35. Modeli klijent-server komunikacije

Mogu da budu pull i push principa.

**Pull** princip podrazumeva da klijent eksplicitno šalje zahtev i traži nešto od servera. Na tom principu funkcionišu protokoli:

- HTTP, HTTPS (sigurna verzija HTTP-a) – klijent šalje serveru zahtev za web stranicom

- POP3 – klijent traži od mail servera da mu pošalje mail-ove koje ima na raspolaganju za njega

- RSS (Rich Site Summary )– to je web feed koji omogućava korisniku da prati promene na većem broju web sajtova tako što RSS često zahteva informacije o promenama od svih tih sajtova i to sve stavlja u jedan stream i to se servira korisniku

Kako bi klijent imao konstantno ažurne informacije po pull principu simulira se push izuzetno čestim pull zahtevima od klijenta ka serveru što je neefikasno jer opterećuje međusobni saobraćaj time što će klijent neprestano da šalje zahteve za ažuriranjima podataka čak i kada nije bilo promena. To se rešava korišćenjem push principa.

**Push** princip podrazumeva da server sam šalje klijentu podatke kada ima novih informacija (promena) bez potrebe da klijent to eksplicitno zatraži slanjem zahteva. Funkcioniše po publisher/subscriber principu gde se klijenti pretplaćuju na neke informacione kanale servera i onda im serveri na čije su kanale pretplaćeni šalju informacije svaki put kada dođe do neke promene. Suština je da je pošiljalac taj koji inicijalizuje prenos podataka a ne primalac. Tipovi push tehnologija:

* web push – protokol baziran na verziji 2 HTTP-a koji omogoćuva da se šalju notifikacije klijentu u real time kada se desi neki događaj (poziv, poruka, ...) i to čak i van konteksta same web stranice za koju je informacija vezana, dakle mi kao korisnik možemo da zatvorimo taj tab u browser-u i da radimo nešto sasvim dugo a taj server može da nastavi da nam šalje noticikacije. Iako to ima svoje prednosti i može da bude korisno takođe predstavlja i ozbilju bezbedonosnu pretnju
* HTTP/2 server push – kada klijent zatraži neku stranu server mže da pretpostavi da će klijentu pored te strane uskoro da bude potreban još neki drugi skup stvari i iz tog razloga mu sve te stvari i pošalje zajedno sa tom stranom. Klijent traženu stranu preuzima, a ostatak podataka smešta u „karantin“. Kada mu nakon toga budu potrebni neki podaci on najpre proverava taj „karantin“ i tek ako tih podataka nema već tamo onda šalje zahtev za njihovo pribavljanje od servera
* websockets – omogućavaju dvosmernu full-duplex komunikaciju koja povezuje klijenta i server i ta veza se ne raskida. Može da se vrši dvosmerna komunikacija slanjem poruka. Zgodni su za realizaciju neke realtime razmene podataka među većim brojem korisnika. Npr. mogao bi da se realizuje neki sistem u kome posotji dokument koji istovremeno mogu da edituju svi učesnici u projektu i da svi istovremeno vide promene koje je napravio svaki od učesnika
* long polling – nije u potpunosti push sistem, tj. koristio se pre nego što su postojale push tehnologije. Klijent šalje serveru pull zahtev ali server ne odgovara na njega odmah već tek kada ima nove podatke koje treba da pošalje, kada odgovori klijentu on odmah šalje novi pull zahtev koji će da bude obrađen na isti način

36. Websocket protokol

Omogućava bidirekcionu full-duplex vezu između servera i klijenta. Oslanja se na TCP na transportnom sloju i preko jedne TCP konekcije omogućava obostranu realtime razmenu podataka između servera i klijenta, iz tog razloga je brži od HTTP-a jer omogućava taj push princip, da server po potrebi šalje klijentu potrebne informacije bez potrebe da ih klijent pre toga eksplicitno zatraži. Tako omogućava živu tj. realtime komunikaciju koja je obostrana i brza. Da bi se uspostavila socket konekcija između klijenta i servera klijent najpre po HTTP- ušalje zahtev, takođe po HTTP-u se vrši handshake procedura nakon čega se uspostavlja websocket konekcija i dalja komunikacija se vrši po njoj.

