1) WWW – osnove (HTTP, HTML, URL)

   WWW (World Wide Web) je sistem međusobno povezanih hipertekst dokumenata i multimedijalnog sadržaja kome se pristupa korišćenjem web pretraživača. Web stranice, multimedijalni sadržaj i drugi tipovi dokumenata se prenose putem HTTP-a (Hypertext Trasfer Protocol). HTTP je stateless protokol, što znači da je svaki request između klijenta i servera nezavisan i sadrži sve informacije potrebne za obradu requesta.

   HTML (Hypertext Markup Language) je standardni markup jezik za struktuiranje web stranice. On obezbeđuje način za opisivanje strukture stranice korišćenjem tagova. Tagovi se navode između znakova '<' i '>'.

   URL (Uniform Resource Locator) je adresa koja se koristi za pristupanje resursima na internetu. URL se obicno stastoji od naziva protokola kojim se podaci prenose (HTTP, HTTPS, FTP...), domena (human-readable adresa koja identifikuje resurs na internetu), puta do resursa (npr. "/proizvodi/laptopovi") i query parametara (omogućavaju prosleđivanje parametara resursu npr. "?q=search&category='elektronika'").

2) Web aplikacije

   Web aplikacija je softverska aplikacija kojoj se pristupa preko interneta korišćenjem web pretraživača. Web aplikacija se sastoji od serverske i klijentske strane i mehanizma za perzistenciju (baze). Web aplikacije se razlikuju od standardnih desktop aplikacija jer ne zahtevaju instalaciju, već se hostuju na web serveru i pristupa im se korišćenjem url-a. Najznačajnije karakteristike web aplikacije su pristupačnost i nezavisnost od platforme, skalabilnost, održivost... Primeri web aplikacija su YouTube, Facebook, Lichess...

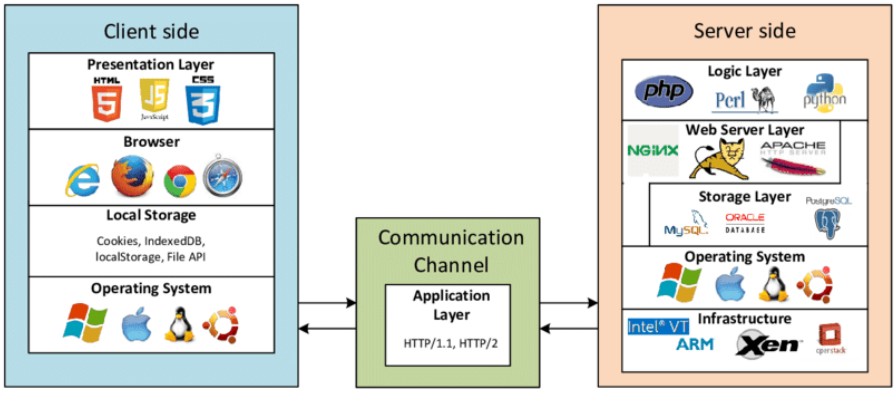
3) Desktop vs Mobilne vs Web aplikacije

   Desktop aplikacije su dizajnirane da se izvršavaju na određenoj platformi (operativnom sistemu) i direktno se instaliraju na korisnički računar. Ažuriranje aplikacije zahteva dodatnu instalaciju, međutim mogu se koristiti i offline. Omogućavaju visoke multimedijalne performanse (3d igre), ali je dodavanje multimedijalnog sadržaja komplikovano.

   Mobilne aplikacije su dizajnirane da se izvršavaju na određenoj mobilnoj platformi (najčešće iOS ili Android operativnom sistemu). Korisnici aplikacije najčešće preuzimaju iz odgovarajućeg app store-a, koji proizvođač operativnog sistema omogući. U zavisnosti od namene aplikacije moguće je koristiti ih i offline.

   Web aplikacije ne zavise od platforme koju korisnik koristi. Izvršavaju se na web serveru i pristupa im se putem url adrese. Ažuriranje aplikacije ne zahteva dodatne instalacije od strane korisnika. Imaju limitiran GUI, ali se multimedijalni sadržaj jednostavno dodaje. Nije im moguće pristupiti bez pristupa internetu.

4) Arhitektura Web aplikacije (dijagram, cloud)



Cloud se odnosi na postupak korišćenja udaljenih servera i mreža, koji su hostovani na internetu, kako bi se pamtili, upravljani i obrađivali podaci, umesto da se oslanja na lokalne resurse. Cloud se koristi u raznim domenima, uključujući i treniranje modela veštačke inteligencije, prikupljanje i obrada podataka, IoT... Glavne prednosti korišćenja clouda su skalabilnost, fleksibilnost, accessibility, cost-efficiency...

5) Klijent strana (tehnologije, izazovi)

Klijentska strana web aplikacije se odnosi na deo aplikacije koji se izvršava u korisničkom pretraživaču. Osnovne komponente klijentske strane su HTML (Hypertext Markup Language) koji se koristi za struktuiranje web strane, CSS (Cascading Style Sheet) koji se koristi za stilizaciju web stranice i JavaScript – progoramski jezik kojim se dodaje iteraktivnost i dinamičnost web stranice. Postoji i veliki broj JavaScript frameworkova koji služe za razvoj klijentske strane (React, Angular, Vue, jQuery).

PWA (Progressive Web Application) su web aplikacije koje kombinuju najbolje osobine web i mobilnih aplikacija. Omogućavaju offline rad (keširaju određene podatke) kao i push notifikacije.

WebAssembly je tehnologija koja omogućava izvršavanje aplikacija pisanih u različitim programskim jezicima da se izvršavaju u web browseru.

