

APPLICAZIONI WEB 1

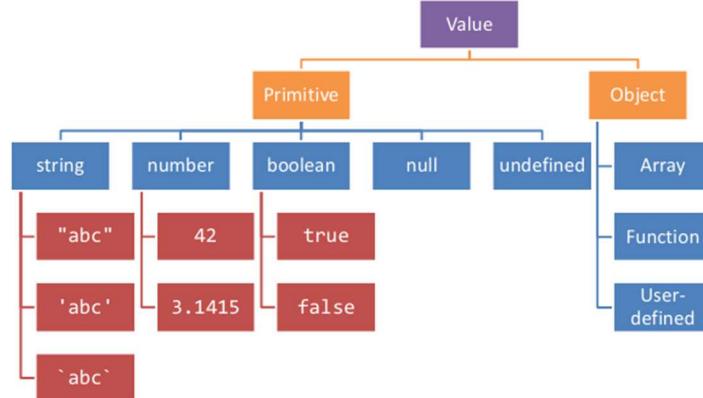
I laboratori vanno fatti scegliendo 1 tema d'esame passato ed ampliandolo ogni settimana con le nuove features.

1) JAVASCRIPT (Versione ES6)

JAVASCRIPT è 1 dei 3 linguaggi (con HTML e CSS) che un browser può eseguire nativamente, ma può anche essere eseguito in locale grazie a **Node.js**. Noi useremo JS **ES6** (**ECMAScript6**, detto anche **ES2015**). JS è “**backwards-compatible**” (retrocompatibile, ovvero tutto quello che viene accettato nello standard non sarà mai eliminato in futuro) ma non “**forwards-compatible**” (nuove versioni non girano su vecchi JS engine [su Chrome e Edge è **V8 Engine**]). È stata però introdotta la “**strict mode**” per disabilitare vecchie e pericolose semantiche.

Ogni programma JS è **1 singolo file** (ogni file viene caricato in modo indipendente); i file/programmi possono comunicare mediante “**global state**” (il meccanismo “**module**” estende lo sharing). Il file viene interamente analizzato e poi eseguito; è basato su libreria standard. È scritto in Unicode (supporta tutti i caratteri, anche le emoji). I punti e virgola “;” sono obbligatori (ma vengono comunque inseriti dall'editor). È case-sensitive e i commenti sono come in C (`// e /**/`). Gli identificatori (es. variabili) iniziano con lettera, \$ e _. La sintassi è “**C-like**”. La prima riga del file dovrà essere **“use strict”** ; (per la strict mode detta prima).

TIPI:



In JS **tutto** appartiene ai tipi primitivi sono **Oggetti** (anche le funzioni). Vediamo i tipi primitivi:

- **Boolean** (`true`, `false`) → quando convertiamo in booleano:
 - `0`, `-0`, `NaN`, `undefined`, `NULL`, `''` = diventano false (“**falsy**”)
 - `3` (number), `'false'` (string), `[]` (vettore vuoto), `{}` (oggetto vuoto)= diventano true (“**truthy**”)I confronti si fanno con `==` (*strict equal*) perché non converte automaticamente gli oggetti confrontati (altrimenti `==` che converte automaticamente).
- **Number** → no distinzione tra int e float (c'è anche `BigInt` per numeroni)
- **Valori speciali:**
 - `undefined` = variabile dichiarata, ma non inizializzata (`undefined` o `void`)
 - `null` = valore vuoto (insieme ad `undefined` sono detti “**nullish values**”)
 - `NaN` = Not a Number (es. valore aritmetico invalido come output di una funzione aritmetica)

VARIABILI: sono riferimenti “**pure**” (si riferiscono al valore) e possono assumere valori e tipi diversi (come in Python) in momenti diversi. Si dichiarano con `let` `a = 5`, `const` `b = 6` (queste 2 non hanno “hoisting” [cioè se uso la variabile prima della sua dichiarazione non mi dà errore], ovvero danno errore; `const` non si cambia) e `var` `c = 10` (questa è soggetta ad “hoisting”, ovvero non dà errore) [sconsigliato l'uso]. Anche qui posso riusare le variabili se sono in un altro **scope** (ovvero se sono in un altro blocco).

Gli **operatori** sono gli stessi degli altri linguaggi (con l'accortezza di `strict equal` e `strict not equal` vista prima):

Name	Shorthand operator	Meaning
Assignment	<code>x = y</code>	<code>x = y</code>
Addition assignment	<code>x += y</code>	<code>x = x + y</code>
Subtraction assignment	<code>x -= y</code>	<code>x = x - y</code>
Multiplication assignment	<code>x *= y</code>	<code>x = x * y</code>
Division assignment	<code>x /= y</code>	<code>x = x / y</code>
Remainder assignment	<code>x %= y</code>	<code>x = x % y</code>
Exponentiation assignment	<code>x **= y</code>	<code>x = x ** y</code>
Left shift assignment	<code>x <= y</code>	<code>x = x << y</code>
Right shift assignment	<code>x >>= y</code>	<code>x = x >> y</code>
Unsigned right shift assignment	<code>x >>>= y</code>	<code>x = x >>> y</code>
Bitwise AND assignment	<code>x &= y</code>	<code>x = x & y</code>
Bitwise XOR assignment	<code>x ^= y</code>	<code>x = x ^ y</code>
Bitwise OR assignment	<code>x = y</code>	<code>x = x y</code>

Operator	Description	Examples returning true
Equal (<code>==</code>)	Returns <code>true</code> if the operands are equal.	<code>3 == var1</code> <code>"3" == var1</code> <code>3 == '3'</code>
Not equal (<code>!=</code>)	Returns <code>true</code> if the operands are not equal.	<code>var1 != 4</code> <code>var2 != "3"</code>
Strict equal (<code>===</code>)	Returns <code>true</code> if the operands are equal and of the same type. See also <code>Object.is</code> and sameness in JS.	<code>3 === var1</code>
Strict not equal (<code>!==</code>)	Returns <code>true</code> if the operands are of the same type but not equal, or are of different type.	<code>var1 !== "3"</code> <code>3 !== '3'</code>
Greater than (<code>></code>)	Returns <code>true</code> if the left operand is greater than the right operand.	<code>var2 > var1</code> <code>var2 > 3</code> <code>"12" > 2</code>
Greater than or equal (<code>>=</code>)	Returns <code>true</code> if the left operand is greater than or equal to the right operand.	<code>var2 >= var1</code> <code>var1 >= 3</code>
Less than (<code><</code>)	Returns <code>true</code> if the left operand is less than the right operand.	<code>var1 < var2</code> <code>"2" < 12</code>
Less than or equal (<code><=</code>)	Returns <code>true</code> if the left operand is less than or equal to the right operand.	<code>var1 <= var2</code> <code>var2 <= 5</code>

Il confronto tra oggetti diversi con gli stessi valori all'interno sarà sempre `false` perché avviene per riferimento.

Le **conversioni** avvengono come mostrato in figura; avvengono automaticamente nei confronti se uso `==` e `!=`, ma non avvengono con `==` e `!=`. Ci sono le funzioni e costanti matematiche accessibili con **Math**.

Per quanto riguarda i costrutti di flusso abbiamo i soliti del C con la stessa sintassi:

- **if - else if - else**

- **switch**

- **for(*i*=0; *i* < N; *i*++)**

⚠ Ci sono anche 2 "for" nuovi:

```
for( let a in {x: 0, y:3} ) {
    console.log(a) ;
}
```

```
x  
y
```

```
for( let a of [4,7] ) {
    console.log(a) ;
}
```

```
4  
7
```

```
for( let a of "hi" ) {
    console.log(a) ;
}
```

```
h  
i
```

- **while; do-while**

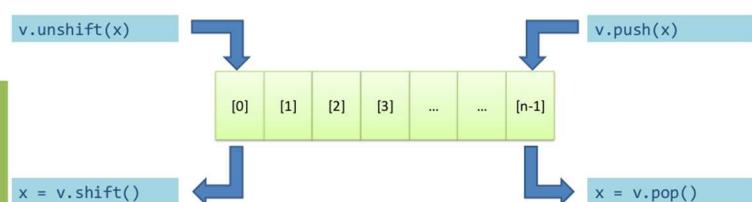
- **try-catch-finally; throw**

Riguardo agli **ARRAY** (`vet[]`; si può accedere alla lunghezza con `vet.length`) vediamo come:

- **CREARE** → `let v = [1, 2, 3]; let v = Array.of(1,2,3); let v = [1, "Ciao", 3.1]; let v = Array.of(1, "Ciao", 3.1);`
- **AGGIUNGERE/RIMUOVERE ELEMENTI:**

```
let v = [] ;  
v[0] = "a" ;  
v[1] = 8 ;  
v.length // 2
```

```
let v = [] ;  
v.push("a") ;  
v.push(8) ;  
v.length // 2
```



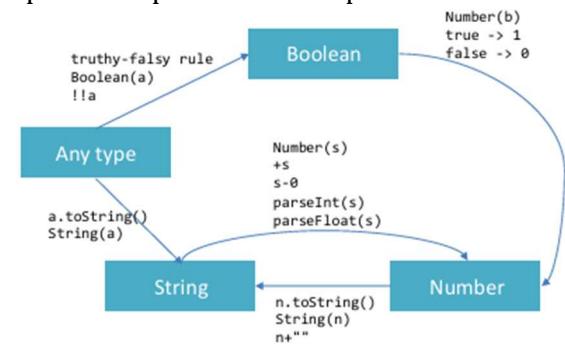
- **COPIARE** → `let v = []; let alias = v` (copia il riferimento a `v`); `let copy = Array.from(v)` (copia il vettore)
- **ITERARE** → ciclo `for` base oppure `for of`
- **METODI:**

- `.concat()`
 - joins two or more arrays and returns a **new** array.
- `.join(delimiter = ',')`
 - joins all elements of an array into a (new) string.
- `.slice(start_index, upto_index)`
 - extracts a section of an array and returns a **new** array.
- `.splice(index, count_to_remove, addElement1, addElement2, ...)`
 - removes elements from an array and (optionally) replaces them, **in place**
- `.reverse()`
 - transposes the elements of an array, **in place**
- `.sort()`
 - sorts the elements of an array **in place**
- `.indexOf(searchElement[, fromIndex])`
 - searches the array for `searchElement` and returns the `index` of the first match
- `.lastIndexOf(searchElement[, fromIndex])`
 - like `indexOf`, but starts at the end
- `.includes(valueToFind[, fromIndex])`
 - search for a certain value among its entries, returning `true` or `false`

⚠ Un metodo particolare è lo **SPREAD (...)**:

```
let [x, ...y] = [1,2,3,4]; // we obtain y == [2,3,4]
```

```
const parts = ['shoulders', 'knees'];
const lyrics = ['head', ...parts, 'and', 'toes']; // ["head", "shoulders",
"knees", "and", "toes"]
```



Riguardo alle **STRINGHE**, una stringa è una sequenza immutabile di caratteri Unicode; anche qui esiste `stringa.length`. Non esiste il `char` ma ci sono solo stringhe da 1 carattere e possono essere definite da '`abc`' o da "`abc`". Tutte le operazioni ritornano nuove stringhe e non modificate (immutabili).

Con **Template literals** si intende:

```
let name = "Bill";
let greeting = `Hello ${ name }.`;
// greeting == "Hello Bill."
```

METODI:

Method	Description
<code>charAt</code> , <code>charCodeAt</code> , <code>codePointAt</code>	Return the character or character code at the specified position in string.
<code>indexOf</code> , <code>lastIndexOf</code>	Return the position of specified substring in the string or last position of specified substring, respectively.
<code>startsWith</code> , <code>endsWith</code> , <code>includes</code>	Returns whether or not the string starts, ends or contains a specified string.
<code>concat</code>	Combines the text of two strings and returns a new string.
<code>fromCharCode</code> , <code>fromCodePoint</code>	Constructs a string from the specified sequence of Unicode values. This is a method of the String class, not a String instance.
<code>split</code>	Splits a String object into an array of strings by separating the string into substrings.
<code>slice</code>	Extracts a section of a string and returns a new string.
<code>substring</code> , <code>substr</code>	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
<code>match</code> , <code>matchAll</code> , <code>replace</code> , <code>search</code>	Work with regular expressions.
<code>toLowerCase</code> , <code>toUpperCase</code>	Return the string in all lowercase or all uppercase, respectively.
<code>normalize</code>	Returns the Unicode Normalization Form of the calling string value.
<code>repeat</code>	Returns a string consisting of the elements of the object repeated the given times.
<code>trim</code>	Trims whitespace from the beginning and end of the string.

