Sodobne tehnologije in arhitektura za implementacijo prototipa

Matevž Mak
matevz.mak@gmail.com
UP FAMNIT - PRIN 2. year
Koper, Slovenia

Matjaž Kljun
matjaz.kljun@upr.si
UP FAMNIT
Koper, Slovenia

ABSTRACT

V tem članku podrobno raziskujemo uporabo sodobnih tehnologij, ogrodij in principov, ki so ključni pri oblikovanju in razvoju prototipov na inovativen način. Glavni cilj je podrobno analizirati izbrane tehnologije, osvetliti njihove prednosti in utemeljiti, zakaj so bile izbrane kot temelj za razvoj prototipov. Osrednji del članka se posveti tehnološkemu sklopu MEAN (MongoDB, Express, Angular in Node.js), ki je v zadnjih letih postal priljubljena izbira za razvoj spletnih aplikacij. V članku razložimo tudi, kako se razvijalci lahko učinkovito poslužujejo sistema za upravljanje različic Git, pri čemer posebej izpostavimo uporabo podmodulov za boljše upravljanje s kodo. V nadaljevanju se poglobimo v uporabo tehnologije Docker, ki omogoča lažjo vzpostavitev in upravljanje zalednih sistemov ter uporabniških vmesnikov na strežniški strani. Na koncu članka predstavimo vrste testiranja, ki jih lahko izvedemo s pomočjo knjižnice k6. S tem orodjem je mogoče izvesti različne oblike testov, vključno s koničnimi in obremenitvenimi testi, kar nam omogoča, da zagotovimo robustnost in učinkovitost naših aplikacij.

1 UVOD

V času izobraževanja se študenti srečujejo z različnimi tehnologijami, ki so namenjene razvoju prototipov. Lahko gre za web aplikacije, mobilne aplikacije, programske rešitve ali kaj drugega. V današnjem hitro spreminjajočem se tehnološkem svetu je izbira prave tehnologije ključna za učinkovit in kakovosten razvoj. V tem članku želimo bralcem pokazati, kako lahko vzpostavijo okolje za razvoj in implementacijo aplikacij na strežniški strani.

Tukaj je izbor tehnologij izjemno pomemben, saj lahko prava kombinacija orodij znatno pohitri razvojni proces. Ta izbor mora omogočati tudi harmonično povezovanje z različnimi sloji v našem prototipu, od uporabniškega vmesnika do podatkovne baze. Da bi naredili premišljen izbor, je potrebno temeljito raziskati namen prototipa, ter kritično oceniti, katera tehnološka zbirka je najbolj primerna za razvoj takšnega prototipa.

V naslednjem poglavju bomo podrobneje opisali uporabo tehnološke zbirke MEAN (MongoDB, Express, Angular in Node.js). Osredotočili se bomo na razčlenitev, kaj v tej zbirki spada pod interakcijo odjemalca (npr. uporabniški vmesnik, varnost) in kaj pod zaledni sistem (npr. upravljanje podatkov, logika aplikacije). Prav tako bomo analizirali prednosti posamezne tehnologije v tej zbirki, ter kako lahko z njihovo uporabo pohitrimo sistem.

Ampak, kako vemo, da je naš sistem resnično hiter in učinkovit? Za to bomo v članku predstavili tudi knjižnico k6, ki je vodilno orodje za izvajanje obremenitvenih testov. Pokazali bomo, kako s pomočjo k6 lahko testiramo naš sistem pod različnimi obremenitvami, prepoznamo morebitne ozke grla in se nato odzovemo z ustreznimi optimizacijami.

S tem člankom želimo bralcem ponuditi celovit vodnik za vzpostavitev robustnega in učinkovitega razvojnega okolja, ki bo kos zahtevam sodobnih aplikacij.

2 TEHNOLOŠKA ZBIRKA MEAN

MEAN je zbirka tehnologij, ki temeljijo na programskem jeziku JavaScript. Ta zbirka vključuje podatkovno bazo MongoDB, ogrodje Express za strežniško stran, Angular za oblikovanje in strukturiranje uporabniških vmesnikov ter izvajalno okolje Node.js za strežniško stran.

Slika 1 prikazuje arhitekturo in tok podatkov v MEAN tehnološkem skladu. Na levi strani imamo odjemalca, ki v našem sistemu izvaja različne zahteve. Logika se nato obdela z Angular ogrodjem, ki pripravi servisni klic za strežniško stran.

