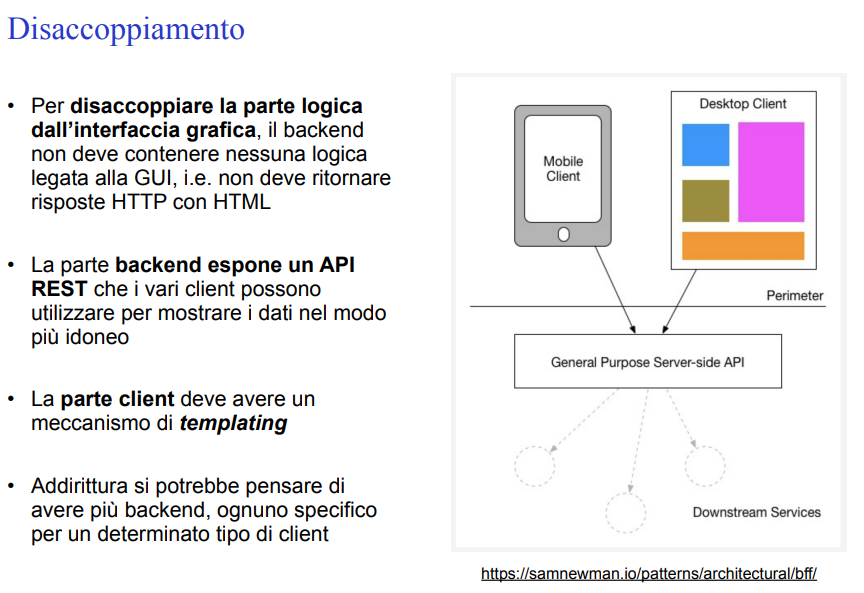
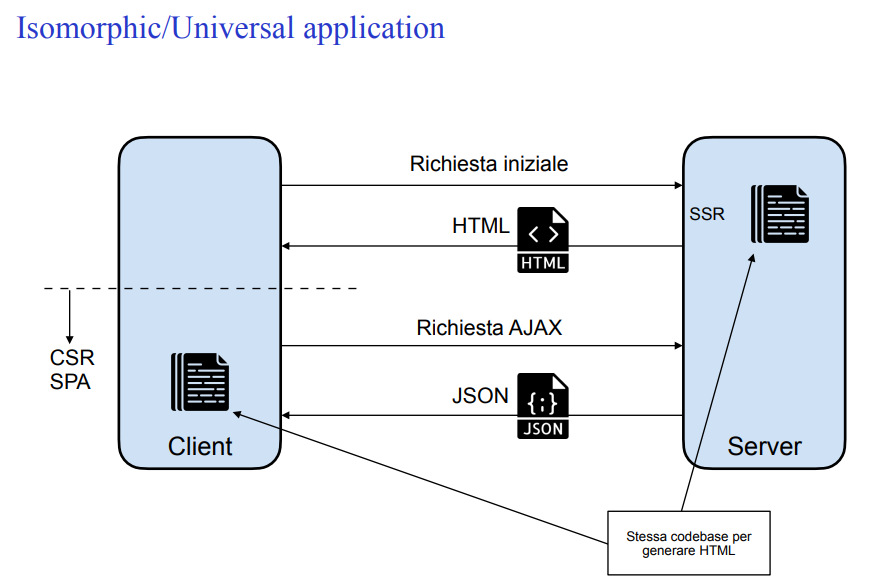
**SSR**: server side rendering🡪 thymeleaf

**CSR**: client side rendering🡪javascript

**SPA:** applicazione che interagisce tramite richieste in background tramite richieste ajax, la pagina non viene mai ricaricata.





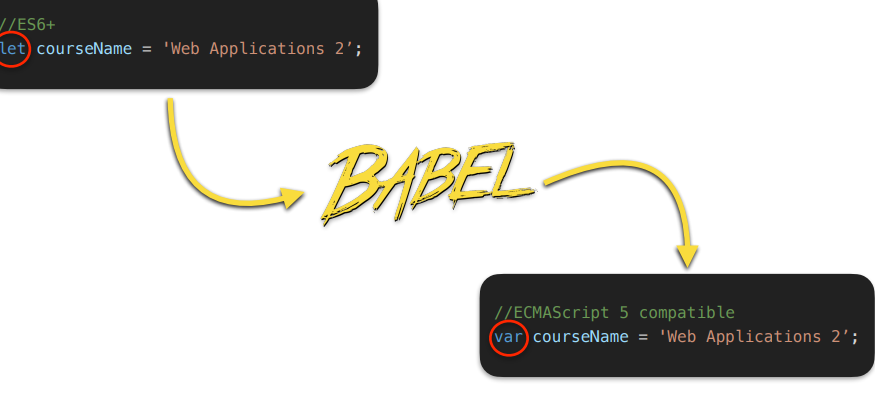
il server deve saper eseguire node.js oppure ci sono interpreti nei linguaggi come java. La prima pagine deve essere renderizzata lato server.

Js è un linguaggio interpretato, v8, spidermonkey, chakra, sono alcuni interpreti usati dai browser.

Supporta tipi, operatori, oggetti e metodi. Tipizzazione dinamica

**Transpilers** è uno strumento che converte il codice più recente in ecmaScript più rencente(Js più recente) in ecmaScript 5, che è supportato da tutti i browser.

