# Primjena Docker tehnologije u transakcijskim sustavima

**Fumić, Patrik**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Patrik Fumić**

# PRIMJENA DOCKER TEHNOLOGIJE U TRANSAKCIJSKIM SUSTAVIMA

**ZAVRŠNI RAD**

**Varaždin, 2019.**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Patrik Fumić**

**Matični broj: 44022/15–R**

**Studij: Informacijski sustavi**

# PRIMJENA DOCKER TEHNOLOGIJE U TRANSAKCIJSKIM SUSTAVIMA

**ZAVRŠNI RAD**

**Mentor:**

Doc. dr. sc. Zlatko Stapić

**Varaždin, Rujan 2019.**

*Patrik Fumić*

**Izjava o izvornosti**

Izjavljujem da je moj završni 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 potvrdio prihvaćanjem odredbi u sustavu FOI-radovi*

**Sažetak**

Ovaj rad se temelji na opisu Docker tehnologije i primjeni iste tehnologije u svrhu pogona aplikacijskih sustava. Detaljno je objašnjena uloga Dockera u općenitom napretku informacijskog sektora kao i njegove prednosti i nedostaci u odnosu na već postojeći standard pokretanja aplikacija na pripadajućim serverima. Navedeni sadržaj je dodatno razjašnjen putem konkretnog primjera instalacije i integracije Docker tehnologije za koju je razrađena tematika troškovnog aspekta, vršnog opterećenja te potreba koje nastaju u obliku potrebnih znanja stručnog osoblja za implementaciju i održavanje spomenutog sustava.

**Ključne riječi:** Docker, virtualizacija, aplikacije, isporuka, transakcijski sustav, kontejnerizacija

# Sadržaj

# 1. Uvod

Informacijske tehnologije su područje koje je od svojih početaka poznato po iznimno brzom napretku i samim time većoj rasprostranjenosti. Upravo ovi prethodno navedeni faktori su uzrok novonastalim problemima u području Informacijskih tehnologija, u nastavku navođenih kao IT. Neki od problema o kojima se govori su brzina izvođenja, troškovi, povećana potreba za hardwareom i kompleksnost u razvoju. Docker se ovdje nameće kao nositelj nove tehnologije, novog pristupa koji donosi rješenja za navedene probleme (Fong, 2017). Fokus ovog rada su konkretno transakcijski sustavi kao korisnik Docker tehnologije i primjer koji je potaknuo interes je Visa, tvrtka koja je povećala brzinu i efikasnost u radu te skalabilnost za deset puta, detaljnije u nastavku rada.

Cilj ovog rada je približiti Docker tehnologiju i pokazati prednost iste u IT sektoru u odnosu na uobičajene načine pakiranja i isporuke aplikacija. Ovaj cilj će se ostvariti kroz niz poglavlja u kojima ćemo prvo proći kroz teoretski dio, gdje ćemo obraditi nešto detaljnije samu Docker tehnologiju i uz to ukratko objasniti što su to transakcijski sustavi i kako ih unaprijediti uz već spomenutu Docker tehnologiju. Nakon toga ćemo na praktičnom primjeru Dockerizirati jednu web aplikaciju koju ćemo bazirati na .NET Core tehnologiji i na taj način prikazati uobičajeni postupak pakiranja i isporuke aplikacije uz pomoć Dockera i ujedno objasniti prednosti i mane ovog načina.

Osobno sam pronašao interes u ovoj temi zbog toga što smatram da je Docker sljedeći veliki korak naprijed u razvoju web aplikacija i njihovom upravljanju. Bez obzira na poteškoće koje će biti potrebno prebroditi u periodu migracije na novu tehnologiju smatram kako je vrijedno toga truda i kako će poznavanje Docker tehnologije u budućnosti biti korisno u cijelom spektru IT profesija na što već i ukazuju neki od preduvjeta za posao na tržištu rada unutar IT-a.

# 2. Korištene metode i alati

Tema ovog rada, ujedno i glavna tehnologija koja se koristi je Docker, preciznije Docker pokretač. Docker je prvi puta predstavljen u ožujku 2013. godine. Predstavio ga je Solomon Hykes, suosnivač uz Sebastiena Pahla tadašnje dotCloud kompanije koja čiji je proizvod bio Docker. U početku namijenjen samo za rad na Linux operativnom sustavu međutim s vremenom omogućen i rad na Windowsu, sama kontrola i rad s Dockerom omogućen na više operativnih sustava uključujući Mac OS (Cleverism, 2019).

Unutar teoretskog dijela ovog rada, koristeći metodu analize, Docker tehnologija je raščlanjena na manje komponente koje će biti detaljno razrađene svaka kao zasebna cjelina, ali isto tako će biti smještena unutar same Docker tehnologije gdje će biti objašnjeno međudjelovanje danih komponenti i na taj način ćemo graditi razumijevanje već spomenute Docker tehnologije počevši od manjih dijelova prema cjelini. U praktičnom dijelu rada ćemo na konkretnoj implementaciji Docker slike kontejnera kreirati i pokrenuti nekoliko kontejnera kojima ćemo u potpunosti proći uobičajeni postupak dockeriziranja aplikacije.

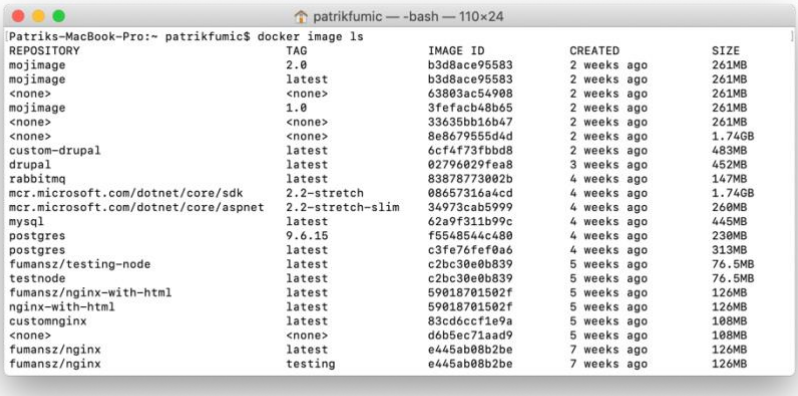
Docker izdanje koje se koristi za potrebe ovog rada je dostupno na službenoj Docker stranici. Radi se o „Comunity Edition“ izdanju koje je u potpunosti besplatno za korištenje, dok se „Enterprise Edition“ vrsta izdanja naplaćuje. Konkretna verzija o kojoj se radi je Docker desktop community za Mac. Docker desktop je kompletna okolina za Mac ili Windows računala. Integrirana Docker platforma koja omogućuje razvoj, uklanjanje grešaka i testiranje takozvanih dokeriziranih aplikacija. Radi se o iznimno jednostavnoj instalaciji koja samostalno integrira cijeli Docker pokretač s Mac operativnim sustavom uključujući Mac Hypervisor razvojnim okvirom, mrežnim sustavom i datotečnim sustavom. Zbog toga što Docker pokretač ne može raditi samostalno na Mac operativnom sustavu pokreće se prilagođena minimalna verzija Linux distribucije uz pomoć ugrađene Mac virtualizacije.


Slika 1. Docker Desktop

U nastavku ovog dijela će biti navedene ostale potrebne komponente, kao i njihovi kratki opisi, za izradu ovog rada. Iako je temelj svega Docker, točnije Docker pokretač, on sam za sebe nije dovoljan. Potrebni su nam vanjski resursi kao što je operativni sustav na kojemu će biti instaliran, program koji ćemo koristiti za upravljanje samim Dockerom i tekst editor koji će nam omogućiti kreiranje i uređivanje takozvanih Docker datoteka o kojima će biti više riječi u narednim poglavljima.

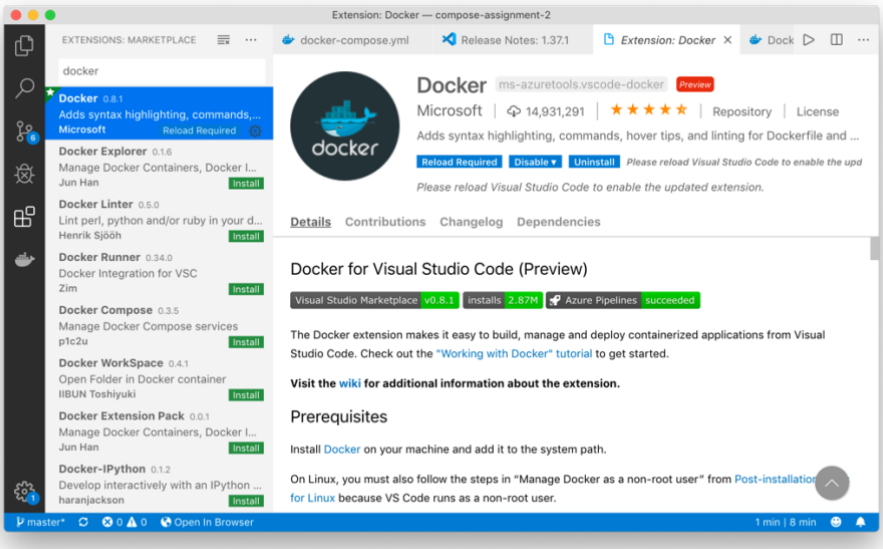
Operativni sustav koji se koristi za potrebe ovog rada već je otkriven na prethodnim stranicama, ovdje će biti navedeno nešto više informacija o istom. Radi se o MacOS Mojave 64 bitnom izdanju. Ujedno kao ugrađeni dio Mac operativnog sustava dolazi Terminal. Terminal je sučelje preko upravljačke linije koje se temelji na Unix ljusci, u našem slučaju radi se o izdanju pod nazivom Bash. Unix ljuska je interpreter upravljačke linije. Pojednostavljeno, ono što nam Terminal omogućuje je upravljanje računalom i raznim drugim alatima koji su instalirani na danom računalu putem tekstualnih naredbi koje rezultiraju u izvršenoj akciji i u nekim slučajevima povratnim informacijama u obliku teksta. Za razliku od uobičajenog grafičkog sučelja, kojeg pružaju svi moderni operativni sustavi, ovakav način rada u početku se može činiti nejasnim, kompliciranim i nedovoljno preglednim, međutim nakon nešto

detaljnijeg upoznavanja s radom Terminala lako je zaključiti da se radi o alatu koji omogućuje izravan i jasan pristup rješenjima određenih zadataka. Terminal će se koristiti u svrhe izvršavanja Docker naredbi kao što su kreiranje Docker slika, kontejnera, mreža i slično.




