SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mislav Perić

RAZVOJ SIGURNIH MIKROSERVISNIH ARHITEKTURA

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mislav Perić

Matični broj:

Studij: Primjena informacijske tehnologije u poslovanju

RAZVOJ SIGURNIH MIKROSERVISNIH ARHITEKTURA

ZAVRŠNI RAD

Mentor/Mentorica:

Izv. prof. dr. sc. Markus Schatten

Varaždin, studeni 2022.

Mislav Perić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu govoriti će primarno o mikroservisnim arhitekturama i njihovom dizajnu. Kroz rad detaljno ćemo opisati ključne koncepte mikroservisa, objasniti što su mikroservisi, principe autentifikacije i autorizacije, kako mikroservisi komuniciraju i kako rade sa bazama podataka. Pokazat ćemo programski jezik Elixir koji će nam služiti za izgradnju mikroservisa, dotaknuti klijentskog dijela sa paketom React i varijacijom JavaScript jezika zvan TypeScript. Osvrnut ćemo se na web server NGINX i njegovu ulogu kao API Gateway i kao poslužitelj statičnih datoteka. Za komunikaciju ćemo se dotaknuti asinkrone komunikacije između servisa koristeći Kafku i sinkrone komunikacije putem HTTP zahtjeva. Pogledat ćemo kako mikroservisi komuniciraju na siguran način putem JWT i koristeći API ključeva kroz Gateway. Pogledat ćemo o koji se tehnologijama radi, kako funkcioniraju i što rade, pokazati primjer implementacije te na kraju donijeti zaključak kako poboljšati našu implementaciju

Ključne riječi: Mikroservisi, Elixir, React, PostgreSQL, Phoenix, Auth0, JWT, NGINX., Docker, Typescript, API, Gateway, Ecto, Git, Kafka

SADRŽAJ ..... ERROR! BOOKMARK NOT DEFINED.

Uvod

Početkom rasta popularnosti interneta, pojavila se i veća potreba za izgradnjom raznih web stranica i web aplikacija. U početku, svaka aplikacija je bila monolit, jedna velika aplikacija koja se zadužena za cijelu komunikaciju i koja je obavljala sve funkcije i procese aplikacije. Kako se sam Internet širio i dolazio u više kućanstava i dolaskom pametnih mobitela koji rade na iPhone i Android operativnim sustavima, poslužitelji imaju zadatak posluživati sve više i više zahtjeva. Eksplozijom prijenosnih uređaja i sve veća dostupnost interneta dovodi potrebu za sve više poslužiteljskog prostora i resursa. Rješenjem koje su velike firme koje procesiraju milion i milione zahtjeva u minuti do sada je bilo zakupiti više prostora i skalirati monolitnu aplikaciju. Početkom 2010. godina pojavljuje se pojam mikro web servisi kao odgovor na probleme skaliranja monolita kod veliki kompanija. Primjernom mikroservisa, gdje je svaki servis odgovaran za neku zadaću, velike kompanije kao Amazon su shvatile da korištenjem mnogo mikroservisa mogu uvelike uštediti na skaliranju. Ne moraju više skalirati cijeli monolit, nego sukladno zahtjevima i potrebama mogu skalirati određene servise.

Tehnologije

Moderni sustavi zahtijevaju sve naprednije alate i arhitekture kako bi što lakše i elegantnije procesirali sve veći broj korisnika i zahtjeva. IT industrija zbog svoje prirode napreduje rapidno i svake godine svjedočimo pregršt novih tehnologija i koncepata. Mikroservisi kao koncept relativno novi koncept koji je doživio porast u popularnosti u zadnjih 5 godina. Sa rastom popularnosti mikroservisa, porasla je popularnost tehnologija koje su pogodne za izgradnju servisa te uz njih je uvedeno mnoštvo novih tehnologija i koncepata kako bi odgovorili na same probleme koji su se pojavili izgradnjom mikroservisa.

1.1. Elixir

Elixir je alternativni jezik za Erlang virtualne mašine koji nam omogućava da pišemo čišći, kompaktniji kod koji je bolji u otkrivanju svrhe napisanog koda. Elixir je open source projekt, originalno započet od strane José Valima. [1]

Rezultat kompaliranog (eng. *compiled*) izvornog koda Elixira je BEAM sukladan bajt datoteka koju možemo pokrenuti na BEAM instanci. Sukladno tome, možemo koristiti Erlang dodatke u Elixiru, i obrnuto. Sve što Elixir radi, može i Erlang.

Jedna od najvažnijih prednosti Elixira mogućnost da drastično smanji repetitivnog (eng. *boilerplate)* koda, što rezultira u jednostavnijem kodu koji je lakše pisati i održavati. Često korištena osnova u Erlang istodobnim (eng. *concurrent*) sistemima je serverski proces. Serverski proces možemo zamisliti kao istodobni (eng. *concurrent*) objekti – oni spremaju privatno stanje i mogu komunicirati s drugim procesima preko poruka. Tipičan Erlang sistem se oslanja na procese, koristeći tisuće ili čak milione aktivnih procesa.

```
-module(sum_server).
-behaviour(gen_server).

-export([
  start/0, sum/3,
  init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
  code_change/3
]).

start() -> gen_server:start(?MODULE, [], []).
sum(Server, A, B) -> gen_server:call(Server, {sum, A, B}).
```

```
init(_) -> {ok, undefined}.
handle_call({sum, A, B}, _From, State) -> {reply, A + B, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

Na prvu ruku, nekome tko ne razumije Erlang i njegov kod, učinilo bi se da ovaj kod radi mnogo više od zbrajanja dva broja. Erlangova mana je da se većina repetitivnog (eng. *boilerplate)* koda ne može maknuti. Taj pristup često dovodi to koda, koji je čak u jednostavnim primjerima, nečitak i teško razumljiv. Pogledajmo sad primjer iste operacije u Elixiru

```
defmodule SumServer do
  use GenServer

  def start do
    GenServer.start(__MODULE__, nil)
  end

  def sum(server, a, b) do
    GenServer.call(server, {:sum, a, b})
  end
```

Ovaj primjer je puno manji i lakši za razumjeti i održavati, ali je jednako kompetentan kao Erlang primjer. Kao što je navedeno iznad, Elixir može raditi sve što Erlang može. Gotovo svakom developeru, bez obzira na prijašnje iskustvo u Elixiru ili Erlangu, razumije što ovaj kod radi i koja je njegova svrha.

SumServer proces je primjer makro naredbe. Makronaredba je dio Elixir koda koja se pokreće u trenutku kompaliranja (eng. *compiled*) koda. Makronaredbe kao argument uzimaju internu reprezentaciju izvornog koda kao input i mogu vratiti alternativni output. Inspiracija dolazi iz programskog jezika Lisp. Primjeri Elixir makronaredbi su konstrukti kao `if, else i unless`. [1]

Elixir, kao i Erlang na kojem je zasnovan, funkcijski jezik. Osnova mu je nepromjenjivi podaci i funkcije koje izmjenjuju te podatke. [1]

Kako svaki jezik ima svoje prednosti, tako ima i nedostatke. Python je relativno spor. Javascript ima relativno čudna ponašanja na nekim primjerima. Na primjer, što mislite da će `console.log(('b' + 'a' + + 'a' + 'a').toLowerCase())` ispisati. Kladim se da vaš odgovor nije bio `` `banana` ``. Naime, tvorci Javascripta su mislili da je dobra ideja u slučaju da pokušate dodati broj stringu, on će pretvoriti taj broj string.

Navode se dva velika nedostatka Elixira: brzina i ekosustav. Erlang kao baza nije brza upravo zbog BEAM-a i ne mogu postići brzinu koda kompaliranih (eng. *compiled)* na samim računalima kao C ili C++, ali ovo nije greška ili loš dizajn od strane Erlang tima. Erlang, a pri tome i Elixir, nisu napravljeni da obradite što više zahtjeva u sekundi. Sustavi izgrađeni na Erlangu ne bi trebali patiti od degradiranih performansa kada, recimo, garbage collector bude pokrenut. Primjer jezika gdje je ovo osjetno je Go, kao što je Discord naveo u svojoj objavi kada su se prebacili s Golanga na Rust. [2]

Drugi razlog, kao navedeno gore, je ekosistem, ili točnije, nedostatak ekosistema. Dok on postoji, naspram jezika kao Ruby, JavaScript, broj javnih repozitorija koji koriste Elixir kao jezik je relativno sitan naspram prijašnje navedenih jezika.

1.1.1. Osnove Elixira

1.1.1.1. Varijable i rad s varijablama

Elixir je dinamično pisan jezik, tako da nije potrebno prije svake varijable pisati tip varijable, kao što trebamo u jezicima kao Go. Tip varijable je određen tipom podatka koje sadrži. Sam čin dodjeljivanja varijable u Elixiru se zove vezanje (eng. *binding).* Konvencija je da varijable počinju s mailom slovom, te da se svaka sljedeća riječ odvaja sa donjom crtom (_). Redeklariranje, ili u slučaju Elixira, ponovno vezanje (eng. *rebinding*) varijable ne mijenja originalnu vrijednost memorije na toj lokaciji, već uzima novi dio memorije i ponovno dodijeli simbolično ime varijable na tu novi lokaciju. Sukladno tome, varijable su promjenjive, dok sami podaci nisu. Elixir sadrži takozvani garbage collector. Kada varijabla više nije korištena, ta memorija je kandidat za garbage collector i nekad u budućnosti će biti oslobođena. [1]

1.1.1.2. Moduli

Modul je kolekcija funkcija. Svaka funkcija u Elixiru mora biti unutar modula. Moduli se definiraju s `defmodule`, unutar modula koristimo *def* da bi definirali funkciju. Imena modula moraju pratiti određena pravila. Mora počinjati s velikim slovom i pisana je u CamelCase stilu. Može sadržati alfanumeričke znakove, donje crte i točku. [1]

1.1.1.3. Funkcije

Funkcije moraju pratiti iste konvencije kao varijable. Počinju s malim slovom ili donjom crtom, popraćene s alfanumeričkim znakovima i donjim crtama i mogu završavati s upitnikom ili uskličnikom. Upitnik često označava da funkcija vraća boolean vrijednost. Postavljanjem uskličnika na kraj funkcije znači da funkcija može vratiti runtime error. Oba pravila su konvencije, ali je najbolje da ih pratimo. Sintaksa funkcije je:

```
def <function_name> do
    …
end
```

Sve što u Elixiru ima povratnu (eng. *return)* vrijednost je izraz (eng. expression*)*. Povratni (eng. *return)* izraz u Elixiru je zadnji izraz (eng. *expression*) u funkciji. Ne postoji eksplicitni povrat (eng. *explicit return)* u Elixiru. [1]

```
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end
end”
```

Pošto je Elixir funkcionalni jezik, često ćemo morati kombinirati funkcije, gdje ćemo davati sljedećoj funkciji rezultate prijašnje funkcije kao argument. U ovom slučaju možemo koristiti pipeline operator `|>`

```
iex(5)> -5 |> abs() |> Integer.to_string() |> IO.puts() [1]
```

1.1.1.4. Argumenti funkcija (*Arity)*

Function Arity još jedan naziv za broj argumenata u funkciji. U Elixiru, dvije funkcije mogu imati isto ime, ali različit broj argumenata. Generalno, dvije funkcije koje imaju isto ime mogu biti zbunjujuće, pa je generalno praksa da je Function Arity rezerviran za funkcije koje mogu primati jedan ili više argumenata, da pri tome rade sličnu stvar, uz to da im se izvedba mijenja ovisno o broju argumenata koji su dani. Primjer takve funkcije možete vidjeti ispod:

```
defmodule MyString do
  def split(string) do
    split(string, " ")
  end

  def split(string, pattern) do
    # Split the string by the pattern
  end
end
```

[3]

Elixir nam dopušta da kreiramo funkcije sa zadane (eng. *default*) vrijednostima za neke argumente.

