1. WWW - osnove (HTTP, HTML, URL)

Web je servis interneta. Predstavlja mrežu dokumenata, to jest resursa, povezanih preko hiperteksta, odnosno specijalnog teksta koji sadrži reference, takozvane hiperlinkove, na druge tekstove i Web dokumente. Resursi na Web-u su identifikovani pomoću URL-a (uniform resource locator). Resursi na web-u i njegovi dokumenti prenose se na način definisan pravilima HTTP-a (hypertext transfer protocol). Resursima se pristupa preko aplikacija koje se nazivaju web browser-i. Preko njih, ostvaruje se veza sa serverom na kome su skladišteni resursi i koji je u stanju da ih šalje klijentima. Osnovni element web-a su stranice koje su kreirane pomoću markup jezika HTML-a. Html sadrži strukturu i sadržaj strane. (Više o HTML-u u posebnom pitanju br. 9).

URL predstavlja jedinstveni identifikator resursa na Web-u. Preko njega se navodi način transfera resursa (protokol), kao i njegova lokacija na Web-u. Razmotrimo sledeći primer:

<http://www.asdf.com/abcd.html>

U ovom primeru, http je protokol koji se koristi za razmenu (više o HTTP-u u posebnom pitanju br. 8). Zatim, nakon www, sledi ime hosta, odnosno mašine na kojoj se nalazi resurs, a .com predstavlja domen u kome je taj resurs registrovan (domen – ’porcija’ interneta koja može da obuhvata druge, manje porcije ili da sadrži računare). Nakon toga sledi ime dokumenta koji se zahteva. Ukoliko se navodi samo ime, to znači da se zahtevani resurs nalazi u root folderu računara kome se pristupa. To ne mora uvek da bude slučaj, tako da je moguće umesto imena navesti putanju od root foldera do željenog resursa. (’Root folder’ nije pravi root folder u celokupnom fajl sistemu mašine na kojoj se resurs nalazi, već se pod ’root’ misli na početni, vršni folder koji je na toj mašini namenjen za skladištenje Web resursa.

1. Web aplikacija

To je aplikacija kojoj klijenti pristupaju pomoću web browser-a. Za aplikacije se vezuje postojanje serverske strane i klijentske strane.

Serverska strana se može sastojati iz nekoliko slojeva. Prvi server kome se pristupi kada se izvrši bilo kakva akcija u browseru je Web server, čija je glavna namena da vrši komunikaciju sa klijentom koristeći HTTP protokol. On prima zahteve klijenta (HTTP request), obrađuje ih i vraća nazad odgovor (HTTP response).

Drugi sloj servera predstavlja aplikacioni server, koji se bavi logikom same aplikacije i može u pozadini vršiti sve stvari vezane za sam rad Web aplikacije, kao što su održavanje mreže, slanje poruka, obrada ulaza i izlaza. Ovo je server na kome je aplikacija koje je kreirana zapravo pokrenuta, i na njemu se izvršavaju njeni izvršni delovi. Takođe, na ovom moguće je generisati stranicu za korisnika. Takva stranica se zatim isporučuje Web serveru čija je jedna od uloga da je isporuči klijentu.

Poslednji sloj predstavlja server baze podataka, odnosno DBMS server, čija je namena da obrađuje upite i manipuliše realnim podacima koji su skladišteni u bazi.

Klijentska strana izvršava se na Web browser-u. Danas, moguće je da se deo logike prikaza prebaci na klijenta, koji bi pomoću skripti pribavljenih od servera mogao da ažurira prikaz. Svakako, primarna namena ove strane je da pruža korisniku interfejs ka samoj aplikaciji, a da sa druge strane komunicira sa Web serverom.

1. Desktop vs mobilne vs web aplikacije

Desktop i mobilne aplikacije su vezane striktno za platformu, dok web aplikacije nisu. Za desktop i mobilne, potrebna je instalacija, dok se web aplikacije izvršavaju u browser-u i ne zahtevaju da budu instalirane. Ažuriranje mobilnih i desktop aplikacija zahteva skidanje neophodnih instalacija i njihovo instaliranje, dok se web aplikacije takoreći samo-ažuriraju iz ugla korisnika, s obzirom na to da on kroz web pretraživač pristupa najnovijoj verziji aplikacije, ažuriranoj na serveru na kome je aplikacija smeštena.

Desktop i mobilne aplikacije ne zahtevaju uvek aktivnu internet konekciju, ali je ona neophodna za rad web aplikacije, s obzirom na to da se komunikacija klijenta i servera odvija upravo preko internet mreže. Naravno, određene desktop i mobilne aplikacije, koje se koriste za komunikaciju na daljinu, takođe zahtevaju internet konekciju.

Desktop i mobilne imaju bolje multimedijalne performanse, i brže se izvršavaju. Web aplikacije su sporije, odnosno odziv web aplikacije je generalno sporiji, s obzirom na to da svaka korisnička akcija zahteva u pozadini i neku razmenu podataka između klijenta i servera.

Web aplikacije imaju ograničen GUI u odnosu na mobilne i desktop aplikacije. Sa druge strane, web aplikacijama se lakše dodaje multimedijalni sadržaj.

Desktop aplikacije su namenje za izvršavanje na računaru ili laptopu i vezane su za platformu i operativni sistem, a mogu zahtevati i određene hardverske resurse da bi mogle da rade. Mobilne su namenje za mobilne telefone i tablete i kreiraju se za određeni mobilni OS (iOS, Android). Instalirane su na samom uređaju i imaju pristup resursima uređaja, npr GPS ili kamera. Nabavljaju se preko AppStore-ova. Mora se obraćati pažnja na veličinu ekrana koje ove aplikacije zauzimaju, zbog ograničene veličine istog kod mobilnih uređaja. Web aplikacije se prilagođavaju svim uređajima. Zbog responsive design-a, web aplikacije danas nalikuju native mobillnim aplikacijama.

Web aplikacije, zbog načina na koji se izvršavaju, nemaju pristup lokalnim podacima i sistemu korisnika, već za svrhe skladištenja mora da se predvidi baza (recimo na serveru koji pokreće aplikaciju, na posebnom serveru, distribuirana u cloud-u).

Glavne razlike između desktop i mobilnih aplikacija proističu iz osobina uređaja za koje su namenjene. Obe vrste aplikacija zavisne su od operativnog sistema uređaja. Desktop aplikacije su često lakše i intuitivnije za korišćenje, s obzirom na veličinu ekrana na kojoj se prikazuju. Takođe, snaga samih desktop mašina pruža mogućnost za izvršavanje kompleksnijih operacija u odnosu na mobilne aplikacije. Uz to, danas je neophodno brinuti o činjenici da mobilne aplikacije treba prilagoditi i vertikalnom i horizontalnom prikazu. Mobilne aplikacije pružaju mogućnost specifičnih funkcionalnosti koje računari nemaju, kao što je na primer rad sa više kamera.

1. Arhitektura web aplikacija (dijagram, cloud)

Arhitektura Web aplikacije opisana je u pitanju broj 2. Ovde će biti pomenute tehnologije za određene slojeve, biće malo detaljnije opisan dijagram dat na slici, i biće reči o cloud-u.

Prezentacioni sloj klijenteske strane je sloj koji je najbliži korisniku, odnosno onaj sa kojim korisnik interaguje. Za pružanje prikaza korisniku, najvažnije su sledeće tri tehnologije:

* HTML za definisanje same strukture stranice, odnosno elemenata koje će stranica da sadrži i primarnog rasporeda tih elemenata u smislu toga ko je roditelj, a ko su deca određenog elementa; ovaj raspored definiše se ugnježdavanjem HTML tagova
* CSS za kreiranje takozvanih stylesheet-ova, odnosno stilskih pravila koja će važiti za HTML elemente; CSS-om se definiše sam izgled stranice, u pogledu rasporeda, veličine elemenata, boje, veličine teksta i ostalih stvari vezanih za stilizovanje stranica
* JavaScript za definisanje ponašanja elemenata u odnosu na korisnikove akcije ili promene u bazi

Za „dostavljanje“ elemenata prezentacionog sloja korisniku, zadužen je specijalni softver – Web browser (pretraživač). Neophodno je da pretraživači budu instalirani na klijentskoj mašini, jer se preko njih vrši komunikacija sa serverom. Najpoznatiji predstavnici među pretraživačima su Google Chrome, Mozila Firefox, Opera, Safari, Microsoft Edge.

Za ispravan rad klijentske strane koristi se i lokalni memorijski prostor klijentske mašine. Iako se sav originalan sadržaj nalazi na serveru, radi efikasnog rada i povećanih performansi neophodno je da klijent povremeno skladišti (kešira) stvari pristigle sa servera. Primer može biti skladištenje i cele stranice, ali najpoznatiji primer jesu kolačići (cookies). Cookies predstavljaju podatke koji se smeste u lokalnu memoriju klijenta nakon pristupanja nekom web sajtu ili aplikaciji. Nakon ponovnog pristupa tom sajtu, po memoriji se traže kolačići vezani za taj sajt, na osnovu kojih sajt može da zna kom sadržaju je korisnik pristupao i da takav sadržaj stavlja u prvi plan na stranicama. Kolačići mogu na primer sadržati informaciju o tome da li je korisnik bio prijavljen i da li ima potrebe da to radi ponovo u sledećem pristupu web sajtu ili aplikaciji. Takođe, lokalno skladište može se koristiti kada je potrebno da korisnik vrši trasfer fajlova preko stranica (recimo, HTML za ove svrhe pruža poseban tip input taga, „type=file“, dok JavaScript pruža posebne funkcije za rad sa fajlovima).

Na dnu klijentske strane, naravno, nalazi se operativni sistem mašine.

Što se tiče servera, on se takođe sastoji iz nekoliko važnih celina. Jedna od njih je logički sloj, koji je zadužen za održavanje biznis-logike same aplikacije i za vršenje obrade podataka iz baze, kako u smeru KA bazi (obrada podataka radi smeštanja, upisa u bazu i ažuriranje baze), tako i u smeru OD baze (obrada i pribavljanje podataka na osnovu kojih se generišu komponente prezentacionog sloja koje se šalju ka klijentu). Ovaj sloj predstavlja odgovornosti aplikacionog servera pomenutog u pitanju broj 2. U svrhe ispunjavanja dužnosti ove logike, moguće je koristiti razne tehnologije – pisanje PHP skripti, korišćenje programskih jezika Perl i Python, ili nekih novijih tehnologija kao što je ASP .NET Core.

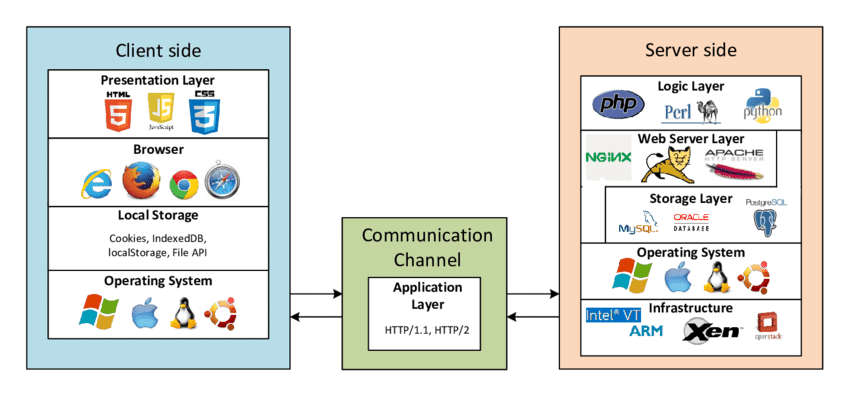
Sledeća serverska celina predstavlja sloj Web servera. Njegove osnovne dužnosti su komunikacija sa klijentom putem HTTP protokola. Danas se u ove svrhe najčešće koriste usluge poznatih web-hosting kompanija. Najpoznatiji Web serveri su Apache, Apache Tomcat, Ngnix. Zakupljivanjem web-hosting usluga nekog od ovih servera, kompletna logika ovog serverskog sloja poveri se već razvijenim kompanijama, koje pružaju veliki broj snažnih računara, tako da je ovaj pristup znatno efikasniji od kreiranja i pokretanja servera na lokalnoj mašini na kojoj je razvijena i aplikacija. Takođe, ako se koriste web-hosting usluge, developer nema potrebu da razmišlja o saobraćaju i o load-balancing tehnikama (više o load-balancing-u u pitanju broj 6).

