Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike
Smjer: Matematika i računarstvo

Domagoj Suhić

Arhitektura Softvera

Seminarski rad
Mentor: doc. dr. sc. Alfonzo Baumgartner

Osijek, 2018.

Sadržaj

1 Uvod

U ovom radu bavit ćemo se arhitekturom softvera. Definirat ćemo same pojmove arhitekture softvera i arhitekturalnog uzorka te predstaviti neke popularne arhitekturalne uzorke: slojevitu, mikrojezgrenu i prostorno baziranu arhitekturu. Za svaki od tih uzoraka ćemo iznijeti prednosti i mane te primjere.

2 Definicija

Arhitektura softvera je *high-level* struktura softverskog sustava koja se sastoji od softverskih elemenata, relacija među njima te vanjski vidljivim svojstvima tih elemenata i relacija. Pojam "vanjski vidljivi" označava da ih drugi elementi mogu pretpostaviti o nekom elementu - bilo da je riječ o servisima koje taj element pruža, performansama, korištenju zajedničkih resursa itd. U modernim sustavima se često koriste sučelja radi podjele elementa na javne i privatne dijelove. Arhitekturu zanima samo javni dio te podjele, a ne detalji koji se odnose isključivo na unutarnju implementaciju.

Svaki sustav posjeduje arhitekturu jer se svaki sustav sastoji od elemenata i relacija među njima. Čak i u trivijalnom slučaju u kojem je sustav samo jedan element, on čini arhitekturu.

Ona nam služi kao nacrt za sustav i projekt koji razvija taj sustav. Odabir prikladne arhitekture prije započinjanja bilo kakvog kodiranja je ključan jer je implementiranje promjena u samoj arhitekturi vrlo skupo.

Arhitekturalni uzorak je općenito, ponovno upotrebljivo, rješenje za problem u razvoju arhitekture softvera koji se često pojavljuje. Iako može predočiti kako sustav izgleda, arhitekturalni uzorak nije arhitektura. Mnoštvo različitih arhitektura može implementirati isti uzorak.

3 Arhitekturalni uzorci

Softverski sustavi bez formalne arhitekture često imaju nedostatke poput usko povezanih komponenti, teško ih je mijenjati i poprilično im je teško odrediti karakteristike bez potpunog poznavanja svake komponente sustava - kakve su performanse sustava, je li arhitektura skalabilna itd.

Korištenje arhitekturalnih uzoraka nam pomaže u definiranju osnovnih karakteristika sustava.

3.1 Slojevita arhitektura

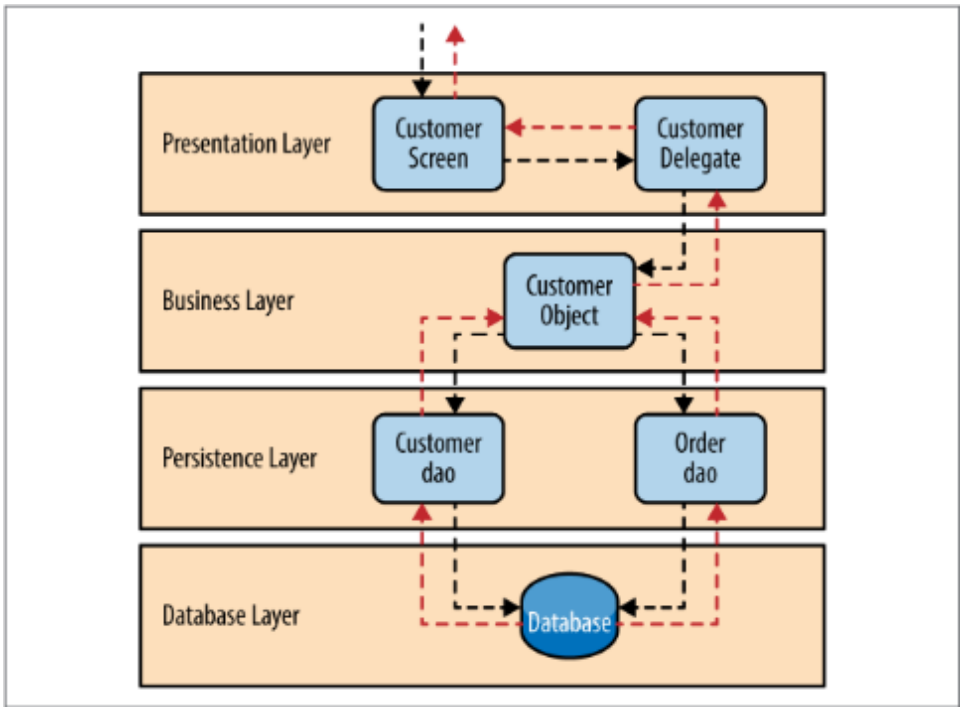
Bez unaprijed jasno definirane arhitekture, većina developera se oslanja na *de facto* standardni arhitekturalni uzorak, slojevitu arhitekturu, u kojem se razdvajanjem modula izvornog koda u pakete implicitno stvaraju slojevi. Nažalost, ovo u praksi često rezultira skupom modula izvornog koda u kojem nisu jasno definirane uloge, dužnosti i odnosi među modulima. Takav arhitekturalni uzorak se naziva *velika vreća blata* anti-uzorak.

