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. Zatim, nakon www, sledi ime hosta, odnosno mašine na kojoj se nalazi resurs, a .com predstavlja domen u kome je taj resurs registrovan. 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.

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

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.

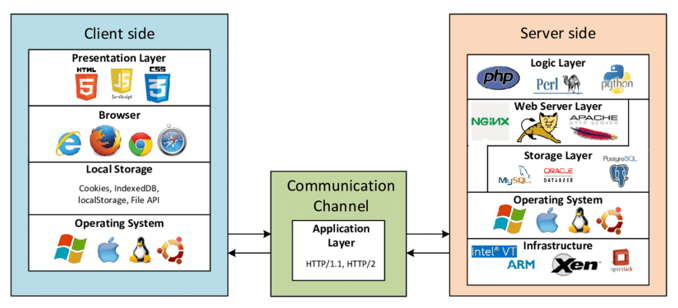
Desktop i mobilne aplikacije ne zahtevaju uvek aktivnu internet konekciju, ali je ona neophodna za rad web aplikacije, s obzirom na to da se komunikacija klijenta i servera odvija upravo preko internet mreže. Naravno, određene desktop i mobilne aplikacije, koje se koriste za komunikaciju na daljinu,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

Desktop i mobilne imaju bolje performanse, i brže se izvršavaju. Web aplikacije su sporije, odnosno odziv web aplikacije je generalno sporiji.

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.

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

1. Arhitektura web aplikacija (dijagram i cloud)

Arhitektura web aplikacije se može razložiti na tri segmenta: 

•Client side (Front-end)

* + Prezentacioni nivo: HTML, CSS, JS
  + Pretraživač: Chrome, Firefox, Safari
  + Keširanje podataka servera: Cookies, File API, localStorage...
  + Operativni sistem: Windows, MAC, Linux, IOS, Android...

•Communication channel (Komunikacioni kanal)

* Aplikativni nivo (HTTP/1.1, HTTP/2...)

•Server side (Back-end)

* Logika: PHP, Node.js, Perl, Python, ASP .NET Core
* Web serveri: Nginx, Apache, IIS...
* Skladištenje: Relacione baze, NoSQL, graf, in-memory, ORACLE, MySQL...
* Operativni sistem
* Infrastruktura: Intel, ARM...

Cloud (cloud computing / računarstvo u oblaku) predstavlja deljenje (distribuciju) kompjuterskih resursa pri zahtevu korisnika, bez njegovog učešća u održavanju istih. Ovo su u stvari kompjuterski centri (serveri) koji uz pomoć Interneta omogućavaju korisnicima da im pri slanju zahteva pristupaju i koriste njihove resurse. Deljenje memorije (cloud storage), deljenje snage računanja (computing power), itd...

1. Klijentska strana (tehnologije, izazovi)

Klijentska strana se odvija u web pretraživaču. Za njenu izgradnju koriste se:

* HTML (struktura i sadržaj)
* CSS (izgled)
* JavaScript (ponašanje).

**Progressive enchancement** – koriste se posebni fajlovi za svaku od tehnologija. Princip izrada klijentske strane gde je fokus pre svega na osnovu (HTML) da bude dostupan i pregledan na što je moguće većem broju uređaja, a onda se dalje vrši nadogradnja te osnove (kićenje) izgledom (CSS) i dalje responsiveness (JavaScript) koji je podržan od manjeg broja uređaja. Suprotan pristup bi bio izgradnja složene strane, a onda uslovlIzazovi 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.

1. Serverska strana tehnologije i 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, autentifikacije, 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.

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 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.

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.

1. Protokoli (HTTP 1.0, 1.1, 2.0, Websocket)

**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.

b. U **HTTP 1.1** verziji, 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. 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.

U **HTTP 2.0** verziji, svi zahtevi se iz tekstualnog prebacuju u binarni format, da bi se omogućilo da i starije aplikacije 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).

**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

1. HTTP - opis, metode zahteva, poruke odgovora

**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.

• **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.

**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

Kodovi oblika **1xx** su kodovi obaveštenja

Kodovi oblika **2xx** označavaju uspešno obrađen zahtev (najpoznatiji je kod „sve je kako treba“, odnosno kod 200, iza koga kao opis najčešće sledi OK)

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)

Oblik **4xx** znači da je došlo do greške za koju je „kriv“ klijent (404 – file not found, 400 – bad request)

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 5, opste, struktura strane, semanticki elementi

**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 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“>).

**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.

**Header i footer** se nalaze na početku odnosno dnu strane.

**Section** predstavlja određen deo strane koji sadrži grupu tematski povezanih jedinica. Na primer, može postojati sekcija „introduction“, sekcija „content“ i slično.

**Article** treba da bude samostalno prikazan, nevezan isključivo za konkretnu stranu i predstavlja element sa konkretnim sadržajem (kao članak u novinama).

**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.

**Details** je tag koji predstavlja neki vid dodatnih informacija. Podrazumevano, element koji sadrži ove informacije moguće je otvarati i zatvarati na klik.

**Figure** tag služi za predstavljanje celine posebnog tipa sadržaja, na primer slike, dijagrama, ilustracije i slično.

**Figcaption** tag predstavlja opis vezan za figure tag.

**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

**Mark** tag služi da naglasi određeni deo teksta (podrazumevano, da oboji pozadinu tog dela teksta drugom bojom)

**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

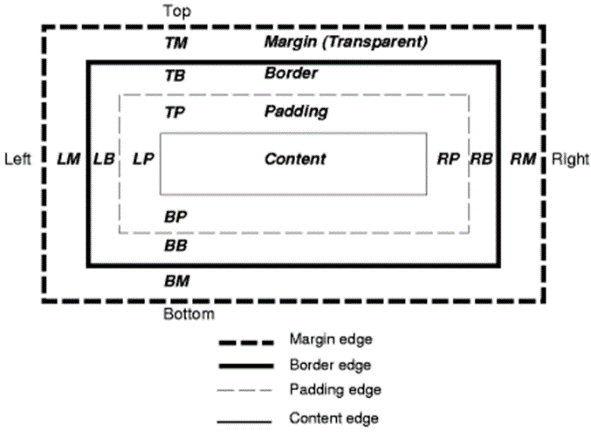
**Summary** tag vezan je za Details tag. Kada se kreira details tag, pojaviće se strelica, čijim klikom se prikazuju informacije, a ponovnim klikom zatvaraju.

**Time** tag se koristi da naznači da je na određenom delu stranice prikazano vreme.

1. CSS, opste, progresivno poboljsanje

**OPSTE:**

**CSS (Cascading Style Sheet) predstavlja jezik za formatiranje, koji se koristi za stilizovanje web stranica.** 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 navodi neki od mnogobrojnih stilskih elemenata, a zatim mu se dodeli vrednost.

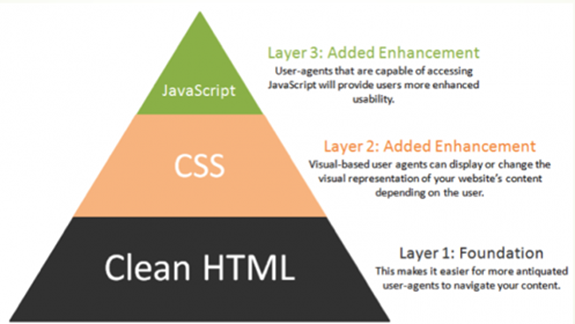


**Margin** predstavlja marginu elementa, odnosno njegovu udaljenost od susednih elemenata.

**Border** predstavlja ivicu samog elementa.

**Padding** predstavlja udaljenost sadržaja elementa od njegove ivice.

**Content** je sam sadržaj elementa.

**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.

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.

**Progressive enchancement** – koriste se posebni fajlovi za svaku od tehnologija. Princip izrada klijentske strane gde je fokus pre svega na osnovu (HTML - markup) da bude dostupan i pregledan na što je moguće većem broju uređaja, a onda se dalje vrši nadogradnja te osnove (kićenje) izgledom (CSS) i dalje responsiveness (JavaScript) koji je podržan od manjeg broja uređaja. **Suprotan pristup bi bio izgradnja složene strane, a onda uslovljavanje iste da pri korišćenju različitih uređaja isključi neke od funkcionalnosti.**

1. CSS selektori

**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;

}

**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;

}

**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;

}

**Selekcija HTML atributa** – selekcija atributa vrši se unutar uglastih zagradi. Postoje neka specijalna značenja u okviru ove selekcije.

**=** ima očekivano značenje, ime atributa sa leve strane mora imati vrednost sa desne

**\*=** 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“

**^=** atribut sa leve mora da počinje onime što je sa desne strane

**$=** atribut sa leve se mora završavati onime sa desne strane

1. Responsive dizajn strane, css grid, flexbox

**Responsive dizajn** je 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.Flexbox 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.

**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.

**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:

**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

**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

**Flex-basis** – predstavlja početnu veličinu flex elementa

**Flex-wrap** – način slaganja dece unutar roditelja (da li će preći u sledeću liniju ili ne)

**Justify-content**: poravnanje u odnosu na glavnu osu prostiranja

**Align-items**: poravnanje u odnosu na osu koja je normalna na glavnu osu prostiranja

**Align-content**: poravnanje linije potomaka u odnosu na poprečnu osu

**Order**: redosled prikazivanja dece; ko ima manji order, biće prikazan bliže početku glavne ose prostiranja

**Align-self:** individualno poravnanje konkretnog deteta

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)**– 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.

**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 notifikacije** – poruke koje se šalju sa servera do korisničkog interfejsa.