Ora parliamo di **OGGETTI e FUNZIONI**:

- **OGGETTI** → diversamente da Java, in JavaScript un oggetto **esiste senza la sua classe** (solitamente gli oggetti vengono creati direttamente). È **dinamico** (si può aggiungere, togliere, ridefinire una proprietà o un metodo) e tutti i metodi e proprietà sono sempre **public** (non esiste altro). Per come funziona JS, non c'è una reale differenza tra proprietà e metodi. Un oggetto ha sintassi `let object1 = {name: value, name: value, ...}` (come si vede in figura) dove `value` può essere primitivo, vettoriale, oggetto, funzioni etc... Ai **singoli campi** si accede e si modificano con 2 notazioni:

```
let person = book.author;           book.title = "Advanced JS";
let name = book["author"];          book["pages"] = 340;
```

```
let book = {
  author : "Enrico",
  title : "Learning JS",
  for: "students",
  pages: 520,
};
```

La notazione con `[]` si chiama "**associative arrays**" perché si accede come un vettore, ma l'indice è il nome della proprietà dell'oggetto. Se faccio `oggetto["nome"] = valore` (oppure `oggetto.nome = valore`) e la proprietà non esiste ancora, la **creo** automaticamente. Le proprietà si eliminano con `delete oggetto["nome"]` (oppure `delete oggetto.nome`) [⚠ se invece faccio `oggetto[variabile]` senza "", sto accedendo ad `oggetto["nome"]` se `variabile = "nome"`]. Se provo ad accedere ad una **proprietà non esistente**, ricevo in ritorno **undefined** (ricorda che `undefined` è falsy).

Le proprietà (come abbiamo visto prima nel for) sono **iterabili**:

```
for( let a in {x: 0, y:3} ) {
  console.log(a);
}
x
y
```

```
let book = {
  author : "Enrico",
  pages: 340,
  chapterPages: [90,50,60,140],
};

for (const prop in book)
  console.log(`$ {prop} = ${book[prop]}`);

author = Enrico
pages = 340
chapterPages = 90,50,60,140
```

Metto in un vettore i nomi delle variabili con `Object.keys(oggetto)` e i valori con `Object.entries(oggetto)`. Anche qui, se voglio **COPPIARE** un oggetto posso creare un riferimento-copia (**alias**) con `let oggetto2 = oggetto`, ma se voglio un'effettiva **copia** devo usare `let oggetto2 = Object.assign({}, oggetto)` in quanto

la sintassi è `Object.assign(destinazione, sorgente)` [destinazione può anche essere un oggetto esistente che voglio modificare] [⚠ ricordiamo che quando si crea un'effettiva copia bisogna ricordarsi che è un duplicato e che non subirà le stesse modifiche dell'originale una volta creato, ovvero "shallow copy"]. Con la `assign` posso **aggiungere proprietà** ad un oggetto facendo `let oggetto2 = Object.assign(oggetto, {name: value})` creando una copia dell'oggetto anche con questa nuova proprietà, oltre che aggiungendola all'oggetto originale.

⚠ In ES9 è stata aggiunta la possibilità di copiare oggetti con lo **spread operator** `let oggetto2 = {...oggetto}`

Ciao

Per **controllare che una proprietà esista** si può usare l'operatore `in` (⚠ non funziona con gli array), ovvero 'proprietà' `in` oggetto (darà true se c'è, false altrimenti).

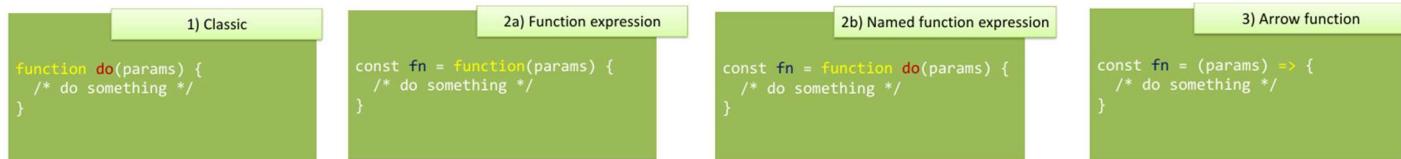
⚠ Un oggetto si può **creare** anche usando `let oggetto = new Object()` oppure con `let oggetto = Object.create({nome: valore, nome: valore, ...})`, ma si preferisce il metodo normale con le graffe

- **FUNZIONI** → delimita il codice con **scope** privato; può accettare parametri e ritornare un valore (anche un oggetto). Le **funzioni sono oggetti** (posso assegnarle ad una **variabile**, posso passarle come **argomento**, posso **ricontrarle**).

I parametri sono passati per valore e possono anche avere un valore di **default** (es. `function(a, b=1)`) in modo che, se non viene passato (cioè `undefined`), nella funzione potrà fare `if (p==undefined) p = default_value`. Posso avere un **numero variabile di parametri** con `function(par1, par2, ...arr)` (es. in figura).

Ci sono 3 modi per **DICHIARARE** una funzione:

```
function sumAll(initVal, ...arr) {
  let sum = initVal;
  for (let a of arr) sum += a;
  return sum;
}
sumAll(0, 2, 4, 5); // 11
```



Ora vediamo a confronto **function expression** con **arrow function**:

<code>function square(x) { let y = x * x ; return y ; }</code>	<code>let cube = function c(x) { let y = square(x)*x ; return y ; }</code>	<code>let fourth = (x) => { return square(x)*square(x) ; } let n = fourth(4) ;</code>
--	--	---

Nelle arrow functions con **più parametri** avrò `let variabile = (par1, par2=2) => { /*corpo funzione*/ }`. Il **valore di ritorno** di **default** è `undefined`. Con **return** si torna 1 solo valore/oggetto/vettore. Nelle **arrow functions** contenenti un solo valore (es. `let fourth = x => square(x)*square(x) ;`) il **return** è implicito e torna il valore.

Si possono avere **NESTED FUNCTIONS** →

```
function hypotenuse(a, b) {
  const square = x => x*x ;
  return Math.sqrt(square(a) + square(b));
}
```

Una **CLOSURE** è quando una nested function eseguita dopo l'esecuzione dell'outer function può ancora accedere allo scope della outer function (es. una **inner function può accedere alle variabili della outer function che la racchiude**, com'è giusto che sia). Si può usare per emulare gli oggetti:

<code>"use strict"; function counter() { let value = 0 ; const getNext = () => { value++; return value; } return getNext ; }</code>	<code>const count1 = counter() ; console.log(count1()) ; console.log(count1()) ; console.log(count1()) ; const count2 = counter() ; console.log(count2()) ; console.log(count2()) ; console.log(count2()) ;</code>	1 2 3 1 2 3
---	---	----------------------------

Si possono anche usare le **IIFE (Immediately Invoked Function Expressions)** →

```
"use strict";  
  
const c = (  
  function () {  
    let n = 0;  
  
    return {  
      count: function () {  
        return n++; },  
      reset: function () {  
        n = 0; }  
    };  
}());
```

<code>console.log(c.count()); console.log(c.count()); c.reset(); console.log(c.count()); console.log(c.count());</code>	0 1 0 1
---	------------------

⚠ Abbiamo detto che gli oggetti si possono creare e usare senza classi, ma se vogliamo fare in modo di avere una classe e di crearne gli oggetti possiamo fare come in Java con this nella classe e il costruttore new

- **DATE** → le date sono ben gestite, ma i fusi orari no. Ci sono varie librerie per gestire le date, noi useremo day.js. Tutti gli oggetti che torna sono nuovi (non modifica date) in quanto come le stringhe sono immutabili:

Creating date objects – dayjs() constructor

```
let now = dayjs() // today
let date1 = dayjs('2019-12-27T16:00');
    // from ISO 8601 format
let date2 = dayjs('20191227');
    // from 8-digit format
let date3 = dayjs(new Date(2019, 11, 27));
    // from JS Date object
let date5 = dayjs.unix(1530471537);
    // from Unix timestamp
```

By default, Day.js parses in local time

Displaying date objects – format()

```
console.log(now.format());
2021-03-02T16:38:38+01:00

console.log(now.format('YYYY-MM [on the] DD'));
2021-03 on the 02

console.log(now.toString());
Tue, 02 Mar 2021 15:43:46 GMT
```

By default, Day.js displays in local time

Ci sono metodi per modificare le date (posticipare o anticipare la data), metodi per settare la data, per trovare year, month, day, hour, minute etc... C'è anche il **toString()** per convertire in stringhe. Si possono anche installare dei plugin su day.js in modo da espandere le funzionalità. C'è anche il **days()** per trovare il momento corrente (crea un oggetto dayjs). Ci sono anche i metodi sugli oggetti dayjs che permettono di capire se una data è dopo o prima (**isAfter** e **isBefore**) o anche **isBetween** (che richiede un plugin esterno). La Z al fondo della stringa data indica UTC. Dentro la proprietà "M" (ovvero mese) vediamo che marzo è 2 e non 3 (perché vanno da 0 a 11).

In generale noi useremo **npm init** per creare un progetto JavaScript (in modo che potremo mettere tutte le dipendenze nel progetto e averlo pronto). Le librerie si installano (es. dayjs) con **npm install** (o solo **i**) **dayjs**. Una volta installate le dipendenze, se le cancello le loro info rimarranno nel **package.json** del mio progetto e quindi con il comando **npm install** si riscaricheranno tutte automaticamente

2) CALLBACKs, METODI FUNZIONALI e INTERAZIONE con DB

CALLBACK = funzione passata in un'altra funzione come argomento, che verrà poi invocata nella outer function (**sincrona** o **asincrona** [non aspetta la fine della funzione; in JavaScript è un multithread asincrono solo emulato]). Avevamo visto una callback sincrona nell'esempio del sorting:

```
let numbers = [4, 2, 5, 1, 3];
numbers.sort(function(a, b) {
    return a - b;
});
console.log(numbers);
```

```
let numbers = [4, 2, 5, 1, 3];
numbers.sort((a, b) => a - b);
console.log(numbers);
```

```
function logQuote(quote) {
    console.log(quote);
}

function createQuote(quote,
callback) {
    const myQuote = `Like I always
say, '${quote}'`;
    callback(myQuote);
}

createQuote("WebApp I rocks!",
logQuote);
```

JavaScript permette di usare **METODI FUNZIONALI** (meno righe di codice) ["**programmazione funzionale**" = si tende a fare tutto con le funzioni]:

```
new_array =
array.filter ( filter_function ) ;
```

```
new_array = [] ;
for (const el of list)
    if ( filter_function(el) )
        new_array.push(el) ;
```

I principi della programmazione funzionale sono:

- **First-class citizens** (le funzioni sono cittadini di prima classe, ovvero possono essere usate come variabili o costanti) [questo perché in JS le funzioni sono oggetti]
- **High-order functions** (funzioni che operano su funzioni e ritornano funzioni)
- **Function composition**
- **Call chaining**
- **Avoiding mutability** (creare nuove istanze di oggetti modificate piuttosto che modificare le esistenti)

La programmazione funzionale si cerca di usare **specialmente sugli array**; infatti per iterare sui vettori, posso usare (al posto del ciclo for):

- **forEach(f)** → processa ogni elemento con la callback f. Può avere 3 parametri:

- o **currentValue** = elemento corrente dell'array
- o **index** (opzionale) = indice dell'elemento corrente dell'array
- o **array** (opzionale) = l'array su cui ho chiamato il forEach

Ritorna **undefined** ed è non-chainable; per fermare un forEach non si può usare un break o una stop, ma bisogna lanciare un eccezione

```
const letters = [..."Hello world"];
let uppercase = "";
letters.forEach(letter => {
    uppercase += letter.toUpperCase();
});
console.log(uppercase); // HELLO WORLD
```

- **every(f), some(f)** → controlla se tutti/alcuni degli elementi soddisfano la callback booleana f

⚠ **every** ha gli stessi argomenti del forEach, ma ritorna un valore **truthy/falsy** (infatti si ferma quando trova un valore falsy rispetto alla callback)

⚠ **some** è come every, ma appena trova un elemento che soddisfa la callback, torna true

```
let a = [1, 2, 3, 4, 5];
a.every(x => x < 10); // => true: all values are < 10
a.every(x => x % 2 === 0); // false: not all even values
```

```
let a = [1, 2, 3, 4, 5];
a.some(x => x%2==0); // => true; a has some even numbers
a.some(isNaN);
```

- **map(f), filter(f)** → ritornano un nuovo array, partendo dal precedente

⚠ **map** ha una callback che deve tornare per forza un valore, in quanto la map torna un nuovo vettore contenente gli elementi ritornati dalla callback

