Note Web Application

Brendon Mendicino

June 6, 2023

Contents

1 Introduction

JS is backward compatible, to be able to use the previous features is use the directive:

```
"use strict";
```

JS has primitive types and non-primitive types, JS is also and strongly typed language, the primitive types are: string, number, boolean, null, undefined. The non-primitive are the objects, which can be: array, function, user-defined.

The all possible false values in JS: `0, -0, NaN, undefined, null, ''`, in JS there are two main comparison operators:

```
a == b    // equal, convert types and compare
a === b   // strict equal, inhibits automatic type conversion
```

In JS you can create variable with:

```
// modern
let a = 10;     // can be changed
const b = 'a'; // constant

// old
var k = 9;
j = 30;
```

The difference between null and undefined, is that variable with null they old a value which is null, on the other way if a variable is declared and nothing is associated with it the value olds by default undefined.

A scope is defined by a **block**, which is created with ...

There two kinds of `foreach` in JS, using `in` allows iterating over objects, while `of` allows iterating over iterable objects:

```
for (let a in object) {
  ...
}

for (let b of iterable) {
  ...
}
```

Using arrays:

```
let a = [1, 2, 'ok', false];
let b = Array.of(1, 2, true);
a.push(5);      // append an element
b.unshift(2);   // insert at the beginning

let copy = Array.from(a);  // shallow copy, it does not deep copy
```

The **destructuring assignment** can be done, it extracts the values from the mast left-hand side:

```
let [x, y] = [1, 2];
[x, y] = [y, x]     // swap
```

The **spread operator** (...) expands on iterable object into it's values:

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

const a = [1, 2];
const b = [0, ...a, 3]; // [0, 1, 2, 3]
```

Spreading can be from the left or from the right, usually the spread operator is used for copying array:

```
const a = [1, 2];
const b = [...a];
```

A **string** is JS is an immutable type (like python) encoded in Unicode. The **template literals** can be done with the **tick** operator ‘ ‘ (expression like Kotlin):

```
let name = 'Bre';
let sur = 'Mend';
// Template literal
let fullName = `${name} ${sur}`;
```

1.1 Objects

JS is **prototype based language**, which means that there are no declarations of classes. In JS property names must be strings and can be modified, the value of the property can be any other type of type or object. To create and object in JS you use curly braces and the defined properties:

```
const movie = {
  title: 'Inception',
  genre: 'sci-fi',
  duration: 180
}

console.log(movie)
console.log(movie['title'])
console.log(movie.title)
```

It is also possible to add a property by simple assigning a new name to a type, it is also possible to delete a property with the keyword `delete`. There are two helper functions:

- `Object.key(object)`: return only the key;
- `Object.entries(object)`: return an array with the key and value;

To copy an object it is possible to use:

```
const copied = Object.assign({}, original)
const withSpread = {...original}    // it also possbible to use the spread
    operator

```

```
// assign can also be used to merge objects
const merged = Object.assign({}, copied, {something: 'test'})
```

1.2 Functions

In JS functions are objects, so it is possible to assign a function to a property or use it in a parameter in another function. There three possible ways to define a function:

```
// 1. Function
function do(a, b = 1) {
  ...
}

// parameters can also hava a deafult value
function some(par1, par2, ...variable) { // ... is the 'rest' operator, like
    varargs, rest parameters can be iterated
  ...
}

// 2. Function Expression
const fn = function(params) { }

// 3. Arrow Function
const func = (params) => { }
```

In JS **Closure** can be created, with closure it is possible to use parameters of the scope where the function is defined, even if that scope does not exist any more.