Kod pravilnog korištenja uzorka slojevite arhitekture, komponente su organizirane u horizontalne slojeve od kojih svaki sloj ima svoju ulogu u aplikaciji. Među komponentama vlada razdioba brige. To znači da se svaka komponenta brine samo o logici koja se odnosi na taj sloj. Pogledajmo primjer:

Pretpostavimo da razvijamo web aplikaciju koja sadrži prezentacijski sloj, sloj poslovne logike te sloj perzistencije. U toj aplikaciji trebamo spremati nekakve unose koje će slati korisnik. Svakom od tih unosa trebamo dodijeliti jedinstveni identifikacijski ključ. Jednostavnosti radi, pretpostavimo da će taj ključ biti prirodan broj i svakom novom unosu inkrementalno dodajemo jedinstveni ključ. Kada korisnik pošalje podatke koje trebamo spremiti, u kojem sloju bi se ključ trebao dodijeliti podatcima? Kako je dodijeljivanje ključa nešto što se tiče poslovne logike, trebali bismo to odraditi u sloju poslovne logike, a ne u npr. prezentacijskom sloju.

Ovakva podjela nam omogućuje jednostavniji razvoj, testiranje i održavanje aplikacija. Kada bismo se u našoj hipotetskoj aplikaciji u kasnijoj verziji odlučili promijeniti način dodijeljivanja ključa (npr. dodijeljivanjem nasumično generiranih brojeva dok ne dobijemo ključ koji već nismo izgenerirali), bilo bi dovoljno promijeniti samo sloj poslovne logike.

Kod ovog uzorka u principu vrijedi pravilo zatvorenosti. To znači da zahtjev mora prolaziti kroz jedan po jedan sloj - od najvišeg do najnižeg. To je još jedan od mehanizama izbjegavanja usko povezanih komponenti. Mana ovog pravila je to što za neke slojeve postoje razlozi zašto bi mogli biti otvoreni, tj. zašto bi bilo u redu da se taj sloj može preskočiti. Primjer bi bio nekakav sloj zajedničkih usluga, poput loggera, za kojeg bi imalo smisla držati ga neposredno ispod sloja poslovne logike. Međutim, to bi značilo da bismo pri svakom pristupanju sloju perzistencije iz sloja poslovne logike nužno morali proći kroz taj sloj, u većini slučajeva bez da išta radimo u njemu. Njegovim otvaranjem dobivamo mogućnost proći kroz njega ako nam je to potrebno ili ga jednostavno zaobići i otići direktno u sloj perzistencije.



Slika 1: Primjer slojevite arhitekture

Sam broj slojeva te njihove uloge ovise od aplikacije do aplikacije. Najčešće korišteni slojevi su: prezentacijski sloj, sloj poslovne logike, sloj perzistencije te sloj baze podataka. Ovaj arhitekturalni uzorak se još naziva i n-layer, gdje je n broj slojeva. Ako su slojevi fizički raspodijeljeni onda pričamo o

n-tier arhitekturi.