Za skladištenje i perzistenciju podataka zadužen je sloj skladištenja. Ovaj sloj takođe podrazumeva i DBMS komponentu koja direktno manipuliše bazom. U svrhe skladištenja koriste se neki od poznatih DBMS-ova, kao što su Oracle DBMS, MySQL, PostgreSQL, SQL Server...

Infrastruktura servera predstavlja server u najnižem, hardverskom pogledu. Od velikog je značaja da ova infrastruktura bude jaka i izdržljiva, s obzirom na to da se su serveri često uska grla u čitavom sistemu Web aplikacije, i da moraju da obrađuju veliki broj pristupa velikom brzinom, kao i da obezbede sigurnost podataka.

Cloud, odnosno princip korišćenja „računarstva u oblaku“, omogućuje razvojnom timu aplikacije (pojedincu, grupi programera, firmi) da čitavu serversku stranu premeste na skup računara lociranih u velikim data-centrima, rasprostranjenim po svetu. Time se izbegava potreba da serverske komponente budu pokrenute na lokalnim računarima, koji zasebno imaju mnogo manju moć nego skup računara u oblaku. Premeštanjem servera na udaljene računare, sav serverski saobraćaj i serverska logika same aplikacije kreće da se izvršava na računarima koji imaju veliku moć. Ovaj princip se odnosi i na bazu podataka, a smeštanjem servera u oblak gubi se potreba i za jakom lokalnom serverskom infrastrukturom, s obzirom na to da lokalni računari više nisu zaduženi za serverske poslove.

Konačno, na dijagramu (na sledećoj strani) vidimo i komponentu aplikacionog sloja. Bitno je naznačiti da ova komponenta ne može da se poistoveti sa aplikacionim serverom. Aplikacioni server, odnosno njegova zaduženja, su na dijagramu predstavljena slojem „Logic layer“. Aplikacioni sloj sa dijagrama predstavlja najviši sloj TCP/IP modela, koji se ogleda u postojanju protokola visokog nivoa, za razmenu podataka između aplikacija. Najvažniji protokol ovog sloja, što se tiče Web aplikacija, svakako je HTTP protokol.



1. Klijent strana (tehnologije, izazovi)

Na klijentskoj strani, imamo HTML kao jezik koji definiše strukturu I sadržaj strane, CSS koji definiše raspored I izgled elemenata I JavaScript koji je zadužen za ponašanje stranice.

Izazovi su:

* Korisničko iskustvo (user experience) koje podrazumeva svaki vid interakcije koji korisnik ima sa aplikacijom, odnosno način na koji korisnik doživljava proizvod, ali i kompaniju koja stoji iza nje; prosto rečeno, da li je korisniku proizvod dopada, ne samo po izgledu, već u svakom mogućem pogledu (performanse, usluge koje pruža, izgled, način na koji su pisane rečenice i tekstovi na stranicama aplikacije...)
* Korisnički interfejs (user interface); UI je takoreći podskup user experience-a; pretežno se svodi na to kako će korisnik da doživi aplikaciju u pogledu njenog izgleda i lakoće korišćenja
* Performansa; važno je da se klijentske performanse povećaju što je moguće više, kako bi korisnici bili zadovoljni proizvodom (na primer, brzina može da se poveća keširanjem podataka na klijentu)
* Cross-browser i cross-platform izvršavanje; ovi izazovi vezani su za činjenicu da je aplikaciju potrebno da podrže svi mogući browser-i i sve platforme (ako je to moguće)
* Mnoštvo tehnologija koje vrlo brzo napreduju. Klijentska strana na početku se ogledala u statičkim web stranama, koje nisu menjale izlgled i korisnik sa njima nije mogao da interaguje, već samo da pristupa sadržaju. One su se menjale jedino ako bi ih neko ručno prepravio. Dinamičke web strane mogu da menjaju prikaz u zavisnosti od korisnika i mogu da izgledaju drugačije svaki put kada se učitaju, u zavisnosti od određenih podataka. Dalji napredak su asinhrone strane, gde se kontrola može vratiti korisniku odmah nakon izdavanja zahteva, tako da server može izvršavati zahtev dok klijentska strana radi nešto drugo. SPA (single-page applications) su one aplikacije koje sadrže jednu stranicu čiji se sadržaj menja u skladu sa zahtevima korisnika, umesto da se učitavaju nove stranice. Kod SPA aplikacija neophodno je da se obezbedi efikasna manipulacija DOM-om (više reči o DOM-u biće kasnije 34.).

Nešto više o raznim klijentskim tehnologijama:

* Dojo toolbox predstavlja JavaScript framework, odnosno skup biblioteka koje omogućavaju razvoj cross-platform aplikacija koristeći tehnike AJAX-a.
* JQuery je JavaScript biblioteka koja je donela veliko olakšanje u kreiranju dinamičkih web strana. Zapravo, JQuery objedinjuje određene mogućnosti koje pruža JavaScript u funkcije, tako da je znatno lakše manipulisati DOM-om, animacijama, event-ima...
* Node.js zapravo predstavlja serversku komponentu koja se koristi za pokretanje koda pisanog u JavaScript-u. Ono što je inovativno je što je Node.js prva tehnologija koja je pružila mogućnost kreiranja „push“ logike aplikacije, gde je server u stanju da pokrene komunikaciju i time obavesti klijente o promenama. Zbog toga je važan i za klijente, s obzirom da uklanja potrebu za neprestanim slanjem zahteva serveru radi provere eventualne promene sadržaja.
* Angular.js predstavlja klijentsku komponentu koja se koristi za kreiranje reaktivnih, SPA aplikacija. Koristi TypeScript (više o njemu kasnije 30., 31., 32., 33.) za definisanje ponašanja stranica.
* Backbone.js je JavaScript biblioteka zasnovana na JSON predstavljanju podataka. Modeli podatak se predstavljaju pomoću key-value parova. Ova tehnologija pruža RESTful JSON interfejs ka nekom korisnikovom API-ju.
* React je frontend tehnologija koja pruža način za kreiranje reaktivnih single-page aplikacija. Popularan je jer pruža mogućnost kreiranja UI komponenata, koje mogu da se koriste i prilagode raznim stranicama.
* Web assmebly (pitanje broj 66)

1. Server strana (tehnologije, izazovi)

Serverska strana ima logički sloj za samu logiku izvršavanja aplikacije (ASP.NET Core, Php, Node.js). Web server je sloj serverske strane zadužen za komuniciranje sa klijentom, prihvatanje i prosleđivanje zahteva (Nginx, Apache). Sloj za skladištenje predstavlja sloj baze podataka (relacione baze, NoSql, in-memory, graf).

Server se može širiti horizontalno – dodaje se novi računar, ili vertikalno – unapređuje se već postojeći računar.

Izazovi vezani za ovu stranu su:

* Skalabilnost – mnogo korisnika može zahtevati pristup serveru. Pristup ovom problemu može biti kreiranje klastera više servera koji se mogu predstavljati klijentima kao jedan server i izvršavaće funkcije jednog servera, samo mnogo brže i pouzdanije.
  + Usko vezan za klastere je princip load balancing-a gde se klijentski saobraćaj distribuira između više servera koristeći razne politike odlučivanja i algoritme. Izazov za server
* Sigurnost – server treba klijentima da garantuje sigurnost njihovih podataka, i da zabrani klijentima da pristupaju podacima za koje nemaju pravo pristupa. Imamo problem autorizacije (davanja prava korisnicima), autentifikacije (predstavljanje korisnika jedinstevnim imenom i šifrom vezanom za to ime (username i password)), zaštita od sql injection-a (ubacivanja dodatnih zahteva u upit od neovlašćene strane kojom ona dobija skrivene i tajne podatke), DDOS-distributed denial of service.
  + Autorizacija predstavlja davanje prava određenim korisnicima za pristup i manipulaciju samo određenim skupom podataka. Jedan od poznatih pristupa ovom problemu jeste OAuth2.0 i takozvani access token-i. OAuth2 podrazumeva da se korisniku pri logovanju na sistem dodeljuje takozvani token. Sam token nosi informaciju o tome da li onaj ko ga poseduje može da koristi određeni API za pristup podacima. Token u osnovi ne zna kome pripada, i to mu nije važno, ali je moguće da u token budu ugrađene i informacije o korisniku, ako API to zahteva (npr. da sadrži ID korisnika da bi API mogao da vrati ime i prezime korisnika sa tim ID-jem i slično).
  + Pored autorizacije i dozvole pristupa,najčešće je neophodno znati kome se token autorizacije (token koji daje OAuth2.0) uopšte može dodeliti. To se postiže sistemima za autentifikaciju, od kojih je najpoznatiji OpenID. On podrazumeva prisustvo takozvanog third-party servisa. Taj servis pružaju OpenID provider-i. Korisnici se registruju korisničkim imenom i šifrom kod provider-a, uz eventualne dodatne podatke i taj provider je jedini koji zna korisničku šifru. Zatim, taj provider daje potvrdu određenom skupu sajtova (takozvani Relying Parties) koji „veruju“ tom provider-u, odnosno pružaju mogućnost prijave korisnika preko naloga kreiranih kod provider-a. Kada korisnik pristupa sajtu, bira način logovanja preko određenog provider-a, koji zatim pruža sajtu potvrdu validnosti podataka koje je korisnik uneo. Tek nakon toga, pomoću OAuth2.0 protokola, korisniku se izdaje token za pristup podacima.
  + SQL injection predstavlja situaciju u kojoj korisnici, umesto očekivanog unosa koji se negde na sajtu od njih zahteva, unesu SQL naredbu, ili njen deo, koja se zatim (putem recimo PHP skripti) izvrši i time dozvoli korisniku neovlašćen pristup bazi. Za sprečavanje ovakvih napada, koristi se pojačana provera važnih input polja (kao što su polja za username i passwod), odnosno njihova validacija, parametrizovani upiti, ili druge tehnike kojima se menja standardni izgled SQL naredbi, kako prosto ubacivanje dela SQL naredbe ne bi narušilo sigurnost (na primer, ukoliko imamo naredbu „SELCT \* FROM TABELA WHERE ATRIBUT = “ + vrednost, odnosno SQL naredbu zadatu preko stringa i konkatenacije, na mesto parametra ’vrednost’, koja se uzima iz nekog input polja, moguće je prolsediti „5 AND ATRIBUT2 = 10“. Na ovaj način korisnik je vrlo prosto proširio SQL naredbu, što ne bi smelo da mu bude dozvoljeno. Parametrizovani upiti, zbog specijalnog načina na koji se izvršavaju (vrši se određeni vid prekompajliranja same naredbe, pre umetanja parametara), sprečavaju da atribut postane deo SQL naredbe, tako da bi u prethodnom primeru ceo string prosleđen preko input polja zapravo bio posmatran kao parametar i ne bi se izvršio kao deo naredbe.)
  + DDOS predstavlja situaciju u kojoj korisnici namerno preopterećuju server, kako bi on iskoristio sve raspložive resurse i time prestao da funkcioniše. Postoje tri tipa ovog napada. Prvi su **Volumetric Attacks** gde klijent učestalo šalje veliki broj lažnih, narušenih paketa mrežnog mrežnog (ICMP) ili transportnog nivoa (UDP), pa server mora da se bavi njima, i na kraju ih odbaci, ali je u međuvremenu nedostupan za ostale, prave klijente. Drugi tip su **napadi aplikacionog sloja**, gde se šalje veliki broj zahteva preko nekog od najviših, aplikacionih protokola (HTTP, SMTP...). Treći tip su takozvani **Protocol Attacks** koji koriste specifičnosti određenih protokola i pravila uspostavljanje konekcije kod njih. Recimo, kod TCP konekcije postoji 3-way handshake procedura, gde klijent šalje takozvanu SYN (sinhronizacionu) poruku, serve odgovara sa SYN ACK, nakon čega klijent šalje ACK i veza je uspostavljena. Protocol Attacks koriste ovu proceduru, i šalju lažne SYN poruke, na koje server odgovori i čeka ACK koji nikada ne stigne. Server je za to vreme već odvojio deo resursa za napadača, pa ako se to ponovi više puta, server može biti iscrpljen. Problem se rešava ograničavanjem broja postojećih konekcija u istom trenutku, filtriranjem nepoželjnih IP adresa, ili rezervisanjem resursa tek nakon što konekcija bude sigurno uspostavljena tj. takozvanim zakašnjenjem alokocije resursa.

