(Slajd1) Pregled arhitektura savremenih veb-aplikacija

Sledeća tema je malo inoviran pregled arhitektura savremenih veb-aplikacija i različitih uzoraka na nivou arhitekture koji se često koriste. Rekli smo da je arhitektura veoma važna u softveru. Najopštija definicija je da arhitekturu čine komponente, konektori (kao najvažniji delovi sistema) i protokoli kojima oni međusobno komuniciraju.

(Slajd 2) Tipovi arhitektura veb aplikacija

Kada se to primeni na veb-aplikacije, preskočićemo onaj uvodni deo koji sigurno znate: najosnovniji model čine klijent i server. Server obavlja procesiranje, na klijentu postoji brauzer, a osnovni protokol komunikacije je HTTP. Klijent šalje zahtev, server mu odgovara, a u toj komunikaciji se koriste tehnologije poput HTML-a, CSS-a i JavaScript-a, za koje pretpostavljam da ih već poznajete.

(Slajd 3) Moderna Veb Arhitektura – Gde se izvršava logika?

Što se tiče same strukture savremenih veb-aplikacija, osnovna podela (čak i kod najmanjih aplikacija) je na:

1. **SSR (Server-Side Rendering)** – gde se glavni deo posla obavlja na serveru.
2. **CSR (Client-Side Rendering)** – gde se većina koda koji aplikacija izvršava šalje klijentu i izvršava u JavaScript-u, dok je server samo back-end.

Ako pogledamo detaljnije **Server-Side Rendering**, u prvom koraku server prima zahtev za određenu stranicu. Zahtevi su strukturirani prema URL-u sajta i obično su logički podeljeni na funkcije i podfunkcije (razdvojene kosim crtama). U SSR-u se najveći deo posla oko obrade zahteva odvija na samom serveru u serverskom skript jeziku. Postoji velika lista takvih jezika, a na ovom kursu su se frejmvorci menjali tokom godina. Trenutno su se ustalili Python i Django.

Kod obične SSR aplikacije u okviru Django frejmvorka koristi se **MVT (Model-View-Template)** pattern. Zahtev prolazi kroz ruter koji određuje koji **View** (funkcija ili klasa) treba da ga obradi. View prima parametre iz zahteva i preko Django API-ja (objektno-relaciono modelovanje) pristupa **Modelu**. Model povlači podatke iz baze (npr. postove nekog korisnika). Kada View završi rad sa modelom i dobije podatke, prosleđuje ih **Template-u**. Template je komponenta koja liči na HTML, ali je pisana mešavinom HTML-a i Pythona. On se popunjava podacima iz modela, formatira HTML i takav, gotov HTML šalje brauzeru. U te šablone se može učitati i malo JavaScript-a za "hidriranje" HTML-a (npr. za otvaranje

menija), ali poenta je da svaki novi klik šalje direktan zahtev serveru koji ponovo renderuje celu stranicu.

(Slajd 4) Moderna Veb Arhitektura – Gde se izvršava logika?

Kod **Client-Side Renderinga**, server prima zahtev klijenta, ali mu šalje samo minimalan HTML. Cela dalja obrada se odvija u brauzeru. On preuzima JavaScript datoteke (često kompresovane ili putem *lazy loadinga* kako bi se ubrzalo učitavanje frejmvoraka). JavaScript biblioteka dalje generiše sadržaj stranice, a po potrebi se obraća serveru preko **RESTful veb-servisa**. Ovi servisi se takođe mogu pisati u Djangu koristeći **Django REST framework**.

CSR aplikacije se obično pišu kao **SPA (Single-Page Application)**. Za razliku od klasičnog SSR-a gde svaki klik uzrokuje reload cele strukture sa servera, ovde stranica stalno "čuči" u brauzeru sa svojim instanciranim **DOM-om (Document Object Model)**. U pozadini se preko API-ja, koristeći AJAX ili slične pozive, povlače samo čisti podaci. Dok god aplikacija radi, to je zapravo jedna ista stranica.

(Slajd 5) Moderna Veb Arhitektura – Gde se izvršava logika?

Postoje i hibridna rešenja. Glavna mana modernih CSR aplikacija, iako su po većini parametara superiorne, jeste sporije inicijalno učitavanje. To je "veliki greh" u veb-programiranju, pa se zato uvode snalaženja za ubrzanje. Što se tiče interaktivnosti, CSR aplikacije su brže jer menjaju samo lokalni objektni model u memoriji brauzera, dok SSR često mora da povlači sav kod nove stranice.

SSR je lakše pisati jer je sve u jednom projektu i u jednom jeziku. CSR je kompleksniji jer odvaja back-end i front-end u dva projekta sa različitim alatima (npr. React, Node, Vite za build i automatsko ažuriranje koda u brauzeru).