37. WebRTC

Web Real Time Communication omogućava peer-to-peer raltime razmenu podataka između dva klijenta bez potrebe servera pri čemu preneti podaci mogu da budu audio, video tipa ili da budu neki opšti podaci. To je open source rešenje koje nije samo po sebi protokol već se bazira na skupu protokola i API-ja. Ima podršku za browser-e, native aplikacije i IoT uređaje i omogućava njihovu realtime komunikaciju bez potrebe za instaliranjima nekih plug-in-ova. (Ima malu manu koja doduše može da se reši instaliranjem add-on-a, ali je svakako mana, leak-uje pravu IP adresu kroz VPN tunele i time ozbiljno narušava bezbednost)

38. Prenos podataka sa klijenta na server (slanje podataka serveru ppreko HTTP-a)

Sa klijenta na server se podaci prenose korišćenjem HTTP-a, koriste se osnovne HTTP metode – get, post, put, delete, ... Npr. po get i post metodi podaci se pokupljaju iz form elementa i smeštaju se u asocijativni niz koji ima oblik takav da je naziv (tj. name) ključ za koji je vezana vrednost (odnosno value) odgovarajućeg elementa. Tako ordanizovan niz podataka se šalje serveru koji taj niz može da izmapira na odgovarajuće objekre.

* GET zahtev – koristi URL za prenos podataka, asocijativni niz oblika ključ1=vrednost1&ključ2=vrednost2 se smešta iza inicijalnog URL-a na koji je nadovezan ?
* POST zahtev – URL ostaje nepromenjen, podaci se šalju zasebno u telu poruke. Ovakav princip omogućava slanje veće količine podataka u odnosu na get zahtev, takođe omogućava slanje poverljivih podataka jer oni neće biti vidljivi svima kroz URL i omogućava slanje binarnih fajlova kao što su na primer slike

39. AJAX

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.

40. CORS

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

Npr. za <http://elfak.ni.ac.rs/studenti/osnovne-studije>

<http://elfak.ni.ac.rs/studenti/master-studije> je same origin jer su sva tri uslova zadovoljena i u pitanju su samo različite putanje, isti je slučaj i sa

<http://elfak.ni.ac.rs/novosti/profesori>

Dok ukoliko se menja bilo šta iz skupa protokol, host port smatra se da je u pitanju cross origin kao kod

<https://elfak.ni.ac.rs/studenti/osnovne-studije> - drugačiji protokol

<http://pmf.ni.ac.rs/studenti/osnovne-studije> - drugačiji host

<http://elfak.ni.ac.rs:100/studenti/osnovne-studije> - drugačiji port (podrazumevani port za http je 80)

41. Formati odgovora servera

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.

