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.

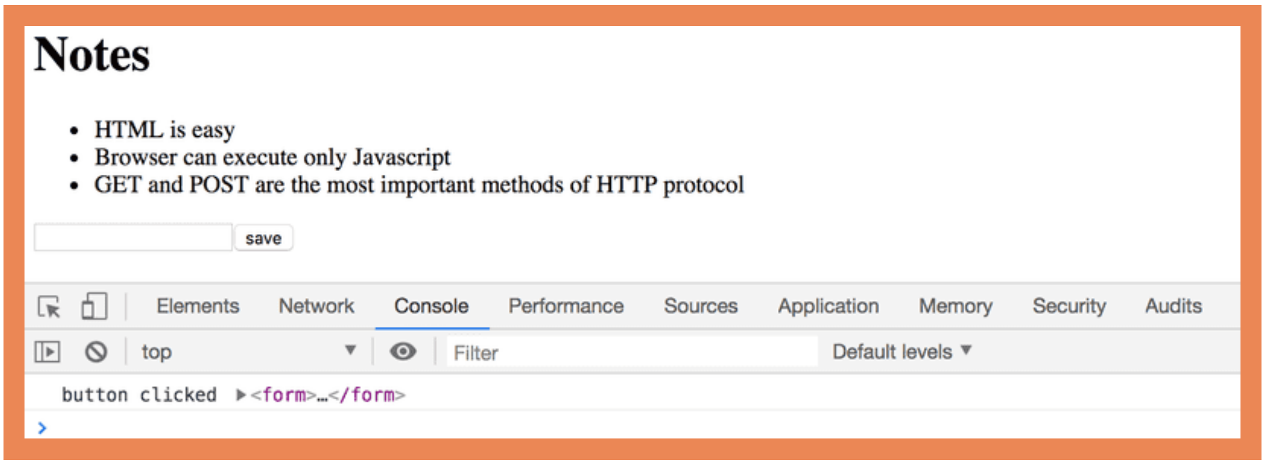
We use the method discussed in part 1 for defining our event handler:

```
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:



Notes

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

```
button clicked  ▶<form>…</form>
```

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.

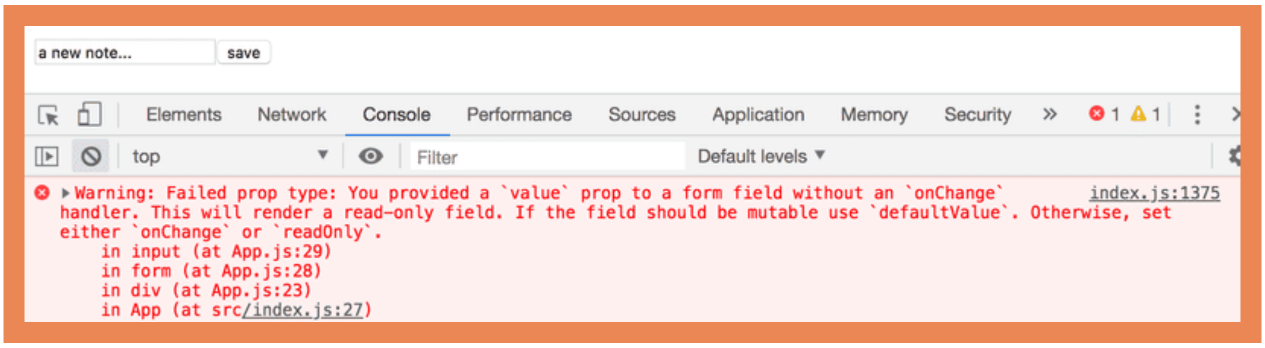
Let's add a new piece of state called `newNote` for storing the user-submitted input and let's set it as the *input* element's *value* attribute:

```
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:



```
▸Warning: Failed prop type: You provided a `value` prop to a form field without an `onChange`    index.js:1375
handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set
either `onChange` or `readOnly`.
    in input (at App.js:29)
    in form (at App.js:28)
    in div (at App.js:23)
    in App (at src/index.js:27)
```