Kada govorimo o integraciji, SSR je statičniji pa AI lakše može da analizira sadržaj sajta, dok CSR ima bolji interfejs prema API-jevima na serveru. Izbor zavisi od vrste aplikacije:

- **E-commerce** sa dosta statičkog sadržaja (katalozi proizvoda) i malo interaktivnosti je pogodan za SSR.
- **Kompleksnije aplikacije** sa puno korisničke interakcije zahtevaju specifične protokole za asinhronu komunikaciju klijenta i servera, što je u CSR-u detaljno razrađeno.

(Slajd 6) Moderna Veb Arhitektura – Gde se izvršava logika?

Hibridni modeli podrazumevaju da se inicijalni prikaz React komponente pre-renderuje na serveru kako bi se dobio razrađen HTML i JavaScript koji se odmah prikazuje. Ovde su bitne metrike poput **LCP (Largest Contentful Paint)** iz Google Core Web Vitals, koje mere vreme od zahteva do pojavljivanja najvećeg elementa na ekranu. Takođe, bitan je **SEO (Search Engine Optimization)** – da li je pretraživačima olakšano rangiranje sajta.

Cilj pre-renderovanja je da se što pre dobije statički HTML na klijentu. Zatim sledi **hidratacija** – preuzimanje JavaScript-a i aktiviranje *event listener*-a, kada aplikacija zapravo "oživi". Nakon toga se aplikacija ponaša kao SPA i više ne povlači ništa pre-renderovano sa servera.

(Slajd 7) CSR arhitektura i AI

Kada posmatramo razvoj aplikacija kroz prizmu veštačke inteligencije, Client-Side Rendering (CSR) arhitektura – koja je danas moderniji i zastupljeniji pristup – pokazuje se kao znatno pogodnija za integraciju sa AI sistemima.

- **Struktura podataka:** CSR arhitektura primorava inženjere da precizno definišu API-je, pre svega RESTful veb servise. To podrazumeva jasno definisanje niza URL-ova i strukture podataka koje oni vraćaju, što je tipično u JSON formatu. Samim definisanjem ovakvih API-jeva vi duboko strukturirate aplikaciju i jasno određujete njene mogućnosti.
- **AI kao klijent:** Ovako čist i strukturiran API omogućava da vaša aplikacija postane dostupna i mašinama. AI agenti mogu direktno da pristupaju tim API-jevima i pozivaju ih ako je potrebno da obrade podatke ili izvrše neku akciju. Integracija veštačke inteligencije u postojeće sisteme i aplikacije trenutno je ogroman trend u industriji, a potreba za automatizacijom i AI analizom podataka prisutna je u svim većim poslovnim sistemima.
- **Generisanje interfejsa:** Što se tiče generisanja korisničkog interfejsa, React i njegov način dekompozicije komponenti pružaju mnogo čistiju i jasniju strukturu. Zbog toga AI modeli znatno lakše generišu React (CSR) komponente nego tradicionalne serverske template-ove (poput onih u Django-u). Kod serverskih template-ova struktura je kompleksnija i manje pregledna jer su HTML, poslovna logika i serverski kod (npr. Python) pomešani unutar istog fajla.
- **Zaključak:** Ključno je razumeti gde se tačno izvršava koji deo koda kako biste pravilno pozicionirali AI module unutar arhitekture. Na osnovu toga donosite odluku da li će AI, na primer, izvršiti sumiranje teksta na serverskoj strani pre slanja podataka klijentu, ili

će React asinhrono pozvati AI modul sa klijentske strane tek nakon što se stranica učita.

(Slajd 8) Višeslojna Arhitektura – Više od podele koda

U modernim uslovima više nema jednoslojnih arhitektura; sve veb-aplikacije su višeslojne. Osnovni koncept je podela na logičke celine.

Tipična aplikacija prati **MVC (Model-View-Controller)** šablon. Django koristi **MVT (Model-View-Template)**, gde je *View* zapravo kontroler.

1. **Prezentacija:** Interakcija sa korisnikom.
2. **Sloj pristupa podacima:** Čuvanje i ažuriranje podataka u bazi. Django koristi **ORM (Object-Relational Mapper)** koji omogućava programeru da radi sa objektima, dok sistem sam mapira te objekte u SQL bazu.
3. **Poslovna logika:** U klasičnom SSR-u to su Django pogledi, a u Django REST sistemu se tu definiše sama logika aplikacije.

(Slajd 9) Višeslojna Arhitektura – Više od podele koda

Treba razlikovati termine **Layers** (slojevi) i **Tiers** (nivoi).

- **Layers (Slojevi)** su logička podela aplikacije i način organizacije koda (npr. prezentacioni sloj, sloj poslovne logike, sloj pristupa podacima).
- **Tiers (Nivoi)** opisuju gde se taj kod fizički izvršava (na kojim serverima).

Aplikacija može biti **monolitna** (jedna baza koda, jedan tim, jedna baza podataka) čak i ako je **tronivovska** (fizički podeljena na tri servera: React, Django i baza podataka).