Izazovi koji postoje prilikom projektovanja klijentske strane su UX (obezveđivanje user experience-a pr. response time vs responsiveness), UI (obezbeđivanje odgovarajućeg korisničkog interfejsa), obezbeđivanje adektvatnih performansi, omogućavanje cross-browser i cross-device izvršavanje (različite rezolucije ekrana – responsive design), kao i odabir odgovarajuće tehnologije koja je pogodna za konkretan slučaj korišćenja.

6) Server strana (tehnologije, izazovi)

Server strana aplikacije se odnosi na deo aplikacije koji se izvršava na serveru. Zadužena je za obrađivanje korisničkih requestova , izvršavanje osnovnie poslovne logije aplikacije i pristup bazi podataka. Tehnologije koje se koriste za izradu logičkog sloja serverske strane su PHP, ASP.NET, Node.js, Python's Django framework, Ruby... Za sam web server se koriste IIS, Apache, Nginx... U zavisnosti od zahteva, mogu se koristiti relacione baze podataka, NO-SQL baze, graf ili in-memory baze...

Izazovi koje se prevazilaze prilikom izrade serverske strane su skalabilnost (kluster i load balanceri), sigurnost (autentifikacija, autorizacija, sql-injection, DDOS), perzistencija podataka....

7) Protokoli (HTTP 1.0, 1.1, 2.0, Websocket)

HTTP i Websocket su protokoli koji se koriste za uspostavljanje veze između klijenata i servera i razmenu podataka.

HTTP 1.0 je prva verzija ovog protokola. Omogućava jednostavnu request-response komumnikaciju između klijenata i web servera. HTTP 1.0 je stateless protokol (protokol bez stanja) što znači da je svaki par request-response nezavisan, tj. svaki request sadrži dovoljno informacija kako bi bio obrađen.

HTTP 1.1 je uveden kao poboljšanje nad HTTP 1.0. Uvedena je keep-alive konekcija (perzistencicja konekcije) kako bi se izbegao problem uspostavljanja konekcije pri svakom zahtevu. Uveden je pipelining, što omogućava da se šalje više requestova kroz jednu konekciju bez čekanja odgovora.

HTTP 2.0 je uveden kako bi se prevazišli problemi sa performansama koji postoje kod HTTP 1.1. Uveden je multiplexing (slanje više requesta i responsea kroz istu konekciju) i server push, što omogućava serveru da inicicra slanje podataka.

Websocket je protokol koji omogućava full-duplex komunikacione kanale preko jedne tcp konekcije. Razlikuje se od HTTP-a po tome što omogućava bidirekcionu komunikaciju u realnom vremenu.

HTTPS (HTTP Secure) je ekstenzija standardnog HTTP protokola koja se koristi radi sigurnije komunikacije na internetu. HTTPS vrši enkripciju podataka koji se šalju kroz mrežu, čineći ih beskorisnim bilo kom osim korisniku. Kriptografija se vrši kriptografskim protokolima kao što su TLS kod HTTP-a i SSL kod HTTPS-a. Ovim je prevaziđena bojazan od Man-in-the-Middle napada.

8) HTTP – opis, metode zahteva, poruke odgovora

HTTP (Hypertext Transfer Protocol) je protokol na aplikativnom nivou koji se koristi za prenos hipertekst dokumenata (HTML) i drugih dokumenata. HTTP je osnova razmene podataka na webu. HTTP je stateless protokol (protokol bez stanja) što znači da je svaki par request-response nezavisan, tj. svaki request sadrži dovoljno informacija kako bi bio obrađen.

Osnovne HTTP metode su

- GET – koristi se za zahtevanje podataka od servera
- POST – koristi se za slanje podataka server (popunjena forma, upload dokumenata)
- PUT – koristi se za ažuriranje resursa na serveru
- DELETE – koristi se za uklanjanje resursa na serveru

Kod HTTP 1.0 svaki request zahteva uspostavljanje konekcije. Nakon slanja requesta vraća se poruka i to:

- 1xx – informativna
- 2xx – uspeh

- 3xx – redirekcija
- 4xx – greška kod klijenta
- 5xx – greška na serveru

9) HTML 5

HTML (Hypertext Markup Language) je podskup XML-a. Osnovna primena HTML-a je struktuiranje sadržaja na web stranicama. HTML dokument se sastoji od elemenata (…), atributa (…), komentara (<!-- … -->) i znakovnih referenci (&quot = ‘). HTML dokument se sastoji iz Doctype (informacije o dokumentu), zaglavnja (informacije o dokumentu) i tela dokumenta koje sadrži sadržaj tj skup HTML elemenata.

HTML elementi se sastoje iz početnog dela (taga) i završnog dela ( ). Neki elementi nemaju početni i krajnji deo (npr.
).

HTML elementi mogu biti blokovi ili inline. Blokovski elementi zauzimaju ceo red (širinu ekrana) u kome se nalaze. Inline elementi ne zahtevaju ceo blok. Svaki HTML element ima pridružen CSS objekat i uokviren je kutijom (box model) čiji je izgled definisam tim CSS objektom. Box model podrazumeva marginu, border, padding i content.

HTML5 je peta revizija HTML-a. Novi elementi koji se javljaju kod HTML 5 su semantički elementi (header, nav, section, article, footer, aside). Oni predstavljaju samostalni deo strane ili aplikacije koji može da se distribuira i koristi.

