



## b Forms

Let's continue expanding our application by allowing users to add new notes.

In order to get our page to update when new notes are added it's best to store the notes in the *App* component's state. Let's import the useState function and use it to define a piece of state that gets initialized with the initial notes array passed in the props.

```
import React, { 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
```

The component uses the `useState` function to initialize the piece of state stored in `notes` with the array of notes passed in the props:

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

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

// ...
}
```

If we wanted to start with an empty list of notes we would set the initial value as an empty array, and since the props would not be used, we could omit the `props` parameter from the function definition:

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

  // ...
}
```

Let's stick with the initial value passed in the props for the time being.

Next, let's add an HTML form to the component that will be used for adding new notes.

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

  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 />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

We have added the `addNote` function as an event handler to the form element that will be called when the form is submitted, by clicking the submit button.

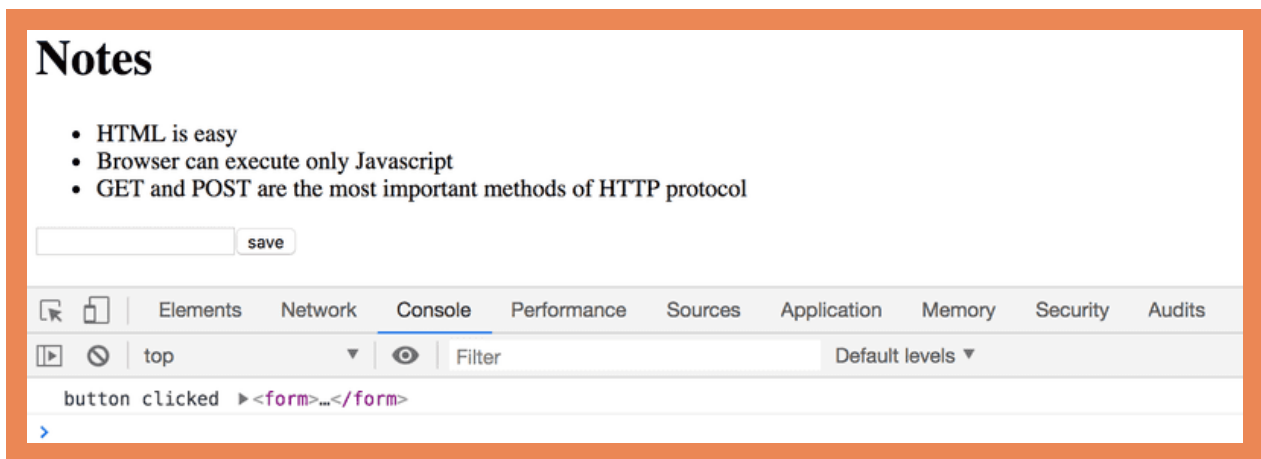
We use the method discussed in [part 1](#) for defining our event handler:

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

The `event` parameter is the [event](#) that triggers the call to the event handler function:

The event handler immediately calls the `event.preventDefault()` method, which prevents the default action of submitting a form. The default action would, among other things, cause the page to reload.

The target of the event stored in `event.target` is logged to the console:



The target in this case is the form that we have defined in our component.

How do we access the data contained in the form's *input* element?

There are many ways to accomplish this; the first method we will take a look at is through the use of so-called [controlled components](#).

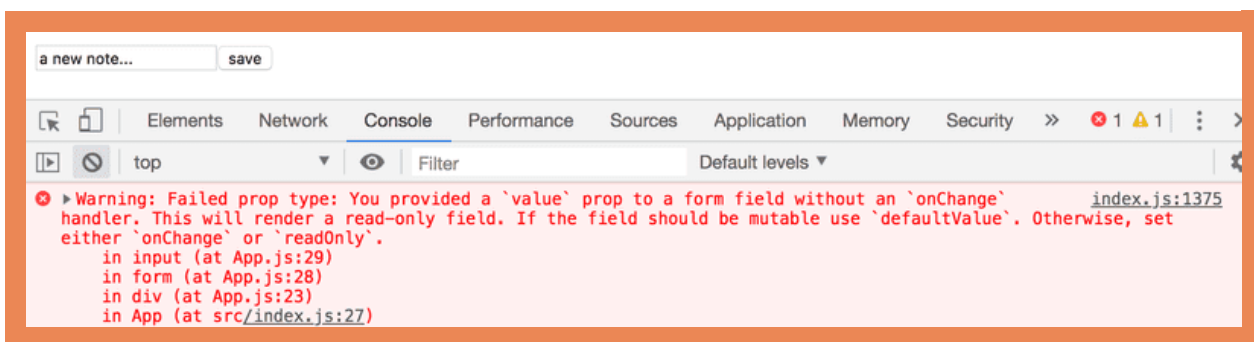
Let's add a new piece of state called `newNote` for storing the user-submitted input and let's set it as the *input* element's *value* attribute:

```
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>
  )
}
```

The placeholder text stored as the initial value of the `newNote` state appears in the *input* element, but the input text can't be edited. The console displays a warning that gives us a clue as to what might be wrong:



Since we assigned a piece of the *App* component's state as the *value* attribute of the input element, the *App* component now controls the behavior of the input element.