**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.

**WEBSOCKET-i**

**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.

1. Websocket protokol

* Komunikacijski protokol aplikativnog nivoa koji pruža punu dupleks bidirekcionu komunikaciju – obe strane simultano šalju svoje poruke.
* Koristi HTTP za handshake, a TCP za komunikaciju (jednu)
* Brža komunikacija od HTTP-a (manji overhead)
* Koristi Upgrade header kako bi HTTP/HTTPS-om naznačio drugoj strani da želi da se pređe na websocket protocol (handshake)
* Nakon uspostavljanja protokola, vrši se komunikacija sve dok je tcp konekcija uspostavljena

**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

1. WebRTC

* **Web Real Time Communication (WEB RTC)** je besplatni, open-source projekat koji nije samo po sebi protokol već se bazira na skupu protokola i API-ja. Ima podršku za browser-e, native aplikacije i
* Omogućava audio i video komunikaciju na web stranama uz pomoć peer-to-peer komunikacije (direktne komunikacije), bez plug-in-a i native aplikacija.
* Uspostavlja peer-to-peer konekciju između klijenata i nakon toga nije neophodan server za njihovu komunikaciju
* Problem nastaje ukoliko u komunikaciji učestvuje više od dvoje ljudi, jer se onda uspostavljaju dodatne veze

1. Prenos podataka sa klijenta na server

Klijent serveru šalje podatke preko HTTProtocol-a i to uz pomoć metoda:

**GET** – querystring je deo URL-a (adresa sadrži podatke) i samim tim je limitaran sadržaj koji se prenosi, u vidu tipo podataka koji se prenose, kao i njihove dužine. Ova metoda ne vrši nikakve izmene na serverskoj strani, već samo (kako ime kaže) nabaljva potrebe podatke sa servera. Potrebno ju je implementirati i nikad ne treba da vrati greške poput (405 - Method Not Allowed, ili 501 – Not Implemented).

**HEAD** – identična GET metodi, s tim što odgovor servera ne sme da sadrži ništa u body-ju

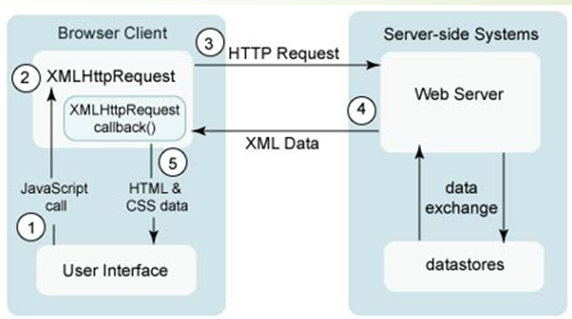
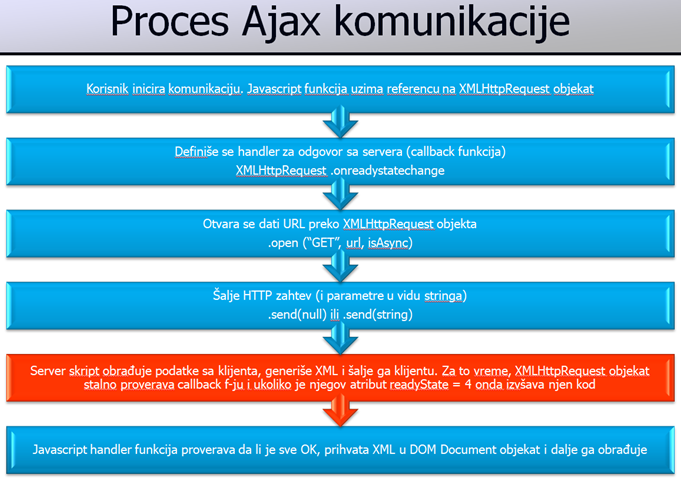
**PUT** – metoda koja se koristi kada je potrebno izvršiti neko ažuriranje podataka ili kreiranje, slična post-u

**POST** – metoda koja podatke šalje kroz svoj body i uglavnom podrazumeva obradu tih podataka na serveru (kreiranje ili izmenu podataka servera). Slanje veće količine podataka, osetljivih pdoataka, binarnih fajlova (slike, itd), podaci se ne vide u adresi, mada je moguće i da ova metoda doda query string u URL

**DELETE** – methoda koja izvršava neki vid birsanja podataka na serveru

1. AJAX

**Asynchronous JavaScript And XML** **(AJAX)** je skup tehnika za web razvijanje koje koriste mnoge web tehnologije na klijentskoj strani kako bi kreirale asinhrone web aplikacije.

Ranije je važio princip da svaki put kada se nešto menja na stranici ona mora cela ponovo da se učita, AJAX uvodi promenu koja podrazumeva da se pribavljaju samo oni podaci koji su potrebni (koji su se promenili) i da se oni ugrade u več postojeću stranu bez potrebe da se ona refresh-uje cela. AJAX znači Asinhroni JavaScript i XML s tim što u skorije vreme od servera kao odgovor klijent dobija podatke u JSON formatu a ne u XML formatu zato što je JSON kompaktiniji od XML-a i omogućava iste funkcionalnosti sa manjim protokom. AJAX princip podrazumeva postojanje objekta uključenog u stranu XMLHttpRequest objekat pomoću kojeg se šalje serveru zahtev za konkretnim podacima, u današnje vreme se to vrši pomoću fetch funkcije u okviru JavaScript-a. Taj zahtev prima server i odgovara na njega slanjem podataka klijentu u JSON formatu koji se onda ugrađuju u postojeću stranu. Dakle omogućava se dinamička promena sadržaja strane bez potrebe za ponovnim učitavanjem cele strane i omogućava se da se vrši asinhroni zahtev za podacima bez remećenja tekućeg izgleda i ponašanja strane.

1. CORS (primeri za same origin)

Da bi se omogućio veći nivo bezbednosti osmišljen je **CORS** mehanizam koji ograničava ko sve može da šalje neke tipove zahteva, najbitnije **AJAX** zahteve, nekom serveru. **Po default-u se koristi same-origin princip** zaštite gde se podrazumeva da zahtev koji je došao od klijenta koji ima isti origin kao i server može biti ispunjen dok nijedan drugi ne može, ali dodatno može da se specificira spisak nekih drugih domena ili neka drugi dodatni uslovi pod kojima se omogućuje da server odgovori na primljen zahtev.

Same origin princip funkcioniše tako da dozvoljava zahtev koji je upućen na URL sa istim protokolom, hostom i portom, ukoliko bilo koji od ovih uslova nije ispunjen smatra se da nije isto poreklo.

* + - **Adrese koje se razlikuju po protokolu**
      * [**https**://localhost:5000/webProg/resurs](https://localhost:5000/webProg/resurs)
      * [**http**://localhost:5000/webProg/resurs](http://localhost:5000/webProg/resurs)
    - **Adrese koje se razlikuju po host-u/domenu**
      * [https://**gais.**cs.umass.edu:5000/webProg/resurs](https://gais.cs.umass.edu:5000/webProg/resurs)
      * [https://**notgaia.**cs.umas.edu:5000/webProg/resurs](https://notgaia.cs.umas.edu:5000/webProg/resurs)
    - **Adrese koje se razlikuju po portu**
      * [https://localhost:**5001**/webProg/resurs](https://localhost:5001/webProg/resurs)
      * [https://localhost:**5000**/webProg/resurs](https://localhost:5000/webProg/resurs)

1. Formati odgovora servera

**To su: HTML, XML, JSON, Prototype buffer**

**XML** : (eXtensible Markup Language), slican HTML-u, sluzi za prenos podataka, a ne za njihovu prezentaciju (to radi HTML), XML elementi nisu predefinisani, vec ih korisnik sam predefinise, podrzan od strane

**JSON** :Jednostavan, kraći od XML-a, parsiranje u JS trivijalno, nezavistan od jezika, ali zato ne postoji podrška za metapodatke, ne postoje ograničenja nad tipovima podataka, nedostatak podrške za namespace.

**Prototype buffer**: metod serijalizacije struktuiranih podataka, razvio ga je Google za cuvanje i razmenu struktuiranih informacija, dizajniran je da bude manji i brzi od XML-a;

Kada se radi o komunikaciji u kojoj učestvuje „thin“ klijent odnosno klijent koji nema na svojoj strani nikakvu poslovnu logiku i ima mogućnost da samo prikaže stranicu dobijenu od servera tj. kada je server u igri zapravo u ulozi web servera onda on klijentu na dobijen zahtev odgovara slanjem stranice u HTML formatu – server šalje klijentu čitavu stranicu i klijent je prikazuje. Sve više se koristi drugačiji pristup u kome učestvuju takozvani „**fat**“ klijenti koji imaju mogućnost izvršavanja dela logike lokalno tako da je dovoljno da server njima prosledi skup podataka koje onda klijent sam ume da ugradi u već postojeću stranicu, dakle bez potrebe da server klijetnu šalje čitavu stranicu. Tada je server zapravo u ulozi web servisa i na zahtev klijenta odgovara slanjem skupa traženih podataka, ti podaci su najčešće u XML ili JSON formatu.

1. XML tehnologije (namespace, schema, xpath, xsl), obrada XML dokumenta (DOM, SAX)

**XML tehnologije** predstavljaju skup modula koji korisnicima XML-a pružaju razne korisne servise. Neke od tih tehnologija su XSL, XPath, XLink, Xquery... XLink je jezik koji omogućava dodavanje elemenata u XML dokumente tako da oni označavaju veze između resursa. XQuery je jezik za izvlačenje podataka iz XML dokumenata.

**XML namespaces** predstavljaju metodu za izbegavanje konflikta pri nazivanju tagova. Obzirom na to da u XML-u korisnik sam definiše tagove često se dešava da se jave tagovi koji imaju isti naziv ali potpuno različit smisao i sadržinu, pogotovo ukoliko se spaja više XML dokumenata sa različitim tipovima podataka u sebi.

**XML Schema** predstavlja način da se definiše struktura XML dokumenta i omogućava validacija dokumenata tj. provera da li odgovaraju prethodno definisanoj strukturi. Pre postojanja XML Scheme u ove strhe koristio se DTD jezik. XML Schema predstavlja napredak u odnosu na DTD zato što se piše u XML-u i omogućava definisanje tipova podataka za tagove, korišćenje namespace-ova i lako se proširuje.

**XPath** je jezik koji omogućava izvlačenje odgovarajućih podataka ili nizova podataka iz XML dokumenata. Sadrži biblioteku standardnih fukcija koje omogućavaju izvlačenje podataka po raznim parametrima

**XSL** je za XML kao što je CSS za HTML jer definiše način na koji će tagovi iz XML fajla biti prikazani.

XSL se sastoji od 3 dela:

* **XSLT** je najvažniji deo XSL-a, on služi za transformaciju XML dokumenta u drugi XML dokument koji će služiti za prikaz.
* **Xpath** se koristi od strane XSLT-a kako bi pronašao odgovarajuće podatke u XML dokumentu
* **XSL-FO** može da bude rezultat obrade XML dokumenta od strane XSLT-a.