-

Da li bi sadržaj mogao samostalno da se prikaže kao RSS unos? Da => <article>
Da li je se sastoji od međusobno povezanog sadžaja? Da => <section>
Ukoliko ne postoji nikakva semantička veza, koristiti

Aside se koristi za sadržaj koji nije glavni fokus artikla ili strane, ali **se odnosi** na artikal
Može biti napisan u okviru artikla
Ne znači da mora da bude lociran bočno od artikla

Nav predstavlja deo strane koji sadrži linkove ka ostalim stranama ili delovima na strani (navigacione linkove)
Ne treba svaku grupu linkova ubaciti u <nav>, već samo glavne navigacione blokove.
Npr, footer sadrži linkove ka copyright-u, uslovima korišćenja sajta i glavnoj strani. Njima ne treba <nav>, već je dovoljan sam <footer> element

-

Pravila za pisanje HTML 5:
Koristiti samozatvarajuće početne delove elemenata (oni koji imaju / na kraju i kojima ne treba završni deo, npr.
) samo za prazne elemente
Sva imena elemenata i atributa se pišu malim slovima
Svi atributi se pišu između apostrofa (') ili navodnika (")
Dodati blanko pre / kod elemenata koji imaju samozatvarajući početni deo, npr:
 umesto

Uključiti završne delove za elemenate koji mogu imati sadržaj ali se ostavljaju prazni (npr. "<img></img>", ne "<img />")

10) CSS, opste, progresivno poboljšanje

Progresivno poboljšanje je pristup razvoju web stranica koji podrazumeva da osnovna funkcionalna verzija web sajta dostavlja svim korisnicima, bez obizra na korišćeni uređaj, bio on PC, laptop ili telefon i bez obzira na korišćeni pretraživač. Počinje se sa osnovnim funckionalnostima (HTML-om). Ceo sadržaj se semantički označava i dodaje mu se stil, korišćenjem spoljašnjeg CSS-a. Ponašanje se isključivo definiše preko eksternog javaskripta.

CSS (Cascading Style Sheets) je jezik koji se koristi za definisanje izgleda HTML dokumenta u pretraživaču. CSS objeki se sastoje od mnoštva atributa. Najkorišćeniji su border (izgled okvira), margin (udaljenost od drugih elemenata), padding (udaljenost okvira od sadržaja), font (izgled slova), background (izgled pozadine)... CSS stil se elementu može pridružiti na tri načina: 1) U style elementu uokvir strane
2) U style atributu u okviru samog elementa
3) U eksternom css fajlu.

Elementi se iz eksternog fajla mogu selektovati na više načina: selekcijom tipova, klasa, ID-a, HTML atributa, kombinacijom pravila.

11) CSS selektori

Elementi mogu biti selektovati na više načina:

1) Korišćenjem tipova – navođenjem tipa elementa, CSS stil će biti primenjen na sve elemente tog tipa. Tipove je moguće grupisati (h1, h2, h3 {…})
2) Selekcijom klasa – svaki element može pripadati jednoj ili više klasa. To se navodi u class atributu. Sintaksa koja se koristi za gađanje klasa je .<naziv klase>
3) Selekcijom id – Sintaksa koja se koristi je #<naziv klase>
4) Selekcijom HTML atributa - <element>[attr=”attr_val”]
5) Selekcijom atributa *= sadrzi pr. [class*=”col-“]
   ^= počinje sa
   $= završava se sa
6) parent > child – Korišćenjem > selektuju se deca
7) parent desc – Korišćenjem „ „ selektuju se svi potomci

Svaki HTML element kaskadno nasleđuje karakteristike CSS objekta pridruženog roditeljskom HTML elementu.

JAVASCRIPT

1) JavaScript (JS), opšte

JavaScript je lightweight interpretirajući (ili just-in-time kompajlirani) programski jezik sa first-class funkcijama. Iako je JavaScript je najpoznatiji kao skripting jezik za razvoj web stranica, veliki broj non-browser okruženja ga takođe koristi (Node.js, Apache CouchDB, Adobe Acrobat). JavaScript je prototype-based, podržava više paradigmi, single-threaded, dinamički tipiziran.

JavaScript's dynamic capabilities include runtime object construction, variable parameter lists, function variables, dynamic script creation (via eval), object introspection (via for...in and Object utilities), and source-code recovery (JavaScript functions store their source text and can be retrieved through toString()).

Javascript – univerzalni jezik klijent strane

Skript se izvršava na klijentskoj mašini pošto se dokument učita, ili u nekom drugom trenutku, npr. kad se aktivira link.

Skripte pružaju autoru sredstva za proširenje funkcionalnosti i interaktivnosti HTML dokumenata:

- dinamički menjati sadržaj dokumenta
- validacija korisničkih podataka u obrascu, kako bi se detektovale greške pre slanja podataka serveru.
- obrada događaja koji se odnose na dokument, kao što su učitavanje i zatvaranje dokumenta, fokus elementa, pokreti kursora miša,itd.
- Komunikacija sa serverom (AJAX, Web sockets)

Najpopularniji jezik Web-a, zastupljen na klijentu (Web pretraživači), na serveru (Node.js) ali i van Web-a: PDF, desktop widžeti

Standardizovan ECMAScript specifikacijom

Java i JavaScript su semantički potpuno različiti jezici

Sintaksa Javascripta je skoro identična Javi, C# ili PHP-u.

Na klijentu se tradicionalno implementira kao interpretatorski jezik

Prototipski skipt jezik dinamičkih tipova

- Prototipski bazirano programiranje – stil objektno orijentisanog programiranja gde se nasleđivanje postiže kloniranjem postojećih objekata, koji služe kao prototipi
- Dinamički tipovi – ista promenjiva može da menja tip podatka

```
var k = 5;
var name = “Hello”;
name = 4;
```

- JS objekti su asocijativni nizovi

obj.x ⇔ obj[‘x’]

- Javascript je case-sensitive!

2) JS tipovi (Primitivni, objekti)

U primitivne tipove u JS spadaju string, number, boolean, undefined (let x), null (let y = null), symbol (uvedeni u ecma6, predstavljaju jedinstveni identifikator, const mySymbol = Symbol('description').

U objekte spadaju:

1) Anonimni objekti const person = { name: 'John', age: 30 }
2) Nizovi const numbers = [1, 2, '3', 4, '5']
3) Funckije function add(a, b) { return a + b; }
4) Date const curDate = new Date();
5) Regex
6) Map const myMap = new Map();
   myMap.set('key1', 'val1');
7) Set ....

