



## e Tyylien lisääminen React-sovellukseen

Sovelluksemme ulkoasu on tällä hetkellä hyvin vaatimaton. Osaan 0 liittyvässä tehtävässä 0.2 tutustuttiin Mozillan CSS-tutoriaaliin.

Katsotaan vielä tämän osan lopussa nopeasti kahta tapaa liittää tyylejä React-sovellukseen. Tapoja on useita, ja tulemme tarkastelemaan muita myöhemmin. Ensimmäisenä liitämme CSS:n sovellukseemme vanhan kansan tapaan yksittäisenä tiedostona, joka on kirjoitettu käsin ilman esiprosessorien apua (tulemme myöhemmin huomaamaan, että tämä ei ole täysin totta).

Tehdään sovelluksen hakemistoon *src* tiedosto *index.css* ja liitetään se sovellukseen lisäämällä tiedostoon *main.jsx* seuraava import:

```
import './index.css'
```

[copy](#)

Lisätään seuraava sääntö tiedostoon *index.css*:

```
h1 {  
  color: green;  
}
```

[copy](#)

CSS-säännöt koostuvat valitsimesta eli *selektorista* ja määrittelystä eli *deklaraatiosta*. Valitsin määrittelee, mihin elementteihin sääntö kohdistuu. Valitsimena on nyt *h1* eli kaikki sovelluksessa käytetyt *h1*-otsikkotägit.

Määrittelyosa asettaa ominaisuuden `color` eli fontin värin arvoksi vihreän (*green*).

Sääntö voi sisältää mielivaltaisen määrän määrittelyjä. Muutetaan edellistä siten, että tekstistä tulee kursivoitua eli fontin tyyliksi asetetaan *italic*:

```
h1 {  
  color: green;  
  font-style: italic;
```

[copy](#)

```
font-style: italic;
```

```
}
```

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



muistiinpanot ovat *li*-tagien sisällä:

```
const Note = ({ note, toggleImportance }) => {  
  const label = note.important  
    ? 'make not important'  
    : 'make important';  
  
  return (  
    <li>  
      {note.content}  
      <button onClick={toggleImportance}>{label}</button>  
    </li>  
  )  
}
```

copy

Lisätään tyyli tiedostoon seuraava (koska osaamiseni tyylikkaiden web-sivujen tekemiseen on lähellä nollaa, nyt käytettävissä tyyliissä ei ole sinänsä mitään järkeä):

```
li {  
  color: grey;  
  padding-top: 3px;  
  font-size: 15px;  
}
```

copy

Tyylien kohdistaminen elementtityyppeihin on kuitenkin ongelmallista. Jos sovelluksessa on myös muita *li*-tageja, kaikki saavat samat tyyli.

Jos haluamme kohdistaa tyyliä nimenomaan muistiinpanoihin, on parempi käyttää class selectoreja.

Normaalissa HTML:ssä luokat määritellään elementtien attribuutin *class* arvona:

```
<li class="note">tekstiä</li>
```

copy

Reactissa tulee kuitenkin classin sijaan käyttää attribuuttia className, joten muutetaan komponenttia *Note* seuraavasti:

```
const Note = ({ note, toggleImportance }) => {  
  const label = note.important  
    ? 'make not important'  
    : 'make important';  
  
  return (  
    <li className='note'>  
      {note.content}  
      <button onClick={toggleImportance}>{label}</button>  
    </li>  
  )  
}
```

copy



```
)  
}
```

Luokkaselektori määritellään syntaksilla `.classname` eli:

```
.note {  
  color: grey;  
  padding-top: 5px;  
  font-size: 15px;  
}
```

copy

Jos nyt lisäät sovellukseen muita li-elementtejä, ne eivät saa muistiinpanoille määriteltyjä tyylejä.

## Parempi virheilmoitus

Aiemmin toteutimme olemassa olemattoman muistiinpanon tärkeyden muutokseen liittyvän virheilmoituksen `alert` -metodilla. Toteutetaan se nyt Reactilla omana komponenttinaan.

Komponentti on yksinkertainen:

```
const Notification = ({ message }) => {  
  if (message === null) {  
    return null  
  }  
  
  return (  
    <div className="error">  
      {message}  
    </div>  
  )  
}
```

copy

Jos propsin `message` arvo on `null`, ei renderöidä mitään. Muussa tapauksessa renderöidään viesti `div`-elementtiin. Elementille on liitetty tyylien lisäämistä varten luokka *error*.

Lisätään komponentin *App* tilaan kenttä *errorMessage* virheviestiä varten. Laitetaan kentälle heti jotain sisältöä, jotta pääsemme testaamaan komponenttia:

```
const App = () => {  
  const [notes, setNotes] = useState([])  
  const [newNote, setNewNote] = useState('')  
  const [showAll, setShowAll] = useState(true)  
  const [errorMessage, setErrorMessage] = useState('some error happened...')  
  
  // ...  
  
  return (  
    <div>  
      <h1>Notes</h1>  
      <Notification message={errorMessage} />  
      <div>  
        <button onClick={() => setShowAll(!showAll)}>  
          show {showAll ? 'important' : 'all'}  
        </button>  

```

copy



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

Lisätään sitten virheviestille sopiva tyyli:

```
.error {
  color: red;
  background: lightgrey;
  font-size: 20px;
  border-style: solid;
  border-radius: 5px;
  padding: 10px;
  margin-bottom: 10px;
}
```

copy

Nyt olemme valmiina lisäämään virheviestin logiikan. Muutetaan metodia `toggleImportanceOf` seuraavasti:

```
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 => {
      setErrorMessage(
        `Note '${note.content}' was already removed from server`
      )
      setTimeout(() => {
        setErrorMessage(null)
      }, 5000)
      setNotes(notes.filter(n => n.id !== id))
    })
}
```

copy

Virheen yhteydessä asetetaan tilaan `errorMessage` sopiva virheviesti. Samalla käynnistetään ajastin, joka asettaa viiden sekunnin kuluttua tilan `errorMessage` -kentän arvoksi `null`.