Slika 2. Terminal

Odabrani uređivač teksta za ovaj rad je Visual Studio Code. Radi se o besplatnom alatu izdanom od strane Microsofta. Iznimno lagan alat koji je kompatibilan sa svim operativnim sustavima. Razlog odabira ovog alata ispred ostalih su njegove bogate već ugrađene mogućnosti, ali i prisutnost takozvanih ekstenzija koje su iznimno moćne i u ovom slučaju koristit ćemo službenu ekstenziju pod nazivom Docker koja će nam olakšati rad s docker datotekama, ali i u radu s Dockerom općenito.




Slika 3. Docker ekstenzija, Visual Studio Code

# 3. Detaljnije o Dockeru

Docker tehnologija kao jedan od načina pakiranja aplikacija i svih potrebnih komponenata za ispravan rad tih aplikacija se u nadolazećem poglavlju razrađuje u usporedbi s virtualnim strojevima. Važno je razumjeti u potpunosti kako Docker tako i uobičajene virtualne strojeve i načine na koji se ostvaraju. Važno je upravo zbog toga što će na taj način biti jasno istaknute neke od prednosti i nedostatak Dockera u odnosu na postavljanje aplikacijskih rješenja na virtualne strojeve, način koji smatramo standardom isporuke aplikacija. U nastavku ovog poglavlja će biti jasno predočena razlika između virtualnih strojeva i Dockera, a nakon toga će biti više riječi o Docker kontejnerima i ostalim komponentama samog Dockera kako bi dobili potpun uvid u način na koji se ostvaruju i na koji rade Docker kontejneri. Na kraju, uz pomoć prikupljenih informacija u ovom poglavlju, ćemo objasniti kakvu ulogu ima korištenje Docker tehnologije kod razvoja i isporuke transakcijskih sustava.

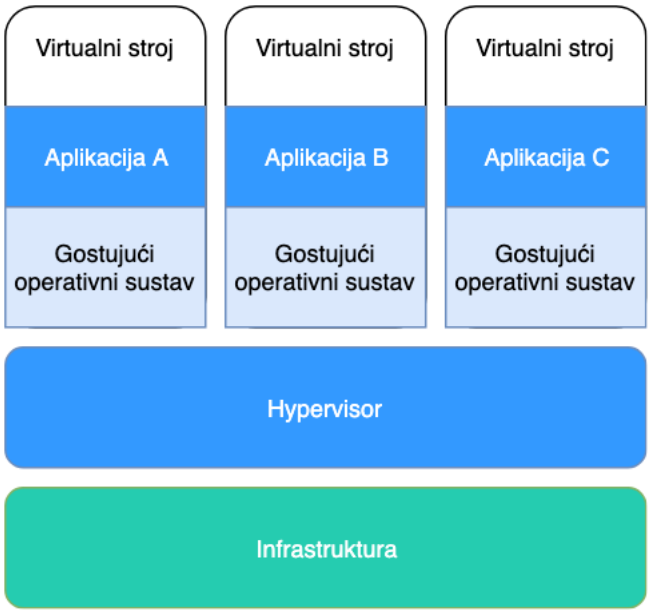
## 3.1. Docker kontejneri i virtualni strojevi

Docker kontejneri, u nastavku navođeni samo kao kontejneri, i virtualni strojevi imaju slične zadatke, a to je izolirati aplikaciju i sve potrebne pripadajuće komponente dane aplikacije u kompletne jedinice koje mogu biti pokrenute bilo gdje. Uz navedenu funkcionalnost kontejneri i virtualni strojevi također smanjuju potrebnu za hardware komponentama tako da omogućuju paralelno izvršavanje više zasebnih jedinica na jednom računalu i na taj način povećavaju efikasnost računala koje se očituje u smanjenim potrebama za hardwareom i samim time manjom potrošnjom energije i u konačnici rezultira manjim ukupnim troškovima. Bez obzira na sličnosti zadataka koje izvršavaju postoje velike razlike između kontejnera i virtualnih strojeva koje ćemo razjasniti u nastavku (Bauer, 2018).

### 3.1.1. Virtualni strojevi

Virtualni stroj je najjednostavnije rečeno sloj softvera koji se predstavlja kao stvarno računalo u njegovoj cijelosti. Taj virtualni stroj izvršava aplikacije upravo kao pravo računalo. Virtualni strojevi funkcioniraju uz pomoć takozvanog hypervisora. Hypervisor može biti softver, firmware ili hardver na kojemu se izvršavaju virtualni strojevi. Sam hypervisor se nalazi na stvarnom računalu koje još nazivamo računalom domaćinom. Računalo domaćim daje svoje resurse na raspolaganje virtualnim strojevima kao što su RAM i CPU. Ovi resursi mogu biti podijeljeni između virtualnih strojeva na bilo koji način koji smatramo adekvatnim za određenu primjenu. Ukoliko na jednom virtualnom stroju očekujemo da će aplikacija koja će se izvršavati biti iznimno zahtjevna, unaprijed ćemo osigurati više resursa upravo tom virtualnom stroju u obliku većeg udjela računalne memorije u obliku više gigabajta ili većeg broja jezgri naše središnje procesorske jedinice (Golden, 2008).

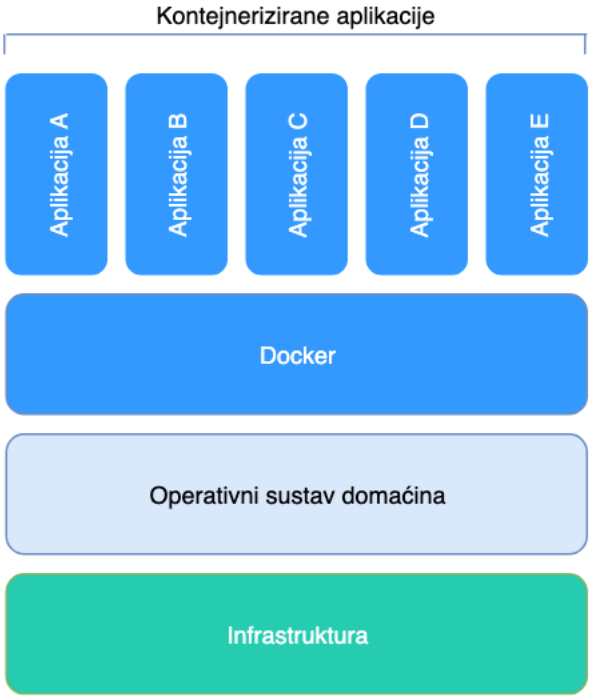
Virtualni stroj koji se izvršava na računalu domaćinu još se naziva i gostujuće računalo. Takav virtualni stroj sadrži aplikaciju i sve ostale potrebne elemente za ispravno izvršavanje već spomenute aplikacije kao što su razne biblioteke i binarne datoteke operativnog sistema. Ono što je važno naglasiti je da isto to gostujuće računalo sadrži cijeli virtualizirani skup hardvera, uključujući mrežne priključke, hardver za skladištenje podataka i središnju procesorsku jedinicu. Na taj način prividno za aplikaciju koja se izvršava virtualni stroj djeluje kao samostalno računalo. Isto tako je moguće na jednom računalu domaćinu pokrenuti više zasebnih gostujućih računala od kojih će svaki prividno biti zasebna jedinka, a resursi računala domaćina će biti raspoređeni između istih (Smith & Nair, 2005).




Slika 4. Virtualni stroj shema (Bauer, 2018)

### 3.1.2. Docker kontejneri

Za razliku od virtualnih strojeva Docker kontejneri pružaju apstrakciju na razini aplikacija u odnosu na virtualne strojeve koji pružaju apstrakciju na razini hardvera. Kontejneri se čine kao virtualni strojevi na više načina. Oni imaju vlastita mrežna sučelja i vlastite IP adrese, mogu postavljati datotečne sustave i slično. Kontejneri su znatno manji u veličini od virtualnih strojeva, uobičajena veličina je nekoliko desetaka megabajta i moguće ih je puno više pokrenuti na istom računalu domaćinu. Ti isti kontejneri dijele jezgru operativnog sustava na način da se svaki od njih izvršava kao zasebni proces unutar vlastitog korisničkog prostora (Docker, 2019b).

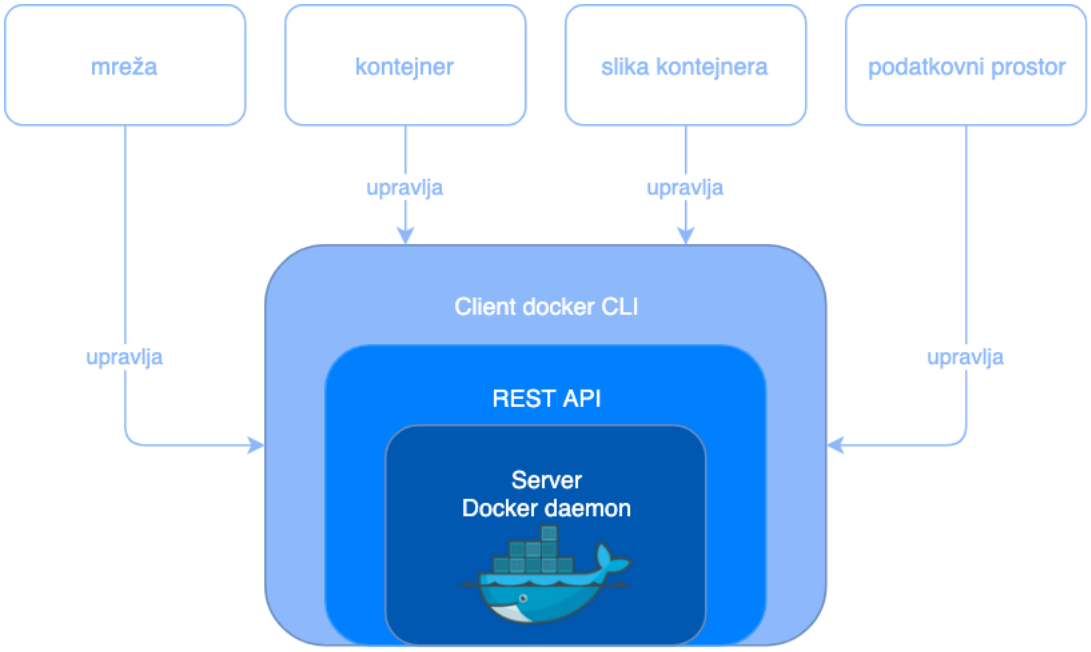



Slika 5. Docker kontejner shema (Bauer, 2018)

Oblici na dijagramu koji sadrže unutar sebe aplikacije predstavljaju kontejnere. Kao što je vidljivo iz samog dijagrama, svaki kontejner dobiva svoj vlastiti korisnički prostor i na taj način dobivamo mogućnost pokretanja više zasebnih kontejnera na jednom računalu domaćinu. Također vidljivo je da se operativni sustav domaćina dijeli sa svima kontejnerima, odnosno stavljen im je na raspolaganje. S obzirom da se operativni sustav dijeli, jedino što kontejneri sadržavaju su potrebne binarne datoteke, biblioteke i naravno aplikacije koje želimo izvršavati unutar kontejnera. Još detaljniji pogled unutar kontejnera će biti razrađen u nastavku ovog pogleda u sklopu razlaganja komponenta Dockera.