Naslednja stopnja vključuje okolje Node.js, v katerega je vključeno ogrodje Express. To ogrodje ustrezno prestreže klic in na podlagi definirane poti v našem programskem vmesniku (API) izvede ustrezno poizvedbo na našo instanco podatkovne baze MongoDB.

V instanci podatkovne baze je naš upravitelj, ki hrani metapodatke naše gruče. Na ta način lahko izvede ustrezno poizvedbo na ustrezen del črepinjenja (angl. **sharding**). Ti podatki se nato vrnejo v Node.js okolje, kjer se preverjajo po ustreznem shematskem modelu. Če je vse preverjanje uspešno, se podatki pošljejo nazaj na Angular instanco, kjer se prikažejo na uporabniškem vmesniku.

Vsaka instanca je locirana v svojem Docker vsebniku (angl. *container*), kar omogoča individualno spremljanje stanja vsakega vsebnika, pregledovanje dnevnikov napak

in podobno. Podrobnejše informacije o Dockerju bodo predstavljene v naslednjem poglavju.

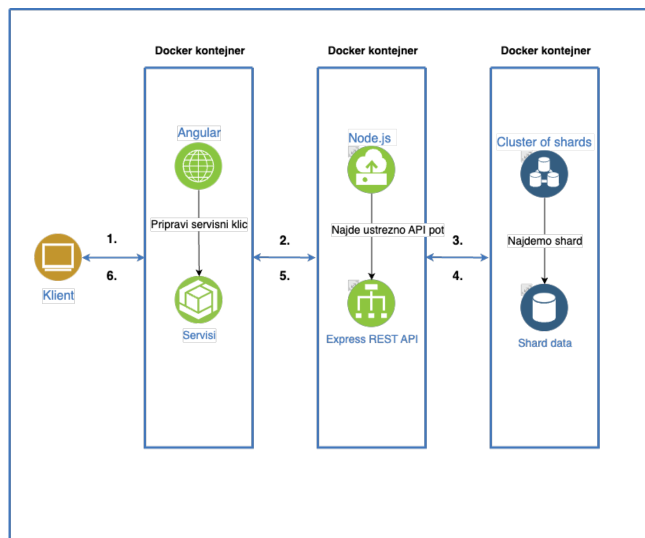

Figure 1: Arhitektura MEAN projekta.

3 ANGULAR

Angular je ogrodje, ki nam pomaga pri organizaciji in implementaciji spletne aplikacije. Gre za ogrodje, ki ga je razvil Google, ker so imeli težave s svojim orodjem Google web toolkit [1]. Angular je osnovan na jeziku TypeScript in komponentni arhitekturi. Vsaka komponenta vključuje datoteke HTML, SCSS in TypeScript. Datoteka HTML omogoča dodajanje gradnikov naše komponente, SCSS pa poskrbi za estetsko oblikovanje teh gradnikov. Logiko komponente dodamo s pomočjo TypeScripta.

Ta struktura omogoča visoko preglednost in ločljivost programske kode, kar bistveno prispeva k vzdržljivosti projektov. Komponente so ponovno uporabne in se lahko učinkovito vključijo v različne dele projekta. Ogrodje prav tako definira svoj usmerjevalnik, s pomočjo katerega lahko določimo vse dostopne poti naše spletne aplikacije. V primeru, da uporabnik dostopa do url-jev, ki niso dostopni znotraj aplikacije, ga obvestimo z opozorilom 404. Za večjo varnost lahko poti zaščitimo s stražarjem, ki zahteva pravilen avtentikacijski žeton za dostop do Angular komponent.

Angular nam omogoča ločeno implementacijo servisov, ki omogočajo dostop do našega zalednega sistema. Velika prednost Angularja je, da temelji na statičnem jeziku TypeScript, kar nam omogoča definiranje statičnih tipov za naše spremenljivke in s tem zagotavlja robustnost našega projekta.

V primeru, da zmanjka idej ali znanja, nam Angular omogoča enostavno integracijo knjižnic iz registra nmp. Za optimizacijo delovanja je vsako funkcionalnost v Angularju mogoče zapakirati v tako imenovani “feature modul”. Ta se nato lahko z metodo “lenega nalaganja” vključi v naše poti komponente, s čimer se zmanjša začetni čas nalaganja.

Angular v osnovi upodablja spletno stran v brskalniku, kar lahko predstavlja težavo, saj strani ne morejo biti optimalno indeksirane s strani spletnih iskalnikov, kot je Google [2]. To je posledica dejstva, da v viru spletne strani ni nobenega besedila. Da bi to težavo odpravili, moramo spletno aplikacijo Angular upodabljati na strežniški strani. Dodatna prednost tega pristopa je, da končni uporabnik lahko vidi spletno stran, še preden je ta v celoti funkcionalna, kar je odlično za mobilne naprave oziroma za nizke hitrosti prenosa podatkov. Za ta namen lahko uporabljamo razširitev Angular Universal, ki omogoča strežniško upodabljanje spletne aplikacije.

