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** [1] programskom jeziku i **Spring Boot** [2] 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** [3] je korišćen za pakovanje i izolaciju aplikacije, dok je **GitHub Actions** [4] iskorišćen za potpunu automatizaciju procesa kontinuirane integracije i isporuke (***CI/CD*** [1]).

Aplikacija je postavljena na **Netcup cloud** [5] 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[2] za komunikaciju između klijenta i servera
- **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** [6]) pokrenuta u posebnom kontejneru
- **Cloud infrastruktura: Netcup VPS**, korišćen za postavljanje i pokretanje sistema u produkciji

[1] *CI/CD - Continuous Integration/Continuous Deployment*

[2] *RESTful API (Representational State Transfer)* je pristup pravljenja web servisa koji koriste *HTTP* protokol za komunikaciju i manipulisanje resursima na serveru.

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 [3].

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.

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

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** [7] 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.

3 Funkcionalnosti sistema

Razvijeni sistem za rezervaciju konferencijskih sala obuhvata tri ključne funkcionalne celine implementirane kao deo jedinstvene *Spring Boot* aplikacije, čime je obezbeđen kompletan tok obrade zahteva korisnika, od autentifikacije do obaveštavanja putem elektronske pošte.

3.1 Servis za autentifikaciju korisnika

Mehanizam autentifikacije zasniva se na JWT [4] standardu, koji omogućava sesije bez čuvanja stanja (*stateless*), što je pogodno za skalabilna *cloud* rešenja. U sistemu su definisane dve korisničke uloge:

- **Korisnik (USER)** - mogućnost kreiranja i upravljanja sopstvenim rezervacijama
- **Menadžer (MANAGER)** - administrativna ovlašćenja: kreiranje korisničkih naloga i pregled analitičkih izveštaja

Registraciju korisnika vrši isključivo menadžer, čime se sprečava nekontrolisan pristup sistemu. Nakon registracije, korisniku se automatski prosleđuje elektronska poruka sa pristupnim podacima. Mehanizam za promenu lozinke funkcioniše putem bezbedne email procedure, gde se zahtevi za reset lozinke proveravaju na osnovu korisničkog imena i registrovane adrese elektronske pošte.

[4] *JSON Web Token*

3.2 Servis za upravljanje rezervacijama

Interfejs baziran na kalendarskom prikazu omogućava korisnicima intuitivan odabir termina. Sistem koristi naprednu logiku za detekciju konflikata, koja pored vremenskog preklapanja uvažava i obavezne periode za pripremu i čišćenje između rezervacija. U slučaju konflikta, korisniku se nudi objašnjenje kako bi u skladu sa tim prilagodio svoje planove.

Dodatno, korisnicima je omogućeno izmenjivanje i otkazivanje rezervacija koje još nisu započete, čime se obezbeđuje fleksibilnost u planiranju.

3.3 Servis za elektronsku poštu

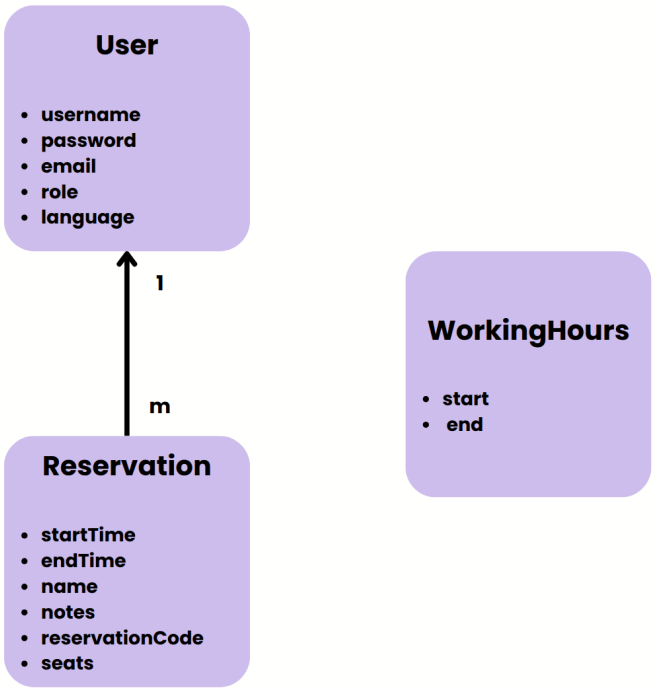
Sistem automatski generiše obaveštenja putem elektronske pošte za sledeće slučajeve:

- Potvrda kreiranja, izmene i otkazivanja rezervacija
- Dobrodošlica korisniku sa pristupnim podacima
- Obaveštavanja o ažuriranjima naloga korisnika
- Izveštaji i obaveštenja za administrativne korisnike

Poruke se generišu na osnovu unapred definisanih šablona koji uključuju dinamičke informacije (datum, vreme, naziv korisnika), kao i vizuelne elemente u skladu sa brendom firme.

3.4 Baza podataka

Struktura baze zasniva se na relacionom modelu, a koristi se *MySQL* kao sistem za upravljanje bazom podataka. Ključne tabele se mogu videti na Slici 1.

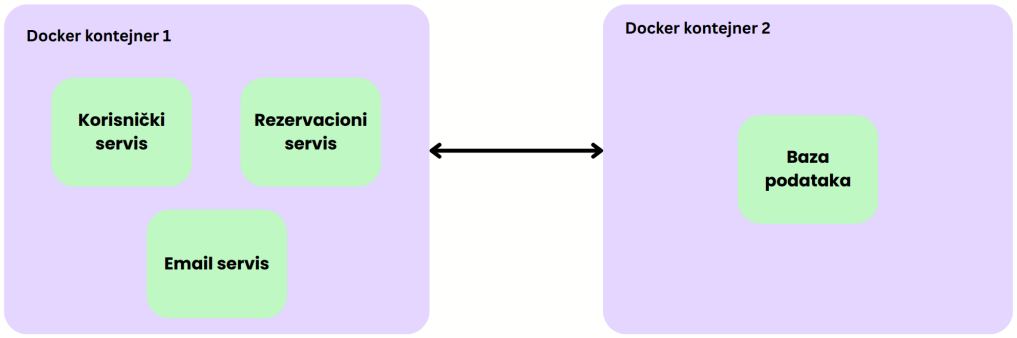


Slika 1: Struktura baze podataka

Tabela `Working_Hours` omogućava definisanje različitih radnih termina u zavisnosti od sezonskih rasporeda, praznika i posebnih događaja. Sistem je projektovan sa mogućnošću proširenja za podršku više sala, uz minimalan uticaj na postojeću logiku.

3.5 Arhitektura sistema

Pregled arhitekture sistema prikazan je na Slici 2.



Slika 2: Dva *Docker* kontejnera su povezana preko virtualne mreže

Aplikacioni kontejner (monolitna arhitektura)

Svi servisi (autentifikacija, rezervacije, elektronska pošta) implementirani su kao deo jedne *Spring Boot* aplikacije i distribuiraju se kao jedinstveni `JAR` [5] fajl. Prednosti ovog pristupa uključuju:

- Direktni pozivi metoda unutar procesa (bez mrežne latencije)
- Zajednički kontekst transakcija i konzistentnost podataka
- Jedinstveni mehanizam za logovanje i rukovanje greškama
- Centralizovano upravljanje konfiguracijom