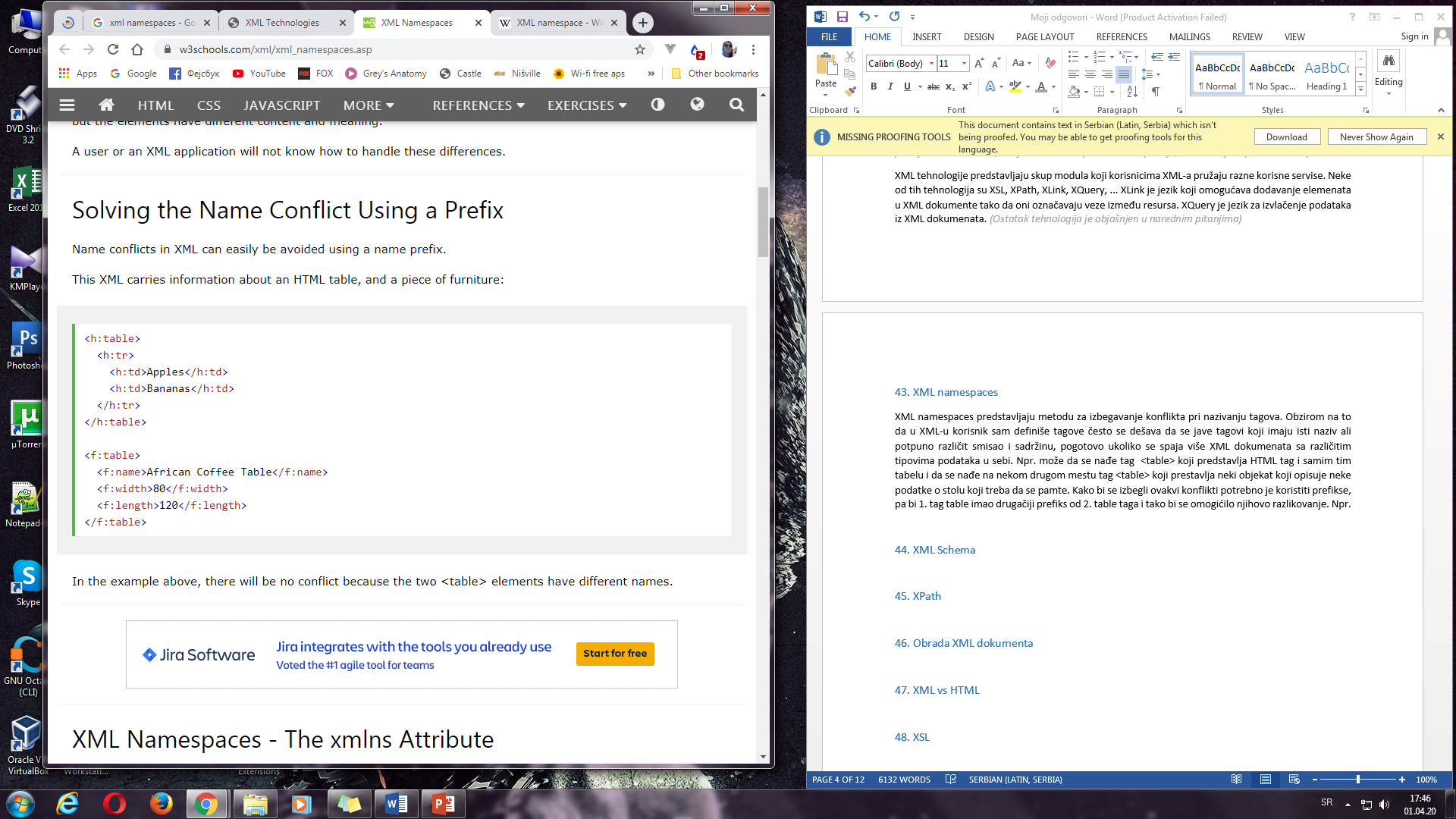
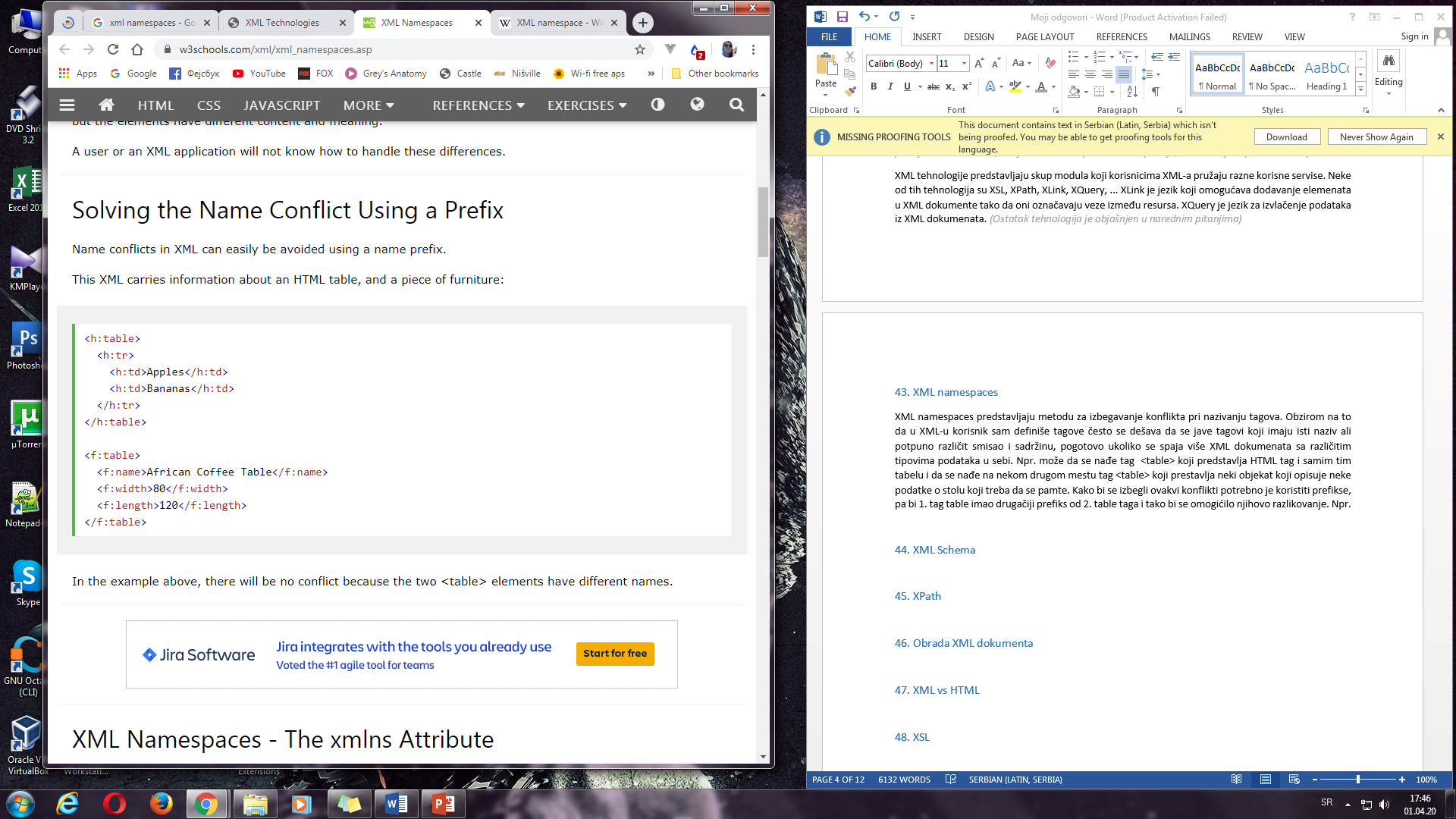
42. XML tehnologije