**Za obradu XML dokumenta koristi se jedan od dva pristupa – DOM ili SAX**

**DOM** – čitav XML dokument se učitava i od njega se formira kompletno stablo podataka definisano tagovima. Tako kreirano stablo se koristi od strane aplikacija za pristup i manipulaciju podacima definisanim u okviru XML-a. Nedostatak je to što mora najpre čitav XML dokument da se učita u memoriju da bi mogao da se obradi i od njega kreira DOM stablo, a prednost je što je to čitavo stablo onda formirano i možemo da pristupamo bilo kom podatku u okviru njega.

**SAX** – Za razliku od DOM-a u SAX-u se ne kreira stablo, uopšte se ne vrši obrada celokupnog XML dokumenta odjednom, već se detektuje element po element prolaskom kroz dokument. U svakom trenutku postoji učitan samo tekući čvor do kog se stiglo u obradi dokumenta. Za SAX se kaže da je event driven jer detekcija početnog taga, krajnjeg taga, atributa ili sadržaja izaziva okidanje odgovarajućeg događaja. Kako funkcioniše protočno za raliku od DOM-a omogućava znatno brži pristup i manje je zahtevan.

1. Poređenje XML i HTML, XML i JSON

**XML VS HTML**

XML je jezik sličan HTML-u po svojoj strukturi, s tim što ne sadrži prethodno predefinisan skup tagova kao HTML već korisnik sam može da definiše tagove po potrebi. Takođe, XML se koristi za prenošenje podataka između klijenta i servera dok se HTML koristi za prezenatciju podataka korisniku. Moguće je izvršiti prevođenje iz XML-a u HTML i time obezbediti da se podaci koji se prenose XML-om korisniku prikažu kroz HTML.

**XML VS JSON**

**JSON i XML su slicni po tome sto:**

− Oba su self-describing i ljudi ih razumeju

− Hijerarhijski su

− Mogu da se parsiraju

− Mogu biti pribavljeni sa XMLHttpRequest-om

**Razlikuju se po tome:**

-JSON ne koristi end tag

-JSON je kraci

-JSON je brzi za citanje i pisanje

-JSON moze da koristi nizove

**Najveca razlika je u tome sto XML mora biti parsovan sa XML parserom. JSON moze**

**biti parsovan standardnom JS funkcijom.**

1. JSON

**JSON** je format podataka koji se koristi za serijalizaciju objekata i prenos podataka kroz mrežu. U današnje vreme najveći broj API-ja, REST Web servisa i AJAX aplikacija koriste JSON za prenos podataka zbog njegove kompaktnosti. JSON format je potpuno nezavisan od jezika i sastoji se od kolekcija parova ime, vrednost, pri čemu ta vrednost može prestavljati skalar, niz, objekat, niz objekata, ...

U JSON-u važe sledeća pravila za prikaz podataka – objekat je neuređen skup parova ime,vrednost, počinje oznakom { i završava se oznakom }, iza svakog imena stoji znak :, a parovi ime,vrednost međusobno su razdvojeni zarezom. Niz je uređena kolekcija vrednosti, počinje oznakom [ i završava se oznakom ] pri čemu su vrednosti međusobno odvojene zarezima. Vrednost može da bude string (u „“ navodnicima), broj, oznaka true/false, null, objekat ili niz.

1. Web server (arhitektura, tehnologije)

Pod **web serverom** se podrazumeva mašina koja prima HTTP zahteve i vraća HTTP odgovore, međutim sada je server postao znatno kompleksniji:

* može **statički** da vraća sadržaj – tj. da napravi html stranu i vraća je takvu celovitu i potpuno spremnu da je klijent samo prikaže
* i može da **dinamički** vraća sadržaj – tj. da server vraća klijetnu samo neki skup informacija u nekom formatu, najčešće je to JSON, a da ih klijent (uglavnom browser) onda ugradi u svoju stranu

**Tehnologije** koje se koriste kod web servera su:

* + - **php**
    - **.NET Framework** - može da se izvršava samo na Microsoft mašinama, što je svega 8% servera
    - **.NET Core** - može da se izvršava na bilo kojoj mašini
    - **Python** + **Django** kao njegov framework

1. ORM alati

**ORM (Object Relational Mapping) se koristi kako bi se omogućila veza između modela, odnosno klasa i objekata i tabela u bazi podataka.**

Omogućava se mapiranje tipa klasa -> tabela u bazi, property klase -> kolona u odgovarajućoj tabeli u bazi. I na taj način se omogućava konzistencija među podacima u modelu i u bazi i lakoća međusobnog mapiranja. Omogućavaju olakšanje u radu sa aplikacijama orijentisanim ka podacima time što omogućavaju da se na nivou aplikacije radi sa nekom apstrakcijom bez potrebe za brigom o samoj bazi podataka, tabelama i kolonama u njoj. Alati za ORM imaju mogućnost dva pristupa:

* **code-first** – najpre se prave modeli i na osnovu tih modela se pravi baza podataja
* **database-first** – kada treba da se napravi model za već postojeću bazu podataka, pa se onda on i pravi na osnovu nje

Neki od ORM alata su:

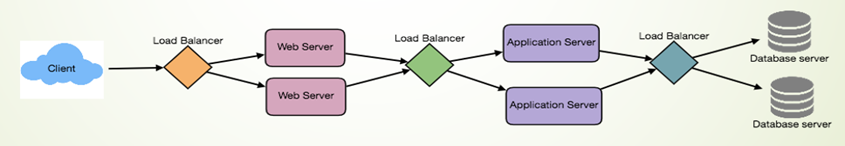
(N)Hybernate i Entity Framwork – namenjeni za .NET framework

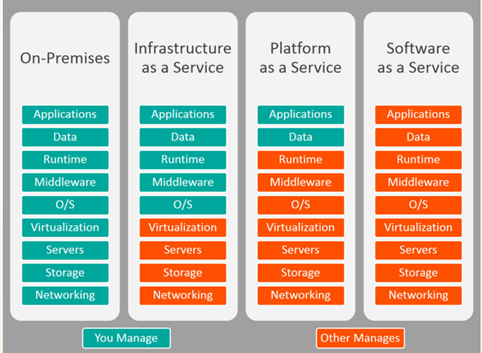
Zend i Doctrine – namenjeni za PHP

1. Raspoređivanje opterećenja servera

Veliki web serveri primaju veliki broj zahteva u svakom trenutku i to su uglavnom serveri koji se sastoje od većeg broja mašina pa je potrebno na neki način obezbediti da se zahtevi raspoređuju po dostupnim mašinama sa što većom efikasnošću i u cilju optimizacije iskorišćenosti resursa.

U tu svrhu koristi se **Load balancing** princip i **load balancer mašine**. Load balancer mašina za neki server predstavlja mašinu koja prva prima sve zahteve i odlučuje kojoj od mašina servera će da prosledi svaki od pristiglih zahteva, tu odluku donosi po nekom algoritmu.



1. Računarstvo u oblaku
   * Računarstvo u oblaku je dostupnost računarskih resursa preko mreže na zahtev klijenta.
   * Računarstvo u oblaku podrazumeva mogućnost pribavljanja resursa računarskih sistema na zahtev, pre svega prostora za skladištenje podataka i procesorske snage. Postoji više različitih nivoa, odnosno slojeva koji se rangiraju po količini resursa koji se iznajmljuju sa udaljenog mesta u odnosu na skup resursa čitavog sistema.
   * Podela na osnovu resursa koji se iznajmljuju:
     + **Infrastructure as a Service (IaaS)** – iznajmljuje se virtuelna mašina, prostor za skladištenje, load balanceri i sl. tako da klijent ne mora da vodi računa o infrastrukturi ali i dalje ima kontrolu nad operativnim sistemom, skladištenjem, aplikaciji i u nekoj meri nad mrežnim komponentama
     + **Platform as a Service (PaaS)** – podrazumeva viši nivo apstrakcije gde korisnik može da iznajmi čitavu platformu i pri tome ne mora da vodi računa o infrastrukturi, skladištenju, operativnom sistemu niti mreži ali i dalje ima kontrolu i on je taj koji upravlja samom aplikacijom
     + **Software as a Service(SaaS)** - predstavlja najviši nivo apstrakcije gde korisnik iznjamljuje čitav softver i ne mora da vodi računa ni o čemu, slobodan je da samo taj softver koristi – primeri za to su npr. dropbox, mail, ...

Računarstvo u oblaku je najčešće usluga koja se plaća „u hodu“ i iz tog razloga je potrebno voditi računa da se udluge cloud-a koriste na što optimalniji način jer se nedostaci u efikasnosti korišćenja skupo naplaćuju.

1. Serverless arhitektura

**Serverless arhitektura** je vrsta **Cloud** usluge, ona ne podrazumeva da ne postoji server (bukvalno) već podrazumeva da korisnik koji implementira neku aplikaciju ne brine uopšte o backend delu, niti o fizičkim mašinama niti o implementaciji već je na njemu samo da razvije frontend deo a da čitavu uslugu backenda po cloud principu iznajmi od nekog ko takvu sulugu nudi kao što su npr. Amazon, Google, Microsoft...

**Naplaćivanje** ovakve usluge se uglavnom vrši po količini utrošenih resursa u određenom vremenskom periodu a ne unapred u zavisnosti od količine resursa koje zahteva.

**FaaS** (Function as a Service) je deo serverless computing-a u širem smislu i podrazumeva na raspolaganju funkcije (procedure) koje omogućavaju ispunjavanje određenih zahteva.

1. Web servisi (RPC, SOAP, REST)

**Web Servisi** su softverski sistemi koji podržavaju interakciju, tj. razmeni podataka, između dve mašine preko mreže u nekom od machine readable formatu kao što su XML ili JSON. U primere web servisa spada **web Api**.

Tipovi web servisa:

* SOAP
* REST

**RPC (Remote Procedure Call)** je forma klijent-server komunikacije koja omogućava da klijent poziva procedure koje nudi server tako da to bude potpuno transparentno za onoga ko taj kod piše, za developera klijentske strane treba da bude potpuno isto da li poziva sopstvenu proceduru ili proceduru web servisa. **Klijent ima takozvane client stub funkcije koje predstavljaju lokalne verzije fukcija servisa.** Često se u okviru Proxy klase enkapsuliraju sve funkcije i svi objekti izloženi u web servisu i daje se interfejs za njihovo korišćenje od strane klijenta, koji ih onda koristi kroz taj objekat kao da je on lokalan objekat. Dakle omogućava apstrakciju komunikacije između klijenta i web servisa.