⚠ **filter** ritorna un array con gli elementi dell'array originale che passano il filtro; la sua callback non deve tornare il valore, ma deve tornare true/false per ogni elemento (se nessun elemento passa il filtro, torna un array vuoto)

```
const a = [1, 2, 3];
const b = a.map(x => x*x);
console.log(b); // [1, 4, 9]
```

```
const a = [5, 4, 3, 2, 1];
a.filter(x => x < 3); // generates [2, 1], values less than 3
a.filter((element, index) => index%2 == 0); // [5, 3, 1]
```

```
const a = [5, 4, 3, 2, 1];
a.reduce( (accumulator, currentValue) =>
    accumulator + currentValue,
    0);
// 15; the sum of the values

a.reduce((acc, val) => acc*val, 1);
// 120; the product of the values

a.reduce((acc, val) => (acc > val) ? acc : val);
// 5; the largest of the values
```

- **reduce** → callback su tutti gli elementi dell'array per calcolare progressivamente il risultato (da array ottengo un **singolo valore**). Ha 2 possibili argomenti: la callback (che riduce 2 valori in 1) e un **initialValue** (opzionale) da passare alla callback (se non specificato, **initialValue = array[0]**). La callback usata con la reduce ha 2 valori: **accumulator** (il valore accumulato) e **currentValue**

⚠ I metodi funzionali si comportano **come gli stream** in Java (si possono **concatenare** [eccetto il forEach])

Esempio di oggetti e costruttori:

```
import dayjs from "dayjs";

// Costruttore di Answer
/// metto come default di score il valore 0 se non presente, ma come faccio a saltarlo? mi conviene metterlo in fondo per usare il default
function Answer(text, username, date, score=0) {
    this.text = text;
    this.username = username;
    this.date = dayjs(date); // converto la stringa data in un oggetto dayjs
    this.score = score;

    // se c'è un metodo toString in JavaScript, questo viene automaticamente chiamato in Stampa (come in Java)
    this.toString = () => {
        return `${this.username} replied '${this.text}' on ${this.date.format('YYYY-MM-DD')}`
    }
}

// Costruttore di Question
function Question(text, username, date) {
    this.text = text;
    this.username = username;
```

```

this.date = dayjs(date);
this.answers = [] // all'inizio, la domanda avrà 0 risposte, si aggiungeranno man mano

// metodo addAnswer (definisco nella classe come arrow function)
this.add = (answer) =>{
    this.answers.push(answer);
}

this.find = (username) => {
    // senza metodo funzionale
    /*const foundAnswers = []
    for(const ans of this.answers){
        if(ans.username === username)
            foundAnswers.push(ans);
    }
    return foundAnswers;*/
    // metodo funzionale
    return this.answers.filter(ans => ans.username === username);
}

this.afterDate = (date) => {
    return this.answers.filter(ans => ans.date.isAfter(dayjs(date)));
}

this.listByDate = () => {

    return [...this.answers].sort((a,b) => (a.date.isAfter(b.date) ? 1 : -1));
    // return [...this.answers].sort((a,b) => (a.date.isAfter(b.date) ? -1 : 1)); con il
sort opposto
    //!! [...array] per copiare array e quindi non modificare l'array originale, ma crearne
una copia e darla in return ordinata
}

this.listByScore = () => {
    // DECREScente
    return [...this.answers].sort((a,b) => b.score - a.score);
}
}

const question = new Question('Is JS better than Python?', 'Mattia Domizio', '2025-03-03');
const firstAnswer = new Answer('Yes', 'Luca Mannella', '2025-03-04', -10);
const secondAnswer = new Answer('No', 'Guido van Rossum', '2025-03-07', 5);
const thirdAnswer = new Answer('No 2', 'Albert Einstein', '2025-03-08');
const fourthAnswer = new Answer('I don\'t know', 'Luca Mannella', '2025-03-09')

question.add(firstAnswer);
question.add(secondAnswer);
question.add(thirdAnswer);
question.add(fourthAnswer);

const answersByLuca = question.find('Luca Mannella');
console.log(question);
console.log("\nAnswers by Luca: " + answersByLuca);
console.log('\nBy date: ' + question.listByDate());
console.log('\nBy score: ' + question.listByScore());
console.log('\nAfter 2025-03-06: ' + question.afterDate('2025-03-06'));

```

PROGRAMMAZIONE ASINCRONA → anche se JS è single-threaded e sincrono, tramite l'ambiente di esecuzione (“Execution Environment”; browsers, Node.js...) e il ciclo di eventi (“Event Loop”; controlla con un polling se gli eventi nella coda vadano eseguiti o no) si può fare **programmazione asincrona** (eventi/trigger che richiamano una parte di codice non scritta sequenzialmente)

```

const deleteAfterTimeout = (task) =>
{
    // do something
}
// runs after 2 seconds
setTimeout(deleteAfterTimeout, 2000,
task)

```

⚠️ Programmazione asincrona **molti importanti nella programmazione web** (quindi specialmente nei browser) per evitare che il sito sembri bloccato

Come abbiamo visto le callbacks sincrone, ora vediamo le **CALLBACK ASINCRONE** (molto utili per gestire l'**input** dell'utente, le **operazioni di I/O**, intervalli di tempo, **interazione con i database**).

Per ritardare l'esecuzione di una funzione c'è:

- **setTimeout(funzione, timeout, parametro1, parametro2...);** → runna la callback function dopo un certo periodo di tempo
- **setInterval(funzione, timeout, parametro1, parametro2...);** → runna callback periodicamente

⚠️ Il timeout value è in **ms** e deve essere < $2^{31} - 1$ (circa 24 giorni)

```
const onese = setTimeout(()=> {
    console.log('hey') ; // after 1s
}, 1000) ;

console.log('hi') ;
```

```
const myFunction = (firstParam,
secondParam) => {
    // do something
}
// runs after 2 seconds
setTimeout(myFunction, 2000,
firstParam, secondParam) ;
```

Per stoppare l'esecuzione periodica di setInterval c'è **clearInterval(nome_variabile_setInterval)**

⚠️ Nelle callback non c'è un modo ufficiale per **gestire errori**, ma solo una **best practice** che prevede mettere nel 1° parametro della callback l'errore, nel 2° il risultato corretto

```
fs.readFile('/file.json', (err, data) => {
    if (err !== null) {
        console.log(err);
        return;
    }
    // no errors, process data
    console.log(data);
});
```

Ora vediamo **DATABASE ACCESS con SQLITE** (in quanto è proprio **esempio di programmazione asincrona**) [noi useremo **sqlite3**]. Ad inizio programma bisogna **importare sqlite3** (`import sqlite from 'sqlite3';`) e creare il **riferimento al DB** con:

```
const db = new sqlite.Database('nome_db.sqlite', (err) => { if (err) throw err; });
```

Le **QUERY** sono da scrivere con **const query = "SELECT..."** e sono usabili dal db con:

- **db.all(query, [parametri], (err, rows) => { })** → esegue la query e, se non si verifica errore, torna tutte le righe (pensato per le query di tipo SELECT) nell'array **rows**
- **db.get(query, [parametri], (err, row) => { })** → esegue la query e, se non si verifica errore, torna solo la 1^ riga risultato in **row**

⚠️ Se la query è **errata**, avremo err; se invece la query è **vuota**, non avremo err (quindi **attenzione!!!**)

- **db.each(query, [parametri], (err, row) => { })** → esegue la query e, se non si verifica errore, per ogni risultato [non lo useremo]

- **db.run(query, [parametri], function (err) { })** → per query che modificano il mio DB ma non tornano un valore (no SELECT; es. CREATE, UPDATE, INSERT)

⚠️ In questa callback ci sono 2 **variabili** date direttamente da sqlite (possiamo usarle con **this** proprio perché **non sto usando la sintassi arrow function** vista nelle altre callback [la arrow function ridefinisce il this, quindi si perde il collegamento con il this di sqlite]):

- **this.changes** = n° di righe modificate
- **this.lastID** = id della riga inserita (con le INSERT)

Come abbiamo visto ad Introduzione alle Applicazioni Web, i **PARAMETRI** vengono passati nelle queries al posto dei **"?"(placeholders)** e non si usa la concatenazione di stringhe (questo perché i parametri vengono **controllati/sanificati**)

```
const sql = 'SELECT * FROM course WHERE code=?';
db.get(sql, [code], (err, row) => {
```

Vediamo un esempio di asincronicità delle interazioni con il db (che verrà poi migliorato con le Promises):
`import sqlite from 'sqlite3';`

```
const db = new sqlite.Database('questions.sqlite', (err) => {
    // throw eccezione di err
    if (err) throw err;
});
```

```

let sql = 'SELECT * FROM answer'; // prendo tutto dalla tabella answer
let results = [];

db.all(sql, [], (err, rows) => {
  if (err) throw err;
  for (let row of rows)
    results.push(row);
});

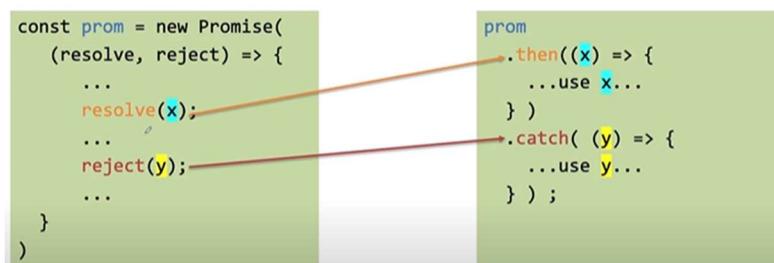
for (let r of results)
  console.log(r);

// Output: [] perché il ciclo for viene eseguito prima che la callback di db.all() venga
// chiamata in quanto è asincrona
// Per risolvere il problema, si può spostare il ciclo for dentro la callback di db.all(),
// invece che salvare i risultati in un vettore e poi stamparli

```

Se vogliamo quindi fare più azioni asincrone diverse ma chiamate tutte nello stesso momento bisognerebbe chiamare tantissime callbacks nested ("callback hell"); per evitare questa situazione infernale e semplificare la programmazione asincrona, sono state introdotte le **PROMISES** ("promesse"), ovvero oggetti che verranno riempiti a fine operazione asincrona con completion o failure. Le promesse possono essere create o consumate; quando vengono consumate, una promessa parte in stato **pending** e poi alla fine se completion = stato **fulfilled**, se failure = stato **rejected**. Per **CREARE** una Promise si usa il suo costruttore con sintassi qui a sotto.

Per **CONSUMARE** una Promise si usa la callback `promise.then((result) => {...}).catch((error) => {...})` dove then gestisce lo stato **fulfilled**, mentre catch lo stato **rejected** (qui a lato); c'è anche il **.finally** che gestisce entrambi gli stati (come già visto in altri linguaggi).



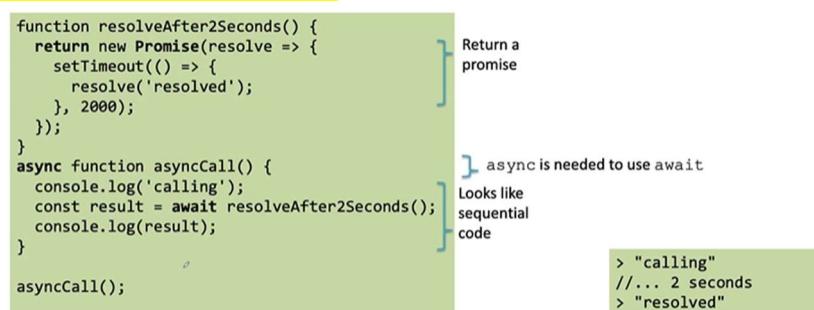
⚠ Le promises si possono anche **concatenare** tra loro! Inoltre, le promesse le vedremo tornate direttamente dal server e le gestiremo e vedremo anche con il **fetch** (qui le consumeremo soltanto, mentre con sqlite le creiamo)

Le promises possono essere **eseguite in parallelo** con:

- **Promise.all()** → prende tutte le promises (messe come argomento) e, se sono state **tutte fulfilled**, ritorna **array con valori di tutte le promises** risolte
- **Promise.race()** → ritorna la **1^ delle promises** nell'array di input (argomento) che viene fulfilled o rejected

Un modo diverso di scrivere le promises (specialmente utile se ho tante promises concatenate, che quindi è meglio di avere tante callback concatenate, ma comunque è ostica come sintassi e confusionaria) è tramite **2 KEYWORDS** da mettere prima della funzione, ovvero:

- **async** → la funzione tornerà una **Promise** (implicita)
- **await** → funziona **solo su una async function**, stoppa il codice fino a che la **Promise** non è fulfilled/rejected



⚠ In pratica il **.then** viene sostituito dalla **await**

3) EXPRESS e API HTTP

Nell'**ARCHITETTURA WEB** (distribuita e multi-client) abbiamo più **CLIENT** (invia richieste HTTP [protocollo di rete "richiesta-risposta" scritto in linguaggio "umano"] [**browser**]) e il **SERVER** (elabora e risponde alle richieste mettendo pagine **HTML generate al momento** nelle risposte [+ ha il **DB** collegato]).