```
function greeter(name) {
  const myname = name;

  const hello = () => {
    return "Hello " + myname;
  }

  return hello;
}

const helloTest = greeter('test');

console.log(helloTest());   // 'Hello test'
```

To create an object there are also **constructor functions**:

```
function Movie(title, director, duration) {
  this.title = title;
  this.director = director;
  this.duration = duration;
  this.isLong = () => this.duration > 120;
}

```

```
cosnt movie = new Movie('Inception', 'Nolan', 180);
console.log(movie.isLong);  // true
```

1.3 Dates

We use `dayjs()` objects in JS to build a data, it is an external library. The return of `dayjs()` fetches the time from the locale time, other than that it can create a data from ISO8601 strings, 8 digit dates, etc. To install this library: `$ npm install dayjs`. The string value of the standard format is in ISO9601 in UTC time, it's important to remember that the days and the month inside the object start *counting from 0*. Other than that the library has some methods to compare different `dayjs` objects, also by choosing the level of *granularity* (year, month, day, ...).

```
const date = dayjs('2023-03-15');
const now = dayjs();

now.isAfter(date, 'day');  // comparing 'now' and 'date' by day
```

1.4 Asynchronous Programming

In JS when passing functions to other functions it's called a **callback**, this functions can be *synchronous* or *asynchronous*.

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

funtion createQuoute(quote, callback) {
  const myQuote = `Like I always say, ${quote}`;
  callback(quote);
}

createQuote('sium', logQuote);
```

In order to have functional features in language there some need properties:

- *functions as first class citizen*;
- *higher-order functions*;
- *function composition*;
- *call chaining*;

In JS arrays have functional methods, for example:

```
a.forEach(item => ...);   // action on every element of the array
a.every(x => x > 10);     // return true if all elements satisfy the condition,
    false otherwise
a.some(x => x < 10);      // return true if at least one element satisfy the
    condition
a.map(x => `${x}`);       // return a new array with every element mapped to a new
     one
a.filter(x => x === 0);   // return a new array with all elements that satisfy the
     condition
a.reduce((x, y) => x + y, 0);   // return a reduced value
```

Even though JS is executed on a single thread it is possible to create concurrent code, for example a function that allows to excute a callback after a certain amount of time is the `setTimeout()` function:

```
const f = (task) => {
  // do something
};

setTimeout(f, 2000, task);
```

This is possible because JS runs in the **Event Loop**, which periodically checks if there are some part of the code that needs to be executed.

There is a function that allows asynchronous callback after a timeout:

```
const onesec = setTimeOut(() \implies {
  console.log('1 second has passed');
}, 1000);
```

There is also the `setInterval()` function that periodically runs:

```
const period = setInterval(() => {}, 2000);
clearInterval(period);
```

1.4.1 Database Access (SQLite)

The module for `sqlite3` allows calling sql queries via his APIs, first there needs to be an open with the database, to open a connection use:

```
const sqlite = require('sqlite3');

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

Example of query:

```
let result = [];
let sql = "SELECT * FROM course LEFT JOIN score ON course.code=score.coursecode";
db.all(sql, (err, row) => {
  if (err) throw err;
  for (let row of rows
});
```

The problem with execution this queries is that they are *all asynchronous*, and they can cause race conditions. The solution to this problem are `Promise`, which helps simplyfing asynchronous programming. A `Promise` handles a `resolve` and a `reject` which needs to be called when the callback fails or succedes. The values passed to `resolve` can be accessed by in the `then` method, which gets called when the `Promise` is completed.

```
function waitPromise(duration) {
  return new Promise((resolve, reject) => {
    if (duration < 0) {
      reject(new Error('...'));
    } else {
      setTimeout(resolve, duration);
    }
  }
}

waitPromise(1000).then((result) => {
  colsole.log('Success :', result);
}).catch((error) => {
  colsole.log('Error :', error);
});
```

A promise has 3 main methods: `then`, `catch`, `finally`, which are similar behaviour to the try catch block in Java. Promises can also work concurrently with `Promise.all()` or `Promise.race()`.

1.4.2 Await/Async

The keywords `async` and `await` allows to convert pieces of code to a `Promise`:

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
}

asyncCall();  // this returns is a promise
```

In fact a function marked as `async` returns a promise.