```
defmodule Calculator do

  def sum(a, b \\ 0) do
    a + b
  end
end
```

U ovom primjeru, Elixir za nas kreira dvije funkcije kroz function arity: jednu koja uzima dva argumenta i jednu koja uzima jedan argument, gdje postavi `b` na vrijednost 0. Zbog function arity-a Elixir ne može primiti dinamičan broj varijabli u funkciji, broj uvijek mora biti finalan i konstantan. [1]

U Elixiru razlikujemo privatne i javne funkcije. Privatne funkcije mogu koristiti samo moduli u kojima je definirana, dok javne funkcije može koristiti cijeli program. Privatne funkcije se deklariraju s `defp`, dok se javne funkcije deklariraju s `def`.

```
def double(a) do
    sum(a, a)
  end

  defp sum(a, b) do
    a + b
  end
end
```

1.1.1.5. Importanje i alijasi

Često uvezemo (eng. *import)* funkcije iz drugih modula kako bi ubrzali i olakšali razvijanje naših programa. Kako bi olakšali taj posao, možemo uvesti (eng. *import*) cijeli modul u naš program. Uvozom (eng. *import*) modula dobivamo mogućnost da pozivamo njegove javne funkcije bez da prije toga pozovemo cijeli modul

```
defmodule Geometry do
  import IO

  def randomthing do
    puts "Calling imported function"
  end
end
```

Još jedan konstrukt, `alias`, nam omogućava da referenciramo modul pod drugim imenom.

```
defmodule Geometry do
  alias IO, as: Careless

  def randomthing do
    Careless.puts "Calling imported function"
  end
end
```

Korištenje `alias` za importiranje modula je korisno kada module koji importiramo ima dugo ime. Uz to, možemo kompletno izostaviti `as` ako uvozimo (eng: *import)* ime funkcije

```
defmodule Geometry do
  alias IO, as: Careless

  defmodule Rectangle do
    def area do
      #do some thing here
    end
  end
end

defmodule Gems do
  alias Geometry.Rectangle
  def hidden do
    Rectangle.area()
  end
end
```

1.1.1.6. Module atributa

Funkcija atributa modula je dvostrana: Mogu biti korišteni kao konstante tijekom kompiliranja (eng. *compile)* i možemo registrirati bilo koji atribut koji se koristi u vremenu izvođenja (eng. *runtime).* Moduli mogu biti registrirani, što znači da ih spremamo u generirani binary i možemo im pristupiti u vremenu izvođenja (eng. *runtime)*. Elixir registrira `@moduledoc` i `@doc` atribute koje možemo koristit za dokumentiranje koda. [1]

```
defmodule Circle do
  @moduledoc "Implements basic circle functions"
  @pi 3.14159
```

```
  @doc "Computes the area of a circle"
  def area(r), do: r*r*@pi

  @doc "Computes the circumference of a circle"
  def circumference(r), do: 2*r*@pi
end
```

Ali, da bi koristili atribute, moramo generirati kompiliranu (eng. *compiled*) datoteku. Kompiliranu (eng. *compiled*) datoteku kreiramo s komandom `elixirc <ime-datotekte.ex>`. Možemo koristiti `ex_doc` alat kako bi kreirali HTML dokumentaciju za projekt. Poanta registriranih atributa je da pružaju metadata informacije, koje mogu koristiti drugi Elixir alati. Type specification je još jedna značajka bazirana na atributima. Ovaj značajka dopušta da dodamo informaciju o tipu podataka za funkcije koji se kasnije može analizirati sa statičnim analitičkim alatima, kao dialyzer [1]

```
defmodule Circle do
  @pi 3.14159

  @spec area(number) :: number
  def area(r), do: r*r*@pi

  @spec circumference(number) :: number
  def circumference(r), do: 2*r*@pi
end
```

1.1.2. Tipovi podataka

Kako je Elixir nastao na Erlangu, sadrži sve Erlang tipove podataka te proširuje izbor sa nekim novima.

1.1.2.1. Brojevi

Brojevi mogu biti integeri ili float, koji se ponašaju kao i brojevi u drugim jezicima. Podržane su standardne aritmetičke operacije, kao oduzimanje, zbrajanje, množenje i dijeljenje, ali operator za dijeljenje (/) uvijek vraća decimalni broj. Kako bi dijelili s cijelim brojevima ili izračunali ostatak, možemo koristiti funkcije iz modula koji se automatski importa `Kernel`, a to su funkcije `div` i `rem`. Ne postoji gornji limit na brojeve, integer uzima koliko mu treba prostora, dok float uzima 32 ili 64 bita, ovisno o arhitekturi virtualne mašine. [1]

1.1.2.2. Atomi

Atomi su konstante kojima je vrijednost njihovo ime. [4] Počinju s : nakon čega dodamo alfanumeričke znakove ili donju crtu. Moguće je koristiti razmake ako ih okružimo s

navodnicima. Sastoje se od dva dijela: teksta i vrijednosti. Atom tekst je sve što stavimo nakon dvotočke i sadržana je u vrijeme izvedbe (eng .*runtime*) u atom tablici. Vrijednost su podaci koji idu u varijablu i oni su samo referenca na atom tablicu. Atomi imaju u drugu sintaksu. Možemo ih deklarirati s početnim velikim slovom, što se zove `alias` i u kompiliranju koda se pretvara u `AnAtom > :"Elixir.AnAtom"`. Umjesto boolean, Elixir koristi atome `:true` i `:false` kao boolean vrijednosti, ali kao lakše pisanje, Elixir nam omogućava da ove atome koristimo bez `:` na početku. Poseban atom je `:nil` vrijednost, koja je jednak `nil` vrijednosti u Go jeziku. U Elixiru, samo su `nil` i `false` vrijednosti falsy, dok su sve ostale vrijednosti truthy. [1]

1.1.2.3. Tuples

Tupli su nešto kao untyped struktura, najviše su korišteni kako bi grupirali fiksan broj elemenata skupa. Sintaksa im je `person = {„Mislav", 23}`. Kako bi uzeli element iz tupli, korisitmo `Kernel.elem` funkciju, koja prima tuple i indeks elementa. Elemente tuple možemo modificirati korištenjem `Kernel.put_elem` funkcije, koja prima kao argumente tuple, indeks i novi vrijednost za taj indeks. Zbog prirode Elixira, `put_elem` ne modificira tuple, nego kreira novi element. [1]

1.1.2.4. Liste

Liste, kao u Pythonu ili u Go slices, su korištene kako bi kreirali dinamične kolekcije podataka varijabilne količine podataka. Sintaksa je slična gore navedenim primjerima `new_list = [1, 2, 3, 4]`. Većina operacija na listama ima `O(n)` kompleksnost, koje iteriraju kroz cijelu listu. [1]

1.1.2.5. Mape

Mape su klasičan primjer ključ:vrijednosti strukture, gdje ključ i vrijednost mogu biti bilo koji pojam. Postoje dva načina korištenja mapa u Elixiru. Prvi su ključ:vrijednosti strukture dinamične veličine, dok je drugi upravljanje zapisima, nekoliko dobro definiranih polja u skupu. Sintaksa mapa je `new_map = %{1 => 1, 2 => 4, 3 => 9}`. Kako bi dohvatili ključ mape, koristimo `new_map[2]`, što nam da output `4`.

Mape se često koriste kako bi prikazali strukturirane podatke. Uzmimo primjer mape koja predstavlja jednu osobu `mislav = %{:name => „Mislav", :age => 23, :works_at => „Axilis"}`. U slučaju da su ključevi atomi, istu mapu možemo ovako zapisati `mislav = %{name: „Mislav", age: 23, works_at: „Axilis"}`. Kako bi dohvatili polje, možemo koristiti `mislav[:name]` što nam daje output `„Mislav"`. Atom ključevi opet dobivaju posebnu sintaksu, u JavaScriptu zvanu *dot-notation,* `mislav.works_at` nam daje output `„Axilis"`. Kako bi promijenili polje koristimo `%{mislav | age: 25}` što nam daje output `%{:name =>`

„Mislav", :age => 25, :works_at => „Axilis"}, ali možemo modificirat samo vrijednosti koje već postoje mapi. [1]

1.1.2.6. Binary i bitstringovi

Binary je komad bitova. Binary možemo kreirati tako da okružimo sekvencu bitova sa << i >> operatorima. Ovaj primjer kreira binary od 3 bita: <<1, 2, 3>>. Možemo specificirati veličinu vrijednosti i reći kompajleru (eng. *compiler*) koliko bitova da koristi za tu vrijednosti: <<257::16>>. Ovaj primjer sprema broj 257 u 16 bita memorije. Output je <<1, 1>> što znači da koristimo dva bajta, koji oba imaju vrijednost 1. Ovo je zbog binarne reprezentacije broja 257, koja ide 0000000100000001. U slučaju da totalna veličina nije množitelj broja 8, binary se naziva bitstring – sekvenca bitova: <<1::1, 0::1, 1::1>>. Možemo zbrajati dva binary ili dva bitstringa s operatorom <>: [1]

1.1.2.7. Stringovi

Elixir ne sadrži tip string. Stringovi su prikazani pomoću binary ili liste. Kod binary stringova najčešći način za pisanje stringova je da ih pišemo unutar navodnika. Rezultat je ispisan kao string, ali u pozadini to je binary – ništa više od sekvencije bitova. Elixir ima podršku za dodavanje izraza (eng. *expression)* u stringove pomoću operatora #{}. Elixir pruža još jedan način deklariranja stringova koji se zove sigil i deklarira se sa ~s(): ~s(This is also a string). Sigili su korisni ako želimo uključiti navodnike unutar stringova. Postoji verzija sigila koja se deklarira sa ~S koja ne uključuje interpolaciju i ne uključuje interpolaciju i ne dopušta korištenje znaka \. Na kraju, postoji posebna heredocs sintaksa koja podržava bolje formatiranje za stringove koji se koriste za bolje formatiranje stringova koji se produžuju na više redova. Deklariraju se sa """ i završni navodnici moraju biti u zasebnom redu. Pošto su stringovi binaries, mogu se dodavati s operatorom <>. [1]

Alternativni način da prikažemo stringove koristeći jednostruke navodnike. Ovo kreira listu znakova, što je zapravo lista brojeva di svaki element predstavlja jedan znak. Liste znakova i binary string nisu međusobno kompatibilni, pa bi trebali što je više moguće koristiti binary stringove. [1]

1.1.2.8. Funkcije prve klase

U Elixiru, funkcija su „građani prve klase", što znači da ih možemo pridružiti varijablama. Kada pridružimo funkciju varijabli, ne pozivamo ju i ne spremamo rezultat u varijablu, nego definicija funkcije je pridružena i možemo koristiti tu varijablu da pozivamo funkciju.