(Slajd 10) Gde se uklapaju AI servisi u logičku podelu po slojevima?

Kada uvodimo **AI servise (LLM)**, javljaju se novi izazovi. Odgovori AI modela nisu uvek apsolutno tačni i zahtevaju vreme za generisanje. Zaključak je da AI treba odvojiti u poseban sloj koji će vršiti **validaciju i sanitizaciju**. Kao što su programeri nekada morali da sanitizuju SQL upite da bi sprečili napade, tako se danas mora paziti na podatke koje AI šalje ka bazi ili korisničkom interfejsu. Takođe, arhitektura treba da omogući laku zamenu provajdera AI usluga bez znanja krajnjeg klijenta, poštujući princip **Dependency Inversion**.

(Slajd 11) Višeslojna arhitektura kao "Ograda" za AI

Ovo je nastavak preporuka vezanih za uključivanje AI servisa u aplikaciju. Višeslojna arhitektura može služiti kao svojevrsna ograda za eventualne negativne posledice AI-ja. Treba razmišljati o uključivanju veštačke inteligencije u okviru izabranog okruženja u meri u kojoj je to moguće. Svakako, sloj pristupa podacima mora imati validacije koje AI ne može da zaobiđe i ne sme direktno uticati na njihovu izmenu bez nadzora čoveka.

Potrebno je zadržati kontrolu nad sistemom u kojem sve veći deo koda može biti generisan i automatizovan. Postoji termin **"vibe coding"** – to je situacija gde date upit, dobijete program i koristite ga bez daljeg gledanja ako vam se čini da je dobar. Ljudska lenjost često vodi ka tome, ali se u ozbiljnim projektima ovaj segment mora shvatiti mnogo savesnije.

(Slajd 12) Distribuirane veb-aplikacije

Distribuirane veb-aplikacije su arhitekturni stil veoma prisutan u savremenim uslovima, naročito kod velikih aplikacija i kompanija. Distribuirana arhitektura je vrsta arhitekture gde su delovi sistema fizički odvojeni i komuniciraju putem mrežnih protokola kako bi radili zajedno. Naglasak je na potpunoj nezavisnosti delova sistema koji se nezavisno razvijaju, rade i skaliraju.

Sledeća tabela razgraničava monolitne i distribuirane aplikacije:

Karakteristika	Monolitne aplikacije	Distribuirane aplikacije
Pakovanje	Obično jedan izvršni fajl na jednom mestu.	Nezavisni servisi koji se puštaju u rad odvojeno (moguće su različite verzije servisa).
Komunikacija	Pretežno **sinhrona** (pozivanje procedura). Asinhrona obrada (npr. Django signali) se koristi samo za sporedne stvari (e-mail obaveštenja).	Različiti mrežni protokoli (HTTP, RESTful API, **RPC** - Remote Procedure Call). Često **asinhrona** komunikacija.
Redovi poruka	Retko se koriste u osnovnoj strukturi.	Koriste se **Message Queues** (npr. **RabbitMQ**) po principu **FIFO** (First In, First Out) za siguran prenos poruka.

Skaliranje	Skalira se cela aplikacija odjednom (**vertikalno**). Postoji fizička granica kapaciteta baze podataka.	Skaliraju se samo potrebni servisi u realnom vremenu (**horizontalno**). Pogodno za oblak (*cloud*).
Otpornost na greške	Otkaz može srušiti ceo sistem.	Izolovanost servisa omogućava **graceful degradation** (smanjena funkcionalnost bez potpuna nedostupnosti).
Baza podataka	Zajednička baza za ceo sistem.	Svaki mikroservis treba da ima svoju bazu. Zajednička baza u mikroservisima je **anti-pattern**.

Monolitne aplikacije je lakše razvijati i one su i dalje adekvatno rešenje za manje firme i projekte; to nije odbačena tehnologija, već kategorija za određenu vrstu rešenja.

(Slajd 13) Dodatne kategorije veb arhitektura

1. **PWA (Progressive Web Apps):** Korisnik ih na telefonu instalira kao lokalnu aplikaciju, ali su one zapravo napravljene pomoću veb-frejmvorka. Koriste **Service Workers** (pozadinske radnike) i keširanje kako bi omogućile rad u *offline* režimu (npr. u avionu).
2. **Serverless veb-aplikacije:** Back-end logika se piše kao čiste funkcije (**FaaS - Function as a Service**). Infrastruktura je apstraktovana, a aplikacija se skalira automatski. Provajderi koriste tzv. **slack time** (slobodne resurse servera) da bi izvršavali ove funkcije. Ovo je isplativo za obrade koje bi inače zahtevale zakup celog servera.

Često se pravi kombinacija: mikroservisna aplikacija gde se određene funkcije izvršavaju kao *serverless*.

