodissamme promisen epäonnistumista mitenkään.

Promisen epäonnistuminen käsitellään antamalla `then` -metodille parametrikksi myös toinen takaisinkutsufunktio, jota kutsutaan siinä tapauksessa jos promise epäonnistuu.

Ehkä yleisempi tapa kuin kahden tapahtumankäsittelijän käyttö on liittää promiseen epäonnistumistilanteen käsittelijä kutsumalla metodia `catch`.

Käytännössä virhetilanteen käsittelijän rekisteröiminen tapahtuisi seuraavasti

```
axios
  .get('http://example.com/probably_will_fail')
  .then(response => {
    console.log('success!')
  })
  .catch(error => {
    console.log('fail')
  })
```

Jos pyyntö epäonnistuu, kutsutaan `catch` -metodin avulla rekisteröityä käsittelijää.

Metodia `catch` hyödynnetään usein siten, että se sijoitetaan syvemmälle promiseketjuun.

Kun sovelluksemme tekee HTTP-operaation syntyy oleellisesti ottaen promiseketju:

```
axios
  .put(`${baseUrl}/${id}`, newObject)
  .then(response => response.data)
  .then(changedNote => {
    // ...
  })
```

Metodilla `catch` voidaan määritellä ketjun lopussa käsittelijäfunktio, jota kutsutaan siinä vaiheessa jos mikä tahansa ketjun promisesta epäonnistuu, eli menee tilaan *rejected*:



```
axios
  .put(`${baseUrl}/${id}`, newObject)
  .then(response => response.data)
  .then(changedNote => {
    // ...
  })
  .catch(error => {
    console.log('fail')
  })
```

Hyödynnetään tätä ominaisuutta, ja sijoitetaan virheenkäsittelijä komponenttiin *App*:

```
const toggleImportanceOf = id => {
  const note = notes.find(n => n.id === id)
  const changedNote = { ...note, important: !note.important }

  noteService
    .update(id, changedNote).then(returnedNote => {
      setNotes(notes.map(note => note.id !== id ? note : returnedNote))
    })
    .catch(error => {
      alert(
        `muistiinpano '${note.content}' on jo valitettavasti poistettu palvelimelta`
      )
      setNotes(notes.filter(n => n.id !== id))
    })
}
```

Virheilmoitus annetaan vanhan kunnon alert -dialogin avulla ja palvelimelta poistettu muistiinpano poistetaan tilasta.

Olemattoman muistiinpanon poistaminen siis tapahtuu metodilla filter, joka muodostaa uuden taulukon, jonka sisällöksi tulee alkuperäisen taulukon sisällöstä ne alkiot, joille parametrina oleva funktio palauttaa arvon true:

```
notes.filter(n => n.id !== id)
```

Alertia tuskin kannattaa käyttää todellisissa React-sovelluksissa. Opimme kohta kehittyneemmän menetelmän käyttäjille tarkoitettujen tiedotteiden antamiseen. Toisaalta on tilanteita, joissa simppele battle tested -menetelmä kuten `alert` riittää aluksi aivan hyvin. Hienomman tavan voi sitten tehdä myöhemmin jos aikaa ja intoa riittää.

Sovelluksen tämän hetkinen koodi on kokonaisuudessaan githubissa, branchissa *part2-6*.



Tehtäviä

2.15: puhelinluettelo step7

Palataan jälleen puhelinluettelon pariin.

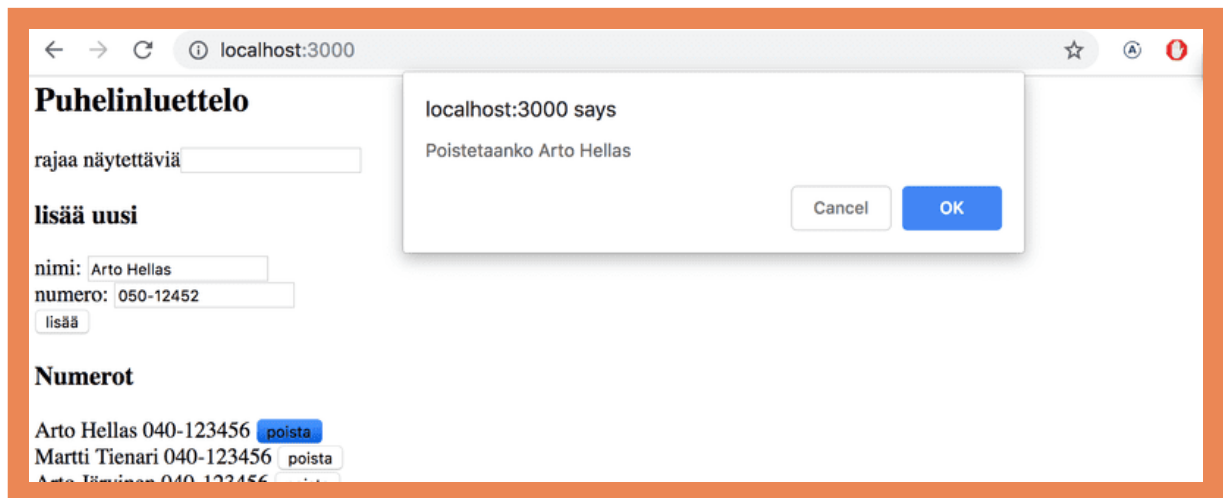
Tällä hetkellä luetteloon lisättäviä uusia numeroita ei synkronoida palvelimelle. Korjaa tilanne.