In order to enable editing of the input element, we have to register an *event handler* that synchronizes the changes made to the input with the component's state:

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

```
const [newNote, setNewNote] = useState(
  '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>
)
```

We have now registered an event handler to the *onChange* attribute of the form's *input* element:

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

The event handler is called every time *a change occurs in the input element*. The event handler function receives the event object as its `event` parameter:

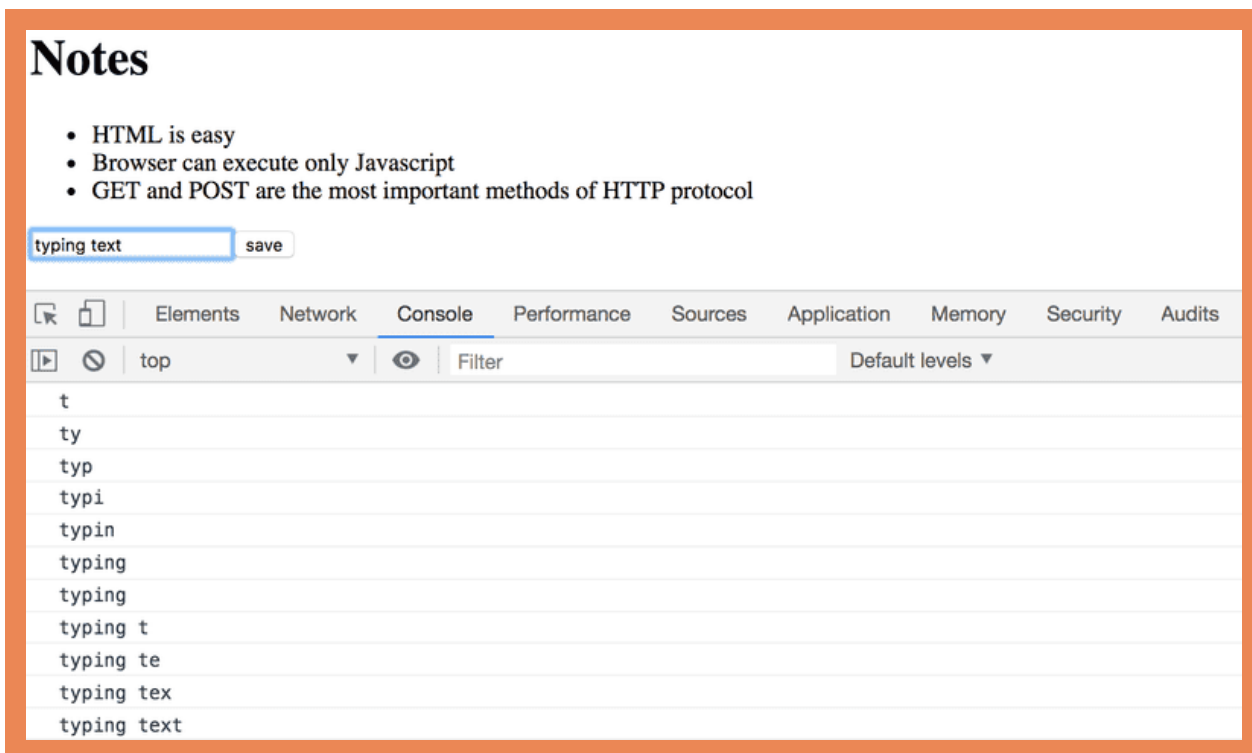
```
const handleNoteChange = (event) => {
  console.log(event.target.value)
  setNewNote(event.target.value)
}
```

The `target` property of the event object now corresponds to the controlled *input* element and

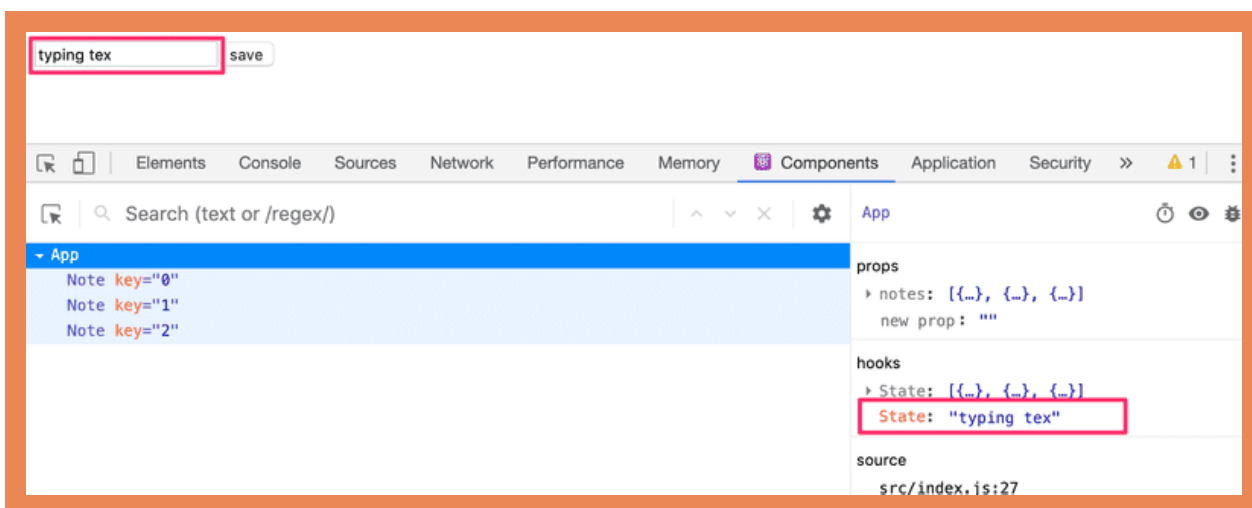
`event.target.value` refers to the input value of that element.