L'architettura classica risponde sempre ad ogni richiesta con una pagina HTML, ma noi vedremo un'architettura diversa nel corso, dove il server alla 1^a richiesta risponde in modo classico (manda pagina HTML con CSS e JS), ma nelle **richieste successive** fa cose diverse (non risponde più con la pagina, ma risponde con un **json** [con i dati nuovi] ed è il client a fare il rendering dei nuovi dati [**SINGLE PAGE APPLICATION**]) [è così che funziona **REACT**].

⚠ Ripasso veloce di **METODI HTTP** (sulle **richieste HTTP**) [la **risorsa** a cui si fa riferimento sono i dati sul server]:

GET	Requests a representation of the specified resource. Should only retrieve data.
HEAD	Asks for a response identical to GET, but without the response body
POST	Submit an entity to the specified resource, often causing a change in state or side effects on the server
PUT	Replaces current representations of the target resource with the request payload
DELETE	Deletes the specified resource
TRACE	Message loop-back test along the path to the target resource
OPTIONS	Describe the communication options for the target resource
CONNECT	Establish a tunnel to the server identified by the target resource
PATCH	Apply partial modifications to a resource

E analogamente sulle **risposte HTTP** (ovvero delle status line composte da 3 campi separati da spazi: **HTTP-version status-code description-of-status-code** → es. **HTTP/1.0 200 OK** oppure **HTTP/1.0 404 Not Found**), dove gli status code hanno formato **3 numeri** che iniziano con:

- **1xx = informational**
- **2xx = success**
- **3xx = redirection**
- **4xx = client error**
- **5xx = server error**

Method	Request Body	Response Body	Idempotent	HTML Forms
GET	No	Yes: resource content	Yes	Yes
HEAD	No	No	Yes	No
POST	Yes: form data or application data	May (usually modification results)	No	Yes
PUT	Yes: application data	May (usually modification results)	Yes	No
DELETE	May	May	Yes	No

I **body** (corpo) di richiesta e risposta sono organizzati così →

Il nostro obiettivo sarà implementare questo semplice server in JavaScript (json è parte di JS) e che sia capace di fare hosting di contenuti statici e dinamici, usando come server web **EXPRESS** (web framework di Node.js) [va installato come sempre con **npm install express**]. Quando avviamo il server express (con **node file.js**), rimane in esecuzione fino ad un **crash** o se interrotto da noi da terminale con **ctrl+C**; se modifichiamo il file, va stoppato e riavviato per la modifica (c'è il modulo **nodemon** [da installare con **npm install -g nodemon**] che rileva le modifiche al file.js e **riavvia automaticamente node se rileva la modifica**) [infatti noi spesso lanceremo i file con **nodemon file.js**].

Un progetto express di base avrà 3 parti:

- **express()** = crea un oggetto app
- **get()** = definisce routes e web pages; le richieste HTTP sono mandate alle **callback** a seconda del **path** (es. '/') e **metodo** (es. **get**); la callback riceve gli oggetti **richiesta** e **risposta** (**req,res**)
- **listen()** = **starta** il server su una **porta** specificata e manda messaggio di log

La parte che viene modificata nei nostri programmi sarà quella a metà, ovvero la parte di logica della nostra pagina. Come abbiamo detto prima, la sintassi è **app.method(path,handler)** dove:

- **app** → istanza di express
- **method** → HTTP request method (get, post, put, delete) [con **app.all()** si prendono tutti i tipi di richiesta]
- **path** → percorso sul server
- **handler** → callback eseguita quando il path è matchato; le **proprietà degli oggetti req e res** sono:

```
// Import package
import express from 'express' ;
// Create application
const app = express() ;

// Define routes and web pages
app.get('/', (req, res) =>
    res.send('Hello World!')) ;

// Activate server
app.listen(3000, () =>
    console.log('Server ready')) ;
```

req (Request object)

Property	Description
.app	holds a reference to the Express app object
.baseUrl	the base path on which the app responds
.body	contains the data submitted in the request body (must be parsed and populated manually before you can access it)
.cookies	contains the cookies sent by the request (needs the cookie-parser middleware)
.hostname	the server hostname
.ip	the server IP
.method	the HTTP method used
.params	the route named parameters
.path	the URL path
.protocol	the request protocol
.query	an object containing all the query strings used in the request
.secure	true if the request is secure (uses HTTPS)
.signedCookies	contains the signed cookies sent by the request (needs the cookie-parser middleware)
.xhr	true if the request is an XMLHttpRequest

res (Response object)

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

<https://expressjs.com/en/guide/routing.html>

Quindi, come vediamo dalla tabella, quando genero una risposta HTTP abbiamo vari metodi su res:

- res.send("Ciao") = imposta il body della risposta e la manda al browser
- res.end() = manda risposta vuota
- res.status(status_code) = setta lo status code della risposta
- res.json() = manda oggetto serializzandolo in un json

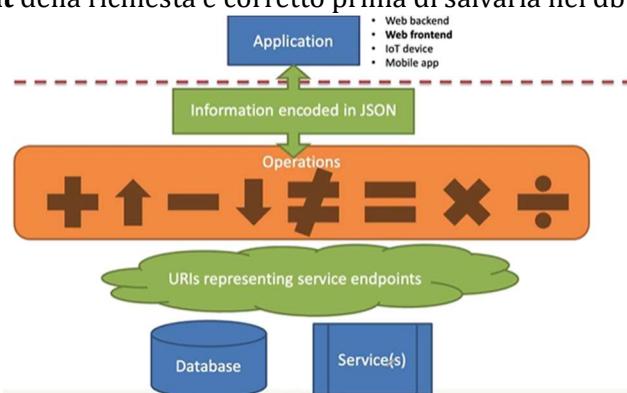
Oltre alle funzionalità di base, si può espandere Express con dei **Middleware** (una funzione chiamata per ogni richiesta → `function(req, res, next)` = riceve (req,res), li processa e chiama `next()` per attivare la middleware function). Si può inserire una middleware in una route specifica con `app.method(path, middlewareCallback, (req,res) => {})`, ovvero tra path e handler; oppure si può registrare una middleware con `app.use(middlewareCallback)` o `app.use(path,middlewareCallback)` su tutto il server [noi per esempio useremo il middleware `json` e il middleware `express.static(root,[options])` per le richieste statiche (es. png) e lo imposteremo con `app.use(express.static("nome_cartella"))` a livello globale di server]

⚠ Se la richiesta HTTP (es. la get qui in tabella) ha ?parametro=valore¶metro=valore..., questi parametri possono essere presi senza middleware, ma usando la proprietà `req.query`; se invece vogliamo il corpo di una richiesta, lo prenderemo con `req.body`, ma useremo dei **middleware** (come `urlencoded` e `json`):

Request method	Parameters	Values available in	Middleware required
GET	URL-encoded /login?user=fc&pass=123	req.query req.query.user req.query.pass	none
POST / PUT	FORM-encoded in the request body	req.body	express.urlencoded()
POST / PUT	JSON stored in the request body { "user": "fc", "pass": "123" }	req.body.user req.body.pass	express.json()

⚠ I PATH possono contenere `regexp` e, se ho la stessa route su più path, possiamo usare dei **vettori di path**; i path possono essere anche parametrici (es. '/users/:userId') con sintassi '`:campo`' e possiamo accedere a questi campi con `res.params.campo`

⚠ Express non ha modo di fare **logging**, ma si può usare un middleware di nome `morgan` per farlo [`npm install morgan; const morgan = require('morgan'); app.use(morgan('dev'));`]; un altro middleware utile è `express-validator` che **valida** se l'**input** della richiesta è corretto prima di salvarla nel db



JSON è un formato leggero di scambio dati e fa parte di JS; il suo **media type** in HTTP è **application/json**. I tipi primitivi sono **string** (" " e non '), **number**, **true/false/null**; abbiamo poi **vettori** [...] e **oggetti** {key: value, key:value...}. Ci sono metodi utili in JS per trattare json:

- **JSON.stringify** → converte **oggetto in JSON** (stringhe json); funziona ricorsivamente nei nested objects/arrays e esclude metodi e valori undefined
- **JSON.parse** → converte la stringa **JSON in un oggetto**, seguendo la struttura del json

A quali URL passiamo questi oggetti json nelle richieste e risposte HTTP? Abbiamo:

- **Collection URI** = rappresenta una collezioni di oggetti dello stesso tipo (<http://api.polito.it/students>)
- **Element URI** = rappresenta il singolo elemento della collezione (<http://api.polito.it/students/s123456>)

⚠ La convenzione è usare nomi e non verbi negli url; su queste collezioni e elementi faremo **operazioni HTTP**: **GET** se voglio nel response body ottenerli, **POST** se voglio crearli, **PUT** se voglio fargli update e **DELETE** per cancellarli (seguendo ciò che è **convenzionale** nell'immagine qui a lato →)

Resource	GET	POST	PUT	DELETE
Collection	Retrieve the list of items	Add a new element to the collection	-	-
Single Element	Retrieve the properties of the element	-	Replace the values of the element properties	Delete the element

Standard Method	HTTP Mapping	HTTP Request Body	HTTP Response Body
List	GET <collection URL>	N/A	Resource* list
Get	GET <resource URL>	N/A	Resource*
Create	POST <collection URL>	Resource	Resource*
Update	PUT or PATCH <resource URL>	Resource	Resource*
Delete	DELETE <resource URL>	N/A	google.protobuf.Empty**

⚠ Qui la convenzione di API Google su questi metodi su collezioni e singoli elementi →

Riprendendo le convenzioni sopra citate, noi ora implementeremo **HTTP API** con Express come normalissime richieste HTTP parametriche (/dogs/:dogId); le richieste dovranno avere tutte **express.json()** come middleware, mentre le risposte useranno **res.json()** per mandare la risposta. Dovremo **sempre validare i parametri in input** (come dicevamo prima).

Per **testare queste HTTP API** (eccetto che le GET che sono facili da vedere direttamente nel browser), useremo **REST Client** (estensione di VSCode): creiamo un file con estensione file.http, scriveremo le HTTP requests che vogliamo testare (separandole con ###) e cliccheremo sul link "**Send Request**" che apparirà (deve comunque essere connesso al server con nodemon o node).

4) HTML & CSS

Non faremo nulla di **HTML & CSS** come teoria, basta andarsi a **rivedere le nozioni di Introduzione alle Applicazioni Web** perché in questo corso non vengono descritti. Useremo molto **Bootstrap**.