3) Promenljive

Promenljive se u JS mogu deklarisati na 3 načina.

0) var – istorijski, danas nije često u upotrebi
1) let – koristi se za promenljive koje je moguće redeklarisati
2) const – koristi se za promenljive koje ne bi trebale biti redeklarisane

Identifikator promenljive može da sadrži brojeve, slova, dolar i _. Ne sme da počinje brojem.

4) – 7) Funckije, razlika između deklaracije i FE, hoisting, lambda fje

```
function max(x, y) {
    if (x > y) return x;
    return y;
}
const c = max(3, 4)
```

FE

```
var square2 = function(a) {
            return a * a;
}
```

Korišćenjem arrow notacije

```
var square2 = a => a * a
```

Hoisting je ponašanje u JS koje podrazumeva da su deklaracije promenljivih i funkcija pomerene na vrh u fazi kompajliranja. Ovo omogućava korišćtenje varijabli i funkcija u kodu pre nego što su deklarisane. Hoistuju se samo deklaracije, ne i definicje.

Funckije i FE se razlikuju po tome što su funckije hoistovane, moguće im je pristupati u kodu pre njihove deklaracije, dok sa FE to nije moguće. FE su korisni kada je funckiju potrebno proslediti kao parametar neke druge funckcije.

Lambda ili anonimne funckije su funkcije bez imena. Moguće je pisati ih korišćenjem arrow notacije ili korišćejnem ključne reči function. Često se koriste kao callback funckije ili IIFE (immediately invoked function expressions).

```
// Anonymous function as a callback
document.addEventListener("click", function() {
  console.log("Document clicked!");
});

// IIFE (Immediately Invoked Function Expression)
(function() {
  console.log("I am an IIFE!");
})();
```

8) Nizovi, ugrađene metode

Niz je struktura koja omogućava čuvanje i manipulaciju elementima koji se nalaze u njemu.
Nizovi u JS mogu biti heterogeni, tj. sadržati različite tipove.
deklaracija niza: let niz = []
let niz = [1, 2, “str”]
Elementima se može pristupiti pomoću indeksa (niz[0])
Novi element se u niz dodaje korišćenjem metode .push()
Poslednji element se izbacuje korišćenjem metode .pop(),
Prvi element se izbacuje korišćenjem metode .shift()
Element se umeće na prvo mesto korišćenjem .unshift()
Slajsovanje .slice(od, do)
Splajsovanje .splice(od, do, vals)
Duzina niza .length
.foreach()

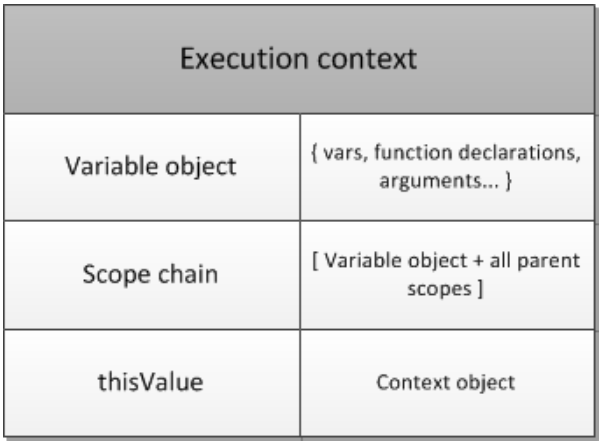
Filter metoda kreira novi niz koji sadrži elemente originalnog niza koji ispunjavaju uslov koji se nameće funkcijom koja se prosleđuje kao parametar. Ne modifikuje originalni niz. Dužina novog niza može biti različita od dužine originalnog.
Map metoda kreira novi niz koji sadrži elemente nad kojima je primenjena funkcija koja je prosleđena kao parametar. Ne modifikuje originalni niz. Dužina ovog niza jednaka je dužini originalnog.
Reduce metoda primenjuje prosleđenu funkciju nad akumulatorom, u redosledu sleva nadesno, dok ne ostane 1 element.
Metode je moguće nadovezivati.
const numbers = [1, 2, 3, 4, 5, 6];

```
// Chaining filter, map, and reduce
const result = numbers
  .filter(function(number) {
    return number % 2 === 0;
  })
  .map(function(number) {
    return number * 2;
  })
  .reduce(function(accumulator, currentNumber) {
    return accumulator + currentNumber;
  }, 0);

console.log(result);
```

Moguće je koristiti arrow notaciju.

10) Closure i kontekst izvršenja funckija



Kontekst izvršenja se sastoji od skupa promenljivih funckije, lanca opsega i this pokazivača. Skup promenljivih funckije čine unutrašnje promenljive, deklaracije funckija i parametri funckije. Lanac opsega predstavlja skup promenljivih funckije i svih roditeljskih opsega.

Closure je osobina javascripta koja omogućava da funckija zadrži pristup promenljivama iz spoljašnjeg opsega, čak i kada je funkcija iz spoljašnjeg opsega završila izvršavanje. Ovo se omogućava postojanjem scope chain-a u kontekstu izvršenja. Closure se može koristiti za enkapsulaciju podataka, i kreiranje privatnih promenljivih.

```
function counter() {
  let count = 0;

  return function() {
    count++;
    console.log(count);
  };
}

const increment = counter();
increment();
increment();
```

```
function createPerson(name) {
  let privateName = name;
  return {
    getName: function() {
      return privateName;
    },
    setName: function(newName) {
      privateName = newName;
    }
  };
}
const person =
createPerson('John');
console.log(person.getName());
person.setName('Jane');
console.log(person.getName());
```

11) IIFE (+klasa sa privatnim članovima)

IIFE (Immediately Invoked Function Expression) je funckijski izraz koji se definiše i izvršava odmah nakon kreiranja. Koristi se za kreiranje privatnog scope-a za promenljive, kako ne bi došlo do polluting global scope-a, aliasing članova (davanje drugih imena) i za globalne objekte (window u browseru?).

```
const counter = (function (pocetnaVrednost) {
   let counter = pocetnaVrednost;
   function display() {
      console.log(`brojac = ${counter}`);
   }
   function reset() {
      counter = 0;
      console.log("resetovan");
   }
   function increaseCounter() {
      counter++;
      display();
   }
   return {
      increase: increaseCounter,
      reset: reset
   }
})(5);
```

Kreiranje privatnih članova pre uvođenja standarda ES-6 vršilo se korišćenjem funckija i iskorišćavanjem closure-a. Primer u odgovoru 10.