1. Protokoli (HTTP 1.0, 1.1, 2.0, Websocket)
   1. Http 1.0 je protokol bez pamćenja stanja (statelss). Server ne pamti podatke o korisniku. HTTP se zasniva na sesijama između klijenta i servera, gde se klijent poveže sa serverom otvaranjem konekcije i slanjem zahteva koji će server da opsluži. Radi pamćenja stanja, koriste se cookies – delovi fajla ili kodovi koje server pošalje klijentu, pa klijent priloži serveru pri ponovnom povezivanju, da bi se sesija nastavila od tačke u kojoj je prekinuta. Cookies se klijentu pošalju u zaglavlju odgovora koji pošalje server. HTTP koristi takozvane request metode za slanje zahteva: GET, POST, PUT, DELETE, HEAD. HTTP protokoli omogućavaju da odgovor od servera sadrži razne vidove podataka, kao što su tekstualni fajlovi, html fajlovi, slike... U verziji 1.0, za svaki par zahtev/odgovor mora se otvoriti nova konekcija, što rad ovog protokola čini sporim.
   2. U verziji 1.1, novina je to što veza postaje perzistentna. Perzistentna veza znači da se zahtevi jednog klijenta ka jednom serveru upućuju preko iste konekcije, i da se ona zatvara kada je više nije potrebno koristiti. Takođe, uvodi se i pipelining – dozvoljava se da se u istom trenutku pošalje više zahteva. Neretko se desi da prvi zahtev koji stigne bude blokiran čekajući neku operaciju, čime blokira sve zahteve iza sebe. Ovaj problem zahteva da se često istovremeno otvori više TCP konekcija, što može biti problem jer serveri ograničavaju broj konekcija koje mogu da prihvate odjednom. Takođe, u verziji 1.1 postoji 24 status koda, dok ih u verziji 1.0 ima 16. Još, u verziji 1.0 nije postojala enkripcija korisničkog imena i šifre, niti je postojalo vreme važenja poruke koja sadrži ove informacije. Tako, kod verzije 1.0, bilo je moguće lako presresti poruke ove sadržine, pribaviti username i šifru i zatim koristiti tu istu poruku za neovlašćenu autentifikaciju mnogo nakon presretanja poruke. Kod verzije 1.1, uvodi se kodiranje ovih podataka, zatim se šalje specijalan one-time value kod klijentu na osnovu koga računa jedinstvenu check-sumu, a cela poruka sa ovim informacijama je validna samo određen vremenski period.
   3. U http 2.0 verziji, svi zahtevi se iz tekstualnog prebacuju u binarni format (iako se prvobitno kreiraju u tekstualnom), da bi se omogućilo da i starije aplikacije, nastale pre 2.0 verzije, mogu da funkcionišu i sa ovom verzijom. Posmatranje konekcija kao binarne omogućava da se kroz jednu TCP konekciju kreira više stream-ova podataka, gde će zahtevi putovati različitim stream-ovima, podeljeni na frame-ove u nižim slojevima konekcije, koji mogu putovati izmešani, ali se na osnovu nekih oznaka (tag-ova) uspešno reasembliraju na odredištu. Važna stvar je da je moguće slati različite podatke kroz više stream-ova istovremeno, preko jedne konekcije. To je prednost u odnosu na 1.1 verziju, jer je u njoj kroz jednu konekciju u jednom trenutku može slati samo jedan skup podataka, a jedini način za istovremeno slanje više njih u verziji 1.1 jeste otvaranje više TCP konekcija, što je skup proces. U ovoj verziji, moguće je da server samoinicijativno obaveštava klijenta o bitnim promenama, bez da klijent redovno opterećuje server slanjem zahteva za sadržajem (push model komunikacije). U 2.0, vrši se i kompresija zaglavlja metodom HPACK, kojom se zaglavlje smanji za nekoliko bajtova, koji prave značajnu razliku, imajući u vidu broj poruka i količinu saobraćaja.
   4. Websocketi – Kompatibilan HTTP serveru. Razlike u odnosu na HTTP konekcije je što je WebSocket konekcija full-duplex. Websocket handshake koristi HTTP Upgrade zaglavlje (deo u zaglavlju) za promenu protokola iz HTTP u Websocket. Omogućava interakciju između web čitača (ili druge klijentske aplikacije) i web servera sa manjim „troškovima“, što olakšava prenos podataka sa i na server. Takođe, iako je za ostvarivanje komunikacije potrebno izvršiti HTTP konekciju, pa je prevesti u WebSocket konekciju, Websocket je brži protokol od HTTP-a jer, nakon konektovanja, nema mnogo podataka koji se šalju, a ne predstavljaju traženi sadržaj (kao na primer zaglavlje odgovora u HTTP-u). WebSocket koristi UTF-8 kodiranje za predstavljanje podataka.

SignalR – implementacija websocket-a kreirana od strane .NET-a

(Profesor je pominjao dve dominantne implementacije websocket-a, ali ne znam koje su, ako neko zna neka dopuni).

1. HTTP – opis, metode zahteva, poruke odgovora

(Opis se u suštini može naći u prethodnom odgovoru).

Metode zahteva predstavljaju metode kojom se određuje kakav zahtev šalje klijent (da li za brisanje, za preuzimanje, izmenu podataka...). Metoda se navodi u zaglavlju HTTP zahteva. Metode su:

* GET metoda, čija je namena pribavljanje podataka
* HEAD metoda kojom se zahteva pribavljanje samo zaglavlja odgovora (dakle bez tela)
* POST metoda za slanje novog resursa na odredište (novih podataka); novi resurs se nalazi u telu HTTP zahteva
* PUT metoda kojom se postojeći podaci na odredištu menjaju podacima u telu zahteva; u suštini, predstavlja ažuriranje; PUT može da se koristi i za kreiranje resursa; razlika u odnosu na POST je što se PUT zahtev šalje na tačno određenu adresu resursa, pa ako na njoj ne postoji resurs, kreira se novi, a ako postoji, postojeći se menja novim; dakle, ako imamo adresu /persons, kojom se može pristupiti generalno nekom skupu osoba, PUT bi se morao poslati na /persons/personID, gde je personID identifikator konkretne osobe; ako ne postoji osoba sa tim ID-jem, kreira se nova, a ako postoji, ažurira se na nove vrednosti; za POST metodu, dovoljno je poslati zahtev na adresu /persons, bez ID-ja, čime se kreira nova osoba sa podacima navedenim u telu zahteva
* DELETE metoda za brisanje određenog resursa
* CONNECT metoda za uspostavljanje veze sa serverom na kome se nalazi navedeni resurs
* OPTIONS metoda za zahtevanje informacija o konekciji uspostavljenoj sa odredištem na kome se nalazi resurs
* TRACE metoda koja služi za praćenje putanje od klijenta koji zahteva resurs, do servera na kome se on nalazi; svaki čvor kroz koji se prođe na putu do servera sa resursom prima TRACE poruku i vraća klijentu odgovor sa statusom 200; ovo se ponavlja duž celog puta do odredišta, osim ako se ranije ne prekorači maksimalan broj skokova, u kom slučaju se slanje TRACE poruke prekida kod onog servera kod koga taj broj skokova dođe do nule (jer broj skokova kreće od maksimuma, pa se smanjuje usput)
* PATCH se koristi za *parcijalno* ažuriranje resursa; takođe se šalje na konkretnu adresu resursa, kao i PUT, ali PATCH nosi u telu samo nove vrednosti onih podataka koji će biti izmenjeni; PUT u telu sadrži sve podatke vezane za resurs, i ceo resurs menja se tim podacima; i PATCH može kreirati novi resurs ako već ne postoji na navedenoj adresi, ali se to ne preporučuje

Poruke odgovora su poruke kojima server šalje odgovor na zahteve klijenta. I one sadrže zaglavlje, na čijem se početku (u prvoj liniji) specificira HTTP verzija, zatim kod odgovora i opis koda odgovora. Kodovi mogu biti sledeći

* 1. Kodovi oblika 1xx su kodovi obaveštenja (npr, ako je telo zahteva veliko, moguće je da klijent prvo pošalje samo zaglavlje zahteva, u kome će navesti informacije o samomo zahtevu, recimo dužinu tela; u tom prvom zahtevu, klijent navodi i Expect: 100-continue, što je znak serveru da proveri zaglavlje i da pošalje odgovor „100 continue“ klijentu, ukoliko je spreman da prihvati telo; nakon toga, klijent šalje telo)
  2. Kodovi oblika 2xx označavaju uspešno obrađen zahtev (najpoznatiji je kod „sve je kako treba“, odnosno kod 200, iza koga kao opis najčešće sledi OK)
  3. Kodovi oblika 3xx su kodovi redirekcije (na primer kod 301 označava da je traženi resurs trajno pomeren na drugu lokaciju, pri čemu se u zaglavlju odgovora navodi i adresa nove lokacije)
  4. Oblik 4xx znači da je došlo do greške za koju je „kriv“ klijent (404 – file not found, 400 – bad request)
  5. Oblik 5xx kazuje da je došlo do greške na strani servera (505 – server ne podržava navedenu verziju http protokola

Nakon koda, u odgovoru slede Header Fields, kojima se šalju dodatne informacije o odgovoru. Neki od njih su

* Connection: da li se konekcija zatvara nakon trenutne komunikacije. Može imati vrednost „close“ ukoliko se zatvara, ili „keep-alive“ ukoliko ostaje otvorena.
* Content-length: dužina tela odgovora u bajtovima.
* Content-type: tip odgovora koji je stigao (text/html, text/plain, image/png, image/jpeg, video/mp4...)
* Date: datum slanja odgovora.
* Expires: datum kada ističe važenje odgovora.
* Location: lokacija traženog resursa, ukoliko se on ne nalazi na kontaktiranom serveru, ali kontaktirani server mu zna lokaciju. Najčešće ide uz kod 301.

Nakon ovog zaglavlja, sledi telo poruke, koje sadrži realne podatke.

1. HTML – opšte, struktura strane

Html je markup jezik (jezik označavanja) koji služi za definisanje strukture i sadržaja web strane. Gradivni elementi HTML strane definisani su tagovima, gde se ključna reč koja predstavlja naziv taga stavlja između znakova <>. HTML kod se render-uje u web pretraživaču, što u suštini znači da se skup tagova interpretira u ono što se vizuelno prikazuje u pretraživaču. Na osnovu tagova kreira se DOM stablo (biće kasnije pitanje samo o DOM-u 34.), koje predstavlja samu strukturu stranice i način na koji su povezani i ugnježdeni tagovi. Tag može imati svoj roditeljski tag (onaj u kome se nalazi) i može imati decu (tagovi unutar njega). Svaki deo stranice počinje tagom za otvaranje koji je oblika <imeTaga>, a završava se tagom za zatvaranje, </imeTaga>. Neki tagovi su samozatvarajući, kao recimo tag za sliku, pa se može navesti samo <img /> umesto para <img></img>. Svaka stranica počinje tagom <html> a završava se sa </html>. Unutar njega, postoje dva glavna dela HTML koda. Prvi je između tagova <head></head> i sadrži naslov, referencu na css i js fajlove itd. , a drugi je između tagova <body></body> i sadrži deo stranice koji se vizuelno prikazuje. Na početku HTML fajla nalazi se <!DOCTYPE> deklaracija, to nije tag već služi da specificira pravila po kojima je kreiran dokument koji sledi. Za HTML5 tu se jednostavno stavlja <!DOCTYPE html> međutim kada je u pitanju neka druga verzija HTML-a ili neki drugi tip fajla tada se potrebno navesti i odgovarajući DTD (Document Type Definition) koji sadrži definiciju strukture fajla i pravila koja ona mora da ispoštuje-

Tagovi ne moraju da se sastoje isključivo od imena. Moguće je manipulisati podrazumevanim svojstvima tagova, koristeći atribute. Oni se navode takođe unutar znakova <>, nakon imena taga, a vrednost im se dodeli pomoću znaka jednakosti. Na primer, moguće je dodeliti klasu tagovima, pomoću atributa class (<div class=“klasa“>).