Jedno od stvari na koju trebamo pripaziti kod ovog uzorka jest tzv. *ponor* anti-uzorak. To znači da imamo velik broj zahtjeva koji prolaze kroz nekoliko slojeva bez da se skoro ikakva logika odvija na tim slojevima. Pogledajmo primjer zahtjeva koji bi ispunjavao taj anti-uzorak:

Dobijemo zahtjev na prezentacijskom sloju dohvaćanja podataka o korisniku. Proslijedimo zahtjev u sloj poslovne logike koji onda samo proslijedi zahtjev u sloj perzistencije u kojem se napravi jednostavan SQL poziv na sloj baze podataka za dohaćanje podataka. Ti podatci se jednostavno vrate nazad do prezentacijskog sloja bez ikakvog transformiranja ili dodatne logike. Svaka aplikacija će imati određen broj zahtjeva koji ispunjavaju taj anti-uzorak. Ključno je minimizirati postotak takvih zahtjeva.

Još jedan potencijalan problem kod ovog uzorka je to što prirodno vodi do razvoja monolitske aplikacije, čak i ako razdvojimo prezentacijske i poslovne slojeve u posebno *deploy*-abilne cjeline. To može potencijalno dovesti do problema s robustnošću, performansama i skalabilnosti.

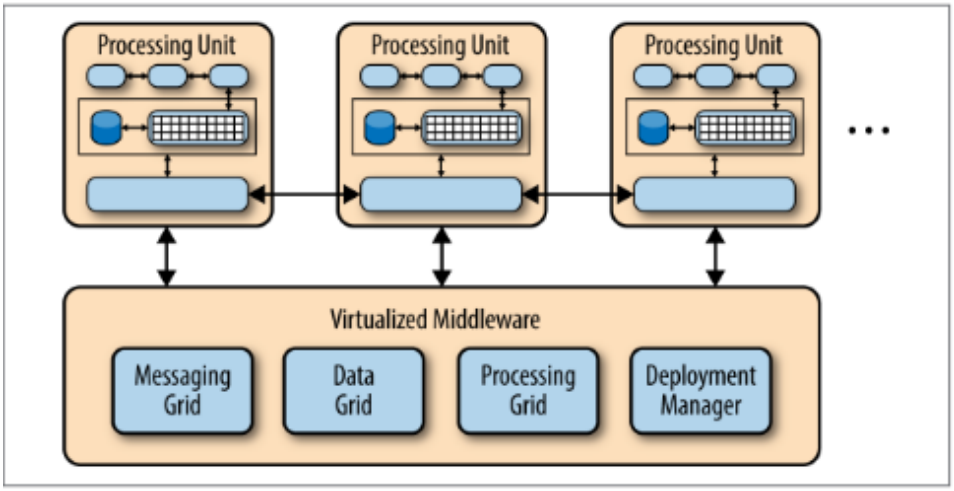
3.2 Prostorno bazirana arhitektura

U tipičnim web aplikacijama tok zahtjeva je sličan. Zahtjev prvo dođe na web server, zatim na server aplikacije te naposlijetku na server baze podataka. Kako se broj korisnika povećava, počinju se pojavljivati *uska grla* - prvo na web serveru. Uobičajen i relativno jeftin i lagan odgovor na to je skaliranje web servera. Međutim, ako je broj korisnika jako velik, skaliranje samo prebacuje usko grlo na server aplikacije. Njegovo skaliranje je pak znatno teže i skuplje, a u konačnici može dovesti do istog problema, odnosno pomicanja uskog grla na server baze podataka. Kada imamo aplikaciju s visokim brojem istovremenih korisnika, čak i uz cachiranje i skaliranje baze podataka, najčešće nam je upravo ona ograničavajući faktor u broju zahtjeva koje možemo istovremeno obrađivati.

Uzorak prostorno bazirane arhitekture je osmišljen baš s namjerom rješavanja problema skalabilnosti i istovremenog obrađivanja zahtjeva. Također je koristan kod aplikacija s vrlo nepredvidivim skokovima u broju istovremenih korisnika. Rješavanje tih problema je često puno bolje odraditi na arhitekturalnoj razini umjesto da pokušamo skalirati bazu podataka u nekoj arhitekturi koja nije prikladna za skaliranje.

Uzorak se bazira na korištenju distribuirane zajedničke memorije. Umjesto korištenja središnje baze podataka, koriste se *in-memory* mreže podataka koje se repliciraju između svih aktivnih obrađivačkih jedinica. Na taj način ostvarujemo skalabilnost. Same obrađivačke jedinice pak možemo paliti ili gasiti ovisno o količini korisnika. Na taj način rješavamo problem nepredvidivih skokova u broju korisnika.

