04-01pt 2 FORMS

I form html sono inconsistenti → sono diversi in base all'elemento di input e il browser.
React semplifica il tutto cercando di unifromare il comportamento attraverso JSX.

Per **textArea e select** (menù a tendina):

- Attributo **value** contiene sempre il valore corrente del campo
- **Default value** contiene il valore di default che è stato dato al campo quando è stato creato

Per gestire gli eventi che cambiano, react fornisce l'elemento **onChange** che gestisce tutto ciò che cambia quando un dato cambia. Passando una funzione ad un onChange si possono gestire gli eventi che cambiano

Le callback sono event handler e sono chiamate con un parametro event object avente un insieme di proprietà:

- event.target → sorgente di evento
- alcuni eventi oltre al targer possono avere alcune proprietà aggiuntive.

Eventi sintetici:

Category	Events
Clipboard	onCopy onCut onPaste
Composition	onCompositionEnd onCompositionStart onCompositionUpdate
Keyboard	onKeyDown **onKeyPress** onKeyUp
Focus	**onFocus onBlur**
Form	**onChange** onInput **onInvalid** onReset **onSubmit**
Generic	onError **onLoad**
Mouse	**onClick** onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp
Pointer	onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut
Selection	onSelect
Touch	onTouchCancel onTouchEnd onTouchMove onTouchStart
UI	onScroll
Wheel	onWheel
Media	onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend onTimeUpdate onVolumeChange onWaiting
Image	onLoad onError
Animation	onAnimationStart onAnimationEnd onAnimationIteration
Transition	onTransitionEnd

Event handler: sono funzioni, tipicamente definite come arrow function

```
const handler = () => { ... }
```

```
handler = function() { ... }
```

- Quando si vuole passare un event handler, si passa il riferimento alla funzione *-senza parentesi tonde, altrimenti la stai eseguendo-*.

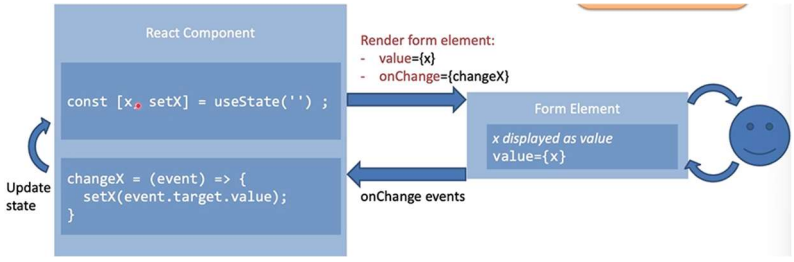
```
return <div handler={handler} />
```

- Se si vogliono passare dei parametri, si può usare un arrow function

```
return <button onClick=
  {()=>props.handler()} />
```

```
return <button onClick=
  {()=>props.handler(a, b)} />
```

Gli **elementi di un form hanno sempre uno stato**, ogni elemento di uno stato ha bisogno di avere il proprio **onChange**.



```
function MyForm (props) {
  const [name, setName] = useState();
  return <form onSubmit={handleSubmit}>
    <label> Name:
      <input type="text" value={name}
        onChange={handleChange} />
    </label>
    <input type="submit" value="Submit" />
  </form> ;
}
```

```
handleSubmit = (event) => {
  console.log('Name submitted: ' +
    name);
  event.preventDefault();
}

handleChange = (event) => {
  setName(event.target.value) ;
};
```

onSubmit → cosa capita quando l'intero form viene mandato.

Nota: un tipo/elemento di type submit scatena in automatico l'evento di onClick e l'evento di submit del form → sul form possiamo usare onSubmit che reagisce al submit (senza che sia necessario reagire all'onClick)

Nota: bisogna sempre chiamare event.preventDefault() per evitare la sottomissione e il ricaricamento della pagina da 0.

Nei form, i dati devono essere validati, usando i validatori interni o usando validator.js.

- Alternatively, use a **library** such as Formik
  - Keep things organized without hiding them too much
  - Form state is inherently ephemeral and local: does not use state management solutions (e.g., Redux/Flux) which would unnecessary complicate things
  - Includes validation, keeping track of the visited fields, and handling form submission
  - https://jaredpalmer.com/formik

→ libreria aggiuntiva per form complessi

Gestire array nello stato → ultime slide

```
// Update item: use map(); if items are objects, always return a new object if modified
...
const [list, setList] = useState([{id:3, val:'Foo'},{id:5, val:'Bar'}]);
...
// i is the id of the item to update
setList(oldList => {
  const list = oldList.map((item) => {
    if (item.id === i) {
      // item.val='NewVal'; return item;  // WRONG: the old object must not be reused
      return {id:item.id, val:'NewVal'};  // return a new object: do not simply change conter
    } else {
      return item;
    }
  });
  return list ;
});
```

→ aggiornare elmeenmti in un array di oggetti:

crea un nuovo stato quado va a modificarsi lo stato e ogni nuovo oggetto all'interno dell'array dello stato. Ci serve fare una copia profonda dello stato (array che contiene lo stato + oggetti contenuti in esso)

```
// Append at the end: use .concat()
// NO .push(): it returns the number of
elements, not the array
...

const [list, setList] = useState(['a',
'b', 'c']);
...

setList(oldList =>
    return oldList.concat(newItem);
)
```

```
// Insert value(s) at the beginning
// use spread operator

...

const [list, setList] = useState(['a',
'b', 'c']);

...

setList(oldList =>
    return [newItem, ...oldList];
)
```

→ aggiungere ad uno stato

```
// Remove item: use filter()

...
const [list, setList] = useState([11, 42,
32]);
...


// i is the index of the element to remove
setList(oldList=> {
   return oldList.filter(
      (item, j) => i !== j );
});
```

```
// Remove first item(s): use destructuring

...
const [list, setList] = useState([11, 42,
32]);
...

setList(oldList => {
   const [first, ...list] = oldList;
   return list ;
});
```

→ rimuovere elementi in uno stato contenente un array