

## d 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)*

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



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. 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 *bodynä* lähetetään luotavan muistiinpanon tiedot.

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

### Datan lähetys palvelimelle



Muutetaan nyt uuden muistiinpanon lisäämisestä huolehtivaa tapahtumankäsittelijää seuraavasti

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

[copy](#)

```
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:

```
▼ {data: {...}, status: 201, statusText: 'Created', headers: AxiosHeaders, config: {...}, ...} ⓘ
  ► config: {transitional: {...}, adapter: Array(2), transformRequest: Array(1), transformResponse: Array(1), timeout: 0, ...}
  ► data: {content: 'POST is used to create new resources', important: false, id: 4}
  ► headers: AxiosHeaders {cache-control: 'no-cache', content-length: '88', content-type: 'application/json; charset=utf-8'}
  ► request: XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequest}
  status: 201
  statusText: "Created"
```

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

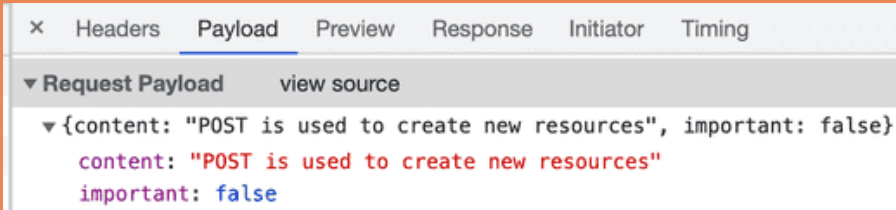
Usein on hyödyllistä tarkastella HTTP-pyyntöjä osan 0 alussa paljon käytetyn konsolin *Network*-välilehden kautta. Välilehti *header* kertoo pyynnön perustiedot ja näyttää pyynnön ja vastauksen *headers*ien arvot:

The screenshot shows the Chrome DevTools Network tab with the 'Headers' sub-tab selected. The 'General' section shows the request details: Request URL is http://localhost:3001/notes, Request Method is POST, Status Code is 201 Created, Remote Address is [::1]:3001, and Referrer Policy is strict-origin-when-cross-origin. The 'Response Headers' section is collapsed. The 'Request Headers' section is expanded, showing a list of headers: Accept: application/json, text/plain, \*/\*, Accept-Encoding: gzip, deflate, br, Accept-Language: en-GB,en-US;q=0.9,en;q=0.8, Connection: keep-alive, Content-Length: 68, Content-Type: application/json, and Host: localhost:3001. A 'View source' link is visible next to the 'Request Headers' section.

Header	Value
Accept	application/json, text/plain, */*
Accept-Encoding	gzip, deflate, br
Accept-Language	en-GB,en-US;q=0.9,en;q=0.8
Connection	keep-alive
Content-Length	68
Content-Type	application/json
Host	localhost:3001

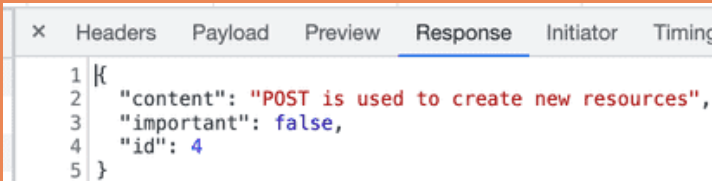
Koska POST-pyynnössä lähettämämme data oli JavaScript-olio, osasi Axios automaattisesti asettaa pyynnön *Content-type*-headerille oikean arvon eli *application/json*.

Välilehdeltä *payload* näemme missä muodossa data lähti:



```
{content: "POST is used to create new resources", important: false}
```

Välilehti *response* on myös hyödyllinen, se kertoo mitä palvelin palautti:



```
{
  "content": "POST is used to create new resources",
  "important": false,
  "id": 4
}
```

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,
    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 detali 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 tärkeämmiksi, myös JavaScriptin runtimen periaatteita ja React-komponenttien toimintaa on pakko tuntea riittävällä tasolla, arvaileminen ei riitä.

Palvelimen tilaa kannattaa tarkastella myös suoraan esim. selaimella:



```
[
  - {
    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
  },
  - {
    content: "POST is used to add data",
    important: false,
    id: 4
  }
]
```

Näin on mahdollista varmistua mm. siitä, siirtyykö kaikki oletettu data palvelimelle.

Kurssin seuraavassa osassa alamme toteuttaa itse myös palvelimella olevan sovelluslogiikan. Tutustumme silloin tarkemmin palvelimen debuggausta auttaviin työkaluihin kuten Postmaniin. Tässä vaiheessa JSON Server -palvelimen tilan tarkkailuun riittänee selain.

Sovelluksen tämänhetkinen koodi on kokonaisuudessaan GitHubissa, branchissa *part2-5*.

## Muistiinpanon tärkeyden muutos

Lisätään muistiinpanojen yhteyteen painikkeet, joilla muistiinpanojen tärkeyttä voi muuttaa.

Muistiinpanon määrittelevän komponentin muutos on seuraava:

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

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

[copy](#)

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([])
```

[copy](#)

```

const [newNote, setNewNote] = useState('')
const [showAll, setShowAll] = useState(true)

// ...

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

// ...

return (
  <div>
    <h1>Notes</h1>
    <div>
      <button onClick={() => setShowAll(!showAll)}>
        show {showAll ? 'important' : 'all' }
      </button>
    </div>
    <ul>
      {notesToShow.map(note =>
        <Note
          key={note.id}
          note={note}
          toggleImportance={() => toggleImportanceOf(note.id)}
        />
      )}
    </ul>
    // ...
  </div>
)
}

```

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

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

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

copy

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

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

copy

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

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

copy

Merkkijonon sisälle voi nyt määritellä "dollari-aaltosulku"-syntaksilla kohtia, joiden sisälle evaluoidaan JavaScript-lausekkeita, esim. muuttujan arvo. Huomaa, että template stringien hipsutyyppi poikkeaa JavaScriptin normaalien merkkijonojen käyttämisestä hipsuista.



Yksittäistä JSON Serverillä olevaa muistiinpanoa voi muuttaa kahdella tavalla: joko *korvaamalla* sen tekemällä HTTP PUT -pyynnön muistiinpanon yksilöivään osoitteeseen tai muuttamalla ainoastaan joidenkin muistiinpanon kenttien arvoja HTTP PATCH -pyynnöllä.

Korvaamme nyt muistiinpanon kokonaan, sillä samalla tulee esille muutama tärkeä Reactiin ja JavaScriptiin liittyvä seikka.

Tapahtumankäsittelijäfunktion lopullinen muoto on seuraava:

```
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))  
  })  
}
```

copy

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 pois lukien kenttä `important` jonka arvo vaihtuu päinvastaiseksi.

Niin sanottua object spread -syntaksia hyödyntävä uuden olion luominen näyttää hieman erikoiselta:

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

copy

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

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 => {  
  // ...  
})
```

copy

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

Kannattaa huomata myös, 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-pyynnön mukana palvelimelle, jossa se korvaa aiemman muistiinpanon.

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

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

copy

Tämä saadaan aikaan metodilla `map` :

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

copy

Operaatio siis luo uuden taulukon vanhan taulukon perusteella. Jokainen uuden taulukon alkio luodaan ehdollisesti 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 komponenttissa. 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  
}
```

copy



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'
```

copy

```
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,
```

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

```
    }
```

```
    noteService
```

```
      .create(noteObject)
```

```
      .then(response => {
```

```
        setNotes(notes.concat(response.data))
```

```
        setNewNote('')
```

```
      })
```

```
  }
```

```
  // ...
```

```
}
```

```
export default App
```

copy

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





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

[copy](#)

Eli kiinnostava asia on parametrin kentässä *response.data*.

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

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

[copy](#)

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)
}

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
}
```

[copy](#)