12) Objekti literali

Objekti literali su comma-separated lista key-value parova, oivičeni sa { }. Svaki key-value par predstavlja polje objekta. Pr.

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30,
  isStudent: false,
  greet: function() {
    console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
  }
};
```

Mogu se i referencirati promenljive. Pr firstname. Vrednosti ključeva mogu biti i dinamičke. U tom slučaju navode se između [ ]. Pr [key]. ES6 omogućava da se unutar definicije objekta literala definiše funkcija. Moguć je i unzip (kao tuple u pythonu).

9) DOM, definicija i najčešće funckije API-ja

DOM (Document Object Model) je standardizovani API (način manipulacije) za strukturu dokumenata (XML, HTML...). DOM nudi standardni skup objekata za predstavljanje HTML i XML dokumenata, kao i standardni interfejs za njihovu manipulaciju i pristup. Pre DOM-a svaki web pretraživač imao je drugačiji pristup objektima, što je sprečavalo cross browser aplikacije. DOM svaki dokument vidi kao stablo.

Najčešće korišćeni načini za pretragu HTML dokumenata u DOM stablu su

document.getElementById("nekiId")
Vraća HTML element sa navedenim ID-om.
element.getElementsByTagName("button")
Vraća niz HTML elemenata navedenog tipa koji su potomci element-a
document.querySelectorAll("table td input[type='checkbox']")
Izvršava CSS selekciju i vraća niz elemenata
document.querySelector("table td input[type='checkbox']")
Izvršava CSS selekciju i vraća prvi nađeni element

Najčešće korišćeni atributi su:

atribut	objašnjenje
style	pristup CSS objektu trenutnog HTML objekta
innerHTML	sadržaj elementa (između početnog i krajnjeg dela elementa)
value	(kod **input** elementa) vrednost elementa
parentNode	HTML objekat koji sadrži trenutni HTML objekat
childNodes	Niz potomaka, HTML objekata (elemenata, tekstualnih čvorova i za formatiranje)
children	Niz potomaka, isključivo HTML elemenata

13) Module šablon za definisanje klasa

Šablon omogućava pravljenje više instanci na osnovu konstruktorske funkcije.

```
1) var Animal = (function() {
2)   // Private static variable
3)   var animalCount = 0;
4)
5)   // Private constructor
6)   function Animal(name) {
7)     // Private instance variable
8)     var privateVariable = 'I am private';
9)
10)      this.name = name;
11)      this.getId = function() {
12)        return privateVariable + animalCount++;
13)      };
14)    }
15)
16)    // Public method
17)    Animal.prototype.eat = function() {
```

```
18)      console.log(this.name + ' is eating.');
19)    };
20)
21)    // Public static method
22)    Animal.getCount = function() {
23)      return animalCount;
24)    };
25)
26)    // Return the constructor function
27)    return Animal;
28)  })();
29)
30)  // Usage
31)  var lion = new Animal('Leo');
32)  lion.eat(); // Outputs: Leo is eating.
33)  console.log(lion.getId()); // Outputs: I am private0
34)
35)  var tiger = new Animal('Tony');
36)  tiger.eat(); // Outputs: Tony is eating.
37)  console.log(tiger.getId()); // Outputs: I am private1
38)
39)  console.log(Animal.getCount()); // Outputs: 2
```

40) Prototipovi (+ prototipski lanac)

U JS prototip je mehanizam kojim se postiže nasleđivanje. Svaki objekat u JS ima prototip i kada se pristupa metodu ili atributu objekta JS traži metod ili atribut u samom objektu, a zatim u prototipu, zatim u prototipu prototipa itd. formirajiući na taj način prototipski lanac. Svaki objekat ima interno svojstvo __proto__ koje je link do prototipskog objekta. Prototip takođe ima svoj prototip. Prototipski lanac se završava objektom koji ima null kao prototip. obj.__proto__ == Object.getPrototypeOf(obj).

Svi objekti u JS su povezani sa prototipskim objektom. To uključuje i metode i propertije koji su dostupni svim objektima u JS.

Prototip ima constructor svojstvo koje pokazuje na konstruktorsku funkciju. Konstruktorska funckija se koristi za kreiranje objekta i takođe je izvedena iz objekta. Poseduje prototip svojstvo koje vodi do prototipskog objekta.

Primitivni tipovi imaju svoje prototipove povezane sa Object prototipom. Vrednost promenljive primitivnog tipa je immutable, dok su objekti mutable.

15) Klase preko prototipova (dodavanje metoda na prototip)

```
[1]   // Constructor function
[2]   function Person(name, age) {
[3]     this.name = name;
[4]     this.age = age;
[5]   }
[6]
[7]   // Adding a method to the prototype
[8]   Person.prototype.greet = function() {
[9]     console.log(`Hello, my name is ${this.name}.`);
[10]  };
```

```
[11]
[12]  // Creating an object
[13]  const john = new Person('John', 30);
[14]
[15]  // Accessing the method from the prototype
[16]  john.greet(); // Outputs: Hello, my name is John.
```

16) ES6 klase, nasleđivanje (i primer)

Od standarda ES6 moguće je koristiti ključnu reč class za kreiranje konstruktorskih funckija i definisanje prototipova. Kakogod, ispod haube, idalje su uključeni prototipovi. Zak lase se u JS kaže da su sintaksni šećer preko prototype-based sistema.