```
square = fn x ->
      x * x
end
```

Ovakve funkcije nisu pridružene globalnom imenu se još nazivaju anonimne funkcije ili lambda. Kao što vidimo, argumenti funkcije nisu okruženi zagradama. Dok ih tehnički možemo okružiti zagradama, konvencija je da ih ne okružimo zagradama. Ove funkcije pozivamo tako da pozovemo ime varijable i stavimo točku, nakon koje dodamo argumente [1]

```
square = fn x ->
      x * x
end
iex(1)> square.(5)
25
```

Kako funkcije mogu biti spremljene u varijable, tako mogu biti argument drugih funkcija. Primjer je `Enum.each` funkcija koja prima dva argumenta: prvi je enumerable, a drugi je lambda koja prima jedan argument. Iterira kroz enumerable i poziva lambdu za svaki element. [1]

```
print_elem = fn x -> IO.puts(x end
Enum.each(
      [1, 2, 3],
      print_elem
)
1
2
3
:ok
```

Postoji kompaktniji način pisanja lambda funkcija. Umjesto da pišemo `fn x -> IO.puts(x) end`, možemo napisati `&IO.puts/1`. Još jedan način da smanjimo lambdu je kombinacija capture `&` operatora, Pretvorimo `fn x, y, z -> x * y * z end` u `&(&1 * &2 + &3)`, Ovaj isječak kreira lambdu koja uzima tri argumenta. Lambda može biti referencirana van djelokruga (eng. *scope)* di je definirana. Sve dok držimo referencu varijabli, indirektno držimo referencu svim varijablama koje koristi, čak ako je ta varijabla iz eksternog djelokruga (eng. *scope).* Closure uzima specifičnu lokaciju memorije, ponovno definiranje varijable ne utječe na prije definiranu lambdu koja referencira isto simbolično ime. Što ovo znači. Kada deklariramo lambdu, trenutna vrijednost neke varijable ostaje u toj lambdi. Ako promijenimo vrijednost te varijable, lambda je „zapamtila“ lokaciju memorije varijable, tako da se vrijednost u lambi ne mijenja. [1]

```
iex(1)> outside_var = 5
iex(2)> lambda = fn -> IO.puts(outside_var) end
iex(3)> outside_var = 6
iex(4)> lambda.()
5
```

U ovom slučaju, originalna lokacija varijable `outside_var` nije kandidate za garbage collection pošto je referenciramo unutar lambde. Lokacija će postati kandidat za garbage colleciton tek kada lamba funkcija otpusti referencu na tu lokaciju.

1.1.2.9. Range

Range je abstrakcija koja nam dopušta da predstavimo niz brojeva.

```
iex(1)> range = 1..2
iex(2)> 2 in range
true
iex(3)> 3 in range
false
```

Range su enumerable pa možemo koristiti `Enum` model. Range nije poseban tip podataka. U svojoj srži, range je samo mapa koja sadrži zadane granice [1]

1.1.2.10. Keyword liste

Keyword lista je posebna vrsta liste gdje je svaki element tuple s dva elementa, gdje je prvi element atom, dok drugi element može biti bilo koje vrste [1]

```
iex(1)> keyword_list = [monday: 1, tu: 2]
```

Keyword liste se često koriste za male ključ:vrijednost strukture, gdje su ključevi atomi, kao što je navedeno iznad. Jedna stvar koju moramo pripaziti kod keyword lista je brzina dohvaćanja podataka. Zbog prirode keyword lista, kompleksnost dohvaćanja podataka je `O(n)` notacija. [1]

Keyword liste se najčešće koriste kada dajemo opcionalne argumente funkciji u Elixiru.

```
iex(1) IO.inspect[100, 200, 300], width: 3, limit: 1)
[100,
…]
```

Kao što vidite, iz opcionalnih argumenata možemo izostaviti `[]` zbog učestalosti korištenja ovog patterna. [1]

Ovaj primjer nam pokazuje kako smo funkciji `IO.inspect` dali dva argumenta: brojeve i keyword listu sa dva elementa. Možemo prihvatiti keyword list kao zadnji argument u funkciji i staviti da je zadane argument prazna list:

```
def my_fun(arg1, arg2, opts \\ []) do
      …
```

Keyword liste, za razliku od mapa, mogu sadržavati više elemenata za isti ključ. Uz to, možemo kontrolirati redoslijed tih elemenata, što nije moguće s mapama. [1]

1.1.2.11. MapSet

MapSet je implementacija seta – struktura podataka koja ima samo unikatne vrijednosti, gle vrijednost može biti bilo kojeg tipa. [1]

```
iex(1)> mapset = MapSet.new(["monday", "monday", "tuesday"])
#MapSet<["monday", "tuesday"]>
```

1.1.2.12. IO Liste

IO liste su posebna vrsta listi koja služi za inkrementalnu izgradnju outputa koji će biti prosljeđen I/O uređaju, kao mreži ili datoteci. Svaki element IO liste mora biti jedno od sljedećih:

- Broj od 0 do 255
- Binary
- IO lista

IO lista je duboko ugniježđena struktura gdje su elementi bitovi ili sekvenca bitova. [1]

```
iex(2)> iolist = []
[]
iex(3)> iolist = [iolist, "This"]
[[], "This"]
iex(4)> iolist = [iolist, "is"]
[[[], "This"], "is"]
iex(5)> iolist = [iolist, "an"]
[[[[], "This"], "is"], "an"]
iex(6)> iolist = [iolist, "IO"]
[[[[[], "This"], "is"], "an"], "IO"]
iex(7)> iolist = [iolist, "list"]
[[[[[[], "This"], "is"], "an"], "IO"], "list"]
iex(8)> IO.puts(iolist)
ThisisanIOlist
:ok
iex(9)>
```

Struktura je izravnana na jednu razinu, gdje vidimo output čitljiv ljudima. IO liste su korisne kada moramo inkrementalno graditi sekvencu bitova. Liste nisu idealne zbog notacije `O(n)` kompleksnosti kada dodajemo elemente listi, u slučaju kada dodajemo IO listi notacija je `O(1)` kompleksnosti, iz razloga što koristimo gniježđenje. U primjeri iznad, dodajemo IO listi

tako da kreirano novu IO listu koja ima dva elementa: prijašnju verziju IO liste i novi element koji želimo dodati [1]

1.1.3. Operatori

U primjerima koji smo prošli do sad suočili smo se s većinom operatora koje Elixir nudi. Elixir, kao i drugi jezici, ima aritmetičke operatore: `+`, `-`. `*` i `/` te funkcioniraju kako bi očekivali, uz iznimku da / operator vraća decimalni broj kao rezultat operacije. Operatori uspoređivanja su uglavnom ono to bi očekivali

Tablica 1. Logički operatori u Elixiru

===, !==	Striktna jednakost/nejednakost
==, !=	Slaba jednakost/nejednakost
<, >, <=, >=	Manje, veće, manje ili jednako, veće ili jednako

(Izvor: Elixir in Action, Saša Jurić)

Kao što vidimo, Erlang i Elixir su odlučili uzeti JavaScript pristup jednakosti i imati slabe i striktne jednakosti. Razlika je da striktna jednakost/nejednakost provjerava tip podataka, dok slaba jednakost/nejednakost ne provjerava tip podataka, nego samo vrijednost. Logički operatori rade na principu boolean atoma, to su: [¡]

- `and`
- `or`
- `not`

Short-circuit operatori rade na principu istinitosti: atomi `false` i `nil` se tretiraju kao neistine vrijednosti, dok se sve ostalo tretira kao istinita vrijednost. Operator `&&` vraća drugi izraz jedino ako je prvi izraz istinit. Operator `||` vrati prvi izraz u slučaju da je istinit, inače vraća drugi izraz. Operator `!` vrši inverziju na vrijednost nad kojom je upotrjebljen. [1]

1.1.4. Makro naredbe

Makro naredbe su jedne od najvažnijih elemenata Elixira. Omogućuju nam izvedbu moćnih transformacija koda u vrijeme kompiliranja koda. Makro naredba sadrži Elixir kod koji može mijenjati semantiku koda koji je predan makro naredbi. Makro naredbe uvijek primaju parsiranu reprezentaciju koda koji joj je predan i ima priliku vratiti alternativnu verziju tog koda. [1]

```
iex(9)> defmodule Unless do
...(9)>   def fun_unless(clause, do: expression) do
...(9)>     if(!clause, do: expression)
...(9)>   end
```

```
...(9)>
...(9)>   defmacro macro_unless(clause, do: expression) do
...(9)>     quote do
...(9)>       if(!unquote(clause), do: unquote(expression))
...(9)>     end
...(9)>   end
...(9)> end
iex(10)> require Unless
Unless
iex(11)> Unless.macro_unless true, do: IO.puts "this should never be
printed"
nil
iex(12)> Unless.fun_unless true, do: IO.puts "this should never be printed"
this should never be printed
nil
```

U slučaju makro implementacije, rečenica nije ispisana, dok u slučaju implementacije funkcije rečenica je ispisana. Zato što su argumenti u slučaju poziva funkcije procijenjeni prije poziva same funkcije. U slučaju makro naredbi, makro naredbe argumente kao quoted expression koje dalje transformiraju u daljnje quoted expression. [5]

1.1.5. Control flow

Elixir, kao i svaki drugi jezik, ima control flow, točnije način kako upravljati tokom koda. Klasični kondicionalni konstrukti su zamijenjeni funkcijama s više klauzula te ne postoje klasične petlje. [1]

1.1.5.1. Pattern matching

Kao što smo prije napomenuli, operator = nije dodjeljivanje. Kada smo koristili operator =, time smo rekli da je varijabla na koju smo koristili taj operator vezana za neku vrijednost. Operator = se u Elixiru zove match operator, a izraz kada vežemo varijablu za vrijednost je primjer pattern matchinga. [1]

1.1.5.2. Match operator

Pogledajmo najosnovniji način korištenja match operatora:

```
iex(1)> a = 1
1
```

Tretiramo ga kao pridruživanje, ali u stvarnosti se događa nešto kompleksnije. U vremenu izvođenja (eng. *runtime*), lijeva strana operatora = je izjednačena desnoj strani. Lijeva strana se zove pattern, gdje je desna strana Elixir izraz. [1]

1.1.5.3. Matching tuples

Ovaj primjer demonstrira jednostavan pattern matching tipa tuple:

```
iex(1)> {name, age} = {„Mislav“, 23}
```

Ovaj izraz pretpostavi da je izraz s desne strane tuple od dva elementa. Kada je izraz izvršen, varijable `name` i `age` vezane za odgovarajuće elemente tupla. Ova funkcionalnost je korisna kada zovemo funkciju koja nam vraća tuple i želimo individualne elemente vezati za različite varijable. Što se događa u slučaju da desna strana ne odgovara patternu. Match operator neuspješno pokušava vezani vrijednosti i program nam vraća grešku [1]

```
iex(2)> {nothing, everything} = "this is an error"
** (MatchError) no match of right hand side value: "this is an error"
```

1.1.5.4. Varijable u patternima

Kada postoji varijabla s lijeve strane patterna, Elixir uvijek pokušava izjednačiti izraz na desnoj strani. U tom procesu, varijabla je vezana za izraz s kojim se pokušava izjednačiti

Nekada nismo zainteresirani za vrijednost na desnoj strani, ali i dalje trebamo izjednačiti izraz. Primjer je kada želimo dobiti trenutno vrijeme dana, a nismo zainteresirani za datum

```
iex(2)> {_, time} = :calendar.local_time()
{{2022, 7, 8}, {18, 33, 49}}
iex(3)> time
{18, 33, 49}
iex(4)>
```

U ovom slučaju, umjesto prve varijable, smo deklarirali anonimnu varijablu koja izjednačava izraz, ali ne veže vrijednost za sebe. Uz to možemo dati deskriptivnu vrijednost varijabli nakon _ znaka.