(Slajd 14) Obrasci za arhitekturu veb aplikacija

Prelazimo na projektne obrasce ili uzorke.

(Slajd 15) Projektni obrasci/uzorci (Design patterns)

Svaki uzorak opisuje problem koji se iznova javlja u našem okruženju, a zatim nudi suštinu rešenja tog problema na način da se ono može koristiti milion puta, a da se nikada ne uradi dva puta identično.

Poenta šablona je da oni nisu "uklesani u kamenu". Dozvoljen je određeni nivo slobodnog tumačenja i prilagođavanja konkretnoj situaciji. Oni su zapravo iskustvo koje vam pomaže da uočite sličnost vašeg problema sa nekim koji je već uspešno rešavan ranije.

(Slajd 16) Izvori informacija o projektnim uzorcima

Kada govorimo o dizajn paternima, najčešće se misli na čuvenu "Gof" (Gang of Four) knjigu (*Design Patterns: Elements of Reusable Object-Oriented Code*). Ona je krajem devedesetih napravila revoluciju u detaljnom projektovanju softvera. Međutim, postoji i drugi autor, Martin Fauler (*Martin Fowler*), koji je napisao ključnu knjigu *Patterns of Enterprise Application Architecture*. On je razmatrao arhitekturu *enterprise* aplikacija koje se koriste u industriji i preduzećima, na šta se fokusiraju i naši kursevi. Toplo preporučujem njegov sajt koji se stalno ažurira novim diskusijama o softverskom inženjerstvu, arhitekturi, pa čak i o ulozi veštačke inteligencije u IT firmama.

(Slajd 17) Obrazac Model-View-Controller

Najstariji i najvažniji obrazac je **Model-View-Controller (MVC)**. On deli aplikaciju na tri celine:

1. **Model:** Odgovoran za upravljanje podacima. Iza njega najčešće stoji relacionu baza, mada može biti i nerelaciona. Kod relacionih baza važna je struktura (entiteti, atributi, relacije) i transakciona obrada. U centralizovanim sistemima transakcije prate **ACID** model (garantuje se konzistentnost ili *rollback* u slučaju greške). Moderne distribuirane baze često koriste **BASE** model i **eventualnu konzistentnost** (*eventual consistency*), gde se podaci usklade nakon određenog vremena.
2. **View (Prikaz/Prezentacija):** Odgovoran za prikazivanje podataka koje model pruža. Kod SSR aplikacija to je obično *template* jezik koji kombinuje HTML i programski kod, dok se kod klijentskih aplikacija koriste JavaScript frejmvorci (React, Angular, Vue).
3. **Controller (Kontroler):** Srž serverske aplikacije. Upravlja modelom i prikazom. Prima zahtev od klijenta, pokreće model za obradu podataka i šalje te podatke prikazu na formatiranje.

(Slajd 18) Dijagram kolaboracije MVC

Kada posmatramo osnovni MVC (Model-View-Controller) arhitektonski šablon, interakcija i protok podataka između komponenti odvijaju se na sledeći način:

- **Prijem zahteva (Controller):** Korisnički zahtev iz veb brauzera (klijenta) stiže do kontrolera, koji služi kao centralna tačka za upravljanje aplikativnom logikom.
- **Rad sa podacima (Model):** Kontroler zatim pristupa modelu kako bi uzeo ili izmenio potrebne podatke iz baze podataka. Model obrađuje taj zahtev i vraća tražene podatke nazad kontroleru.
- **Prosleđivanje i formatiranje (View):** Nakon što dobije podatke od modela, kontroler ih šalje pogledu (View). Zadatak pogleda je da te sirove podatke formatira i pripremi za prikaz korisniku.
- **Odgovor klijentu:** Na kraju, tako formatiran prikaz se šalje nazad klijentu u brauzer – bilo direktno iz pogleda, ili ponovo posredstvom kontrolera koji zaokružuje ceo ciklus obrade zahteva.

U **Django** frejmvorku ovaj obrazac se naziva **MVT (Model-View-Template)**:

- **Model** je isti.
- **View** u Djangu je zapravo **Kontroler**.
- **Template** je ono što je u standardnom MVC-u **View**.

Pored ovih, postoje i komponente poput **Rutera** (mapira URL na odgovarajući View) i **Middleware** komponenata koje mogu modifikovati objekt zahteva (*request*) pre nego što on stigne do kontrolera.

(Slajd 19) Inverzija kontrole (IoC) i princip inverzije zavisnosti (DIP)

Podsetimo se principa **inverzije zavisnosti (DIP)**: moduli višeg nivoa (npr. Django View) ne bi trebalo da zavise direktno od modula nižeg nivoa (npr. OpenAI klijent). Oba treba da zavise od **apstrakcija** (interfejsa).

(Slajd 20) Zašto je DIP "spas" u AI projektovanju?