```
1) class Animal {
2)   constructor(name) {
3)     this.name = name;
4)   }
5)
6)   eat() {
7)     console.log(`${this.name} is eating.`);
8)   }
9) }
10)       class Dog extends Animal {
11)         constructor(name, breed) {
12)           super(name);
13)           this.breed = breed;
14)         }
15)
16)         bark() {
17)           console.log(`${this.name} is barking.`);
18)         }
19)
20)         eat() {
21)           super.eat();
22)           console.log(`${this.name} prefers bones.`);
23)         }
24)       }
25)       const genericAnimal = new Animal('Generic Animal');
26)       const myDog = new Dog('Buddy', 'Golden Retriever');
27)       genericAnimal.eat(); // Outputs: Generic Animal is eating.
28)       myDog.eat(); // Outputs: Buddy is eating. Buddy prefers bones.
29)       myDog.bark(); // Outputs: Buddy is barking
```

17) ES6 Property članovi klase

```
[1]class Circle {
[2]  constructor(radius) {
[3]    this._radius = radius;
[4]  }
[5]
[6]  get diameter() {
[7]    return this._radius * 2;
[8]  }
[9]
[10]         set diameter(diameter) {
[11]           this._radius = diameter / 2;
[12]         }
[13]
[14]         get area() {
[15]           return Math.PI * this._radius ** 2;
```

```
[16]         }
[17]       }
[18]
[19]       const myCircle = new Circle(5);
[20]       console.log(myCircle.diameter); // Outputs: 10
[21]       console.log(myCircle.area); // Outputs: 78.53981633974
[22]
[23]       myCircle.diameter = 12;
[24]       console.log(myCircle.area); // Outputs: 113.0973355292
```

18) ES6 moduli

EcmaScript 2015 uveo native podršku za module u JS. ES6 koristi ključne reči import i export kako bi postigao podršku.

U js fajlu iz koga će konstante/ metodi biti uvezeni, ispred deklaracije dodaje se ključna reč export. Primer fajl mat.js. Kako bi se neka funckionalnost uvezla u fajl koristi se ključna reč import.

// mat.js

```
// Exporting variables
export const PI = 3.14159265359;

// Exporting functions
export function add(x, y) {
  return x + y;
}

// Exporting a class
export class Calculator {
  static multiply(x, y) {
    return x * y;
  }
}
```

```
import { PI } from './math';

// Importing functions
import { add } from './math';

// Importing a class
import { Calculator } from
'./math';

console.log(PI); // Outputs:
3.14159265359
console.log(add(5, 3)); // Outputs:
8

const result =
Calculator.multiply(4, 6);
console.log(result); // Outputs: 24
```

Uvođenje podrške za module obezbeđuje clean i organizovan način za struktuiranje koda u većim aplikacijama, što pomaže sprečavanju zagađenja globalnog scope-a, i omogućava da kod bude reused u više delova aplikacije.

19. Asinhrono programiranje, Callback fje

JavaScript je single-threaded jezik. Asinhrono programiranje omogućava izvršavanje koda bez blokiranja ostatka programa. Asinhroni kod dozvoljava drugim operacijama da nastave sa izvršavanjem dok se čeka izvršenje druge operacije. Asinhrono programiranje je efikasno kod IO taskova (network request, file ops).

Callback fje su fje koje se prosleđuju kao argumenti drugih funckcija, i izvršavaju se nakon završetka specifične operacije. Callback fje se koriste kod asinhronog programiranja kako bi izvršile neki posao nad rezultatom asinhrone operacije.

```
function fetchData(callback) {
  setTimeout(function() {
    const data = 'This is some data.';
    callback(data);
  }, 2000);
```

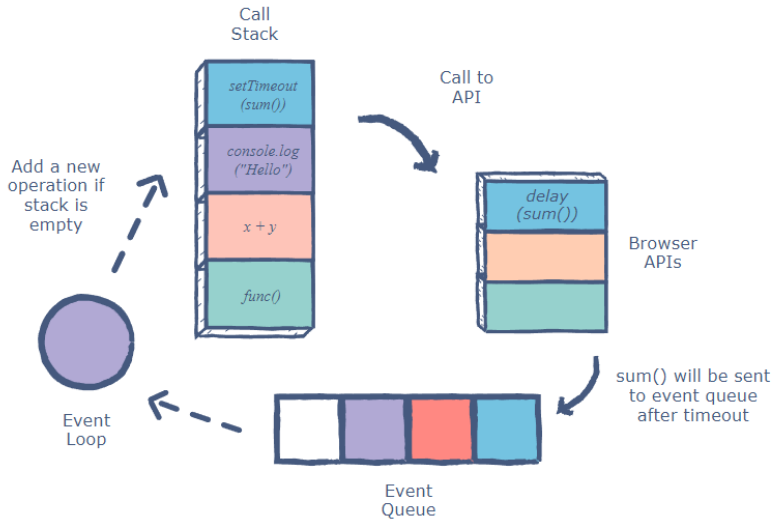
```
}

// Using the callback to handle the result
fetchData(function(result) {
  console.log(result);
});
```

Problemi koji se javljaju prilikom korišćena asinhronog programiranja su CallbackHell (ugnježdavanje callback funkcija), kontrola grešaka koristeći callback fje i debugging.

20. Event loop

Event loop je mehanizam koji omogućava neblokirajuće asinhrono izvršavanje programa u jednonitnom okruženju. Kada je CallStack prazan, event loop će iz Event Queue-a ubaciti događaj. Pozivom asinhrone metode poziva se Browser API. Kada prođe određeno vreme (u slučaju setTimeout funkcije (u funckiji setTimeout je minimalno vreme, a ne garantovano)) callback funkcija će biti dodata u event queue. Nakon što se CallStack isprazni, event loop će dodati funkciju iz event queue-a na callstack.



21. Promise, async await

Promise je objekat koji predstavlja eventualno izvršenje ili neuspeh izvršenja asinhrone operacije. Ima tri stanja: pending, fulfilled i rejected.