Note that we did not need to call the `event.preventDefault()` method like we did in the `onSubmit` event handler. This is because there is no default action that occurs on an input change, unlike on a form submission.

You can follow along in the console to see how the event handler is called:



You did remember to install [React devtools](#), right? Good. You can directly view how the state changes from the React Devtools tab:



Now the `App` component's `newNote` state reflects the current value of the input, which means that we can complete the `addNote` function for creating new notes:

```
const addNote = (event) => {  
  event.preventDefault()  
}
```

```
const noteObject = {
  content: newNote,
  date: new Date().toISOString(),
  important: Math.random() < 0.5,
  id: notes.length + 1,
}

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

First we create a new object for the note called `noteObject` that will receive its content from the component's `newNote` state. The unique identifier `id` is generated based on the total number of notes. This method works for our application since notes are never deleted. With the help of the `Math.random()` function, our note has a 50% chance of being marked as important.

The new note is added to the list of notes using the `concat` array method, introduced in [part 1](#):

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

The method does not mutate the original `notes` array, but rather creates *a new copy of the array with the new item added to the end*. This is important since we must never mutate state directly in React!

The event handler also resets the value of the controlled input element by calling the `setNewNote` function of the `newNote` state:

```
setNewNote('')
```

You can find the code for our current application in its entirety in the *part2-2* branch of [this GitHub repository](#).

## Filtering Displayed Elements

Let's add some new functionality to our application that allows us to only view the important notes.

Let's add a piece of state to the *App* component that keeps track of which notes should be displayed:

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

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

// ...
}
```

Let's change the component so that it stores a list of all the notes to be displayed in the `notesToShow` variable. The items of the list depend on the state of the component:

```
import React, { 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 === true)

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

The definition of the `notesToShow` variable is rather compact:

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

The definition uses the conditional operator also found in many other programming languages.

The operator functions as follows. If we have:



```
const result = condition ? val1 : val2
```

the `result` variable will be set to the value of `val1` if `condition` is true. If `condition` is false, the `result` variable will be set to the value of `val2` .

If the value of `showAll` is false, the `notesToShow` variable will be assigned to a list that only contains notes that have the `important` property set to true. Filtering is done with the help of the array `filter` method:

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

The comparison operator is in fact redundant, since the value of `note.important` is either *true* or *false* which means that we can simply write:

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

The reason we showed the comparison operator first was to emphasize an important detail: in JavaScript `val1 == val2` does not work as expected in all situations and it's safer to use `val1 === val2` exclusively in comparisons. You can read more about the topic [here](#).

You can test out the filtering functionality by changing the initial value of the `showAll` state.

Next let's add functionality that enables users to toggle the `showAll` state of the application from the user interface.

The relevant changes are shown below:

```
import React, { 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)

  // ...

  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>
)
}

```

The displayed notes (all versus important) are controlled with a button. The event handler for the button is so simple that it has been defined directly in the attribute of the button element. The event handler switches the value of `showAll` from true to false and vice versa:

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

The text of the button depends on the value of the `showAll` state:

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

You can find the code for our current application in its entirety in the *part2-3* branch of [this GitHub repository](#).

## Exercises 2.6.-2.10.

In the first exercise, we will start working on an application that will be further developed in the later exercises. In related sets of exercises it is sufficient to return the final version of your application. You may also make a separate commit after you have finished each part of the exercise set, but doing so is not required.

WARNING create-react-app will automatically turn your project into a git-repository unless you create your application inside of an existing git repository. It's likely that you do not want your project to be a repository, so simply run the `rm -rf .git` command at the root of your application.

## 2.6: The Phonebook Step1

Let's create a simple phonebook. *In this part we will only be adding names to the phonebook.*

Let us start by implementing the addition of a person to phonebook.

You can use the code below as a starting point for the *App* component of your application:

```
import React, { 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
```

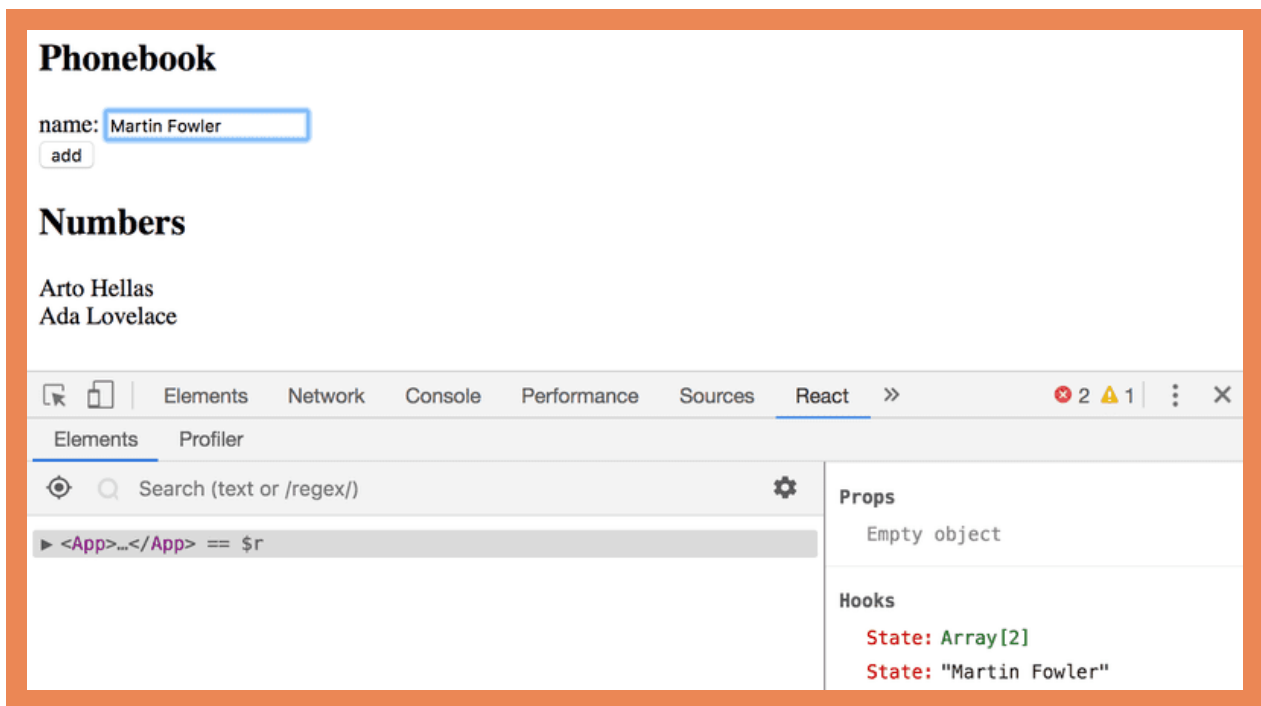
The `newName` state is meant for controlling the form input element.

Sometimes it can be useful to render state and other variables as text for debugging purposes. You can temporarily add the following element to the rendered component:

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

It's also important to put what we learned in the debugging React applications chapter of part one into good use. The React developer tools extension especially, is incredibly useful for tracking changes that occur in the application's state.

After finishing this exercise your application should look something like this:



Note the use of the React developer tools extension in the picture above!

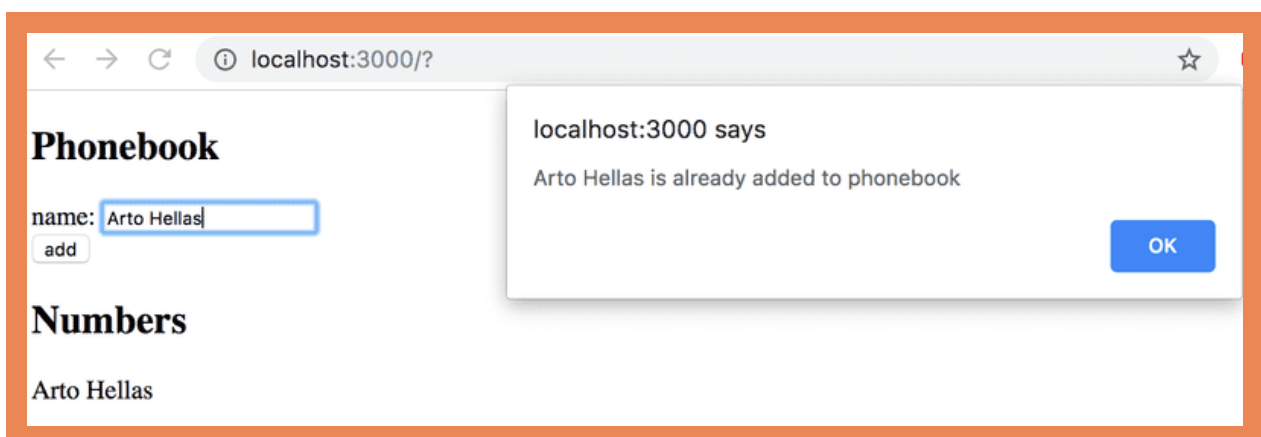
NB:

- you can use the person's name as value of the *key* property
- remember to prevent the default action of submitting HTML forms!

## 2.7: The Phonebook Step2

Prevent the user from being able to add names that already exist in the phonebook. JavaScript arrays have numerous suitable methods for accomplishing this task.

Issue a warning with the alert command when such an action is attempted:



Hint: when you are forming strings that contain values from variables, it is recommended to use a template string:

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