Lopputulos näyttää seuraavalta:





Sovelluksen tämänhetkinen koodi on kokonaisuudessaan [GitHubissa](#), branchissa *part2-7*.

## Inline-tyylit

React mahdollistaa tyylien kirjoittamisen myös suoraan komponenttien koodin joukkoon niin sanoittuina [inline-tyyleinä](#).

Periaate inline-tyyliä määrittelyssä on erittäin yksinkertainen. Mihin tahansa React-komponenttiin tai elementtiin voi liittää attribuutin [style](#), jolle annetaan arvoksi JavaScript-oliona määritelty joukko CSS-sääntöjä.

CSS-säännöt määritellään JavaScriptin avulla hieman eri tavalla kuin normaaleissa CSS-tiedostoissa. Jos haluamme asettaa jollekin elementille esimerkiksi vihreän, kursivoidun ja 16 pikselin korkuisen fontin, määrittely ilmaistaan CSS-syntaksilla seuraavasti:


```
{
  color: green;
  font-style: italic;
  font-size: 16px;
}
```

copy

Vastaava tyyli kirjoitetaan Reactin inline-tyylin määrittelevänä oliona seuraavasti:

```
{
  color: 'green',
  fontStyle: 'italic',
  fontSize: 16
}
```

copy

Jokainen CSS-sääntö on olion kenttä, joten ne erotetaan JavaScript-syntaksin mukaan pilkuilla. Pikseleinä ilmaistut numeroarvot voidaan määritellä kokonaislukuina. Merkittävin ero normaaliin CSS:ään on väliviivan sisältämien CSS-ominaisuuksien kirjoittaminen camelCase-muodossa. 

Voimme nyt lisätä sovellukseemme alapalkin muodostavan komponentin *Footer* ja määritellä sille inline-tyylit seuraavasti:

```
const Footer = () => {  
  const footerStyle = {  
    color: 'green',  
    fontStyle: 'italic',  
    fontSize: 16  
  }  
  
  return (  
    <div style={footerStyle}>  
      <br />  
      <em>Note app, Department of Computer Science, University of Helsinki 2023</em>  
    </div>  
  )  
}  
  
const App = () => {  
  // ...  
  
  return (  
    <div>  
      <h1>Notes</h1>  
  
      <Notification message={errorMessage} />  
  
      // ...  
  
      <Footer />  
    </div>  
  )  
}
```

copy

Inline-tyyleillä on tiettyjä rajoituksia, esim. ns. pseudo-selektoreja ei ole mahdollisuutta käyttää (ainakaan helposti).

Inline-tyylit ja muutamat myöhemmin kurssilla katsomamme tavat lisätä tyylejä Reactiin ovat periaatteessa täysin vastoin vanhoja hyviä periaatteita, joiden mukaan web-sovellusten ulkoasujen määrittely eli CSS tulee erottaa sisällön (HTML) ja toiminnallisuuden (JavaScript) määrittelystä. Vanha koulukunta pyrkiikin siihen, että sovelluksen CSS, HTML ja JavaScript kirjoitetaan omiin tiedostoihinsa.

CSS:n, HTML:n ja JavaScriptin erottelu omiin tiedostoihinsa ei kuitenkaan ole välttämättä erityisen skaalautuva ratkaisu suurissa ohjelmistoissa. Reactissa onkin periaatteena jakaa sovelluksen koodi eri tiedostoihin noudattaen sovelluksen loogisia toiminnallisia kokonaisuuksia.

Toiminnallisen kokonaisuuden strukturointiyksikkö on React-komponentti, joka määrittelee niin sisällön rakenteen kuvaavan HTML:n, toiminnan määrittelevät JavaScript-funktiot kuin komponentin tyylinkin yhdessä paikassa siten, että komponenteista tulee mahdollisimman riippumattomia ja yleiskäyttöisiä.

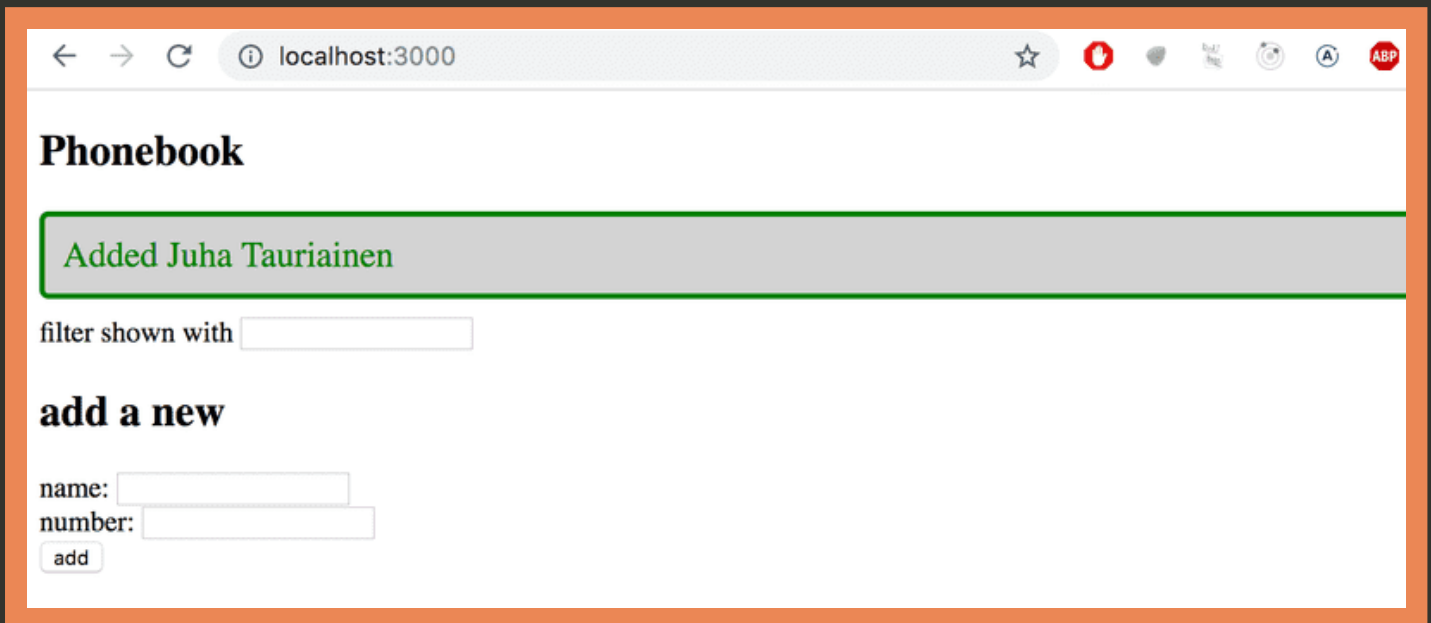
Sovelluksen lopullinen koodi on kokonaisuudessaan [GitHubissa](#), branchissa *part2-8*.