This method can be combined with the database queries:

```
async function main() {
}

main();
```

2 HTML/CSS

CSS has different mesuraments units:

- `em`: unit size relative to the font size present in the current element;
- `rem`: unit relative to the font size of the root element;
- `vw`: relative to 1% of the width of the viewport;
- `vh`: relative to 1% of the height of the viewport;

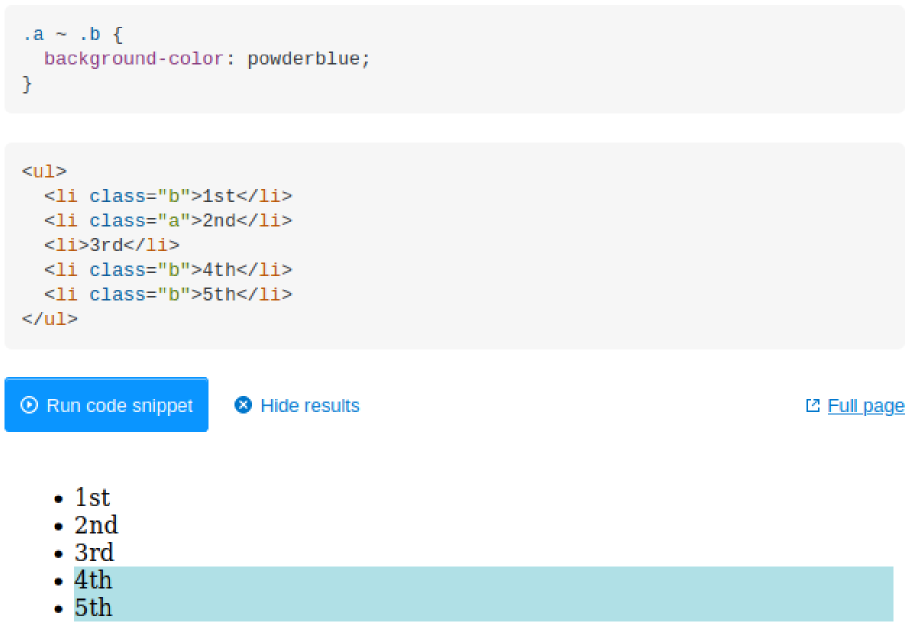
CSS has aldo **pseudo selector** which represent changes based on the state of an element.

```
a:visited { color: green; }
```

In CSS there 4 position schemes: **static**, **relative**, **absolute**, **fixe**.

In CSS the flex schema allows for direct control over the element of the page, it allows modifying: direction, sizes, alignment, position, spacing, ...

In CSS the ˜ selector is called **subsequent sibling combinator**, the element represented by the first sequence precedes (not necessarily immediately) the element presented by the second one.



```
.a ~ .b {
  background-color: powderblue;
}
```

```
<ul>
  <li class="b">1st</li>
  <li class="a">2nd</li>
  <li>3rd</li>
  <li class="b">4th</li>
  <li class="b">5th</li>
</ul>
```

```
• 1st
• 2nd
• 3rd
• 4th
• 5th
```

Figure 1: Subsequent Sibling Operator

2.1 Responsive

It's possible to achieve a responsive layout by using `media query`:

```
@media(min-width:900) { }
```

3 JS inside HTML

The perferred way to include javascript code inside the html is:

```
<script async src="script.js"></script>
// or better
<script defer src="script.js"></script>
```

Where does the code run?

The main objects of the browser:

- **DOM**: Document Object Model
- **BOM**: Browser Object Model, non-standard

The BOM has a `window` object, which contains: `console, document, history, location, localStorage, sessionStorage`.

The DOM is

The DOM can be accessed like a squence of Nodes. To find a node there are vaarius methods, like:

- `document.getElementById(value)`
- `document.getElementsByTagName(value)`
- `document.getElementsByClassName(value)`
- `document.querySelector(css)`
- `document.querySelectorAll(css)`

From each node there are many mothod to access all the neigbhor nodes.

3.1 Event Handling

...

4 React

React is a framework that allows DOM manipulation with a level of abstractions, and while using it won't be necessary to touch the DOM directly.

