

Fullstack

Osa 2

Lomakkeiden käsittely



b Lomakkeiden käsittely

Jatketaan sovelluksen laajentamista siten, että se mahdollistaa uusien muistiinpanojen lisäämisen.

Muistiinpanojen tallettaminen komponentin tilaan

Jotta saisimme sivun päivittymään uusien muistiinpanojen lisäyksen yhteydessä, on parasta sijoittaa muistiinpanot komponentin *App* tilaan. Eli importataan funktio useState ja määritellään sen avulla komponentille tila, joka saa aluksi arvokseen propsina välitettävän muistiinpanot alustavan taulukon:

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

const App = (props) => {
  const [notes, setNotes] = useState(props.notes)

  return (
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
    </div>
  )
}

export default App
```

copy



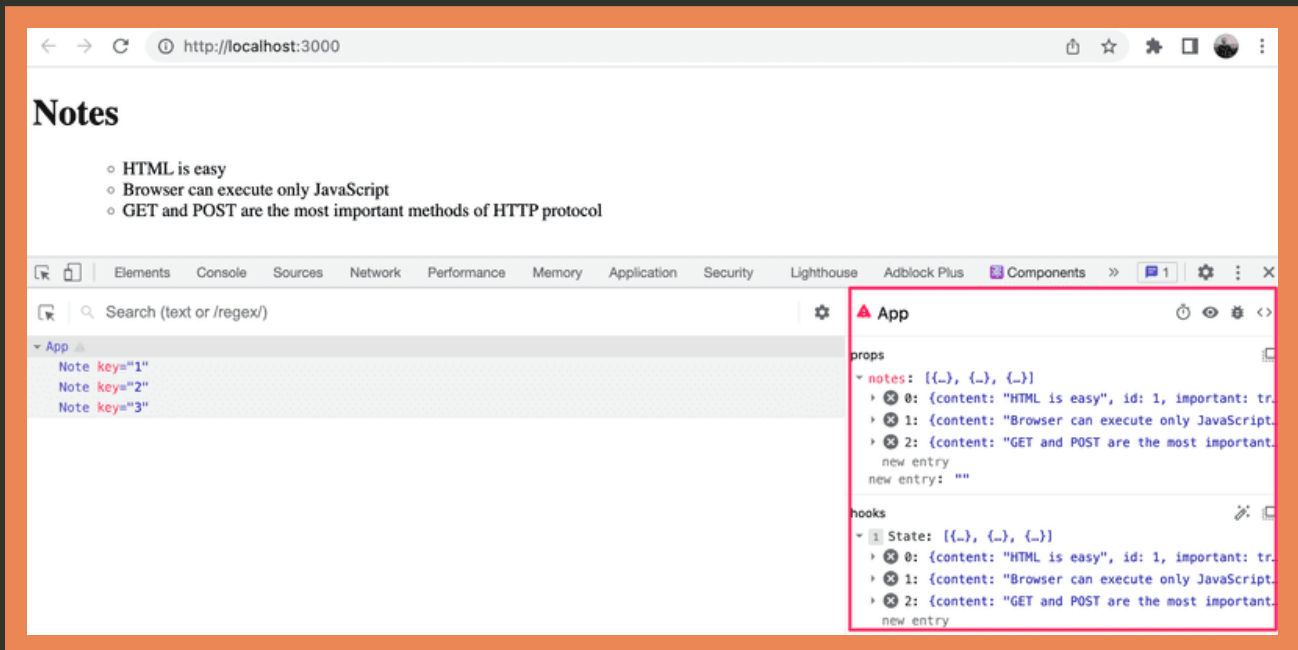
Komponentti siis alustaa funktion `useState` avulla tilan `notes` arvoksi propseina välitettävän alustavan muistiinpanojen listan:

```
const App = (props) => {
  const [notes, setNotes] = useState(props.notes)