[5] Java ARchive

Docker fajl za izgradnju kontejnera izgleda ovako:

```
FROM openjdk:17-jdk-slim
COPY conf_room-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 8082
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Kontejner sa bazom podataka

Baza podataka koristi *MySQL 8.0*, postavljena u posebnom kontejneru sa sledećim karakteristikama:

- Provera ispravnosti startovanja sistema (*health check*)
- Upotreba trajnih volumena za očuvanje podataka

Mreža i komunikacija između kontejnera

Komunikacija između kontejnera odvija se unutar prilagožene **Docker** mreže, što omogućava:

- Otkrivanje servisa po imenu kontejnera
- Izolaciju mrežnog sloja od host sistema
- Konfigurisanje putem `.env` fajla
- Bezbednu razmenu podataka unutar sistema

3.6 Automatizacija putem CI/CD procesa

GitHub Actions automatizacija

Proces automatskog postavljanja aplikacije pokreće se svakim slanjem izmena na glavnu granu repozitorijuma (`main`). Sledeći koraci se izvršavaju automatski:

- Preuzimanje koda i izgradnja aplikacije (*Maven build*)
- Usmeravanje konekcije ka serveru (*SSH setup*)
- Čišćenje prethodnih fajlova i prebacivanje novih
- Ponovno pokretanje kontejnera sa novom verzijom

Bezbednost je obezbeđena korišćenjem enkriptovanih ključeva i promenljivih okruženja (*GitHub Secrets*).

Prednosti automatizovanog postavljanja

Vreme potrebno za postavljanje aplikacije smanjeno je sa oko 15 minuta na manje od 3 minuta, što značajno ubrzava testiranje novih funkcionalnosti. Automatizacija takođe eliminiše problem razlika između lokalnog i produkcionog okruženja.

3.7 Postavljanje aplikacije na cloud (Netcup)

Sistem je postavljen na **Netcup VPS** okruženje, koje predstavlja povoljno rešenje za očekivani broj korisnika. Izvršavanje aplikacije direktno preko *Spring Boot runtime* okruženja omogućava efikasno korišćenje resursa.

Trenutni nivo *VPS*-a je dovoljan za postojeći obim saobraćaja, a sistem je projektovan tako da podržava buduće unapređenje u skladu sa rastom broja korisnika.

4 Razlozi za izbor monolitne arhitekture

4.1 Analiza konteksta

Izbor arhitekture u ovom projektu rezultat je specifičnih poslovnih i tehničkih ograničenja, na osnovu kojih je monolitna arhitektura predstavljala najracionalnije rešenje u datim okolnostima.

Zahtevi klijenta i očekivano opterećenje sistema

Klijent (poslovni centar) postavio je jasne funkcionalne zahteve:

- Očekivani broj istovremenih korisnika: desetine korisnika, što je znatno ispod praga koji bi opravdao potrebu za mikroservisnom arhitekturom
- Scenario sa jednom zgradom i jednom salom - jednostavne *CRUD* [6] operacije
- Predvidivost nivoa opterećenja servisa (tokom radnog vremena)
- Potreba za brzim i ekonomičnim rešenjem

[6] Create, Read, Update, Delete

Iz *ROI* [7] aspekta, uvođenje složenije distribuirane arhitekture bi predstavljalo bespotrebno komplikovanje u odnosu na realne potrebe.

Ograničenja u vezi sa timom i rokovima

Sistem je razvijen u okviru dvočlanog tima (frontend i backend/DevOps), što je u velikoj meri uticalo na izbor arhitekture:

- Troškovi koordinacije u mikroservisnim okruženjima bi bili nesrazmerno visoki za ovako mali tim
- Jedinstvena baza koda olakšava razmenu znanja i sinhronizaciju razvoja
- Vremenski okvir od samo mesec i po dana predstavljao je dodatno ograničenje koje isključuje primenu kompleksnih arhitektura

U skladu sa Konvejovim zakonom (*Conway's Law* [8]) - arhitektura softvera je povezana sa strukturom tima i organizacije. U našem slučaju, jedan član je odgovoran za frontend, a drugi za backend i DevOps aspekte, što je dodatno favorizovalo monolitnu arhitekturu.