## Tehtävät 2.16.-2.17.

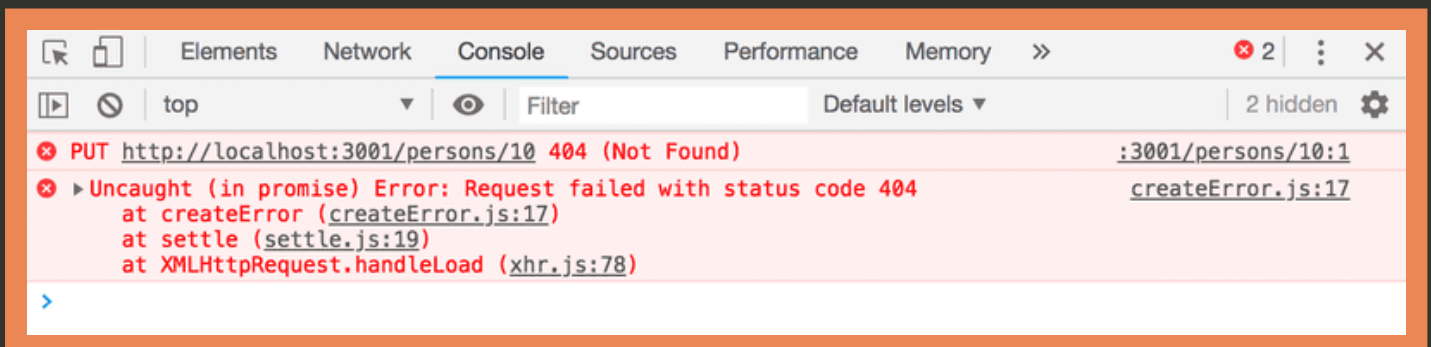
### 2.16: puhelinluettelo step11

Toteuta osan 2 esimerkin parempi virheilmoitus tyyliin ruudulla muutaman sekunnin näkyvä ilmoitus, joka kertoo onnistuneista operaatioista (henkilön lisäys ja poisto sekä numeron muutos):



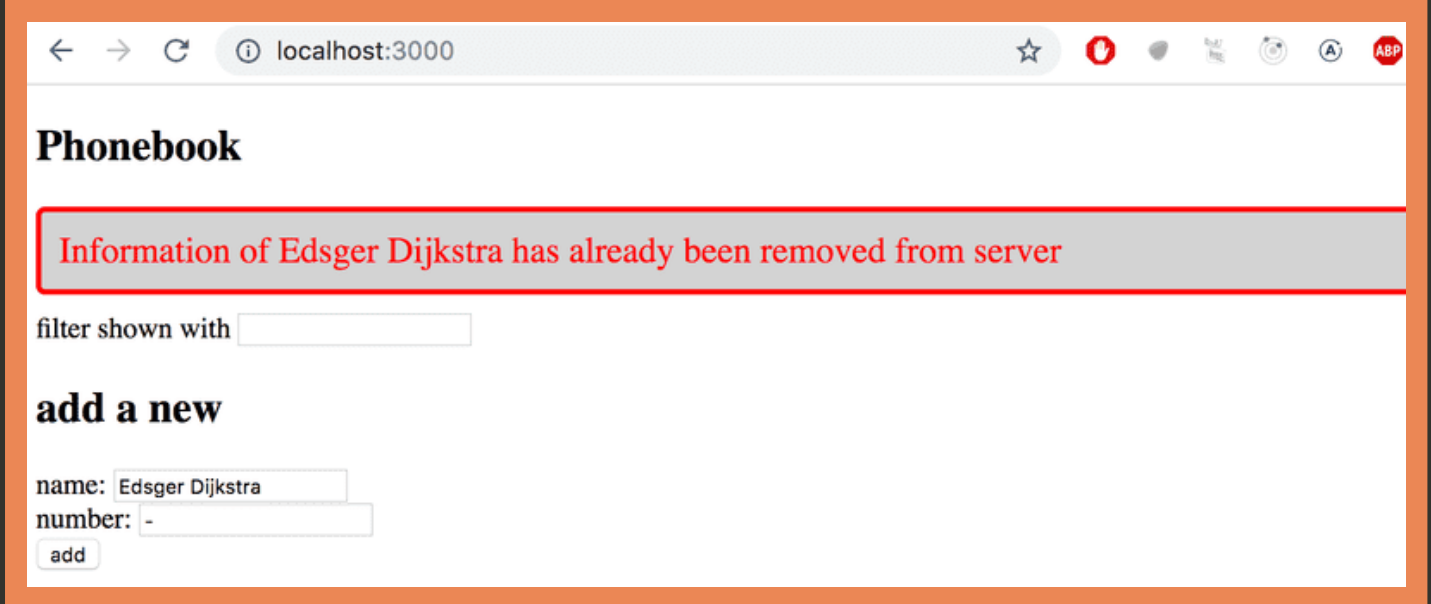
### 2.17\*: puhelinluettelo step12

Avaa sovelluksesi kahteen selaimeen. **Jos poistat jonkun henkilön selaimella 1** hieman ennen kuin yrität *muuttaa henkilön numeroa* selaimella 2, tapahtuu virhetilanne:



Korjaa ongelma osan 2 esimerkin promise ja virheet hengessä ja siten, että käyttäjälle ilmoitetaan operaation epäonnistumisesta. Onnistuneen ja epäonnistuneen operaation ilmoitusten tulee erota toisistaan:





**HUOM:** Vaikka käsittelet poikkeuksen koodissa, virheilmoitus tulostuu silti konsoliin.

## Muutama tärkeä huomio

Osan lopussa on vielä muutama hieman haastavampi tehtävä. Voit tässä vaiheessa jättää tehtävät tekemättä jos ne tuottavat liian paljon päänvaivaa, palaamme samoihin teemoihin uudelleen myöhemmin. Materiaali kannattanee jokatapauksessa lukea läpi.

