# Uvod

Cesto prilikom ovih system design intervjua ume da se desi I da nema tacnog odgovora nego se testira kompunikacija I problem solving skills pa vole da ih ubacuju u intervjue.

Bices ocenjen na osnovu toga koliko dobro analiziras nejasne probleme I kako resaves te probleme korak po korak.   
Takodje testira se i kako objasnjavas ideje, razgovaras sa drugima, evaluiras i optimizujes sistem.

The system design questions are open-ended.

Smer intervjua zavisi od osobe koja te intervjuise. Neki zele high level arhitekturu da se pokriju svi apsekti dok neki zele fokus na odredjenu sekciju.

Tipicno, system requirements, constraints & bottlenecks sistema koji “osmisljavas” treba dobro da se razumeju kako bi se oblikovao smer intervjuera i intervjuisanog.

## Chapter 1: Scale from zero to millions of users

Ovde cemo napraviti sistem koji podrzava jednog usera i lagano ga scaleovati na milione usera.

### Single server setup

Ovo je prvi korak.

Da bi zapoceli sa necim jednostavnim, prvomoramo imati jelte jednostavno.   
Slika ispod predstavlja single server setup gde je sve na jednom server; web app, db, cache i slicno.

A diagram of a computer system

AI-generated content may be incorrect.

Sta se desava:

1. Prvo user posalje DNS(Domain Name Server)-u upit za dobijanje IP adrese naseg servera. DNS je paid service kog daje neki 3rd party i ne hostuje se nasim serverima.
2. IP(Internet Protocol) adresa se vraca uredjaju.
3. Jednom kad je IP adresa pridobijena, salje se HTTP(Hyper Text Transfer Protocol) request direktno na tvoj server
4. Web server vraca HTML stranicu/JSON response za renderovanje

Pogledajmo traffic source. To su web application & mobile application.

* Web Application koristi kombinaciju server side jezika da resava “biznis logiku”, skladistenje i client side jezika(HTML & JS) za prezentaciju.
* Mobile application: HTTP protocol je komunikacioni protokol izmedju mobilne aplikacije i web servera. JSON je cesto koristeni API response format.

### Database

Recimo rastu ti korisnici. Jedan server nije dovoljan i sad nam treba vise servera. Jedan za web/mobile traffic, a drugi za bazu. Razdvajanjem web/mobile traffic(web tier) & baze(data tier), serveri sada mogu nezavisno da skaliraju.

A diagram of a computer network

AI-generated content may be incorrect.

### Koju bazu koristiti

Relaciona & nerelaciona.

Relacione znas vec sta su.   
Nerelacione(NoSQL)(CouchDB, Neo4j, Cassandra, HBase, Amazon DynamoDB) se dele u x4 kategorije:

1. Key-value stores
2. Graph stores
3. Column stores
4. Document Stores

Join operacije ovde cesto nisu podrzane.

Razlozi da se odlucis za nerelacione baze:  
- Treba ti super low latency  
- Podaci su ti nestruktuisani ili nemas “relacione” podatke  
- Trebas samo da serializujes i deserializujes podatke(JSON/XAML/YAML etc..)  
- Trebas da skladistis velike(massive) kolicine podataka

### Vertical vs Horizontal scale

Vertical scaling(scale up) znaci dodavanje snage procesorom/ramo memorijom na serveru.

Horizontalno(scale out) znaci dodavanje servera u tvoj pool resursa.

Kad je saobracaj nizak, vertikalno skaliranje je skroz okej i jednostavnost ove opcije je njena glavna prednost. Nazalost, ima ogranicenja:  
- Ima hard limit. Nemoguce je dodati “bezbrojno” CPU-a i RAM-a na single server  
- Vertical scale nema failover i redundanost. Ako jedan server pukne, puca sve.

Horizontalno skaliranje je pozeljnije za vece aplikacije upravo zbog ovih ogranicenja.

Znaci u gore primeru su se korisnici kacili na x1 server. Ako je taj server nedostupan, ne mozes mu pristupiti. Ako se vise korisnika prikaci na server istovremeno i dostignes web serverov load limit, korisnici iskuse generalno sporiji response ili ne mogu da se povezu na server.