5) DOM + REACT

REACT è una delle librerie **front-end** più popolari per lo sviluppo di WebApp, che garantisce:

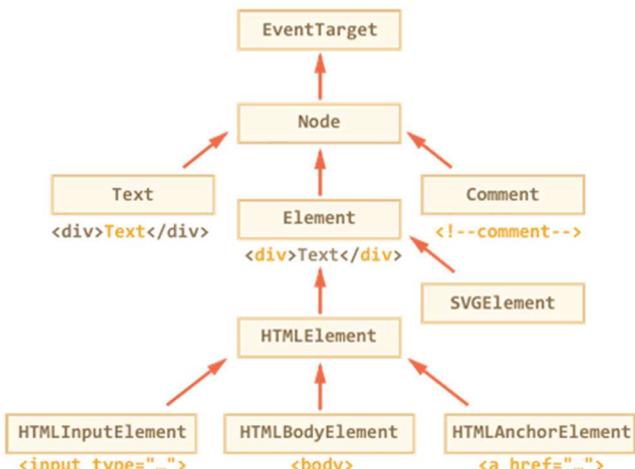
- **uniformità** → fornisce metodi DOM uniformi e una gerarchia esplicita
- **componenti di alto livello** (rispetto ai normali elementi HTML)
- **gestione automatica** degli **eventi** e degli **aggiornamenti**
- pattern e architetture standard + estensioni e plugin
- **gestione esplicita** dello **stato**

Prima di entrare nella sintassi di React, vediamo i **modelli ad oggetti del browser**:

- ❖ **BOM** (Browser Object Model) = fornisce interfaccia per **interagire col browser** (es. gestire URL, history...)
- ❖ **DOM** (Document Object Model) = rappresenta la struttura della pagina web come un **tree HTML** (rappresentazione interna della pagina web creata dal browser). Fornisce un **API JavaScript** per accedere ai metadati della pagina, modificare la struttura della pagina HTML, gli stili CSS e gestire eventi. **I suoi nodi:**
 - **Document** = root (nodo radice)
 - **Element** = tag HTML
 - **Attr** = attributo di un Element (tag HTML)
 - **Text** = testo di un Element
 - **Comment** = commento HTML (<!-- -->)
 - **DocumentType** = <!DOCTYPE> della pagina

JS nei browser usa un modello di programmazione **event-driven** (tutto è scatenato dal verificarsi di un evento). Gli eventi dipendono da:

- **elemento** che genera l'evento
- **tipo** di evento generato
- ⚠ **Eventi comuni** sono eventi UI (scrolling, resizing...), eventi di focus, eventi mouse, eventi tastiera...
- ⚠ Alcuni di questi eventi eseguono azioni predefinite, **bloccabili** con `event.preventDefault()`



Gli **oggetti principali del browser** sono:

- **window** = finestra del browser; contiene tutte le variabili globali JavaScript + accesso agli elementi del **BOM**. Sue proprietà utili sono:
 - **console** = per visualizzare messaggi di debug
 - **document** = per accedere al DOM
 - **history** = per navigare nella cronologia del browser
 - **location** = per fornire info sull'URL corrente (modificabile per ricaricare pagine)
 - **localStorage/sessionStorage** = per salvare dati localmente nel browser
- **document** = **DOM** della pagina caricata (accessibile con `window.document`)

DESIGN di REACT = **approccio dichiarativo** (non si manipola direttamente il **DOM**, ma si descrive come deve apparire l'interfaccia [**rendering**]) + **design funzionale** (tutto è suddiviso in **funzioni riutilizzabili**; **tutto viene ri-renderizzato da zero ad ogni cambiamento [di state e props]** grazie al **Virtual-DOM** [questo non è un processo lento perché React **calcola solo le differenze tra Virtual-DOM e V-DOM precedente, applicando poi queste modifiche minime al DOM reale**]; lo stato dell'app è gestito in modo esplicito).

Proprio perché è **“funzionale”**, l'interfaccia utente è funzione dello stato e delle proprietà (cioè i dati passati ai componenti), ovvero **UI = f(state, props)** [**alcuni** dipendono solo dalle **props** e **non dallo state**].

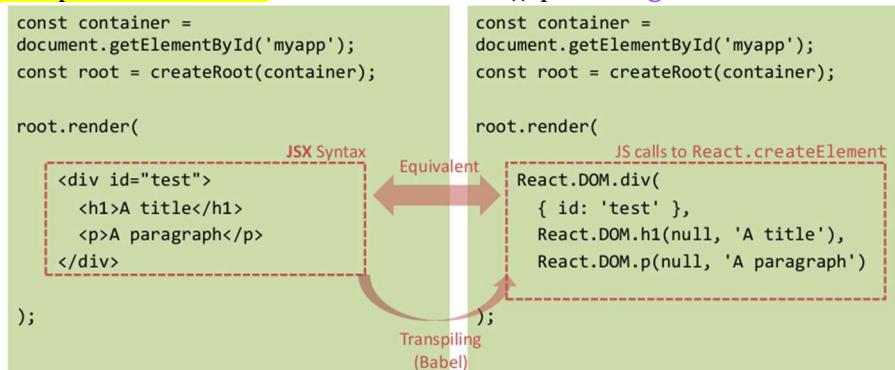
Inoltre c'è **immutabilità**:

- **props** → **immutabili** (non modificabili dal componente che le riceve). Passate dal componente padre al figlio, contengono dati di configurazione e possono contenere callbacks per interagire col padre
- **state** → non modificabili direttamente, ma solo con `useState` o `useEffect`. Locale al componente, se cambia il componente viene re-renderizzato
- **funzioni** devono essere **pure** (senza effetti collaterali)
- ⚠ Il vantaggio è **idempotenza** (re-rendering lo stesso componente porta allo stesso output) e > **prevedibilità**

React implementa il suo sistema di eventi **Synthetic Events** per ottimizzare la gestione degli eventi nei browser:

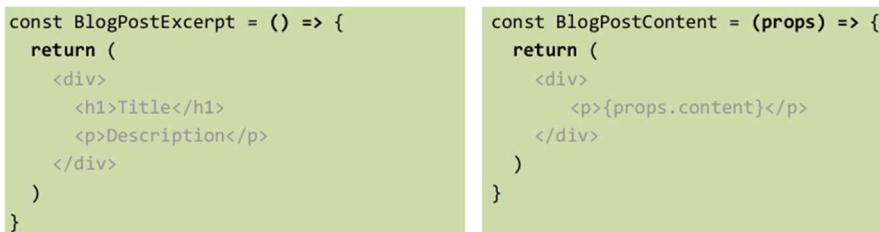
- 1 gestore di eventi per ogni componente (> prestazioni)
- normalizzazione cross-browser (eventi gestiti in modo uguale tra browser)
- decoupling dagli eventi nativi del DOM (eventi controllati prima che vengano propagati)

Per renderizzare un componente in React, si usa **createRoot()** per collegarlo ad un nodo del DOM:



Tutto quello che è in una web page è un componente (possono essere anche nested e possiamo riusarli); **ReactDOM.createRoot().render()** crea un componente e lo inserisce in un DOM container.

Un componente React può essere definito come una **funzione che restituisce un elemento JSX**:



React usa 2 tipi principali di componenti:

- **Presentational Components ("Dumb")** → generano solo elementi del DOM, non gestiscono lo stato dell'app, ma possono avere uno stato interno per scopi visivi
- **Container Components ("Smart")** → gestiscono lo stato dell'app, possono interagire col back-end e creano presentational components per mostrare i dati

React segue un **flusso di dati unidirezionale**, ovvero lo **state** viene passato ai componenti figli tramite **props**. Quando un componente figlio vuole modificare i dati del padre, usa una **callback** fornita tramite **props**. A ciò è legato il **"corollario"**, ovvero ogni stato è posseduto da **1 solo componente**, quindi deve essere spostato **più in alto** nella gerarchia dei componenti se deve essere condiviso tra più elementi sottostanti.

Ora vediamo come creare la nostra 1^ applicazione React: va importato React, vogliamo usare **JSX** (quindi richiesto **Babel**) e serve un server web per runnare.

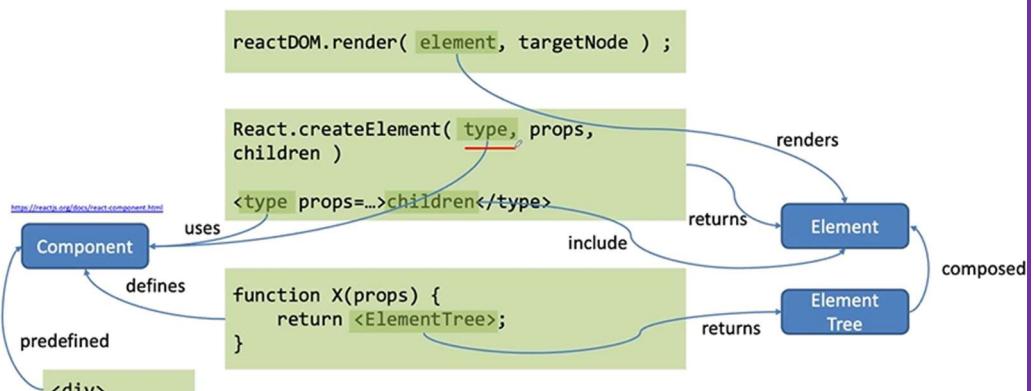
Per creare un progetto React, noi useremo **npm create vite@latest nameApp**; poi sceglieremo dal menù React, poi JavaScript. Da terminale faremo **cd nameApp** per spostarci nella directory del progetto. Poi da terminale faremo **npm run dev** e poi andremo a <http://localhost:5173>. Se volessimo aggiungere Bootstrap al nostro progetto React, dobbiamo fare entrambi **npm install react-bootstrap** e **npm install bootstrap**, e poi faremo gli **import** come in un normale file JS (noi scriveremo il codice dentro la funzione **App()** di **App.jsx** che viene creata in **src**).

Vediamo alcune veloci info su

ELEMENTI e JSX

Quando creiamo un elemento in React, diciamo il tipo dell'elemento (questo tipo è il componente). **L'elemento è l'istanza del componente.**

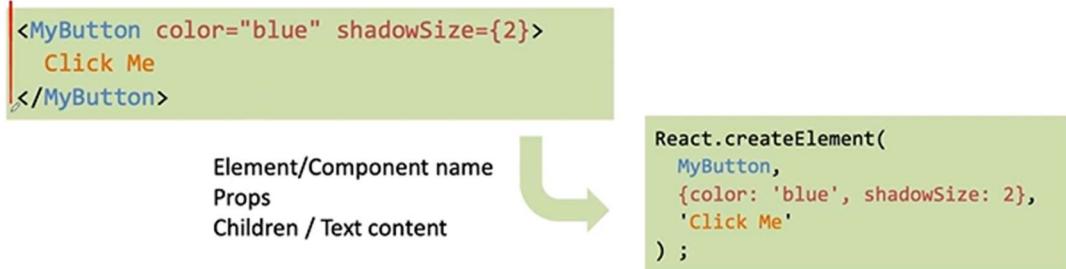
Tutti gli elementi fanno parte dell'Element Tree.



Gli **ELEMENTI** contengono solo **tipo**, **props** e **ogni loro figlio** (al suo interno); infatti si creano con `React.createElement(type, props, children)` dove **props** è un oggetto JS [gli attributi HTML vengono trascritti così come sono eccetto se hanno un nome che crea problemi con il linguaggio (es. `class` → `className`)]

⚠ Una convenzione è che gli **elementi del DOM** sono scritti in minuscolo, mentre i **componenti React** iniziano con la app maiuscola

Noi però non useremo la sintassi appena vista, ma useremo la sintassi di **JSX** (JavaScript Syntax Extension) [sx]:



Gli **elementi scritti in JSX** nel browser vengono **trasportati in HTML normale** (un file `jsx` ha estensione `file.jsx`). In JSX si può sempre usare **sintassi JS** e la sintassi generalmente è `const element = codice HTML` (è una sorta di **mix tra JS e HTML**). Tutti i figli di un componente si possono accedere con `props.children`

- ⚠ I valori `undefined`, `null` e i booleani (`true` e `false`) **non sono renderizzati**, ma solo letti a livello di logica
- ⚠ Non ci sono **commenti** in JSX, l'unico modo è `/* ... */`
- ⚠ Gli attributi di `style` diventano oggetti; come in JS c'è l'operatore di `spread (...)`

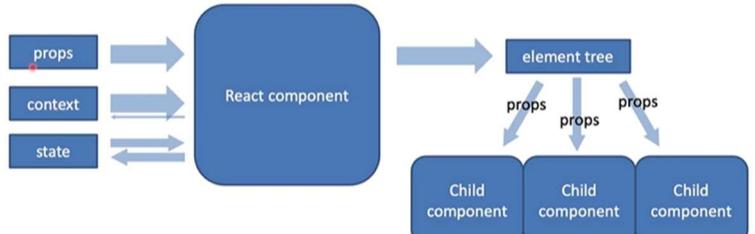
I **COMPONENTI** sono **come funzioni** usate per renderizzare un elemento della pagina; hanno una **props** (opzionale) come input, mentre hanno un **return** (obbligatorio) con la lista/albero degli elementi. Ogni componente si costruisce componendo altri componenti (e le loro `props`) [**modulare**; l'obiettivo è avere componenti minimi come base, con bassa complessità].

Ogni **elemento di una lista** deve avere un attributo denominato **“key”** (e ogni elemento della lista deve avere un valore univoco di questa key in quanto è usato internamente da React per capiere se qualcosa è stato cambiato nel rendering della lista; es. `<li key={number}>{number}` → la sintassi `<>...</>` è un **<React.Fragment>** che serve per contenere più elementi [container vuoto])

```
function NumberList(props) {  
  const numbers = props.numbers;  
  const listItems = numbers.map(  
    (number) => <li>{number}</li> );  
  return (<ul>{listItems}</ul>);  
}  
  
function App(props) {  
  const numbers = [1, 2, 3, 4, 5];  
  return <NumberList  
    numbers={numbers}/>;  
}
```

Vediamo meglio ora **COMPONENTI e STATI** (finché non agiamo sullo stato, non cambia la renderizzazione di un componente). I **Componenti** definiti come funzioni hanno il vantaggio di essere semplici, ma sono **funzioni pure** (**non** hanno **stato** al loro interno e **non** hanno **side-effects** [non devono influenzare altri componenti]); per questo sono stati introdotti gli **HOOK** (né vedremo 3: 1 per lo **stato** (`useState`), 1 per i **side-effects** (`useEffect`), e 1 per il **contesto** (`useContext`)) per aggirare questi problemi delle funzioni pure.