Tržište veštačke inteligencije razvija se neverovatnom brzinom, zbog čega primena principa inverzije zavisnosti (Dependency Inversion Principle - DIP) postaje ključna za stabilnost i fleksibilnost softverskih sistema.

- **Agilnost modela:** AI modeli i parametri njihovog korišćenja menjaju se bukvalno na nedeljnom nivou. Cene API poziva konstantno variraju, a provajderi se smenjuju – čas se pojavi napredniji model poput GPT-5, čas pojeftini Claude, ili se pojavi opcija da se koristi lokalna Llama 3 unutar sopstvene infrastrukture. Zbog tih čestih promena na tržištu, aplikacija ne sme biti čvrsto vezana za jednog provajdera.
- **Prednosti primene DIP-a:**
    - **Hot-swapping:** Oslanjanjem na apstrakcije (interfejse), omogućava se laka zamena provajdera servisa u hodu (hot-swapping). Možete promeniti kompletan AI model koji se nalazi u pozadini, a da pritom ne morate da menjate nijednu liniju osnovne biznis logike aplikacije.
    - **Testabilnost (Mocking):** Kada zavisite od interfejsa, u procesu testiranja možete jednostavno "podmetnuti" lažne klase (Mock objekte). Možete hardkodovati podatke koji odmah vraćaju fiksni tekst, čime testirate aplikaciju brzo i besplatno, bez stvarnog slanja zahteva OpenAI servisu i trošenja novca na API pozive.
    - **Sigurnost:** Pristup AI modelima zahteva korišćenje API ključeva i drugih osetljivih podataka za identifikaciju korisnika. Primena DIP-a omogućava da se svi ti bezbednosni parametri izoluju i čuvaju na jednom mestu, unutar konkretne klase adaptera, umesto da budu rizično razbacani svuda po izvornom kodu aplikacije.

(Slajd 21) Praktična implementacija – Dependency Injection (DI)

Na DIP se naslanjaju dva ključna arhitekturna obrasca:

1. Dependency Injection (DI)

Tehnika kojom jedan objekat snabdeva drugi objekat njegovim zavisnostima. Umesto da klasa sama instancira klijent (npr. OpenAI Client) sa svim API ključevima, zavisnost se ubacuje spolja, najčešće kroz konstruktor ili poseban metod.

2. Inversion of Control (IoC)

Umesto da vaša klasa kontroliše zavisnost, kontrola se predaje spoljašnjem sistemu koji se zove **IoC kontejner**. On upravlja instanciranjem klasa i "ubrizgavanjem" potrebnih objekata na osnovu konfiguracije.

(Slajd 22) DI Primer: Injektovanje klijenta za eksterni API

Pretpostavimo da želimo da šaljemo obaveštenja. Da bismo ispoštovali ove principe:

(Slajd 23) DI Primer: Injektovanje klijenta za eksterni API

1. **Servisna klasa:** Kreiramo NotificationClient u fajlu services.py. On sadrži logiku za slanje (API ključ, URL, slanje JSON poruke).

(Slajd 24) DI Primer: Injektovanje klijenta za eksterni API

2. **Middleware:** Umesto da svaki View sam pravi klijenta, kreiramo NotificationServiceMiddleware. Njegov posao je da pročita parametre iz settings.py, instancira klijenta i "okači" ga kao atribut na request objekat (npr. request.notifications).

(Slajd 25) DI Primer: Injektovanje klijenta za eksterni API

3. **Konfiguracija:** U settings.py registrujemo našu middleware klasu kako bi je Django automatski pozivao pri svakom HTTP zahtevu.

(Slajd 26) DI Primer: Injektovanje klijenta za eksterni API

4. **View:** Na kraju, View (koji može biti pisan kao klasa ili prosta funkcija) samo poziva request.notifications.send(user_id, message). On ne zna ništa o API ključevima ili URL-ovima.

(Slajdovi 31,32) Da li je Django Middleware IoC kontejner?

Django Middleware možemo nazvati **"IoC kontejnerom za siromašne"**. Ima neke odlike (inverzija kontrole, injektovanje podataka, centralizovana konfiguracija), ali mu nedostaju:

- **Upravljanje životnim ciklusom:** Pravi IoC kontejner precizno kontroliše da li je objekat *Singleton* (jedna instanca za sve) ili se pravi nova instanca svaki put (*Factory*).
- **Granularnost:** Middleware je vezan isključivo za HTTP ciklus. Ne može lako injektovati zavisnosti u obične Python klase ili pozadinske procese (poput **Celery**-ja).

Za pravi IoC u Python-u može se koristiti biblioteka poput **"dependency-injector"**, koja je generička i može se integrisati sa Djangom ili koristiti samostalno.

Slajd 33 -- Arhitektura vođena događajima

U ovom arhitekturnom šablonu **događaji** su centralni za arhitekturu, a komponente sistema interaguju tako što proizvode i konzumiraju događaje.