[7] Return on Investment

4.2 Prednosti monolitne arhitekture u datom kontekstu

Prednosti u razvoju i implementaciji

Monolitni pristup omogućio je jedinstveno razvojno okruženje:

- Komunikacija unutar procesa: direktni pozivi metoda umesto `HTTP` poziva (kašnjenje reda veličine mikrosekundi u poređenju sa milisekundama)
- Zajednički kontekst transakcija: *ACID* [8] svojstva obuhvataju sve komponente
- Automatska konfiguracija putem *Spring Boot*-a

Operativna jednostavnost

Postavljanje i održavanje aplikacije znatno su jednostavniji:

- Jedinstveni artefakt za izgradnju, testiranje i postavljanje
- Nadzor i logovanje vrše se u okviru jedne aplikacione instance
- Niski infrastrukturni troškovi - dovoljan osnovni VPS bez potrebe za platformama za orkestraciju

[8] *Atomicity, Consistency, Isolation, Durability*

4.3 Kritički osvrt

Analiza kompromisa

Ograničenja monolitne arhitekture su poznata i svesno prihvaćena:

- Skalabilnost: vertikalna skalabilnost zadovoljava trenutne potrebe; horizontalna skalabilnost se može uvesti kasnije
- Tehnološka ograničenja: svi servisi koriste isti stek tehnologija
- Skaliranje tima: ovakav pristup nije pogodan za veliki tim, ali je optimalan za dvočlani tim
- Održavanje: monolit može postati teško održiv sa rastom broja funkcionalnosti

Kada bi mikroservisna arhitektura bila primerenija

Mikroservisna arhitektura postaje pogodnija u sledećim scenarijima:

- Tim sa više od pet članova: opravdava razdvajanje sistema na servise radi efikasnijeg razvoja
- Broj korisnika veći od 1000 istovremenih: potreba za nezavisnim skaliranjem pojedinih komponenti
- Regulatorni zahtevi: potreba za izolacijom podataka i većom kontrolom pristupa

Validacija odluke

Izbor monolitne arhitekture u ovom slučaju pokazao se kao pragmatično i opravdano rešenje:

- Planirani rok ispoštovan - sistem postavljen u roku od 1.5 meseca
- Performanse su bile zadovoljavajuće za očekivani nivo opterećenja
- Klijent je u potpunosti zadovoljen - sistem ispunjava sve poslovne zahteve

Strategija za buduću migraciju: eventualna tranzicija ka mikroservisima može započeti izdvajanjem servisa za slanje email obaveštenja, zatim servisa za autentifikaciju, i postepeno dovesti do potpune dekompozicije, ukoliko to budu opravdavale nove poslovne potrebe.

5 Skalabilnost i buduća modularizacija

5.1 Horizontalno skaliranje monolitne arhitekture

U situacijama kada broj istovremenih korisnika pređe prag od 100 korisnika, moguće je sprovesti horizontalno skaliranje postojeće monolitne arhitekture bez potrebe za njenom potpunom transformacijom.

Distribucija opterećenja (*Load Balancing*)

Implementacijom balansa opterećenja (npr. korišćenjem alata kao što je `nginx` [9]) omogućava se raspodela dolaznih zahteva na više instanci iste aplikacije. Ova strategija ne zahteva izmene u kôdu, već se zasniva na replikaciji aplikacije na više servera.

Povećanje broja instanci zahteva koordinaciju konekcija prema bazi podataka. Svaka instanca održava svoj sopstveni "pool" konekcija, ali ukupan broj istovremenih veza mora biti pažljivo ograničen i praćen na nivou baze.

Strategije skaliranja baze podataka

Dalje skaliranje baze moguće je implementacijom tzv. *"read replica"* arhitekture, gde se operacije upisa šalju ka glavnom (*master*) serveru, dok se upiti za čitanje (posebno analitički zahtevi) prosleđuju sekundarnim (*read-only*) instancama baze.

Uvođenje keš sloja kao što je `Redis` [10] može značajno poboljšati performanse sistema, naročito pri učestalim pristupima statičnim podacima (radno vreme, sesije korisnika, rezultati detekcije konflikata). Keš strategija mora obezbediti konzistentnost podataka prilikom izmene termina.