Angular je del trojice najbolj priljubljenih ogrodij JavaScript, kjer se mu pridružujeta še React in Vue.js. Ena izmed pomembnih značilnosti Angularja je napredno upravljanje s podatki. V tem kontekstu je še posebej pomembna metoda dvosmerne vezave podatkov. Na eni strani imamo predlogo, na drugi pa komponento. Podatki potujejo iz komponente v predlogo prek vezave lastnosti in obratno – iz predloge do komponente prek vezave dogodkov, ki jih sproži uporabnik [1].

Prav tako ima vsaka komponenta svoj življenjski cikel. Angular nam omogoča manipulacijo komponent v različnih fazah tega cikla, kar zagotavlja boljšo uporabniško izkušnjo [1].

4 NODE.JS

Node.js je izvajalno okolje za JavaScript, ki nam omogoča implementacijo programskega vmesnika ali API-ja[1] za zaledni sistem, ki ga lahko naše spletne aplikacije kličejo preko servisov. Pri drugih programskih jezikih se za vsak klic ustvari svoja nit. Problem se pojavi, ko naš strežnik dobi N+1-klicev ob zmogljivosti N-klicev - zadnji klic mora čakati, da se sprosti ena izmed N-niti [3].

Ta izziv se pogosto rešuje z nadgradnjo procesorske moči našega strežnika ali z dodajanjem pomnilnika, vendar so to pogosto finančno zahtevne nadgradnje. Poleg tega obstajajo omejitve zaradi časovno potratnih vhodno/izhodnih V/I procesov. Na primer: če se izvaja klic, ki vsebuje sistemske izpise, mora nit čakati, dokler se izpisi ne zabeležijo v konzoli. Nekateri to označujejo kot blokiranje niti, vendar gre dejansko za počasno kodo [3].

Node.js deluje z eno glavno nitjo. Prednosti tega so asinhroni klici in zanka dogodkov. Asinhroni povratni

[1] *Application programming interface* je programski vmesnik, ki omogoča način, kako dva ali več računalniških programov komunicira med seboj. Gre za vrsto programskega vmesnika, ki ponuja storitev drugim delom programske opreme.

klici so lahko spletni klici programskih vmesnikov, asinhroni JavaScript in XML[2] klici, funkcije, ki se izvedejo po določenem časovnem intervalu, itd. Algoritem običajno obravnava ukaze sekvenčno, vrstico za vrstico, pri čemer vsaka vrstica vsebuje ukaz, ki se doda na sklad ukazov.

V primeru Node.js se ukazi dodajajo na sklad, toda ko pride do asinhronih klicev, se ne dajo na sklad, ampak se izvedejo ločeno s pomočjo C knjižnice Libuv. Algoritem se kljub temu nadaljuje in izvaja ukaze vrstico za vrstico. Ko se asinhroni povratni klic konča, se doda v zanko dogodkov. Ko se algoritem konča in je sklad ukazov izpraznjen, se preveri zanka dogodkov za morebitne preostale ukaze, ki jih je treba izvesti.

Tukaj je prikazan primer algoritma [4]:

```
console.log('One');
console.log('Two');
setTimeout(function() {
console.log('Three');
}, 2000);
console.log('Four');
console.log('Five');
```

Za boljše razumevanje prikažimo izpis tega algoritma, ki deluje na sinhronem principu:

```
One
Two
... (2 sekundi pavza) ...
Three
Four
Five
```

V zgornjem primeru algoritem vrstično izvaja ukaze in čaka dve sekundi pred izpisom 'Three'. Sinhrono delovanje algoritma tako ustvarja pavzo, kar lahko vodi do neizkoriščenosti strežniških kapacitet.

Če pa algoritem obravnavamo na asinhron način, dobimo drugačen rezultat:

```
One
Two
Four
Five
... (2 sekundi zamude) ...
Three
```

V tem primeru algoritem ne čaka, ampak nadaljuje z izvajanjem ukazov, zato 'Three' izpiše z zamudo. To pomeni, da nimamo blokirane niti in posledično dosežemo večjo učinkovitost strežnika. O metodah za optimizacijo zalednega sistema bomo podrobneje govorili v naslednjih poglavjih.