1. HTML5 - semantički elementi

Od nesemantičkih elemenata u html5, postoje div i span, koji predstavljaju samo kontejnere, gradivne elemente koji sadrže druge elemente strane i koriste se za stilizovanje i lakše uređivanje strane. Div je blok element (smešta se u novi red, a element koji sledi takođe ide u novi red), a span je inline element (nalazi se u istoj liniji i sa prethodnim i sa narednim elementom, osim ako se ne navede drugačije).

Semantički elementi su elementi koji predstavljaju samostalni deo strane koji može da se distribuira i koristi (header, footer, article, aside, nav). Oni su „samoopisni“, odnosno na osnovu naziva je jasno i browser-u i developer-u šta element predstavlja i gde će se na strani prikazati.

* 1. Header i footer se nalaze na početku odnosno dnu strane.
  2. Section predstavlja određen deo strane koji sadrži grupu tematski povezanih jedinica. Na primer, može postojati sekcija „introduction“, sekcija „content“ i slično.
  3. Article treba da bude samostalno prikazan, nevezan isključivo za konkretnu stranu i predstavlja element sa konkretnim sadržajem (kao članak u novinama).
  4. Aside ima sadržaj, ali ne glavni sadržaj strane, već nešto što je za nju vezano i odnosi se na tu stranicu (na određen article element). Može biti u okviru artikla. Ne nalazi se nužno bočno od artikla, ali najčešće predstavlja neki vid sidebar-a.
  5. Details je tag koji predstavlja neki vid dodatnih informacija. Podrazumevano, element koji sadrži ove informacije moguće je otvarati i zatvarati na klik.
  6. Figure tag služi za predstavljanje celine posebnog tipa sadržaja, na primer slike, dijagrama, ilustracije i slično.
  7. Figcaption tag predstavlja opis vezan za figure tag.
  8. Main tag služi za predstavljanje glavnog sadržaja strane, koji ne bi trebalo da se ponavlja, i ne bi trebalo da sadrži elemente koji se generalno ponavljaju u dokumentu (sidebar, navbar, logo i slično).
  9. Mark tag služi da naglasi određeni deo teksta (podrazumevano, da oboji pozadinu tog dela teksta drugom bojom)
  10. Nav tag služi za predstavljanje skupa linkova, ali bi u nav tagu trebalo da se nađu samo glavni linkovi, odnosno osnovni skup elemenata za navigaciju (oni linkovi koji bi, na stilizovanoj strani, bili na vrhu strane, najupečatljiviji).
  11. Summary tag vezan je za Details tag. Kada se kreira details tag, pojaviće se strelica, čijim klikom se prikazuju informacije, a ponovnim klikom zatvaraju. Bez summary taga unutar details taga, pored strelice za otvaranje pisaće „Details“, a sa summary tagom, pored details će pisati tekst iz summary taga. (Na levoj slici je primer bez summary-ja, a na desnoj sa njim).

* 1. Time tag se koristi da naznači da je na određenom delu stranice prikazano vreme. Ne prikazuje se ni na kakav specijalan način, ali je moguće dodati atribut datetime=“15.04.2021 10:00“, kojim se navedeno vreme prevodi u zapis čitljiv mašini, i može da se iskoristi za kreiranje podsetnika, beleženje nekih datuma i slično.

1. CSS – opšte

CSS (Cascading Style Sheet) predstavlja jezik za formatiranje, koji se koristi za stilizovanje web stranica (mada može da se koristi i za stilizovanje drugih XML dokumenata). Pre CSS-a, stilizovanje HTML elemenata moglo se vršiti ili direktnom primenom stilova za tačno jedan tag, ili pisanjem stilova koji su se odnosili na samo jedan HTML dokument. Sa pojavom CSS-a, postalo je moguće definisati stilove u odvojenim fajlovima (.css fajlovima) i uključiti ih u proizvoljan broj HTML dokumenata, čime se postiže manje opterećivanje HTML dokumenata i smanjuje potreba za dupliranjem stilova. Stilizovanje se vrši tako što se unutar css selektora (više o njima u sledećem pitanju) navodi neki od mnogobrojnih stilskih elemenata, a zatim mu se dodeli vrednost:

h1

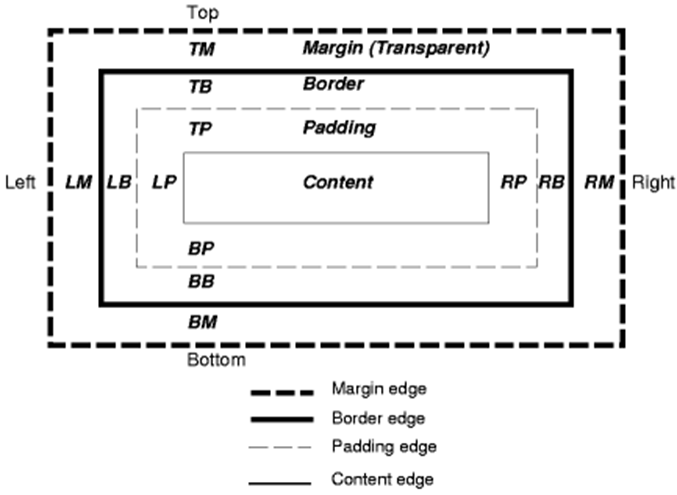
{

border: 1px solid black;

background-color: red;

}

Svaki HTML element ima pridružen CSS objekat, koji ima nekoliko važnih segmenata:



Margin predstavlja marginu elementa, odnosno njegovu udaljenost od susednih elemenata.

Border predstavlja ivicu samog elementa.

Padding predstavlja udaljenost sadržaja elementa od njegove ivice.

Content je sam sadržaj elementa.

1. CSS – selektori
   1. Selekcija tipova – navede se tip elementa (tag) koji se treba stilizovati:

h1

{

border: 1px solid black;

background-color: red;

}

- moguće je i grupisati tipove i time stilizovati više njih

h1, h2, h3

{

border: 1px solid black;

background-color: red;

}

* 1. Selekcija klasa – moguće je u html-u navesti klasu kojoj pripada neki konkretan element. U CSS-u se onda klasa navodi stavljanjem tačke ispred imena klase:

.klasa

{

border: 1px solid black;

background-color: red;

}

- ponovo je moguće grupisanje:

.klasa1.klasa2

{

border: 1px solid black;

background-color: red;

}

* 1. Selekcija ID-a – u html-u dodeliti elementu id (<tag id=“ID1“>), a zatim u CSS-u navesti id sa znakom # ispred ID-ja:

#ID1

{

border: 1px solid black;

background-color: red;

}

* 1. Selekcija HTML atributa – selekcija atributa vrši se unutar uglastih zagradi. Postoje neka specijalna značenja u okviru ove selekcije.
     1. = ima očekivano značenje, ime atributa sa leve strane mora imati vrednost sa desne
     2. \*= atribut sa leve strane mora da sadrži ono što je sa desne ([class\*=“aaa“] će se odnositi na klase sa nazivom „aaa“, „aaab“, „baaa“ ili na bilo koju klasu čiji nazv ima u sebi „aaa“
     3. ^= atribut sa leve mora da počinje onime što je sa desne strane
     4. $= atribut sa leve se mora završavati onime sa desne strane

1. Progresivno poboljšanje

Progresivno poboljšanje se odnosi na piramidalno unapređivanje same stranice, na čijem se dnu nalazi sam, potpuno čist HTML, na sledećem nivou je CSS gde se definiše način na koji je strana stilizovana i uređena, a na vrhu je javascript koji dodaje strani funkcionalnosti na front-end-u. Progresivno poboljšanje se bavi razlikom u verzijama i mogućnostima raznih web pretraživača. Zbog ovog principa, čak i vrlo starim pretraživačima nudi se mogućnost pristupa najosnovnijem sloju stranice, odnosno čistom html-u i osnovnim funkcionalnostima. Za naprednije pretraživače, dostupne su unapređene verzije, koje sadrže prvo stranu stilizovanu css-om, a na kraju i najviše funkcionalnosti ugrađene nenametljivim javascriptom. Nenametljiv javascript podrazumeva da se delovi skripte za definisanje dodatnih funkcionalnosti ne ugrađuju posebno u pojedinačne elemente HTML-a, već da se HTML poveže sa eksternom skriptom. Da se ugrađuje u svaki element, browser koji ne podržava neku JS funkcionalnost ostao bi uskraćen za celokupnu funkcionalnost tog elemente. Na primer, ako se direktno u ’href’ atribut <a> taga ubaci JS kod za prikaz pop-up-a, browser koji ovo ne podržava ne bi uopšte imao pristup tom tagu, i ne bi ga mogao iskoristiti.

Dakle, progresivno poboljšanje podrazumeva da je svim korisnicima, bez obzira na to koliko im je napredan browser, treba obezbediti osnovni set HTML elemenata i njihove osnovne funkcionalnosti. Zatim se ugrađuju dodatni CSS stilovi (preko eksternih sheet-ova) za naprednije pretraživače, a na kraju i dodatne funkcionalnosti pomoću eksternih JavaScript skripti.

1. Dizajn strane – css grid, flexbox

CSS grid layout je način uređivanja strane kojim se izbegava korišćenje atributa kao što su float i position, kojima se elementi teško raspoređuju na strani. Postoji jedan element koji sadrži sve ostale, grid item-e i naziva se grid-container. Element ovo postaje ako mu se u CSS fajlu navede display: grid ili inline-grid. Time sva njegova deca postaju grid-items. Grid praktično predstavlja matricu grid-item-a. Postoje takozvane grid-column-lines (ili row umesto column) koje predstavljaju linije između elemenata u matrici. Prva row-linija nalazi se iznad prvog reda elemenata, a poslednja se nalazi ispod poslednjeg reda (slično i za kolone). Elementi se pozicioniraju tako što se navede grid-column-start i grid-column-end (ili row), čime se označi od koje do koje linije se element prostire. Razmaci između elemenata se zovu gaps, pa se pomoću grid-row-gap (ili column) postavlja razmak između elemenata. Ako se navede samo grid-gap: 50px 100px, to znači da će razmaci između kolona biti 50, a između redova 100 px. Ako se navede grid-gap: 50 px, oba razmaka su 50px.