Ricorda struttura di un Componente React →



⚠ Lo **state** vive nel componente e mantiene i dati (non condivisibile tra componenti); per inviare un valore tra componenti, si usano le **props** (dati immutabili passati dai componenti padri ai figli). Il **context** è una sorta di global implicit props, che vengono automaticamente passate a tutti i componenti interessati (li vedremo avanti)

Gli **HOOK** sono facili da usare: va importato (`import {useState} from 'react'`), definito (`const [nomeStato, nomeFunzionePerSettareStato] = useState(valoreInizialeStato)`) e usato (uso lo stato come una **variabile** e la **funzione** come tale).

```
function WelcomeButton(props) {  
  let [english, setEnglish] =  
    useState(true);  
  
  return (<button>  
    {english ? 'Hello' : 'Ciao'}  
  </button>);  
}
```

- ⚠ Le modifiche allo stato **vanno fatte con la sua funzione** (es. `setHidden(false)`; oppure `setSteps((steps => (steps + 1))` [se il valore corrente dello stato influisce sul successivo])
- ⚠ Solitamente i cambi di stato sono dati da **eventi asincroni** (eventi del DOM [es. clicco bottone, mando form...]); se ci sono funzioni che devono chiamare la funzione dello stato (es. quando clicco un bottone, va chiamato il setter dello stato) posso usare la “**callback che chiama la funzione**” → `onClick={()=>setEnglish((eng)=>(!eng))}`)
- ⚠ Se ho un programma dove i figli devono cambiare lo stato del componente padre, devo fare il passaggio della proprietà dello stato tramite `props`

6) FORM, CONTEXT e REACT ROUTER

I **FORM** nativi HTML sono inconsistenti (diversi modi di gestire i valori e gli eventi); **REACT**, tramite **JSX**, crea un’interfaccia uniforme (eventi sintetici). Nei **FORM JSX**:

- `value` = contiene sempre il valore attuale del campo
- `defaultValue` = contiene il valore iniziale del campo

React fornisce un evento “**onChange**” **coerente**: si passa una funzione a `onChange` per reagire ai cambiamenti; `onChange` si attiva alla digitazione di ogni carattere e anche su radio-buttons, select-buttons e checkbox-buttons. Un **GESTORE DI EVENTI** riceve 1 oggetto **evento come parametro**; le **proprietà comuni** agli event handlers sono per esempio `event.target` (sorgente dell’evento); altri eventi possono avere proprietà specifiche in base al tipo.

EVENTI SINTETICI (**Syntethic Events**) → sono wrapper degli eventi del DOM; hanno gli stessi attributi di `DOMEvent`. Nel caso di un **form element**, abbiamo `target.value` (valore in input corrente) e `target.name` (nome dell’elemento in input). Vediamo un **elenco di eventi sintetici**:

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

Come definire gli EVENT HANDLERS? Usare un **arrow function** (`const handler = () => {...}`) o una **function expression** (`handler = function() {...}`); passare il nome della funzione come una **prop** (`return <div handler = {handler} />`) e poi **accedervi dalle props** (`return <button onClick={props.handler} />` oppure `{()=>props.handler()}`) oppure se ha parametri `{()=>props.handler(a,b)}`).

A chi appartiene lo **state**? Gli elementi dei form sono “**inherently stateful**” (mantengono un valore); i componenti React gestiscono lo **state** (noi useremo **controlled form component** che **mantengono nel loro stato il valore da mostrare nel form** [gli **uncontrolled** manipolano direttamente l’html risultante; **non hanno stato**]). Vediamo un **esempio** (form che ha uno state sul `name`, l’unico campo del form) →

```
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);
};
```

⚠ Bisogna sempre chiamare la `event.preventDefault()` nei form, per evitare la sottomissione del form (e quindi il ricaricamento della pagina); inoltre, anche nei form si possono validare i campi inseriti dall'utente (es. `validator.js`, simile a quello che abbiamo usato in Express)

La `useActionState` è un hook che semplifica la gestione dei forms (al posto di usare i componenti controllati). Rimuove il dover definire gli stati singoli sui componenti, ma ne definiamo 1 su tutto il form (non viene rirenderizzato ad ogni pressione di un pulsante). Crea uno **state** di un componente che verrà aggiornato quando una *form action* è invocata; ritorna una nuova **action** (da usare nel form), il più recente **form state** (che viene settato inizialmente allo state scelto) e un **loading state** opzionale (se si vuole usare quando la action sta avvenendo).

Come si vede dall'esempio a lato, la sua sintassi è `const [state, formAction, isPending] = useActionState(increment, 0):`

- **state** = nome del form state
- **formAction** = nome della funzione da usare nell'attributo **action** del form (da attivare quando sottometto il form)
- **isPending** = uno **stato** che dice se la form action è ancora pendente
- **increment** = la **action function** (ovvero cosa avviene quando il form viene sottomesso)
- **0** = **initial state** (l'uso di un initial state è raccomandato)

```
import { useActionState } from "react";

const increment = async (previousState, formData) => {
  return previousState + 1;
}

function SimpleForm() {
  const [formState, formAction, isPending] = useActionState(increment, 0);
  return (
    <form action={formAction}>
      {formState}
      <button type="submit">Increment</button>
    </form>
  )
}
```

La `actionFunction => async(prevState, formData) {...}` (che nell'esempio qui è la `increment`) ha **prevState** (il più recente form state) e **formData** (i dati mandati col form); questa funzione chiama automaticamente la `event.preventDefault()` (non lo dobbiamo fare noi a differenza di come visto prima); deve essere definita come asincrona (async).

Vediamo ora il **CONTESTO** (Context) in React, una sorta di “**props globali**” per evitare di passare le **props** a tutti i componenti che ne hanno bisogno. In React il flusso di informazione è unidirezionale e i componenti devono essere funzionali; se però avessimo 1 **props** da mandare a tutti i componenti, questo è **sconveniente**: quindi si introduce il contesto (una sorta di “*teletrasporto di props*”). Esempi di uso conveniente di contesto sono l'impostazione di **light/dark mode** della pagina web (info mandata a tutti i componenti della pagina web), l'impostazione della **lingua** oppure lo **stato loggato/non loggato** (es. l'icona dell'avatar dell'account).

Per usare il contesto bisogna avere context:

1. **DEFINITION** → `const ExContext = React.createContext(defaultValue)` [l'oggetto `Context` creato contiene il `Provider` e il `Consumer` a cui dopo accediamo]
2. **PROVIDER** → `<ExContext.Provider value=...>` (componente JSX che “inietta” il context value a tutti i nested components al suo interno [tutti i componenti da modificare (tutti i suoi Consumer) devono essere quindi **dentro** Provider; quando cambia il value del Provider, tutti i Consumer vengono rirenderizzati])
3. **CONSUMER** → `<ExContext.Consumer>` (fa il render di una funzione che riceve il value corrente del contesto come parametro) oppure `useContext(ExContext)` (usa un hook per accedere al valore corrente di contesto)

⚠ La convenzione dice di creare un file separato `esempioContext.js` con dentro la definizione del contesto e fare `export default` contesto

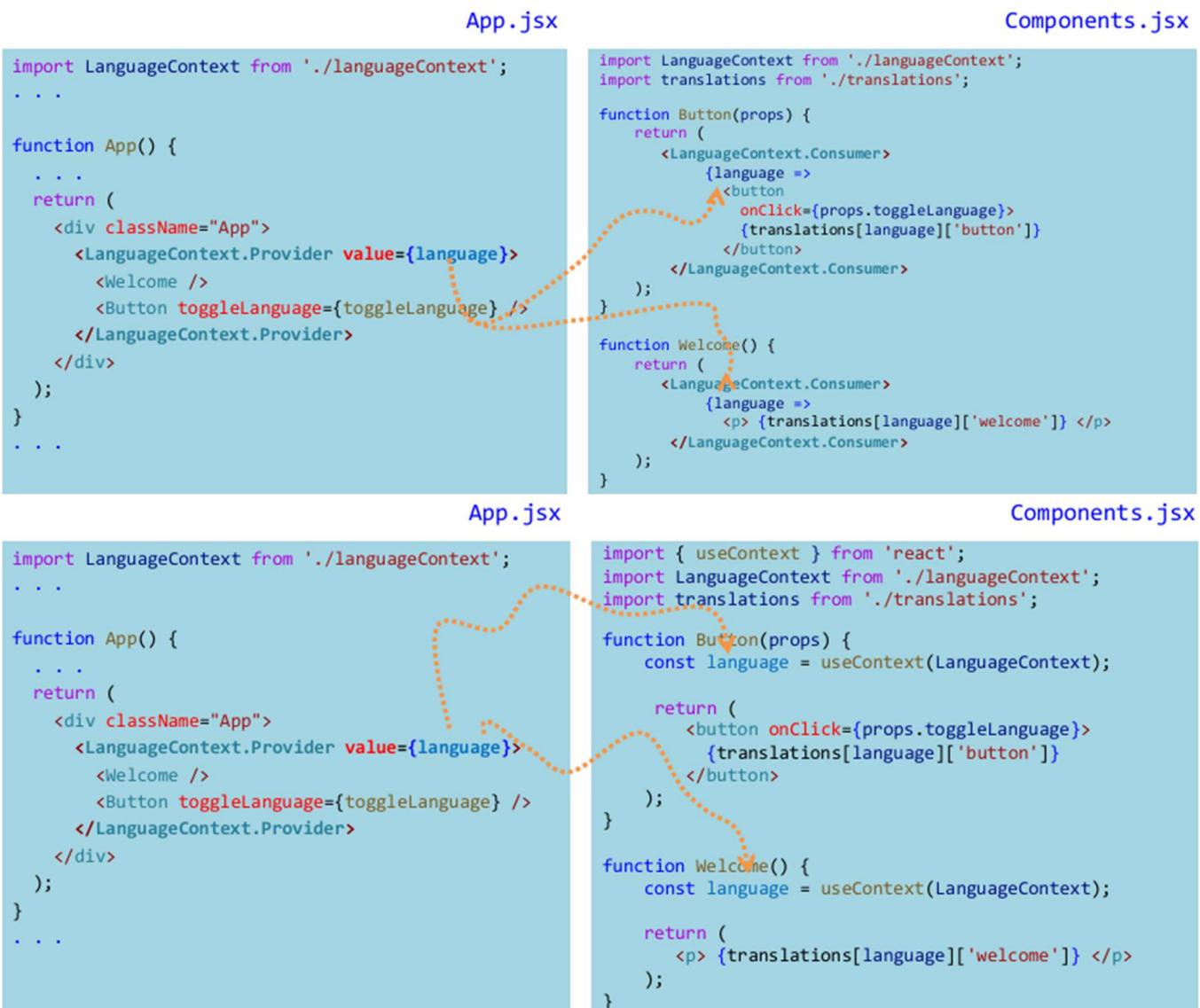
`languageContext.js`

```
import React from 'react';

const LanguageContext = React.createContext();

export default LanguageContext;
```

Vediamo un esempio delle 2 sintassi per il Consumer (`useContext` rimane la migliore, anche perché permette di accedere a più contesti):



Quando un Consumer deve aggiornare il context, deve passare dal `Provider`, il quale deve fornire una `callback` per fare l'aggiornamento (come abbiamo visto con le `props`). Questa callback può essere passata in 2 modi: come `prop` (ma sconveniente) oppure come `parte del context value` (dentro il `context value`)

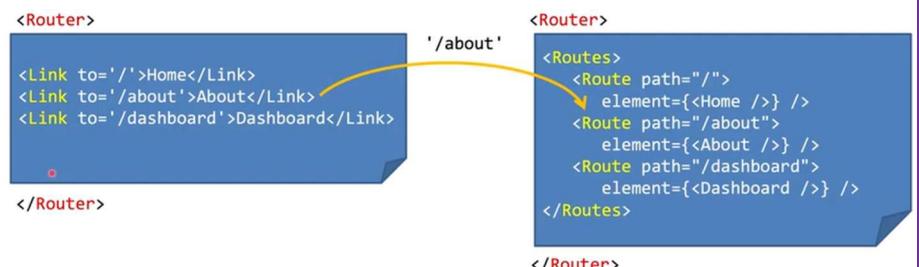
- ⚠ Ricorda che lo `state` è `parte del componente che contiene il Provider` (non del Provider né del context object)
- ⚠ Il contesto non va usato per pigrizia (non sostituisce le props, ma va usato solo nei casi corretti appena visti)

Parliamo ora di **REACT ROUTER** (Context) [*React as REST Client*], usato per pagine web con più pagine (non 1 sola come abbiamo fatto finora). Un Router JavaScript gestisce la **modifica della posizione dell'app (URL)** e determina **quali componenti React vadano renderizzate ad un certo URL** (mostra/nasconde i componenti a seconda delle routes correnti). Noi useremo come libreria **react-router** (`npm install react-router`); React-Router usa **normali componenti**:

- i link alle nuove pagine sono gestiti con `<Link>`, `<NavLink>` e `<Navigate>`
- per determinare quali componenti renderizzare si usa `<Route>` e `<Routes>`
- definisce hooks come `useNavigate`, `useParams`, `useSearchParams`

Tutta la app è wrapped in un container
`<Router>` →

⚠ In questo corso, noi inizializziamo i Router in modo **Dichiarativo (meno features)**



Nel modo Dichiаративо, ci sono 2 routers usabili:

- <BrowserRouter> → usa normali URL; noi useremo questo (`import { BrowserRouter } from 'react-router'`)
- <HashRouter> → usa # negli URL (non lo useremo)

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import { BrowserRouter } from 'react-router';
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </StrictMode>,
)
```

Quindi, andremo a modificare `main.jsx` (unica modifica del corso a questo file) dove importiamo il BrowserRouter e mettiamo tutto dentro il BrowserRouter.