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}
/>
```

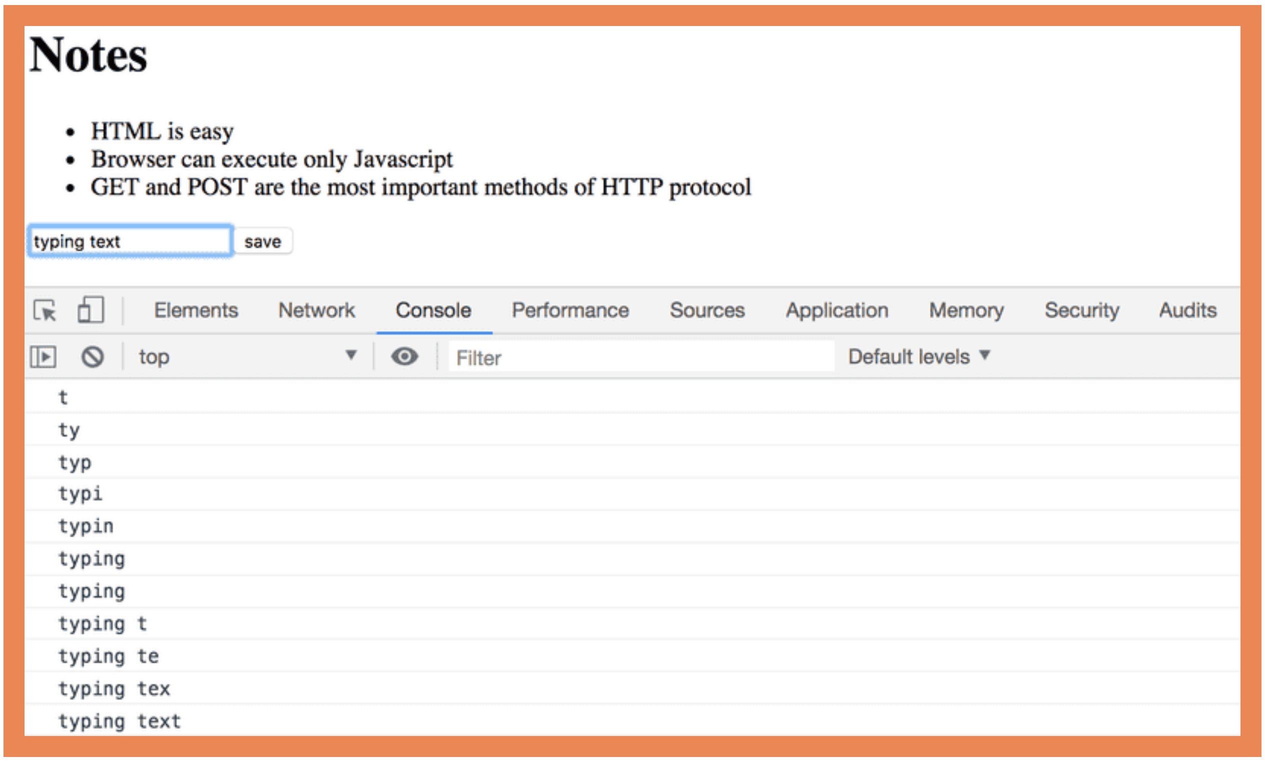
The event handler is called every time *a change occurs in the input element*. The event handler function receives the event object as its `event` parameter:

```
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.

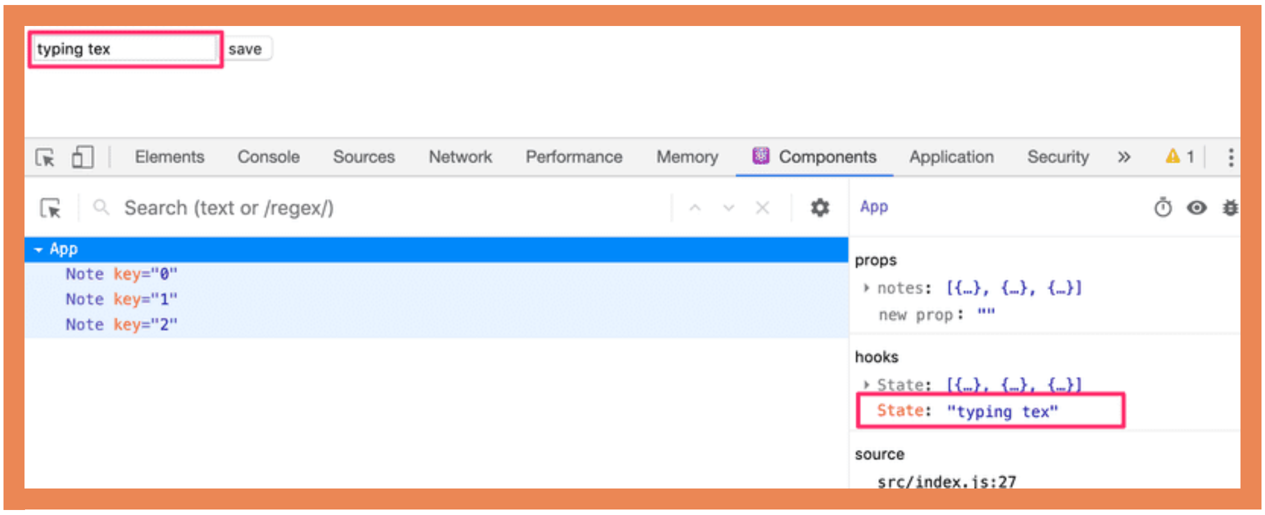
Note that we did not need to call the `event.preventDefault()` method like we did in the *onSubmit* event handler. This is because there is no default action that occurs on an input change, unlike on a form submission.

