Računarski fakultet

Tema

Skaliranje sistema za rezervaciju: Analiza različitih arhitekturalnih pristupa i strategija za skaliranje servisa

Kurs: Praktikum in računarstva u oblaku

Mentor:
prof. Mirjana Radivojević

Student:
Vanja Kovinić

Beograd, 2025.

Sadržaj

1 Uvod

1.1 Problem i motivacija

Upravljanje resursima u modernim poslovnim okruženjima predstavlja ključni izazov za efikasno funkcionisanje organizacija. Rezervacija sala za sastanke, kao jedan od najčešćih operativnih procesa, često je opterećena manuelnim postupcima koji dovode do konflikata u rasporedu, duplih rezervacija i neoptimalnog korišćenja prostora.

U konkretnom slučaju koji je analiziran u ovom radu, vlasnik poslovnog centra koji iznajmljuje sale za sastanke suočavao se sa potpunim odsustvom digitalnog sistema za rezervacije. Ceo proces rezervacije bio je centralizovan kroz jednu sekretaricu koja je putem email komunikacije primala zahteve, proveravala dostupnost i potvrđivala rezervacije. Ovakav pristup stvorio je više kritičnih problema:

- **Nedovoljna preglednost**: Klijenti nisu imali mogućnost da vide raspoloživost sala u realnom vremenu, što je otežavalo planiranje.
- **Ograničena fleksibilnost**: Promene u rasporedu bile su teške za implementaciju, što je dovodilo do dodatnih konflikata.
- **Neproduktivno trošenje vremena**: Značajan deo radnog vremena sekretarice bio je posvećen repetitivnim zadacima umesto obavljanju kompleksnih i strateških zadataka.
- **Neprofesionalan imidž**: Manuelni proces ostavljao je utisak zastarele i neorganizovane kompanije kod klijenata.
- **Nedostatak analitike**: Bez digitalnog sistema, bilo je nemoguće pratiti trendove korišćenja sala, što je otežavalo donošenje poslovnih odluka.
- **Skalabilnost problema**: Sa rastom broja klijenata, sistem je postajao sve manje održiv.

1.2 Tehnologije i pristup

Implementacija sistema zasniva se na Java programskom jeziku i Spring Boot okruženju, koje čini osnovni sloj aplikacione logike. Uz savremene pristupe kontejnerizacije i principe razvoja cloud-native aplikacija, postignuta je visoka prenosivost i konzistentnost sistema u različitim okruženjima. Docker je korišćen za pakovanje i izolaciju aplikacije, dok je GitHub Actions iskorišćen za potpunu automatizaciju procesa kontinuirane integracije i isporuke (CI/CD).

Aplikacija je postavljena na Netcup cloud infrastrukturu, čime je obezbeđeno ekonomično i skalabilno rešenje za produkcijski hosting.

Ključne tehnologije korišćene u projektu su:

- **Razvojna platforma:** Java i Spring Boot, sa implementacijom RESTful API-ja
- **Kontejnerizacija:** Docker, za pakovanje aplikacije i izolaciju od okruženja
- **Orkestracija:** Docker Compose, za upravljanje više kontejnera u okviru jednog sistema
- **Automatizacija (CI/CD):** GitHub Actions, za automatsko testiranje i postavljanje aplikacije na server
- **Baza podataka:** Relaciona baza podataka (MySQL) pokrenuta u posebnom kontejneru
- **Cloud infrastruktura:** Netcup VPS, korišćen za postavljanje i pokretanje sistema u produkciji

Poseban akcenat stavljen je na automatizaciju procesa postavljanja aplikacije, što omogućava brzo isporučivanje novih funkcionalnosti kroz jednostavno slanje izmena u repozitorijum (git push). Ovakav pristup značajno smanjuje vreme potrebno za implementaciju i testiranje izmena. Izvorni kod i konfiguracija dostupni su na GitHub platformi [1].

[1] `https://github.com/Kovelja009/conf_room`

2 Arhitekturni obrasci: Monolit vs Mikroservisi

2.1 Teoretske osnove

Monolitna arhitektura predstavlja tradicionalni pristup razvoju softverskih sistema, pri kojem se celokupna funkcionalnost implementira u okviru jedne celovite i zajednički postavljene aplikacije. U ovom modelu, sve komponente sistema – upravljanje korisnicima, poslovna logika, pristup bazi podataka i prikaz podataka (interfejs) – integrisane su unutar jednog izvršnog procesa. Komunikacija između delova sistema odvija se putem direktnih poziva funkcija unutar iste aplikacije, što može doprineti boljim performansama, ali istovremeno dovodi do jake povezanosti (tight coupling) između modula i smanjenja fleksibilnosti u razvoju i održavanju.

S druge strane, **mikroservisna arhitektura** predstavlja savremeniji pristup koji uvodi koncept distribuiranih sistema. Aplikacija se u ovom modelu sastoji od niza nezavisnih servisa, od kojih je svaki odgovoran za tačno određenu poslovnu funkcionalnost. Svaki mikroservis poseduje sopstvenu bazu podataka i može se razvijati, testirati i postavljati potpuno nezavisno od ostalih servisa. Ovakav pristup omogućava veću fleksibilnost i skalabilnost, ali istovremeno uvodi dodatnu složenost, naročito u oblastima kao što su otkrivanje servisa (service discovery), međuservisna komunikacija i upravljanje transakcijama koje obuhvataju više servisa.