  // ...
}
```

copy

Voimme vielä havainnollistaa tilanteen React Developer Toolsin avulla:



Jos haluaisimme lähteä liikkeelle tyhjästä muistiinpanojen listasta, annettaisiin tilan alkuarvoksi tyhjä taulukko, ja koska komponentti ei käyttäisi ollenkaan propseja, voitaisiin parametri `props` jättää kokonaan määrittelemättä:

```
const App = () => {
  const [notes, setNotes] = useState([])

  // ...
}
```

copy

Jätetään kuitenkin toistaiseksi tilalle alkuarvon asettava määrittely voimaan.

Lisätään seuraavaksi komponenttiin lomake eli HTML form uuden muistiinpanon lisäämistä varten:

```
const App = (props) => {
  const [notes, setNotes] = useState(props.notes)

  const addNote = (event) => {
    event.preventDefault()
    console.log('button clicked', event.target)
  }

  return (
    <div>
```

copy



```

    <h1>Notes</h1>
    <ul>
      {notes.map(note =>
        <Note key={note.id} note={note} />
      )}
    </ul>
    <form onSubmit={addNote}>
      <input />
      <button type="submit">save</button>
    </form>
  </div>
)
}

```

Lomakkeelle on lisätty myös tapahtumankäsittelijäksi funktio `addNote` reagoimaan sen "lähettämiseen" eli napin painamiseen.

Tapahtumankäsittelijä on osasta 1 tuttuun tapaan määritelty seuraavasti:

```

const addNote = (event) => {
  event.preventDefault()
  console.log('button clicked', event.target)
}

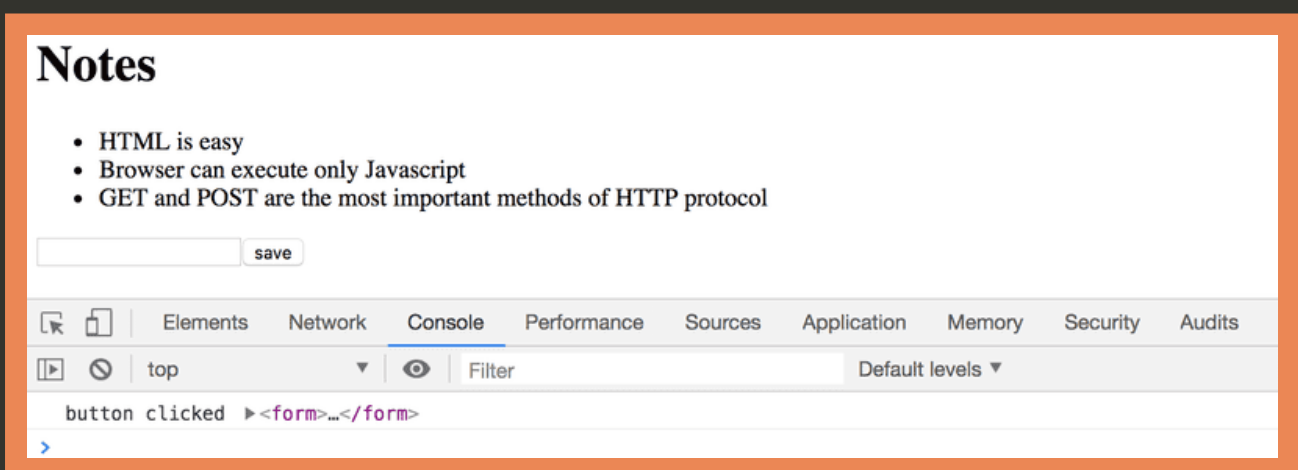
```

[copy](#)

Parametrin `event` arvona on metodin kutsun aiheuttama tapahtuma.

Tapahtumankäsittelijä kutsuu heti tapahtuman metodia `event.preventDefault()` jolla se estää lomakkeen lähetyksen oletusarvoisen toiminnan, joka aiheuttaisi mm. sivun uudelleenlatautumisen.

Tapahtuman kohde eli `event.target` on tulostettu konsoliin:



Kohteena on siis komponentin määrittelemä lomake.

Miten pääsemme käsiksi lomakkeen *input*-komponenttiin syötettyyn dataan?

Kontrolloitu komponentti

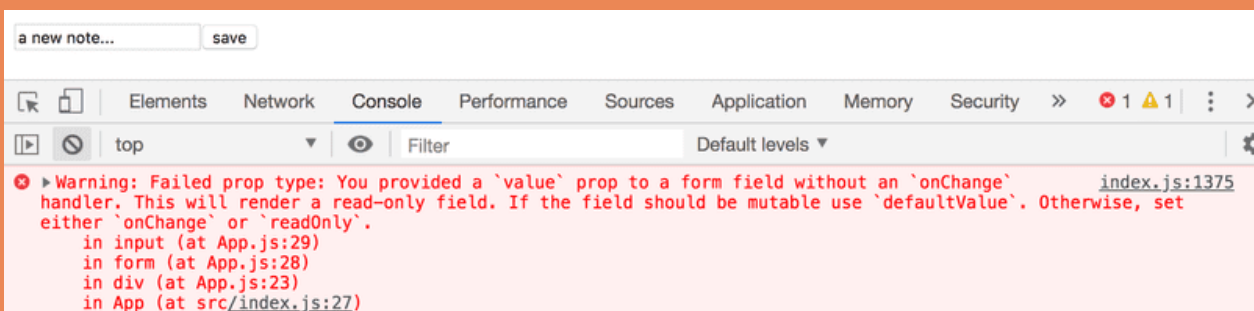


Tapoja on useampia, joista tutustumme ensin kontrolloituina komponentteina toteutettuihin lomakkeisiin.

Lisätään komponentille *App* tila *newNote* lomakkeen syötettä varten ja määritellään se *input*-komponentin attribuutin *value* arvoksi:

```
const App = (props) => {  
  const [notes, setNotes] = useState(props.notes)  
  const [newNote, setNewNote] = useState(  
    'a new note...'  
  )  
  
  const addNote = (event) => {  
    event.preventDefault()  
    console.log('button clicked', event.target)  
  }  
  
  return (  
    <div>  
      <h1>Notes</h1>  
      <ul>  
        {notes.map(note =>  
          <Note key={note.id} note={note} />  
        )}  
      </ul>  
      <form onSubmit={addNote}>  
        <input value={newNote} />  
        <button type="submit">save</button>  
      </form>  
    </div>  
  )  
}
```

Tilaan *newNote* määritelty "placeholder"-teksti *a new note...* ilmestyy syötekomponenttiin, mutta tekstiä ei voi muuttaa. Konsoliin tuleekin ikävä varoitus joka kertoo mistä on kyse:



Koska määrittelimme syötekomponentille *value*-attribuutiksi komponentin *App* tilassa olevan muuttujan, alkaa *App* kontrolloimaan syötekomponentin toimintaa.

Jotta kontrolloidun syötekomponentin editoiminen olisi mahdollista, täytyy sille rekisteröidä *tapahtumankäsittelijä*, joka synkronoi syötekenttään tehdyt muutokset komponentin *App* tilaan:

```
const App = (props) => {  
  const [notes, setNotes] = useState(props.notes)  
  const [newNote, setNewNote] = useState(  
    'a new note...'  
  )  
  
  const addNote = (event) => {  
    event.preventDefault()  
    console.log('button clicked', event.target)  
  }  
  
  return (  
    <div>  
      <h1>Notes</h1>  
      <ul>  
        {notes.map(note =>  
          <Note key={note.id} note={note} />  
        )}  
      </ul>  
      <form onSubmit={addNote}>  
        <input value={newNote} />  
        <button type="submit">save</button>  
      </form>  
    </div>  
  )  
}
```

```

    'a new note...'
  )

  // ...

  const handleNoteChange = (event) => {
    console.log(event.target.value)
    setNewNote(event.target.value)
  }

  return (
    <div>
      <h1>Notes</h1>
      <ul>
        {notes.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleNoteChange}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}

```

Lomakkeen *input*-komponentille on nyt rekisteröity tapahtumankäsittelijä tilanteeseen *onChange*:

```

<input
  value={newNote}
  onChange={handleNoteChange}
/>

```

[copy](#)

Tapahtumankäsittelijää kutsutaan *aina kun syötekomponentissa tapahtuu jotain*.

Tapahtumankäsittelijämetodi saa parametriksi tapahtumaolion `event`.