Obrađivačke jedinice se sastoje od komponenti aplikacije, bilo da su to komponente bazirane na web-u ili backend poslovna logika. Manje web aplikacije često imaju samo jednu obrađivačku jedinicu dok veće aplikacije dijele funkcionalnost u više obrađivačkih jedinica. Uz module aplikacije i već spomenutu mrežu podataka, obrađivačke jedinice sadrže i *engine* za replikaciju pomoću kojeg se promjene u jednoj jedinici prenose na druge aktive jedinice. Također mogu sadržavati i perzistentno spremište podataka u slučaju da dođe do pada aplikacije.



Slika 2: Primjer prostorno bazirane arhitekture

Osim obrađivačkih jedinica, ovaj uzorak sadrži još jednu bitnu komponentu, a to je *virtualni middleware*. On je u suštini upravljač arhitekture i brine se za zahtjeve, replikaciju podataka, distribuirano obrađivanje podataka i uključivanje/isključivanje obrađivačkih jedinica. Sastoji se od četiri arhitekturalne komponente: Mreže za poruke, podatkovne mreže, obrađivačke

mreže i upravljača *deployment*-a.

3.2.1 Mreža za poruke

Mreža za poruke upravlja ulaznim zahtjevima. Kada zahtjev dođe do virtualnog middleware-a, ova mreža odredi koje su aktivne obrađivačke jedinice sposobne obraditi zahtjev te proslijedi zahtjev jednoj od njih. Sama metoda odabira jedinice kojoj će se zahtjev proslijediti ovisi o implementaciji.

3.2.2 Podatkovna mreža

Podatkovna mreža je vjerojatno najvažnija komponenta u ovom uzorku. Ona u međudjelovanju s replikacijskim *engine*-om u obrađivačkim jedinicama upravlja repliciranjem podataka među obrađivačkim jedinicama kada se dogode promjene. Iznimno je važno da se ovaj proces odvija u vrlo kratkom vremenskom roku jer mreža za podatke može proslijediti zahtjev bilo kojoj aktivnoj obrađivačkoj jedinici. U pravilu se replikacija događa paralelno, a vrijeme potrebno za sinkronizaciju se ponekad mjeri i u mikrosekundama.

3.2.3 Obrađivačka mreža

Obrađivačka mreža je komponenta koja se brine o usklađivanju rada obrađivačkih jedinica. Ona se koristi kada imamo aplikaciju u kojoj imamo više obrađivačkih jedinica te se svaka od njih bavi dijelom aplikacije. Kada dobijemo zahtjev za koji će nam biti potrebno pokrenuti više jedinica, onda ova komponenta usklađuje rad među njima.

3.2.4 Upravljač *deployment*-a

Ova komponenta konstanto prati vrijeme potrebno za obrađivanje zahtjeva te količinu korisnika. U ovisnostu o tim vrijednostima, dinamički uključuje ili isključuje obrađivačke jedinice.

3.2.5 Mane

Iako je ovaj uzorak dobar izbor za manje web aplikacije s varijabilnim brojem istovremenih korisnika, nije potpuno prikladan za tradicionalne aplikacije s velikim relacijskim bazama podataka. Bez obzira što uzorak ne

zahtjeva središnje spremište podataka, ono često bude uključeno radi inicijalnog učitavanja podataka u mreže i perzistencije podataka tijekom paralelnih repliciranja. Također, ovaj uzorak je poprilično složen i njegova implementacija je skupa.

3.3 Mikrojezgrena arhitektura

Uzorak mikrojezgrene arhitekture je prirodan uzorak za implementiranje aplikacija baziranih na proizvodu. To su aplikacije koje su upakirane za preuzimanje u verzijama kao tipičan *third party* proizvod. Ovaj uzorak omogućuje lako dodavanje dodatnih *feature*-a u proizvod. Iz tog ralozga je prikladana i za interne aplikacije tvrtki koje postoje u različitim verzijama i imaju modularan izbor *feature*-a. Također, prikladan je i za razvoj operativnih sustava, a tako je i dobio ime.

Ovaj uzorak se sastoji od dvije arhitekturalne komponente: središnjeg sustava i uključivih modula. Logika aplikacije je podijeljena između osnove, koju čini središnji sustav, te nezavisnih uključivih modula. Na taj način je osigurana proširivost i fleksibilnost.