Flexbox dizajn je drugi način da se olakša raspoređivanje elemenata. Glavni element je flex-container kome se sa display: flex dodeli ovo svojstvo, a pomoću flex-direction navodi u kom smeru će se raspoređivati elementi unutar njega: row (horizontalno, sleva na desno), row-reverse (kontra), column (odozgo na dole), column-reverse (odozdo na gore). Najbitnija svojstva su sledeća:

* 1. Flex-grow – specificira koliki udeo u prostoru zauzima dati element pri širenju strane (odnosi se na osu po kojoj se prostiru elementi); ako se navede nekom elementu flex-grow: 2, on će rasti duplo brže od onih kojima je postavljeno na 1
  2. Flex-shrink – suprotno od flex-grow, dakle koliko će se brzo smanjivati element; ako se nekom elementu navede flex-shrink: 2, smanjivaće se duplo brže od onih kojima je postavljeno na 1
  3. Flex-basis – predstavlja početnu veličinu flex elementa
  4. Flex-wrap – način slaganja dece unutar roditelja (da li će preći u sledeću liniju ili ne, kada se dođe do kraja linije po kojoj se prostiru; na primer, ako je flex-direction: row, i u redu može stati 5 elemenata, ako je flex-wrap: wrap, onda će 6. element biti postavljen u sledećem redu, kao i 7. i tako dalje)
  5. Justify-content: poravnanje u odnosu na glavnu osu prostiranja
  6. Align-items: poravnanje u odnosu na osu koja je normalna na glavnu osu prostiranja
  7. Align-content: poravnanje linije potomaka u odnosu na poprečnu osu
  8. Order: redosled prikazivanja dece; ko ima manji order, biće prikazan bliže početku glavne ose prostiranja
  9. Align-self: individualno poravnanje konkretnog deteta

NAPOMENA: sva pomenuta svojstva se elementima postavljaju preko stilova (u CSS-u, ali može i preko style atributa u HTML tagu); link ka boljem opisu flex svojstava: <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>

1. Responsive dizajn

Pristup u kome se strane prilagođavaju veličini ekrana uređaja, a samim tim se i elementi prilagode novoj veličini strane. Omogućava da se strana prikazuje na odgovarajuć način i na telefonu i na pc-u, tabletu, lap topu.

Responsve dizajn je veoma važan, kada je u pitanju korisnikov doživljaj stranice. Razne tehnike se primenjuju kako bi se ovakav dizajn postigao. Felxbox metode pružaju veliku mogućnost po tom pitanju, s obzirom na to da je moguće definisati udeo koji će elementi imati na stranici, njihov raspored pri smanjenjivanju i povećanju stranice i tome slično. Takođe, ovakvom dizajnu pomaže i izbegavanje postavljanja apsolutnih veličina elemenata (u pikselima, ili nekoj drugoj mernoj jedinici), i, umeto toga, korišćenje procenata za postavljanje veličine, margina i tome slično (moguće je recimo u nekom stilu postaviti width: 50%, čime će element biti duplo manji od roditelja, i ovaj odnos će se održati na bilo kom uređaju, pri bilo kojoj veličini stranice).

Danas se često zajedno sa responsive dizajnom pominje i adaptivni dizajn. Iako je responsive dizajn nešto opštiji pojam, koji podrazumeva prilagođavanje stranice bilo kojoj veličini ekrana, i adaptivni dizajn je važan. On podrazumeva da se, pri stilizovanju stranice, uzmu u obzir određene „prelomne tačke“ što se tiče veličina ekrana, i da elementi u određenom rasponu veličine stranice imaju statičke stilove, a da se svojstva tih stilova menjaju pri prelasku neke od „prelomnih tačaka“. Za ovu svrhu koriste se media queries, kojima se definiše veličina ekrana pri kojoj se vrši promena svojstava određenog stila:

.class1 {

    width: 200px;

    display: flex;

    flex-direction: row;

}

@media only screen and (max-width: 450px) {

    .class1 {

        width: 100px;

        flex-direction: column;

    }

}

Na ovom primeru, definisan je određen stil za klasu class1. Zatim je, pomoću media query-ja, postavljena prelomna tačka od 450px. Za ovaj konkretan primer, kada širina ekrana postane manja od 450px, širina elementa sa klasom .class1 se menja na 100px, a flex-direction postaje column. (Da je bilo min-width:450px, stil bi se promenio onda kada širina ekrana postane veća od 450 px).

1. JS opšte

(Pretežno iz Ljubenovićeve skripte). JavaScript je skriptni dinamički, slabo tipiziran jezik.

Dinamički je, jer se tipovi dodeljuju dinamički (u toku izvršenja), promenljive su uglavnom smeštene u dinamičkoj memoriji i moguće je dodavati nove funkcije i nove promenljive u toku samog izvršenja koda.

Slabo tipiziran znači da je moguće vršiti operacije između dva podatka koji se ne poklapaju po tipovima. Zbog toga je, recimo, moguća konkatenacija broja na string, ili čak množenje stringa i broja.

Baziran je na prototipovima sa funkcijama prve klase (to znači da funkcije mogu biti dodeljene promenljivama, prosleđene kao parametar, biti povratna vrednost drugih funkcija i posmatraju se kao atribut-članovi klase). Podržava objektno-orijentisano (klase i objekti), imperativno (naredbama se opisuje kako program treba da radi i menja mu se stanje u toku rada) i funkcionalno programiranje (moguće je čitav rad koda predstaviti funkcijama, prosleđivati jednu funkciju drugoj, koristiti čiste lambda funkcije kao što su map, find, filter, reduce i slično). Ima API za rad sa tekstom, nizovima, datumima i regularnim izrazima, ali ne podržava nikakav ulaz niti izlaz i oslanja se na okruženje u koje je ugrađen (najčešće pretraživač, ali može biti i Adobe Acrobat, Photoshop...). Podržavaju ga svi moderni pretraživači, a njegova svrha u izradi web strana je da im doda funkcionalnosti. Od tipova podataka, podržava: Number, String, Boolean, Object (gde spadaju Function, Array, Date i RegExp), null i undefined. Razlika između null i undefined je ta što vrednost null dodeljuje sam programer, da označi da je vrednost promenljive „ništa“, a undefined se automatski dodeljuje neinicijalizovanim promenljivama. Svaka vrednost se može kastovati u logičku, pri čemu false, 0, prazan string, NaN, null i undefined postaju false, a sve ostalo postaje true.

1. Funkcije

Funkcije su deo koda u JS-u koji se može pozvati na izvršenje više puta, navođenjem identifikatora funkcije. Deklarišu se pomoću ključne reči function, iza koje sledi ime funkcije, pa lista argumenata, pa zatim telo. Funkcija može imati globalni opseg, ali se može deklarisati i unutar druge funkcije, pri čemu je vidljiva samo unutar nje. Poziv fje može se navesti pre deklaracije, jer JS sve deklaracije „vidi“ pre izvršnog dela koda. Funkcijski izrazi nastaju kada se nekoj promenljivoj dodeli funkcija (najobičnijim operatorom dodele):

let x = function(a1, a2){ ... }

Poziv izraza se vrši navođenjem imena promenljive i argumenata u zagradi (u suštini isto kao i poziv obične funkcije). Ne može se pozivati pre nego što se funkcija dodeli promenljivoj, jer JS prvo vidi deklaracije, ali ne i izvršne linije, u koje spadaju i dodele. U ovom slučaju, promenljivoj je dodeljena anonimna funkcija. Može se dodeliti i imenovana funkcija, ali se to ime može koristiti samo u lokalnom opsegu te f-je (za rekurziju):

let x = function func(a1){ ... func(5)...} //ime funkcije „func“ može se koristiti samo ovde.

Skraćeni način zapisivanja funkcije koja se dodeljuje je sledeći:

let f = (x) => {return x+1;} ili, još kraće: let f = (x)=>x+1;

Ako funkcija ima samo jedan parametar, može i: let f = x=>x+1;

Funkciji se uvek može proslediti proizvoljan broj parametara (bez obzira na njen potpis). To je korisno ako funkciju koristimo ovako:

function func(){

for(let i = 0; i<arguments.length; i++)

... arguments[i]...

}

Ovako se funkciji može proslediti koliko god parametara. Zbog ove osobine, JS ne dozvoljava da postoje dve funkcije sa istim imenom, makar imale i različit broj argumenata u potpisu.

U JS-u, funkcije mogu biti argumenti i povratne vrednosti drugih funkcija. Kada se neka funkcija prosledi drugoj kao argument, poziva se pomoću apply:

function f(imeFje)

{ ...imeFje.apply...}

1. Razlike između deklaracije i izraza funkcije (FE)

Odgovor je obrađen u prethodnom pitanju. U suštini glavne razlike su što izraz predstavlja dodelu funkcije promenljivoj i što se izraz ne može pozivati pre dodele funkcije promenljivoj, a funkcija koja je klasično deklarisana može se pozvati pre deklaracije.

1. Hoisting

Hoisting predstavlja mehanizam javascripta kojim se deklaracije „pomeraju na vrh“, odnosno vide se pre izvršnog dela koda. Zato se funkcije, kao i promenljive, mogu koristiti pre nego što su deklarisane. Sasvim je validno napisati:

x=“Dobar dan“;

console.log(x);

var x;

Hoisting pomera deklaracije na vrh opsega u kome se nalaze (na vrh cele skripte, ako su van funkciije, na vrh funkcije ako su unutar nje). Hoisting se ne može primeniti na let i const promenljive, zato što se na njih odnosi problem koji se naziva temporal dead zone. Naime, za var promenljive važi da se pri deklaraciji ujedno i inicijalizuju na undefined. Za const i let važi da se pri interpretaciji JS koda njihovo ime registruje kao postojeće, još pre nego što se prođe kroz liniju koja sadrži npr. kod „let promenljiva“. Pre nego što se prođe taj deo koda, promenljiva je neicijalizovana i nalazi se u „temporal dead zone“. Ako se tada referencira po imenu, ime će se prepoznati, ali će doći do greške koja kaže da je promenljiva neinicijalizovana. Tek kad se prođe deo koji sadrži kod „let promenljiva“, doći će do inicijaizacije na undefined. Zato se hoisting u ovom slučaju ne može primeniti.

Temporal dead zone (odnosi se na let i const promenljive) je deo koda između referenciranja promenljive po imenu i linije u kojoj je ona deklarisana. Ime promenljive je poznato, i ukoliko se bude iskoristilo kao referenca, znaće se na šta se odnosi, ali ta promenljiva još uvek nije inicijalizovana i nastupiće greška. Ukoliko imamo sledeći kod:

console.log(prom);

let/const prom; (ako bismo ovde dodelili vrednost, npr let/const prom=5, ništa se ne bi promenilo)

nastupiće greška i kod će pući.

Sa druge strane, ako koristimo var:

console.log(prom);

var prom;

rezultat bi bio „undefined“. Ako bismo u liniji deklaracije odradili i inicijalizaciju, ništa se ne bi promenilo, zbog već pomenutog mehanizma gde se deklaracija pomera na vrh scope-a, ali ne i operacija dodele.

1. Lambda funkcije

Lambda funkcije (ili arrow funkcije) su novina koju uvodi JS ES6. Pre njih, postojala je mogućnost kreiranja takozvanih anonimnih funkcija (dakle funkcija koje nemaju identifikator kao naziv), ali se morala koristiti ključna reč ’function’:

let func = function(a, b) { c = a+b; return c;}

U ES6, moguće je kreirati anonimnu funkciju, korišćenjem ’strelice’, =>, koja omogućuje da se izbaci korišćenje ključne reči ’function’:

let func = (a, b) => { c=a+b; return c;}

Ukoliko se funkcija svodi na jednu liniju, kojom se može predstaviti povratna vrednost, mogu se izbaciti uglaste zagrade i pisati samo:

let func = (a, b) => a+b;

Dodatno, ako se radi o funkciji sa jednim parametrom, imamo:

let func = a => a+1;

Ono što je vrlo korisno kod lambda izraza, je to što funkcije predstavljene lambda izrazom nemaju za sebe vezanu vrednost kojom se pristupa korišćenjem identifikatora ’this’, već ako se u telu lambda funkcije iskoristi ’this’, odnosiće se na ono na šta se odnosi ’this’ u scope-u u kome se nalazi lambda funkcija (jasnije je iz primera):

class Klasa {

    constructor(incr) {

        this.increment = incr;

        setInterval(function () {

            this.increment++;

        }, 1000);

    }

}

U ovom slučaju, ’this’ koje je prvo iskorišćeno se odnosi na objekat klase koji se kreira, ali je ’this’ unutar setInterval f-je lokalno this za tu funkciju, pa se increment kao atribut klase neće inkrementirati.

class Klasa {

    constructor(incr) {

        this.increment = incr;

        setInterval(() => {  this.increment++; }, 1000);

    }

}

U ovom slučaju, ’this’ na oba mesta pripada klasi, pa će se dobiti željeni efekat.