```
iex(4)> {_date, time} = :calendar.local_time(
...(4)> )
{{2022, 7, 8}, {18, 36, 2}}
iex(5)> date
** (CompileError) iex:5: undefined function date/0 (there is no such
import)
iex(5)> time
{18, 36, 2}
```

U nekim slučajevima trebamo provjeriti vrijednost varijable što možemo s pin operatorom ^ [1]

```
iex(1)> name = "Mislav"
"Mislav"
```

```
iex(2)> {^name, _} = {"Mislav", 24}
{"Mislav", 24}
iex(5)> {^name, _} = {"s", 24}
** (MatchError) no match of right hand side value: {"s", 24}
```

1.1.5.5. Matching lists

Matching listi radi na sličan princip kao matchanje tupli:

```
iex(1)> [first, second, third] = [1, 2, 3]
[1, 2, 3]
```

Matchanje listi je se često oslanja na njihovu rekurzivnu prirodu. Pomoću pattern matchinga možemo staviti dva elementa u različite vrijednosti: [1]

```
iex(2)> [head | tail] = [1, 2, 3]
[1, 2, 3]
iex(3)> head
1
iex(4)> tail
[2, 3]
```

1.1.5.6. Matching maps

Za matchanje mapi koristimo sljedeću sintaksu:

```
iex(1)> %{name: name, age: age} = %{name: "Mislav", age: 24}
%{age: 24, name: "Mislav"}
iex(2)> name
"Mislav"
iex(3)> age
24
```

Mape se koriste kako bi reprezentirali strukturirane podatke. U tim slučajevima možemo bit zainteresirani za samo par polja. [1]

1.1.5.7. Matching with functions

Matchanje na bazi uzoraka se koristi u specifikaciji funkcijskih argumenata. U osnovnoj funkciji:

```
def func(arg1, arg2) do
      ...
end
```

`arg1` i `arg2` su patterni i možemo koristiti standardne mehanizme matchanja. Pogledamo primjer ovoga:

```
defmodule Rectangle do
      def area({a, b}) do
```

```
            a * b
      end
end
```

Funkcija `Rectangle.area/1` očekuje da je njezin argument tuple veličine dva elementa. Nakon toga veže odgovarajuće elemente tuplea u varijable i vraća rezultat funkcije. [1]

1.1.5.8. Funkcije s više klauzula

Klauzula je definicija u funkciji koja je specificirana s `def` konstruktom. Ako imamo više definicija iste funkcije sa istim brojem argumenata, ta funkcija ima više klauzula.

```
defmodule Geometry do
      def area({:rectangle, a, b}) do
            a * b
      end

      def area({:square, a}) do
            a * a
      end
end
```

Kao što vidimo, imam dvije klauzule iste funkcije. Ovisno koje argumente joj pružimo, dobit ćemo drugačije rezultate. Kada pozove funkciju, runtime prolazi kroz sve funkcije koje su definirane redom i izvršava prvu funkciju za koju odgovaraju svi argumenti. U slučaju da ne može pronaći funkciju, vrati nam grešku. [1]

Sa gledišta pozivatelja, postoji samo jedna funkcija. Time što funkcija ima više klauzula ne mijenja činjenicu da je to samo jedna funkcija. [1]

Možemo dodati klauzulu kako bi pokrili slučaj kada nisu pruženi odgovarajući argumenti.

```
defmodule Geometry do
      def area({:rectangle, a, b}) do
            a * b
      end

      def area({:square, a}) do
            a * a
      end

      def area(unknown) do
            {:error, {:unknown_shape, unknown}}
```

```
    end
end
```

`area(unknown)` klauzula radi samo za `area/1`. Kako su funkcije `area/1` i `area/2` različite funkcije, ne možemo definirati klauzulu koja bi pokrila slučaj kada predamo više od jednog argumenta. [1]

1.1.5.9. Guards

Guards su ekstenzija osnovnog mehanizma izjednačavanja. Dopuštaju šira definiranja očekivanja koja moraju biti zadovoljena kako bi cijeli pattern bio izjednačen. Guard se može specificirat da pridodamo `when` klauzu nakon liste argumenata. [1]

```
defmodule TestNum do
    def test(x) when x < 0 do
        :negative
    end
    def test(0), do: :zero
    def test(x) when x > 0 do
        :positive
    end
end
```

Guard je logički izraz koji postavlja dodatne uvjete na klauzu. Prva klauzula će bit pozvana jedino ako joj damo negativan broj i tako dalje. Iznenađujuće, ako funkciji ne damo broj, izvršiti će se klauzula kao da je funkcija dobila pozitivan broj. Odgovor je u tome da u se u Elixiru izrazi mogu uspoređivati operatorima < i > čak i kada nisu istog tipa. U ovom slučaju, type ordering određuje rezultata [1]

```
number < atom < reference < fun < port < pid < tuple < map < list <
bitstring(binary)
```

Ovo možemo popraviti tako da provjerimo tip podataka u funkciji

```
…
def test(x) when is_number(x) and x > 0
…
```

U ovom slučaju će funkcija vratiti grešku ako joj damo argument koji nije broj.

Set operatora i funkcija koje možemo pozvati u sklopu guarda su vrlo limitirane. Ne možemo zvati naše funkcije i većina funkcija neće raditi. Ovo su primjeri operatora i funkcija koje možemo zvati:

- Operatori uspoređivanja
- Boolean operator

- Aritmetički operatori
- `in` i `not in` operatori dok je desna strana lista ili range
- Funkcije provjere tipa
- Funkcije koje rade na ugrađenim tipovima podataka [6]

1.1.5.10. Višeklauzalne lambe

Anonimne funkcije (lambde) mogu sadržavati više klauzula.

```
test_num = fn
x when is_number(x) and x < 0 -> :negative
0 -> :zero
x when is_number(x) and x > 0 -> :positive
end

IO.puts(test_num.(32))
```

U slučaju lambdi, klauzula završava kada sljedeća počinje ili kada je definicija lambde završena s `end` [1]

1.1.5.11. Kondicionali

Elixir pruža standardan načine kondicionala sa konstruktima kao `if` i `case` [1]

1.1.5.12. Klasični konstrukti grananja

Funkcije s više klauzula nisu rješenja svaki problem. Korištenjem njih zahtjeva kreiranje posebne funkcije i davanje argumenata. U slučaju kada nemamo potrebe za tim ili nam to dodatno zakomplicira kod, možemo koristit klasične konstrukte grananja, kao `if, unless, cond i case`.

- `if` i `unless`:

```
if condition do
      …
else
      …
end
```

`if` funkcionira kao i u drugim jezicima. Ovisno o uvjetu, izvršiti će se jedna ili druga grana. Možemo skratiti deklariranje `if`

```
if condition, do something else: do another thing
```

`unless` makronaredba je zapravo invezija `if` naredbe

```
unless condition, do: something, else: another thing
```

- `cond`

```
cond do
      expression_1 ->
      …
      expression_2 ->
      …
…
end
```

`cond` makronaredba se može zamislit kao `if-else-if` pattern. Uzima listu izraza i izvršava blok prvog izraza koji je istinit. Rezultata `cond` makronaredbe je rezultata izvršenog bloka. Ako nije izvršen ni jedan blok koda, `cond` vrati grešku.

- `case`

```
case expression do
      pattern_1 ->
      …
      pattern_2 ->
      …
end
```

`case` makronaredba procjenjuje izraz i rezultat uspoređuje od dane klauzule. Prva klauzula kojoj je usporedba točna se izvršava i rezultat te klauzule je rezultat cijelog `case` izraza. [1]

1.1.5.13. With

Zadnji konstrukti koji ćemo spomenuti je `with` specijalna forma, koja je korisna kada trebamo lančano izvesti nekoliko izraza i vratiti grešku na prvi izraz koji ne možemo ispuniti.

```
def extract_user(user) do
  with {:ok, login} <- extract_user(user),
       {:ok, email} <- extract_email(user) do
         {:ok, %{login: login, email: email}}
       end
end
```

[1]

1.1.5.14. Petlje i itera cije

Petlje i iteracije funkcioniraju malo drugačije u Elixiru. Konstrukti kao `do` i `while` ne postoje. Glavni alat iteracija u Elixiru je rekurzija. [1]

1.1.5.15. Iteracija s rekurzijom i tail function calls

Rekurzija s puno poziva može uzrokovati stack overflow i srušiti program. Za te slučajeve postoji nešto što se zove tail function calls. Ako je zadnja stvar koja funkcija radi zvanje druge funkcije, pozivanje funkcije ne rezultira guranjem na stack, nego se izvrši nešto slično `goto` akciji. [1]

```
defmodule Summing do
  def sum(list) do
    do_sum(0, list)
  end

  defp do_sum(cur_sum, []) do
    cur_sum
  end

  defp do_sum(cur_sum, [head | tail]) do
    new_sum = cur_sum + head
    do_sum(new_sum, tail)
  end
end

IO.puts(Summing.sum([1, 2, 3, 4]))
```

1.2. Docker i kontejneri

Kontejner je jedinica programa koja pakira kod i sve pakete koji su mi potrebni za rad kako bi aplikacija radila brzo i pouzdano na bilo kojem okruženju. [7]. Kontejneri pružaju način da standardiziramo postavljanje na serversko okruženje. Kontejneri u sebi sadrže cijeli operacijsku sustav, uglavnom minimaliziran samo na funkcionalnosti koje su potrebne za odvijanje tog programa (dobar primjer je Alpine Linux koji se često koristi zbog svoje male veličine) kako bi se reducirali prostor potreban za izvedbu i serverske troškove. Kontejneri nam omogućavaju da pokrećemo aplikacije neovisno o serverskoj konfiguraciji i da uvijek imamo isto okruženje ne ovisno o poslužitelju.

Docker je 2013. napravio velike promjene na tržištu. Docker je imao kritičnu ulogu u adaptiranu kontejnera mainstream. Omogućio je da kontejneri budu dostupni kroz open source alate i Docker slike koje se mogu koristiti u više navrata. Docker je omogućio da programeri pišu programe i pokreću lokalno ili na poslužitelju bez da brinu o tome kako je poslužitelj konfiguriran. [7]

Docker funkcionira na principu Dockerfile-ova. Dockerfile je tekstualan dokument koji sadrži sve komande koje bi korisnik pozvao kako bi pokrenuo aplikaciju. [8] Dockerfile bi trebao sadržavati sve komande kako bi se osiguralo dohvaćanje pravih paketa, pripremanje okruženja i pokretanje aplikacije.