5.2 Postepena tranzicija ka mikroservisnoj arhitekturi

Ukoliko se u budućnosti ukaže potreba za višim stepenom skalabilnosti (preko 1000 korisnika ili razvojni tim sa više od 5 članova), sistem može migrirati ka mikroservisnoj arhitekturi, po uzoru na tzv. *"Strangler Fig Pattern"*[9].

Sekvenca modularizacije

Prvi korak u modularizaciji predstavlja izdvajanje **servisa za slanje elektronske pošte**, budući da je slabije povezan sa ostatkom sistema. Ova funkcionalnost se prirodno odvija asinhrono i može se izdvojiti implementacijom reda poruka (npr. `RabbitMQ` [12]), gde monolit emituje događaje, a novi servis ih obrađuje nezavisno.

Izdvajanje **servisa za autentifikaciju** predstavlja **složeniji korak**, jer je autentifikacija prisutna u gotovo svim delovima sistema. Zbog toga je neophodno uspostaviti **centralizovani servis** za upravljanje instancama servisa koji mora biti visoko dostupan i bezbedan, jer se sve ostale komponente sistema oslanjaju na njega. U ovoj fazi migracije uvodi se i *API Gateway*, koji preuzima ulogu centralizovanog mesta za primanje korisničkih zahteva i njihovo prosleđivanje odgovarajućim servisima, uz dodatne funkcionalnosti kao što su **autentifikacija, autorizacija i limitiranje saobraćaja**.

Konačna dekompozicija vodi ka potpunom razdvajanju sistema na nezavisne mikroservise, pri čemu svaki poseduje sopstvenu bazu podataka i nezavisan životni ciklus. Ovaj pristup omogućava, na primer, dodavanje "Servisa za upravljanje poslovnicama" bez uticaja na postojeće funkcionalnosti.

[9] *"Strangler Fig"* obrazac migracije označava postepenu zamenu funkcionalnosti unutar monolita sa novim mikroservisima bez potrebe za potpunim prekidom postojećeg sistema. [11]

Definisanje granica servisa

Granice između servisa treba da budu definisane na osnovu poslovnih celina, a ne po tehničkim slojevima. Na primer:

- Servis za upravljanje korisnicima (engl. *User Management*) - zadužen za identitet i kontrolu pristupa
- Servis za rezervacije (engl. *Booking*) - logika rezervacija, detekcija konflikata, upravljanje kalendarom
- Servis za obaveštenja - eksterno komuniciranje putem email-a i drugih kanala

Svaki servis mora posedovati vlasništvo nad sopstvenim podacima i interakcije sa drugim servisima moraju se odvijati isključivo preko jasno definisanih *API*-ja ili asinhronih događaja.

5.3 Međuservisna komunikacija

Sinhrona i asinhrona komunikacija

Sinhrona komunikacija (kroz *REST API*) pogodna je za operacije koje zahtevaju trenutni odgovor, kao što su provere dostupnosti ili autentifikacija korisnika. U ovakvim scenarijima, preporučuje se implementacija "prekidača kola" (*Circuit Breaker*) radi otpornosti sistema - u slučaju pada zavisnog servisa, pozivajući servis mora se ponašati predvidivo i stabilno.

Asinhrona komunikacija putem redova poruka (*message queues*) pogodna je za sve operacije koje ne zahtevaju neposrednu potvrdu - na primer slanje obaveštenja ili logovanje aktivnosti. Ovakav pristup omogućava slabije povezivanje servisa (*loose coupling*) i veću otpornost sistema.

5.4 Orkestracija pomoću Kubernetes platforme

Automatsko skaliranje i nadzor

Kubernetes omogućava automatsko skaliranje broja instanci servisa (***pods***) u zavisnosti od opterećenja (CPU, memorija, broj zahteva). Moguće je koristiti i prediktivne algoritme na osnovu istorije prethodnih opterećenja.

Servisi se otkrivaju automatski putem ugrađenog *DNS* sistema, dok postepena ažuriranja (*rolling updates*) omogućavaju neprekidno korišćenje servisa. *Health checks* testovi osiguravaju da samo funkcionalni servisi primaju saobraćaj.