Enää ei palautetakaan suoraan Axiosin palauttamaa promisea, vaan otetaan promise ensin muuttujaan `request` ja kutsutaan sille metodia `then` :

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

[copy](#)

Täydellisessä muodossa kirjoitettuna viimeinen rivi olisi:

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

copy

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-pyynnön 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,
      important: Math.random() > 0.5
    }

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

  // ...
}
```

copy



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 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
}
```

copy

Eksportattava asia on siis seuraava, hieman erikoiselta näyttävä olio:

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

copy

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 tiiviimmässä muodossa:



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

[copy](#)

Moduulin määrittely yksinkertaistettuna:

```
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 }
```

[copy](#)

Tässä tiiviimmässä 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
```

[copy](#)

Vanhassa JavaScriptissä olio täytyi määritellä seuraavaan tyyliin:

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

[copy](#)

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

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

[copy](#)

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

## Promise ja virheet

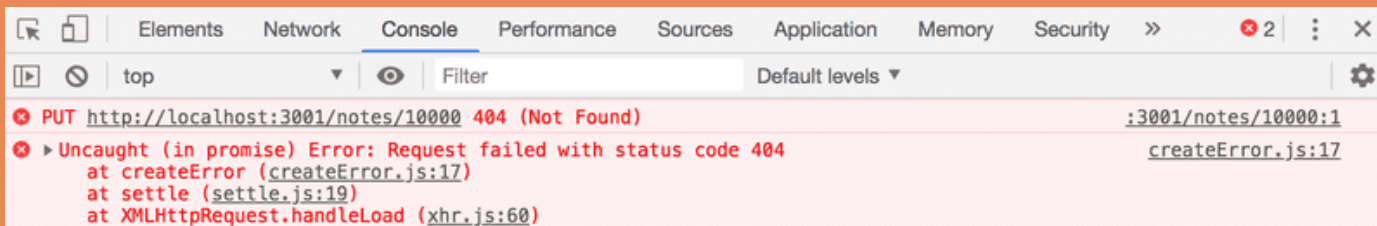
Jos sovelluksemme mahdollistaisi muistiinpanojen poistamisen, voisi syntyä tilanne, jossa käyttäjä yrittää muuttaa sellaisen muistiinpanon tärkeyttä, joka on jo poistettu järjestelmästä.

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

```
const getAll = () => {
  const request = axios.get(baseUrl)
  const nonExisting = {
    id: 10000,
    content: 'This note is not saved to server',
    important: true,
  }
  return request.then(response => response.data.concat(nonExisting))
}
```

copy

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 parametriksi myös toinen takaisinkutsufunktio, jota kutsutaan promisen epäonnistuessa.

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 => {
```

copy



```
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 => {
    // ...
  })
```