Eräs sovelluksessamme tekemä ratkaisu piilottaa yhden hyvin tyypillisen virhetilanteen, mihin tulet varmasti törmäämään monta kertaa.

Alustimme muistiinpanot muistavan tilan alkuarvoksi tyhjän taulun:

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

copy

Tämä on onkin luonnollinen tapa alustaa tila, muistiinpanot muodostavat joukon, joten tyhjä taulukko on luonteva alkuarvo muuttujalle.

Niissä tilanteissa, missä tilaan talletetaan "yksi asia" tilan luonteva alkuarvo on usein `null`, joka kertoo että tilassa ei ole vielä mitään. Kokeillaan miten käy jos alustamme nyt tilan nulliksi:

```
const App = () => {  
  const [notes, setNotes] = useState(null)  
  
  // ...  
}
```

copy

Sovellus hajooa:





```
✖ ▶ Uncaught TypeError: Cannot read properties of null (reading 'map')
    at App (App.js:53:1)
    at renderWithHooks (react-dom.development.js:16305:1)
    at mountIndeterminateComponent (react-dom.development.js:20074:1)
    at beginWork (react-dom.development.js:21587:1)
    at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)
    at invokeGuardedCallback (react-dom.development.js:4277:1)
    at beginWork$1 (react-dom.development.js:27451:1)
    at performUnitOfWork (react-dom.development.js:26557:1)
    at workLoopSync (react-dom.development.js:26466:1)
```

Virheilmoitus kertoo vian syyn ja sijainnin. Ongelmallinen kohta on seuraava:

```
// notesToShow gets the value of notes
const notesToShow = showAll
  ? notes
  : notes.filter(note => note.important)

// ...

{notesToShow.map(note =>
  <Note key={note.id} note={note} />
)}
```

copy

Virheilmoitus siis on

```
Cannot read properties of null (reading 'map')
```

copy

Muuttuja `notesToShow` saa arvokseen tilan `notes` arvon ja koodi yrittää kutsua olemattomalle oliolle (jonka arvo on null) metodia `map`.

Mistä tämä johtuu?

Efektohookki asettaa tilaan `notes` palvelimen palauttavat muistiinpanot funktiolla `setNotes` :

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

copy

Ongelma on kuitenkin siinä, että efekti suoritetaan vasta *ensimmäisen renderöinnin jälkeen*. Koska tilalle `notes` on asetettu alkuarvo null:

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

  // ...
```

copy



ensimmäisen renderöinnin tapahtuessa tullaan suorittamaan

```
notesToShow = notes

// ...

notesToShow.map(note => ...)
```

copy

ja tämä aiheuttaa ongelman, sillä arvolle `null` ei voida kutsua metodia `map` .

Kun annoimme tilalle `notes` alkuarvoksi tyhjän taulukon, ei samaa ongelmaa esiinny, tyhjälle taulukolle on luvallista kutsua metodia `map` .

Sopiva tilan alustaminen siis "peitti" ongelman, joka johtuu siitä että muistiinpanoja ei ole vielä alustettu palvelimelta haettavalla datalla.

Toinen tapa kiertää ongelma on tehdä *ehdollinen renderöinti*, ja palauttaa ainoastaan `null` jos komponentin tila ei ole vielä alustettu:

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

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

  // do not render anything if notes is still null
  if (!notes) {
    return null
  }

  // ...
}
```

copy

Nyt ensimmäisellä renderöinnillä ei renderöidä mitään. Kun muistiinpanot saapuvat palvelimelta, asetetaan ne tilaan `notes` kutsumalla funktiota `setNotes` . Tämä saa aikaan uuden renderöinnin ja muistiinpanot piirtyvät ruudulle.

Tämä tapa sopii erityisesti niihin tilanteisiin, joissa tilaa ei voi alustaa muuten komponentille sopivaan, renderöinnin mahdollistavaan alkuarvoon kuten tyhjäksi taulukoksi.

Toinen huomiomme liittyy `useEffect`in toiseen parametriin:

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

copy



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

Funktion `useEffect` toista parametria käytetään tarkentamaan sitä, miten usein efekti suoritetaan. Periaate on se, että efekti suoritetaan aina ensimmäisen renderöinnin yhteydessä *ja* silloin kuin toisena parametrina olevan taulukon sisältö muuttuu.

Kun toisena parametrina on tyhjä taulukko `[]`, sen sisältö ei koskaan muutu ja efekti suoritetaan ainoastaan komponentin ensimmäisen renderöinnin jälkeen. Tämä on juuri se mitä haluamme kun alustamme sovelluksen tilan.

On kuitenkin tilanteita, missä efekti halutaan suorittaa muulloinkin, esim. komponentin tilan muuttuessa sopivalla tavalla.