## 3.2. Docker pokretač

Docker pokretač (eng. Docker engine) je sloj na kojemu se Docker izvršava. Radi se o klijent – server tipu aplikacije koja upravlja kontejnerima, slikama kontejnera, izdanjima, mrežom i još nekolicinom elemenata koji se koriste pri pokretanju Docker kontejnera. Docker pokretač se sastoji od tri glavne komponente. Prva komponenta je server koji se izvršava na računalu domaćinu pod nazivom Docker pozadinski proces. Druga komponenta je REST aplikacijsko programsko sučelje koje sadrži specificirana sučelja putem kojih ostale aplikacije mogu komunicirati s pozadinskim procesima i izdavati upute istom. Treća i posljednja komponenta je Docker klijent odnosno 'docker' naredba koja se koristi u Terminalu preko upravljačke linije. Radi se o klijentu uz pomoć kojega komuniciramo s Docker pozadinskim procesima kako bi izvršili naredbe (Docker, 2019a).

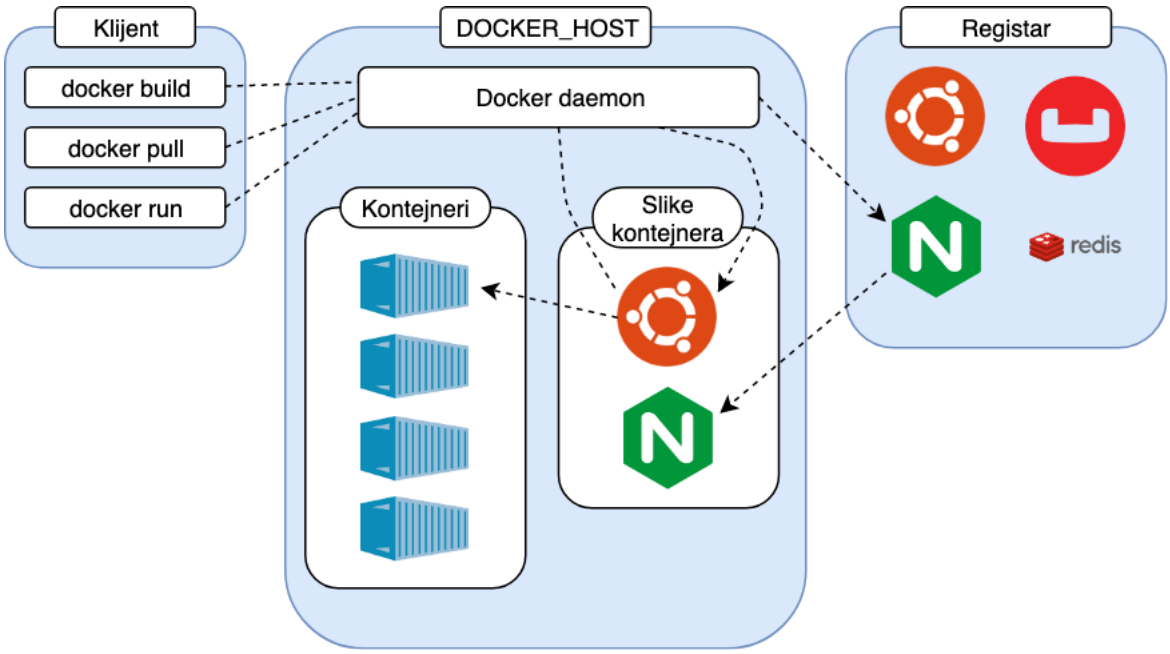



Slika 6. Docker pokretač shema (Docker, 2019a)

Sučelje preko upravljačke linije koristi REST aplikacijsko programsko sučelje kako bi upravljalo Docker pozadinskim procesima preko skripti ili izravno unesenim naredbama putem sučelja preko upravljačke linije. U slučaju vanjskih aplikacija koje izvršavaju naredbe u Dockeru oni to izvršavaju na način da se izravno spajaju na aplikacijsko programsko sučelje.

## 3.3. Docker arhitektura

Kao što je već bilo navedeno na prethodnim stranicama rada, Docker koristi klijent – server arhitekturu. Docker klijent koristi Docker pozadinske procese koji su zaduženi za sve važne akcije kao što su sastavljanje, pokretanje i distribuiranje Docker kontejnera. Docker pozadinski procesi se ne moraju nužno nalaziti na istom računalo kao i Docker klijent, postoji i mogućnost upravljanja udaljenim Docker pozadinskim procesima (Docker, 2019a).




Slika 7. Docker arhitektura shema (Docker, 2019a)

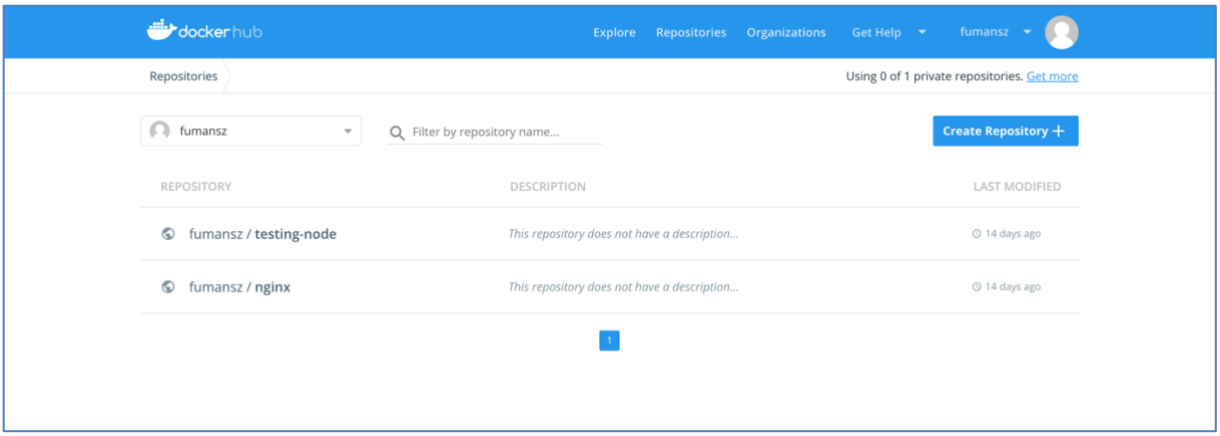
Na dijagramu je prikazane tri komponente i njihovo međudjelovanje. Jasno vidimo kako klijent šalje naredbe prema Docker pozadinskim procesima koji se nalaze unutar DOCKER_HOST komponente i koja je dalje zadužena za izvršavanje tih istih naredbi. Unutar DOCKER_HOST komponente se nalaze još slike kontejnera i sami kontejneri o kojima će biti više riječi u nastavku ovog poglavlja. Zadnja komponenta koja je preostala na dijagramu je registar koji u sebi sadrži mnoštvo slika kontejnera koje se mogu preuzeti za korištenje i daljnje izmjene.

Docker pozadinski procesi čekaju naredbe od strane Docker aplikacijskog programskog sučelja i upravljaju Docker objektima kao što su slike, kontejner i mreže. Kao što je već navedeno na početku ovog poglavlja Docker pozadinski procesi se mogu nalaziti na računalu domaćinu, ali isto tako se mogu nalaziti na nekom drugom računalu. Bez obzira gdje se nalazi korisnik nikada ne komunicira izravno s njim, nego se ona odvija preko Docker klijenta.

Docker klijent je ono što krajnji korisnik koristi za komunikaciju s Dockerom. U našem slučaju je to aplikacija Terminal uz pomoć 'docker' naredbe. Međutim to može biti bilo koja

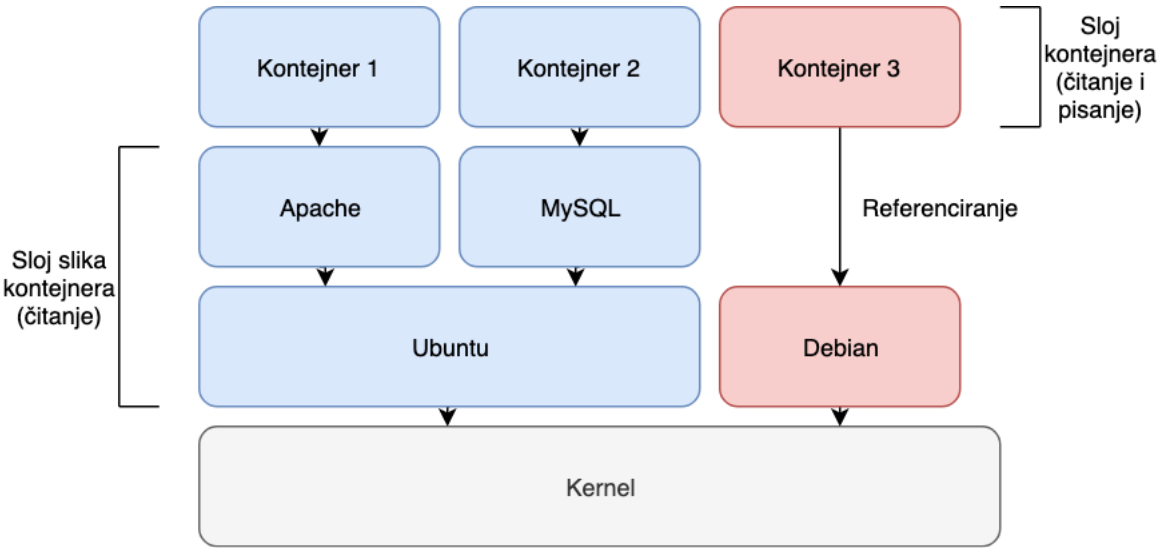
druga aplikacija koja nam omogućava komunikaciju s Dockerom. Na klijenta možemo gledati kao sučelje s Dockerom.