```

const handleNoteChange = (event) => {
  console.log(event.target.value)
  setNewNote(event.target.value)
}

```

[copy](#)

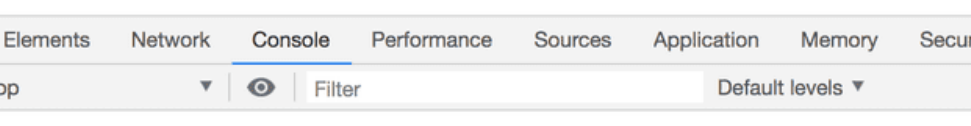
Tapahtumaolion kenttä `target` vastaa nyt kontrolloitua *input*-kenttää ja `event.target.value` viittaa inputin syötekentän arvoon.

Huomaa, että toisin kuin lomakkeen lähettämistä vastaavan tapahtuman *onSubmit* käsittelijässä, nyt oletusarvoisen toiminnan estävää metodikutsua `event.preventDefault()` ei tarvita, sillä  syötekentän muutoksella ei ole oletusarvoista toimintaa toisin kuin lomakkeen lähettämisellä.

Voit seurata konsolista miten tapahtumankäsittelijää kutsutaan:

Notes

- HTML is easy
- Browser can execute only Javascript
- GET and POST are the most important methods of HTTP protocol



The screenshot shows the Chrome DevTools interface with the Console tab selected. The top bar contains the text 'typing text' and a 'save' button. The Console panel displays a list of log entries, with the most recent entry being 'typing text'. The entry is highlighted in blue. The console also shows a search bar with the text 'Filter' and a 'Default levels' dropdown menu.

Olethan jo asentanut React Developer Toolsin? Developer Toolsista näet, miten tila muuttuu syötekenttään kirjoitettaessa:

The screenshot shows the React DevTools Components panel. The 'App' component is selected, and its state is displayed as `State: "typing tex"`. The state is highlighted with a red box. The console also shows three log messages: `Note key="0"`, `Note key="1"`, and `Note key="2"`.

Nyt komponentin *App* tila `newNote` heijastaa koko ajan syötekentän arvoa, joten voimme viimeistellä uuden muistiinpanon lisäämisestä huolehtivan metodin `addNote` :

```
const addNote = (event) => {
  event.preventDefault()
  const noteObject = {
    content: newNote,
    important: Math.random() > 0.5,
    id: notes.length + 1,
  }

  setNotes(notes.concat(noteObject))
}
```

```
setNewNote('')
}
```

Ensin luodaan uutta muistiinpanoa vastaava olio `noteObject`, jonka sisältökentän arvo saadaan komponentin tilasta `newNote`. Yksikäsitteinen tunnus eli *id* generoidaan kaikkien muistiinpanojen lukumäärän perusteella. Koska muistiinpanoja ei poisteta, menetelmä toimii sovelluksessamme. Komennon `Math.random()` avulla muistiinpanosta tulee 50 %:n todennäköisyydellä tärkeä.

Uusi muistiinpano lisätään vanhojen joukkoon oikeaoppisesti käyttämällä osasta 1 tuttua taulukon metodia concat:

```
setNotes(notes.concat(noteObject))
```

copy

Metodi ei muuta alkuperäistä tilaa `notes` vaan luo *uuden taulukon, joka sisältää myös lisättävän alkion*. Tämä on tärkeää, sillä Reactin tilaa ei saa muuttaa suoraan!

Tapahtumankäsittelijä tyhjentää myös syötekenttää kontrolloivan tilan `newNote` sen funktiolla `setNewNote`.