**SOAP (Simple Object Access Protocol)** omogućava da servis nudi određeni skup operacija i to se postiže korišćenjem RPC koncepta i Proxy klase koja se generiše preko WSDL-a. WSDL predstavlja jezik koji je namenjen za komunikaciju među mašinama i nije opšte čitljiv i razumljiv za čoveka jer i nije njemu namenjen. SOAP je protokol i njegove poruke se prenose preko HTTP-a, ta komunikacija se vrši u nekom formatu koji ima izgled XML-a sa posebnom šemom, ima najpre deo kojim se taj XML dokument identifinuje kao SOAP poruka, zatim deo koji sadrži informacije o aplikaciji i nakon toga i sam sadržaj poruke koja se šalje.

**REST (Representational state transfer)** je više neki arhitekturni pristup nego protokol, koristi princip da je sve resurs i da svaki taj resurs ima jedinstven ID preko kojeg mu se pristupa, taj ID je zapravo URL i pomoću njega se omogućava da klijent gađa bilo koji resurs na serveru. Glavna razlika između REST-a i SOAP-a je što se u REST-u ne vrši poziv funkcija servisa već se nad objektima koje nudi servis pozivaju HTTP funkcije. HTTP funkcije su te koje definišu šta može da sve može da se izvrši nad objektima servisa, to su metode GET, POST, PUT, DELETE.

**GET** omogućava preuzimanje odgovarajućeg objekta/niza

**PUT** omogućava ažuriranje postojećeg objekta/niza na osnovu prosleđenog, u slučaju objekta ukoliko takav objekat ne posotji kreira novi

**POST** omogućava kreiranje novog objekta (elementa niza),

**DELETE** omogućava brisanje objekta/niza.

1. RESTful servisi

**(PRVO KAZEM STA JE REST)**

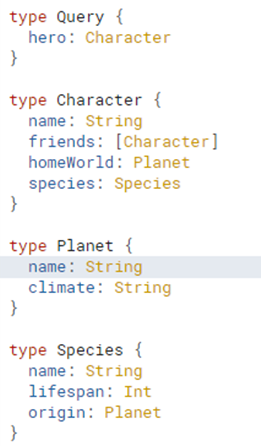
Representational state transfer (REST) je softverski arhitekturni stil koji definise set

ogranicenja kojih se treba pridrzavati za kreiranje Web servisa. Ti servisi se onda nazivaju

RESTful Web Servisi, i oni omogucavaju sistemima koji salju zahteve da manipulisu

tekstualnim reprezentacijama Web resursa korisenjem uniformnih i predefinisanih

stateless operacija.



1. GraphQL

**GraphQL** predstavlja jezik koji služi za slanje query-ja serveru za željene podatke i definiše samo način na koji će klijent da traži podatke od servera, a ne i rad sa bazom i izvlačenje tih podataka na serverskoj strani. GraphQL servis se kreira tako što se za podatke definišu tipovi i za svaki od tipova definišu se polja koja on sadrži. Za svako polje svakog tipa zatim se piše funkcija koja definiše šta se dešava kada se dobije query koji zahteva to polje. Dakle, nije GraphQL taj koji zna sam kako treba da interaguje sa bazom da bi izvukao podatak za određeni query već se način obrade primljnog query-ja definiše tako što se za svako polje svakog tipa definiše šta se radi kada se ono nađe u query-ju. Query u GraphQL-u omogućava da klijent navede tačno koji podaci su mu potrebni jer može da navodi liste različitih ili istih tipova koji su mu potrebni u jednom query-ju i pored toga može i da ih ugnježdava – ako npr. neki tip sadrži polje koje je takođe opisano tipom onda je moguće definisati tačno koja od polja tog ugnježdenog tipa želimo. Time se omogućava da se u okviru samo jednog zahteva specificira sve što nam je potrebno uz to omogućujući da se za tražene podatke egzaktno specificiraju koja polja su nam potrebna, bez potrebe za nužnim pribavljanjem svih polja nekog tipa, to je velika pretnost u odnosu na REST koji prvenstveno zahtevao slanje više zasebnih zahteva za nešto što se u GraphQL-u rešava jednim zahtevom i uz to REST vraća celokupne podatke za traženi resurs dok je u GraphQL-u moguće specificirati tačno pta nam od podataka treba (Analogija je da preko REST-a jednom odemo i potražimo hamburger i dobijemo ceo paket – zemičku, meso, krastavac, paradajz i sir iako nam možda sve to nije trebalo a ako nam treba i palačinka onda moramo da odemo ponovo i opet dobijamo ceo paket – palačinku, krem, plazmu, šlag i čokoladni preliv, a u GraphQL-u odemo jednom i kažemo tačno šta nam treba i upravo to i dobijemo – npr. zemičku sa mesom i krastavcem i palačinku sa kremom jer smo samo to i hteli). U GraphQL se pored query-ja koji se koristi samo za pribavljanje podataka, kao GET funkcija, definišu i mutacije koje se koriste za modifikovanje već postojećih podataka na serveru i dodavanje novih, pri čemu je struktura ista kao kod query-ja samo se navode parametri koji definišu podatke koji se dodaju/menjaju, mogućnosti su takođe iste, jednom mutacijom možemo da promenimo/dodamo veći broj podataka i to specifično podatke koje želimo. Mutacija nakon što izvrši tražene izmene vrati izmenjene podatke tako da zahtevaoc može i odmah da dobije feedback o promeni.

1. SPA

**SPA (Single Page Application)** je web aplikacija koja je bazirana na samo jednoj web strani koja na interakciju sa korisnikom reaguje komunikacijom sa RESTful web servisom koji joj vraća odgovarajuče, potrebne podatke u JSON formatu koje ona obrađuje i ugrađuje u tekuću stranicu bez da se ta stranica refresh-uje u bilo kom trenutku niti da se pređe na neku drugu stranicu. Dakle, za razliku od standardnog pristupa gde se svaki put kada se nešto menja stranica ponovo učitava u celosti ili se prelazi na potpuno novu stranicu kod SPA je u pitanju samo jedna stranica koja po AJAX principu komunicira sa Web API-jem i omogućava sve što je potrebno za efikasnu i brzu komunikaciju sa klijentom.

1. PWA

**PWA (Progressive Web Application)** je standard nastao iz SPA, odnosno ako SPA ispunjava PWA standarde onda se naziva PWA.

**PWA** podrazumeva aplikaciju koja se izvršava u browser-u ali to korisniku ne deluje tako, on može aplikaciju da skine na svoj računar, da joj pristupa preko ikonice i da je koristi i offline i sve tako da aplikacija u potpunosti izgleda kao da je native samo što je zapravo web aplikacija. To se postiže uz pomoć service worker-a koji predstavljaju zasebne niti koje izvršavaju skripte u browser-u u pozadini, nezavisno od neke konkretne web stranice, kako bi obavile poslove koji nisu vezani ni za jednu web stranicu niti interakciju sa korisnikom. One omogućavaju pozadinsku sinhronizaciju PWA aplikacije tako da ona uvek ima ažurne podatke i takođe omogućavaju push notifikacije koje će korisnika obaveštavati o promenama u aplikaciji čak i kada je on ne koristi.

Service worker-i omogućavaju PWA da odaju osećaj native aplikacije i da imaju performanse na visokom nivou kao i da jesu native zadržavajući prednosti web aplikacije kao što su mali zahtevi za prostorom za skladištenje podataka, real-time ažuriranja i napredni search engine-i.

Dodatna prednost PWA u odnosu na native aplikacije je to što su native aplikacije vezane za platformu i moraju da se menjaju svaki put kada se desi promena na platformi što nije slučaj sa PWA, takođe, distribucija native aplikacije vezana je za app store i potrebno je čekati na odobrenje pre nego što aplikacija postane dostupna korisnicima, dok su PWA linkabilne, izvršavaju se na browser-u i hostuju se na nekom linku, potrebno je samo poslati taj link i svako može da skine aplikaciju sa njega, bez bilo kakve potrebe za app store-om niti čekanjem na odobrenje. Uz aplikaciju ide Web App Manifest koji predstavlja JSON fajl koji sadrži informacije o celoj aplikaciji, koje resurse koristi i generalno podatke o konfiguraciji.

1. Mikorservisi
   * **Mikroservisna** **ahitektura** je tip servisu-orijentisane arhitekture, strukturni stil, koji aplikaciju uređuje kao kolekciju manjih servisa (procesa) sa sopstvenim funkcionalnostima i bazama koji svi zajedno služe za izvršavanje zajedničog zahteva.
   * **Svaki servis je manja aplikacija sa bazom**
   * **API Gateway je posrednik između servisa i klijenta**
   * **Prednosti**:
     + Brži razvoj (manja kompleksnost pojedinačnih servisa)
     + Paralelizacija razvoja (jedan tim jedan servis)
     + Sopstveni skup tehnologija (jedan mikroservis == technology stack)
     + Nezavisan deploy
     + Nezavisno skaliranje
   * **Mane**:
     + Komplikacije pri međusobnoj komunikaciji
     + Složenije tranzakcije / konzistentnost zbog većeg broja baza podataka
     + Testiranje
     + Složeni deploy celine
   * **Koristi Websockets**
2. gRPC

**gRPC** je Google-ova tehnologija za **Remote Procedure Call** koja za komunikaciju ne koristi XML ili JSON već bitsku serijalizaciju što kominikaciju čini mnogo bržom i jednostavinijom. Koristi se uglavnom za međusobnu komunikaciju servera (ne u komunikaciji klijent-server) jer je način proenošenja podataka takav da je razumljiv samo za mašine, ne i za korisnike. Bitska serijalizacija zahteva mnogo manju procesorsku moć za obradu i mnogo manji protok za prenos zato se gRPC dosta koristi kod IoT uređaja koji su uglavnom mali, slabi uređaji ili uređaji koji se nalaze na udaljenim mestima te su povezani na mreže sa malom propusnošću pa je optimalan protok ključna stvar. Kod gRPC-a takođe komunikacija može da bude bidirekciona pri čemu komunikacija u oba smera funkcioniše potpuno nezavisno

1. Virtualizacija

**Virtuelizacijom** se omogućuje da se izbegne potreba za fizičkim okruženjem već da se kreira simulirano ili virtuelno okruženje računara. Hardverska virtuelizacija npr. podrazumeva se na nekoj mašini odmah iznad sloja hardvera umetne sloj virtuelizacije koji omogućava da se na tom istom hardveru u naizgled odvojenim okruženjima podižu različiti operativni sistemi pri čemu svakom od njih izgleda kao da je on sam i jedini koji se izvršava na nekoj hardverskoj platformi. Dakle kreira se virtuelno okruženje koje je izolovano, a zapravo, realno deli hardver i njegove resurse sa skupom drugih virtuelnih okruženja. Monitor koji omogućava kreiranje i upravljanje virtuelnim okruženjima ili virtuelnim mašinama zove se Hypervisor i on je zadužen da upravlja realnim hardverskim resursima i dodeljuje ih virtuelnim mašinama tako da one nemaju osećaj njegovog posotojanje već da imaju osećaj kao da pristušaju direktno hardverskim resursima mašine na kojoj se izvršavaju. U zavisnosti od implementacije postoje takozvana „bare meral“ rešenja u kojima se Hypervisor izvršava direktno na hardveru i kontroliše da i direktno upravlja njegovim resursima, drugo rešenje podrazumeva da se na mašini domaćinu izvršava neki operativni sistem koji je u direktnoj komunikaciji sa hardverom i koji je taj koji se izvršava direktno i zapravo na toj mašini a da se Hypervisor nalazi iznad njega i preko tog host OS-a upravlja virtuelnim resursima koje dodeljuje virtuelnim mašinama koje onda mogu da se ugrađuju iznad njega.