Za taj problem, resenje je **LOAD BALANCER.**

### Load balancer

Load balancer podjednako rasporedjuje dolazeci saobracaj izmedju web servera koji su definisani u “load-balanced set”.

A diagram of a network

AI-generated content may be incorrect.

U gore slici je definisano kako load balancer izgleda.

Korisnici se povezuju na public IP load balancera direktno. Na taj nacin, web serverima se vise ne pristupa direktno od strane klijenata. Za bolji security, private IP’s se koriste za komunikaciju izmedju servera.

Private IP je IP adresa kojoj mogu pristupati samo serveri unutar iste mreze, ali ne i van te mreze(e.g. internet).

Posle load balancera i dodavanja drugog servera, sada smo resili problem failovera i poboljsali dostupnost **web tier**-a.

* Ako server 1 ode offline, sav saobracaj usmerice se na server 2. Na taj nacin website nece biti offline. Takodje dodacemo novi zdravi server u server pool da bi se balansiralo opterecenje
* Ako website saobracaj naglo poraste i dva servera nisu dovoljna da izdrze taj saobracaj, load balancer moze ovaj problem lako da sredi dodavanjem novih servera u web server pool kojim ce LB kasnije balansirati.

To je nas **web tier**. Sta je sa **data tier**? Trenutni dizajn ima samo x1 database bez failovera i redundantnosti. Database replication je cesta tehnika za resavanje tih problema.

### Database replication

Database replication se cesto koristi u master/slave odnosu izmedju originala(master) i kopija(slaves) baza.

Master baza cesto podrzava samo write/update/delete operacije. Slave baza dobija kopije podataka od master baze i podrzava samo read operacije. Sve data modifying komande poput insert, delete ili update moraju biti poslate na master bazu.

Vecina aplikacija zahteva mnogo veci odnos u reads naspram writes pa je cesto i broj slave baza u sistemu bezi od broja master baza.

Na slici ispod moze se videti master baza naspram slave baza.

A diagram of a computer server

AI-generated content may be incorrect.

Prednosti database replicationa:

* **Bolje performanse**. read operacije se distribuiraju across slave nodes jer sada mozes vise upita procesirati paralelno.
* **Reliability**(pouzdanost). Ako jedan db server padne, ne moras brinuti o data loss-u jer su podaci replicirani duz sistema.
* **Visoka dostupnost**. Repliciranjem podataka kroz vise ralzicitih lokacija, website ti moze biti dostupan cak iako jedna baza padne.

U predhonom primeri imali smo load balanser koji je pomagao da se preusmeri saobracaj. Kako to ovde?

* Ako je samo jedna slave baza dostupna i ona ode offline, read operacije ce se privremeno preusmeriti ka master bazi. Cim se problem pronadje, nova slave baza ce zameniti staru. U slucaju da imamo vise slave baza, one ce uzeti taj saobracaj.
* Ako master baza ode offline, slave baza ce biti promovisana kao nova master baza. Sve db opecaije se privremeno biti izvrsene na novoj master bazi. Nova slave baza ce zamniti staru za data replication. U production systems, promotion je kompleksniji jer novi master mozda nema sve podatke pa je neophodno uz recovery scripts podici bazu da bude up to date. Postoje multi-master & circular replication metode koje mogu pomoci, ali to je out of scope za ovu knjigu.

A diagram of a computer network

AI-generated content may be incorrect.

Arhitektura nakon dodavanja replikacije u nas sistem.

Ukratno, jasno ti je sta se ovde desava. Sad kad se malo bolje razumemo u web/data tiers, mozemo da predjeno na sledecu tacku, a to je poboljsanje load & response times. Dodavanjem cache layera i shifting static contenta(js/css/video files) u CDN(Content Delivery Network) ce nam tu mnogo pomoci.

### Cache

Cache je privremeni storage area koja cuva rezultate “skupih” upita ili cesto pristupanim podacima u memoriji tako da se requestovi brze opsluzuju. U gore slici, svaki put kad se nova web stranica ucitava, neophodno je odraditi jedan ili vise db poziva. Performanse aplikacije su itekako zahvacene ovim.