```
setNewNote('')
```

copy

Sovelluksen tämänhetkinen koodi on kokonaisuudessaan GitHubissa, branchissä *part2-2*.

Näytettävien elementtien filtteröinti

Tehdään sovellukseen toiminto, joka mahdollistaa ainoastaan tärkeiden muistiinpanojen näyttämisen.

Lisätään komponentin *App* tilaan tieto siitä, näytetäänkö muistiinpanoista kaikki vai ainoastaan tärkeät:

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

  // ...
}
```

copy

Muutetaan komponenttia siten, että se tallettaa muuttujaan `notesToShow` näytettävien muistiinpanojen listan riippuen siitä, tuleeko näyttää kaikki vai vain tärkeät:

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

copy



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

  // ...

  const notesToShow = showAll
    ? notes
    : notes.filter(note => note.important === true)

  return (
    <div>
      <h1>Notes</h1>
      <ul>
        {notesToShow.map(note =>
          <Note key={note.id} note={note} />
        )}
      </ul>
      // ...
    </div>
  )
}
```

Muuttujan `notesToShow` määrittely on melko kompakti:

```
const notesToShow = showAll
  ? notes
  : notes.filter(note => note.important === true)
```

[copy](#)

Käytössä on monissa muissakin kielissä oleva ehdollinen operaattori.

Lausekkeella

```
const tulos = ehto ? val1 : val2
```

[copy](#)

muuttujan `tulos` arvoksi asetetaan `val1` :n arvo jos `ehto` on tosi. Jos `ehto` ei ole tosi, muuttujan `tulos` arvoksi tulee `val2` :n arvo.

Eli jos tilan arvo `showAll` on epätosi, muuttuja `notesToShow` saa arvokseen vain ne muistiinpanot, joiden `important` -kentän arvo on tosi. Filtröinti tapahtuu taulukon metodilla `filter`:

```
notes.filter(note => note.important === true)
```

[copy](#)

Vertailuoperaatio on oikeastaan turha. Koska `note.important` on arvoltaan joko *true* tai *false*,[↑] riittää kun kirjoitamme:


```
notes.filter(note => note.important)
```

copy

Tässä käytettiin kuitenkin ensin vertailuoperaattoria mm. korostamaan erästä tärkeää seikkaa: JavaScriptissa `arvo1 == arvo2` ei toimi kaikissa tilanteissa loogisesti ja onkin varmempi käyttää aina vertailuissa muotoa `arvo1 === arvo2`. Enemmän aiheesta on [täällä](#).

Filtteröinnin toimivuutta voi jo nyt kokeilla vaihtelemalla sitä, miten tilan kentän `showAll` alkuarvo määritellään funktion `useState` parametrina.

Lisätään sitten toiminnallisuus, joka mahdollistaa `showAll` :in tilan muuttamisen sovelluksesta:

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

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

  // ...

  const notesToShow = showAll
    ? notes
    : notes.filter(note => note.important)

  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} />
        )}
      </ul>
      // ...
    </div>
  )
}
```

copy

Näkyviä muistiinpanoja (kaikki vai ainoastaan tärkeät) siis kontrolloidaan napin avulla. Napin tapahtumankäsittelijä on niin yksinkertainen, että se on kirjoitettu suoraan napin attribuutiksi. Tapahtumankäsittelijä muuttaa `showAll` :n arvon truesta falseksi ja päinvastoin:

```
() => setShowAll(!showAll)
```

copy ↑

Napin teksti riippuu tilan `showAll` arvosta:

```
show {showAll ? 'important' : 'all' }
```

[copy](#)

Sovelluksen tämänhetkinen koodi on kokonaisuudessaan [GitHubissa](#) branchissa *part2-3*.

Tehtävät 2.6.-2.10.

Seuraavassa tehtävässä aloitettavaa ohjelmaa kehitellään eteenpäin muutamassa seuraavassa tehtävässä. Tässä ja kurssin aikana muissakin vastaantulevissa tehtäväsarjoissa ohjelman lopullisen version palauttaminen riittää, voit toki halutessasi tehdä commitin jokaisen tehtävän jälkeisestä tilanteesta, mutta se ei ole välttämätöntä.