If the `newName` variable holds the value *Arto Hellas*, the template string expression returns the string

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

The same could be done in a more Java-like fashion by using the plus operator:

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

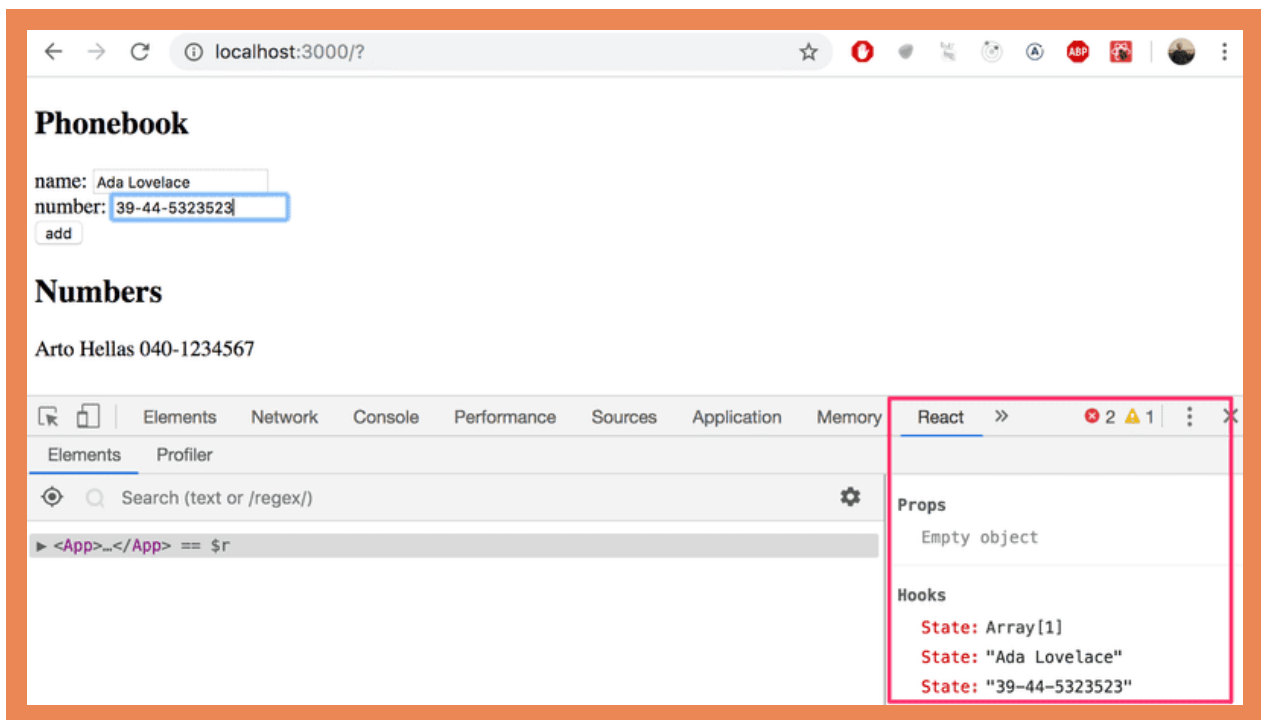
Using template strings is the more idiomatic option and the sign of a true JavaScript professional.

## 2.8: The Phonebook Step3

Expand your application by allowing users to add phone numbers to the phone book. You will need to add a second *input* element to the form (along with its own event handler):

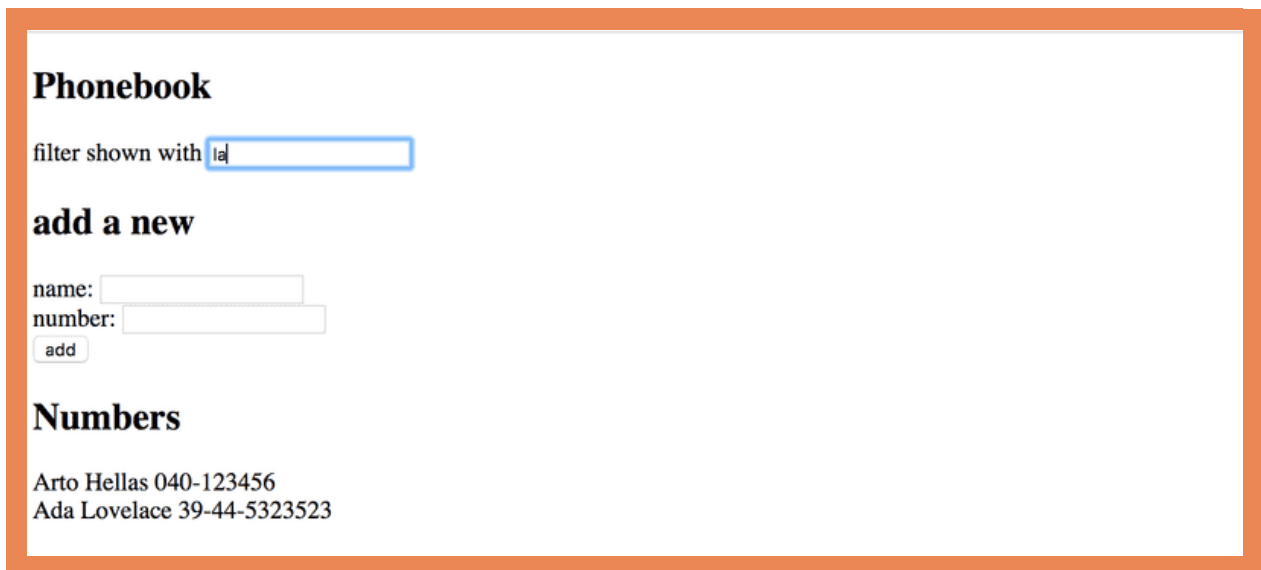
```
<form>
  <div>name: <input /></div>
  <div>number: <input /></div>
  <div><button type="submit">add</button></div>
</form>
```

At this point the application could look something like this. The image also displays the application's state with the help of React developer tools:



## 2.9\*: The Phonebook Step4

Implement a search field that can be used to filter the list of people by name:



You can implement the search field as an *input* element that is placed outside the HTML form. The filtering logic shown in the image is *case insensitive*, meaning that the search term *arto* also returns results that contain Arto with an uppercase A.

NB: When you are working on new functionality, it's often useful to "hardcode" some dummy data into your application, e.g.