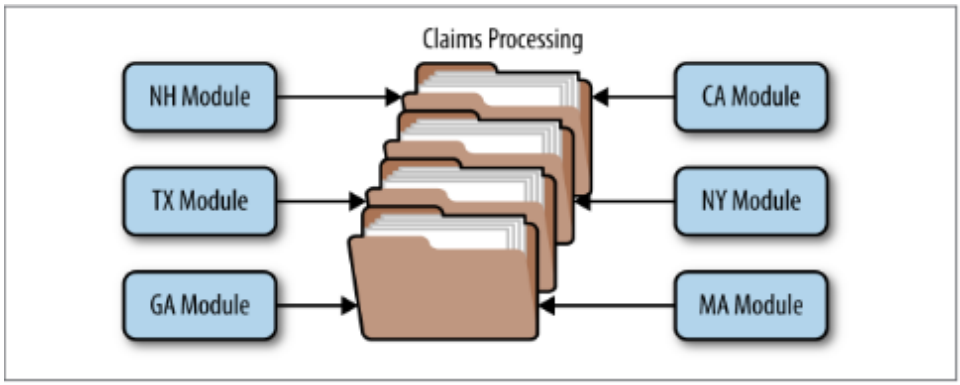
Središnji sustav u principu sadrži samo onu minimalnu funkcionalnost koja je potrebna za rad sustava. U poslovnim aplikacijama se središnji sustav definira tako da sadrži općenitu poslovnu logiku, bez pravila za specijalne slučajeve.

Uključivi moduli su nezavisne komponente koje sadrže *feature*-e čija je svrha poboljšanje ili proširivanje rada središnjeg sustava. Iako su moduli u principu nezavisni jedni o drugima, moguće je dizajnirati uključive module koji zahtjevaju druge module kako bi funkcionirali. U svakom slučaju, komunikaciju među modulima treba svesti na minimum kako se ne bi pojavili problemi sa zavisnosti među modulima.

Ako želimo koristiti ovaj uzorak, potrebno je implementirati nekakav način da središnji sustav bude svjestan koji su moduli dostupni te kako doći do njih. Najčešći način rješavanja tog problema je implementiranje *registry*-ja za uključive module. On bi sadržavao informacije o svakom modulu. Uključive module možemo povezati sa središnjim sustavom na više načina. Možemo koristiti *messaging*, web servise itd.

Primjeri ovog uzorka su različiti IDE ili internet pretraživači. Kada se instaliraju imaju osnovnu funkcionalnost, a instaliranjem različitih modula dobivamo kustomizirani proizvod. Kao primjer kod velikih poslovnih aplikacija možemo uzeti npr. aplikaciju za obrađivanje zahtjeva isplate osiguranja u SAD-u.

Svaka savezna država ima svoja pravila i regulacije, što stvara ogroman skup uvjeta koje treba provjeriti pri obradi svakog zahtjeva. Ako pak takvu aplikaciju implementiramo koristeći ovaj arhitekturalni uzorak, olakšat ćemo si posao. U središnji sustav stavimo osnovnu logiku koja je osiguravajućem društvu potrebna za obrađivanje zahtjeva, bez pravila za specijalne slučajeve. Svaki uključivi modul sadrži specifična pravila za jednu saveznu državu. Sada pravila za jednu saveznu državu možemo mijenjati, brisati i dodavati s vrlo malo ili nimalo utjecaja na središnji sustav ili ostale uključive module.



Slika 3: Primjer mikrojezgrene arhitekture

Vjerojatno najveća prednost ovog uzorka je to što ga možemo umetnuti u neki drugi uzorak. Ne moramo cijelu aplikaciju implementirati koristeći ovaj uzorak, već možemo samo jedan dio koji je podložan promjenama.

4 Zaključak

U ovom radu smo predstavili pojam arhitekture softvera i neke popularne uzorke arhitekture softvera. Odabir pravog uzorka, tj. arhitekture, je iznimno važna stvar. Odabirom neprikladne arhitekture si otežavamo ostvarivanje zahtjeva iz specifikacije softverskog projekta kojeg implementiramo.

5 Literatura

Literatura

[1] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley, 2003.

[2] M. Richards, *Software Architecture Patterns*, O'Reilly, 2015.