[2] *Extensible markup language* je zapis za izmenjavo strukturiranih dokumentov na spletu.

Express

Express je visokoprilagodljivo ogrodje za Node.js, ki zagotavlja obsežen nabor funkcionalnosti za razvoj različnih vrst aplikacij. To vključuje izdelavo API-jev, enostranskih, večstranskih in hibridnih spletnih aplikacij [5].

Ena izmed glavnih prednosti Expressa je enostavnost razvoja API-jev. Zaradi intuitivne in fleksibilne narave tega ogrodja lahko razvijalci prihranijo veliko časa. Poleg tega je Express napisan v JavaScriptu, ki je eden najbolj priljubljenih programskih jezikov, znan po svoji enostavnosti in široki podpori. Zaradi tega se Express brezhibno integrira z ostalimi JavaScript ogrodji, kot sta Angular in React.

Express se ponaša tudi s svojo učinkovitostjo in hitrostjo. Je asinhrono ogrodje, kar pomeni, da lahko enostavno obvladuje veliko število zahtevkov hkrati, ne da bi se to negativno odrazilo na njegovo zmogljivost. V praksi lahko razvijalci vzpostavijo lokalni strežnik z nekaj preprostimi vrsticami kode, kar še dodatno povečuje produktivnost.

Kljub temu ima Express tudi nekaj slabosti. Med njimi je najpomembnejša pomanjkanje stroge strukture kode. To pomeni, da morajo razvijalci sami vzpostaviti strukturo aplikacije, kar lahko vodi do neurejenosti in slabše berljivosti kode, če ni ustrezno upravljano. Prav tako Express občasno prikazuje težave s povratnimi funkcijami, ki se lahko manifestirajo kot zapleteni in težko sledljivi vzorci kode [5].

Tako kot vsako orodje ima tudi Express svoje prednosti in slabosti. Kljub temu pa so njegova fleksibilnost, enostavnost uporabe in dobro uveljavljena podpora v JavaScript skupnosti ključ do njegove priljubljenosti pri razvijalcih po vsem svetu [5].

5 MONGODB

MongoDB je objektno orientirana NoSQL[3] baza podatkov, ki deluje na principu urejenih parov (ključ, vrednost). V njej so tabele zamenjane z zbirko dokumentov, kar omogoča veliko fleksibilnost pri shranjevanju podatkov. Za razliko od tradicionalnih relacijskih baz podatkov MongoDB ne zahteva vnaprej definirane sheme za ustvarjanje dokumentov. Ta lastnost je koristna pri testiranju, vendar se v produkcijskem okolju redko uporablja.

Vsak dokument v MongoDB bazi podatkov je shranjen v obliki JSON[4] objekta, kar poudarja objektno orientiranost te baze. Ob ustvarjanju se vsakemu dokumentu dodeli unikaten identifikator, ki služi kot glavni indeks zbirke. Ta identifikator omogoča hitro in enostavno iskanje specifičnih dokumentov v zbirki.

[3] *Not only structured query language* oziroma ne samo strukturiran povpraševalni jezik za delo s podatkovnimi bazami.

[4] *JavaScript Object Notation* je standardizirana oblika za pomnjenje in izmenjavo podatkov v besedilni obliki.

V shemo je možno dodati več indeksov, kar izboljša hitrost iskanja po specifičnih atributih. V odsotnosti ustreznega indeksa mora MongoDB izvesti iskanje po celotni zbirki, kar je lahko zamudno in neučinkovito. Zato je smiselno uporabiti indekse za tiste atribute, po katerih se pogosto išče.

MongoDB prav tako podpira koncept črepinjenja (angl. *sharding*), kar omogoča vertikalno in horizontalno skaliranje podatkovne baze [4]. Ta značilnost je še posebej pomembna pri obvladovanju velikih količin podatkov in zagotavljanju hitre in učinkovite obdelave. Podrobnosti o optimizaciji podatkovne baze bomo obravnavali v nadaljevanju.

6 GIT PODMODULI

V okviru našega sistema smo aktivno uporabljali Git[5] podmodule, ki omogočajo vključitev drugih Git repozitorijev kot podprojektov v naš glavni repozitorij. Ta pristop znatno poenostavlja vzdrževanje celotnega sistema in njegovih posameznih komponent. V našem primeru smo kot podprojekte v glavni repozitorij integrirali dva dodatna repozitorija, in sicer repozitorij zalednega sistema in repozitorij za uporabniški vmesnik.