```
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' }
  ])

  // ...
}

```

This saves you from having to manually input data into your application for testing out your new functionality.

## 2.10: The Phonebook Step5

If you have implemented your application in a single component, refactor it by extracting suitable parts into new components. Maintain the application's state and all event handlers in the *App* root component.

It is sufficient to extract *three* components from the application. Good candidates for separate components are, for example, the search filter, the form for adding new people into the phonebook, a component that renders all people from the phonebook, and a component that renders a single person's details.

The application's root component could look similar to this after the refactoring. The refactored root component below only renders titles and lets the extracted components take care of the rest.

```

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

  return (
    <div>
      <h2>Phonebook</h2>
      <Filter ... />
      <h3>Add a new</h3>
      <h3>Numbers</h3>
      <Persons ... />
    </div>
  )
}

```

[Propose changes to material](#)

Part 2a  
Part 2b  
Part 2c  
Part 2d  
Part 2e  
Part 2f  
Part 2g  
Part 2h  
Part 2i  
Part 2j  
Part 2k  
Part 2l  
Part 2m  
Part 2n  
Part 2o  
Part 2p  
Part 2q  
Part 2r  
Part 2s  
Part 2t  
Part 2u  
Part 2v  
Part 2w  
Part 2x  
Part 2y  
Part 2z  
Part 2aa  
Part 2ab  
Part 2ac  
Part 2ad  
Part 2ae  
Part 2af  
Part 2ag  
Part 2ah  
Part 2ai  
Part 2aj  
Part 2ak  
Part 2al  
Part 2am  
Part 2an  
Part 2ao  
Part 2ap  
Part 2aq  
Part 2ar  
Part 2as  
Part 2at  
Part 2au  
Part 2av  
Part 2aw  
Part 2ax  
Part 2ay  
Part 2az  
Part 2ba  
Part 2bb  
Part 2bc  
Part 2bd  
Part 2be  
Part 2bf  
Part 2bg  
Part 2bh  
Part 2bi  
Part 2bj  
Part 2bk  
Part 2bl  
Part 2bm  
Part 2bn  
Part 2bo  
Part 2bp  
Part 2bq  
Part 2br  
Part 2bs  
Part 2bt  
Part 2bu  
Part 2bv  
Part 2bw  
Part 2bx  
Part 2by  
Part 2bz  
Part 2ca  
Part 2cb  
Part 2cc  
Part 2cd  
Part 2ce  
Part 2cf  
Part 2cg  
Part 2ch  
Part 2ci  
Part 2cj  
Part 2ck  
Part 2cl  
Part 2cm  
Part 2cn  
Part 2co  
Part 2cp  
Part 2cq  
Part 2cr  
Part 2cs  
Part 2ct  
Part 2cu  
Part 2cv  
Part 2cw  
Part 2cx  
Part 2cy  
Part 2cz  
Part 2da  
Part 2db  
Part 2dc  
Part 2dd  
Part 2de  
Part 2df  
Part 2dg  
Part 2dh  
Part 2di  
Part 2dj  
Part 2dk  
Part 2dl  
Part 2dm  
Part 2dn  
Part 2do  
Part 2dp  
Part 2dq  
Part 2dr  
Part 2ds  
Part 2dt  
Part 2du  
Part 2dv  
Part 2dw  
Part 2dx  
Part 2dy  
Part 2dz  
Part 2ea  
Part 2eb  
Part 2ec  
Part 2ed  
Part 2ee  
Part 2ef  
Part 2eg  
Part 2eh  
Part 2ei  
Part 2ej  
Part 2ek  
Part 2el  
Part 2em  
Part 2en  
Part 2eo  
Part 2ep  
Part 2eq  
Part 2er  
Part 2es  
Part 2et  
Part 2eu  
Part 2ev  
Part 2ew  
Part 2ex  
Part 2ey  
Part 2ez  
Part 2fa  
Part 2fb  
Part 2fc  
Part 2fd  
Part 2fe  
Part 2ff  
Part 2fg  
Part 2fh  
Part 2fi  
Part 2fj  
Part 2fk  
Part 2fl  
Part 2fm  
Part 2fn  