```
<Routes>
  <Route path="/" element={<Home/>} />
  <Route path="/news" element={<NewsFeed/>} />
</Routes>
```

Le versioni alternative dei componenti a seconda dell'URL devono essere dentro un wrapper <Routes></Routes> ed ogni versione alternativa deve essere una <Route path="URL" element={Elementi da renderizzare} />. Se la URL fa il match con più path, viene preso il più "specifico"

⚠ Le <Route> si possono anche **nestare** 1 dentro l'altra e i loro path vengono concatenati come stringhe; i figli che fanno match in Route nested sono messi in un componente <Outlet> nell'albero di rendering del padre

Ci sono delle **Routes speciali** (sotto esempio):

- **Index** = <Route index element={<Home> /}> → route figlio che non ha un path (è la route figlio di default per una route padre); viene renderizzato nel <Outlet> del padre all'indirizzo del padre
- **Layout** = route senza path che viene sempre matchata (utile per il fare wrapping dei layout delle route figlie)
- **No Match** (`path="*"`) = prende il match se non lo prende nessuno (utile per le pagine "Not found")

```
function App() {
  return (
    <div>
      <h1>Basic Example</h1>
      <Routes>
        <Route path="/" element={<Layout />} />
        <Route index element={<Home />} />
        <Route path="about" element={<About />} />
        <Route path="dashboard" element={<Dashboard />} />
        <Route path="*" element={<NoMatch />} />
      </Routes>
    </div>
  );
}

function Layout() {
  return (
    <div>
      <nav>... main navigation menu ...</nav>
      <hr />
      <Outlet />
    </div>
  );
}

function Home() {
  return (
    <div>
      <h2>Home</h2>
    </div>
  );
}
```

Come ci spostiamo tra le Route? 3 opzioni principali (con esempi sotto):

- <Link to=> → crea un link consapevole del routing sotto (rimancia il classico tag <a> dei link html)
- `useNavigate()` → ritorna una funzione per triggerare la navigazione (utile negli event handlers)

```
function Home() {
  return (
    <div>
      <h1>Home</h1>
      <nav>
        <Link to="/">Home</Link>
        {" "}
        <Link to="about">About</Link>
      </nav>
    </div>
  );
}

function Invoices() {
  const navigate = useNavigate();
  return (
    <div>
      <NewInvoiceForm
        onSubmit={(event) => {
          const newInvoice = create(event.target);
          navigate(`/invoices/${newInvoice.id}`);
        }}
      />
    </div>
  );
}
```

- <NavLink> → come Link ma sa quando si è nella pagina navigata (usato con il CSS con la parola **active**)

Le Route possono essere **STATICHE** (stringhe) e **DINAMICHE** (con sintassi :parametro [come abbiamo già visto in precedenza]; posso usare `useParams()` per accedere a questi parametri [ritorna coppie chiavi-valori]).

Durante la navigazione, potrebbe essere utile **scambiare informazioni tra pagine**: si può usare la `useLocation()` e la `useNavigate()` per fare ciò [attenzione] però che l'oggetto passato tra le pagine (es. `userData` in questo esempio) viene **trasformato in stringa**, quindi con oggetti complessi non funziona].

Qui in React si possono avere anche “**query search parameters**” (es. quelle che negli URL sono `/products?sort=date&filter=valid`): si prendono con `let [params, setParams] = useSearchParams()` (`params` = oggetto URLSearchParams; `setParams` = rimpiazza l'intero set di parametri correnti e riceve oggetto key:value).

```
const navigate = useNavigate() ;
// go to URL and send information
navigate( url, {state: userData} ) ;

<Link to={url}
      state={userData} >
  ...
</Link>

const location = useLocation();
const userData = location.state;
```

7) REACT LIFE CYCLE & FETCH

CICLO DI VITA DEI COMPONENTI REACT → il **RENDERING** del componente è la fase più importante, ma ogni componente react affronta anche altre 3 fasi:

- **Mounting** = quando il componente viene creato e inserito nel DOM
- **Updating** = il componente viene rirenderizzato (es. al cambio di stato)
- **Unmounting** = il componente viene rimosso dal DOM

Questi 3 eventi sono a loro divisi in 2 fasi:

1. **RENDER** phase = tutto ciò visto finora (`useState`, `useContext`...); le funzioni non devono avere side effects
 2. **COMMIT** phase = aggiornamento del DOM e dopo di esso `useEffect` e `useLayoutEffect` (side effects)
- ⚠ **Side effect** = tutti i calcoli eseguiti da una funzione che non riguardano l'output della funzione stessa (es. usare `console.log` in una funzione, **data fetching**, usare timeouts ...)

Vediamo quindi l'hook `useEffect()` che **risolve il problema di usare side effect** nella fase di rendering in quando basta **mettere le operazioni di side effect all'interno di un blocco useEffect**. Come usarlo? Ha **2 parametri**: `useEffect(setup, [dependencies])`

- **setup** → **callback** di ciò deve fare il nostro side effect; la callback **non può essere definita come async**
- **dependencies** → **array opzionale**, dice quando eseguire la callback specificata e secondo quali dipendenze.
 - Se **non è specificato**, la callback del side effect viene chiamata ad ogni rendering
 - Se è **[]**, il side effect viene chiamato 1 volta subito dopo il rendering iniziale (fase di mounting)
 - Se è **[prop1, prop2, ..., state1, state2]**, viene chiamato anche 1 volta dopo il rendering iniziale, ma viene chiamato anche ogni volta che le dipendenze (in questo esempio le props o gli stati) cambiano valore

```
function QuickGate(props) {
  const [open, setOpen] = useState(false);

  useEffect(()=>{
    setTimeout(()=>setOpen(false), 500)
  }, [open]) ;

  const openMe = () => {
    setOpen(true);
  } ;

  return <div onClick={openMe}>
    {open ? <span>GO</span> : <span>STOP</span>}
  </div>;
}
```

⚠ Conviene mettere **tutti i valori del componente che cambiano nel tempo** e sono usati dall'effect (altrimenti non sarà sincronizzato con le modifiche nel complesso)

La **FETCH API** è un modo per **mandare e gestire richieste HTTP** (asincrone); consiste nel metodo `fetch()` che ha come parametri **URL della risorsa** e **oggetto** con i parametri della richiesta (opzionale); default request type **GET**.

Ritorna una **Promise** (che verrà **risolta quando la load è terminata**, questo perché **asincrona**); quando viene risolta, conterrà un oggetto **Response** (con cui potremo accedere ai dettagli della richiesta HTTP e il contenuto) [ricorda che **Promise** è una sorta di **Result** di Rust]

```
fetch('http://example.com/exams.json')
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    console.log(data);
  })
```

```
const response = await
  fetch('http://example.com/exams.json');

const data = await response.json();

console.log(data);
```

La Response ha anche delle proprietà utili:

- `Response.ok` (boolean): HTTP successful (code 200-299)
- `Response.status`, `Response.statusText`
- `Response.headers`: collection of HTTP headers of the response
- `Response.url`: final URL (potentially after HTTP redirects)
- `Response.body`: a readable stream of the body content

⚠ La **Promise** non va a buon fine (**rejected**) **solo per errori non-HTTP** (quindi di rete e non errori HTTP); quindi codici errati HTTP (es. 404 not found) torna comunque una Promise fullfilled. Vediamo esempi di **error handling**:

```
fetch(url)
  .then(response => {
    if (!response.ok) { throw Error(response.statusText) }
    let type = response.headers.get('Content-Type');
    if (type !== 'application/json') {
      //then() returns a rejected promise if something is thrown
      throw new TypeError(`Expected JSON, got ${type}`)
    }
    return response;
  })
  .then(response => {
    //...
  })
  .catch(err => console.log(err)) // either the throw value or other errors
```

Per **prendere la Promise** posso usare una sintassi del tipo `const fetchResponsePromise = fetch(resource [, init])` dove l'oggetto **init** ha delle proprietà come **method** (quale metodo HTTP [default = GET come detto prima]), **headers** (headers HTTP che voglio usare), **body** (body HTTP [es. in una POST]), mode, credentials, ... Es:

```
let objectToSend = {'title': 'Do homework' , 'urgent': true, 'private': false,
'shareWithIds': [3, 24, 58] };

fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(objectToSend), // Conversion in JSON format
})
.catch(function (error) {
  console.log('Failed to store data on server: ', error);
});
```

Per quanto riguarda la Response, se voglio accedervi posso usare i seguenti **metodi**, ma quando li uso “**consumo**” **il body della risposta** (non più accessibile); questi metodi inoltre ritorna una **Promise con il corpo della risposta** [quindi **avremo 2 Promise: 1 dopo la fetch e 1 per leggere il response.body**]:

- `response.text()`: as plain text (string)
- `response.json()`: as a JS object, by parsing the body as JSON
- `response.formData()`: as a FormData object
- `response.blob()`: as Blob (binary data with type)
- `response.arrayBuffer()`: as ArrayBuffer (low-level representation of binary data)

Se voglio delle **fetch sequenziali**:

```
const getFirstUserData = async () => {
  const response = await fetch('/users.json'); // get users list
  const users = await response.json(); // parse JSON

  const user = users[0]; // pick first user

  const userResponse = await fetch(`/users/${user.name}`); // get user data
  const userData = await userResponse.json(); // parse JSON

  return userData;
}
```

Se voglio fare delle **fetch parallele**:

```
// array of URLs
const urls = [url1, url2];

// Convert to an array of Promises
const promises = urls.map(url => fetch(url));
// Wait only for the fetch Promise

// Run all promises in parallel, wait for all
Promise.all(promises)
  .then(results => { // process according to the order needed by the app
    for (const res of results) res.text().then(t => console.log(t));
  })
  .catch(e => console.error(e))
```

⚠ La fetch API è già installata nel browser, non devo installare nulla

Riunendoci al discorso di prima, possiamo usare combinare **useState** e **useEffect** con il metodo **fetch**.

Esempio (da notare anche che come dicevamo prima, non posso usare `async` nella definizione della callback parametro di `useEffect`, perciò raggiro questa cosa e definisco dentro il corpo della callback una funzione `async` che userò all'interno della callback stessa):

```
useEffect( ()=>{
  const fetchFlipped = async () => {
    const response = await fetch('/flip?text=' + text);
    const responseBody = await response.json();
    setFlipped(responseBody.text);
  };
  fetchFlipped(text);
}, [text]);
```