Podmodule v Gitu odlikuje možnost izdelovanja različic, saj vsak podprojekt ohranja svojo samostojno zgodovino različic. To nam omogoča, da v glavnem repozitoriju jasno določimo, katera različica podprojekta oz. zunanjega repozitorija se bo uporabljala.

Poleg tega podmoduli omogočajo enostaven pregled nad zgodovino sprememb, kar je bistveno za sledenje razvoju in odpravljanje morebitnih napak. Še več, podmoduli predstavljajo učinkovito orodje za ponovno uporabo kode, saj lahko posamezne podprojekte enostavno pakiramo in uporabimo kot zunanje knjižnice za druge projekte.

S tem pristopom smo povečali modularnost, transparentnost in učinkovitost našega razvojnega procesa. Znotraj našega sistema smo ustvarili osrednji repozitorij, ki vsebuje zaganjalne datoteke za izgradnjo zalednega sistema in grafičnega vmesnika s pomočjo Docker tehnologije. Tako zaledni sistem kot uporabniški vmesnik predstavljata dva ločena repozitorija, vključena v glavni repozitorij.

7 DOCKER

Docker je orodje za programske aplikacije, ki omogoča ustvarjanje, upravljanje in orkestracijo vsebnikov. Izvira iz odprtokodnega projekta Moby, katerega kodo lahko najdemo na spletni strani GitHub. Glavno vzdrževanje odprtokodnega projekta izvaja podjetje Docker, ki ponuja tudi poslovno različico tehnologije.

Vsak Docker vsebnik ima svoje lokalno pomnilniško območje, v katerem so zloženi plasti Docker slike in datotečni sistem. Na tem nivoju se izvajajo operacije branja in pisanja. Na voljo so različni gonilniki za pomnilnik, izbira katerih vpliva na zmogljivost in stabilnost. V primeru distribuiranega sistema lahko za eno vozlišče določimo samo en gonilnik. Vsi vsebniki, ki so del tega vozlišča, bodo uporabljali isti gonilnik [6].

Docker pogon

Docker pogon je odgovoren za poganjanje in upravljanje Docker vsebnikov. Zasnovan je modularno, kar omogoča zamenjavo posameznih komponent, s čimer povečamo prilagodljivost. Glavne komponente, ki sestavljajo Docker pogon, vključujejo Docker odjemalca, deamona, containerd in runc. Skupno delovanje teh modulov omogoča ustvarjanje in zagon Docker vsebnikov [6]. Slika 2 prikazuje gradnike Dockerja in relacije med njimi.

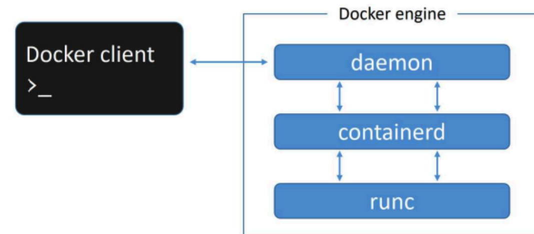

Figure 2: Gradniki Dockerja [6].

Docker slike

Docker slike lahko opišemo kot objekte, ki vsebujejo datotečni sistem operacijskega sistema in aplikacijo. Te slike delujejo kot podlaga za virtualne stroje. Vsaka slika ima svoj edinstven identifikator, po katerem lahko referenciramo slike, ali pa uporabimo njihovo ime. Docker ima na voljo svoj register slik, iz katerega lahko povlečemo poljubne Docker slike. Najbolj priljubljen register je Docker Hub [6].

Docker vsebniki

Znotraj slik lahko zaženemo Docker vsebnike, kar nam omogoča delo v virtualnem okolju, ki se obnaša kot "nov računalnik". Kot primer:

```
> docker container run -it microsoft
/powershell:nanoserver

Izhod:
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation.
All rights reserved.
PS C:\>
```

[5] *Git* je sistem za porazdeljeno dokumentiranje sprememb in izdajanje različic datotek.

Ukaz ‘run‘ zažene nov vsebnik, zastavica ‘-it’ pa Dockerju naroči, da mora vsebnik delovati interaktivno in da naj uporabi Microsoft PowerShell. Na koncu se določi, katero ukazno vrstico naj izvede znotraj vsebnika. Izhod prikazuje, da se je uspešno zagnal Windows PowerShell.

Iz razvijalskega vidika lahko Dockerju nakažemo, kako naj ustvari Docker sliko naše aplikacije. Za ta primer je treba dodati Dockerfile, kjer so definirani ukazi, kako se aplikacija vzpostavi.