React has a functional approach, which allows bulding a web page in a declarative approach. On any change that is acted on a component all the other components are rerendered. Fot this reason React has a **virtual DOM** which is built on top of the DOM which will eventually push his changes to the original DOM.

The basic information that are shared between components are the `state` and `props` (properties), which are passed to functions inside the component.

On re-rendering when the virtual DOM is stabilized, the difference between the virtual DOM and the DOM are computed and only then the changes are moved on the actual DOM, this is why this re-rendering is not so heavy.

There are event that are normalized across the browser, this are called **synthetic events**.

If we want to write a minimal React application:

```
const container = document.getElementById('root');

const root = createRoot(container);
root.render(<h1>Hello, world!</h1>);
```

React uses `jsx` that are translated into *react elements*, this `jsx` will be then be translated to plain javascript by React. To define a component in React we do:

```
const BlogPostExcerpt = (props) => {
  return (
    <div>
      <h1>Title</h1>
      <p>{props.content}</p>
    </div>
  )
}
```

There are two types of component:

- **presentation component** don't manage the state;
- **container component** manages the state of all his children;

`props` can only be passed from a parent to his children, if the changes shoul be performed from the a children to a parent then, a callback needs to be passed.

$$\boxed{\text{React flow: view} \implies \text{actions} \implies \text{state} \implies \text{view} \implies \dots}$$

- A **state** is always **owned be one compoment**.
- Changing state on a childred should not affect the state of a parent.

To create a React application

```
$ npm create vite@latest my-app
// Choose React, javascript + SWC
$ cd my-app
$ npm install
$ npm run dev
```

In `jsx` the attributes of a component (`props`) are translated into js objects

```
color="red" -> {color: "red"}
tone={2} -> {tone: 2}
// it's possible compute exressions
color={active ? 'green' : 'red'}
```

Some attributes in HTML do not a value like `selected disabled`, but in `jsx` they can have a value: `selected={true}`

In React when using a list it's mandatory to use give a key to each element of the list (the key is unique in the list, not globally), this is done because React internally can recognise if the list has changed, if this is not done there are undefined behaviur in the page.

```
<li key={todo.id}>{todo.text}</li>
```

When returning a gruop of components they should always be packed in a root component, when returing a group of components there is *React Fragment* which makes it more easy.

4.1 State

Ad we have seen using pure functions to create Components in react, this is very simple but in react the various components need a state, to use this feature **Hooks** have been proposed which intruduce a way to indroduce a state. One of them is `useState`. While **props** are passed from father to child, **state** is private to the component and hold his data, the **context** is a global variabl eavailable to every component.

To create a hook, we use the `useState` and it gives back the current value and a callback to cahnge that satate.

```
function ShortText(props) {
  const [hidden, setHidden] = useState(true);
  return (
    <span>
      {hidden ? "Hidden" : "Not hidden"}
      <a onClick={() => setHidden(hidden => !hidden)}>hide</a>
    </span>
  );
}
```

All the modification of the state need to be done through `setVariable`. The value can be set by passing a value or with a function.

Can a child mutate the state of the parent? This simplest way is to pass a the `setVaraible` to the child inside a callback through the `props`

```
function App() {
  const [hidden, setHidded] = useState(tru);

  function hide() {
    setHidden(hidden => !hidden);
  }

  return (
    <>
      <span>
        {hidden ? "Hidden" : "Not hidden"}
      </span>
      <Change hide={hide} />
    </>
  );
}

function Change(props) {
  return (
    <a onClick={() => props.hide()}>Hide</a>
  );
}
```

4.2 Forms

In HTML form are inconsistent, because of that React tried to formalize the input and output of forms, in fact everything that can accept an input in React the element has a `value`, which then will be translated to HTML by the React engine.

In `jsx` to subscribe to the input event there is an attribute called `onChange`. There are events that wrap all the events that you can catch in the DOM, that are called **synthetic events**.

Forms are inherently statefull, there are specifics way to bind the input with the event of the form. In React there are **controlled** and **uncontrolled** forms component.

When submitting a form the `onSubit` needs to be used.