XML (eXtensible Murkup language) 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 definisati pravila koja treba neki XML fajl da ispuni po svojoj strukturi, validacija ispunjenosti tih pravila se vrši direktivom <!DOCTYPE ... > na početku fajla za koji želimo da izvršimo validaciju. *(Možda i nije baš za ovo pitanje, ali ga neka i ovde jer je ovo prvo pitanje vezano za XML, ovaj deo se nalazi ponovo u 47. pitanju, tamo mu je zapravo i mesto)*

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. *(Ostatak tehnologija je objašnjen u narednim pitanjima)*

43. XML namespaces

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. Npr. može da se nađe tag <table> koji predstavlja HTML tag i samim tim tabelu i da se nađe na nekom drugom mestu tag <table> koji prestavlja neki objekat koji opisuje neke podatke o stolu koji treba da se pamte. Kako bi se izbegli ovakvi konflikti potrebno je koristiti prefikse, pa bi 1. tag table imao drugačiji prefiks od 2. table taga i tako bi se omogićilo njihovo razlikovanje. Npr.



Za prvi slučaj tj. HTML tag uveden je prefiks h, a za drugi prefiks f i time je omogućeno razlikovanje. Kako bi se omogućilo korišćenje prefiksa potrebno je definisati namespace, svaki prefiks označava pripadanje jednom namespace-u. Namespace se definiše direktivom *xmlns:prefix = „URI“*  koja se piše kao atribut u tagu za koji želimo da važi, a samim tim onda važi i za sve ugnježdene tagove koji se u njemu nađu tako da se svi ugnježdeni tagovi sa istim prefiksom posmatraju kao deo istog namespace-a. URI naveden u okviru namespace-a se ne koristi od strane parser-a za proveru informacija već je njegova svrha da obezbedi da svaki namespace ima jedinstveno ime.

44. XML Schema

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.