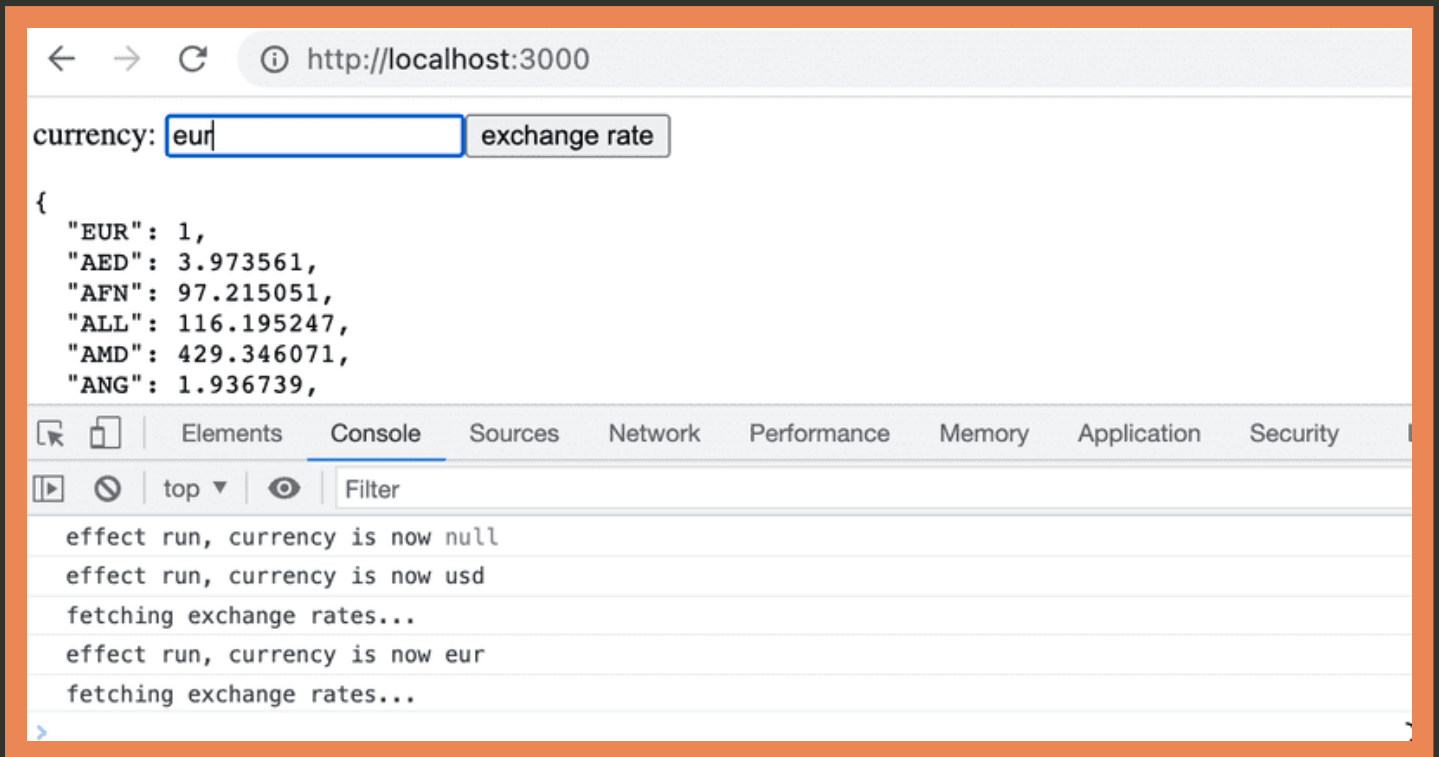
Tarkastellaan seuraavaa yksinkertaista sovellusta, jonka avulla voidaan kysellä valuuttojen vaihtokursseja Exchange rate API -palvelusta:

```
import { useState, useEffect } from 'react'  
import axios from 'axios'  
  
const App = () => {  
  const [value, setValue] = useState('')  
  const [rates, setRates] = useState({})  
  const [currency, setCurrency] = useState(null)  
  
  useEffect(() => {  
    console.log('effect run, currency is now', currency)  
  
    // skip if currency is not defined  
    if (currency) {  
      console.log('fetching exchange rates...')  
      axios  
        .get(`https://open.er-api.com/v6/latest/${currency}`)  
        .then(response => {  
          setRates(response.data.rates)  
        })  
    }  
  }, [currency])  
  
  const handleChange = (event) => {  
    setValue(event.target.value)  
  }  
  
  const onSearch = (event) => {  
    event.preventDefault()  
    setCurrency(value)  
  }  
  
  return (  
    <div>  
      <form onSubmit={onSearch}>  
        currency: <input value={value} onChange={handleChange} />  
        <button type="submit">exchange rate</button>  
      </form>  
      <pre>  
        {JSON.stringify(rates, null, 2)}  
      </pre>  
    </div>  
  )  
}
```

copy



Sovelluksen käyttöliittymässä on lomake, jonka syötekenttään halutun valuutan nimi kirjoitetaan. Jos valuutta on olemassa, renderöi sovellus valuutan vaihtokurssit muihin valuuttoihin:



Sovellus asettaa käyttäjän hakulomakkeelle kirjoittaman valuutan nimen tilaan `currency` sillä hetkellä kun nappia painetaan.

Kun `currency` saa uuden arvon, sovellus tekee `useEffect`in määrittelemässä funktiossa haun valuuttakurssit kertovaan rajapintaan:

```
const App = () => {
  // ...
  const [currency, setCurrency] = useState(null)

  useEffect(() => {
    console.log('effect run, currency is now', currency)

    // skip if currency is not defined
    if (currency) {
      console.log('fetching exchange rates...')
      axios
        .get(`https://open.er-api.com/v6/latest/${currency}`)
        .then(response => {
          setRates(response.data.rates)
        })
    }
  }, [currency])
  // ...
}
```

Efektifunktio siis suoritetaan ensimmäisen renderöinnin jälkeen, ja *aina* sen jälkeen kun sen toisena parametrina oleva taulukko eli esimerkin tapauksessa `[currency]` muuttuu. Eli kun tila `currency` saa uuden arvon, muuttuu taulukon sisältö ja efektifunktio suoritetaan.



```
if (currency) {  
  // haetaan valuuttakurssit  
}
```

copy

joka estää valuuttakurssien hakemisen ensimmäisen renderöinnin yhteydessä, eli siinä vaiheessa kuin muuttujalla `currency` on vasta alkuarvo eli tyhjää merkkijono.