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



```
t
ty
typ
typi
typin
typing
typing
typing t
typing te
typing tex
typing text
```

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



```
App
  Note key="0"
  Note key="1"
  Note key="2"

props
  notes: [{…}, {…}, {…}]
  new prop: ""
hooks
  State: [{…}, {…}, {…}]
  State: "typing tex"
source
  src/index.js:27
```

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
```

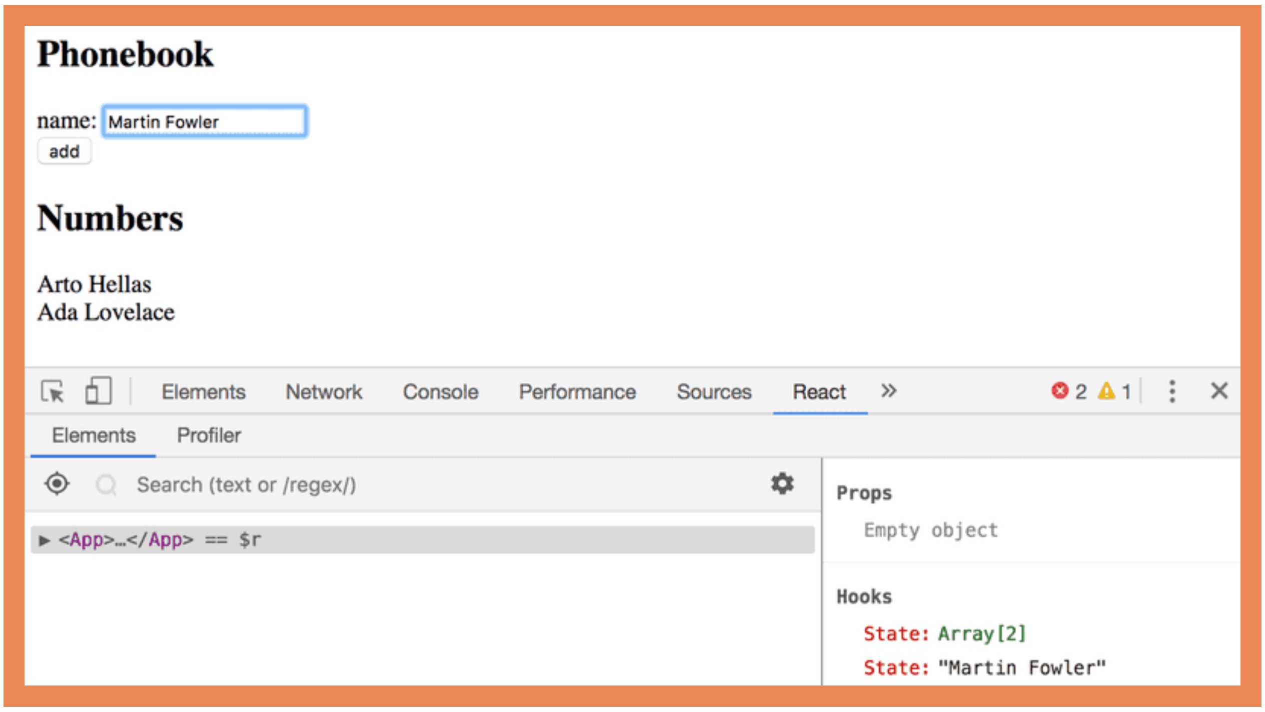
The `newName` state is meant for controlling the form input element.

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:



Phonebook

name: Martin Fowler

add

Numbers

Arto Hellas
Ada Lovelace