```
const fetchData = new Promise((resolve, reject) => {
  // Asynchronous operation
  setTimeout(() => {
    const data = 'This is some data.';
    resolve(data); // Resolve the promise with the data
    // or
    // reject('Error occurred'); // Reject the promise with an error
  }, 2000);
});
```

```
// Consuming the Promise using then and catch
fetchData
  .then(result => {
    console.log(result);
  })
  .catch(error => {
    console.error(error);
  });
```

Async await je sintaksni šećer nad Promisima. Async funkcija vraća promis, dok je await ključna reč koja se koristi kako bi se sačekala rezolucija Promisa.

```
async function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      const data = 'This is some data.';
      resolve(data);
    }, 2000);
  });
}

async function fetchDataAndLog() {
  try {
    const result = await fetchData();
    console.log(result);
  } catch (error) {
    console.error(error);
  }
}

fetchDataAndLog();
```

22. Spread operator

Spread operator (…) je JS operator koji omogućava razlaganje elemenata iz niza, objekta ili drugih iterables u neki drugi. Koristi se za kreiranje kopija, kombinovanje nizova itd.

```
// 1. Copying Arrays
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];

console.log(copiedArray); // Outputs: [1, 2, 3]
console.log(originalArray === copiedArray); // Outputs: false

// 2. Combining Arrays
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const combinedArray = [...array1, ...array2];

console.log(combinedArray); // Outputs: [1, 2, 3, 4, 5, 6]

// 3. Passing Arguments to Functions
function addNumbers(a, b, c) {
  return a + b + c;
}

const numbers = [1, 2, 3];
const sum = addNumbers(...numbers);
```

```
console.log(sum); // Outputs: 6

// 4. Copying Objects
const originalObject = { name: 'John', age: 30 };
const copiedObject = { ...originalObject };

console.log(copiedObject); // Outputs: { name: 'John', age: 30 }
console.log(originalObject === copiedObject); // Outputs: false

// 5. Merging Objects
const object1 = { name: 'John' };
const object2 = { age: 30 };
const mergedObject = { ...object1, ...object2 };

console.log(mergedObject); // Outputs: { name: 'John', age: 30 }

// 6. Rest Parameters in Functions
function sumNumbers(...numbers) {
  return numbers.reduce((sum, num) => sum + num, 0);
}

console.log(sumNumbers(1, 2, 3, 4)); // Outputs: 10

// 7. Cloning Arrays with Push
const originalArray2 = [1, 2, 3];
const clonedArray2 = [];

clonedArray2.push(...originalArray2);

console.log(clonedArray2); // Outputs: [1, 2, 3]

// 8. String to Array
const str = 'hello';
const charArray = [...str];

console.log(charArray); // Outputs: ['h', 'e', 'l', 'l', 'o']
```

Type Script

1) Typescript

TypeScript je programski jezik razvijen od strane Microsofta. Predstavlja superset JavaScripta, što znači da je svaki validan JS kod ujedno i validan TS kod. TypeScript uvodi statičku tipizaciju u JS, što omogućava hvatanje grešaka u compile-time, a ne u runtime-u. TS kod se u fazi prevođenja prvo prevodi u JS kod.

Definicija funkcije:

```
function add(a: number, b: number): number {
    return a + b;
}
```

Enumeracije:

```
enum Color {
    Red,
    Green,
    Blue,
}

let myColor: Color = Color.Green;
```

2) TS interfejsi, alijasi, poređenje tipova, unija

Interfejs služi za definisanje strukture objekta. Razlika između interfejsa i klase u TS je u svrsi koju služe. Interfejs se primarno koristi za definisanje strukture objekta, definišući „ugovor“ koji opisuje svojstva, ali ne i implementaciju. Interfejsi ne mogu biti direktno instancicrani, za razliku od klasa. Prilikom provere da li neki objekat zadovoljava uslove nekog interfejsa vrši se provera da li se svaki attr iz interfejsa nalazi u konkretnom objektu. Duck typing (if it walks like duck and it quaks like duck then it must be duck). Objekat će biti tretiran i kao da ispunjuje uslove interfejsa, ako pored ugovora dodaje jos neke dodatne stvari.

```
interface Person {
    name: string;
    age: number;
    greet(): string;
}
```

U TS alijasi se koriste za kreiranje custom tipova. Za kreiranje alijasa koristi se ključna reč type iza koje se navodi naziv novog tipa. Pr. type Centimeters = number;

Alijasi se mogu primenjivati i nad primitivnim i nad korisničkim tipovima. Pr.