Part 2fo  
Part 2fp  
Part 2fq  
Part 2fr  
Part 2fs  
Part 2ft  
Part 2fu  
Part 2fv  
Part 2fw  
Part 2fx  
Part 2fy  
Part 2fz  
Part 2ga  
Part 2gb  
Part 2gc  
Part 2gd  
Part 2ge  
Part 2gf  
Part 2gg  
Part 2gh  
Part 2gi  
Part 2gj  
Part 2gk  
Part 2gl  
Part 2gm  
Part 2gn  
Part 2go  
Part 2gp  
Part 2gq  
Part 2gr  
Part 2gs  
Part 2gt  
Part 2gu  
Part 2gv  
Part 2gw  
Part 2gx  
Part 2gy  
Part 2gz  
Part 2ha  
Part 2hb  
Part 2hc  
Part 2hd  
Part 2he  
Part 2hf  
Part 2hg  
Part 2hh  
Part 2hi  
Part 2hj  
Part 2hk  
Part 2hl  
Part 2hm  
Part 2hn  
Part 2ho  
Part 2hp  
Part 2hq  
Part 2hr  
Part 2hs  
Part 2ht  
Part 2hu  
Part 2hv  
Part 2hw  
Part 2hx  
Part 2hy  
Part 2hz  
Part 2ia  
Part 2ib  
Part 2ic  
Part 2id  
Part 2ie  
Part 2if  
Part 2ig  
Part 2ih  
Part 2ii  
Part 2ij  
Part 2ik  
Part 2il  
Part 2im  
Part 2in  
Part 2io  
Part 2ip  
Part 2iq  
Part 2ir  
Part 2is  
Part 2it  
Part 2iu  
Part 2iv  
Part 2iw  
Part 2ix  
Part 2iy  
Part 2iz  
Part 2ja  
Part 2jb  
Part 2jc  
Part 2jd  
Part 2je  
Part 2jf  
Part 2jg  
Part 2jh  
Part 2ji  
Part 2jj  
Part 2jk  
Part 2jl  
Part 2jm  
Part 2jn  
Part 2jo  
Part 2jp  
Part 2jq  
Part 2jr  
Part 2js  
Part 2jt  
Part 2ju  
Part 2jv  
Part 2jw  
Part 2jx  
Part 2jy  
Part 2jz  
Part 2ka  
Part 2kb  
Part 2kc  
Part 2kd  
Part 2ke  
Part 2kf  
Part 2kg  
Part 2kh  
Part 2ki  
Part 2kj  
Part 2kk  
Part 2kl  
Part 2km  
Part 2kn  
Part 2ko  
Part 2kp  
Part 2kq  
Part 2kr  
Part 2ks  
Part 2kt  
Part 2ku  
Part 2kv  
Part 2kw  
Part 2kx  
Part 2ky  
Part 2kz  
Part 2la  
Part 2lb  
Part 2lc  
Part 2ld  
Part 2le  
Part 2lf  
Part 2lg  
Part 2lh  
Part 2li  
Part 2lj  
Part 2lk  
Part 2ll  
Part 2lm  
Part 2ln  
Part 2lo  
Part 2lp  
Part 2lq  
Part 2lr  
Part 2ls  
Part 2lt  
Part 2lu  
Part 2lv  
Part 2lw  
Part 2lx  
Part 2ly  
Part 2lz  
Part 2ma  
Part 2mb  
Part 2mc  
Part 2md  
Part 2me  
Part 2mf  
Part 2mg  
Part 2mh  
Part 2mi  
Part 2mj  
Part 2mk  
Part 2ml  
Part 2mm  
Part 2mn  
Part 2mo  
Part 2mp  
Part 2mq  
Part 2mr  
Part 2ms  
Part 2mt  
Part 2mu  
Part 2mv  
Part 2mw  
Part 2mx  
Part 2my  
Part 2mz  
Part 2na  
Part 2nb  
Part 2nc  
Part 2nd  
Part 2ne  
Part 2nf  
Part 2ng  
Part 2nh  
Part 2ni  
Part 2nj  
Part 2nk  
Part 2nl  
Part 2nm  
Part 2nn  
Part 2no  
Part 2np  
Part 2nq  
Part 2nr  
Part 2ns  
Part 2nt  
Part 2nu  
Part 2nv  
Part 2nw  
Part 2nx  
Part 2ny  
Part 2nz  
Part 2oa  
Part 2ob  
Part 2oc  
Part 2od  
Part 2oe  
Part 2of  
Part 2og  
Part 2oh  
Part 2oi  
Part 2oj  
Part 2ok  
Part 2ol  
Part 2om  
Part 2on  
Part 2oo  
Part 2op  
Part 2oq  
Part 2or  
Part 2os  
Part 2ot  
Part 2ou  
Part 2ov  
Part 2ow  
Part 2ox  
Part 2oy  
Part 2oz  
Part 2pa  
Part 2pb  
Part 2pc  
Part 2pd  
Part 2pe  
Part 2pf  
Part 2pg  
Part 2ph  
Part 2pi  
Part 2pj  
Part 2pk  
Part 2pl  
Part 2pm  
Part 2pn  
Part 2po  
Part 2pp  
Part 2pq  
Part 2pr  
Part 2ps  
Part 2pt  
Part 2pu  
Part 2pv  
Part 2pw  
Part 2px  
Part 2py  
Part 2pz  
Part 2qa  
Part 2qb  
Part 2qc  
Part 2qd  
Part 2qe  