Iz vsake Docker slike lahko generiramo enega ali več vsebnikov, med katerimi nato vzpostavimo povezavo. Če želimo izbrisati specifično Docker sliko, moramo pred tem izbrisati vse vsebnike, ki so povezani s to sliko [6]. To relacijo prikazuje Slika 3.

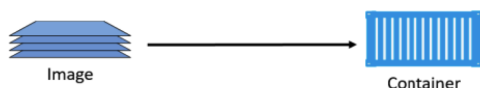

Figure 3: Povezanost Docker slik in vsebnikov [6].

Uporaba Dockerja v našem sistemu

V okviru sistema smo implementirali štiri ključne komponente: zaledni sistem, uporabniški vmesnik, MongoDB in Mongo-Express. Te komponente so bile med seboj povezane in so delovale skladno, da bi zagotovile celovito delovanje aplikacije. Vsaka komponenta je bila definirana kot storitev znotraj Docker Compose datoteke.

Zaledni sistem je predstavljal osrednji del sistema in je bil zasnovan s pomočjo posebej pripravljene Dockerfile datoteke. Za zagotavljanje zanesljivega delovanja je bil zaledni sistem konfiguriran tako, da se je samodejno ponovno zagnal ob morebitnem zastoju. Različne okoljske spremenljivke so bile uporabljene za konfiguracijo delovanja zalednega sistema (kar izboljša varnost), vključno s podatki za dostop do MongoDB strežnika in SMTP strežnika[6]. Zaledni sistem je bil povezan z internim Docker omrežjem sudlee_network in je preusmerjal vrata na zunanji svet.

Uporabniški vmesnik je bil prav tako zasnovan s pomočjo Dockerfile datoteke. Podobno kot zaledni sistem se je tudi uporabniški vmesnik samodejno ponovno zagnal ob morebitnem zastoju. Storitev je bila povezana z istim internim Docker omrežjem kot zaledni sistem.

Kot osnovo za shranjevanje podatkov smo uporabili MongoDB bazo podatkov. Za to storitev smo uporabili Docker sliko MongoDB verzije 5, nastavljeno na focal. Tudi ta komponenta je bila konfigurirana z okoljskimi spremenljivkami, medtem ko so bili podatki shranjeni v Docker prostoru za shranjevanje studlee-mongo. MongoDB je bil prav tako povezan z internim Docker omrežjem sudlee_network.

Za upravljanje MongoDB baze podatkov smo uporabili Mongo-Express, spletni vmesnik, ki je bil zasnovan na Docker sliki mongo-express. Ta storitev je bila odvisna od zgoraj omenjene MongoDB storitve in je bila prav tako povezana z internim Docker omrežjem sudlee_network.

Na koncu smo v Docker Compose datoteki definirali še prostor za shranjevanje studlee-mongo in interno Docker omrežje sudlee_network, ki sta bila uporabljena s strani zgoraj opisanih storitev.

V našem sistemu smo definirali dve Dockerfile datoteki, eno za zaledni sistem in drugo za uporabniški vmesnik. Vsaka datoteka vključuje niz ukazov, ki zagotavljajo postavitev obeh okolij, ob upoštevanju, da vsako od njiju temelji na različnih tehnologijah. Oba sistema sta nato nastavljena, da komunicirata z zunanjim svetom prek svojih specifičnih vrat.

8 TESTIRANJE CELOTNEGA SISTEMA

Obremenitveni test

Za test obremenitve smo strukturirali test, ki je imel naslednje faze:

- Začetna faza povečanja, kjer se obremenitev postopoma povečuje do cilja 800 virtualnih uporabnikov v obdobju 5 minut.
- Faza vzdrževanja, kjer obremenitev ostane konstantna pri 800 virtualnih uporabnikih v trajanju 10 minut.
- Nazadnje faza zmanjšanja, kjer število virtualnih uporabnikov pade na nič v obdobju 5 minut.

Časi odziva in stopnje napak so bili neprekinjeno spremljani. Naše mejne vrednosti so bile nastavljene tako, da bi stopnja neuspeha zahteve morala biti manjša od 1 % in da bi 99 % zahtev moralo biti končanih pod 200 ms. Cilj tega testa je bil izmeriti maksimalno obremenitev, ki jo lahko vsak strežnik obvlada, preden krši te mejne vrednosti zmogljivosti.

Test konic

Za izvedbo obremenitvenega testa smo uporabili podatke, ki simulirajo povečanje obremenitve - tako imenovani test konic obremenitve (angl. *spike test*). Ta scenarij testiranja je posebej uporaben za preverjanje, kako se sistem odziva na nenadno in veliko povečanje uporabnikov oziroma zahtevkov.