Also to update the state when it data is an array the `setValue` needs to be used.

```
const [list, setList] = useState(array);

setList(oldList => {
  return oldList.map(item => {
    if (item.id === id) {
      rerturn { id: item.id, val: 'NewVal' };
    } else {
      return item;
    }
  });
});
```

4.3 Context

When passing information to the child we need to pass it to the props, every time without forgetting it, by using the context we make an information available to the whole application. When we create a context we need to define who is the **provider** (`React.createContext()`) and who is the **consumer**, the context can contain only one object.

Example 4.1

We can import the languege context to change the language in the whole application

```
export const LanguageContext = React.createContext();

/**/
import languageContext from './languageContext';

<LangugeContext.Provider value={language}>
  <Button toggle={toggle} />
</LangugeContext.Provider>

return Button(props) {
  const language = useContext(LanguageContext);

  return (
    ...
  );
}
```

Consumer

```
<ExContext.Consumer>
{value => /* render something based on the context value */ }
</ExContext.Consumer>

```

5 Routing

In react we need a library to handle routing, this app can read/write the url input ad be able to load the correct React component, to install

```
npm install react-router-dom
```

To use the router we need to use `<Link>` component and not `<a>`

```
<Link to'/'>Home</Link>
<Link to'/about'>About</Link>

<Routes>
  <Route path='/' element={<Home />} />
  <Route path='/about' element={<About />} />
</Routes>
```

There are two ways of creating a router in react, by using *functions* or *components*. We will use `BrowserRouter`, `HashRouter` also exitst but we won't use it.

We can have:

- **static**: `/user`
- **dynamic**: `/:userId`
- **star**: `/*`

React will always match to most specific path, and it's also possible to *nest routes*.

There are some special routes like `paht='/'` which is identified as the home it can be identified with `index` attribute, and the `paht='*'` which is the `<Layout>` route that will catch everithing, by doing that it is possible to disply error pages to users in case they mispell the URL. To navigate to a new page we use `<Link>` or `useNavigation()`, `<NavLink>` allows us to know if a link is active or not.

To use dynamic routes

```
<Route paht='/user/:id' elemnt={<User />} />

return User(props) {
  const {id} = useParams();

  return (...);
}
```

We can also pass information when navigating

```
const navigate = useNavigation();
navigate(url, {state: userData});

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

```

```
/* ... */
const location = useLocation();
const userdate = location.state;
```

The state is serialized when it's sent to the route, it's important to remember that only strings can be serialized properly when passing this state, so when usign more complex object we need a way to modify them to able be serialized properly.

6 Server Side

6.1 Express

The goal is to have e database in the backend that is able to talk with the React application. **Express** is one of the most used modules in Node to handle http requests. One usefull modules is `nodemon` which allows recompilation of the server when there are changes.

```
$ npm install -g nodemon
$ nodemon index.js
```

Express has three basic sections

- create the application server

  ```
  const express = require('express');
  const app = express();

  ```

- define the behaviour of the server

  ```
  app.get('/', (req, res) => res.send('Hello world!'));

  ```

- using the app created to launch a server

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

  ```

When handling the the different http methods we use `res` and `req` which are the **response** sent to the server and **request** that will be sent back to the client.

Express has **Middlewares** (between the request and the respones), which is a function called for every request or for specific ones. To create a middleware

```
express.static(root, [options]);
// root: static content like css, html, ...
```

- GET (`localhost/some?user=brendon`): `req.query`, `req.query.user`;
- POST/PUT normal body: `req.body`, `req.body.user`.
  Middleware: `express.urlencoded()`;
- POST/PUT json body: `req.body`, `req.body.user`.
  Middleware: `express.json()`;

To enable **loggin** we user `morgan`

```
const morgan = require('morgan');
app.use(morgan...);
```

there are also middlewares for html validation.

6.2 HTTP API

Most of the API written today use json as format, in JavaScript there are some methods to coverert or parse json from/to a string (`JSON.stringify`, `JSON.parse`).