```
FROM elixir:1.13.4

RUN mkdir orders
COPY . /orders
WORKDIR /orders

RUN mix local.hex --force
RUN mix local.rebar --force
RUN mix deps.clean snappyer
RUN mix deps.clean crc32cer
RUN mix deps.get --force
RUN mix do compile

CMD ["/orders/entrypoint.sh"]
```

Dockerfile uvijek počinje sa deklaracijom koju sliku koristi kao bazu za daljnju izgradnju, popraćeno s ostatkom komadni koje će izvršiti.

Docker Compose je dodatan alat za orkestraciju više Docker kontejnera. Kroz Compose definiramo određene postavke za kontejnere pod istu Docker mrežu kako bi komunicirali u funkcionirali kao cjelina. Docker Compose se konfigurira sa `.yaml` dokumentima u sljedećem formatu:

```
version: "3.9"

services:
  frontend:
    container_name: product-frontend
    build: client
    ports:
      - 6060:80
  products:
    container_name: products-service
    build: services/main
    environment:
      - PGUSER=postgres
      - PGPASSWORD=root
```

```
- PGDATABASE=products
- PGHOST=product-database
- PGPORT=5432
```

Docker compose počinje uglavnom s deklaracijom verzije. Dok je ova deklaracija prije bila obavezna, ali to je opcionalno od v1.27.0. U slučaju da `version` nije dodan, Docker Compose će koristiti zadnju verziju Docker Compose. [9]

1.3. Kafka i event streaming

Event streaming je proces prenošenja promjena u podacima kroz sustav. [11] Okidači tih promjena su događaji, od toga dolazi ime „Event“. Event streaming dohvaća podatke iz izvora kao base podataka, senzori i mnoge druge. Ključni element je da se promjene dohvaćaju čim se okinu. Ovaj proces osigurava kontinuirani tok podataka da je prava informacija na pravom mjestu u pravo vrijeme. [10]

Kafka se prvo pojavila u srpnju 2011. godine kada je postala dio Apache inkubator projekta. Prva verzija je puštena van u siječnju 2012. godine. [12]

Kafka je distribuirani sustav poslužitelja i klijenata koji komuniciraju preko TCP mrežnog protokola. Na poslužiteljima su pokrenuti takozvani brokeri, koji formiraju podatkovni sloj. Uz to, pokrenut je Kafka Connect API koji kontinuirano importa i exporta podatke kao event streaming. Klijenti su naši servisi koje vrše interakciju sa Kafkom. Servisi čitaju ili pišu podatke u Kafka topice. [10]

Produceri su servisi koji produciraju event u Kafku i tako šalju podatke, dok consumeri slušaju evente te čitaju i procesiraju podatke [10]

Produceri i consumeri su kompletno neovisni jedno o drugom što dovodi do visoke skalabilnosti. Kafka kao cijela je iznimno konfigurabilna, tako da se može prilagoditi bilo kojem slučaju kojem nam treba. [10]

1.4. NGINX

NGINX je brzi, efikasni i skalabilni web server, reverse proxy i akcelerator. Prva verzija NGINX je kreirana 2002. godine, a 2004. godine je projekt postao javno dostupan. NGINX danas je de-facto standard za brze i skalabilne stranice. [13]

NGINX koristi asinkron pristup baziran na događajima gdje su zahtjevi obrađeni na jednoj dretvi. NGINX ako dio svoj paketa nudi reverse proxy, IPv6, load balancing, websocket, TLS/SSL, posluživanje statičnih datoteka kao HTML datoteke i može služiti kao gateway.

NGINX, uz Apache, je među najpopularnijim web serverima, gdje stoji na 2. mjestu nakon Apache. (https://kinsta.com/knowledgebase/what-is-nginx/)

U našem primjeru, NGINX koristimo za dvije stvari. Prvo, NGINX će na klijentskoj strani služiti kao web server koji poslužuje statične datoteke koje nam je dao React.

Drugo, NGINX služi kao API gateway između našeg klijenta i servisa. API Gateway je web server koji osigurava i posreduje promet između svih naših servisa i klijenta. API Gateway autentificira naše zahtjeve, proslijedi zahtjeve na prave servise i brine se o greškama. [14]

1.5. PostgreSQL

PostgreSQL je open source sustav relacijske baze koji koristi SQL jezik kako bi sigurno spremao i skalirao kompleksne podatke. Počeci PostgreSQL-a se mogu pratiti sve do 1986. godine kao dio POSTGRES projekta na University of California at Berkeley. [15]

PostgreSQL je na dobro glasu zbog svoje dokazane arhitekture, pouzdanosti, integriteta podataka i proširivosti. Od 2001. godine provodi ACID na svojoj bazi i tako osigurava pouzdanost koji nam pruža. PostgreSQL se primarno koristi kao skladište za web, mobilne, geospacijalne i analitičke aplikacije [16]. Zadnja verzija PostgreSQL – a je 14.5 [17]

1.6. React

React je JavaScript paket koji se koristi za izgradnju korisničkih sučelja. [18] Core React tim je Meta open source tim koji se brine o održavaju i razvoju paketa. React je daleko najpopularniji paket za razvoj korisničkih sučelja, gdje je prema Stack Overflow anketu za 2022 godinu bio na prvom mjestu sa čak 42.62% tržišta. [19] React je započeo svoje dane kao rani prototip zvan FaxJS gdje je 2011. godine kreiran za interne potrebe Facebook-a. React 2013. godine službeno postaje open source projekt. [20]

2018. godine sa verzijom v16.7.0 React dobiva jedan od značajnijih ažuriranja u njegovoj povijest. Sa 16.7.0, u fazi predlaganja, predstavljeni su takozvani funkcionalni komponenti i React Hooks. [21] Funkcionalni komponenti odmiču React kao ekosustav od klasnog načina pitanja i predstavljaju koncepte kao immutability i pure functions. Dok React ne implementira ove koncepte do kraja, generalno dobar React kod će pokušavati poštovati ove koncepte koliko god može. Danas, React sa vezijom 18 je kompletno usvojio funkcionalne komponente i funkcionalne komponente de-facto način za pisanje React koda

Mikroservisi

Mikroservisi kao arhitektura u zadnjih 10 godina postaju iznimno popularne. Sve je više kompanija kao Netlix, Uber i Amazon koji su usvojili do neke mjere mikroservisne arhitekture. Mikroservisne su se pokazale kao iznimno kompetentan i kvalitetan tip strukture sustava koji, kao i svi ostali stilovi, donose svoje prednosti i mane.

1.7. Povijest mikroservisa

Podrijetlo mikroservisa dolazi iz napora da se pruži bolja komunikacija i interoperabilnost sustava. Mikroservisi kao struktura su uglavnom opisani kao neovisni servisi koji međusobno surađuju i komuniciraju kako bi stvorili strukturirani sustav. Dr. Peter Rodgers je 2005. godine iskoristio pojam „Micro-Web-Services“ tijekom prezentacije o računalima u oblaku. [22] Izraz „microservices“ se prvi put pojavio na konferenciji za sistemske arhitekte 2011. godine [23]

Uvođenje Simple Object Access Protocol (SOAP) je bio ključan korak u razvoju mikroservisa. [22]. SOAP se prvi put pojavio 1998. godine od strane Microsofta, gdje su Bob Atkinson, Don Box, Dave Winer i Moshen Al-Ghosein dizajnirali i pustili sustav u opticaj. SOAP je bio ključan korak zato što su servisi dobili mogućnost razmjene bogati i strukturirani informacija putem HTTP-a koristeći XML dokumente. SOAP je zapravo standard koji je omogućio programerima lakšu razmjenu kroz to da ne moraju definirati standard razmjene za svaki servis. [24]

1.8. Mikroservisi – što su

Mikroservisi su tip serversko orijentirane arhitekture. Uglavnom su mikroservisi orijentirani na poslovnu domenu, s pristupom da jedan mikroservis odgovara jednom poslovnoj domeni. Svaki mikroservis ima otvoren jedna ili više API-a gdje drugi servisi ili krajnji korisnici pristupaju funkcionalnostima. Zbog same prirode mikroservisa, generalno svaki mikroservis radi sa zasebnom bazom, ne bi se trebala događati situacija gdje dva servisa vrše operacije nad istom bazom. Ključni koncepti mikroservisa uključuju:

1. Nezavisnost

Sve promjene koje se vrše nad servisom ne ovise o drugim servisima. Svaka promjena se nezavisno postavlja na produkciju bez da ih moramo primijeniti na druge servise. Kako bi

osigurali nezavisnost, servisi moraju biti što manje ovisini jednu o drugima. Svaki servis mora postojati kao jedinica za sebe, neovisan o bilo kojem drugom servisu

2. Modeliranje poslovne domene

Tehnike kao „Domain-Driven Design“ nam omogućuju da strukturiramo servis kako bi što preciznije imitirali domenu u kojoj je sustav. Kod mikroservisa, koristimo poslovnu domenu kao granice pojedinačnih sustava.

3. Posjedovanje podataka

Kao što je navedeno prije, svakoj bazi bi trebalo imati pristup samo jedan servis. Mogli bi reći da je servis sučelje za tu bazu. Ako servis treba podatke iz baze za koju nije odgovoran, trebao bi pitati servis koji je odgovoran za bazu gdje se ti podaci nalaze.

4. Veličina

Često se pojavljuje koliko veliki bi servis trebao biti. Odgovor na ovo pitanje najviše ovisi o kontekstu i samom zadatku servisa. Jednostavan, a u isto vrijeme i kompleksan je odgovor da bi servisi trebali biti do veličine do koje ih je lako razumjeti. Ako imamo problem s razumijevanjem servisa u smislu koja mu je zadaća, postoji mogućnost da smo natrpali previše funkcionalnosti u taj servis i da bi bila dobra ideja razbiti taj servis na više manjih servisa.

5. Fleksibilnost

Fleksibilnost se odnosi na stavku da bi trebali ostaviti opcije otvorene da dodamo nove funkcionalnosti servisu koje možda nismo ni zamislili sada. Kako se poslovna domena mijenja tako bi se trebala mijenjati i naša mikroservisna arhitektura

6. Usklađivanje arhitekture i organizacije

U standardnim monolitima imamo podjele na „frontend“ i „backend“ developere. Dok ova podjela ima šanse funkcionirati u timovima koji su orijentirani na mikroservisne arhitekture, podjela gdje timovi posjeduju servise ima mnogo više smisla. Timovi posjeduju servise, rade izmjene na strani klijenta ili poslužitelja i puštaju izmjene u opticaj. Drugi timovi koji posjeduju svoje servise nisu zainteresirani kako se te izmjene unutar timova rade, samo da se one događaju te eventualno ako trebaju reagirati na te izmjene (npr. promjene na javnom API pozivu) [25]

1.9. Prednosti mikroservisa

1.9.1. Tehnološka heterogenost

Jedna od velikih prednosti mikroservisne arhitekture je tehnološka heterogenost. Za razliku od monolita, gdje su svi procesi unutar jedne aplikacije pisani istim jezikom, kod mikroservisa možemo imati više servisa svaka pisana u svojem jeziku. Na kraju dana, mikroservisi komuniciraju preko mreže servera, pa servisi s kojim komuniciraju nisu zainteresirani u kojoj tehnologiji je taj specifični servis napisan. Ovo nam omogućava da adaptiramo nove tehnologije od kojih bi mogli imati benefite, a implementacija za cijeli sustav bi bila riskantna. U slučaju da implementacija prođe bez problem i pokaže se uspješnom, možemo proširiti implementaciju na cijeli sustav. U slučaju da odlučimo da se ne isplati implementirati dalje, lagano vratimo taj jedan servis na staru tehnologiju [25]

1.9.2. Otpornost

Kod monolita, u slučaju greške nekog dijela sustava, cijelu sustav zbog same prirode pada, čime sam sustav postaje nedostupan. Manjak segmentacije procesa unutar sustava je jedna od najvećih mana monolitne arhitekture. Mikroservisi nam pružaju relativno elegantno rješenje na ovaj problem. Pažljivom segmentacijom odgovornosti i granica sustava možemo osigurati da u slučaju pada jednog servisa, ostatak sustava nesmetano poslužuje svoje funkcionalnosti. Pad jednog servisa ne mora značiti pad cijelog sustava. [25]

1.9.3. Skalabilnost

Dok inicijalni trošak mikroservisa će zasigurno biti veći od inicijalnih troškova monolita, mikroservisi kompenziraju za tu manu kod troškova skaliranja.- Rastom prometa na našim servisima morat ćemo skalirati resurse kako bi podnijeli veći broj zahtjeva. U slučaju monolita, nemamo drugu opciju nego skalirati cijeli sustav. Ovo nam može stvoriti potencijalne probleme u smislu troška. Nisu sve funkcionalnosti jednako kompleksne i ne zahtijevaju sve funkcionalnosti jednaku količinu resursa. Kod mikroservisa možemo skalirati individualne servise prema količini koja im je potrebna da bi nesmetano obavljali dužnosti. Rijetko kada će dva servisa zahtijevati istu količinu resursa kako bi uspješno obavljali svoje dužnosti. [25]

1.9.4. Kompleksnost

Sama ideja mikroservisa je razbijanje kompleksnost monolita na manje jedinice koje je lakše razumjeti. Dok je jedan veliki projekt relativno teško razumjeti u potpunosti, kada imamo

mnogo malih servisa od kojih svaki radi jasno definiranu zadaću, lakše nam je razumjeti tok podataka u sustavu i vizualizirati kako se cijeli sustav ponaša [26]

1.9.5. Vlasništvo servisa

Velika prednosti mikroservisa, pogotovo u kompanijama koje broje veliki broj programskih inženjera kao što su Amazon, Netflix i Uber, je timsko posjedovanje servisa. Jedan tim posjeduje jedan servis, te je on odgovoran za životni ciklus tog servisa, te ga ostali timovi tretiraju kao crnu kutiju s kojom komuniciraju. [25]

1.10. Mane mikroservisa

1.10.1. Iskustvo razvoja

Jedan od velikih bolnih točaka mikroservisa je sam razvoj. Ako imamo runtime koji je memorijski intenzivan, kao JVM ili Node, mogu nas limitirati koliko mikroservisa možemo pokrenuti na našem računalu prije nego što iskusimo probleme s performansama. [25]

1.10.2. Tehničko predznanje

Mikroservisi kao arhitektura zahtijevaju puno više tehničkog predznanja nego monolitne arhitekture. Dok je relativno jednostavno pokrenuti monolitni sustav, kod mikroservisa imamo puno više stvari o kojima se moram brinuti kako bi strukturirali uspješnu mikroservisnu arhitekturu. Stavke kao što će pojedini servis raditi, kako će servisi komunicirati, koje servise skalirati, što napraviti kada je jedan od servisa nedostupan. Uz sve ovo navedeno, postoji realna opasnost od toga da padnemo u zamku novih tehnologija, gdje uvođenjem jedne nove tehnologije pokrenemo lavinu uvođenja novih nedokazanih tehnologija što nam dugoročno stvori mnogo problema. [25]

1.10.3. Trošak

Kao što je navedeno prije, jedan od aspekata su povećani troškovi vezani uz mikroservise. U inicijalnom postavljanju arhitekture, trebamo zakupiti više serverskog prostora kako bi mogli pokrenuti sve servise sa dovoljno memorije za njihov nesmetan rad. Što više mikroservisa inicijalno imamo, veći su zahtjevi za prostorom i memorijom. [25]

1.10.4. Sigurnost