copy

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

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

copy

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(
        `the note '${note.content}' was already deleted from server`
      )
      setNotes(notes.filter(n => n.id !== id))
    })
}
```

copy

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

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



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 tämänhetkinen koodi on kokonaisuudessaan [GitHubissa](#), branchissa `part2-6`.

## Full stack -sovelluskehittäjän vala

On taas tehtävien aika. Tehtävien haastavuus alkaa nousta, sillä koodin toimivuuteen vaikuttaa myös se, kommunikoiko React-koodi oikein JSON Serverin kanssa.

Meidän onkin syytä päivittää websovelluskehittäjän vala *Full stack -sovelluskehittäjän valaksi*, eli muistuttaa itseämme siitä, että frontendin koodin lisäksi seuraamme koko ajan sitä, miten frontend ja backend kommunikoivat.

Full stack -ohjelmointi on *todella* hankalaa, ja sen takia lupaan hyödyntää kaikkia ohjelmointia helpottavia keinoja:

- pidän selaimen konsolin koko ajan auki
- *tarkkailen säännöllisesti selaimen network-välilehdeltä, että frontendin ja backendin välinen kommunikaatio tapahtuu oletusteni mukaan*
- *tarkkailen säännöllisesti palvelimella olevan datan tilaa, ja varmistan että frontendin lähettämä data siirtyy sinne kuten oletin*
- etenen pienin askelin
- käytän koodissa runsaasti `console.log` -komentoja varmistamaan sen, että varmasti ymmärrän jokaisen kirjoittamani koodirivin, sekä etsiessäni koodista mahdollisia bugin aiheuttajia
- jos koodini ei toimi, en kirjoita enää yhtään lisää koodia, vaan alan poistamaan toiminnan rikkoneita rivejä tai palaan suosiolla tilanteeseen, missä koodi vielä toimi
- kun kysyn apua kurssin Discord- tai Telegram-kanavalla, tai muualla internetissä, muotoilen kysymyksen järkevästi, esim. [täällä](#) esiteltyyn tapaan

## Tehtävät 2.12.-2.15.

### 2.12: puhelinluettelo step7

Palataan jälleen puhelinluettelon pariin.

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

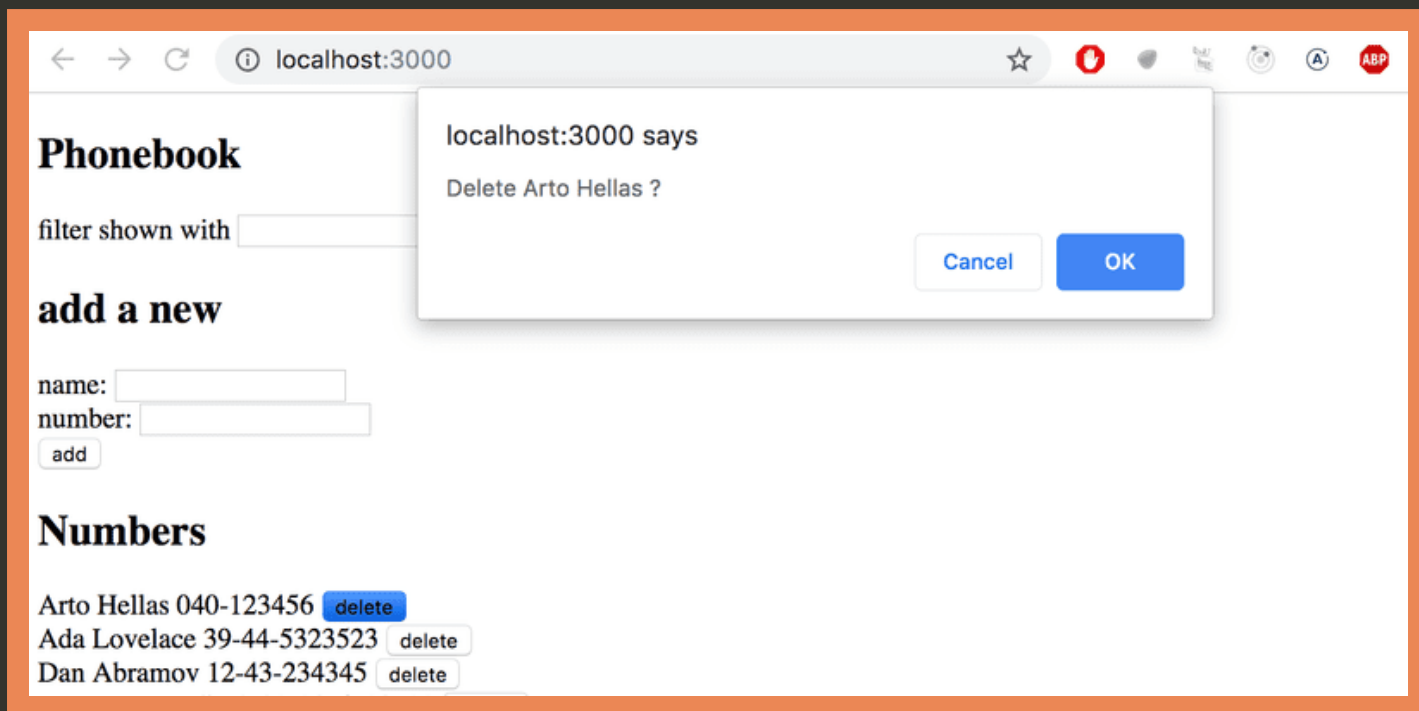
### 2.13: puhelinluettelo step8

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



## 2.14: puhelinluettelo step9

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



Tiettyä henkilöä vastaava resurssi tuhotaan palvelimelta 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*. Pyynnön 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) => {
  // ...
}
```