```
const express = require('express');
const flip = require('flip-text');

const app = express();

app.get('/flip', (req, res) => {
  const text = req.query.text;
  const flipped = flip(text);
  res.json({text: flipped});
});

app.listen(3001, ()=>{console.log('running')})
```

Vediamo ora **ASPETTI AVANZATI** su **useEffect** e **fetch**:

- Se devo gestire delle **risposte lente** (es. da server lontani), aggiungo un nuovo stato `const [waiting, setWaiting] = useState(true)` in modo da avvisare l'utente che i dati stanno arrivando, ma non sono ancora arrivati; quando arriva la risposta, posso usare `setWaiting(false)` in modo da comunicare all'utente che l'attesa è terminata

Se invece sono in un form, posso usare il **built-in loading state**

- Si può usare una funzione `cleanup` per pulire tutto ciò che viene fatto dallo `useEffect` una volta che ha finito
- `useEffect` viene chiamata da React (in modalità sviluppo) **2 volte per ogni chiamata** [stress-test di React]

⚠ Ricorda che: **Application State** = preso dal server (salvato); **Presentation State** = non salvato in back-end

React chiama la:

- **REIDRATAZIONE** = quando prendiamo le informazioni per metterle nell'**Application State** (es. a lezione abbiamo usato la `fetch` per chiedere al server le informazioni di interesse e le mettiamo in un componente)
 - ⚠ Se ho **più browser connessi al server**, come fa uno di questi browser a sapere se un altro browser ha modificato le informazioni nel server? **[SINCRONIZZAZIONE]** In questo corso (non è la soluzione!!!), cerchiamo di refreshare il più possibile le informazioni richieste dal server [es. GET dopo ogni POST, PUT ...]

⚠ Per il progetto d'esame, fare attenzione al fatto che il professore può aprire il nostro sito da più browser e non deve rompersi niente!!!

- **DEIDRATAZIONE** = quando prendiamo l'Application State dall'applicazione React (modifica che parte dallo **stato locale** [es. un bottone che al click modifica qualcosa] ma che deve essere propagata al server API → questa situazione è **rischiosa se non si verifica la correttezza prima della propagazione**)

Quello che si deve fare è **REIDRATAZIONE + DISIDRATAZIONE**: aggiornare lo stato in parallelo in modo che l'utente veda che la sua modifica è stata effettuata, ma etichettando la modifica come **temporanea** (non definitiva ancora); poi **refresho tutto il componente se e quando il server completa l'update**.

Si può verificare un **infinite loop** quando:

- **dimentico l'array delle dipendenze**
- quando **uso un oggetto o un array come elemento nell'array delle dipendenze**
- ⚠ **Non usare oggetti/array come dipendenze, ma le loro proprietà** (quindi non **object**, ma **object.value**) nel caso di oggetti oppure usare uno stato aggiuntivo/proprietà utili dell'array

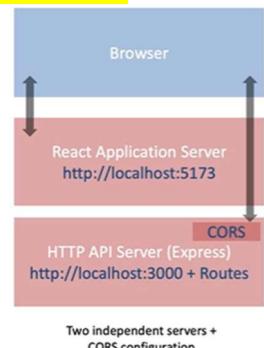
⚠ Tutti gli hooks vanno chiamati sempre al top di una funzione React; mai chiamato dentro le funzioni annidate (loop, if etc...), altrimenti non garantiamo l'ordine delle chiamate agli hook

8) INTERAZIONE CLIENT-SERVER in REACT

Per come abbiamo impostato il corso finora, abbiamo il **problema dei 2 server** (server di React lanciato da **npm run dev** + server Express per le route [REST API Server] → noi vogliamo che la fetch parli con il server Express). Il problema è che il server Express non capisce JSX (che invece è il linguaggio dell'altro server); dato che noi usiamo entrambi i server in locale, non abbiamo molti problemi, ma **se fossero su 2 macchine diverse?**

Questa situazione è un **problema, ma anche un vantaggio**: in questo modo il server React permette di modificare i componenti web e **rimane distaccato dall'interazione con i nostri dati e DB** (di cui si occupa il server Express). **Al contempo non possiamo chiamare da localhost:5001** (quello di React) l'altro localhost di Express.

Quindi, **come aggiriamo questa limitazione?** (in modo che rimanga solo il vantaggio di avere 2 server staccati, eliminando i limiti). L'idea è **manteniamo i server indipendenti** ma uso **CORS (Cross-Origin Resource Sharing)** in modo da **permettere di chiamare le routes anche su 2 localhost diversi** [immagine a dx]



Quindi noi **lanceremo sempre 2 server** (**npm run dev** per il server React + **node index.js** per il server Express [oppure nodemon index.js]) ma useremo **CORS** per avere **interazioni tra origin diverse** (configurare male CORS = problemi di sicurezza). **CORS lo configuriamo in Express** (non è da fare in React); **lo installiamo e poi:**

```
// index.mjs (node express server)
import cors from 'cors'; // npm install cors

//Enable All CORS Requests (for this server)
app.use(cors());
//Use ONLY for development, otherwise restrict domain
```

⚠ Questo è un esempio, nella realtà, a fini lavorativi, andranno **settati dei limiti su cosa CORS permette di scambiare** (mentre nell'esempio qui sopra, permette di scambiare tutto tra origin diverse da qualunque origin)

Esempio con anche React affianco:

API.mjs in the React Application

```
const APIURL=new URL('http://localhost:3000');

async function getCourses() {
  return fetch(new URL('/courses', APIURL))
    .then((response)=>{
      if(response.ok) {
        return response.json();
      } else {
        throw response.statusText;
      }
    })
    .catch((error)=>{
      throw error;
    });
}
```

Called in useEffect()

index.mjs for the API Server

```
import express from 'express';
import cors from 'cors';

const app = express();
const port = 3000;
app.use(cors());

app.get('/courses', (req, res) => {
  dao.listCourses()
    .then((courses) => res.json(courses))
    .catch((dbErrorObj)=>
      res.status(503)
      .json(dbErrorObj));
});

app.listen(port, () => console.log(`Example app listening at http://localhost:${port}`));
```

Calls dao.mjs

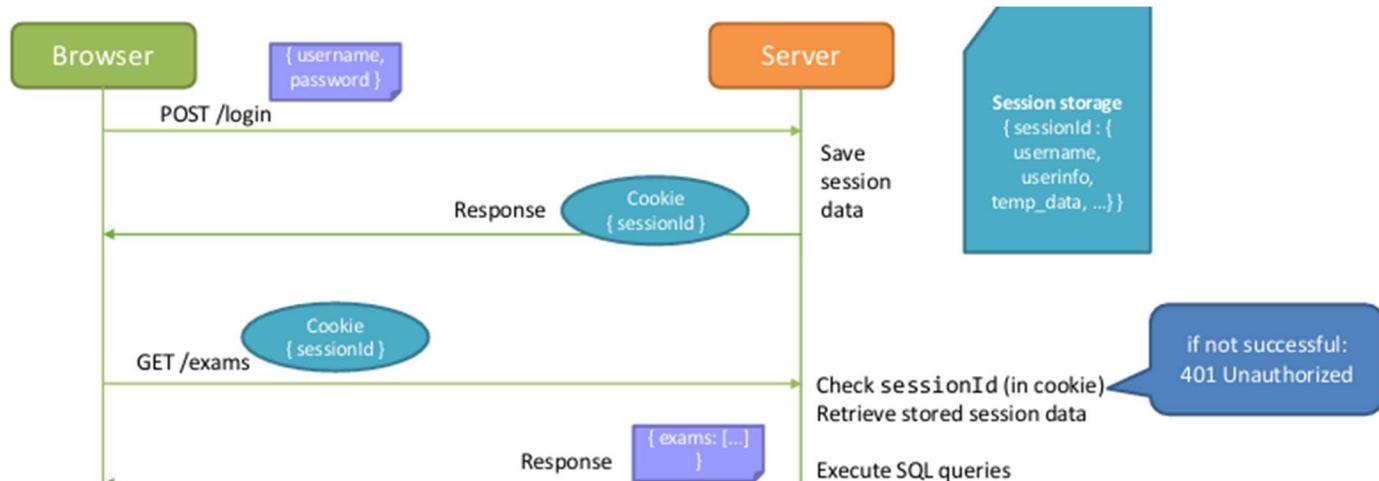
9) AUTENTICAZIONE

AUTENTICAZIONE = verificare l'identità di chi accede (di solito fatto con credenziali: username, password);
AUTORIZZAZIONE = decidere se un certo utente ha i permessi per accedere ad una risorsa/servizio [noi le usiamo in modo congiunto]

HTTP è **stateless** (ogni richiesta è indipendente e deve essere “self-contained”), ma un webapp potrebbe voler mantenere delle informazioni tra iterazioni diverse → **SESSIONE** = dati temporanei e interattivi scambiati tra 2 parti (devices) [prevede 1 o più messaggi verso ogni parte]. Dopo l'autenticazione, il client riceve un **session ID** dal server (permette al server di riconoscere delle richieste HTTP come “autenticate”); questo viene memorizzato dal client e deve essere mandato dal client ad ogni richiesta che fa parte della sessione (questo per riconoscerle come “autenticate”); non deve essere un dato sensibile e solitamente viene messo/mandato come **COOKIES**.

Un **cookie** è una porzione dell'informazione memorizzata nel browser (nel “cookie storage”) gestita in modo automatico dal browser. Viene automaticamente mandato dal browser ai servers quando avviene una richiesta allo stesso **domain** e **path**. I suoi attributi (di solito settati dal server) sono **name**, **value**, **secure** (mandato solo con HTTPS), **httpOnly** (inaccessibile dal codice JS che runna nel browser) e **expiration date**.

Noi faremo **autenticazione (auth)** “**session-based**”:



⚠ Usa sempre **httpOnly** cookies e **secure** cookies con HTTPS

⚠ Usa sempre codice già scritto per autenticazione, non inventarne di nuovo perché fallirà!

In REACT (**sul client**) come scriviamo un **Login Form**? 2 opzioni: lo creo come React component con local state (**useState**) oppure posso crearlo come React component con **useActionState** (come ogni form fatto finora).

Sul server invece? Noi useremo un **middleware** flessibile e modulare per l'autenticazione, ovvero **PASSPORTJS** (**npm i passport**) [supporta diverse strategie di auth, usa best practices e si adatta a diversi tipi di DB]

Ma prima di usare passport, va **modificata la nostra express app per usarlo** con questi passi:

1. **LocaStrategy** (**npm i passport-local**) → dice con cosa avviene il login (noi username e password) tramite la funzione **verify (username, password, callback)** che controlla le credenziali: se ok, la callback ritorna l'utente come 2° parametro; se errore, al posto dell'utente troviamo false
⚠ I campi di verify vanno chiamati proprio **username** e **password**, in quanto vengono estratti direttamente dall'HTTP body della richiesta

La password non va salvata “plain” ma va fatta passare dentro **scrypt** (una funzione “**secure**” di **password hashing**; inclusa in Node dentro il modulo crypto). Le funzioni usate sono 2:

- **crypto.scrypt (password, salt, keylen, function(err, hashedPassword))** dove il “salt” deve essere random e di almeno 16 bytes [**const salt = crypto.randomBytes(16)**]
- **crypto.timingSafeEqual(storedPassword, hashedPassword)** [confronta le password, ma devono essere state hashate con lo stesso salt]

2. **Express-session** → è il **middleware per creare la sessione da usare con PassportJS** (utile solo per sviluppo, non per produzione)

```

import session from 'express-session';

// enable sessions in Express
app.use(session({
    // set up here express-session
    secret: "a secret phrase of your choice",
    resave: false,
    saveUninitialized: false,
}));

// init Passport to use sessions
app.use(passport.authenticate('session'));

```

3. Decidere e configurare **quali user info sono associate ad una specifica sessione**: lo facciamo con **serializeUser** (sceglie quali user info memorizzare nella sessione) e **deserializeUser** (le stesse user info che sono state serializzate prima, saranno restored quando la sessione è autenticata da questa funzione)

```

passport.serializeUser((user, cb) => {
    cb(null, {id: user.id, email: user.username, name: user.name});
});

passport.deserializeUser((user, cb) => {
    return cb(null, user);
});

```

Dopo tutto ciò, dobbiamo usare **passport.authenticate('local')** per fare login con Passport:

```

app.post('/api/login', passport.authenticate('local'), (req,res) => {

    // This function is called if authentication is successful.
    // req.user contains the authenticated user.
    res.json(req.user.username);

});

```

Quali informazioni ritornare a React (client)? Il suggerimento è salvare le info in a **Context** (o **State**) e chiedere al server quando ci servono queste informazioni (in un **useEffect** tramite l'**API**). Dopo il login? **Alcune routes del server possono essere protette**, ovvero rispondere solo ad utenti **autenticati** o **con un certo privilegio**; lo si fa in 2 modi:

- modo **facile** → fare check usando **req.isAuthenticated()** all'inizio di ogni callback body in ogni route da proteggere
- modo con **middleware**

Per mandare i cookies su stesso dominio ma su porte diverse, dobbiamo usare CORS e mettere in **corsOptions** anche **credentials: true**, mentre nel client va messo in tutte le fetch requests all'API protetta questa sintassi:

```

const response = await fetch(SERVER_URL + '/api/exams', {
    credentials: 'include',
});

```

Per il **Logout** possiamo usare **req.logout(() => {res.end();})**