### Cache tier

Cache tier je privremeni data store layer mnogo brzi od baze.   
Benefiti:  
- rasterecenija baza  
- bolje performanse sistema  
- mogucnost skaliranja cache tiera nezavisno

A diagram of a computer

AI-generated content may be incorrect.

Nakon dobijanja requesta, web server prvo proverava ako cache ima dostupan resposne. Ako ima, onda se vrati. Ako nema, onda se query upucuje bazi, rezultat se memorise u cache i uzvraca se serveru.

Ta napomenuta **caching strategy** se naziva **read-through cache.** Druge strategije su dostupne u zavisnosti od tipa podataka, velicine i obrasca pristupanja(access patterns).

Vecina cache servera nude API za vecinu programskih jezika pa je i interakcija sa istima poprilicno jednostavna.

Primer za **memcached:**  
A close up of a text

AI-generated content may be incorrect.

### Considerations for using cache

* **Odluci o tome kada zelis da koristis cache**.  
  Valja za scenario u kom imas puno citanja, a malo modifikacija. Cache nije idealan za cuvanje podataka jer je “volatile memory” u pitanju. Ako se cache server restartuje, gubis sve podatke.
* **Expiration policy.** Dobra je praksa imati expiration policy kojom jednom kada cachirani podaci isteknu, ciste se iz cachea
* **Consistency.** Ovo ukljucuje sinhronizaciju data stores-a i cachea. Inconsistency se desava kada data modifying oepracije na data store & cache nisu u single transaction. Skaliranjem preko “vise regiona” je izazovno. Facebook ima “Scaling memcache at facebook” document.
* **Mitigating failures.** Cache server i dalje moze da pukne i da bude potencijalni single point of failure(SPOF). SPOF znaci da je to u pitanju deo sistema koji kada pukne, moze da znaci da ceo sistem puca. Visestruki cache serveri u razlicitim data centrima su recommended. Drugi recommendation je da pogledas koji ti je required memory i uvek imas buffer iznad za odredjeni %. A diagram of a router

  AI-generated content may be incorrect.
* **Eviction policy.** Jednom kad je cache full, zahtev da se dodaju itemi u cache mogu rezultovati da se postojeci itemi obrisu. To se naziva **cache eviction**. **Least recently used(LRU) je najcesci eviction policy**. Postoje **i Least Frequently Used(LFU)** ili **First in First Out(FIFO).**

### CDN - Content Delivery Network

CDN je mreza globalno distribuiranih servera koji se koriste da dostave staticki sadrzaj. CDN serveri uglavnom taj sadrzaj cacheuju(slike, video, js, css, html pages etc.)

**Dynamic content caching** je relativno nov koncept i van opsega knjige. Uglavnom ideja je da se na osnovu request path, query strings, cooking i request headera omogucava caching HTML stranica.

Ovo je kako CDN funkcionise na high level. Kad korisnik pristupa websiteu, cdn server najblizi korisniku ce dostaviti staticki sadrzaj. Intitivno, sto su dalji korisnici od CDN-a, to se sajt sporije ucitava. NPR CDN serveri koji su u San Francisko, a korisnici u LA ce dobiti sadrzaj brze od korisnika u EU.

A diagram of a computer network

AI-generated content may be incorrect.Slika 1-9 koja pokazuje cdn na high level.

A diagram of a cloud computing system

AI-generated content may be incorrect.

Slika 1-10 CDN workflow.

U sustini slicno kao cache. Korisnik posalje zahtev za neku sliku, cdn prihvati zahtev i ukoliko nema sliku u cache-u, onda se prosledjuje zahtev serveru. Rezultat servera(slika) se skladisti u CDN i vraca korisniku. Svaki sledeci korisnik koji posalje zahtev za tu sliku ce je dobiti direktno od CDN-a bez da komunicira sa serverom.

Da se razumemo, taj “server”, moze da bude i online storage poput amazon S3. U pitanju je origin jelte. U trenutku kada “server” vraca sliku CDN-u, on moze opciono postaviti TTL HTTP header koji naznacava koliko dugo ce se slika cacheovati.

### Considerations of using a CDN