Jos käyttäjä siis kirjoittaa hakukenttään esim. *eur*, suorittaa sovellus Axiosin avulla HTTP GET -pyynnön osoitteeseen <https://open.er-api.com/v6/latest/eur> ja tallentaa vastauksen tilaan `rates`.

Kun käyttäjä tämän jälkeen kirjoittaa hakukenttään jonkin toisen arvon, esim. *usd* suoritetaan efekti jälleen ja uuden valuutan kurssit haetaan.

Tässä esitelty tapa API-kyselyjen tekemiseen saattaa tuntua hieman hankalalta. Tämä kyseinen sovellus olisikin voitu tehdä kokonaan ilman `useEffectin` käyttöä, ja tehdä API-kyselyt suoraan lomakkeen napin painamisen hoitavassa käsittelijäfunktiossa:

```
const onSearch = (event) => {  
  event.preventDefault()  
  axios  
    .get(`https://open.er-api.com/v6/latest/${value}`)  
    .then(response => {  
      setRates(response.data.rates)  
    })  
}
```

copy

On kuitenkin tilanteita, missä vastaava tekniikka ei onnistu. Esim. eräs tapa tehtävässä 2.20 tarvittavien kaupungin säätiöiden hakemiseen on nimenomaan `useEffectin` hyödyntäminen. Tehtävässä selviää myös hyvin ilman kyseistä kikkaa, esim. malliratkaisu ei sitä tarvitse.

## Tehtävät 2.18.-2.20.

### 2.18\* maiden tiedot, step1

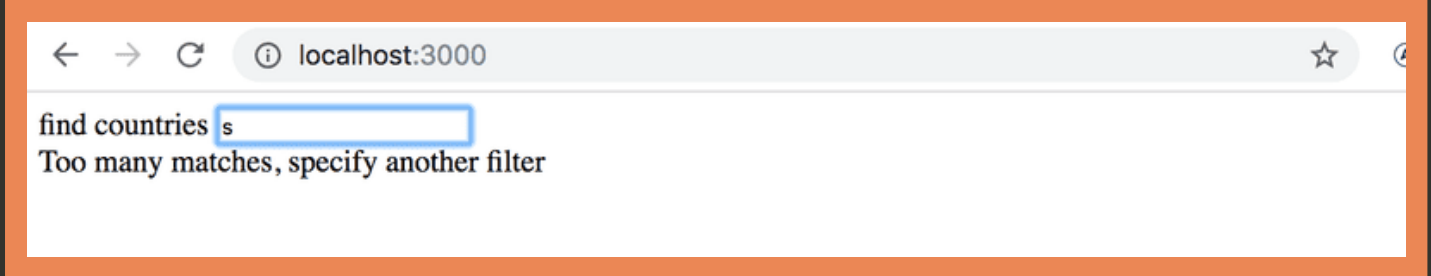
Siirrytään osan lopuksi hieman toisenlaiseen teemaan.

Osoitteesta <https://studies.cs.helsinki.fi/restcountries/> löytyy palvelu, joka tarjoaa paljon eri maihin liittyvää tietoa koneluettavassa muodossa ns. REST API:n välityksellä. Tee sovellus, jonka avulla voit tarkastella eri maiden tietoja.

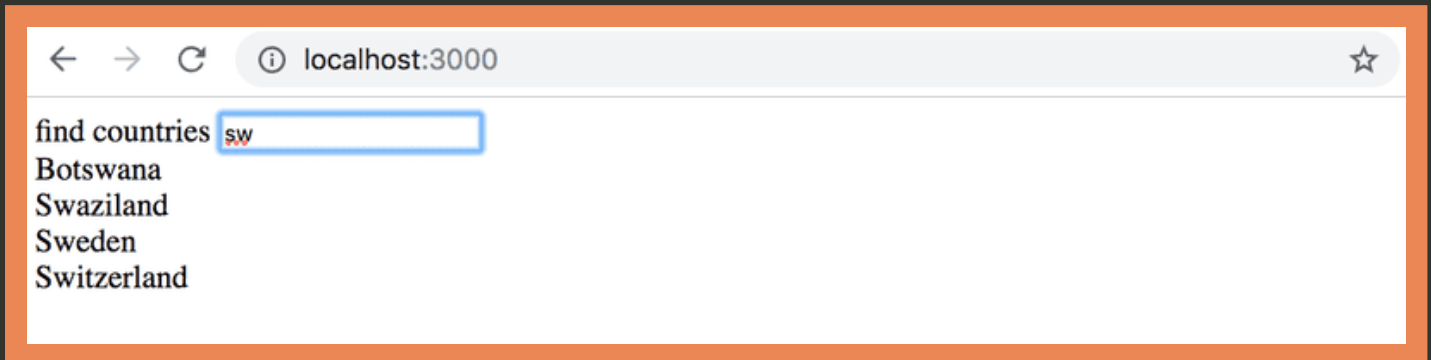
Sovelluksen käyttöliittymä on yksinkertainen. Näytettävä maa haetaan kirjoittamalla hakuehto hakukenttään.

Jos ehdon täyttäviä maita on liikaa (yli kymmenen), kehoitetaan tarkentamaan hakuehtoa:

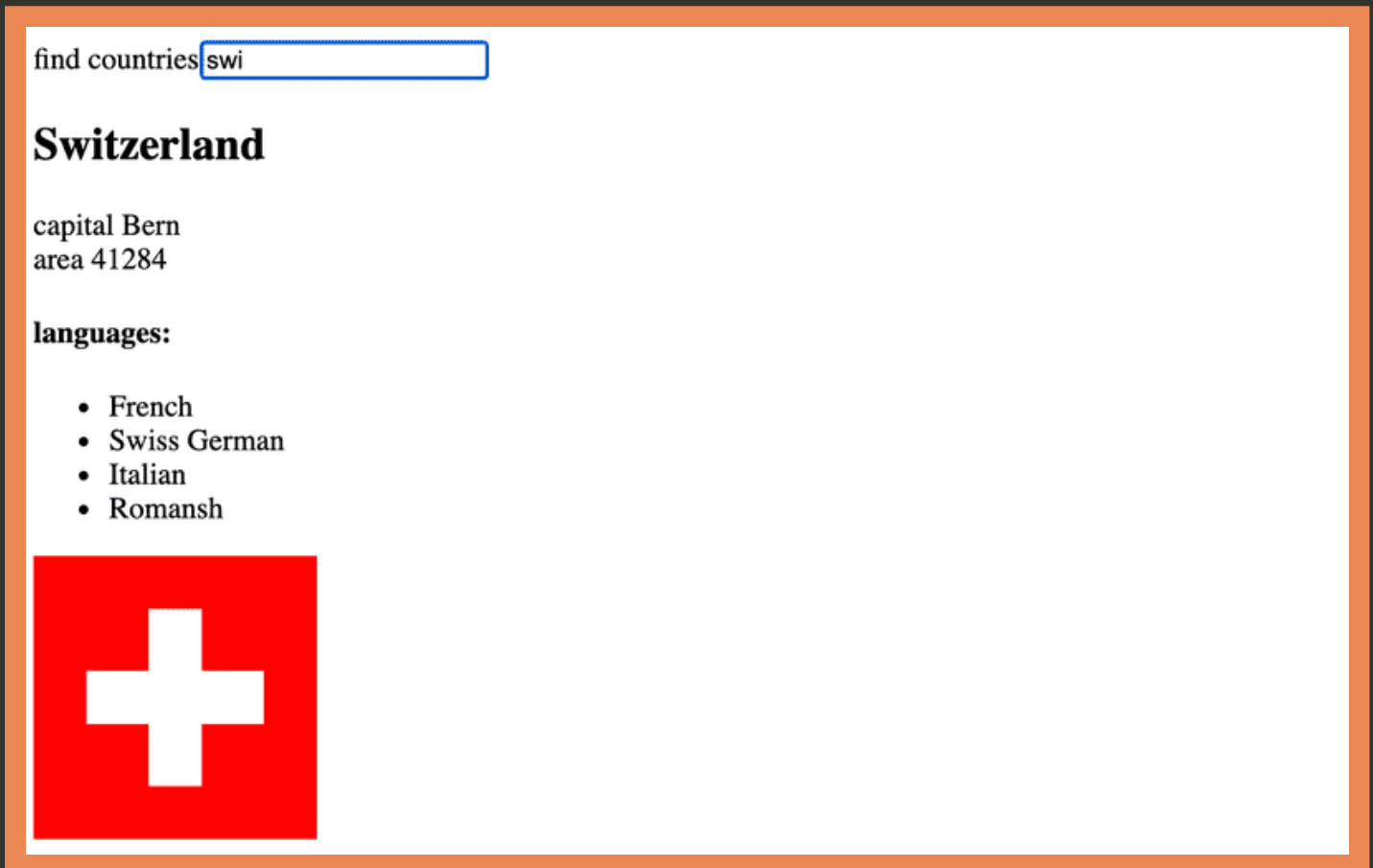




Jos maita on kymmenen tai alle mutta enemmän kuin yksi, näytetään hakuehdon täyttävät maat:



Kun ehdon täyttäviä maita on enää yksi, näytetään maan perustiedot, lippu sekä maassa puhutut kielet:



**Huom1:** Riittää, että sovelluksesi toimii suurimmalle osalle maista. Jotkut maat kuten Sudan voivat tuottaa ongelmia, sillä maan nimi on toisen maan (South Sudan) osa. Näistä corner caseista ei tarvitse välittää.

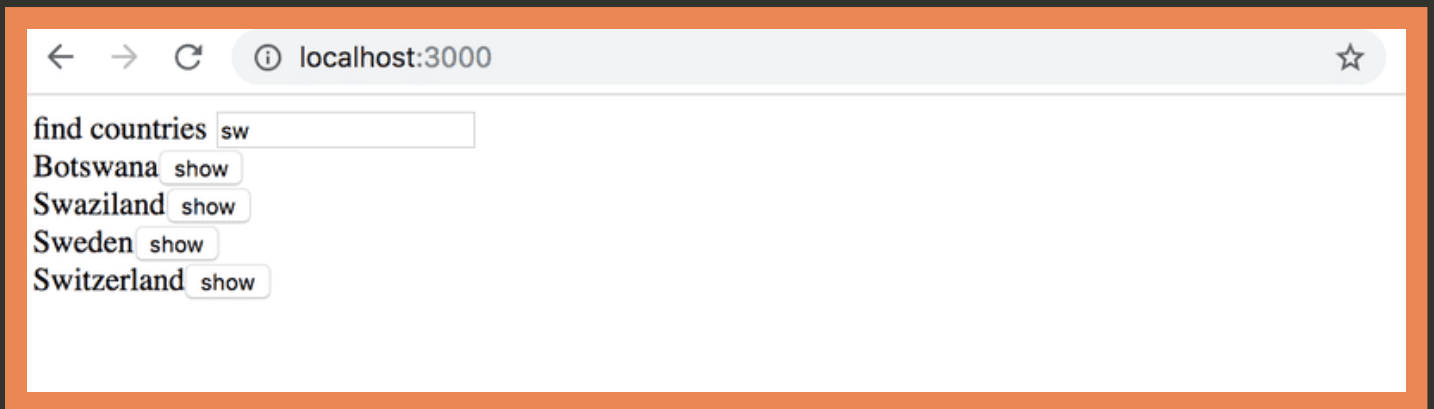
**Huom2:** Saatat törmätä ongelmiin tässä tehtävässä, jos määrittelet komponentteja "väärässä paikassa". Nyt kannattaakin ehdottomasti kerrata edellisen osan luku älä määrittele komponenttia komponentin sisällä.

2.19\*: maiden tiedot, step2

Tässä osassa on vielä paljon tekemistä, joten älä juutu tähän tehtävään!