Reagovanje na događaje može biti sinhrono , gde se ceo događaj obradi pre nego što se pređe na obradu sledećeg, ili asinhrono, gde aplikacija može da radi druge stvari dok na primer čeka na ulaz/izlaz.

Slajd 34 – Sinhrono i asinhrono programiranje

Ljudi kada komuniciraju telefonom, to je primer sinhrone komunikacije, a sms ili imejl je primer asinhrone.

U programiranju imamo više taskova koji se izvršavaju nezavisno jedan od drugog, ali postoji dogovoreni mehanizam kako da jedan obavesti drugi da je završio i da mu pošalje rezultat.

Slajd 35 – Konkurentno i paralelno programiranje

Ako koristimo asyncio konstrukcije u python-u onda asinhrone taskove realizujemo konkurentnim pristupom, gde jedan proces naizmenično izvršava deo po deo taska u kooperativnom multitaskingu, dakle task dobrovoljno prepušta kontrolu drugom tasku. Ovime se može postići preklapanje obrade na procesoru sa sporim ulazom-izlazom.

Pravi paralelizam postiže se korišćenjem multiprocessing odnosno Celery biblioteke za distrubirane aplikacije koja koristi red čekanja za komunikaciju.

Slajd 36 – Event-Driven Architecture (EDA) i Message Brokers

Umesto da funkcije pozivaju jedna drugu direktno, mi komuniciramo preko događaja. Znači, ne kažemo: ‘pozovi ovu funkciju’, nego kažemo: ‘desio se događaj’.

Tipičan primer: korisnik objavi post.
U sinhronom pristupu sekvencijalno bi se radilo slanje notifikacija, AI obrada, indeksiranje...
Umesto toga, šalje se dogadjaj *PostCreated* .

Ko učestvuje u tome?

- **Proizvođač** – recimo Django view, koji generiše događaj

- **Posrednik za razmenu poruka** – Redis ili RabbitMQ, koji čuva poruke
- **Potrošač** – worker, recimo Celery, koji obrađuje događaj

Prednost ovoga pristupa je što sistem postaje skalabilan. Ako imamo puno zahteva, samo dodamo još workera.

Slajd 37 – EDA u praksi – Statusna arhitektura

E sad dolazi praktičan problem.

Ako AI obrada traje 10 sekundi, a korisnik očekuje odgovor odmah — šta radimo?

Ne možemo samo da blokiramo klijenta i njegov zahtev.

Zato koristimo sledeći pristup:

- server odmah vraća odgovor, recimo HTTP 202 Accepted
- obrada ide u pozadini, task ide u Redis red čekanja, Celery worker ga preuzima
- rezultat dolazi kasnije

Kako korisnik dobija rezultat?

Tri opcije:

1. **Polling** – klijent pita server: 'Je l' gotovo?' svakih par sekundi
2. **WebSockets** – server sam gura rezultat kad je spreman
3. **Webhooks** – eksterni servis obaveštava naš server

Ključna stvar je da korisnik ne sme da trpi zbog spore obrade zahteva.

Slajd 38 – WebSockets vs WebHooks

Postoji razlika između mehanizama WebSockets i WebHooks:

- **WebSockets** imaju stalnu konekciju i komunikacija je dvosmerna
- Kod **WebHooks** se dobija povratni HTTP poziv sa servera kada se nešto desi

Na primer, ako pravite Google Docs, tj. Treba vam interakcija u realnom vremenu onda ćete koristiti WebSockets. Ako samo čekate neku potvrdu o uplati, onda WebHooks.

Slajd 39 – Dizajn vođen domenom (DDD)

Ovde uvodimo jedan drugačiji pristup projektovanju sistema – *Domain-Driven Design*, odnosno dizajn vođen domenom.

Slajd 40 – Dizajn vođen domenom (DDD)

Osnovna ideja DDD-a je da se fokus pomeri sa tehničkih detalja na sam problem koji rešavamo, odnosno na domen poslovanja. Drugim rečima, umesto da odmah razmišljamo o bazama, frejmvorcima i slojevima, prvo modelujemo realni svet – pojmove, pravila i procese.

DDD posebno dolazi do izražaja kod kompleksnih sistema, gde poslovna logika nije trivijalna i gde je važno da svi u timu – i programeri i domen eksperti – imaju zajedničko razumevanje sistema.

Slajd 41,42 – Ključni koncepti DDD-a

DDD uvodi nekoliko ključnih pojmova.

Prvi je ***ubiquitous language***, odnosno zajednički jezik. To znači da svi koriste iste termine – i u kodu i u komunikaciji.

Zatim imamo ***entitete*** i ***vrednosne objekte***, ključni gradivni blokovi domenskog modela. Oni modeluju realne poslovne koncepte. Entiteti imaju identitet koji se ne menja kroz vreme, dok su value objekti definisani svojim vrednostima.