Općenito autentifikacija, autorizacija i zaštita samih servisa je teža kod mikroservisne arhitekture nego kod monolitnih sustava. Uz to, postoji realna šansa curenja podatka u tranzitu dok se razmjenjuju među servisima, dok kod monolita ovo nije toliko izražen problem zbog intraprocesne prirode komunikacije. [25]

1.10.5. Latencija

Mikroservisi dodaju latenciju koja prije nije bila prisutna kod monolita kroz to da se procesiranja koja su prije bila u jednom procesu rascjepkaju na više servisa. Pod latenciju u ovom slučaju spadaju vrijeme dok se izvrše zahtjevi preko mreže, sama serilizacija i deserilizacija podataka kako bi bili dostavljeni između servisa, što prije nije bio faktor o kojem smo se morali brinuti. [25]

1.10.6. Konzistentnost podataka

Pomak ka mikroservisnom sistemu dolazimo do problema spremanja kontinuiteta podataka. Dok smo se u prošlosti kod monolita oslanjali na transakcije u bazi podataka da izvršavaju promjene podataka, distribuirani sustavi nam ne mogu pružiti sigurnost kod promjene podataka. Moramo osigurati da cjelokupni sustav ima razvijene metode kako osigurati da podaci koje zapisujemo i čitamo iz baze budu konzistentni. [25]

1.11. Komunikacija između mikroservisa

Komunikacije je vitala dio mikroservisne arhitekture. U puno slučajeva sam odabir komunikacije, bilo ona sinkrona ili asinkrona, će uvelike utjecati na sami dizajn aplikacije i to podatka.

Sama priroda komunikacije se mjenja uz intra-procesne u inter-procesnu. Servisi komuniciraju preko mreže umjesto poziva unutar isto procesa. Najveći utjecaj kada prelazimo sa intra-procesne na inter-procesnu komunikaciju je vidljiv u performansama. Inter-procesni pozivi su temeljno drugačiji. Dok vršimo intra-procesni poziv, sustav može provesti razne optimizacije kako bi ubrzao poziv. Kod inter-procesnog poziva limitirani smo raznim faktorima kao brzina same mreže, količina prometa na servisu i brzina samog servisa.

Tehnologije koje možemo odabrati za inter-procesnu komunikaciju su mnoge. Od standardnih HTTP poziva, korištenje REST-a, RPC poziva do message queue sistema kao RabbitMQ ili event streaminga kao Kafka, odabir je obilan. Ali nisu sve tehnologije napravljene

za sve situacije. Ako trebamo običnu request – response funkcionalnost, stvari kao Kafka su vjerojatno loš odabir za to. [25]

1.11.1. Sinkrona komunikacija

Sinkrona komunikacija je po prirodi blokirajuća. Radi na principu request – response, gdje po slanju zahtjeva nekom servisu, blokira daljnju izvedbu procesa dok ne dobije odgovor od servisa gdje je poslan zahtjev.

Prednosti ovog načina komunikacije su jednostavnost i predvidljivost. Ovi zahtjevi su relativno jednostavno, znamo što se događa kada ih izvršimo i znamo što će se dogoditi kada dobijemo odgovor nazad

Najveća mana ovog pristupa je vremena ovisnost o drugim servisima na koje vršimo poziv. Kada se poziv krene izvršavati, moramo čekati dok ne dobijemo odgovor. Što se događa ako je servis na koji radimo poziv zatrpan pozivima i performanse degradiraju. Što ako servis nije dostupan u trenutku kada ga pozivamo. Za sve ove slučajeve moramo na servisu koji šalje poziv pokriti scenarije i imati dovoljno detaljan sustav bilježenja grešaka ako dođe do istih.

Primjer gdje se koriste ovakvi pozivi su SQL upiti u bazi ili dohvaćanje podataka od drugog servisa. Klasičan primjer ovakve komunikacije su HTTP pozivi na drugi servis [25]

1.11.2. Asinkrona komunikacija

Asinkrona komunikacija po prirodi nije blokirajuća, točnije nakon slanja poziva servis koji je poslao poziv nastavlja s daljnjom provedbom svoje funkcije. Postoje razne vrste asinkrone komunikacije, kao komunikacija kroz podatke, event driven interaction i vrsta request – response zahtjeva gdje servis ne ovisi o odgovoru

Prednosti asinkrone komunikacije je vremena neovisnost o drugom servisu. Dok inicijalni servis napravi poziv slobodan je nastaviti daljnji izvedbu funkcije dok drugi servis procesira podatke. Servisi koji primaju pozive ne moraju uopće biti dostupni tijekom inicijalnog slanja poziva, nego mogu kasnije pročitati poruku koja je došla pozivom. Primjer ovoga je Kafka event streaming, gdje jedan servis piše poruke u Kafka topice, gdje te poruke stoje dok ih netko ne pročita, u ovom slučaju drugi servis. Uz to, asinkrona komunikacija se može koristit za pozive koji bi mogli trajati dugo vremena.

Mane asinkrone komunikacije je kompleksnost i mogućnost odabira. Dok je relativno jednostavno postaviti jednostavnu asinkronu komunikaciju, čim trebamo skalirati komunikaciju možemo naići na probleme sa skaliranjem komunikacije. Uz to, mnoge opcije tehnologije za provedbu asinkrone komunikacije može biti teško odabrati pravu za naš slučaj. [25]

1.12. Transakcije i baze podataka

Generalno, pod transakcije smatramo jednu ili više akcija koje želimo tretirati kao jednu jedinicu. Kada vršimo više promjena kao dio jedne operacije, želimo potvrditi da su sve operacije izvršene uspješno. Sa bazom podataka, transakcije koristimo kao sredstvo kontrole da su jedna ili više promjena uspješno provedene. [25]

U mikroservisnim arhitekturama, generalno postoje dva pristupa organiziranja baza podataka: baza podataka po servisu i dijeljena baza podatka. [27] Ovaj rad se bazi podjelom jedne baze po jednom servisu, zato što se korištenje jedne baze za sve servise generalno smatra anti-patternom. [28] Kod korištenja jedne baze za cijeli sustav se pojavljuje mnoštvo problema u kojima bi istaknuo race condition, gdje dvije ili više transakcija na bazama podataka pišu ili čitaju iste podatke, što može dovesti do gubitka podataka, čitanja krivih podataka i manja konzistentnosti podataka. [29]

1.12.1. ACID

ACID je akronim za atomicity, consistency, isolation, i durability, što su svojstva transakcije baze podataka koja osiguravaju integritet podataka neovisno o greškama. [30] ACID kao koncept nije podržan od strane svih baza podataka. Većina novijih relacijskih baza podataka implementira ACID kao dio svoje strukture, međutim to nije slučaj kod nerelacijskih baza podataka. MongoDB, jedna od najpopularnijih nerelacijskih baza podataka je dugo podržavala ACID na promjenama za samo jedan dokument. [25]

Najpopularniji način kako baze podataka implementiraju ACID je kroz blokiranje podataka. Podaci su blokirani, točnije ni jedna druga transakcija im ne može pristupiti, sve dok prijašnja transakcija ne završi, neovisno o finalnom stanju transakcije. [25]

Negativna strana ACIDa je da sustavi koji implementiraju ACID su generalno spori kod čitanja i pisanja upravo zbog blokiranja podataka. [25]

1.12.2. Sage

Saga je algoritam koji koordinira promjene u stanju ali izbjegava blokiranje resursa na dugo vrijeme. Glavna ideja je bila smisliti najbolji način kako provesti operacije bolje znane kao *long lived transactions* (LLT). Hector Garcia-Molina i Kenneth Salem u svojem radu „Sagas“ dolaze do zaključka kako ne bi blokirali cijele tablice, LTT bi trebalo razbiti na sekvence transakcija, gdje je cilj da transakcija traje što kraće. [31] U mikroservisima sage imaju ulogu kontrolera. Sage izvršavaju transakcije kroz više sustava i kontroliraju sam tok transakcija. U slučaju da jedna od transakcija bude neuspješna, zadataka saga je vratiti stanje baza

podataka kako je bilo prije početka saga, točnije mora poništiti sve transakcije koje su bile provedene do trenutka gdje se pojavila greška. [25]

1.12.3. Reference na druge baze podataka

Jedan problem kada razbijemo jednu bazu na više baza podataka je referenciranje podataka u drugoj bazi. Postoje mnogi načini kako riješiti ovaj problem, ali u našem primjeru je odabrano zapisivanje primarnog ključa podatka za tablicu koja se referencira.

1.13. Autentifikacija i autorizacija

Ključni koncept svakog sustava koji provodi interakciju s korisnicima su autorizacija i autentifikacija.

Autentifikacija se odnosi na proces dokazivanja istinitosti činjenice ili dokumenta. [32] U kontekstu mikroservisnih arhitektura, autentifikacija se odnosi na proces utvrđivanja identiteta korisnika koji trenutačno koristi našu aplikaciju. Autentifikacija se najčešće vrši putem kombinacije korisničkog imena i lozinke, dok postoje autentifikacije kroz druge servisa kao Google ili GitHub koristeći SSO. Kao dodatak autentifikaciji postoje razni načini dodatne autentifikacije, često zvani multi factor authentification (MFA) koji služe kao dodatna linija obrane protiv krađe identiteta i korisničkih računa. MFA su najčešće implementirani putem kodova koji dolaze na e-mail adresu ili mobilni broj korisnika ili authenticator mobilne aplikacije koje generiraju set kodova za prijavu na aplikacije. Naš primjer mikroservisa će koristiti autentifikaciju korisnika putem korisničkog imena ili lozinke ili prijavom putem servisa kao Google kroz Auth0 servis.

Autorizacija je proces davanja ovlasti korisniku da pristupi određenom resursu. [33] Preduvjet je da je korisnik već autentificiran i prijavljen u sustav. Kroz sustav autorizacije možemo upravljati korisnicima i koliko pristupa sustavu ima određen korisnik. To se odnosi na razne dijelove sustava i resursa.

Autentifikacija i autorizacija su koncepti koju idu jedno uz drugo. Bez autorizacije nema efektivne autentifikacije te autorizacija ne može postojati bez autentifikacije.

U mikroservisima imamo dva tipa autentifikacije: autentifikacija između servisa i autentifikacija korisnika [25]

1.13.1. Autentifikacija između servisa

Jedan od najjednostavnijih načina autentifikacije, koji ćemo koristiti u našem primjeru, je generiranje API ključeva koji se šalju prilikom svakog zahtjeva. Ovi ključevi se najčešće provjeravaju tijekom komunikacije kroz API gateway, kao nginx. [25] Mana ovog pristupa je što su ključevi fizički zapisani, pa postoji realna šansa da ključevi procure što bi omogućilo napadačima pristup cijelom sustavu. [34]

Još jedan način koji je relativno popularan je korištenje JSON Web Token-a za autentifikaciju servisa. Funkcioniraju tako da se generira token koji sadrži podatke potrebne za autentifikaciju. Primjer tokena možemo vidjeti na https://jwt.io/. Tokeni su napravljeni tako da budu kratkoročni (uglavnom do 30 minuta) tako da u slučaju da procure negdje, sami tokeni nakon određenog perioda postanu beskorisni. [34] JWT tokeni su naišli na mnogo kritika od programerske zajednice pod premisom da nisu dovoljno sigurni

1.13.2. Autentifikacija korisnika

Kao što je nadvedeno prije, autentifikacija korisnika se uglavnom vrši kroz kombinaciju korisničkog imena i lozinke. U zadnje vrijeme je česta pojava Multi factor authentification (MFA), pogotovo u okruženjima gdje korisnici imaju pristup osjetljivim podacima, kao recimo državne institucije ili velike kompanije. Sve češće su prijave putem biometrije, gdje otisak prsta ili prepoznavanje lica služi kao vrsta autentifikacije. [25]

Single Sign on (SSO) je metoda autentifikacije korisnika kroz druge eksterne servise, kao Google. Funkcionira na način da aplikacija komunicira s centralnom domenom koja je odgovorna za autentifikaciju korisnika. Centralna domena vrati sesiju ili ključ putem kojeg aplikacija zna da je identitet korisnika potvrđen i korisnik je prijavljen. [35]

1.13.3. Confused deputy problem i rješenja

Confused deputy problem se javlja kada jedan od servisa niže u lancu radi stvari za koje nije dizajniran da radi. Recimo da imamo dva servisa: order i products. Kada je korisnik prijavljen, može vidjeti sve svoje narudžbe. Dok korisnik dohvaća svoje narudžbe, orders servis kontaktira products servis da vidi stanje proizvoda koje je korisnik naručio. Ali što sprječava korisnika da koristeći program kao Postman, dohvati svoj token i napravi zahtjev tako da promjeni parametre korisnika da dohvati narudžbe drugog korisnika. Ovo je klasičan slučaj gdje pružamo autentifikaciju, ali ne pružamo dovoljnu autorizaciju korisnika. [25]

Ovaj problem se može riješiti centraliziranje i decentraliziranjem autorizacije. Možemo provesti svu potrebnu autorizaciju čim je prvi zahtjev pokrenut. Na ovaj način bi centralizirani svu autorizaciju korisnika. Međutim ovaj pristup se kosi s načelima mikroservisa i generalno

nije preferirani način rješavanja ovog problema. Mikroservisi postaju ovisni o jednom servisu, servisu koji je zadužen za autorizaciju korisnika. [25]

Decentralizirani pristup prebacuje odgovornost autorizacije na servis koji obrađuje zahtjev. Svaki servis radi promjenu tokena i sukladno tome nastavlja daljinu obradu zahtjeva. Ali to dolazimo do još jednog problema. Kako sigurno prenijeti informacije o korisniku. Uvijek možemo staviti informacije o korisniku u zaglavlje zahtjeva, međutim ako ne kriptiramo te podatke, korisnici mogu vrlo lagano promijeniti podatke i imitirati druge korisnike. U spas nam dolaze kriptirano tokeni. Pogledat ćemo primjer JWT tokena, koje smo spomenuli prije. [25]

1.13.4. JSON web token

Prvi draft JSON web tokena je dovršen ne tako davne 2011. godine od strane John Bradley-a [36]. JWT je otvoren standard (RFC 7519) koji predstavlja kompaktan način za sigurno prenošenje informacija između dvije strane. JSON web tokeni mogu biti potpisani privatnim ključem, tako da uz kombinaciju s javnim ključem osiguravaju da je poruka došla od te osobe te da nije bila modificirana u tranzitu. JSON web tokeni se smatraju kratkoročnim i jednokratnim tokenima, tako da bi trebali imati relativno kratki rok trajanja upravo zbog sigurnosti. [37]

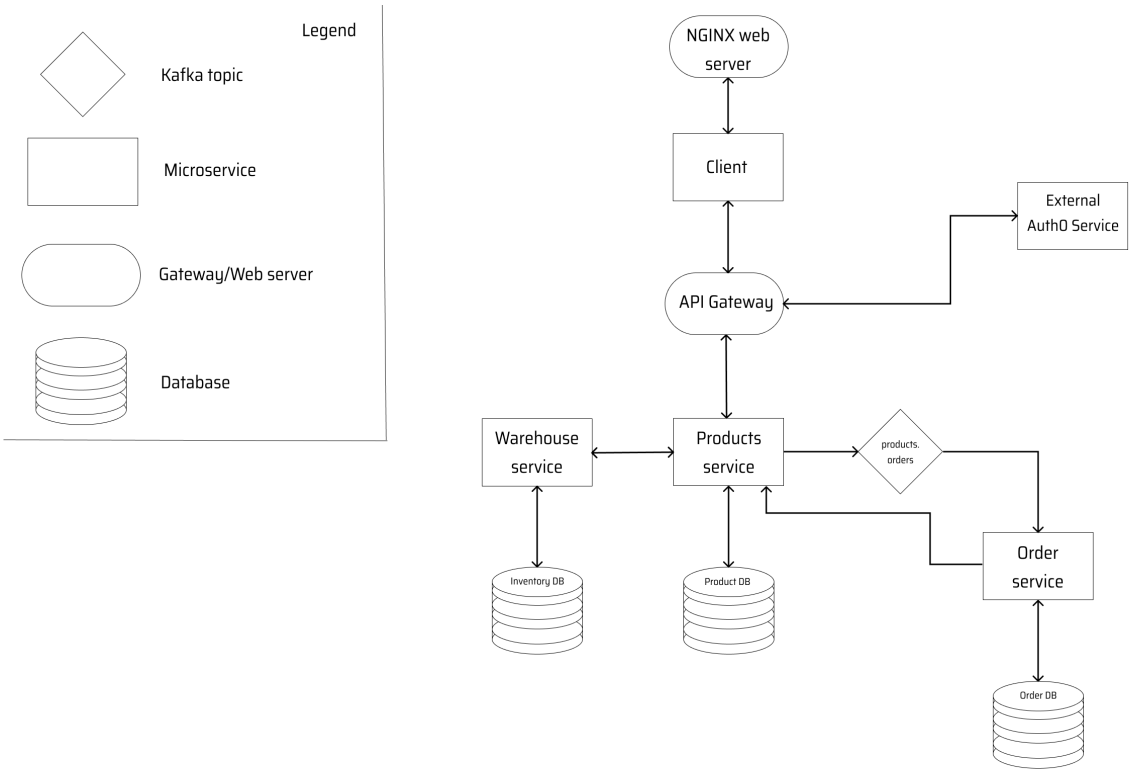
Format JSON web tokena je JavaScript Oriented Notation (JSON) struktura, što znači da zbog svojstva JSON-a JWT može sadržavati što god. JWT u svojoj dokumentaciji navodi nekoliko polja koja bi bilo dobro koristiti ako želimo da su naši tokeni kompatibilni sa paketima za rad s tokenima, ali ovo nije standard koji je strogo primijenjen upravo zbog prirode JWT-a. Kao što smo rekli, JWT ima JSON tijelo zahtjeva koje sadrži podatke