Ono što je do sada poznato je da se Docker kontejneri temelje na slikama za kontejnere. Docker registri su mjesto na kojemu se nalazi mnoštvo pohranjenih slika za kontejnere i oni su javno dostupni svima. Docker je izvorno postavljen tako da traži slike za kontejnere u Docker repozitoriju. Postoji i mogućnost pokretanja vlastitih registara. Također je moguće postaviti svoje u potpunosti prilagođene ili izmijenjene postojeće slike kontejnera na servis kako bi bile dostupne ostalima. Uz pomoć naredbe 'docker pull' se dohvaća tražena slika kontejnera s postavljenog registra, a uz pomoć naredbe 'docker push' se postavlja odabrana slika na odabrani registar.




Slika 8. Primjer vlastitog repozitorija slika kontejnera

Docker koristi Union datotečni sustav kako bi kreirao sliku kontejnera. Datotečni sustav o kojemu govorimo možemo razmatrati kao stog, što znači da datoteke i folderi različitih datotečnih sustava mogu biti stavljeni jedni na druge i na taj način kreirati jedan datotečni sustav.




Slika 9. Union datotečni sustav shema

Drugim riječima, zbog održavanja datotečnih sustava ponekada je logično držati određene setove datoteka na odvojenim lokacijama, ali ih je korisniku potrebno predstaviti kao jedinstvenu cjelinu. Upravo to nam omogućava Union datotečni sustav koji dozvoljava da datoteke budu fizički na različitim lokacijama na mediju na kojemu se nalazi, ali u isto vrijeme su logički spojeni u obliku jednog pogleda. Logički skup spojenih foldera se naziva **unija**, a svaki fizički direktorij se još naziva i **grana**. Ono što nam to znači u pogledu performansi i zauzeća medija na kojemu se podaci nalaze je to da se sadržaji direktorija, koji se nalaze unutar preklopljenih grana koje sačinjavaju jedan **stog**, prikazuju kao jedan spojeni direktorij. To znači da nemamo potrebu kreirati kopije pojedinih slojeva, nego uz pomoć pokazivača možemo pristupiti istima i na taj način sačinjavati logičke cjeline. U pogledu vezanom izravno za Docker postoje dvije velike prednosti. Prva je ta da ne postoji potreba za dupliciranjem cijelog seta datoteka svaki put kada se kreira i pokreće novi Docker kontejner. Na taj način je kreiranje Docker kontejnera brz postupak bez da zauzima mnogo resursa. Druga prednost je to što su slojevi odvojeni i ukoliko je potrebno napraviti izmjene na određenom sloju i kasnije ih ažurirati na određenim Docker kontejnerima, samo će se ažurirati onaj sloj koji je promijenjen što uvelike ubrzava i olakšava ovaj postupak (Wright, 2004).

Zadnji element koji je potrebno navesti je Docker datoteka. Docker datoteka je mjesto gdje korisnik unosi sve potrebne instrukcije koje će kasnije sačinjavati skriptu koju možemo izvršiti više puta kako bi kreirali Docker sliku. Jednom kad smo unijeli sve naredbe koju su potrebne za kreiranje naše slike, uz pomoć 'docker build' naredbe kreiramo željenu Docker sliku.

## 3.4. Docker kontejneri kao objekti

Docker slike kontejnera možemo razmatrati kao klase u objektno orijentiranom programiranju. Klase nam služe kao nacrti za instanciranje mnoštva objekata i ujedno imaju mogućnost nasljeđivanja ostalih klasa kako bi na temelju već postojećih klasa proširivanjem istih dobili prilagođene klase (Black, 2013). Isto tako Docker slike služe kao nacrti za kontejnere koje ćemo kreirati i iz jednog nacrta postupkom instanciranja moguće je kreirati više objekata, u ovom slučaju Docker kontejnera. Docker slike se često temelje na drugim Docker slikama i proširuju iste na isti način kako klase nasljeđuju druge klase i proširuju njihove funkcionalnosti. Ova mogućnost olakšava kreiranje prilagođenih Docker slika na način da ne moramo krenuti od prazne slike već možemo svoju prilagođenu Docker sliku temeljiti na već odgovarajućoj postojećoj koju možemo preuzeti s Docker repozitorija. Ovaj postupak možemo definirati u Docker datoteci i kasnije ga izvršiti kako bi dobili željenu Docker sliku. Svaka instrukcija u Docker datoteci kreira jedan sloj u slici i kod promjene unutar Docker datoteke i

ponovnog kreiranja izmijenjene slike samo se izvršavaju promjene na onom sloju na kojemu su primijenjene, ostali slojevi ostaju netaknuti. Ovaj dio tehnologije omogućava slikama njihovo iznimno brzo izvršavanje i malu potrebu za prostorom u odnosu na druge načine virtualizacije (Docker, 2019a).

U odnosu na Docker sliku, Docker kontejner predstavlja objekt koji se instancira na temelju klase koja ga definira, odnosno na temelju Docker slike. Svaki kontejner predstavlja zaseban objekt i u pravilu su međusobno izolirani jedni od drugih i od svojih računala domaćina ukoliko nije definirano drugačije. Kontejner baš kao i objekt evidentira promjene unutar sebe i ukoliko se jedan od kontejnera ukloni, promjene na tom kontejneru, u odnosu na sliku kontejnera na temelju koje je nastao, neće biti nigdje vidljive i neće im se više moći pristupiti.

Ovime smo zaključili teoretski dio vezan za Docker tehnologiju. Objasnili smo općenitu ulogu Dockera i detaljnije obradili segmente koji u potpunosti sačinjavaju Docker tehnologiju. U nastavku ćemo obraditi transakcijske sustave i smjestiti ih u kontekst Docker tehnologije, odnosno objasniti ćemo kako točno Docker tehnologija ima ogroman potencijal unaprijediti već spomenute transakcijske sustave i prikazati na konkretnom primjeru detaljan postupak same primjene Dockera.

# 4. Docker tehnologija u transakcijskim sustavima

U prethodnom poglavlju smo obradili Docker tehnologiju, objasnili upotrebu iste i sada smo spremni dublje ući u samu tematiku ovog rada, a to ćemo postići na način da ćemo pojasniti ulogu već spomenute Docker tehnologije unutar transakcijskih sustava. U početku ovog poglavlja ćemo pobliže objasniti same transakcijske sustave kako bi bilo lakše razumjeti zašto se koriste određeni pristupi koji će biti navedeni u nastavku. Nakon toga ćemo se na primjeru globalne tvrtke Vise upoznati kako se primjenjuje Docker unutar tvrtke koja se specijalizira u upravljanju transakcijama gdje ćemo objasniti način na koji se cjelokupan postupak razlaže na manje dijelove kojima je kasnije lakše upravljati i održavati ih. Docker ovdje ima veliku ulogu i rezultati koje pruža značajno mogu unaprijediti same transakcijske sustave.

## 4.1. Općenito o transakcijskim sustavima

Postoji više definicija transakcijskih sustava. Definicija kojom ćemo se voditi u ovom radu je sljedeća. Transakcijski sustav je sustav koji se sastoji od skupa predefiniranih operacija koje se izvršavaju u stvarnom svijetu, u obliku prijenosa konkretnih sredstava, ali i na razini aplikacije u obliku zapisa unutar baze podataka. Izvršavanje operacija rezultira promjenom stanja u aplikaciji (Grey & Reuter, n.d.). Gray u svom radu "The Transaction Concept: Virtues and Limitations" (1981) navodi četiri važna svojstva (Gray, 1981). Svojstva su sljedeća:

- **Atomarnost** – Većina transakcija se sastoji od više operacija, ovo svojstvo osigurava da se svaku transakciju tretira kao zasebnu jedinicu, što znači da postoje samo dva moguća ishoda, a to je da se sve operacije uspješno izvrše i izvršena transakcija se evidentira kao promjena stanja unutar aplikacije ili da jedna ili više operacija ne uspije i cijela transakcija se poništi.
- **Konzistentnost** – Ovo svojstvo osigurava da svaka transakcija može samo uzrokovati prijelaz iz jednog ispravnog stanja u drugo ispravno stanje. Svaki zapis u bazu podataka može biti izvršen samo ako zadovoljava sve predefinirane zahtjeve koji osiguravaju ispravnost podataka u već spomenutoj bazi podataka.
- **Izoliranost** – Moderni transakcijski sustavi izvršavaju istovremeno velik broj transakcija. Ovo svojstvo osigurava da je krajnji rezultat svih izvršenih transakcija jednak onome koji bi bio u slučaju da su se sve transakcije izvršavale sekvencijalno.
- **Trajnost** – Jednom kada je transakcija izvršena ovo svojstvo osigurava da promjene koje su nastale kao rezultat transakcije ostanu prisutne kao takve unutar sustava bez obzira na vanjske utjecaje, čak i u slučaju privremenog pada sustava.

Navedena svojstva su ključan faktor u realiziranju funkcionalnog transakcijskog sustava. Radi se o svojstvima koja osiguravaju ispravnost svake proveden transakcije čak i u slučaju pojave neočekivanih događaja kao što su pad sustava ili gubitak struje i kao takva njihova provedba je neizostavan dio transakcijskog sustava.