Sintaksa:

<schema>*Ovde može da se definiše namespace i time se omogući definisanje više šema u jednom fajlu*

<element name = „ime taga“ type = „tip“>

*Unutra se definišu podaci koji su ugnježdeni, odnosno tagovi koje kao ugnježdene mora da sadrži definisan tag*

</element>

45. XPath

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. Generalno izgleda tako što se navode putanje kako bi se definisalo tačno koji podatak iz dokumenta nam je potreban. Te putanje u mnogome podsećaju na standarde putanje koje se koriste za pristup fajlovima u fajl sistemu s tim što postoji i mogućnost korišćenja standardnih funkcija za dodatnu specifikaciju parametara.

46. Obrada XML dokumenta

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.

47. XML vs HTML

XML (eXtensible Murkup language) 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 (jer on za to služi) korisniku prikažu kroz HTML (jer on za to služi). To se postiže korišćenjem XPath za selekciju željenih podataka iz XML dokumenta i njihovim navođenjem u HTML tagovima.

48. XSL

XSL je za XML kao što je CSS za HTML jer definiše način na koji će tagovi iz XML fajla biti prikazani. XML ima problema kod prikaza zato što njegovi tagovi nisu prethodno definisani pa pretraživači ne znaju kako treba da ih prikažu (što kod HTML-a niej slučaj, on ima skup tačno definisanih tagova i za njih se tačno zna kako treba da se prikažu, s tim što je u CSS-u moguće definisati dodatne specifičnosti o prikazu). XSL se sastoji od 3 dela:

* XSLT
* XPath
* XSL-FO

**XSLT** je najvažniji deo XSL-a, on služi za transformaciju XML dokumenta u drugi XML dokument koji će služiti za prikaz, kroz XSLT moguće je dodati/izbaciti elemente ili atribute, preurediti i sortirati elemente i birati tačno kako će se i šta tačno prikazivati od podataka definisanih inicijanim XML dokumentom koji je ušao u obradu.

**XPath** se koristi od strane XSLT-a kako bi pronašao odgovarajuće podatke u XML dokumentu, tj. koristi se za navigaciju kroz elemente i atribute priloženog XML dokumenta.

XSL-FO može da bude rezultat obrade XML dokumenta od strane XSLT-a. Tako generisani XSL-FO (Formatting Object) se prosleđuje aplikaciji koja konvertuje ovaj dokument u čitljiv format, često u PDF dokument.

49. 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.

50. JSON vs XML

I JSON i XML se koriste za prenos podataka između klijenta i servera na web-u. Sličnosti između njih su to što i jedan i drugi omogućavaju proizvoljno definisanje struktura objekata od strane korisnika i sadrže hijerarhijsku strukturu. Ali razlikuju se po tome što je JSON znatno kraći, ne koristi krajnje tagove već početak i kraj elementa označava sa {}, JSON može da definiše nizove, takođe JSON se izuzetno trivijalno prevodu u JavaScript klase dok je kod XML-a potreban komplikovaniji parser koji će pomoću DOM-a ili SAX-a da odredi preslikavanje. Takođe JSON nema mogućnost definisanja namespace-ova a XML ima, JSON nema mogućnost definisanja tipova podataka i nema podršku za metapodatke. JSON se danas znatno više koristi u web komunikaciji od XML-a jer se JSON znatno lakše parsira i generaln je brži i lakši za korišćenje, pogotovo AJAX baziranim aplikacijama.

51. Web server

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 koji može da se izvršava na bilo kojoj mašini
* Python + Django kao njegov framework

52. Troslojna arhitektura web aplikacije

Podrazumeva organizaciju web aplikacije u tri sloja – sloj prezentacije, sloj logike i sloj perzistencije. Sloj prezentacije direktno komunicira sa klijentom, on je taj koji prima HTTP zahtev u smeru komunikacije od klijenta ka serveru i koji genreiše HTML ili JSON odgovor u suprotnom smeru komunikacije. Sloj logike je taj koji sadrži logiku domena tj. klase i objekte, dok je sloj perzistencije zadužen sa trajno čuvanje podataka, omogućuje mapiranje iz formata objekata i klasa u format baze podataka i obrnuto. U detaljnijem opisu:

* Prezentacioni sloj – najčešće je organizovan u MVC formatu – Model, View Controller. Controller predstavlja ulaznu tačku, njemu se korisnik obraća pri slanju zahteva i on je taj koji orkestrira sve, koji kaže šta treba da se radi, zove servis koji izvlači Model koji predstavlja klase i objekte koji sadrže podatke. Na osnovu podataka iz Modela Controller ažurira View koji predstavlja html stranu koju korisnik vidi (tj. u šablon html strane (View-a) se ugrađuju dobijeni podaci). S tim što u slučaju web api-ja, prezentacioni sloj nema View, on šalje Model klijentu a klijent je taj koji Model ugrađuje u View
* Sloj perzistencije – koristi ORM (Object Realtional Mapping) kojim se omogućava mapiranje tipa klasa -> tabela u bazi, property klase -> kolona u odgovarajućoj tabeli u bazi i na taj način se omogućava da se po prijemu zahteva uvek prosleđuju ažurni podaci jer se modeli iz sloja logike direktno mapiraju i povezuju sa bazom podataka u ovom sloju

53. 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

54. 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, povećanja propusnosti, minimizacije vremena potrebnog za obradu zahteva i izbegavanja preopterećenja bilo koje konkretne mašine servera i na taj način povećanje pouzdanosti i pristupačnosti. 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. Taj alogritam može biti različit, može da se uvek bira mašina koja u datom trenutku opslužuje najmanji broj klijenata, mašina koja je geografski najbliža (tj. ona koja ima najkraći response time), mašina sa najmanje saobraćaja, može da koristi Round Robin princip ...

55. Računarstvo u oblaku

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. Princip suprotan Cloude Computing-u je On Premise princip koji podrazumeva da kada neko želi da oformi i koristi čitav jedan sistem on ga i sam pribavlja i sam njime upravlja – npr. neka osoba kupi serverski računar, konfiguriše ga, smesti u neke svoje prostorije i u potpunosti upravlja fizički svim potrebnim resursima. Nasuprot tome računarstvo u oblaku nudi mogućnost da se neki ili svi resursi iznajme sa nekog udaljenog mesta. Postoje 3 sloja Cloud Computing-a, of najnižeg ka najvišem to su:

* 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.

(I serverless computing spada u cloud)

56. Serverless arhitektura

Serverless arhitektura je vrsta Cloud usluge, ona ne podrazumeva, kako bi se po bukvalnoj interpretaciji naslova mislilo, da ne postoji server 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.

57. Web servisi

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. Web API predstavalja Microsoft tehnologiju, njegov prezentacioni deo (MVC) ne sadrži View komponentu, već se podrazumeva da je klijent kompleksan, tj. da je on taj koji zna da dobijene podatke od web API-ja u formatu JSON ili XML ugradi u već postojeću HTML stranu. Još jedan od koncepata web servisa je RPC.

Tipovi web servisa:

* SOAP
* REST

58. RPC

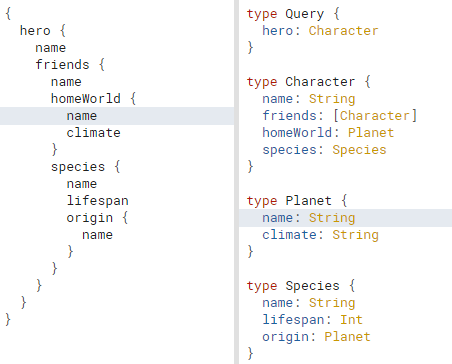
Remote Procedure Call je forma klijent-server komunikacije koja omogućava da klijent poziva procedure koje nudi server tako da to bude potpuno transparentno za onoga ko taj kod piše, za developera klijentske strane treba da bude potpuno isto da li poziva sopstvenu proceduru ili proceduru web servisa. Klijent ima takozvane client stub funkcije koje predstavljaju lokalne verzije fukcija servisa odnosno jedna client stub funkcija predstavlja interfejs prema jednoj funkciji web servisa koja može da se pozove. 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.

59. SOAP servisi

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. Koristi se da bi se opisali podaci (Type) zatim operacije u servisu i poruke koje se prenose (Interface), uz to i formati tih poruka i detalji o protokolima web servisa koji se koriste (Binding) i da bi se definisao skup sistemskih poziva koji su izloženi web protokolima (Service). SOAP je protokol i njegove poruke se prenose preko HTTP-a, ta komunikacija se vrši u nekom formatu koji ima izgled XML-a sa poebnom 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.