2.6: puhelinluettelo step1

Toteutetaan yksinkertainen puhelinluettelo. ***Aluksi luetteloon lisätään vain nimiä.***

Toteutetaan tässä tehtävässä henkilön lisäys puhelinluetteloon.

Voit ottaa sovelluksesi komponentin *App* pohjaksi seuraavan:

```
import { useState } from 'react'

const App = () => {
  const [persons, setPersons] = useState([
    { name: 'Arto Hellas' }
  ])
  const [newName, setNewName] = useState('')

  return (
    <div>
      <h2>Phonebook</h2>
      <form>
        <div>
          name: <input />
        </div>
        <div>
          <button type="submit">add</button>
        </div>
      </form>
      <h2>Numbers</h2>
      ...
    </div>
  )
}

export default App
```

[copy](#)

Tila `newName` on tarkoitettu lomakkeen kentän kontrollointiin.

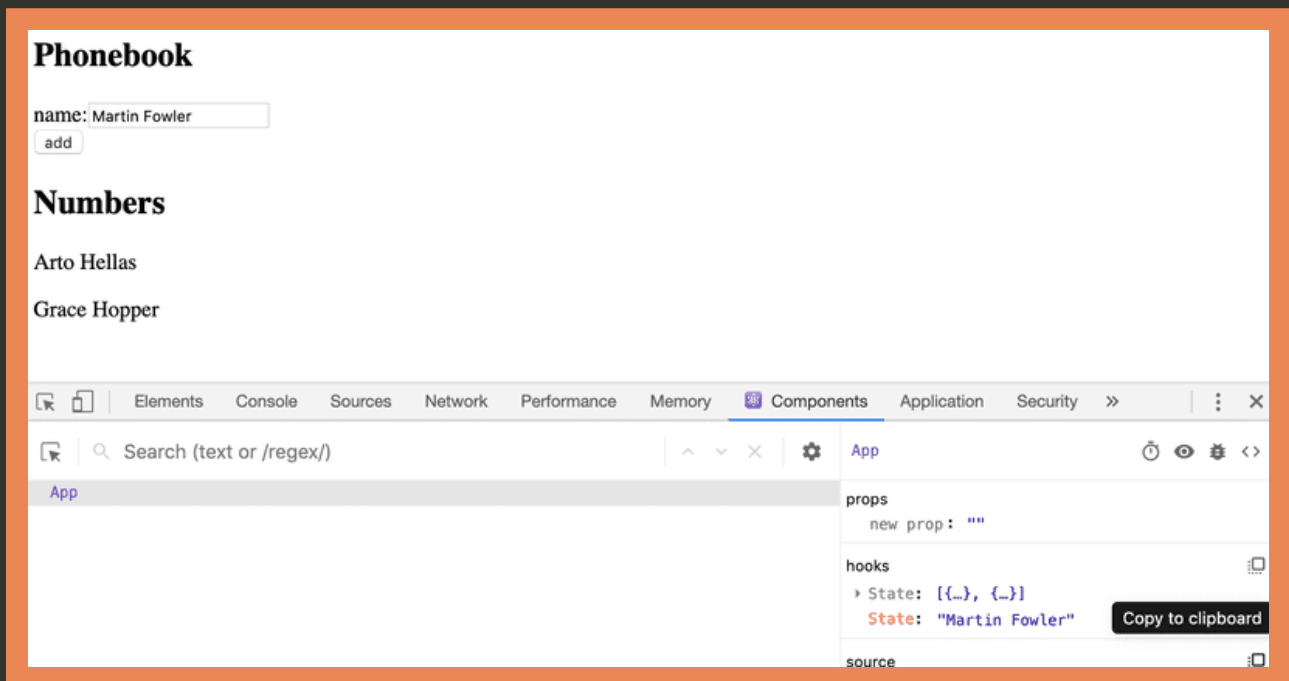
Joskus tilaa tallettavia ja tarvittaessa muitakin muuttujia voi olla hyödyllistä renderöidä debugatessa komponenttiin, eli voit tilapäisesti lisätä komponentin palauttamaan koodiin esim. seuraavan:

```
<div>debug: {newName}</div>
```

[copy](#)

Muista myös osan 1 luku React-sovellusten debuggaus, erityisesti React Developer Tools on välillä todella kätevä komponentin tilan muutosten seuraamisessa.