```
{
  "something": "some",
  "number": 1,
  "listOfNumbers": [
    1, 2, 3
  ],
  "object": {
    "some": "some",
    "id": 10
  }
}
```

A standard is to use URIs, the way to define one is to use:

- use nouns (not verbs)
-

```
http://api.polito.it/students/s123456
```

We represent relationships with

```
/collection/identifier/relationships
```

7 Fetch API

When have 3 problems when developing a React application and an API server:

1. How can the two apps communicate?
2. How can the two apps live in one server? (The two apps cannot have tow different addresses)
3. How to handle the changes that are happening outside a component?

The first problem is solved with a **Fetch API**. The goal is to load data asynchronously, the reason is that we don't know when the response will arrive, the API is a generic browser function, that will return a `Promise` that will resolve into an object that contains the response body.

```
const res = await fetch('https://example.com/exams.json');
const data = await res.json();
console.log(data);
```

Also calling `res.json()` return a `Promise`, the reason is that the body may be very long, and we don't want to block the program if that's the case. When sending a fetch request, we can add an optional object `fetch(resource [, init])`, the init object has:

- method
- headers
- body
- mode
- credentials
- signal

```
fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.strigify(objectToSend)
})
  .catch((err) => {
    console.log(err)
  })
```

The methods that can be called on a response can only be called once, the reason is that body is consumed with any operation.

8 Client Server Interaction

How to resolve the second problem? Even we want to include the React application inside Express, it won't understand jsx, while on the other hand a React application is not customizable. There are two possible solutions:

- continue to use two different servers (the API server will need CORS Cross-Origin Resources Sharing)
- create a bundle the React application (made only of html, css, js), and having the Express server serve those pages

We will use the two separate servers. A Browser can only make http request from js to the domain provided the response. The CORS must be enabled by the server

9 React Lifecycle

Every component follows some phases, that are dictated by React: **mounting** the component (first in the virtual DOM and the in the real DOM), **updating**, **unmounting**. There are some actions that can only be done in a certain phase, for example `useState()` can only be used in the **updating** phase, or `useEffect()` that works during all the three phases, and will allow us to **handle side effects** in React.

```
function GreetBAD(props) {
  const message = ‘Hello, ${props.name}‘;

  console.log(‘Greetings: ${message}‘);

  return <div>{message}</div>;
}
```

This `log` will be executed in a way that is not predictable, and could be executed more than one time (every time the component is updated). `useEffect` has a *callback* (that contains the logic) and a *list of dependencies* (`useEffect` will execute the callback only if a dependency has changed during the updating phase)

Sometimes there may be a slow internet connection, this might give the impression to the user that everything is frozen if we don't hint him that he just have to wait and that the app is working correctly. A way to handle this is to use a state that tells the app if it's waiting for a response, and by showing an appropriate message.

9.1 Handling API Calls in React

Usually the state comes from the backend, which should be the **single source of truth**, the problem is that there are many clients interacting with the server, this is why we need to periodically check for updates, this state typically is globally managed. In React we also have to manage the **presentation state**, those are information that **should only be displayed to the user**, and must not be stored on a database.

React defines two terms **Rehydrating** and **Dehydrating**, which translates to: *getting the state from the server* (when the app mounts or periodically) and *pushing the state to the server* (something when the state changes). Rehydrating is done by using `useEffect` to fetch the data from the server at mount time, and then we want rehydrate from the server to get the new changes, when should ask for data? The **better than nothing solution** is: As frequently ad possible, every time we update something we also need to check if something changed, the real solution would be for the server to communicate every change to all his clients using **WebSockets**, in fact this is not possible with HTTP.

When dehydrating there could be two possible ways f handling updates, the **optimistic version** whould be to manually update the list state in the app *assuming* the server updated everything correctly,

```
...
```

or we could use a **conservative** approach, and every time we update we wait for the server response that all went well

```
...
```

A good solution would be a mix of the two approaches, which would translate to: set the state with the supposed correct state and **mark** it, when the server responds with an OK status adjust the state to the according status.

10 Session

A Session between the client and the server it's enstablished at a certain point, usually when the user needs to be authenticate to communicate with the server, it's just a data interchange and a base mechanism is the intercahnge of a **sesion ID** which is stored by the server and identifies the user who wants to send request to that server.

Cookie are small portion of data that are automatically handled by the browsers, for example to keep the session ID, the property that a cookie have are: `name`, `value`, `secure`, `httpOnly`, etc...

When we want to login a user he generally sends a POST to a server, with username and password, and it will create a session storage data, the sessionID will be stored there and then send back to the user, after that every request the user does he send the sessionID back to the server, the server will check if it exist and then send a response back or fail if none is found.