## 4.2. Svojstva transakcijskih sustava

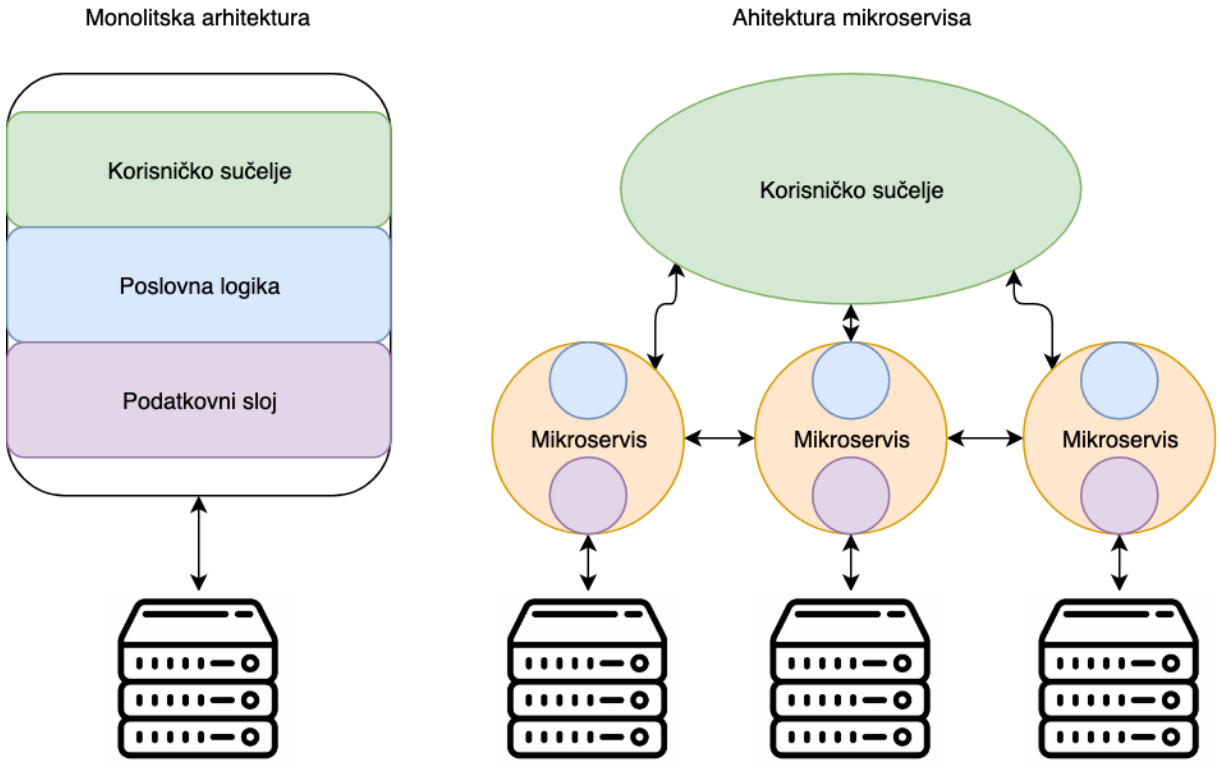
Transakcijski sustavi prikupljaju, pohranjuju, izmjenjuju i dohvaćaju podatke vezane za transakcije određenih subjekata. Sama transakcija je akcija koja stvara novo stanje ili mijenja postojeće stanje unutar baze podataka. Svaki transakcijski sustav treba imati sljedeća svojstva kako bi bio uspješan u svojoj primjeni. Potrebno je **brzo izvršavanje** dane transakcije kako krajnji korisnici ne bi bili prisiljeni čekati da sustav odgovori na njihov zahtjev. Iduće svojstvo koje je od ključne važnosti je **pouzdanost**. I najmanja nepravilnost u sustavu može uzrokovati lančanu reakciju neispravnih unosa i na taj način napraviti ogromnu financijsku štetu, ali i uzrokovati ostale poteškoće subjektima koji su u pitanju. **Rigidnost** je sljedeće svojstvo koje svaki transakcijski sustav mora ostvarivati. Želimo da svaka transakcija bez obzira na korisnika ili vrijeme dana bude obrađena na isti način, što znači da se uvijek izvršava čvrsto definiran skup operacija potrebnih za uspješno kompletiranje transakcije. Zadnje svojstvo koje transakcijski sustav mora zadovoljavati je **kontrolirano izvršavanje**. To znači da ukoliko postoje definirane uloge i prava sustav se mora striktno pridržavati istih. Dopuštati određenim ulogama samo djelovanje unutar njihove domene i korištenje samo njima dodijeljenim resursima (Powers, 2000).

## 4.3. Visa kao primjer implementacije Docker tehnologije

Postoji mnoštvo načina na koji se Docker tehnologija može primjenjivati unutar informacijskih sustava, međutim smjer u kojemu će ići nastavak ovog rada će biti temeljen na primjeru Vise i njihovom načinu razvoja vlastitog poslovanja. Visa je američka kompanija nastala 2007. godine koja se specijalizira u elektroničkom plaćanju diljem cijelog svijeta (Reuters, 2019). Radi se o jednoj od najpoznatijih svjetskih kompanija koja implementira vlastiti transakcijski sustav uz pomoć Docker tehnologije. Isti taj sustav godišnje obradi 130 milijardi transakcija ukupnog iznosa 5.8 trilijuna američkih dolara.

Glavni cilj prije prelaska na Docker tehnologiju im je bio poboljšati dva segmenta unutar svog poslovanja, a to su brzina i efikasnost. Način na koji će se raditi na poboljšanju dva spomenuta segmenta je da se uz pomoć Docker tehnologije, dosad monolitska aplikacija,

prebaci na mikroservisni aplikacijski model, o kojemu će biti više riječi u nastavku ovog poglavlja.




Slika 10. Odnos monolitske arhitekture i arhitekture mikroservisa

Docker omogućava zakazivanje izvršavanja određenih akcija u određenom trenutku i umrežavanje kontejnera, što je nužno kod kreiranja međusobno povezanih mikroservisa. Također kao i kod svakog drugog većeg projekta za razvoj softvera potrebno je kreirati različite okoline i različite verzije danog softvera kako bi razvoj bio što efikasniji, ali ujedno ograničiti pristup danim okolinama i verzijama na temelju pojedine uloge. Sve su ovo zahtjevi koje Docker omogućava u svom korištenju. Ovi zahtjevi će također biti obrađeni u praktičnom dijelu rada gdje će biti prikazan konkretan postupak kreiranje različitih okolina i sigurnosnih mehanizama.

Visa je započela postupak dockeriziranja svojih aplikacija sa svoje dvije ključne aplikacije. Radi se o središnjoj aplikaciji za upravljanje transakcijama i o aplikaciji za upravljanje rizicima. Obje aplikaciju su monolitske aplikacije koje su ponovno napisane kao mnoštvo mikroservisa koji će onda naknadno biti smješteni unutar Docker kontejnera. Ove aplikacije su trenutno u produkciji, dostupne su u mnoštvu regija i dnevno obrađuju 100 000 transakcija. Spomenute aplikacije se sastoje od 100 kontejnera koji imaju mogućnost skalirati se na 800 kontejnera u slučaju vršne potražnje za uslugama. Kontejneri kao usluga i njihove mogućnosti skaliranja će biti dodatno objašnjeni u nastavku ovog poglavlja.

Glavni napredak prelaska na Docker tehnologiju je vidljiv u nekoliko područja. Prvo je puno jednostavnije i brže odgovaranje na porast zahtjeva na pojedinu ulogu, u nekoliko sekundi je moguće pokrenuti veći broj kontejnera koji će se nalaziti iza sloja koji upravlja raspodjelom zahtjeva i na taj način smanjiti opterećenje pojedinog kontejnera i omogućiti korisnicima brži odgovor na njihov zahtjev. Iduće na redu je održavanje, koje je uvelike olakšano upotrebom kontejnera. Isporuka zakrpa ili redovno održavanje je olakšano na način da je moguće nakon kreiranja nove slike kontejnera samo isporučiti danu sliku na sve potrebne okoline i na taj način imamo isporučenu novu verziju aplikacije. Još jedna od prednosti za razvojne programere je to što nakon što su aplikacije jednom smještene unutar kontejnera, više nije potrebno pojedinom razvojnom programeru poznavati infrastrukturu koja je potrebna za izvršavanje spomenute aplikacije. Potrebna infrastruktura je uključena unutar kontejnera i kao takva je u stanju spremnom za pokretanje aplikacije. Zadnje područje koje će biti navedeno je bolje iskorištavanje dostupne infrastrukture koja služi za isporuku aplikacija. Mogućnost brzog povećanja i smanjenja broja kontejnera za pojedini mikroservis omogućuje puno efikasnije korištenje dostupne infrastrukture za odgovaranje na fluktuirajuće promjene u zaprimanju zahtjeva prema pojedinim mikroservisima (Fong, 2017)(Fong, 2017).