```
{
      „name“ : „Mislav“,
      „exp“: 12345789
}
```

JWT kao sam je običan string te ima strukturu podijeljena na tri dijela odvojena točkom `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiTWlzbGF2IiwiZXhwIjoxMjM0NTc4OX0.4n8ljDElmOoDzJe0acR9Zn66x_TI2DwH3nJG8DhDMRg` koji poprima ovu strukturu: `header.payload.signature`

Struktura aplikacije

Kako bi prikazali sve prijašnje spomenute tehnologije i arhitekturu, kao primjer imamo jednu relativno jednostavnu aplikaciju koja je napravljena u stilu mikroservisne strukture. Aplikacija je web shop koji se sastoji od 6 modula



Slika 1. Dijagram arhitekture aplikacije

Klijentski dio napisan u Typescriptu i Reactu koji služi kako bi korisnici mogli pregledavati proizvode i stavljati narudžbe. React aplikacija se gradi preko Docker slike i datoteke koje se dobivaju od Reacta se poslužuju preko NGINX web servera.

```
server {
  listen 80;

  location / {
    root   /usr/share/nginx/html;
    index  index.html index.htm;
    try_files $uri $uri/ /index.html;
  }
```

```
  error_page   500 502 503 504  /50x.html;

  location = /50x.html {
    root   /usr/share/nginx/html;
  }
}
```

Docker slika za klijentski dio uključuje multi-step build kako bi uspješno poslužili datoteke od React-a

```
FROM node:latest as build

WORKDIR /client

ENV PATH /client/node_modules/.bin:$PATH
COPY package.json ./
COPY package-lock.json ./
RUN npm ci --silent
COPY . ./
RUN npm run build

FROM nginx:1.15
COPY --from=build /client/dist /usr/share/nginx/html
COPY nginx/nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

API gateway je kofiguriran tako da sluša zahtjeve na portu 80. Svaki mikroservis ima definiran svoj `upstream` koji usmjerava zahtjev na određeni Docker kontejner

```
upstream product {
    server products-service:4000;
}
```

Ovisno o ruti na koji dolazi zahtjev, API Gateway je zadužen da prosljedi zahtjev na određeni `upstream`

```
    location / {
        proxy_pass http://product-frontend:80;
    }
    location ~/api/products/(.*)$ {
        auth_request /_validate_apikey;
        proxy_pass http://product/api/v1/products/$1;
    }
    location ~/api/warehouse/(.*)$ {
```

```
        auth_request /_validate_apikey;
        proxy_pass http://warehouse/api/v1/warehouse/$1;
    }
    location ~/api/orders/(.*)$ {
        auth_request /_validate_apikey;
        proxy_pass http://orders/api/v1/orders/$1;
    }
```

Ovom konfiguracijom svaki zahtjev koji dođe na rutu `/` će biti proslijeđen na naš kontejner koji poslužuje klijentski dio aplikacije. Svaki od zahtjeva prije nego što je preusmjeren prolazi kroz `/_validate_apikey` rutu koja ima posao provjeriti API key za naše servise

```
    location = /_validate_apikey {
        internal;

        if ($http_apikey = "") {
            return 401; # Unauthorized
        }
        if ($api_client_name = "") {
            return 403; # Forbidden
        }

        return 204; # OK (no content)
    }
```

API ključ je postavljen u istom NGINX dokumentu za konfiguraciju, gdje u slučaju da ključ nije prisutan podstavlja se na prazan string, čime će zahtjev bit odbijen

```
map $http_apikey $api_client_name {
    default "";

    "key_thing" "client_one";
}
```

Ovaj ključ se pridodaje zahtjevu koristeći parametar u zaglavlju tijekom HTTP zahtjeva.