Part 2qf  
Part 2qg  
Part 2qh  
Part 2qi  
Part 2qj  
Part 2qk  
Part 2ql  
Part 2qm  
Part 2qn  
Part 2qo  
Part 2qp  
Part 2qq  
Part 2qr  
Part 2qs  
Part 2qt  
Part 2qu  
Part 2qv  
Part 2qw  
Part 2qx  
Part 2qy  
Part 2qz  
Part 2ra  
Part 2rb  
Part 2rc  
Part 2rd  
Part 2re  
Part 2rf  
Part 2rg  
Part 2rh  
Part 2ri  
Part 2rj  
Part 2rk  
Part 2rl  
Part 2rm  
Part 2rn  
Part 2ro  
Part 2rp  
Part 2rq  
Part 2rr  
Part 2rs  
Part 2rt  
Part 2ru  
Part 2rv  
Part 2rw  
Part 2rx  
Part 2ry  
Part 2rz  
Part 2sa  
Part 2sb  
Part 2sc  
Part 2sd  
Part 2se  
Part 2sf  
Part 2sg  
Part 2sh  
Part 2si  
Part 2sj  
Part 2sk  
Part 2sl  
Part 2sm  
Part 2sn  
Part 2so  
Part 2sp  
Part 2sq  
Part 2sr  
Part 2ss  
Part 2st  
Part 2su  
Part 2sv  
Part 2sw  
Part 2sx  
Part 2sy  
Part 2sz  
Part 2ta  
Part 2tb  
Part 2tc  
Part 2td  
Part 2te  
Part 2tf  
Part 2tg  
Part 2th  
Part 2ti  
Part 2tj  
Part 2tk  
Part 2tl  
Part 2tm  
Part 2tn  
Part 2to  
Part 2tp  
Part 2tq  
Part 2tr  
Part 2ts  
Part 2tt  
Part 2tu  
Part 2tv  
Part 2tw  
Part 2tx  
Part 2ty  
Part 2tz  
Part 2ua  
Part 2ub  
Part 2uc  
Part 2ud  
Part 2ue  
Part 2uf  
Part 2ug  
Part 2uh  
Part 2ui  
Part 2uj  
Part 2uk  
Part 2ul  
Part 2um  
Part 2un  
Part 2uo  
Part 2up  
Part 2uq  
Part 2ur  
Part 2us  
Part 2ut  
Part 2uu  
Part 2uv  
Part 2uw  
Part 2ux  
Part 2uy  
Part 2uz  
Part 2va  
Part 2vb  
Part 2vc  
Part 2vd  
Part 2ve  
Part 2vf  
Part 2vg  
Part 2vh  
Part 2vi  
Part 2vj  
Part 2vk  
Part 2vl  
Part 2vm  
Part 2vn  
Part 2vo  
Part 2vp  
Part 2vq  
Part 2vr  
Part 2vs  
Part 2vt  
Part 2vu  
Part 2vv  
Part 2vw  
Part 2vx  
Part 2vy  
Part 2vz  
Part 2wa  
Part 2wb  
Part 2wc  
Part 2wd  
Part 2we  
Part 2wf  
Part 2wg  
Part 2wh  
Part 2wi  
Part 2wj  
Part 2wk  
Part 2wl  
Part 2wm  
Part 2wn  
Part 2wo  
Part 2wp  
Part 2wq  
Part 2wr  
Part 2ws  
Part 2wt  
Part 2wu  
Part 2wv  
Part 2ww  
Part 2wx  
Part 2wy  
Part 2wz  
Part 2xa  
Part 2xb  
Part 2xc  
Part 2xd  
Part 2xe  
Part 2xf  
Part 2xg  
Part 2xh  
Part 2xi  
Part 2xj  
Part 2xk  
Part 2xl  
Part 2xm  
Part 2xn  
Part 2xo  
Part 2xp  
Part 2xq  
Part 2xr  
Part 2xs  
Part 2xt  
Part 2xu  
Part 2xv  
Part 2xw  
Part 2xx  
Part 2xy  
Part 2xz  
Part 2ya  
Part 2yb  
Part 2yc  
Part 2yd  
Part 2ye  
Part 2yf  
Part 2yg  
Part 2yh  
Part 2yi  
Part 2yj  
Part 2yk  
Part 2yl  
Part 2ym  
Part 2yn  
Part 2yo  
Part 2yp  
Part 2yq  
Part 2yr  
Part 2ys  
Part 2yt  
Part 2yu  
Part 2yv  
Part 2yw  
Part 2yx  
Part 2yy  
Part 2yz  
Part 2za  
Part 2zb  
Part 2zc  
Part 2zd  
Part 2ze  
Part 2zf  
Part 2zg  
Part 2zh  
Part 2zi  
Part 2zj  
Part 2zk  
Part 2zl  
Part 2zm  
Part 2zn  
Part 2zo  
Part 2zp  
Part 2zq  
Part 2zr  
Part 2zs  
Part 2zt  
Part 2zu  
Part 2zv  
Part 2zw  
Part 2zx  
Part 2zy  
Part 2zz

Part 2c  
Next part

NB: You might run into problems in this exercise if you define your components "in the wrong

place". Now would be a good time to rehearse the chapter do not define a component in another component from last part.



**HOUSTON**