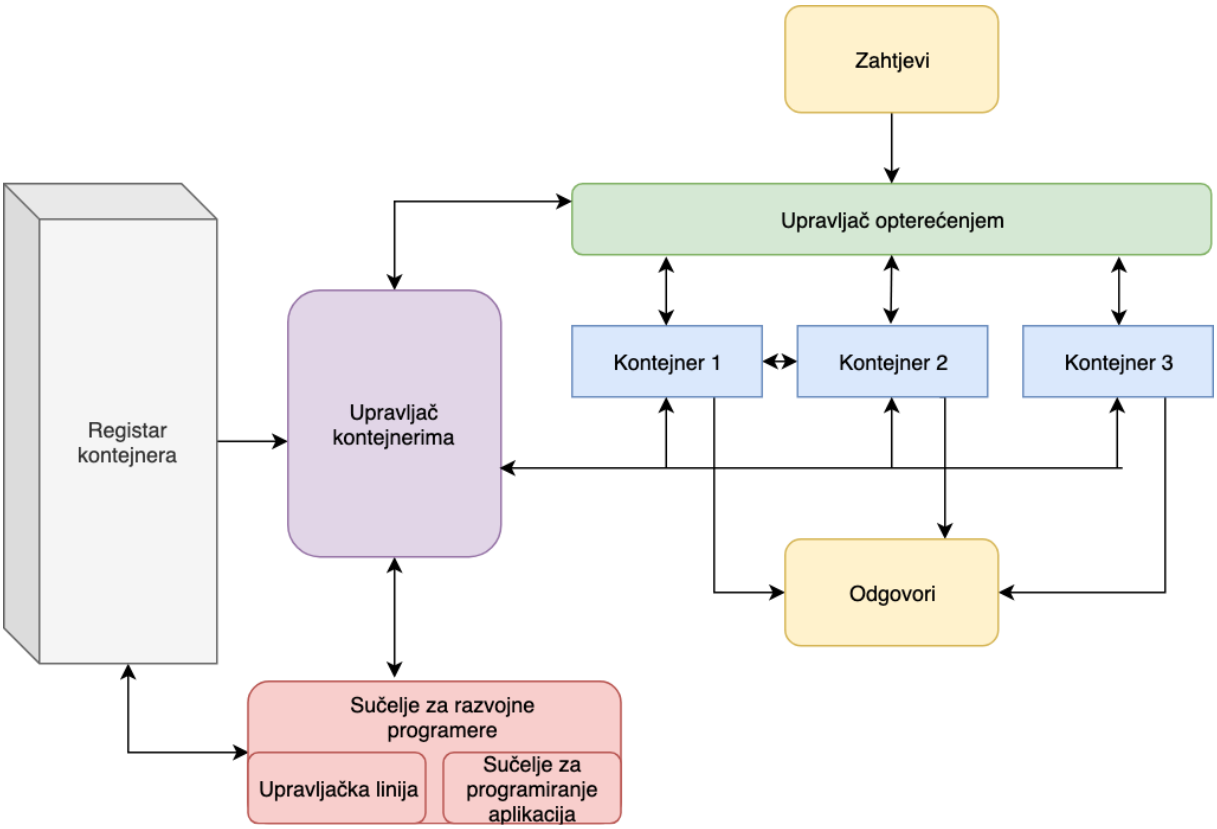
Tehnologija i postupci koji su navedeni u ovom odjeljku kao sredstva koje Visa koristi u napretku svog poslovanja će dalje biti obrađeni detaljnije u nastavku ovog poglavlja. Prvo će biti obrađeni kontejneri kao usluga odnosno engleski izraz 'Containers-as-a-Service' i u istom kontekstu će biti objašnjen postupak razvoja u obliku neprekidne integracije i neprekidne isporuke, a nakon toga i razlika monolitske aplikacije u odnosu na mikroservisni model aplikacije gdje je svaki od mikroservisa spremljen unutar vlastitog kontejnera.

## 4.4. Kontejneri kao usluga

Kao što je navedeno u prethodnom odjeljku, Visa koristi kontejnere kao uslugu u svom poslovanju, u nastavku će biti objašnjeno prema Dietrichu (2019) koja je svrha kontejnera kao usluge i koje su prednosti korištenja kontejnera kao usluge.

Kad govorimo o kontejnerima kao usluzi radi se o usluzi koja omogućava jednostavno upravljanje kontejnera između ostalog u obliku učitavanja novog sadržaja u kontejnerima, pokretanja i skaliranja kontejnera. Glavna svrha kontejnera kao usluga je to da se dobije kompletna usluga koja će nam omogućiti upravljanje kontejnerima bez da mi sami moramo postavljati okolinu koja će nam omogućiti isto. Uobičajeno je da usluga nudi sučelje za programiranje aplikacija ili sučelje upravljačke linije, dok neke od njih raspolažu i sa grafičkim sučeljem u obliku stolne ili web aplikacije. Kontejneri kao usluga uvelike olakšavaju isporuku

kontejnera. Također olakšava upravljanje već spomenutim kontejnerima, kao i konfiguriranje i upravljanje većim skupom međusobno zavisnih kontejnera.




Slika 11. Kontejneri kao usluga

Jedna od glavnih komponenta kontejnera kao usluga je registar kontejnera. Radi se o repozitoriju koji sadržava sve kreirane i učitane slike kontejnera s kojima radimo. Jednom kada su slike kontejnera učitane, sustav koji upravlja kontejnera je u mogućnosti preuzeti određenu sliku kontejnera i na temelju nje kreirati i pokrenuti traženi kontejner. Najpoznatiji registar kontejnera je Docker Hub, ali isto tako je moguće koristiti privatne repozitorije.

Druga komponenta koju je važno izdvojiti je samo upravljanje kontejnerima od kojih je jedan konkretan primjer Google Kubernetes pokretač, o kojemu neće biti previše riječi u ovom radu, ali ćemo u nastavku navesti nekoliko njegovih svojstva. Kubernetes omogućava automatizirani postupak koji pokreće kontejnere, zatim prati njihov rad i u slučaju da postoji potreba izvršava prikladne akcije. U slučaju da jedna ili više instanca prestanu odgovarati na zahtjeve ili dođe do naglog porasta broja zahtjeva prema kontejnerima, upravljanje kontejnerima koje je automatizirano će samostalno pokrenuti onoliko instanca kontejnera koliko je potrebno kako bi se izgladilo opterećenje sustava. Također kontrolira komunikaciju između pojedinih instanca kontejnera kako ne bi došlo do grešaka, kao što su pozivi određenih usluga prije nego što su u potpunosti iste pokrenute.

Zadnja komponenta koji ćemo izdvojiti su alati za razvojne programere. Kao što je već navedeno u početku ovog odjeljka kontejneri kao usluga pružaju sučelje za programiranje aplikacija i sučelje upravljačke linije. Iako je poželjno da je što veći dio upravljanja kontejnera automatiziran postupak, ponekada je potrebno da razvojni programeri ručno prate ili upravljanju samim kontejnerima i upravo uz pomoć ovih alata im je to i omogućeno (Dietrich, n.d.).

Zaključujemo da kontejneri kao usluga imaju prvenstveno kao svrhu, smanjiti opterećenje razvojnih programera i automatizirati što je više moguće upravljanje kontejnerima i na taj način uštedjeti vrijeme razvojnih programera, a na taj način unaprijediti poslovanje i smanjiti troškove korisnicima spomenute usluge.

## 4.5. Neprekidna integracija i neprekidna isporuka

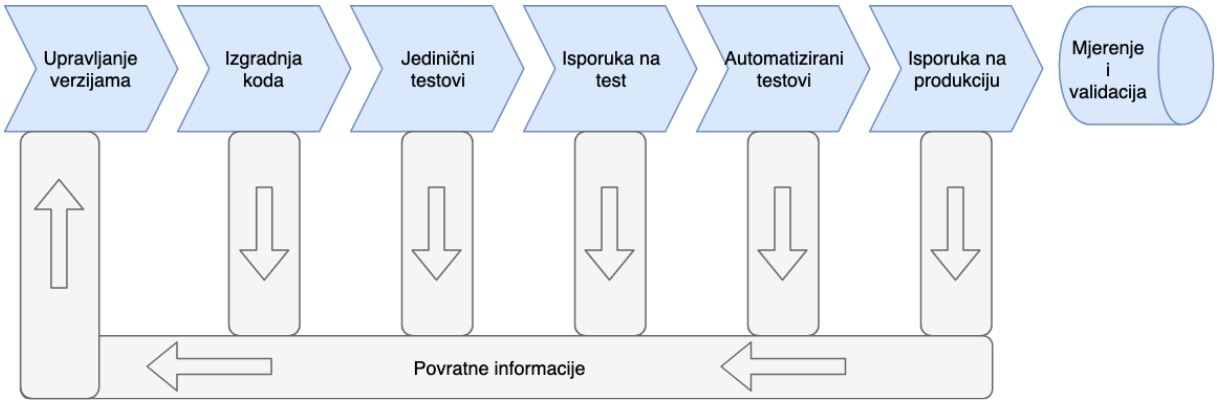
Iako trenutni odjeljak nije izravno povezan sa samom tematikom ovog rada, spominjemo ga u slučaju Vise i njihovog prelaska na Docker tehnologiju. Smatram kako je osnovno poznavanje pristupa Neprekidne integracije i neprekidne isporuke korisno iz razloga što je ovo čest pristup koji se realizira uz pomoć Docker tehnologije na kojemu možemo uvidjeti prednosti Dockera kod česte isporuke softvera.

Neprekidna integracija i neprekidna isporuka su termini za način razvoja softvera. Radi se o novom pristupu koji tjera razvojne programere da uvijek svoj kod drže u stanju spremnom za produkciju, što znači da kod treba često spajati i da je potrebna velika kohezija tima koji radi na danom zadatku. Ovo je veoma zahtjevan pristup i zato je automatiziranje pojedinih procesa ključno u ovom načinu razvoja softvera.

Neprekidna integracija označava način rada u kojem razvojni programeri spajaju svoj dovršeni kod što učestalije s glavnim kodom. Ovo je iznimno zahtjevno za razvojne programere ovisno o kompleksnosti projekta i samih tehnologija koji se koriste na spomenutom projektu. Ovaj postupak se nastoji olakšati automatiziranim postupcima. Prvenstveno se radi o testnoj okolini i automatiziranim testovima koji se nalaze na istoj koji uvelike olakšavaju ovaj postupak kako bi osigurali ispravno funkcioniranje glavnog koda koji se uvijek nastoji održati u stanju u kojemu ga je uvijek moguće pokrenuti bez većih grešaka.

Neprekidna isporuka je logičan slijed na neprekidnu integraciju. U pitanju je način rada kojim se nastoji održati glavni kod uvijek spremnim za isporuku, tako da su sve sadržane značajke funkcionalne i da je zadnja verzija dane aplikacije spremna za korištenje u toj fazi. Ovim putem se osigurava automatiziranost pakiranja i isporuke koda i ispravna uporaba metodologije neprekidne isporuke omogućava lakše dostavljanje novih značajki bez potrebe

za zaustavljanjem servisa koji ih pokreće. Ova zadnja stavka je iznimno važna za velike transakcijske sustave čije usluge uvijek moraju biti dostupne (Manturewicz, 2019).




Slika 12. Neprekidna integracija i neprekidna isporuka shema (Tuli, n.d.)