1. Kontejnerizacija, Docker, compose

**Kontejnerizacija** **je virtuelizacija na nivou operativnog sistema**. U ovom slučaju jedan operativni sistem, odnosno njegov kernel omogućava postojanje više odvojenih korisničnik prostora u okviru njega. Ti prostori se nazivaju kontejneri i oni predstavljaju svojevrsno unapređenje u odnosu na virtuelne mašine jer kontejneri postoje u okviru jednog operativnog sistema, dakle dele sve skupe resurse, i samim tim su jeftinije i manje obimno rešenje od virtuelnih mašina, a opet daju osećaj procesu koji se izvršava u okviru njih da se izvršava samostalno na realnoj mašini a ne u kontejneru. S tim što proces koji se izvršava na običnom operativnom sistemu vidi sve resurse računara na kome se izvršava, a onaj koji se izvršava u kontejneru vidi samo sadržinu tog kontejnera i uređaje koji su mu dodeljeni.

Kontejnerizacija je počela da se primenjuje pojavom **Docker**-a. Docker predstavlja skup PaaS (Platform as a Service) proizvoda koji omogućavaju razvoj kontejnerizovanih aplikacija, tj. aplikacija koje ne zavise od okruženja u kome se izvršavaju čime se znatno ubrzava razvoj aplikacije jer se izbegava potreba za podešavaljem okruženja i potreba za prilagođavanjem aplikacije okruženju već se daje mogućnost da se aplikacija razvija u kontrolisanom okruženju kontejnera ali tkao da se garantuje da će ona moći da se uniformno izvršava u svim okruženjima eliminišući time i kasnije potrebe dopodešavanja aplikacije kako bi radila u durgim okruženjima i probleme da aplikacija različito radi ili ne radi uopšte za neka okruženja u odnosu na ono na kome je razvijana. Kada se napiše aplikacija pomoću Docker-a njenim kompajliranjem dobija se takozvani Docker Image, koji može da se interpretikao kao klasa kojom se definiše ta aplikacija, na osnovu te „klase“, odnosno Image-a se onda pokretanjem u različitim okruženjima kreiraju različite instance odanosno kontejneri koji suštinski predstavljaju istu aplikaciju ali su prilagođeni okruženju u kome se izvršavaju. Docker praktično služi za to da upakuje aplikaciju u Image tako da kroz njega aplikacija može da pristupi tačno određenom skupu resursa koji su joj potrebni u okviru operativnog sistema fizičke ili virtuelne mašine.

1. Orkestracija kontejnera (Kubernetes, Docker Swarm)

Orkestracija kontejnera je pojam koji se koristi u sistemima u kojima se koristi arhitektura mikroservisa i kontejnerizacija. Dakle, kada imamo aplikaciju ili uslugu, opštije rečeno, koja je podeljena na neki skup servisa određene veličine i imamo određeni broj kontejnera na raspolaganju za korišćenje od starne tih servisa, onda je potrebno vršiti orkestraciju tih kontejnera odnosno raspodelu kontejnera po servisima, kontrolu svakog od kontejnera, ... i to sve tako da se omogući što bolje funkcionisanje celine. Postojanje kontejnera nam uslovno rečeno omogućava da napravimo neku lokalnu privatnu mrežu kontejnera koji međusobno razmenjuju podatke i pozivaju fukncije a spolja se vide kao celina kojoj se pristupa preko jednog entry point-a. Ceo taj sistem treba orkestirati, treba izvršiti podelu kontejnera po servisima, definisati načine komunikacije među njima, alokaciju i raspodelu resursa, dinamičku raspodelu kontejnera po servisima tako da se u zavisnosti od potreba sistema vrši skaliranje opterećenijih servisa na račun onih koji nemaju opterećenje, preraspodelu preostalih kontejnera u slučaju otkaza nekog od hostova i još mnogo toga. Zbog masovnosti obaveza koje podrazumeva orkestracija uglatnom se koriste neki alati koji je olakšavaju, neki od njih su Kubernetes i Docker Swarm. Kada se koriste ovakvi alati najčešće se u JSON fajlu opisuje konfiguracija aplikacije u kojoj se piše odakle se dobavljaju Docker Image-i, kako se uspostavlja veza između kontejnera i gde su smešteni podaci vezani za taj kontejner, kontejneri se onda raspoređuju po hostovima, kada se kontejner nalazi na hostu alat za orkestraciju brine o njegovom životnom ciklusu shodno upustvima iz konfiguracionog fajla. Dakle, najopštije rečeno orkestracija predstavlja automatizaciju procesa raspodele kontejnera, njihovog upravljanja, skaliranja i međusobnog povezivanja. Iako obavljaju naizgled isti posao Kubernetes i Docker Swarm to rade na različite načine, pri čemu važi pravilo da Kubernetes predstavlja neki zlatni standard u ovoj oblasti zato što ima mogućnost automatskog skaliranja (ne mroamo mi da razmišljamo koliko nam instanci nekog kontejnera treba, Kubernetes će da skalira broj kontejnera tako da nikada ne dođe do otkaza datog servisa), regularne provere ispravnosti svih elemenata sistema što ne omogućava Docker Swarm ali upravo iz tih razloga je Kubernetes mnogo kompleksniji i zahtevniji u odnosu na Docker Swarm.

1. Web Assembly

**Web Assembly** je jezik niskog nivoa nalik na asembler ali ne zavisi od mašine na kojoj se izvršava već je vezan za web platformu. Dakle kao neka forma univerzalnog asemblerskog jezika za web. Koristi se u web-u i omogućava, zbog toga što je jezik niskog nivoa i sadrži kompaktne binarne instrukcije, da se aplikacije izvršavaju sa performansama prlibližnim native aplikacijama. Izvršava se na istoj virtuelnoj mašini na kojoj se izvršava i JavaScript i iz razloga što je arhitektura takva da je ta VM odvojena od skupa API-ja iz Web Assembly-ja mogu da se pozivom web API-ja koriste iste funkcionalnosti kao što mogu kroz JavaScript kod. Time se postiže kompatibilnost ostalih jezika višeg nivoa sa JavaScript-om jer ti jezici mogu da se kompajliraju u Web assembly i da se onda njima dobijaju iste mogućnosti kao sa JavaScript-om jer se izvršavaju u istom okruženju i imaju pristup istom skupu resursa. Izvršenje Web Assembly-ja podržavaju svi browser-i. (Web Assembly je prilagođen čak i za debagiranje i gledanje od strane programera, jer iako je u osnovi u binarnom formatu sadrži i tekstualni format koji je čitljiv za čoveka i služi u te svrhe. Razlog zašto je Web Assembly po performansama toliko efikasan je to što je Virtuelna mašina koja ga izvršava optimizovana tako da koristi zajedničke hardverske mogućnosti koje ima veliki broj platformi.)

1. Blazor

**Blazor** predstavlja framework koji omogićava kreiranje interaktivnih User Interface-a za klijentsku stranu web aplikacije u C# tj. koristeći .NET umesto JavaScript ali uz mogućnost da se iz tog C# koda na lak način pozivaju JavaScript API-ji i biblioteke. Dakle, ne gubi se podrška velike web JavaScript zajednice ali se dodaje mogućnost korišćenja C# biblioteka i koda i logike koja se koristi za serversku stranu u klijentskoj strani aplikacije. Blazor omogućava da se klijentski kod izvršava:

Na serveru (Blazor Server) – Tada se aplikacija hostuje na .NET Core serveru u Razor formatu tako da je klijent jednostavan a čitava logika se izvršava na serveru. Browser na klijentu prevlači jednostavnu stranicu, a UI događaji sa klijenta se, preko SignalR konekcije koja predstavlja sistem prenosa poruka u realnom vremenu, vrećaju serveru koji ih obrađuje i vraća promene UI nazad klijentu, te promene se onda integrišu u njegov DOM.

U browser-u pomoću Web Assembly-ja (Blazor web assembly) – ovakva implementacija podseća na SPA jer je cela logika na klijentu. Samim tim je velišina stranice koja se prevlači na klijentovu mašinu prilično veća od one kod Blazor Server-a ali zato Blazor Web Assembly aplikacije postižu veliku brzinu rada jer se mahom izvršavaju na samoj klijentovoj mašini. Činjenica da se pravi C# kod izvršava klijentu (samo što se on kompajlira u web assembly) omogućava da se u kodu koriste biblioteke i delovi koda sa serverske strane aplikacije

Blazor može da radi u svim modernim browser-ima i ne zahteva nikakve dodatne plug-in-ove.

1. Blockchain \*\*NEW\*\*

**Blockchain** je baza podataka koja se ne nalazi na jednom mestu, već je čine manje baze (blokovi) koje su međusobno digitalno povezani, a koji sadrže informacije o digitalnim transakcijama bilo koje vrste: od vlasničkih listova, preko podataka iz knjige rođenih, do ugovora kojim se regulišu autorska prava. Prilikom njihove razmene nema nikakvog regulatora osim same mreže koja sadrži informacije o svim transakcijama koje su ikada izvedene. Dakle, za razliku od klasične online baze podataka, Blockchain tehnologija omogućava komunikaciju sa nekoliko računara (servera) između kojih se transakcija obavlja.