1. Closure i kontekst izvršenja funkcija

Closure u javascript-u je kombinacija neke funkcije i njenog takozvanog leksičkog okruženja. Leksičko okruženje predstavlja skup svih identifikatora (i njima pridruženih vrednosti) koji se u trenutku kreiranja funkcije nalaze u scope-u u kome se ona kreira:

let a=5;

let b=10;

function func(){...}

Leksičko okruženje ove funkcije sadrži a, b i njihove vrednosti. Takođe, u javascript-u postoji mehanizam koji se naziva „Scope Chain“ koji nalaže da se, kada se u funkciji koristi neki identifikator, on prvo potraži u lokalnom scope-u, pa ako se ne nađe onda u jednom iznad i tako dalje sve dok se ne nađe. Closure koristi scope chain, kao i činjenicu da funkcija zna za svoje leksičko okruženje. Closure se svodi na to da, ako se pozove ugnježdena funkcija čak i nakon što se njena roditeljska funkcija završila, ugnježdena će imati pristup vrednostima promenljivih koje su se nalazile u scope-u roditeljske funkcije:

function func() {

    let prom = 5;

-> function ugnjezdena() {

        return prom + 5;

    }

    return ugnjezdena;

}

let X = func();

console.log(X());

Ovaj kod će u konzoli prikazati 10. Pri inicijalizaciji promenljive X, pozvana je na izvršenje funkcija func. U liniji na koju pokazuje strelica kreiran je Closure za funkciju ugnjezdena, čime ona postaje svesna svog leksičkog okruženja. U promenljivu X smeštena je funkcija ’ugnjezdena’. Kada se navede X(), ta funkcija se poziva na izvršenje, ali bez obzira na to što je func završila sa izvršavanjem, funkcija ’ugnjezdena’ čuva svoje leksičko okruženje, odnosno u ovom slučaju scope svoje roditeljske funkcije. Kada se dođe do linije u kojoj se povećava vrednost promenljive prom(), prvo se traži lokalni scope funkcije ’ugnjezdena’. Nakon što se ne nađe, prelazi se na viši scope, odnosno leksičko okruženje ove funkcije, u kome se nalazi prom kao privatni član funkcije func, pa se njegova vrednost inkrementira.

1. IIFE

IIFE ili Imediately Invoked Function Expression je funkcija koja se izvrši čim se deklariše. Da bi se postigao ovaj efekat, deklaracija funkcije se zapisuje unutar zagrada:

(function func(a) { return a+1;})(5); //odmah nakon ove linije rezultat će biti 6, nema potrebe za pozivanjem funkcije navođenjem njenog imena

Nije važno da li se lista prosleđenih argumenata piše van zagrada u kojima je deklaracije funkcije (kao u primeru iznad), ili se zapiše unutar njih:

(function func(a) { return a+1; } (5)) //rezultat će biti isti kao u primeru iznad

Funcija koju zapišemo unutar zagrada može biti arrow funkcija i može biti anonimna. Čak i ako ima ime, ovakva funkcija ne može se pozvati ponovo navođenjem njenog imena. To se dešava zbog toga što stavljanje funkcije unutar zagrada od nje zapravo čini izraz koji se treba izvršiti kada se u kodu do njega stigne, kao i svaki drugi izraz (odatle i ’Expression’ u nazivu ovog mehanizma). Ova činjenica čini funkcije pogodne za simuliranje Singleton obrasca. Recimo da imamo sledeći slučaj:

function createCounter() {

    let counter = 0;

    function reset() {

        counter = 0;

    }

    function increase() {

        counter++;

    }

    return {

        incr: increase,

        res: reset

    }

}

Prvo treba razjasniti neke stvari. Ovde smo deklaracijom promenljive unutar same funkcije kreirali nešto što se može nazvati privatnom promenljivom. Zatim, unutar funkcije deklarisali smo dve nove. Ako ove unutrašnje funkcije pozovemo van njhove roditeljske, one će imati pristup counter promenljivoj zbog closure-a. Na kraju, u bloku za return vidimo način na koji je moguće vratiti više stvari iz jedne funkcije. Ovo što vidimo u bloku za return je zapravo kreiranje takozvanog anonimnog objekta. On ima privatne članove incr i res preko kojih je moguće pozvati funkcije increase i reset. Kada se kreiraju ovi privatni članovi, nakon dve tačke navode se imena funkcija bez liste argumenata, jer bismo dodavanjem i liste argumenata zapravo pozvali te funkcije na izvršenje. Sada, možemo uraditi sledeće:

const count = createCounter();

count.incr(); //nakon ove linije, counter postaje 1

count.incr(); //sada count postaje 2

count.res(); //sada se count vraća na nulu

count.incr(); //ponovo je 1

const count2 = createCounter();

count.2incr(); //count2 imaće svoj privatni counter, koji ovde postaje 1

count2.res(); //ovo resetuje counter iz count2, ali counter u count ostaje 1

Ovde smo videli demonstraciju closure mehanizma kao i kreiranje anonimnih objekata i privatnih članova. Ako bismo želeli da postoji samo jedna instanca counter-a, morali bismo da iskoristimo prednost IIFE-a, koja nalaže da se IIFE funkcija ne može pozvati ponovo na izvršenje:

const count = (

    function createCounter() {

    let counter = 0;

    function reset() {

        counter = 0;

    }

    function increase() {

        counter++;

    }

    return {

        incr: increase,

        res: reset

    }

})();

Promenljiva count sada sadrži ono što je IIFE vratio, odnosno u njemu se nalazi anonimni objekat koji sadrži incr i res kao privatne članove. Sada možemo pozivati incr i res, ali ne možemo ponovo pozvati na izvršenje funkciju createCounter:

const count2 = createCounter(); //GREŠKA!

count.incr();

count.res();

1. Nizovi

U javascriptu, niz predstavlja poseban tip promenljive koja u sebi može u trenutku da čuva više od jedne vrednosti. Elementima niza (vrednostima koje se čuvaju u toj promenljivoj) pristupa se preko indeksa. Postoje dva načina da se kreira niz. Prvi je korišćenjem uglastih zagrada:

const niz1 = [1, 2, 3]; //kreira niz od tri elementa

const niz2 = [ ]; kreira se prazan niz

Drugi način je korišćenjem ključnih reči ’new’ i ’Array’:

const niz1 = new Array(1, 2, 3); //kreira se niz od 3 element

const niz2 = new Array(); //kreira se prazan niz

Ne postoji nikakva razlika između ova dva načina za kreiranje niza.

Članovima se pristupa navođenjem indeksa, dok se celom nizu može pristupiti navođenjem samo imena niza:

const niz1 = [1, 2, 3];

document.body.innerHTML = niz1; //innerHTML će postati ceo niz, odnosno na stranici će se prikazati svi brojevi odvojeni zarezom (dakle 1,2,3)

Nizovi u JS-u zapravo su specijalan tip objekta. Ako se iskoristi operator typeof nad imenom niza ( typeof(niz1) ), vratiće object. Ipak, imajući u vidu da se članovima niza pristupa preko indeksa, a ne preko imena člana (kao kod objekata), nizovi se posmatraju kao nizovi kakve inače poznajemo.

Nizovi mogu sadržati raznorodne elemente (stringove, brojeve, karaktere, objekte, funkcije, druge nizove). Ova osobina proizilazi upravo iz toga što je u JS niz zapravo objekat, a objekti mogu sadržati članove bilo kog tipa.

**Svojstva i metode nizova:**

- properti length vraća broj elemenata niza

- metoda sort sortira niz; default sortiranje je u rastućem redosledu; niz se sortira tako što se svaki element prevede u string, a zatim se sortira leksikografski, poredeći njihove utf-16 kodove; tako, 1000 će biti manje od 2, zato što je 1 pre 2 po utf-16 kodu; sort može kao parametar da primi funkciju po kojoj se sortiraju elementi, koja mora da vrati numeričku vrednost; ta funkcija prima dva parametra, odnosno elemente koje treba da poredi; recimo da ta funkcija prima parametre a i b, i da se oni tim redosledom nalaze u listi argumenata; ako funkcija vrati nulu, a i b neće menjati mesta; ako povratna vrednost bude manja od 0, a će ići na indeks koji je manji od indeksa na kome je b; ako se vrati veće od 0, b ide na indeks manji od indeksa na kome je a

- nizovi imaju funkciju forEach kojom se obilaze svi elementi niza

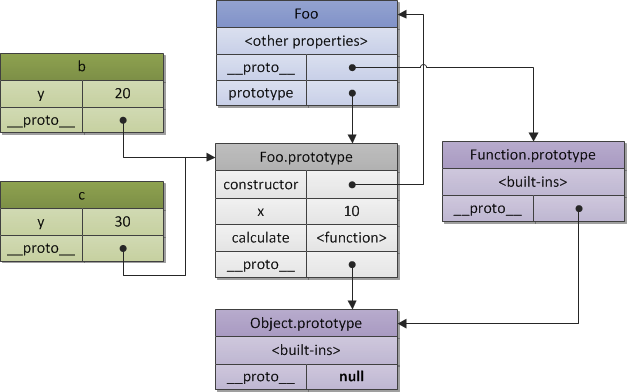
- funkcijom push dodaje se element na kraj niza; elementi se mogu dodati u niz i preko indeksa (niz[4]=“abc“), ali ukoliko dodamo recimo na indeks 4, a poslednji popunjen indeks je indeks 2, onda niz[3] postaje undefined; ako ne koristimo push, bezbednije je dodati element kao:  
niz[niz.length]=“abc“, čime se dodaje element takođe na kraj niza; funkcija push vraća novu dužinu niza

- funkcija pop() izbacuje i vraća poslednji element niza

- funkcija shift izbacuje i vraća prvi član niza

- funkcija unshift dodaje član na početak niza i vraća njegovu novu dužinu

1. Prototipi (prototipovi?)



Svi JS objekti svoje propertije i metode nasleđuju od prototipa. On se manifestuje kroz properti po imenu „prototype“ konstruktorske funkcije tog objekta:

function Product(id, kolicina)

{

    this.id=id;

    this.kolicina=kolicina;

    this.PovecajKolicinu = () => {

        this.kolicina++;

    }

    this.display = function display() {

      console.log(this.kolicina)

    }

}

const prod1 = new Product(10, 20);

prod1.PovecajKolicinu()

prod1.display() //ispisuje 21

Korišćenjem „this“ pri deklaraciji i definiciji atributa i metoda unutar funkcije, kreiramo nešto što bi se moglo smatrati public članovima klase. Ukoliko želimo efekat privatnih članova, jednostavno ne koristimo this, već const, let i var za atribute, a metode kreiramo, ali ne dodelimo nijednoj promenljivoj (ne pravimo funkcijski izraz).

Prototipu klase Product pristupamo pomoću Product.prototype. Ovako se pristupa prototipu koji se odnosi na samu konstruktorsku funkciju. Ali, pored toga, svaki objekat kreiran pomoću te konstruktorske funkcije ima svoj prototip, kome se pristupa preko imeObjekta.\_\_proto\_\_ (dve donje crte!!!). U ovom slučaju, prototipu objekta prod1 se pristupa preko prod1.\_\_proto\_\_. Ipak, ne postoji razlika između objekta kome se pristupi pomoću Product.prototype i objekta koji se dobije pomoću prod1.\_\_proto\_\_, naprotiv, to je **jedan isti objekat**. (Pogledati sliku. U primeru sa slike, Foo je konstruktorska funkcija, i njen .prototype pokazuje na Foo.prototype, dok su a i b objekti kreirani pomoću ove konstruktorske funkcije, čiji pokazivač \_\_proto\_\_ ukazuje na istu stvar)

Preko .prototype, mogu se dodati nove metode ili atributi u konstruktorsku funkciju:

Product.prototype.menjajKolicinu = function(kol) { this.kolicina += kol;} //dodavanje metode

Product.prototype.cena = 500; //dodavanje atributa

Prototip jednog objekta može „iznad sebe“ imati prototip od koga on nasleđuje metode i propertije. Ovaj mehanizam zove se „prototype chain“. Recimo, u ovom primeru, Product.prototype iznad sebe ima još jedan prototip, kome se može pristupiti na dva načina:

- prvi je preko Product.prototype.\_\_proto\_\_

- drugi je preko prod1.\_\_proto\_\_.\_\_proto\_\_

Oba načina pristupaju istom objektu, i to Object.prototype objektu, koji je **uvek na vrhu prototype chain-a!!!** Product nasleđuje atribute i funkcije iz Object.prototype, što znači da će te atribute i osobine naslediti i objekti kreirani pomoću Product funkcije, odnosno prod1 u našem primeru.