## 4.6. Mikroservisi u odnosu na monolitske aplikacije

Iako nije sasvim nov pojam u području razvoja softvera, arhitektura softvera u obliku mikroservisa tek je proteklih nekoliko počela uzimati maha. Visa je jedna od mnogih transakcijskih tvrtki koja je prepoznala potencijal u ovoj arhitekturi i na kraju je donesena odluka da se ista i implementira. Radi se o metodologiji razvoja aplikacije na način da se jedna aplikacija razloži na mnoštvo manjih servisa, odnosno usluga od kojih će svaka biti cjelina sama za sebe i biti neovisna o drugim uslugama. Iste te usluge međusobno komuniciraju putem predefiniranih sučelja najčešće putem jednostavnih protokola kao što je HTTP i na taj način sačinjavaju aplikaciju u njezinoj cijelosti.

Ovaj je pristup razvoju softvera iznimno pogodan za transakcijske sustave gdje želimo razložiti cijeli sustav na manje servise. Na ovaj način smanjujemo kompleksnost za pojedini tim razvojnih programera koji je zadužen za razvoj pojedinog servisa, također ubrzavamo proces razvoja, smanjujemo veličinu pojedine isporuke i omogućavamo efikasnije skaliranje pojedinog servisa. U nastavku ćemo detaljnije proći kroz upravo navedene prednosti arhitekture mikroservisa u pogledu transakcijskih sustava, ali i ostalih velikih monolitskih sustava.

Moderne monolitske aplikacije se najčešće sastoje od tri dijela, a to su klijentska strana, server strana i baza podatka, uobičajeno relacijska. Ovaj pristup omogućava da sva logika koja je potrebna za razrješavanje zahtjeva budi obrađena u jednom procesu što nam dozvoljava korištenje osnovnih svojstava pojedinih programskih jezika i objektno orijentirane paradigme, a to je implementiranje klasa, funkcija i imenskog prostora. Monolitske aplikacije se mogu testirati na računalu razvojnog programera uzevši u obzir da cijela aplikacija mora biti funkcionalna. Također postoji mogućnost horizontalnog skaliranja na način da se pokrene više instanca iste aplikacije i da zaprimanje zahtjeva balansiranjem opterećenja bude jednoliko raspoređeno između svih pokrenutih instanci. Monolitske aplikacije su i dalje standard i mogu biti u potpunosti uspješne, međutim kod velikih projekata javljaju se određeni izazovi koje se nastoji riješiti arhitekturom mikroservisa. Ukoliko se monolitska aplikacija mijenja u bilo kojem razmjeru ona u svojoj cijelosti mora biti ponovno izgrađena i isporučena, također cijela aplikacija mora biti rađena u jednoj tehnologiji i razvojni programeri koji rade na danoj aplikaciji u pojedinim slučajevima ovise jedni o drugima u mjeri da se posao mora čekati dok se ne usklade razine na kojima se nalazi aplikacija kako bi se moglo nastaviti s radom.

Sve navedene poteškoće se nastoje riješiti arhitekturom koja se temelji na mikroservisima. S obzirom da se radi o više manjih nezavisnih aplikacija, ukoliko dođe do potrebe za izmjenom aplikacije u cijelosti, mijenjaju se samo oni servisi odnosno usluge koji su definirani zahtjevom

i samo te usluge se ponovno izgrađuju i isporučuju. Još jedna od prednosti je to što svaka manja aplikacija koja pruža određenu uslugu može biti napisana u različitom programskom jeziku od ostalih aplikacija. Na ovaj način razvoj samog softvera može teći puno brže nego kod standardnog pristupa (Lewis & Fowler, 2014).

# 5. Primjena Docker tehnologije

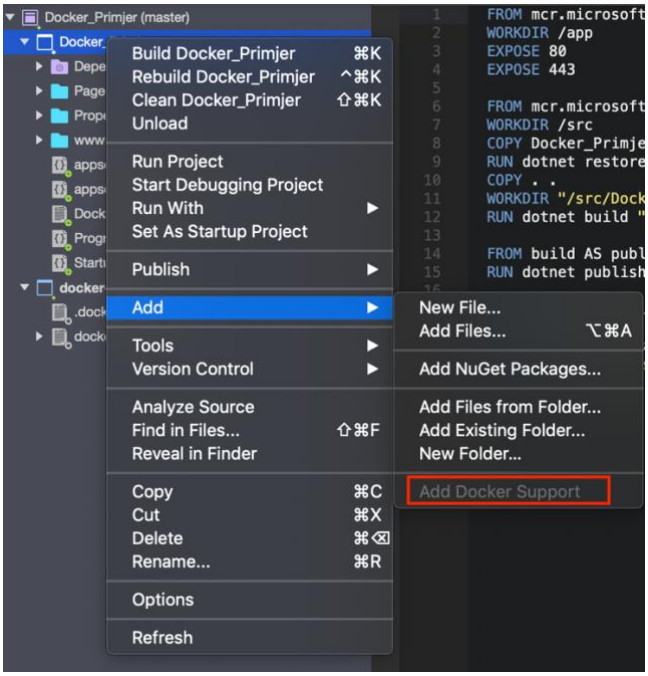
U ovom poglavlju je prikazan postupak dokeriziranja web sučelja za programiranje aplikacija unutar .NET Core tehnologije. Na kraju ćemo imati potpun kontejner koji sadrži sve što je potrebno za pokretanje jedne takve aplikacije, uz to je obrađena većina slučajeva koje je potrebno poznavati u uobičajenom postupku dokeriziranje ovakvog primjera.

## 5.1. Kreiranje .NET Core web sučelja za aplikacijsko programiranje i dodavanje Docker podrške

Ovo poglavlje ćemo započeti s uobičajenim postupkom kreiranja .NET Core web sučelja za aplikacijskog programiranje putem sučelja upravljačke linije. Nakon što smo otvorili terminal i pozicionirali se unutar željenog foldera. Izvršavamo sljedeću naredbu.


Slika 13. Kreiranje web sučelja za aplikacijsko programiranje

Izvršavanje ove naredbe nam kreira sve potrebne datoteke za početak rada vezano za .NET Core dio. Iduće je potrebno uključiti podršku za Docker. Postoji više načina između ostalog i ručno kreiranje i dodavanje Docker datoteka, međutim u ovom radu ćemo se poslužiti Visual Studio integriranom razvojnom okolinom i uz pomoć istog kreirati potrebne datoteke koje ćemo naknadno izmijeniti.




Slika 14. Dodavanje podrške za Docker

Odabirom opcije dodavanja podrške za Docker kreiraju se četiri datoteke, od kojih je jedna standardan Dockerfile, a preostale su Docker-compose datoteke. Kao što je već objašnjeno na prethodnim stranicama ovog rada Dockerfile je datoteka na temelju koje ćemo kreirati svoju prilagođenu sliku kontejnera na temelju koje ćemo kasnije kreirati naše kontejnere.

```
Dockerfile > ...
1  FROM mcr.microsoft.com/dotnet/core/aspnet:2.2-stretch-slim AS base
2  WORKDIR /app
3  ENV ASPNETCORE_URLS http://+:89
4  EXPOSE 89
5  #EXPOSE 443
6  COPY ${source:-obj/Docker/publish} .
7  ENTRYPOINT ["dotnet", "Docker_Primjer.dll"]
```

Slika 15. Dockerfile

Prvo se definira slika kontejnera na kojoj će se temeljiti buduća slika kontejnera kreirana iz ovog Dockerfilea s ključnom riječi 'FROM'. Nakon toga definiramo radni direktorij i postavljamo potrebne atribute okoline. U idućem korak otvaramo port 89 koji će predstavljati port preko kojega ćemo pristupati kontejneru. Na kraju određujemo odakle ćemo kopirati potrebne datoteke, gdje ćemo ih unutar slike prebaciti i uz pomoć 'ENTRYPOINT' naredbe postavljamo da se budući kontejneri izvršavaju kao izvršna datoteka. '.' na kraju naredbe zadužene kopiranje označava to da želimo prebaciti na trenutnu lokaciju slike unutar koje se nalazimo. Važno je napomenuti da je dobra praksa držati elemente sklone promjenama na dnu

Dockerfilea zbog načina na koji se izvršavaju promjene, a to je da se one izvršavaju od onog dijela u kodu gdje je uočena prva promjena u odnosu na staru verziju pa sve do kraja datoteke.

docker-compose.override.yml

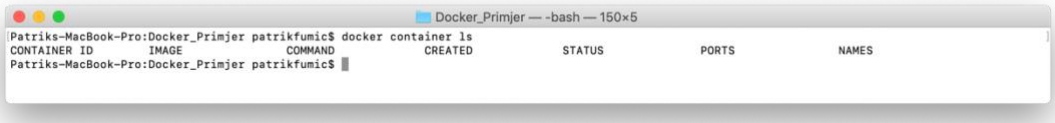
```yaml
version: '3.4'

services:
  docker_primjer:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      #- ASPNETCORE_URLS=https://+:443;http://+:89
      #- ASPNETCORE_HTTPS_PORT=44378
    ports:
      - "89"
      #- "47632:89"
      #- "44378:443"
    volumes:
      - ~/.aspnet/https:/root/.aspnet/https:ro
      - ~/.microsoft/usersecrets:/root/.microsoft/usersecrets:ro
```

Slika 16. Docker-compose datoteka

Docker-compose datoteka je datoteka YAML formata koja predstavlja konfiguracijsku datoteku za 'docker-compose' naredbu. Ova mogućnost omogućava isporuku, uparivanje i konfiguraciju više Docker kontejnera istovremeno koji najčešće zajedno sačinjavaju jednu veću funkcionalnost (Turnbull, 2019). Ponovno definiramo okolinu, otvaramo portove, zadnji dio je autogeneriran i nismo ga morali mijenjati, o samim lokacijama za pohranu podataka će biti više riječi na kraju ovog poglavlja.

## 5.2. Pokretanje aplikacije

Aplikaciju možemo pokrenuti na standardan način preko upravljačke linije uz 'dotnet run' naredbu, ali ono što mi želimo je pokrenuti našu aplikaciju unutar Docker kontejnera. Prvo ćemo provjeriti postoji li trenutno koji pokrenuti kontejner.