```
▸ <App>…</App> == $r

Props
  Empty object

Hooks
  State: Array[2]
  State: "Martin Fowler"
```

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

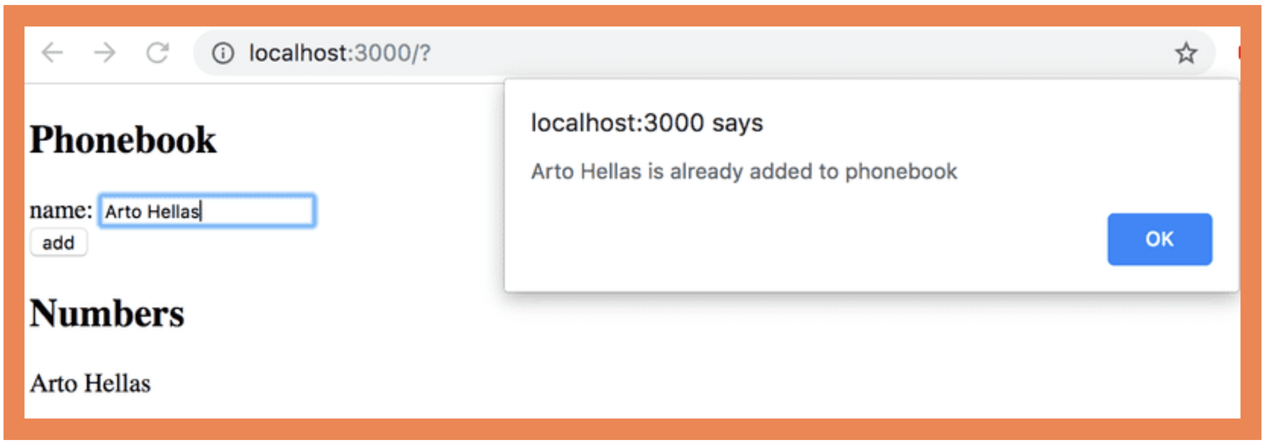
NB:

- 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:



localhost:3000/?

Phonebook

name: Arto Hellas

add

Numbers

Arto Hellas

localhost:3000 says

Arto Hellas is already added to phonebook

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'
```

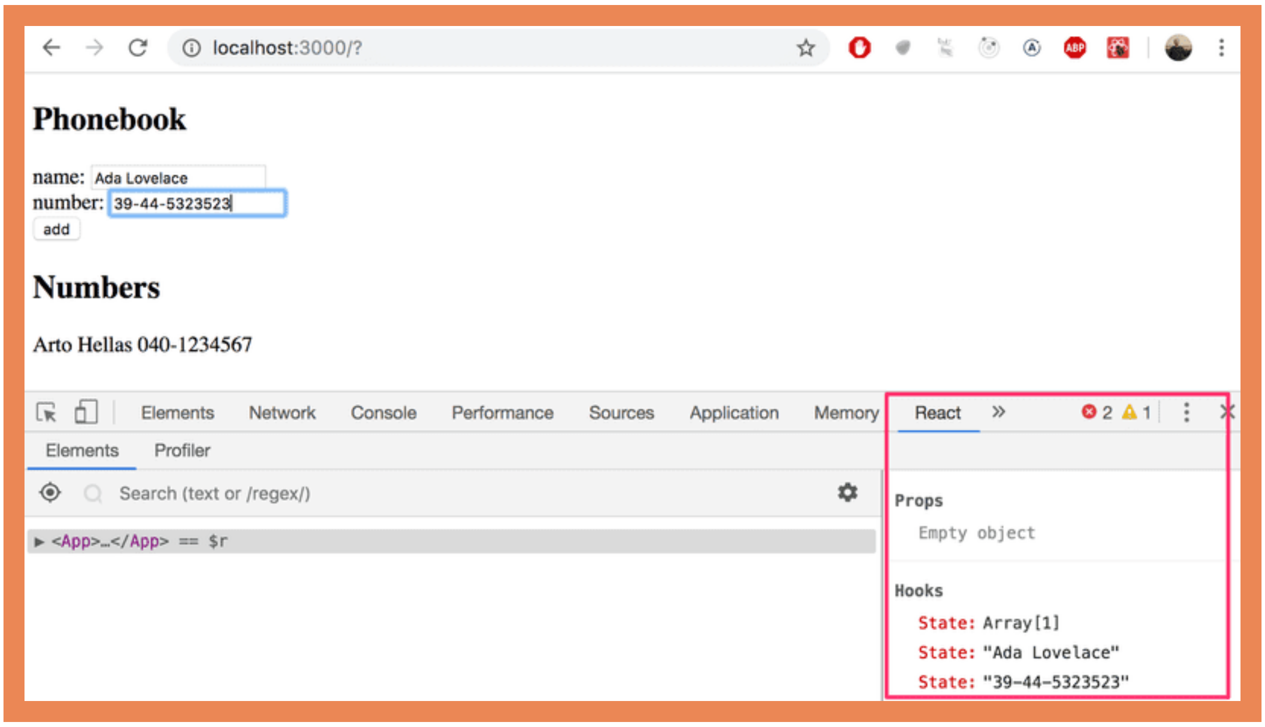
Using template strings is the more idiomatic option and the sign of a true JavaScript professional.