Sa slike se još vidi da Foo, sa obzirom na to da je funkcija, sadrži svoj \_\_proto\_\_, koji ukazuje na Function.prototype. Bez obzira na to, Function.prototype ima svoj \_\_proto\_\_, koji ukazuje na Object.prototype, jer, kao što je već rečeno, na vrhu je uvek Object..prototype.

(Za bolje razumevanje neophodno je pratiti sliku!)

(Stvari koje su pominjane na času na kome su se radili prototipovi:

- „prod1 instanceof Product“ vraća „true“

- „prod1.hasOwnProperty(„id“)“ vraća „true“)

U ES6 uvodi se pojam klasa, čijim se korišćenjem odstranjuje potreba za eksplicitnim korišćenjem prototipova i pisanjem konstruktorskih funkcija. Klase imaju sopstvenu funkciju po imenu „constructor“ u kojoj se postavljaju atributi i metode koje će objekat imati. Uvođenje klase ne znači da se prototipovi ne mogu koristiti na opisani način. Mogu, ali nema potrebe za time.

1. Konstruktori preko IIFE-a (nisam siguran)

Konstruktori preko IIFE-a postižu se tako što se funkcija uokviri u zagrade, čime se kreira izraz koji se odmah poziva. Unutar te funkcije, navede se konstruktorska funkcija kojom se postave atributi klase, a eventualne metode dodaju se pomoću prototipa.

let Product = (function(){

    function Product(price, name){

        this.price=price;

        this.name=name;

    }

    Product.prototype.increasePrice = function(){

        this.price++;

    }

    Product.prototype.display = function(){

        console.log(this.price, this.name);

    }

    return Product;

})();

let prod = new Product(20, "AAA");

prod.display(); //štampa 20 „AAA“

prod.increasePrice();

prod.display(); //štampa 21 „AAA“

1. Objekat literali

Promenljive u JS-u mogu biti posmatrane kao objekti. Najčešće, promenljive imaju po jednu dodeljenu vrednost. Ako želimo da bez koišćenja klasa i konstruktorskih funkcija kreiramo promenljivu koja može čuvati više vrednosti, odnosno objekat, koristimo objekat-literale:

let person = {firstName: „John“, lastName: „Doe“, age:50, eyeColor: „blue“};

Ovo je objekat literal, kojim se objektu „person“ dodelilo više različitih vrednosti. Ime svake vrednosti koju objekat čuva (firstName, lastName, age, eyeColor) predstavlja naziv propertija koji sadrži objekat. Ovim propertijima moguće je pristupiti na dva načina:

* person.firstName
* person[„firstName“]

Objekti takođe mogu imati i metode, što su zapravo propertiji čije su vrednosti funkcije:

let person = {... fullName: function(){ return this.firstName + „ „ + this.lastName},...}

U ovom slučaju, „this“ se odnosi na „person“.

Metodi nekog objekta pristupa se kao: person.fullName();

1. ES6 Klase (+ primer)

ES6 uvodi ključnu reč „class“ koja služi za kreiranje klasa u JS-u. Takođe, postoji i ključna reč „constructor“ koja se koristi za funkciju koja je zapravo konstruktor klase. Toj funkciji moguće je proslediti parametre. Unutar nje, this se koristi za označavanje atributa klase. Kasnije u kodu klase, da bi se iskoristili atributi, treba se iskoristiti „this“ i naziv koji je dodeljen atributu unutar konstruktora. U klasi, moguće je definisati metode, gde se za njihov potpis ne koristi ni „this“ ni „function“, već se samo navede ime metode, lista parametara i zatim implementacija. Kada se metoda koristi u kodu, neophodno je i this. Na mestu gde želimo da koristimo ovu klasu, objekte kreiramo pomoću operatora new. Da bi klasa bila vidljiva u ostalim fajlovima koji je koriste, ispred „class“ stavlja se ključna reč „export“. Na mestu gde se klasa koristi (u tom fajlu), stoji „imoprt {imeKlase} from „putanjaDoFajla.js“

Primer:

class Primer{

  constructor(*arg1*, *arg2*){

    this.arg1=arg1;

    this.arg2=arg2;

}

  povecajArgumente(){ this.arg1+=1; this.arg2+=1;}

  pozoviFunkciju(){ this.povecajArgumente();}

  display() {console.log(this.arg1 + " " + this.arg2)}

}

U drugom fajlu:

import {Primer} from „./primer.js“

const pr1=new Primer(4, 5);

pr1.pozoviFunkciju();

pr1.display(); //štampa „5, 6“

Klase u ES6 podržavaju static metode i atribute, pa zbog toga postoji „static“ ključna reč. Ovakvi članovi se koriste sa ImeKlase.imeAtributa, gde je imeAtributa static atribut, kome je dodeljena neka vrednost.

Za razliku od funkcija, klase ne podržavaju hoisting, tako da, ako je deklaracija klase u istom fajlu gde se ona i koristi, prvo mora da ide deklaracija, pa tek onda korišćenje.

Slično kao kod konstruktorske funkcije, članovi koji se navode koristeći „this“ su javni, dok se klasično deklarisani članovi (pomoću var, let, const za atribute, i function za metode) ponašaju kao privatni članovi (pričamo o konstruktoru; u njemu je moguće kreirati klasične promenljive i funkcije i dobiti privid privatnih članova; ipak, ako se ovako kreirane promenljive i funkcije ne dodele ničemu u konstruktoru, nije ih moguće iskoristiti nikako van konstruktora, ni u klasi ni van nje).

1. ES6 nasleđivanje (+primer)

ES6 podržava koncept nasleđivanja vrlo sličan onome na koji smo navikli u OO jezicima. Uvodi se ključna reč extends, nakon koje se navodi klasa iz koje se izvodi. Takođe, ključna reč super koristi se da se pristupi konstruktoru, metodama i atributima nadklase.

class Shape{

    constructor(name){

        this.name=name;

    }

    getName(){

        return this.name;

    }

}

class Circle extends Shape {

    constructor(radius){

        super("circle");

        this.radius=radius;

    }

    getDescription(){

        alert(`this is a ${super.getName()} with a radius of ${this.radius}`);

    }

}

const c = new Circle(5);

c.getDescription();

U ovom primeru, klasa Circle izvedena je iz klase Shape. U konstruktoru klase Circle, poziva se konstruktor nadklase i inicijalizuje atribut name, nasleđen iz nadklase. U metodi getDescription postoji i poziv metode iz nadklase, koji je takođe realizovan pomoću ključne reči „super“.

U ovom primeru, da se funkcija getDescription kojim slučajem zvala getName, i da se u main-u pozvalo c.getName(), pozvala bi se funkcija iz izvedene klase. Da ovakva funkcija nije postojala uopšte u izvedenoj klasi, a da se pozivala u main-u (c.getName()), onda bi se pozvala funkcija iz osnovne klase.

1. ES5 klase (+primer)

U ES5 ne postoji ključna reč „class“. U suštini, „class“ u ES6 samo kreira lakšu sintaksu za pravljenje klasa u JS-u. Originalno, klase su se kreirale pomoću funkcija. Princip je isti: članovi kreirani pomoću „this“ postaju javni, a članovi koji su deklarisani na standardni način (var, let, const, function) postaju privatni.

Novi članovi se u runtime-u mogu dodati korišćenjem propertija „prototype“.

function Product(id, name){

    let kol; //privatna promenljiva

    this.id = id; //javna promenljiva (kao i sve ostale sa „this“)

    this.name = name;

    this.price;

    this.setPrice = (pr)=> this.price=pr; //metode

    this.setKol = (k) => kol=k;

    this.getKol = () => console.log(kol);

}

const prod1 = new Product(21, "AAA");//kreiranje novog objekta

const prod2 = new Product(44, "BBB");

console.log(prod1.id, prod1.name);

console.log(prod2.id, prod2.name);

Product.prototype.getPrice = function(){

    console.log(this.price);

} //dodavanje nove metode korišćenjem prototipa; nakon ove linije,  
 //svaki objekat će u svom \_\_proto\_\_ propertiju da ima funkciju getPrice

prod1.setPrice(45);

prod2.setPrice(92);

prod1.getPrice();

prod2.getPrice();

prod1.setKol(25);

prod2.setKol(87);

prod1.getKol();

prod2.getKol();

Što se nasleđivanja tiče, potrebno je da izvedena klasa nasledi sve članove osnovne, odnosno, da prototip izvedene klase ima sve metode i atribute koje ima i osnovna. U tu svrhu, koristi se funkcija Object.create, koja kreira objekat čiji prototip postaje argument prosleđen ovoj funkciji.

function Animal(className) {

    this.className = className;

}

Animal.prototype.getClass = function() {

    return "Animal class is : " + this.className;

}

function Dog(name) {

    Animal.call(this, "Animal");

    this.name = name;

}

Dog.prototype = Object.create(Animal.prototype);

Dog.prototype.constructor = Dog;

Dog.prototype.getName = function() {

    return this.name;

}

let d = new Dog("Tommy");

Kreirali smo nadklasu Animal koja ima atribut className i metodu getClass. Zatim smo kreirali klasu Dog. Unutar nje, koristili smo funkciju call. Pomoću nje, pozvali smo konstruktor klase Animal, ali se funkciji call kao parametar prosledi objekat na koji će se pozvani konstruktor odnositi (kad se uđe u konstruktor Animal i stigne do this.className, „this“ se neće odnositi na Animal, nego na Dog, jer smo u call prosledili Dog, tako što smo kao parametar naveli „this“; pogledaj na internetu objašnjenje za funkciju call, bolje je objašnjeno).

Na ovaj način, učinili smo da Dog ima svoj parametar className, koji je „nasledio“ iz Animal. Zatim smo u Dog dodali atribut „name“. U sledećoj liniji (ovoj sa Object.create), učinili smo da prototip klase Dog dobije sve ono što ima i prototip klase Animal, i time simulirali nasleđivanje. Zatim, da bismo se osigurali da Dog ima svoj konstruktor, eksplicitno smo rekli na koju funkciju konstruktor klase Dog treba da ukazuje. Konačno, dodali smo getName kao novu metodu klase Dog.

(Slajdovi imaju čudan primer za izvođenje motora iz vozila)

1. Typescript

Typescript je klijentska tehnologija. Predstavlja „superset“ javascript-a, odnosno JS kod koga su uvedeni tipovi. Važna stvar je da se typescript prevodi u JS koji se izvršava u browser-u, a ne izvršava se sam typescript. Takođe, u html-u se ne može direktno kao src za skriptu navesti .ts fajl, već se mora prvo prevesti u .js. Neki od osnovnih tipva su boolean, number, string, array, enum, any (kazuje da parametar ili promenljiva može biti bilo kog tipa). Tipovi se navode tako što se nakon imena promenljive stavi :tip (npr. let broj : number). Ako se kao parametar funkcije navede jedan tip, a prosledi drugi, typescript će prijaviti grešku. Isto važi i za povratnu vrednost funkcije, koja se navodi na isti način kao i tip parametra i promenljive (:tip nakon potpisa funkcije). Ako se ne navede ništa, funkcija može vratiti bilo šta.