Tu su i ***agregati***, logička grupa povezanih entiteta i value objekata. Agregati su jedinica za promenu podataka uz očuvanje poslovnih pravila.

Factory-ji koji kreiraju objekte i agregate.

Repozitorijumi služe za pristup agregatima, to jest, aplikacija se ponaša kao da su svi objekti u memoriji, a repozitorijum obezbeđuje transparentno čuvanje, pronalaženje i brisanje agregata u bazi.

Ilustrativni primer bi bio da imate fabriku (Factory) koja sastavlja delove u ispravno Vozilo (Aggregate). Kada kupite auto, vi ga vozite i održavate, odnosno menjate stanje preko agregata. Kada ga ne koristite, on stoji u Garaži (Repository) odakle ga možete ponovo uzeti.

Servisi orkestriraju interakcije (poslovnu logiku) između ovih objekata.

Cilj svih ovih koncepata je da domen jasno preslikamo u strukturu koda.

Slajd 43 – Slojevi u DDD arhitekturi

Kada primenimo DDD u web aplikacijama, tipično ga kombinujemo sa poznatim arhitekturama, kao što su slojevita arhitektura ili MVC.

Imamo domenski sloj, koji je najvažniji jer se tu nalazi poslovna logika.

Iznad njega je aplikacioni sloj, koji orkestrira operacije, ali ne sadrži poslovna pravila.

Ispod imamo infrastrukturni sloj – baze, mrežu, framework-e.

Poenta je da domen bude izolovan i nezavisan od tehničkih detalja.

Ovakva separacija pomaže održivosti i testabilnosti sistema.

Ovo je posebno važno kod velikih aplikacija koje se razvijaju duže vreme.

Slajd 44 – Primer: aplikacija za naručivanje proizvoda

U ovom primeru Django aplikacije za naručivanje proizvoda po DDD metodologiji razradićemo serversku stranu, koja pruža API na koji može da se poveže bilo kakav klijent.

Za razliku od “proste” Django aplikacije koja bi imala samo view-e i model-e koji su izvedeni iz Django ORMa, ovde imamo odvojene domenske, aplikativne, infrastrukturne i prezentacione slojeve.

Aplikativni sloj je tanak sloj koji koordiniše rad domenskih objekata i infrastrukturnih servisa da bi izvršio slučajeve korišćenja sistema bez uvođenja sopstvene poslovne logike.

U domenskom sloju smešteni su entiteti, vrednosni objekti i osnovni interfejs repozitorijuma. Odgovara za održavanje poslovnih pravila aplikacije.

U infrastrukturnom sloju implementirani su repozitorijumi koji se naslanjaju na Django modele.

Konačno imamo i django poglede koje možemo smestiti u prezentacioni sloj i koji direktno komuniciraju sa klijentom.

Sada ćemo detaljnije videte šta se u kom sloju nalazi.

Slajd 45 – Domenski sloj

Domenski sloj sadrži **entitet narudžba** (Order) koji ima svoj id, stavke i status narudžbe i osnovne poslovne operacije kao što su dodavanje nove stavke i potvrda narudžbe.

Vrednosni objekat Money (valuta i suma, nema poseban id.

Postoji i **interfejs repozitorijuma narudžbi** sa stavkama čuvanja u bazi i pretrage po id-u.

U domenskom sloju treba da se implementiraju **poslovna pravila** koja održavaju konzistenciju objekata, na primer, da narudžba ne može da se potvrdi ako nema bar jednu stavku, ili da u novčanu sumu može da se doda samo ista valuta.

Slajd 46 – Aplikativni sloj

Aplikativni sloj sadrži **servisnu klasu za narudžbe** koja realizuje osnovne slučajeve korišćenja – kreiranje narudžbe, dodavanje stavke u narudžbu i potvrda narudžbe.

Slajd 47 – Infrastrukturni sloj

Infrastrukturni sloj sadrži **Django modele**, to jest, klase koji obezbeđuju objektno-relaciono mapiranje entitetski objekata u bazu, sa specifikacijom tipova objekata u tabelama, ključevima i tako dalje. O detaljima ORM pričaćemo u sledećoj lekciji.

Konkretno Order i OrderItem su u relaciji 1:N i to je realizovano u bazi sa dve tabele, pri čemu druga tabela koja podržava OrderItem sadrži id Ordera kao strani ključ prve tabele.

Slajd 48 – Infrastrukturni sloj

I **implementacija repozitorijuma narudžbi**, koja se oslanja na Django modele, pripada sloju infrastrukture. Implementira čuvanje narudžbi u bazi i dohvatanje narudžbe prema id-u.

Primetiti da find_orders_by_id vraća čist domenski objekat koji nema nikakve zavisnosti prema Django ORMu.

Slajd 49 – Prezentacioni sloj

Prezentacioni sloj čine **Django pogledi** koji treba da private podatke iz klijentovog post zahteva, iniciraju realizaciju slučaja korišćenja i vrate rezultat klijentu formatiran u json formatu.