U kontekstu razvoja za cloud okruženje, oba arhitekturna pristupa mogu se uspešno implementirati uz korišćenje kontejnerskih tehnologija. Docker omogućava konzistentno postavljanje aplikacija bez obzira na konkretno okruženje u kojem se izvršavaju, dok platforme za orkestraciju poput Kubernetesa nude napredne mogućnosti za upravljanje servisima, automatsko skaliranje i raspodelu opterećenja. Konačan izbor arhitekture zavisi od specifičnih zahteva aplikacije, veličine i organizacije razvojnog tima, kao i infrastrukturnih i operativnih ograničenja.

2.2 Komparativna analiza

Poređenje monolitne i mikroservisne arhitekture prikazano je u Tabeli 1. Ova tabela sumira ključne aspekte oba pristupa, uključujući skalabilnost, izolaciju grešaka, kompleksnost razvoja, performanse, složenost CI/CD procesa i tehnologije koje se koriste.

Aspekt	**Monolit**	**Mikroservisi**	**Kontekst primene**
Skalabilnost	Vertikalna	Horizontalna	Monolit za mali broj korisnika, mikroservisi za veći broj i opterećenje
Izolacija grešaka	Niska	Visoka	Manji sistemi mogu tolerisati jednu tačku otkaza
Kompleksnost razvoja	Niska	Visoka	Monolit za male timove, mikroservisi za kompleksne sisteme
Performanse	Brže (lokalni pozivi)	Sporije (mreža)	Direktan metod vs HTTP/REST
Deployment	Jednostavan	Složen	CI/CD zahteva više skripti za mikroservise
Komunikacija	Lokalni pozivi	Međuservisna komunikacija	Direktan pristup vs API pozivi
CI/CD složenost	Jedan tok	Više tokova	Veći broj servisa, nezavisna postavljanja
Tehnologije	Jedinstven stek	Potencijalno više tehnologija	Manji timovi preferiraju doslednost
Monitoring	Centralizovan	Distribuiran	Lakše praćenje u monolitu

Tabela 1: Uporedna analiza monolitne i mikroservisne arhitekture

2.3 Faktori za donošenje odluke

Izbor arhitekture samog servisa zavisi od više povezanih faktora koje je potrebno pažljivo analizirati u konkretnom poslovnom i tehničkom kontekstu:

Struktura tima i organizacioni faktori

Conway-ev zakon sugeriše da arhitektura sistema odražava strukturu komunikacije unutar organizacije koja ga razvija. U slučaju malih timova (npr. dva programera), što je karakteristično za analizirani projekat, **monolitna arhitektura** se često pokazuje kao optimalan izbor jer omogućava:

- jedinstveno razvojno okruženje bez dodatnog koordinacionog opterećenja između *frontend* i *backend* programera,
- pojednostavljeno orkestriranje koda - direktan pristup zajedničkom kodu,
- brže donošenje odluka bez zavisnosti između timova i dileme oko vlasništva nad servisima.

Količina saobraćaja i zahtevi u pogledu performansi

Očekivano opterećenje predstavlja ključni faktor u izboru arhitekture:

- **Nizak nivo saobraćaja** (desetine istovremenih korisnika): performanse monolita su u potpunosti dovoljne, dok bi mrežna latencija karakteristična za mikroservise predstavljala nepotreban trošak.
- **Srednji nivo saobraćaja** (stotine korisnika): prelazna zona u kojoj su oba pristupa moguća i zavise od dodatnih faktora.
- **Visok nivo saobraćaja** (hiljade i više korisnika): mikroservisna arhitektura omogućava horizontalno skaliranje i bolje upravljanje opterećenjem, što je ključno za očuvanje performansi i dostupnosti sistema.

Brzina razvoja i učestalost deployment-a aplikacije

Učestalost *deployment-a* značajno utiče na odabir arhitekture:

- **Čest deployment** (npr. više puta dnevno tokom razvoja): automatizacija CI/CD procesa postaje od suštinskog značaja, ali je jednostavnija za implementaciju u okviru monolita.
- **Koordinacija više servisa**: mikroservisi zahtevaju naprednu orkestraciju kako bi se obezbedila sinhronizovana postavljanja.

Tehnološka ograničenja i operativna zrelost

DevOps kapaciteti organizacije predstavljaju značajan ograničavajući faktor:

- **Ograničeno operativno iskustvo**: postavljanje i nadgledanje monolita je jednostavnije i zahteva manje alata i znanja.
- **Napredni operativni kapaciteti**: mikroservisi zahtevaju složenije mehanizme za otkrivanje servisa (*service discovery*), balansiranje opterećenja (*load balancing*) i distribuirani nadzor sistema.
- **Kompleksnost komunikacije između servisa**: direktni pozivi metoda naspram HTTP/REST API poziva za razmenu podataka.

Konačna odluka treba da uspostavi ravnotežu između trenutnih potreba za brzinom razvoja i dugoročnih zahteva za skalabilnošću, uzimajući u obzir sposobnosti postojećeg tima i predviđeni rast sistema.