Inicijalno dohvaćanje korisničkih podataka i autentifikacije korisnika se vrši kroz vanjski servis Auth0. Koristeći Auth0 integriran je sustav prijave korisnika. Prijava korisnika se vrši putem Auth0 paketa koji je dostupan na Node Package Manager koji ima metode za prijavu korisnika dostupne

```
import { useAuth0 } from "@auth0/auth0-react";
..
  const {
    loginWithRedirect,
```

```
    user,
    isAuthenticated,
    logout,
    getAccessTokenSilently,
  } = useAuth0();
```

Metodom `loginWithRedirect` otvara se zasebni prozor u pregledniku gdje korisnik ima mogućnost prijave e-mailom i lozinkom ili preko Google računa.

Kada se korisni prijavi, koristeći metodu `getAccessTokenSliently` dobijem access token kako bi mogli pristupiti mikroservisima

```
  useEffect(() => {
    const getUserMetadata = async () => {
      const domain = "dev-jy4007c0.us.auth0.com";

      try {
        const accessToken = await getAccessTokenSilently({
          audience: `https://${domain}/api/v2/`,
          scope: "read:current_user",
        });

        localStorage.setItem("access_token", accessToken);
        localStorage.setItem("sub", user?.sub);
      } catch (e) {
        console.log(e.message);
      }
    };

    getUserMetadata();
  }, [getAccessTokenSilently, user]);
```

Products servis za vrijeme kreiranja narudžbe za određenog korisnika šalje zahtjev na Auth0 servise koji vraćaju e-mail adresu iz danog Access tokena

```
    case get_token(conn) do
      {:ok, token} ->
        req = Req.new(url: URI.encode("https://dev-
jy4007c0.us.auth0.com/api/v2/users/" <> sub))
        req = Req.Request.put_header(req, "Authorization", "Bearer " <>
token)

        Enum.each(params["_json"], fn item ->
          value = %{
```

```
            email: Req.get!(req).body["email"],
            product_id: item["product_id"],
            amount: item["amount"]
          }

          encoded = Jason.encode!(value)
          :brod.produce_sync(:kafka_client, "products.orders", :hash,
"order_set", encoded)
        end)

      {:error, error} ->
        handle_error_response(conn, error)
    end
```

Svi zahtjevi na servise prolaze kroz Products servis. Products servis je odgovoran za dohvaćanje proizvoda kao primaran zadatak. Prilikom kupnje proizvoda, Products servis rezervira proizvode od Warehouse servisa i šalje event u Kafka topic kako bi Orders servis zabilježio narudžbu. Prilikom dohvaćanja proizvoda od Products servisa, Products servis zatraži stanje proizvoda u bazi i prikazuje proizvode kojih ima na stanju.

```
  def reserve_stock(items) do
    Enum.each(
      Map.get(items, "_json"),
      fn item ->
        product = Stock.get_product_by_fk(item["product_id"])

        if Map.get(product, :product_id) - item["amount"] < 0 do
          raise "Product request is more than amount in stock"
        end
      end
    )

    Enum.each(
      Map.get(items, "_json"),
      fn item ->
        product = Stock.get_product_by_fk(item["product_id"])

        Product.changeset(product, %{
          amount: Map.get(product, :amount) - item["amount"]
        })
        |> Repo.update()
      end
```

```
    )
  End
```

Orders servis sluša Kafka topic i sukladno događajima u tom topicu zapisuje nove narudžbe.

```
 @impl :brod_group_subscriber_v2

  def handle_message(message, _state) do
    {_, _, _, _, message_data} = message

    Enum.each(message_data, fn msg ->
      {_, _, _, data, _, _, _} = msg
      decodedData = Jason.decode!(data)

      {:ok, order} =
        Transactions.create_order(%{
          account: decodedData["email"],
          status: "PROCESSING"
        })

      id = Map.get(order, :id)

      Transactions.create_order_items(%{
        order_id: id,
        order_item: decodedData["product_id"],
        amount: decodedData["amount"]
      })
    end)

    {:ok, :commit, []}
  End
```

Zaključak

Kroz ovaj rad uvidjeli smo da su mikroservisi vrlo kompetentna arhitektura koja donosi puno pozitivnih stvari. Međutim kao što vrijedi za bilo koju arhitekturu koja postoji, mikroservisna arhitektura nije rješenje za sve naše probleme. Sama arhitektura, kao i sve ostale arhitekture, nosi značajke negativne strane, što je bilo vidljivo kroz razvoj primjera. Za manje programerske timove, monolitne aplikacije su mongo bolji odabir isključivo zbog procesa razvoja. Mikroservisne arhitekture funkcioniraju odlično u velikim kompanijama gdje se svaki manji tim pojedinačno može brinuti o jednom servisu, gdje ga oni praktički posjeduju. Smanjeni troškovi skaliranja, smanjeni scope projekta po timu i relativna jednostavno servisa se pokazala kao dobitna kombinacija u svijetu gdje tehnologija sve brže i brže napreduje

Popis literature

[1] Saša Jurić, *Elixir in Action, 2nd Edition* (2019.) Manning

[2] Jesse Howarth (2020.) Why Discord is switching from Go to Rust. Preuzeto 10.07.2022. https://discord.com/blog/why-discord-is-switching-from-go-to-rust

[3] Philip Brown (2016.) Understanding Function Arity in Elixir. Preuzeto 11.07.2022. https://www.culttt.com/2016/05/02/understanding-function-arity-elixir

[4] Elixir-lang.org (n.d.) Atom. Preuzeto 13.07.2022 https://elixir-lang.org/getting-started/basic-types.html#atoms

[5] Elixir-lang.org (n.d.) Macros. Preuzeto 13.07.2022 https://elixir-lang.org/getting-started/meta/macros.html

[6] Hexdocs (n.d.) Guards. Preuzeto 13.07.2022 https://hexdocs.pm/elixir/guards.html#list-of-allowed-expressions)

[7] Docker (n.d.) Use containers to Build, Share and Run your applications. Preuzeto 01.09.2022 https://www.docker.com/resources/what-container/

[8] Docker (n.d.) Dockerfile. Preuzeto 01.09.2022 https://docs.docker.com/engine/reference/builder/#:~:text=A%20Dockerfile%20is%20a%20text,can%20use%20in%20a%20Dockerfile%20.)

[9] Docker (n.d.) Docker Compose. Preuzeto 01.09.2022. https://docs.docker.com/compose/

[10] Kafka (n.d.) Introduction, Preuzeto 03.09.2022. https://kafka.apache.org/intro

[11] Solace (13.04.2021). What is Event Streaming Platform, Preuzeto 03.09.2022. https://solace.com/blog/what-is-an-event-streaming-platform/

[12] Linkedin Engineering (06.01.2012) First Apache release for Kafka is out, Preuzeto 03.09.2022. https://engineering.linkedin.com/kafka/first-apache-release-kafka-out

[13] NGINX (n.d.) What is NGINX? How different is it from Apache (for example)?. Preuzeto 31.08.2022, https://www.nginx.com/faq/what-is-nginx-how-different-is-it-from-e-g-apache/

[14] NGINX (n.d) F5 NGINX Plus: API Gateway. Preuzeto 31.08.2022, https://www.nginx.com/products/nginx/api-gateway/

[15] PostgreSQL (n.d.) About, Preuzeto 31.08.2022, https://www.postgresql.org/about/

[16] Amazon (n.d.) What is PostgreSQL. Preuzeto 31.08.2022. https://aws.amazon.com/rds/postgresql/what-is-postgresql/

[17] PostgreSQL (n.d.) PostgreSQL versions. Preuzeto 06.09.2022. https://www.postgresql.org/about/news/postgresql-145-138-1212-1117-1022-and-15-beta-3-released-2496/

[18] React (n.d.) React Preuzeto 04.09.2022 https://reactjs.org/

[19] tkrotoff (n.d.) Front-end frameworks popularity (React, Vue, Angular and Svelte). Preuzeto 07.09.2022 https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190).

[20] RisingStack, (31.05.2022) The History of React.js on a Timeline Preuzeto 03.09.2022 https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/

[21] Andre Pedriago (30.10.2018) The Rise of Functional Components – React v16.7.0 (now with Hooks!) Preuzeto 03.09.2022 https://blog.weareredlight.com/the-rise-of-functional-components-react-v16-7-0-now-with-hooks-90cac72b8ab7

[22] Dataversity (22.04.2021) A Brief History of Microservices. Preuzeto 29.08.2022 https://www.dataversity.net/a-brief-history-of-microservices/#

[23] DZone (10.03.2016) A Quick Primer on Microservices. Preuzeto 29.08.2022 https://dzone.com/articles/a-quick-primer-on-microservices

[24] CIO Wiki (n.d.) Simple Object Access Protocol (SOAP). Preuzeto 29.08.2022 https://cio-wiki.org/wiki/Simple_Object_Access_Protocol_(SOAP)

[25] Sam Newman, *Building Microservices: Designing Fine-Grained Systems, 2nd Edition*, (2021.) O'Reilly Media, Inc

[26] NGINX (19.05.2015) Introduction to Microservices. Preuzeto 29.08.2022. https://www.nginx.com/blog/introduction-to-microservices/

[27] Baeldung (31.03.2022) Database Design in a Microservice Architecture- Preuzeto 01.09.2022. https://www.baeldung.com/cs/microservices-db-design

[28] Developer.com (02.11.2021) Overcoming the Common Microservices Anti-Patterns. Preuzeto 01.09.2022 https://www.developer.com/design/solving-microservices-anti-patterns/

[29] Ketan Bhatt (03.07.2020) Race Conditions/Concurrency Defects in Databases: A Catalogue Preuzeto 01.09.2022 https://ketanbhatt.com/db-concurrency-defects/

[30] Keboola (12.08.2021) What are the ACID properties of transactions and why do they matter in data engineering? Preuzeto 02.09.2022 https://www.keboola.com/blog/acid-transactions

[31] Hector Garcsa-Molina i Kenneth Salem (n.d.) Sagas. Preuzeto 02.09.2022 https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

[32] Auth0 (n.d.) What is Authentication. Preuzeto 03.09.2022. https://auth0.com/intro-to-iam/what-is-authentication/

[33] Auth0 (n.d.) What is Autorization. Preuzeto 03.09.2022 https://auth0.com/intro-to-iam/what-is-authorization/

[34] Doit (28.06.2022) Authentication between microservices: Is it really that hard?. Preuzeto 03.09.2022. https://www.doit-intl.com/authentication-between-microservices-is-it-really-that-hard/

[35] Auth0 (n.d.) Single Sign-on. Preuzeto 03.09.2022. https://auth0.com/docs/authenticate/single-sign-on

[36] Thread Safe (n.d.) JWT 5 years in making. (A history) Preuzeto 03.09.2022. http://www.thread-safe.com/2015/01/jwt-5-years-in-making-history.html

[37] JWT (n.d.) Introduction to JSON Web Tokens. Preuzeto 04.09.2022. https://auth0.com/intro-to-iam/what-is-authorization/

Popis slika

Popis tablica

Prilozi radu

Programski kod aplikacije: https://github.com/mislavpericaxilis/product-services