**JavaScript**

1. Javascript (JS), opste

**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, 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. JS tipovi (Primitivni, Objekti)

Podela tipova podataka:

**Primitivni tipovi**:

1) string

2) number

3) boolean

4) undefined

5) null

6) symbol (ES2015)

**Primitivni tip** vrednosti se čuvaju u brzom delu memorije koja se naziva “stack” i imaju nepromenjivu (“immutable“) vrednost. To znači da nakon stvaranja i skladištenja u memoriju ta vrednost na tom mestu se više ne može menjati. Upravo je njihova nepromenjivost razlog što je za poredjenje promenjivih koje sadrže primitivan tip vrednosti, dovoljno da porede samo vrednosti.

**Referentni tipovi:**

Objekat

**Referentani tip** vrednosti se čuva u sporijem delu memorije koja se naziva “heap”, i za razliku od primitivnog tipa, referentni tip vrednosti može da se menja tokom vremena. Dodeljivanje referentne vrednosti promenjivoj izvršava istu akciju kao i kod primitiva: vrednost zauzima mesto u memoriji, a pointer promenjive se definiše da ukazuje na to mesto u memoriji.

1. Promenljive

* **JavaScript prepoznaje različite tipove podataka, ali nije strogo tipiziran jezik**, što znači **da jedna ista promenljiva u programu može menjati tip** vrednosti koju sadrži.
* JavaScript ne zahteva deklarisanje promenljivih unapred, kao drugi (strogo tipizirani) jezici.
* Deklaracija promenljive se obavlja sa: var, let, const kljucnom reci
* Var se sad ne koristi zbog bug-a sa scope-om.
* Let se najcesce koristi. Ubacen je u JS od 2015te.
* Const se koristi za promenljive koje ne menjaju vrednost.
* Nakon deklaracije promenjliva nema vrednost vodi se kao undefined.
* Dodela vrednosti promenljivoj sa = .
* JavaScript najpre prođe kroz kontekst (funkciju) i "evidentira" sve promenljive koje su deklarisane u njemu, a tek onda izvršava naredbe (u koje spada i operator dodele, odnosno definicija promenljive).

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 funkcijskog 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. razlika izmedju deklaracije i izraza fje (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 – Podizanje fja

**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).** Za var promenljive važi da se pri deklaraciji ujedno i inicijalizuju na undefined. Hoisting se primenjuje i na promenljive deklarisane sa let i const s tim da one za razliku od var-a nemaju defaultnu vrednost (kao var undefind).Desice se greska ukoliko se pokusa citanje promenljive deklarisane sa let i const pre inicijalizacije!! **Ako se tada referencira po imenu, ime će se prepoznati, ali će doći do greške koja kaže da je promenljiva neinicijalizovana.**

Primer:

console.log(num); // Throws ReferenceError exception as the variable value is uninitialized

let num = 6; // Initialization

**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 fje

**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. Nizovi, ugradjene metode

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. DOM, definicija i najcesce funkcije API-ja

**DOM (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. Closure i konteksti izvrsenja fja

**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 kada se u funkciji koristi neki identifikator, on prvo potraži u lokalnom scope-u, pa ako se ne nađe onda u jednom scope-u 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 (+ klasa sa privatnim članovima)

**IIFE** **(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(lambda)** 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();

**(+ klasa sa privatnim članovima)**

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 korišćenja 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:

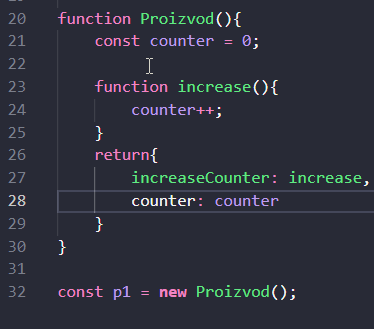
let person = {... fullName: function(){ return this.firstName + „ „ + this.lastName},...}

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

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

1. Module šablon za definisanje klasa **\*NEW NEW\***

**MODULE PATERN!**



Module patern sluzi da wrap-uje set promenljivih i funkcija u jednom scope-u.

* Koristi se za definisanje objekata i specificiranje promenljivih i funkcija kojima se može pristupiti izvan opsega funkcije.
* Izlažemo određena svojstva i funkcije kao javne i takođe možemo ograničiti opseg svojstava i funkcija unutar samog objekta, čineći ih privatnim.
* To znači da se tim promenljivim ne može pristupiti van opsega funkcije.
* Možemo postići podatke koji skrivaju apstrakciju koristeći ovaj obrazac

**Znaci pravi se closure u funkciji increase koja koristi spoljnu promenljivu counter**

„Proizvod“ odnosno ime „klase“ je spoljna funkcija, increase je unutrasnja funkcija funkcije Proizvod odnosno njen property.

Ispod nje se nalazi objekat literal. Ukoliko se ime poklapa sa onim sto vracamo referencu moze se zapise samo jednom, primer:

return {

increase,

counter

}

**Umesto:**

return {

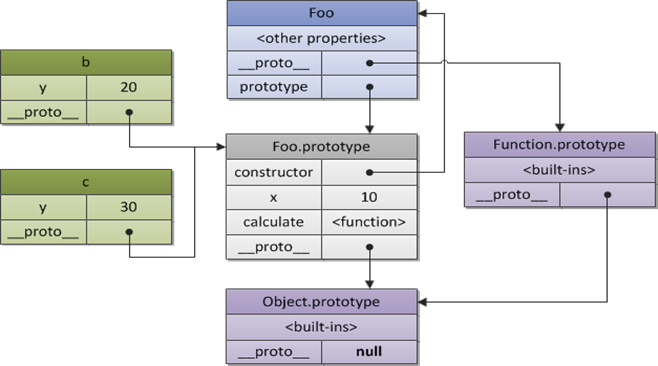
increase: increase,

counter: counter

}

TO JE TZV. „SINTAKSNI SECER“ jebo ga secer.

1. Prototipovi (+ Prototipski lanac)



**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**.

**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:**

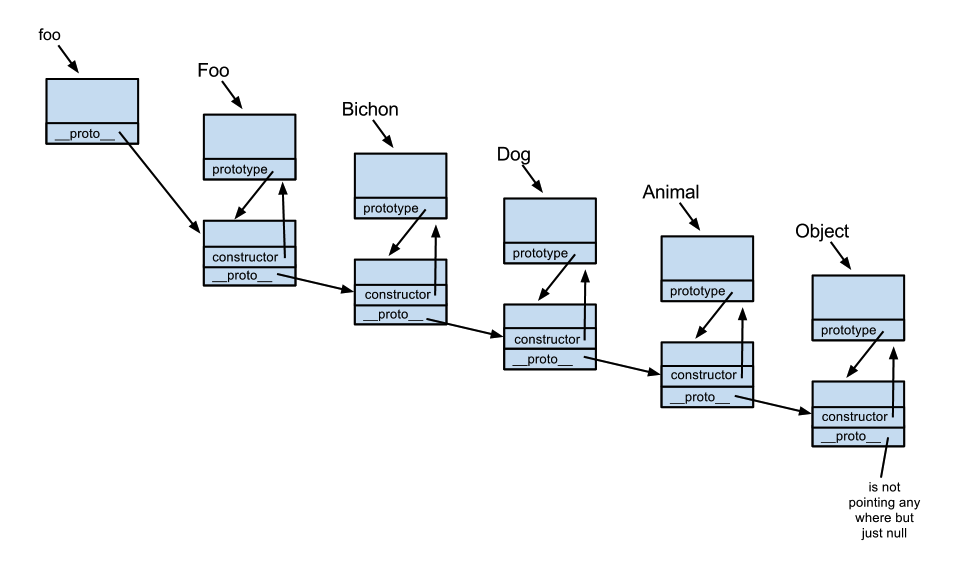
Product.prototype.menjajKolicinu = function(kol) { this.kolicina += kol;} - dodavanje metode

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:

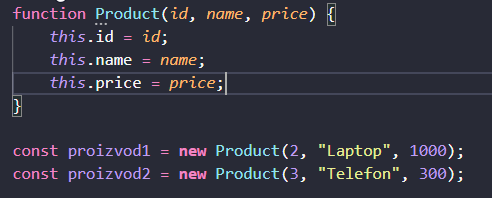
- „prod1 instanceof Product“ vraća „true“

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

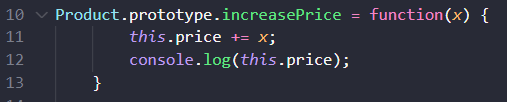
U ES6 uvodi se pojam klasa, čijim se korišćenjem odstranjuje potreba za eksplicitnim korišćenjem prototipova i pisanjem konstruktorskih funkcija. Klase imaju sopstvenu funkciju po imenu „constructor“ u kojoj se postavljaju atributi i metode koje će objekat imati. Uvođenje klase ne znači da se prototipovi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

1. Klase preko prototipova (dodavanje metoda na prototip) **\*NEW NEW\***

**Klasa preko prototipa.**



**Dodavanje metoda na prototip**:



Ovim dodavanjem konstruktorska funkcija je ostala ista.

U toku izvrsenja programa je dodata funkcija.

Product – konstruktorska fja.

Product.protorype – prototipski objekat

prototype – svojstvo, postoji samo kod fja, kod objekata ne

proizvod1.\_\_proto\_\_ - prototipski objekat

1. ES6 Klase , nasleđivanje (i 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

**NASLEDJIVANJE + 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. ES6 Property članovi klase

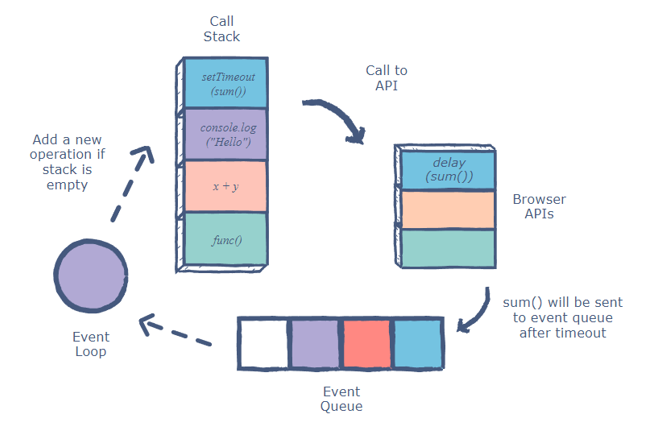


1. ES6 moduli

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

1. Asihrono programiranje, Callback fje

**Srz** **asinhronog programiranja** **je EVENT LOOP.**



**Call Stack** je stek poziva funkcija. Kad se neka funkcija pozove stavlja se na stek kad se zavrsi skida se sa steka.(isto i za operacije tipa x+y).