6 Zaključak

U ovom radu prikazana je praktična realizacija sistema za rezervaciju prostorija, sa posebnim osvrtom na izbor arhitekture u savremenom *cloud* okruženju. Pokazano je da **uslovi i potrebe konkretnog sistema igraju ključnu ulogu** prilikom izbora arhitekture.

Ključni nalazi

Odluka da se koristi monolitna arhitektura bila je u potpunosti opravdana datim ograničenjima: mali tim (dva razvojna inženjera), kratak vremenski okvir (mesec i po dana), kao i očekivano nisko opterećenje sistema (nekoliko desetina korisnika). Ovakav pristup omogućio je uspešnu isporuku sistema u predviđenom roku.

Primenom *Conway-jevog zakona* potvrđeno je da je arhitektura sistema prirodno proizašla iz organizacije i strukture tima.

Upotreba *cloud-native* tehnologija, kao što su **Docker** i **GitHub Actions**, značajno je ubrzala razvojni ciklus kroz automatizaciju postavljanja aplikacije na *clould*. Smanjenjem vremena potrebnog za postavljanje sa 15 na svega 3 minuta omogućeno je brzo i često postavljanje novih verzija, što je bilo ključno za iterativni razvoj.

Budući smerovi razvoja

U narednim fazama razvoja sistem se može proširivati u dva pravca: funkcionalno i kroz promenu arhitekture.

Funkcionalno, sistem se može unaprediti podrškom za više sala, integracijom sa eksternim kalendarima i uvođenjem napredne statistike za administratore. Takođe, dalji razvoj može obuhvatiti bolji nadzor i automatsko testiranje radi povećanja pouzdanosti i održivosti.

Iz aspekta arhitekture, najpre je moguće izdvojiti servis za slanje imejlova kao nezavisnu komponentu, čime bi se započeo prelazak ka mikroservisnoj arhitekturi. U daljem koraku, primenom orkestracionih alata poput **Kubernetes**-a, može se omogućiti automatsko skaliranje i lakše upravljanje višestrukim servisima.

Na taj način, sistem ostaje fleksibilan i tehnološki održiv, sa jasnim pravcem ka daljoj modularizaciji i proširenju u skladu sa rastućim poslovnim potrebama.

Literatura

[1] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[2] Spring Boot Team. *Spring Boot Reference Documentation*. Version 3.2.2. Spring. 2025. URL: https://docs.spring.io/spring-boot/ (visited on 06/27/2025).

[3] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[4] github. *GitHub*. 2025. URL: https://github.com/.

[5] netcup GmbH. *netcup: Your Partner for Web Hosting, Servers, Domains*. https://www.netcup.com/en. Accessed: 2025-06-27. Germany: netcup GmbH, 2025.

[6] MySQL Development Team. *MySQL Reference Manual*. Version 4.0. MySQL AB. Sweden, 2002. URL: https://dev.mysql.com/doc/refman/4.0/en/ (visited on 06/27/2025).

[7] Kubernetes Contributors. *Kubernetes Documentation*. Open-source container orchestration system. Cloud Native Computing Foundation. 2025. URL: https://kubernetes.io/ (visited on 06/27/2025).

[8] John McManus. "Conway's Law: A Focus on Information Systems Development". In: *ITNOW* 61 (Dec. 2019), pp. 50–51. DOI: 10.1093/itnow/bwz112.

[9] Will Reese. "Nginx: the high-performance web server and reverse proxy". In: *Linux J.* 2008.173 (Sept. 2008). ISSN: 1075-3583.

[10] Redis Contributors. *Redis Documentation*. Version 8.0. Redis Ltd. 2025. URL: https://redis.io/docs/latest/ (visited on 06/27/2025).

[11] Martin Fowler. *Strangler Fig Application*. Accessed: 2025-06-27. 2004. URL: https://martinfowler.com/bliki/StranglerFigApplication.html (visited on 06/27/2025).

[12] RabbitMQ Team. *RabbitMQ Documentation*. Version 3.13. VMware, Inc. 2025. URL: https://www.rabbitmq.com/documentation.html (visited on 06/27/2025).