2.16: puhelinluettelo step8

Siirrä palvelimen kanssa kommunikoinnista vastaava toiminnallisuus omaan moduuliin tämän osan materiaalissa olevan esimerkin tapaan.

2.17: puhelinluettelo step9

Tee ohjelmaan mahdollisuus yhteystietojen poistamiseen. Poistaminen voi tapahtua esim. nimen yhteyteen liitettyllä napilla. Poiston suorittaminen voidaan varmistaa käyttäjältä window.confirm -metodilla:



Palvelimelta tiettyä henkilöä vastaava resurssi tuhotaan tekemällä HTTP DELETE -pyyntö resurssia vastaavaan *URL*:iin, eli jos poistaisimme esim. käyttäjän, jonka *id* on 2, tulisi tapauksessamme tehdä HTTP DELETE osoitteeseen *localhost:3001/persons/2*. Pyyntöön mukana ei lähetetä mitään dataa.

Axios -kirjaston avulla HTTP DELETE -pyyntö tehdään samaan tapaan kuin muutkin pyynnot.

Huom: et voi käyttää Javascriptissa muuttujan nimeä `delete` sillä kyseessä on kielen varattu sana, eli seuraava ei onnistu:

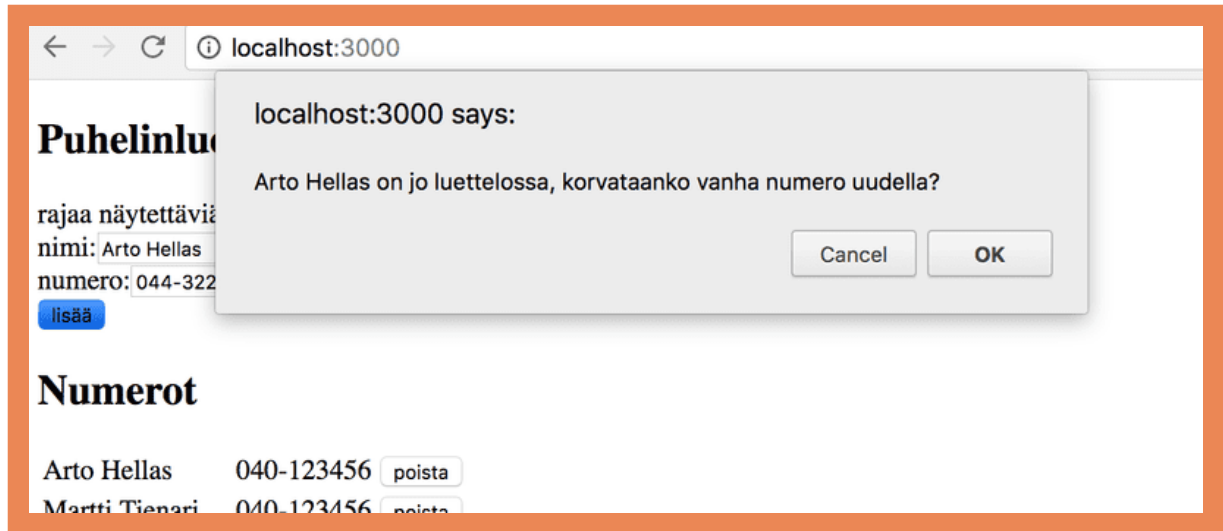
```
// käytä jotain muuta muuttujan nimeä
const delete = (id) => {
  // ...
}
```



2.18*: puhelinluettelo step10

Muuta toiminnallisuutta siten, että jos jo olemassaolevalle henkilölle lisätään numero, korvaa lisätty numero aiemman numeron. Korvaaminen kannattaa tehdä HTTP PUT -pyynnöllä.

Jos henkilön tiedot löytyvät jo luettelosta, voi ohjelma kysyä käyttäjältä varmistuksen korvataanko numero:



Ehdota muutosta materiaalin sisältöön

< Osa 2c
Edellinen osa

Osa 2e >
Seuraava osa

Kurssista

Kurssin sisältö

FAQ

Kurssilla mukana

Haaste



UNIVERSITY OF HELSINKI

HOUSTON