Kada dodje do poziva neke asinhrone funkcije (pr. setTimeout ili fetch) ta funkcija se prosledjuje nekom API-u (browseru, nodejs-u itd..) i tek tu mogu da se asinhrono izvrsavaju stvari jer su API odovojeni. Dok se tu asinhrono obavljaju stvari CallStack tece/radi normalno odnosno ne blokira se da ceka izvrsenje asinhrone funkcije.

Kada API zavrsi on tu fju koja mu je prosledjena na izvrsenje stavlja u Event Queue. Kako se koji asinhroni dogadjaj zavrsi API ga stavlja u taj red (Event Queue).

**EventQueue ne okida dogadjaj ( ne prosledjuje dalje ) dok god je CallStack pun!!**

**Znaci CALLBACK funkcija nece da se prosledi preko EventLoop-a CallStacku dok god je CallStack pun. Tek kad se CallStack isprazni onda se prosledjuje.**

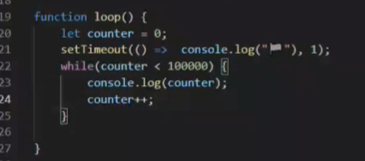
Primer:

SetTimeout(() => {

Console.log(„nesto“);

}, 2000)

**Ovo ne znaci da ce da se „nesto“ ispise u konzoli za tacno 2 sekunde, nego za minimalno 2 sekunde u zavisnosti kolko je CallStack pun!!!!.**

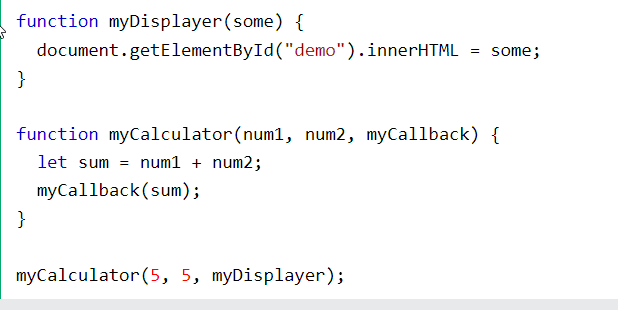


Ovo je dobar primer da se razume. Kad istekne timer od 1 milisekunde treba da se odstampa zastavica u konzoli, medjutim ispod toga je while petlja sa 10 000 iteracija.Znaci call stack ce biti zauzet dok se izvrsava ta while petlja, zastavica nece da se ispise na ekranu dok se ne zavrsi petlja odnosno dok se ne isprazni call stack. Za to vreme ce zastavica da ceka u EventQueue-u. Znaci zastavica ce da se iscrta na ekranu kad counter dostigne vrednost 10 000, iako je asinhrona metoda.

Ako se ugnjezdavaju asinhrone funkcije dolazi do kreiranja „**callback hell-a**“. **Primise** se uvodi da ne bi doslo do toga.

**Callback funkcija je funkcija koja se prosledjuje kao parametar druge funkcije da bi se naknadno pozvala u njoj(posle izvrsenog nekog dela koda, ili u nasem slucaju kad istekne timer).**

Callback funkcija se prosledjuje drugoj funkciji kao parametar bez „()“ odnosno prosledjuje se samo pokazivac fje.



1. Event loop

Ovo sam reko iznad gore, ne znam zasto je odvojio ova dva pitanja kad su isti djavo.

1. Promise, Async await

**Promise je ugradjen u JavaScript. To je prost wraper objekat koji sadrzi jednu CallBack funkciju**.

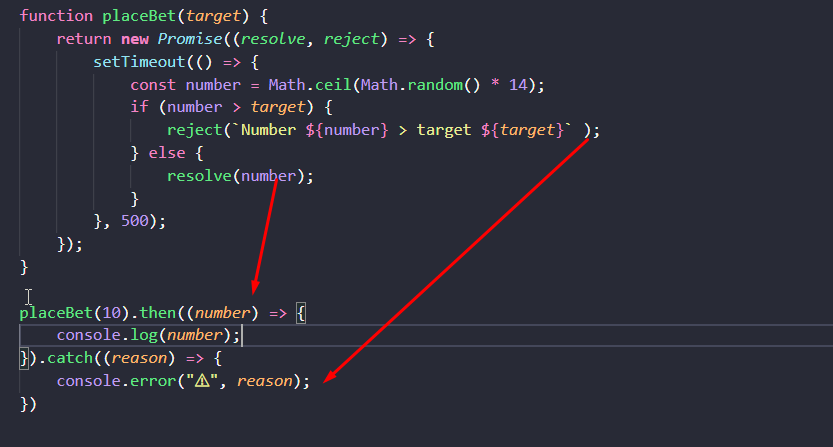
Ta callback funkcija ima dva paramtera: **resolve**(uspesno) i **reject**(neuspesno, greska).

„Promise je obecanje da ce to nekada u buducnosti da se izvrsi xD“

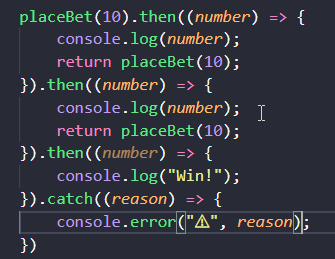
Promise (kad se poziva funkcija u kojoj se on nalazi) ima then, catch i finaly.

Then za uspesno, catch za greske.

Ideja sa promise-om je da se zameni callback hell.



Ako hocemo da resimo problem sa callback hell-om odnosno umesto ugnjezdavanja asinhronih funkcija mozemo ovo:



Catch obuhvata sve errore za svaki reject objekta.

Dok then mora da se pise za svaki.

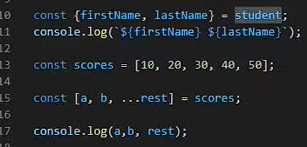
**Kljucnom reci „async“ ispred function formiramo asinhronu funkciju.**

Sa **await** ispred poziva funkcije koja u deklaraciji ima **async** cekamo da se ta funkcija zavrsi pa nastavljamo dalje.

Odnoso kao da je sinhrona akcija

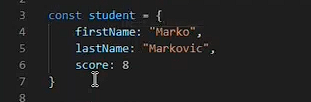
1. JS operatori (spread, op dekonstrukcije, desktrukciona dodela)

**Destrukciona dodela** ( Destruction assingment ) sluzi za izdvajanje nekih property-a (recimo iz objekta) i njihovo smestanje u promenljivu.



**Spread operator** uzima objekat i razlaze ga na clanove, a zatim ga sklapa u novi objekat. Malte ne, pravi kopiju objekta (rasklapa ga pa ponovo sklapa).

Ukoliko je objekat primitivan tip kreira njegov clone odnosno kreira novu promenljivu sa novom memorijskom lokacijom. Dok ako property u objektu nisu primitivni onda ce po referenci. Odnosno izmena novog objekta se primenjuje i na stari.





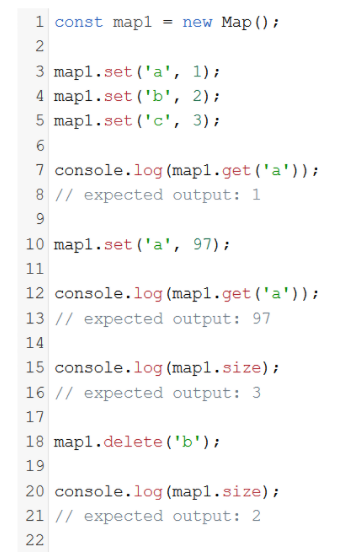
Takodje sa spread operatorom moze se dodati novi property pored originalnih.



Student3 ce imati dodatatni property „school“ sa vrednoscu „CS“ pored originalni „firstName“, „lastName“, „score“.

1. JS Strukture (Map vs Object, Proxy)

**Map** objekat ima **kljuc-vrednost parove** i pamti redosled ubacivanja kljuceva. Bilo koje vrednosti mogu da se uzimaju za kljuceve i vrednosti za razliku od Objekta gde kljuc moze da bude string broj ili symbol. Za razliku od objekta map-e koriste funcije get i set za vracanje i postavljanje kljuceva i vrednosti.



Map-e su iteratibilne, moze da se koristi for…of statement.

Kada je potrebno da se stalno dodaju ili uklanjaju parovi kljuca-vrednosti map-e su bolje po performansama od objekata.

Map-e ne podrzavaju serilizaciju nativno, ali je moguce to odraditi(JSON.stringify, JSON.parse).



Funkcija size -> vraca broj kljuc-vrednost parova.

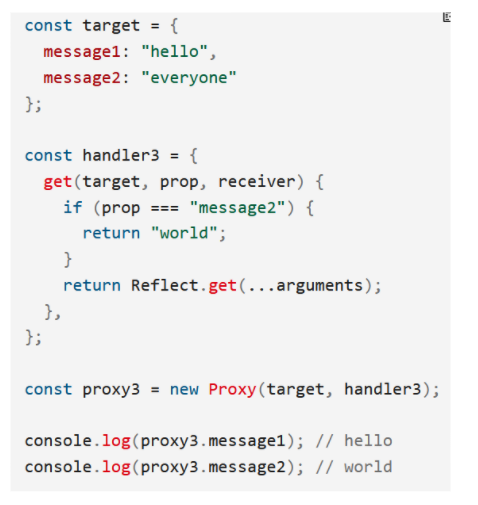
**Proxy** :

Proxy objekat omogucava da se napravi objekat koji moze da se koristi umesto originalnog objekta i pri tom da mu se redefinisu neke operacije kao sto su get set i definicija propertija.

Za kreiranje proxy-ja potrebno je dva parametra:

Target : originalni objekat od koga zelimo da napravimo proxy

Handler objekat koji definise operacije koje ce da se redefinisu.



**TypeScript**

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.