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:



Phonebook

name: Ada Lovelace
number: 39-44-5323523
add

Numbers

Arto Hellas 040-1234567

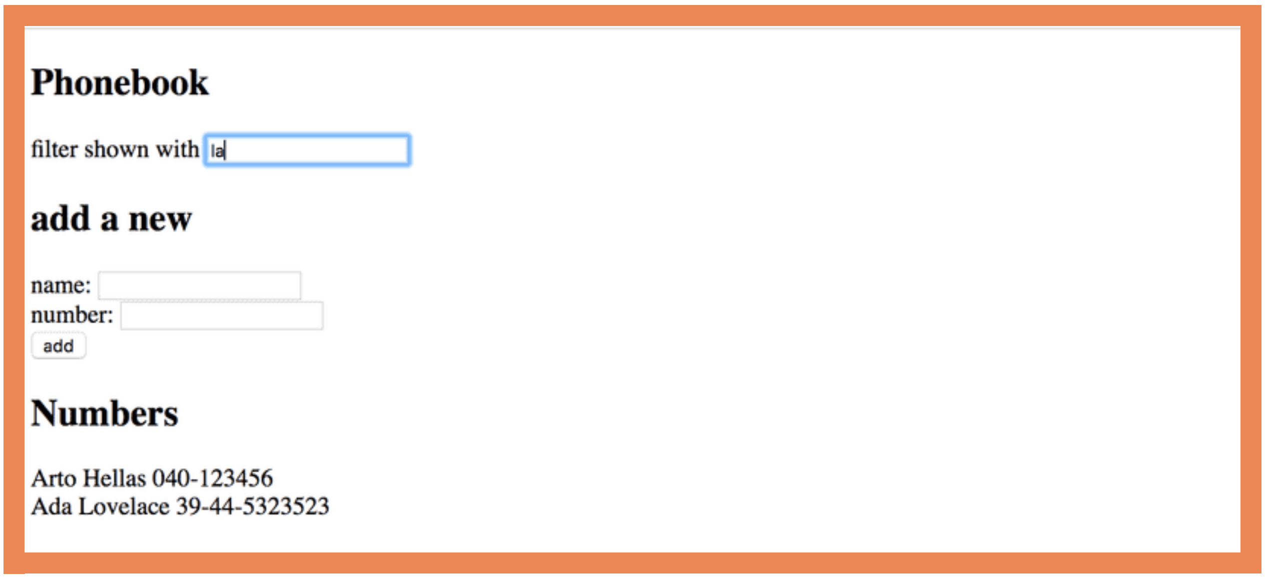
```
▶ <App>…</App> == $r
```

```
Props
  Empty object

Hooks
  State: Array[1]
  State: "Ada Lovelace"
  State: "39-44-5323523"
```

2.9*: The Phonebook Step4

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



Phonebook

filter shown with la

add a new

name:
number:
add

Numbers

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

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>

      <PersonForm 
        ...
      />

      <h3>Numbers</h3>

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

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

About course

Course contents

FAQ

Partners

Challenge

Propose changes to material

Part 2a
Previous part

Part 2c
Next part

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

HOUSTON