copy

## 2.15\*: puhelinluettelo step10

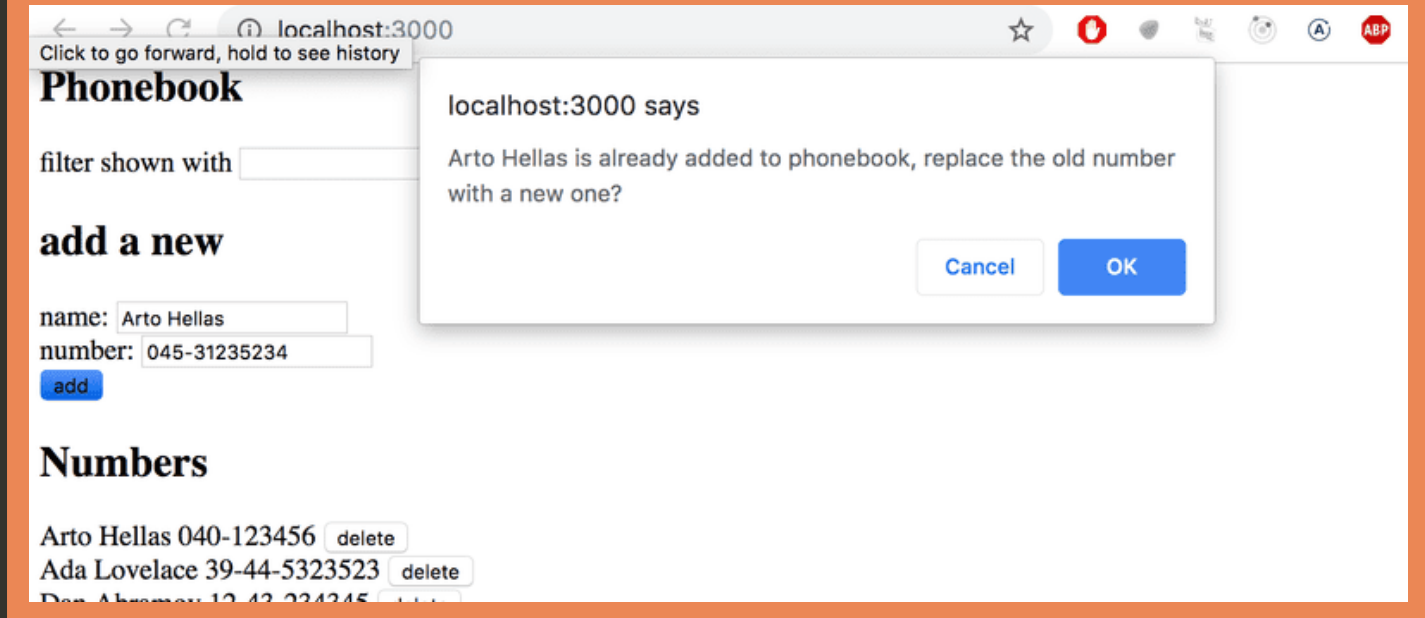
*Miksi tehtävä on merkattu tähdellä? Selitys asiaan täällä.*

Muuta toiminnallisuutta siten, että jos jo olemassa olevalle 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:







## Ehdota muutosta materiaalin sisältöön

Osa 2c

Edellinen osa

Osa 2e

Seuraava osa

Kurssista

Kurssin sisältö

FAQ

Kurssilla mukana

Haaste





**UNIVERSITY OF HELSINKI**



**HOUSTON**