Za ta test smo uporabili več faz obremenitve. Začeli smo s 100 uporabniki za 10 sekund, kar predstavlja normalno obremenitev. Nato smo to število ohranili konstantno za naslednjo minuto. Naslednji korak je predstavljal vrhunec našega testa. V tej fazi smo povečali obremenitev na 1400 uporabnikov v obdobju treh minut ter tako simulirali nenadno povečanje obremenitve. Po dosegu vrha smo začeli z zmanjševanjem obremenitve. V 10 sekundah smo število

[6] *Simple mail transport protocol* je internetni protokol za prenos elektronske pošte.

uporabnikov zmanjšali na 100. To število smo ohranili za naslednjih tri minute. Na koncu smo obremenitev popolnoma odstranili, tako da smo število uporabnikov v 10 sekundah zmanjšali na nič.

V tem testu smo uporabili tudi dve mejni vrednosti: stopnja napake zahtevkov naj bi bila manjša od 1 % in 95 % vseh zahtevkov naj bi bilo dokončanih v manj kot 500 ms.

Replikacija podatkovne baze

Replikacijska množica v MongoDB je skupina procesov, imenovanih *mongod*, ki ohranjajo enak nabor podatkov. Značilnost teh replikacijskih množic je, da prispevajo k redundanci in visoki dostopnosti našega sistema. V tem poglavju bomo podrobneje opisali arhitekturo različnih replikacijskih množic in opravila, ki jih te izvajajo [7].

Osrednja zamisel je distribucijska razporeditev podatkov na več strežnikov. Vsak strežnik vključuje tudi replikacijo, s čimer izboljšamo redundanco in dostopnost. Zaradi porazdeljene narave sistema se zmanjša verjetnost odpovedi sistema na eni točki. Na primer, če en strežnik preneha delovati, je na voljo drug strežnik, ki bo v tem času prevzel njegovo vlogo.

Replikacijska množica vključuje vozlišča, ki hranijo podatke, in vozlišče, ki deluje kot arbiter. Če se natančneje osredotočimo na strukturo vozlišč, ki hranijo podatke, lahko ugotovimo, da je lahko samo eno izmed njih določeno kot primarno vozlišče, preostala pa so sekundarna [7]. Na Sliki 4 lahko vidimo strukturo replikacijske množice.

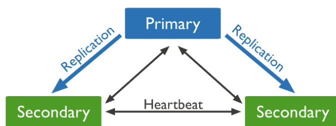

Figure 4: Struktura replikacijske množice [7].

Do podatkov v primarnem vozlišču lahko dostopamo z operacijami branja in pisanja. Primarno vozlišče vsebuje zapisnik vseh sprememb in operacij, ki so se zgodile na določenem naboru podatkov. Sekundarna vozlišča replicirajo metapodatke primarnega vozlišča, tako da postanejo identična primarnemu. V primeru, da primarno vozlišče ni na voljo, imamo sekundarno vozlišče, ki je potencialni kandidat za prevzem vloge primarnega, in v tem primeru se sproži postopek volitev za izbiro novega primarnega kandidata [7]. Na Sliki 5 lahko vidimo, kako poteka izbor novega primarnega vozlišča.

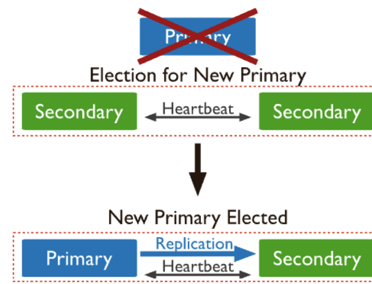

Figure 5: Izbor novega primarnega vozlišča [7].

9 PARALELNO IZVAJANJE ZALEDNEGA SISTEMA

V poglavju o Node.js smo poudarili, da deluje z enojnim glavnim procesom, kar omogoča neblokirajočo obdelavo nalog. Med izvajanjem procesa lahko izvajamo asinhrone klice, medtem ko glavna nit nadaljuje z izvajanjem ostalih delov algoritma. Strežnik, ki gosti naš sistem, vsebuje dvojedrni procesor, kar nam omogoča, da lahko naš strežniški sistem hkrati izvaja dva procesa.

Da bi optimalno izkoristili večjedrnost našega strežnika, v našo Node.js aplikacijo vključimo modul gruče. Ta ustvari več instanc naše aplikacije in jih vzporedno izvaja. Modul zagotavlja tudi to, da se obremenitev enakomerno porazdeli med posamezne instance aplikacije s pomočjo algoritma krožnega dodeljevanja (angl. *round robin scheduling*). V primeru odpovedi ene od instanc aplikacije Node.js funkcionalnost prevzamejo preostale delujoče instance [8].