Kod ima greške samo na nivou ts-a. Kod sa greškom u ts-u može se prevesti u validan kod u js-u, koji se može izvršiti. Tj. Provere koje nalaže TypeScript se izvršavaju samo u toku pisanja koda, biće označeno i podvučeno crveno ono što po pravilima Typescript-a nije validno tj. „javiće se greška“ ali obzirom na to da se Typescript prevodi u JavaScript i kao takav izvršava, ove greške neće da se javljaju u runtime-u, već će izvršenje da teče normalno.

Kompajliranje (prevođenje u .js) se postiže kucanjem tsc imeFajla.ts u terminalu. U fajlu tsconfig.json, imamo između ostalog i compilerOptions deo. U njemu možemo navesti

„outDir“: „direktorijumZaJs“, čime navodimo u kom direktorijumu će se nalaziti dobijeni .js kod. ImeFajla.ts prevodi se u ImeFajla.js, tako da, ako navedemo kao outDir na primer „./mojDir“, onda bismo u html-u kao src naveli „./mojDir/ImeFajla.js“. Ako kao „target“ u compilerOptions navedemo „es5“, sve klase koje imamo u ts će se prevesti u IIFE konstruktorske funkcije. Često se tu bira baš „es5“ zbog kompatibilnosti sa svim browser-ima.

Klase u typescript-u rade slično kao u js-u, s tim što je moguće navođenje tipova atributima i parametrima funkcija, ali i navođenje prava pristupa. Podržava sva tri prava, private, protected i public, sa istim značenjem kao u OO jezicima. Takođe, atributi se navode najnormalnije, kao u OO jezicima, tako što im se navede ime i nakon toga tip, a ispred imena moguće je da stoji pravo pristupa, s tim što ako ga nema, podrazumeva se public pravo. Kada se atributi koriste, navodi se „this.imeAtributa“.

Nasleđivanje takođe radi slično kao u js. Ključna reč je „extends“, konstruktor osnovne klase se poziva sa „super(parametri)“, a metode osnovne pomoću „super.imeMetode(parametri)“.

U typescript-u je moguće kreirati i interfejse pomoću ključne reči ’interface’. Koriste se da bi se naložilo koje sve atribute (ili metode) mora da ima objekat koji se prosledi na mesto ovog interfejsa, ili klasa koja ga implementira:

interface IProizvod {

    naziv: string;

    cena: number;

}

function radiNesto(param: IProizvod) {

    console.log(param.cena);

}

const obj = {naziv: "aaa", cena:20, kolicina: 30};

radiNesto(obj);

U datom primeru, objekat koji smo prosledili funkciji se može prihvatiti, zato što se proverava da li on ima atribute (propertije) koje nalaže interfejs, bez obzira na to da li ima još nešto dodatno. Ovaj mehanizam poznat je pod nazivom Duck-typing („ako hoda kao patka i kvače kao patka, onda i jeste patka“). Po tom mehanizmu, vrši se samo provera odgovarajućih članova interfejsa, dakle, da li u objektu koji se prosleđuje postoje članovi istog tipa i imena kao u interfejsu. Ukoliko postoje, prihvataju se.

Ipak, da se umesto ’obj’ direktno na mesto parametra funkcije radiNesto prosledilo ovo između vitičastih zagrada (inicijalizacija objekta ’obj’), typescript bi prijavio grešku kod poslednjeg člana, ali bi kod opet mogao i uspeo bi da se prevede u validan js.

1. Typescript klase (+primer)

Sve stvari za klase navedene su u pitanju iznad. Ovde će biti dat primer.

class Product {

    price: number;

    name: string;

    constructor(pr: number, nm: string)

    {

        this.price = pr;

        this.name = nm;

    }

    private display() {

        console.log(this.price, this. name);

    }

    increase() {

        this.price++;

        this.display();

    }

}

let pr = new Product(15, "AAA");

pr.increase();

Na ovaj način, kreirali smo klasu Product, a zatim i objekat te klase.

1. Typescript nasleđivanje (+primer)

Objašnjeno u pitanju 30. Ovde će biti dat primer.

class Shape{

    protected name: string;

    constructor(nm:string){

        this.name = nm;

    }

    public getName() {

        return this.name;

    }

}

class Circle extends Shape{

    private radius: number;

    constructor(rad: number){

        super("Circle");

        this.radius = rad;

    }

    public getDescription(){

        console.log(`This is a ${super.getName()} with the radius of ${this.radius}`);

    }

}

let c = new Circle(15);

c.getDescription();

Atribut „name“ u klasi Shape je zaštićen i njemu se može pristupiti i iz klase Circle. Super se koristi za pozivanje konstruktora i metode getName iz osnovne klase, mada se getName može pozvati i pomoću this.getName, jer je ta metoda nasleđena i vidljiva i u izvedenoj klasi.

Typescript podržava preklapanje imena metoda, ali samo ukoliko obe metode imaju istu povratnu vrednost. U ovom slučaju, ne bi bilo dozvoljeno da getDescription preimenujemo u getName, zbog toga što ne bi imalo istu povratnu vrednost kao getName u osnovnoj klasi. Ako prepravimo tako da imaju istu povratnu vrednost, mogli bismo da imamo sledeći slučaj:

class Shape {

    protected name: string;

    constructor(nm:string) {

        this.name = nm;

    }

    public getName() {

        return this.name;

    }

}

class Circle extends Shape {

    private radius: number;

    constructor(rad: number) {

        super("Circle");

        this.radius = rad;

    }

    public getName() {

        return `This is a ${super.getName()} with the radius of ${this.radius}`;

    }

}

let c: Shape = new Circle(15);

console.log(c.getName());

U ovom slučaju, imamo nešto nalik na virtualni mehanizam. Promenljiva c je klase Shape, ali ukazuje na objekat tipa Circle, što dovodi do toga da se poziva metoda getName() iz klase Circle. Rezultat u konzoli bi bio: This is a Circle with the radius of 15.

Ako jedna funkcija nema navedenu povratnu vrednost, a vraća recimo string, moguće je preklopiti tu funkciju:

* funkcijom sa istim imenom koja vraća string, i takođe nema naveden povratni tip
* funkcijom sa istim imenom koja vraća string, a ima povratnu vrednost „any“
* funkcijom sa istim imenom koja vraća string, a ima povratnu vrednost „string“

1. Typescript šabloni (+primer)

U typescript-u moguće je kreirati šablonske klase koje imaju nešto nalik tipskom parametru u c++ jeziku. Unutar znakova <> navodi se oznaka tipa koji se može iskoristiti u deklaraciji klase, a pri kreiranju objekta biće zamenjen pravim tipom:

class Printer<T> {

    printValue: T;

    constructor(pv: T) {

        this.printValue = pv;

    }

    print() {

        console.log(this.printValue);

    }

}

let p = new Printer<string>("Hello");

p.print();

let p1 = new Printer<number>(123);

p1.print();

1. DOM, definicija i najčešće funkcije API-ja

Document Object Model je standardni skup objekata uređenih u strukturu kojom se predstavljaju dokumenti na web strani. Takođe, on predstavlja API zato što nudi mogućnost da se programerskim pristupom vidi i manipulše strukturom jednog dokumenta. Pre nastanka DOM-a, nije postojao jedinstven način na koji browser-i posmatraju dokumente, već je svaki čitač drugačije pristupao objektima. Stranice koje su radile u jednom čitaču, nisu radile u ostalima. Uvođenjem DOM-a i posmatranjem dokumenata preko njega, dokumenti su počeli da budu posmatrani kao stablo, odnosno hijerarhija čvorova. Svaki tag predstavlja čvor i roditelj je direktnim podtagovima (onima odmah ispod njega), a predak svim ugnježdenim tagovima unutar njega. DOM smešta svaki dokument, odnosno celo njegovo stablo, u memoriju, gde mu se direktno pristupa.

Neke od najčešćih funkcija API-ja:

* document.getElementById(„id“) – vraća html element sa zadatim id-jem
* element.getElementsByTagName(„imeTaga“) – vraća niz html elemenata koji su potomci element-a (ispod njega u DOM stablu), a pripadaju navedenom tipu taga
* document.querySelectorAll(CSS selektor) – izvršava css selekciju i vraća sve elemente koji ispunjavaju uslov selekcije
* document.querySelector(CSS selektor) – kao prethodno, samo vraća jedan element, i to prvi nađeni
* Funkcije kojima se sprovodi proces dodavanja elementa u DOM HTML stranice:
  + kreiranje novog elementa: const element = document.createElement(nazivTaga) – element ovde postoji samo u lokalnoj promenljivoj, još uvek nije u dom stablu
  + definisanje vrednost atributa – nema specijalnih funkcija, atributima se pristupa kao u OO jezicima, pomoću tačke i imena atributa (npr element.innerHTML); koje atribute element ima, zavisi od tipa tog elementa
  + dodavanje elementa u DOM – prvo se pronađe element kome želimo da nadovežemo onaj element koji dodajemo, pa zatim pomoću funkcije appendChild(promenljivaUKojojJeSmeštenElement) dodamo željeni element; element na koji se nadovezuje se pronalazi na neki od načina navedenih iznad

1. Modeli klijent – server komunikacije

Modeli komunikacije između servera i klijenta dele se na PULL i PUSH.

* Kod PULL modela klijent mora da zahteva podatke od servera. Nedostatak ovog modela je činjenica da, ukoliko je potrebno da klijentska strana stalno bude ažurirana, ona mora da neprestano šalje zahteve serveru. Server koji opslužuje mnogo ovakvih korisnika postaje preopterećen. Primeri ovog modela su:
  + HTTP (HTTPS – sigurnija verzija http-a, ima ugrađen SSL bezbedonosni protokol kojim se vrši enkripcija i digitalno potpisivanje podataka) – u ovom protokolu, klijent mora da šalje serveru http request svaki put kada želi da se određena akcija izvrši, pa tako i kada želi da od servera dobije podatke; server odgovara http response-om.
  + POP3 – protokol koji se koristi u email komunikaciji. Postoji takozvani mailbox, odnosno server čija je uloga da skladišti i dostavlja klijentu mail-ove pristigle sa druge strane. Postoji ceo protokol razmene poruka kojom klijent zahteva čitanje konkretnog mail-a.
  + RSS – stari web mehanizam kojim su klijenti mogli da prate promene na nekoliko odabranih web sajtova. RSS povremeno šalje zahteve svim izabranim sajtovima kako bi dobio ažurirane podatke.
* Kod PUSH modela, server samoinicijativno šalje promene klijentima. Bazira se na publisher/subscriber odnosu, gde su subscriber-i klijenti koji su zainteresovani za update-ove na serverskoj strani. Server predstavlja publisher-a koji promene objavljuje svim klijentima koji su subscribe-ovani. Primeri:
  + Web Push notifiacije – poruke koje se šalju sa servera do korisničkog interfejsa. Najčešće se koriste za notifikacije ili poruke na društvenim mrežama.
  + HTTP/2 – kada klijent pošalje zahtev, server pošalje ono što je traženo, ali pored toga i stvari za koje je siguran (ili je velike verovatnoća) da će takođe trebati klijentu. Na primer, u ranijim verzijama, kada klijent zatraži html stranicu, CSS stylesheet vezan za tu stranicu se nije slao odmah, već je browser morao da pošalje eksplicitno dodatni novi zahtev. Ovako, šalje se odmah. Ukoliko se pošalje nešto što klijentu nije odmah neophodno, to se čuva u takozvanom karantinu, što je neka vrsta keša, odakle se tim podacima može brzo pristupiti.
  + websocket-i (ima posebno pitanje o tome)
  + Server sent events (html5) – ista logika kao i kod ostatka push-model primera. Server je u stanju da sam obavesti klijenta.
  + Long polling – u ovom mehanizmu, nakon što klijent pošalje zahtev, on stiže do servera, otvara se veza, i server je ne zatvara sve dok ne bude imao koje podatke da pošalje. Kada se oni „pojave“, server ih šalje klijentu, poruka stiže do njega i veza se zavtvara, ali čim primi podatke, klijent šalje novi zahtev, koji ponovo otvara vezu i tako dalje... Na ovaj način, server i klijent su sve vreme u vezi, a klijent ostaje obavešten i server je ažuran što se slanja podataka tiče.