Esempio: Babel

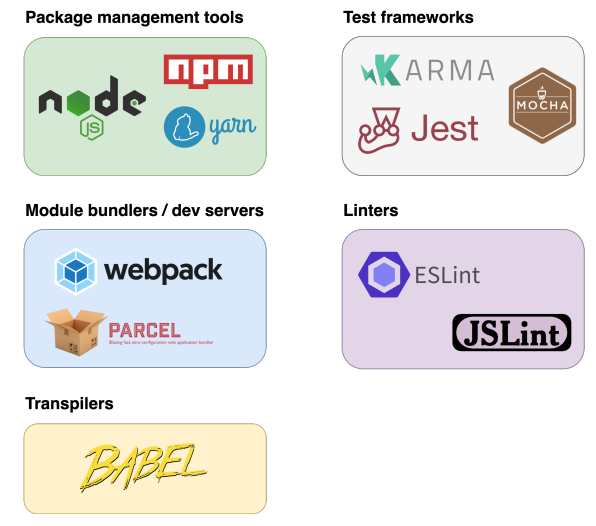


Js oggi non è usato solo per pagine web, ma viene usato anche per applicazioni lato server (NodeJS)

Con NodeJS non si può accedere al dom

Package manager: npm

Il file è package.json



Var : scope di funzione

Let : scope di blocco

Const : scope di blocco, ma sono immutable, costanti

Window elemento root, se una variabile è globale si può richiamare con *window.nome\_variabile*

Se si prova a richiamare una variabile fuori dal blocco di appartenenza si ottiene reference error.

**Usare let, preferibile**

Le variabili possono essere usate anche prima della dichiarazione, hoisting è un meccanismo che porta la dichiarazione in cima, l’assegnamento seguo il flusso del codice se ho

*console.log(a)*

*var a=2*

*console.log(a)*

***output****: undefined, 2*

a viene portato in cima e viene assegnato dopo il primo console log

un array vuoto è true, null, undefined e NaN sono false.

Concatenare string: tamplate leteral

` blablabl ${var1}` (` : Alt+96)

**Oggetti**

gli oggetti in js sono visti come una map

let myObj={

Key1:value1,

Key3:’value2

}

**Shorthand property**: omette i nomi delle chiavi, e utilizzerà il nome della variabile utilizzata per assegnare i valori

Let name1=”..”

Let surname2=”…”

Obj={name1,surname2}

Oggetto risultante

Obj={ name1:…, surname2:….}

Per accedere ai campo **oggetto.nome\_campo**

Destructoring

Const {name,type,stats}=gameCharacter; -> verranno create le variabili specificate e assegnerà la corrispettiva chiave dell’oggetto passato a sinistra. i campi devono corrispondere ai nomi delle chiavi a meno che non si specifichi con chiave:nome della varibile

x={a:1,b:2};

const {a:f,b}=x;

… spread operator: si usa per destrutturare un oggetto e tutto quello che non è stato destrutturato andrà a finire nella variabile alla quale è stato anteposto …

Il nome può essere uno qualunque.

**Shallow copy e deep copy**

Shallow copy: copio il riferimento del vettore/oggetto e gli oggetti sono condivisi

**Object.assign({}, oggettoDaClonare)**

Oppure

**Const shallowCopy={…oggettoDaClonare}**

**DeepCopy**

Per fare una deepCopy:

**JSON.parse(JSON.stringify(oggettoDaClonare))**

**Funzioni**

In JS ci sono più tipi di funzioni:

* **Function Declarations**: function nomeFunzione(args){…}
* **Function Expressions**: let a=function (args){…}; ….. a(args);
* **Arrow Functions**: (args)=>{….}

Le funzioni in js possono essere degli oggetti eseguibili.

IIFE: immediately invoked function expressions. Richiama le funzioni subito dopo la dichiarazione.

**(function (){….})()** la funzione viene eseguita subito dopo la dichiarazione.

Le arrow functions e function expressions non sono soggette all’hoisting

**This**  nelle arrow function si aggancia al this della funzione padre più vicina.

**Function Scope**

Ciò che accade in una funzione è detto scope/contesto se creo in una funzione in un'altra creo un altro scope e gli scope più interni possono vedere gli scope dei livelli superiori.

Se ci sono 2 variabili con lo stesso nome andrà a prendere il valore della variabile nello scope più vicino.

**Function Closures**

È la combinazione di una funzione ed uno scope contenente dei riferimenti.

Mantiene i riferimenti allo scope di definizione e non quello di esecuzione.

Viene usato per mantenere la privacy delle variabili.

**Callback**

si passa una funzione anonima o arrow function nel parametro della funzione e viene richiamata al suo interno.

**This**

Indica il contesto di esecuzione di una funzione.

Il valore this nella funzione si può modificare tramite call, apply e bind.

Call e apply consentono di chiamare una funzione modificando this

Bind consente di eseguire una funzione con un this differente.

Nel contesto globale this, cambia in base a dove si utilizza.

Console.log(this) in una pagina web restituirà window

Se eseguito in nodejs ritornerà undefined perché eseguito in modalità strict.

Se faccio this all’interno di un oggetto, esso è il contesto.

Call e apply fanno la stessa cosa cambia solo come vengono passati i parametri

**Utilities per array**

**Map** consente di generare un nuovo array che viene popolato applicando la funzione fornita ad ogni elemento dell’array di partenza.

Immagine che contiene testo, monitor, argento

Descrizione generata automaticamente

**Filter** permette di generare un nuovo array che viene popolato applicando una condizione espressa con una funzione fornita (callback) ad ogni elemento dell'array di partenza.

Immagine che contiene testo

Descrizione generata automaticamente

**Reduce** permette di eseguire una funzione reducer su ogni elemento di un array. La funzione da fornire (callback) prende come parametri obbligatori un accumulatore (per default all'inizio è il primo elemento) e l'elemento corrente.

**Immagine che contiene testo

Descrizione generata automaticamente**

In pratica ad ogni iterazione viene assegnato all'accumulatore il valore ritornato dalla funzione. Il nuovo accumulatore verrà passato alla prossima iterazione.