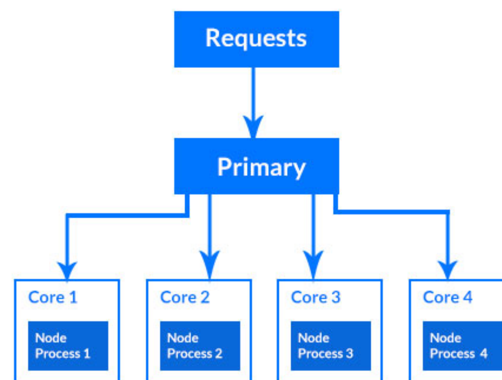

Figure 6: Delitev bremenitve na delavce [8].

Na Sliki 6 je prikazano delovanje tega sistema. Ko odjemalec izvede klic, ta doseže primarno vozlišče, ki na podlagi algoritma krožnega dodeljevanja določi, kateri procesor bo nalogo izvedel. S tem zagotovimo uravnoteženo obremenitev procesorjev in preprečimo, da bi bil en procesor bolj obremenjen kot drugi.

10 REZPRAVA

V tej razpravi smo se osredotočili na uporabo tehnološke zbirke MEAN (MongoDB, Express.js, AngularJS, Node.js) za postavitev in vodenje zahtevnejših projektov. Poudariti je treba, da medtem ko MEAN predstavlja močno in prilagodljivo zbirko, je izbor tehnologije vedno kontekstualen. Vsak razvojni tim bi moral skrbno razmisliti o svojih specifičnih potrebah in omejitvah, preden se odloči za določeno zbirko orodij. To je ključno ne samo za zagotavljanje učinkovitosti, temveč tudi za spodbujanje kakovosti in zanesljivosti celotnega sistema.

Avtomatizacija je drugi ključni element, ki ga ne smemo zanemariti. V našem primeru smo uporabili Jenkins, odprtokodno orodje za neprekinjeno integracijo in dostavo, ki avtomatizira gradnjo, testiranje in razporeditev aplikacij. Ko razvijalec pošlje kodo v repozitorij na GitHubu, Jenkins sproži postopek, ki lahko vključuje gradnjo koda, testiranje in razporeditev. To ne samo pospeši razvojni proces, ampak tudi zmanjša možnost napak, saj Jenkins avtomatsko pošlje opozorila v primeru težav pri zgradnji ali vzpostavitvi sistema.

Poleg tega je pomembno omeniti varnost in skalabilnost sistema. V našem primeru smo se osredotočili na zavarovanje sistema z uporabo replikacije podatkovnih baz in drugih mehanizmov za odpornost proti napakam. To povečuje zanesljivost sistema in omogoča gladko delovanje tudi v primeru okvar. Optimizacija sistema je prav tako kritična, saj zagotavlja, da se viri uporabljajo na najbolj učinkovit način. Priporočamo izvedbo obremenitvenih testov, ki lahko razkrijejo, kako različni strežniški procesorji reagirajo na povečane obremenitve, kar omogoča prilagoditev sistema za maksimalno zmogljivost.

V zaključku bi želeli poudariti, da je uspeh vsakega projekta v veliki meri odvisen od širokega nabora dejavnikov. Ti segajo od pravilne izbire tehnologije do učinkovite avtomatizacije in stalne optimizacije. Z izbiro orodij, ki najbolje ustrezajo potrebam in ciljem projekta, lahko razvijalci znatno pospešijo razvojni cikel in izboljšajo končni izdelek.

REFERENCES

[1] E. Saks, “Javascript frameworks: Angular vs react vs vue.,” 2019.
[2] “Server-side rendering (ssr) with angular universal,” 2023. Accessed: 2023-08-19.
[3] J. Ramón, “Everything you need to know about node.js,” 2023. Accessed: 2023-08-19.
[4] M. Satheesh, B. J. D’mello, and J. Krol, *Web development with MongoDB and NodeJs.* Packt Publishing Ltd, 2015.
[5] A. Sharma, “What is express js in node js?,” 2023. Accessed: 2023-08-19.
[6] N. Poulton, *Docker Deep Dive: Zero to Docker in a single book.* NIGEL POULTON LTD, 2020.
[7] “Replication,” 2023. Accessed: 2023-08-19.
[8] S. Ulili, “How to scale node.js applications with clustering?,” 2023. Accessed: 2023-08-19.