60. RESTful

REST 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. Svaka od tih funkcija može biti izvršena nad pojedinačnim objektom ili nad nizom objekata, pri čemu ako URL sadrži ID onda se zna da se to odnosi na jedan konkretan objekat, a u suprotnom se odnosi na niz. 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), dok DELETE omogućava brisanje objekta/niza. GET je jedina funkcija koja ne menja stanje u bazi. Za prenos podataka koristi JSON.

61. 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.

poslat query:

{

hero {

name

# Queries can have comments!

friends {

name

}

}

}

vraćen JSON:

{

"data": {

"hero": {

"name": "R2-D2",

"friends": [

{

"name": "Luke Skywalker"

},

{

"name": "Han Solo"

},

{

"name": "Leia Organa"

}

]

}

}

}

<https://graphql.org/learn/> i <https://graphql.org/learn/queries/>

62. 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 klijnet-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.

63. Mikroservisi

Web sever može biti organizovan u formi monolitne arhitekture ili arhitekture mikroservisa. Ako je organizovan kao jedna velika aplikacija koja sadrži kompletnu logiku i sve delove usluge zajedno onda je u pitanju monolitna arhitektura. Ukoliko je sa druge strane usluga koju nudi web server podeljena na manje celine koje su zasebno implementirane kao posebne aplikacije sa sopstvenom bazom koje međusobno komuniciraju onda je u pitanju arhitektura mikroservisa i svaka od tih posebnih aplikacija predstavlja jedan mikroservis. Klijenta ne dotiče kakva je organizacija web server-a, on u svakom slučaju ima jednu tačku pristupa, samo što je u slučaju monolitne arhitekture to upravo i pristup toj jednoj celokupnoj aplikaciji a kod arhitekture mikoservisa postoji API Gateway koji predstavlja posrednika između servisa i klijenata. Prednosti mikroservisa u odnosu na monolitnu arhitekturu su to što je razvoj znatno brži, svaki mikroservis je neuporedivo manji od celokupne monolitne aplikacije a kako su mikroservisi nezavisni moguće je da razvoj svakog servisa vrši zaseban tim paralelno sa razvojem svih ostalih mikroservisa, takođe, skup tehnologija koji se koristi za razvoj jednog mikroservisa može da bude specifičan za svaki od njih, može se vršiti nezavisan deploy svakog od mikroservisa, takođe, omogućava se da se u slučaju preopterećenja jednog ili dela mikroservisa izvrši skaliranje samo tog/te grupe mikroservisa koji su prebukirani bez potrebe skaliranja čitave aplikacije (uključujući i delove koji nisu prebukirani) što je slučaj kod skaliranja web servera sa monolitnom arhitekturom. (Skaliranje – kreiranje dodatnih instanci servisa koji je bave istim poslom kao i skalirani servis kako bi se smanjilo opterećenje). Sa druge strane postoje i neki nedostaci korišćenja mikroservisne arhitekture, a to su činjenice da je međusobna komunikacija mikroservisa jednog web servera izuzetno kompleksna, da je, kako svaka aplikacija koja predstavlja mikroservis sadrži svoju zasebnu bazu podataka, vršenje transakcija znatno kompleksnije jer je potrebno voditi računa o očuvanju konzistencije u svim bazama podataka, da je testiranje otežano jer su funkcionalnosti na taj način podeljene i da je deploy svih mikroservisa kao jedne celine znatno složeniji od deploy-a servera kao jedinstvene monolitne aplikacije.

64. 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.

65. 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 sevice 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. Pisanje ovakvih aplikacija nezavisno je od konkretnog browser-a preko kojeg će aplikacija da radi u pozadini i nezavisno je od uređaja na kome će da se izvršava jer je responsive. 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.

66. 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.)

67. 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.

68. Virtuelizacija

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 kreiranjei upravljanje virtuelnim okruženjima ili virtuelnim mašinama yove 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.

69. Kontejnerizacija i docker

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.

70. 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.