Slajd 50 – Šta je ovde u skladu sa DDD

U primeru je ilustrovana čista podela, gde svaki sloj komunicira sa sebi susednim. Domenski objekti ne importuju django modele. Repozitorijum je taj koji prevodi izmedju ORMa i domena. Prezentacioni sloj ne sadrži poslovnu logiku.

Neke od tipičnih grešaka pri primeni DDD na django aplikaciju je mešanje ORM modela i domenskih objekata i stavljanje poslovne logike direktno u modele, stavljanje servisnih metoda direktno u okviru pogleda i preskakanje sloja repozitorijuma (što se opet svodi na mešanje ORM modela i domena).

Slajd 51 – Ostali važni pojmovi u DDD

Od ostalih važnih pojmova treba pomenuti da se domen u nekim situacijama ne modeluje kao jedan veliki, univerzalan model za celu firmu, već se deli na više lokalnih modela. **Ograničeni kontekst** je granica unutar koje određeni (lokalni) domenski model ima jasno i nedvosmisleno značenje. U dva različita konteksta isti pojam na primer, Proizvod, može da ima različita značenja: prodaju zanima cena i popust, a špediciju dimenzije i težina.

Tada i univerzalni (ubiquitous) jezik nije univerzalan na nivou cele kompanije, nego samo unutar bounded contexta. Granica konteksta je i granica važenja jezika.

Mapiranjem konteksta definiše se kako konteksti međusobno komuniciraju.

Slajd 52 – Ostali važni pojmovi u DDD (Context Map)

Struktura domena se definiše kroz **Mapu konteksta**, gde se određuje kako ti "lokalni svetovi" međusobno komuniciraju, na primer preko domenskih događaja ili API poziva.

Bounded context može se implementirati jednom django aplikacijom ili mikroservisom na primer.

Kada podaci prelaze iz jednog konteksta u drugi, koristi se ***Anti-Corruption Layer***. To je bukvalno "prevodilac" koji uzima pojam iz jednog jezika i mapira ga u pojam drugog jezika.

Slajdovi 53,54 – Primer mapiranja konteksta

Ovo je primer mape konteksta za domen aplikacije za elektronsku kupovinu.

Orders Context (narudžbine) predstavlja glavni domen, komunicira sa klijentom ali ne zna ništa o plaćanju ili održavanju kataloga

Payments Context je odgovoran za naplatu, novčane transakcije i facture, ali ne zna detalje porudžbine, samo prima "signal" da treba platiti. Tipična komunikacija sa narudžbinama bi bila putem primljenog dogadjaja OrderConfirmed, ili primanjem API poziva "naplati orderId".

Catalog Context drži proizvode i cene, a Orders ga koristi za prikaz proizvoda ili proveru cena.

Slajd 55 – Prednosti i mane DDD-a

DDD ima brojne prednosti.

Pre svega, omogućava bolje razumevanje problema i jasniju komunikaciju u timu.

Takođe, dovodi do koda koji bolje odražava poslovnu logiku i lakše se održava.

Međutim, DDD nije besplatan – uvodi dodatnu kompleksnost i zahteva disciplinu i iskustvo.

Za male i jednostavne aplikacije često je previše težak i nepotreban.

Slajd 56 – Kada (ne)koristiti DDD

DDD ima smisla koristiti kada imamo kompleksan domen, veliki tim i dugoročni razvoj sistema.

Ako je aplikacija mala ili kratkoročna, jednostavniji pristupi su često bolji.

Dakle, kao i kod drugih arhitektonskih odluka, važno je proceniti kontekst i ne primenjivati DDD mehanički.

Slajd 57 – Rezime predavanja

Možemo na kraju da sumiramo šta smo sve u ovoj lekciji obradili:

- Pojmovi **CSR** i **SSR** – objasnili su nam gde se izvršava logika
- Šabloni tipa **MVC i podele na slojeve** – pomažu nam kako da organizujemo kod
- Šabloni **DIP i IoC** pomažu da napravimo fleksibilan sistem
- Arhitekturom pokretanom događajima rešavamo sporost obrade zahteva i skaliranje sistema

Svi ovi koncepti nisu izolovani, već se upotrebljavaju zajedno.

Slajd 58 – Zaključak

I sad dolazimo do možda najvažnije poruke ovog predavanja.

Arhitektura nije samo tehnička stvar — ona postaje način komunikacije sa AI-jem.

Možete posmatrati arhitekturu kao *prompt*.

Ako je sistem dobro dizajniran:

- AI može da radi bezbedno
- možete ga menjati bez lomljenja sistema
- imate kontrolu

Takođe, jako važna stvar — *čovek u kontrolnoj petlji*.

Ne smete dozvoliti da AI direktno utiče na korisnike bez kontrole.
Mora postojati sloj gde čovek verifikuje rezultate.