```
function LoginForm(props) => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");

  doLogin = (event) => {
    event.preventDefault();
    if (formIsValid)
      // make a POST request to the server
      props.userLoginCallback(username, password);
    else
      // Show invalid form
  }

  // ...
}
```

The server will receive the the username and password, one way to make life easier is by using the middleware **passport**, before using it in express it needs to be configured first, it's possible to choose the kind of authtentication, the data that will be stored in the DB, the hashing function, etc...

In `passport` the `LocalStrategy` defines how to authenticate a user, which takes as input a `verify` callback

```
const passport = require("passport");
const LocalStrategy = requre("possport-local");

passport.user(new LocalStrateegy( function verify(
  username, password, callback) {
    dao.getUser(username, password).then((user) => {
      if (!user)
        return callback(null, false, { message: "incorrect username and/or
    password." });
      return callback(null, user);
    });
  }
});
```

`callaback` tells `passport` the authenticated user or sends an error message.

When storing password in the server it's **mandatory not to store them in plaintext**, it's also mandatory to use a **salt** to be protected from dictionary attacks, when storing passwords it's a best practice to use some cryptgraphic hashing function, in order to do that we can use the built-in node module `crypto` using the `scrypt` function, an example is

```
const keyLen = 32;
const salt = crypto.scrypt.randomBytes(16);
crypto.scrypt(password, salt, keyLen, (err, hasedPassword) => {
  if (!crypto.timingSafeEqual(storedPassword, hashedPassword));
    throw Error(err);
}
```

using a DB

```
exports.getUser = (email, password) => {
  return new Promise((resolve, reject) => {
    const sql = 'SELECT * FROM user WHERE email = ?';
    db.get(sql, [email], (err, row) => {
      if (err) { reject(err); }
      else if (row === undefined) { resolve(false); }
      else {
        const user = {id: row.id, username: row.email};
        const salt = row.salt;
        crypto.scrypt(password, salt, 32, (err, hashedPassword) => {
          if (err) reject(err);
          if(!crypto.timingSafeEqual(Buffer.from( row.password, 'hex'),
    hashedPassword))
            resolve(false);
          else resolve(user);
        });
      }
    });
  });
};
```

We can also add `express-session` which sotres the session data in memory by deafault, **which should be avoided in production**.

```
const session = require('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'));
```

After enabling the session we should decide which infromation to store inside the sessionID, by using `serializeUser` and `deserializeUser`

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

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

After setting up everything we use `passport` to authenticate a user

```
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);
});
```

`'local'` will look for `username` and `password` inside `req.body`.

We need to modify the server for receiving authentication when using React, the reason is that the browser sends the cookies only to the same origin, the problem is that we are using cors, so we need to tell the browser to also include the cookies for the cross origin and to the server to accept the cookies from cross origin.

```
// Client
const response = await fetch(SERVER_URL + "/api/list", {
  credentials: "include",
});

// Server
const corsOptions = {
  origin: "http://localhost:5173",
  credentials: true,
};
```

```
app.use(cors(corsOptions));
```

Some routes can only be accessed by authenticated users, we can use a middleware to make checking simpler, this is achieved by using `req.isAuthenticated()` provided by `passport`

```
const isLoggedIn = (req, res, next) => {
  if (req.isAuthenticated())
    return next();

  return res.status(400).json({ message: "Unauthorized" });
}

app.get("/api/list", isLoggedIn, (req, res) => {});
```

To **logout** we use `req.logout()`.