**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.

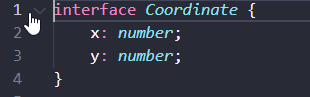
1. TS interfejsi, aliasi, poređenje tipova, unija

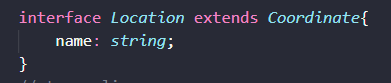
* **INTERFEJS**

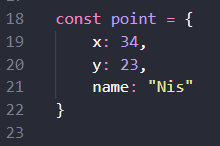
**Interfejs** u typescriptu se ne prevodi u javascript, tako nesto ne postoji u javascript-u.

Interfejs je za razliku od ostalih oop jezika **ne predstavlja** manifest ponasanja (kako ce nesto da se ponasa) nego samo opisuje izgled nekog objekta.

Interfejsi mogu da se nasledjuju.





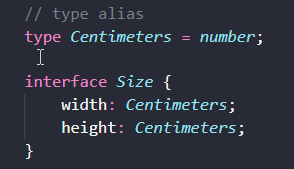


* **ALIAS**

**Alijasi** tipa kreiraju novo ime za tip. Pseudonimi tipova su ponekad slični interfejsima, ali mogu da imenuju primitive, unije, tuple i sve druge tipove koje biste inače morali da pišete ručno.

**Aliasing** zapravo ne stvara novi tip - stvara novo ime koje se odnosi na taj tip.

Da ne bi kucao widthInCentimeters umesto number, stavim alias za number koji se naziva centimeters, e onda recimo dodje neki drugi kolega programer koji treba da kreira objekat on ce da zna da sam ja zeleo da height i width budu u centimetrima i unosice centimetre a ne recimo piksele.(dobra stvar)

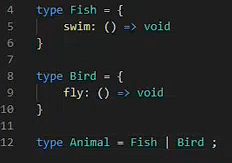


* **UNIJA**

**Unija** sluzi za kreiranje sopstvenih tipova, odnosno kreiranje tipa kombinacijom vec postojacih tipova.



U ovom primeru id moze biti ILI number ILI string.



* **POREDJENJE TIPOVA ???**

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");

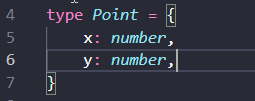
p.print();

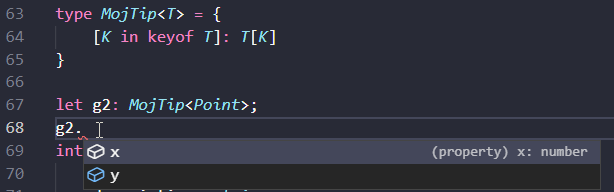
let p1 = new Printer<number>(123);

p1.print();

1. Napredni tipovi (upiti u defiiniciji tipa)

Pisem sve ovo od ispod, suzavanje, uniju, typeof, keyof, i narrowing





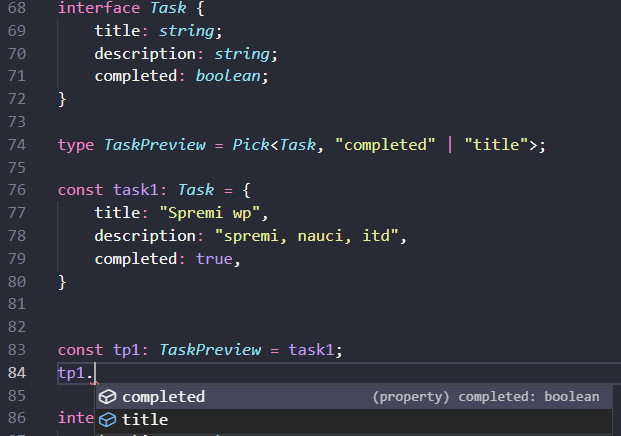
Ovim ce MojTip imati iste property-e od tipa Point sa atributima koji su istog tipa. Odnosno ako je x i y u Point tipa number, u tipu MojTip ce isto x i y biti number.

[K in keyof T] : T[k] – „Za sve kljuceve iz T ( u nasem slucaju Point) koristi iste tipove kao sto su u T (Point).

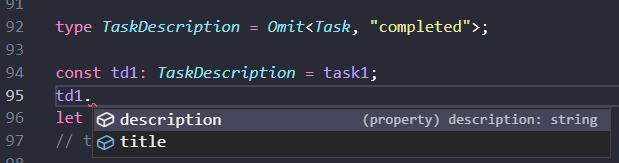
Ako iz nekog tipa zelimo da izaberemo samo odredjene property-e, pr. Iz Task da izvucemo completed i title. Kreiramo novi type TaskPreview tako sto koristimo „**Pick**“.

type TaskPreview = Pick<Task, „completed“ | „title“>

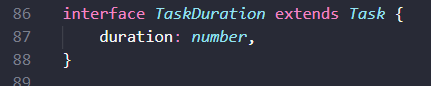
Ovim ce da se kreira novi tip TaskPreview sa izabranim propertijima iz tipa Task.



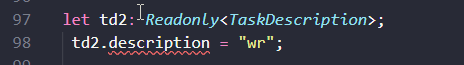
Ako zelimo da napravimo novi tip na osnovu nekog starog i da koristimo sve property-e osim nekog odredjenog uradicemo to sa **Omit**.



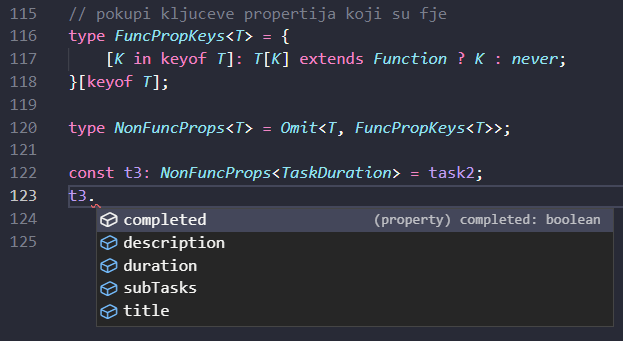
Ako zelimo da pored propertya jednog type-a imamo jos neki property koji nam je potreban kreiracemo jos jedan type(interface) koji nasledjuje prethodni i u njemu cemo dodati propertye koje zelimo. Ovim ce novi property imate sve property-e starog Type-a plus nove propertye. Isto kao sa klasama radi se nasledjivanje sa kljucnom reci **extends**.



Sa **Readonly**<(nekiTIP ili interfejs)> kada pravimo neku promenljivu iz nje ce moci samo da se citaju atributi ne i da se upisuje. OVO JE SAMO U COMPILE TIME-u.



Ako zelimo da kreiramo tip na osnovu nekog drugog tipa ali bez property-funkcija koje on sadrzi:



1. Sužavanje tipova

**Narrowing ili suzavanje tipova je proces u kome se neka unija tipova suzava na sve manji broj tipova dok na kraju ne ostane samo jedan.**

**Postoji vise „vrsta“ narrowinga:**

* **Typeof (type guards)** – primer ispod.
* **Truthiness narrowing** – koirsiti &&, ||, if statements, Boolean negations (!) za narrowing.
* **Equality narrowing** –koristi ===, !==, ==, != za narrowing.
* **In operator narrowing** – proverava da li objekat sadrzi property sa zadatim imenom.
* **Instanceof narrowing** – koristi prototype chain za narrowing.

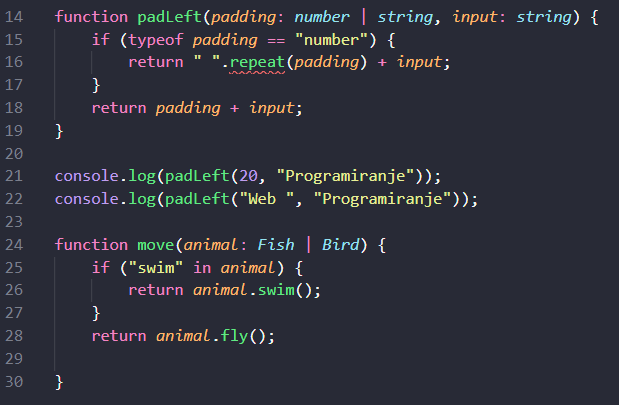
**typeof** provera da li je neki **primitivni tip ili je objekat**. NE SLUZI DA PROVERIS IZ KOJE JE KLASE OBJEKAT KAO U C#.!!

TypeScript expects this to return a certain set of strings:

* "string"
* "number"
* "bigint"
* "boolean"
* "symbol"
* "undefined"
* "object"
* "function"

type Fish = { swim: () => void };

type Bird = { fly: () => void };



Ovaj drugi primer ispod proverava da li je animal fish ili bird i u zavisnosti od toga poziva odredjenu fju koja se nalazi u top type-u.

**VISE NA LINKU:**

[**https://www.typescriptlang.org/docs/handbook/2/narrowing.html**](https://www.typescriptlang.org/docs/handbook/2/narrowing.html)

1. Štićenje tipova (type guards)

Treba se napise primer za padLeft sa prethodnog pitanja, u sustini pise se za typeof (type guard).

1. Keyof operator

Type je „isto“ sto i interfejs.

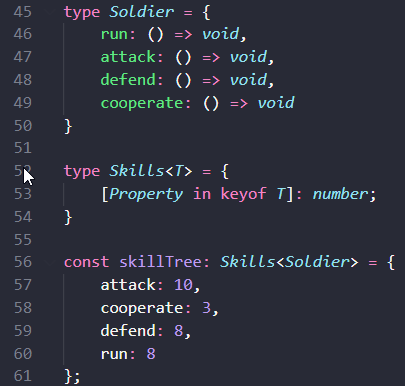


1. Mapirani tipovi

Napravimo neki tip recimo soldier koji kao property-e (kljuceve) ima neke funkcije.

E sad ako hocemo da napravimo tip skills koji je genericki odnosno moze da se odnosi na vojnika zombija itd. Kad kazemo „[Property in keyof T]: number“ to znaci da ce skills imati sve property-e od T (recimo ako T bude nas vojnik, skills ce imati: run, attack...) a ovim „: number“ kazemo da ce ti property-i biti NUMBER, ne kao u originalu funkcije, nego ce biti tipa number i moci ce da im se dodeli vrednost.

Ovim smo malte ne MAPIRALI odnosno transformisali jedan tip u drugi.



1. Diskriminišuće unije

<https://www.typescriptlang.org/docs/handbook/2/narrowing.html#discriminated-unions>

<https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html>

1. Potpisi funkcija (alias za potpis, višestruki potpis)