```
type Point = {
    x: number;
    y: number;
};

type ID = string | number;

type Status = "active" | "inactive";

// Function type alias
type MathOperation = (a: number, b: number) => number;
```

```
// Union type alias
type Result = number | string;
```

U TS unije predstavljaju način da se predstavi vrednost koja može biti više različitih tipova. Kreira se korišćenjem pipe | simbola.

Poređenje tipova odnosi se na mogućnost da se proveri da li je određena vrednost nekog tipa. Načini za proveru da li je neka vrednost/promenljiva određenog tipa:

1) Typeof – if (typeof value === „number“)

2) Instanceof – if (animal instanceof Dog)

3) Type assertion -
```
let value: any = "Hello, TypeScript!";
let length: number = (value as string).length;
```

3) Typescript klase

```
[1]  class Car {
[2]      // Public members are accessible from outside the class
[3]      public readonly brand: string;
[4]
[5]      // Private members are only accessible within the class
[6]      private readonly fuel: string;
[7]
[8]      // Protected members are accessible within the class and its
   subclasses
[9]      protected speed: number;
[10]
[11]     // Static member shared among all instances of the class
[12]     static totalCars: number = 0;
[13]
[14]     // Constructor with parameters
[15]     constructor(brand: string, fuel: string) {
[16]         this.brand = brand;
[17]         this.fuel = fuel;
[18]         this.speed = 0;
[19]         Car.totalCars++; // Incrementing the static member
[20]     }
[21]
[22]     // Public method
[23]     accelerate(): void {
[24]         this.speed += 10;
[25]         console.log(`Accelerating. Current speed: ${this.speed}
   km/h`);
[26]     }
[27]
[28]     // Private method
[29]     private refuel(): void {
[30]         console.log(`Refueling with ${this.fuel}`);
[31]     }
[32]
[33]     // Protected method
[34]     protected honk(): void {
[35]         console.log("Honk! Honk!");
[36]     }
[37]
[38]     // Public method using private and protected members
[39]     startCar(): void {
```

```
[40]         this.refuel(); // Accessing private method
[41]         this.honk();   // Accessing protected method
[42]         console.log(`${this.brand} car is ready to go!`);
[43]     }
[44]
[45]     // Static method that can be called on the class itself
[46]     static getTotalCars(): number {
[47]         return Car.totalCars;
[48]     }
[49] }
[50]
[51] // Creating instances of the class
[52] const car1 = new Car("Toyota", "Gasoline");
[53] const car2 = new Car("Honda", "Electric");
[54]
[55] // Accessing readonly property
[56] console.log(car1.brand); // Output: Toyota
[57]
[58] // Attempting to modify readonly property (results in a compilation
   error)
[59] // car1.brand = "Nissan"; // Error: Cannot assign to 'brand' because
   it is a read-only property.
[60]
[61] // Accessing static member
[62] console.log(Car.getTotalCars()); // Output: 2
[63]
[64] // Calling instance methods
[65] car1.accelerate();
[66] car2.accelerate();
[67]
[68] // Accessing instance property
[69] console.log(car1.speed); // Output: 10
[70]
[71] // Calling static method
[72] console.log(Car.getTotalCars()); // Output: 2
```

4) Typescrip nasleđivanje

```
[1]  // Interface representing behaviors of an animal
[2]  interface Animal {
[3]      name: string;
[4]      makeSound(): void;
[5]  }
[6]
[7]  // Abstract class providing a base for animals
[8]  abstract class BaseAnimal implements Animal {
[9]      constructor(public name: string) {}
[10]
[11]     // Abstract method that must be implemented by subclasses
[12]     abstract makeSound(): void;
[13]
[14]     // Common method with implementation
[15]     sleep(): void {
[16]         console.log(`${this.name} is sleeping.`);
[17]     }
[18] }
[19]
```

```
[20] // Concrete classes for specific types of animals
[21]
[22] class Lion extends BaseAnimal {
[23]     makeSound(): void {
[24]         console.log(`${this.name} roars loudly.`);
[25]     }
[26] }
[27]
[28] class Elephant extends BaseAnimal {
[29]     makeSound(): void {
[30]         console.log(`${this.name} trumpets.`);
[31]     }
[32] }
[33]
[34] class Penguin extends BaseAnimal {
[35]     makeSound(): void {
[36]         console.log(`${this.name} makes a honking sound.`);
[37]     }
[38]
[39]     // Additional method specific to Penguins
[40]     swim(): void {
[41]         console.log(`${this.name} is swimming.`);
[42]     }
[43] }
[44]
[45] // Function demonstrating polymorphism with animals
[46] function interactWithAnimal(animal: Animal): void {
[47]     console.log(`Interacting with ${animal.name}:`);
[48]     animal.makeSound();
[49]     animal.sleep();
[50]
[51]     // Check if the animal can swim (using type assertion)
[52]     if ((animal as Penguin).swim) {
[53]         (animal as Penguin).swim();
[54]     }
[55]
[56]     console.log(); // Empty line for clarity
[57] }
[58]
[59] // Creating instances of different animals
[60] const lion = new Lion("Leo");
[61] const elephant = new Elephant("Ellie");
[62] const penguin = new Penguin("Penny");
[63]
[64] // Interacting with different animals
[65] interactWithAnimal(lion);
[66] interactWithAnimal(elephant);
[67] interactWithAnimal(penguin);
```

5) TS Šabloni

Generičke funckije:

```
[1]  // A generic function that echoes the input value
[2]  function echo<T>(value: T): T {
[3]      return value;
[4]  }
[5]
```

```
[6]  // Usage
[7]  let result: string = echo("Hello, TypeScript!");
[8]  let anotherResult: number = echo(42);
```

Generičke klase:

```
// Abstract class representing an Animal
abstract class Animal {
    constructor(public name: string) {}

    abstract makeSound(): void;
}

// Concrete class extending Animal
class Lion extends Animal {
    makeSound(): void {
        console.log(`${this.name} roars.`);
    }
}

// Generic container class
class Container<T extends Animal> {
    private items: T[] = [];

    addItem(item: T): void {
        this.items.push(item);
    }

    displayItems(): void {
        console.log("Items in the container:");
        this.items.forEach((item) => {
            console.log(`- ${item.name}`);
            item.makeSound();
        });
    }
}

// Usage
const lion1 = new Lion("Simba");
const lion2 = new Lion("Nala");

// Creating a container for Lions
const lionContainer = new Container<Lion>();
lionContainer.addItem(lion1);
lionContainer.addItem(lion2);

// Displaying items in the Lion container
lionContainer.displayItems();
```