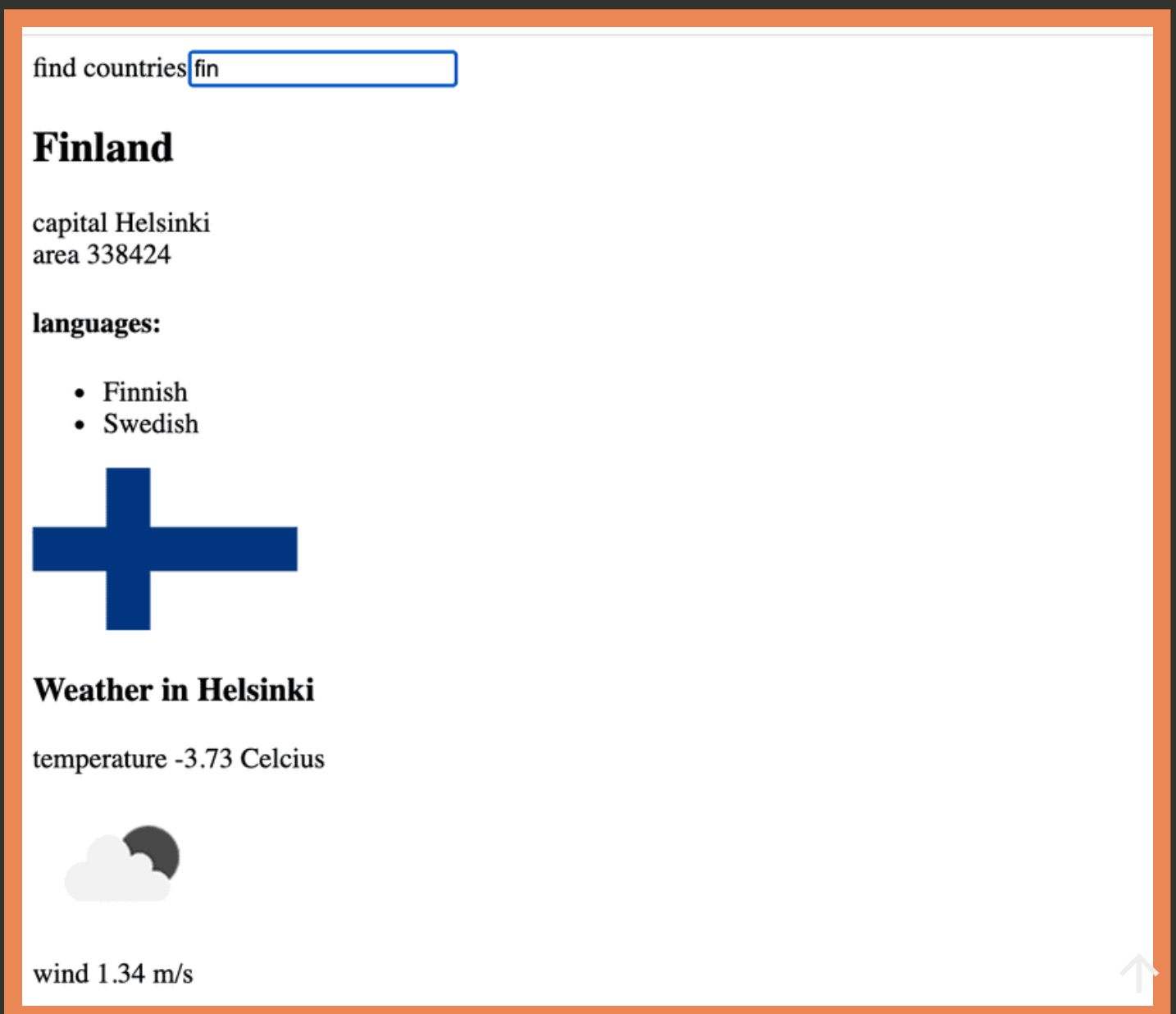
Paranna edellisen tehtävän maasovellusta siten, että kun sivulla näkyy useiden maiden nimiä, tulee maan nimen viereen nappi, jota klikkaamalla pääsee suoraan maan näkymään:



Tässäkin tehtävässä riittää, että ohjelmasi toimii suurella osalla maita ja maat, joiden nimi sisältyy johonkin muuhun maahan (kuten Sudan) voit unohtaa.

## 2.20\*: maiden tiedot, step3

Lisää yksittäisen maan näkymään pääkaupungin säätiedotus. Sää tiedotuksen tarjoavia palveluita on kymmeniä. Itse käytin <https://openweathermap.org/>:ia. Huomaa että api-avaimen luomisen jälkeen saattaa kulua hetki ennen kuin avain alkaa toimia.



Jos käytät Open weather mapia, täällä on ohje sääikonien generointiin.

**Huom:** Tarvitset melkein kaikkia säätietoja tarjoavia palveluja käyttääksesi API-avaimen. Älä talleta avainta versionhallintaan eli älä kirjoita avainta suoraan koodiin. Avaimen arvo kannattaa määritellä ns. ympäristömuuttujana.

Oletetaan että API-avaimen arvo on *54l41n3n4v41m34rv0*. Kun ohjelma käynnistetään seuraavasti

```
VITE_SOME_KEY=54l41n3n4v41m34rv0 && npm run dev // Linux/macOS Bash
($env:VITE_SOME_KEY="54l41n3n4v41m34rv0") -and (npm run dev) // Windows PowerShell
set "VITE_SOME_KEY=54l41n3n4v41m34rv0" && npm run dev // Windows cmd.exe
```

copy

koodista päästään avaimen arvoon käsiksi olion `import.meta.env` kautta:

```
const api_key = import.meta.env.VITE_SOME_KEY
// muuttujassa api_key on nyt käynnistyksessä annettu API-avaimen arvo
```

copy

Tämä oli osan viimeinen tehtävä ja on aika sekä puskea koodi GitHubiin että merkitä tehdyt tehtävät palautussovellukseen.

## Ehdota muutosta materiaalin sisältöön

Osa 2d  
Edellinen osa

Osa 3  
Seuraava osa

Kurssista

Kurssin sisältö

FAQ

Kurssilla mukana

Haaste







**UNIVERSITY OF HELSINKI**



**HOUSTON**