Sovellus voi näyttää tässä vaiheessa seuraavalta:



Huomaa React Developer Toolsin käyttö!

Huom:

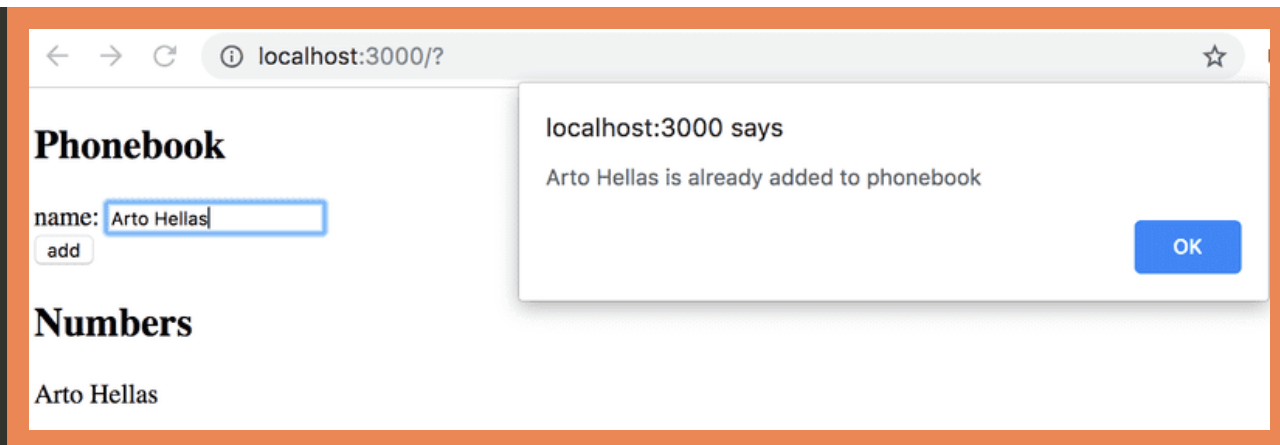
- voit käyttää kentän *key* arvona henkilön nimeä
- muista estää lomakkeen lähetyksen oletusarvoinen toiminta

2.7: puhelinluettelo step2

Jos lisättävä nimi on jo sovelluksen tiedossa, estä lisäys. Taulukolla on lukuisia sopivia metodeja tehtävän tekemiseen.

Anna tilanteessa virheilmoitus komennolla alert:





Muistutus edellisestä osasta: kun muodostat JavaScriptissä muuttujaan perustuvan merkkijonon, tyylikkään tapa asian hoitamiseen on template string:

```
`${newName} is already added to phonebook`
```

[copy](#)

Jos muuttujalla `newName` on arvona *Arto Hellas*, on tuloksena merkkijono

```
`Arto Hellas is already added to phonebook`
```

[copy](#)

Template stringin käyttö antaa ammattimaisen vaikutelman, vaikka sama toki hoituisi javamaisesti myös merkkijonojen plus-metodilla:

```
newName + ' is already added to phonebook'
```

[copy](#)

2.8: puhelinluettelo step3

Lisää sovellukseen mahdollisuus antaa henkilöille puhelinnumero. Tarvitset siis lomakkeeseen myös toisen *input*-elementin (ja sille oman muutoksenkäsittelijän):

```
<form>
  <div>name: <input /></div>
  <div>number: <input /></div>
  <div><button type="submit">add</button></div>
</form>
```

[copy](#)

Sovellus voi näyttää tässä vaiheessa seuraavalta. Kuvassa myös React Developer Tools:in tarjoama näkymä komponentin *App* tilaan:

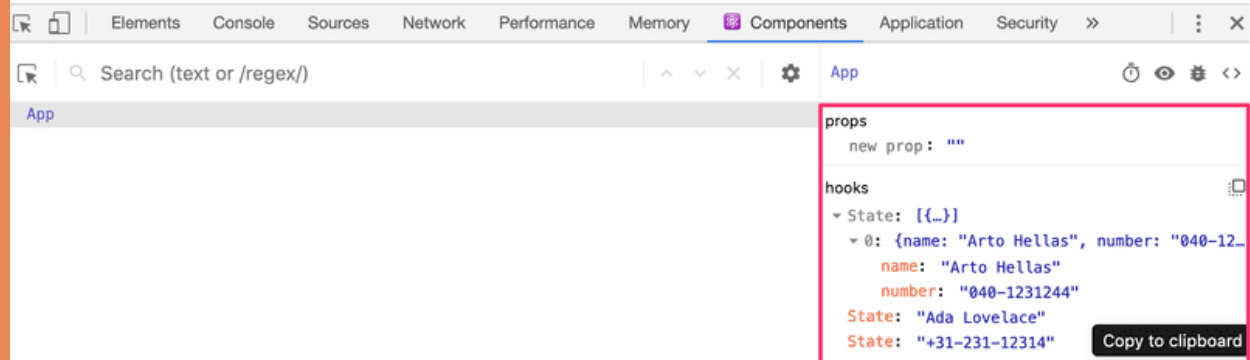


Phonebook

name:
number:

Numbers

Arto Hellas 040-1231244



2.9*: puhelinluettelo step4

Tee lomakkeeseen hakukenttä, jonka avulla näytettävien nimien listaa voidaan rajata:

Phonebook

filter shown with

add a new

name:
number:

Numbers

Arto Hellas 040-123456
Ada Lovelace 39-44-5323523

Rajausehdon syöttämisen voi hoitaa omana lomakkeeseen kuulumattomana *input*-elementtinä. Kuvassa rajausehdosta on tehty *case-insensitiivinen* eli ehto *arto* löytää isolla kirjaimella kirjoitetun Arton.

Huom: Testiaineiston syöttäminen manuaalisesti selainta käyttäen on useimmiten turhaa manuaalista työtä. Yleensä on järkevämpää 'kovakoodata' sovellukseen jotain testidataa:

```
const App = () => {  
  const [persons, setPersons] = useState([  
    { name: 'Arto Hellas', number: '040-123456' },  
  ])
```



```
{ name: 'Ada Lovelace', number: '39-44-5323523' },  
{ name: 'Dan Abramov', number: '12-43-234345' },  
{ name: 'Mary Poppendieck', number: '39-23-6423122' }  
])  
  
// ...  
}
```

`{() => fs}`



komponentteja. Pidä kuitenkin edelleen kaikki tila- sekä tapahtumankäsittelijäfunktiot juurikomponentissa *App*.

Riittää että erotat sovelluksesta *kolme* komponenttia. Hyviä kandidaatteja ovat filteröintilomake, uuden henkilön lisäävä lomake, kaikki henkilöt renderöivä komponentti sekä yksittäisen henkilön renderöivä komponentti.

Sovelluksen juurikomponentin ei tarvitse refaktoroinnin jälkeen renderöidä suoraan muuta kuin otsikoita. Komponentti voi näyttää suunnilleen seuraavalta:

```
const App = () => {  
  // ...  
  
  return (  
    <div>  
      <h2>Phonebook</h2>  
  
      <Filter ... />  
  
      <h3>Add a new</h3>  
  
      <PersonForm  
        ...  
      />  
  
      <h3>Numbers</h3>  
  
      <Persons ... />  
    </div>  
  )  
}
```

copy

HUOM: Saatat törmätä ongelmiin jos määrittelet komponentteja "väärässä paikassa". Nyt kannattaakin ehdottomasti kerrata edellisen osan luku älä määrittele komponenttia komponentin sisällä.

Ehdota muutosta materiaalin sisältöön



[Edellinen osa](#)[Seuraava osa](#)[Kurssista](#)[Kurssin sisältö](#)[FAQ](#)[Kurssilla mukana](#)[Haaste](#)**UNIVERSITY OF HELSINKI**

HOUSTON