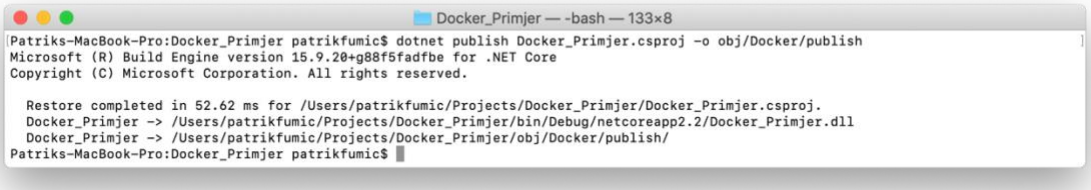



Slika 17. Provjera pokrenutih kontejnera

Uz pomoć prikazane naredbe možemo vidjeti da trenutno nemamo pokrenutih kontejnera. Ono što trebamo napraviti prije samog kreiranja slike kontejnera je objaviti našu aplikaciju unutar foldera koji smo naveli u Dockerfile kao izvorni folder. Prvi korak je pokretanje 'dotnet restore' naredbe kako bi dohvatili sve komponente o kojima ovisi naša aplikacija, ukoliko nešto nedostaje.

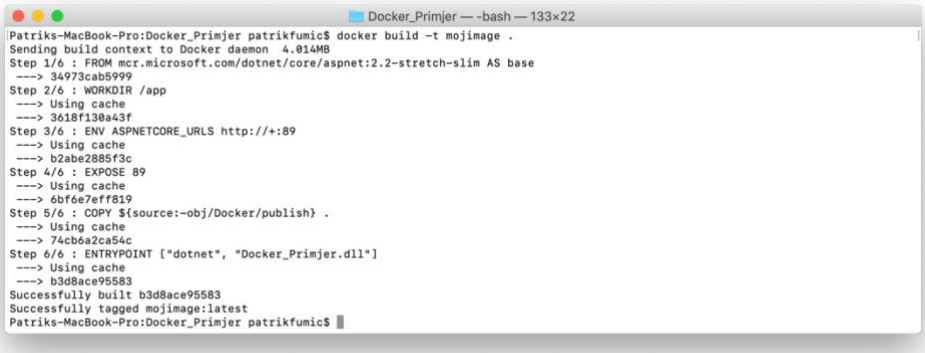

Slika 18. Povrat paketa

Nakon uspješno izvršene naredbe, sada moramo objaviti našu aplikaciju u očekivani folder, kako je prikazano na sljedećoj slici.



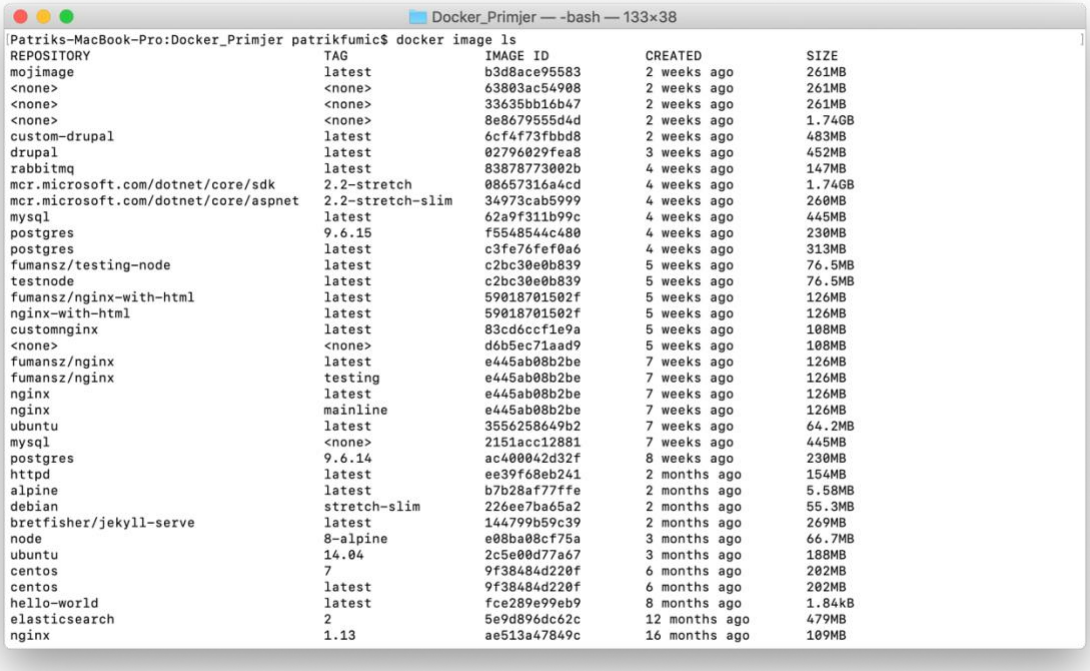

Slika 19. Objavljivanje aplikacije

Sada kada je aplikacija uspješno objavljena možemo kreirati svoju prilagođenu sliku kontejnera uz pomoć naredbe prikazane na sljedećoj slici.




Slika 20. Kreiranje prilagođene slike kontejnera

Prethodnom naredbom smo kreirali sliku kontejnera s nazivom 'mojimage' i točkom na kraju naredbe smo uputili Docker pokretač da tu sliku želimo kreirati na temelju Dockerfilea koji se nalazi unutar foldera gdje se izvršava dana naredba. Sljedeća slika će prikazati našu kreiranu sliku kontejnera na vrhu popisa.

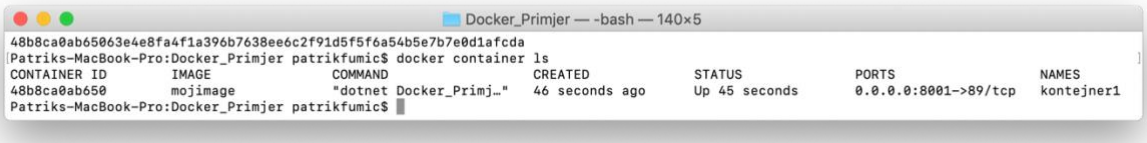



Slika 21. Prikaz kreiranih slika kontejnera.

Zadnji postupak koji nam preostaje je pokretanje kontejnera na temelju upravo kreirane slike i provjera ispravnosti danog kontejnera unutar našeg preglednika.

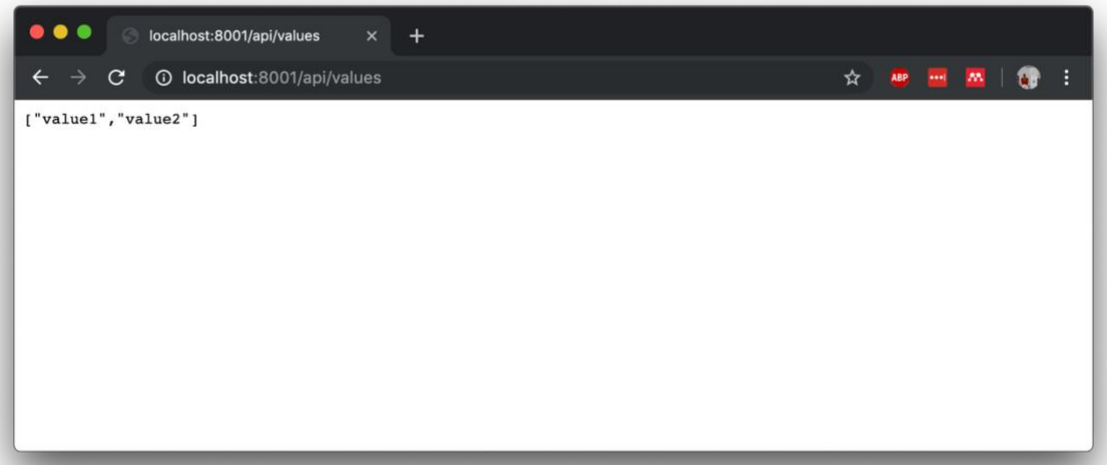

Slika 22. Pokretanje kontejnera

Kontejner smo pokrenuli na način da smo rekli da želimo da se pokrene u pozadini uz pomoć oznake '-d' i odredili smo port koji želimo koristiti na računalu domaćinu za pristup našem kontejneru, a to je port 8001 koji smo usmjerili prema portu 89 koji pripada našem kontejneru. Na kraju smo imenovali kontejner kao 'kontejner1'. Na idućoj slici možemo vidjeti potvrdu kako je naš kontejner pokrenut.



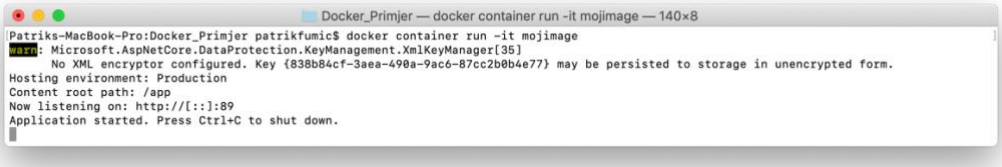

Slika 23. Uspješno pokretanje prvog kontejnera

Potrebno je još provjeriti ispravnost našeg kontejnera na način da otvorimo naš preglednik i pokušamo pristupiti localhost adresi na portu 8001 kojeg smo definirali u prethodnoj naredbi. Na sljedećoj slici možemo vidjeti da dobivamo standardan unaprijed zadan prikaz .NET Core sučelja za aplikacijsko programiranje što nam govori da je kontejner ispravno pokrenut.



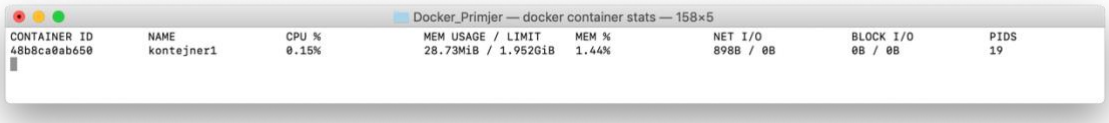

Slika 24. Prikaz rezultata pokrenutog kontejnera

Za dodatne mogućnosti rada s kontejnerom, kreiran je još jedan kontejner s nazivom 'kontejner-primjer'. Kontejner možemo pokrenuti i na način da odmah budemo pozicionirani unutar samog kontejnera uz pomoć oznake '-it' koja označava riječ interaktivno i spremni za izvršavanje naredbi u istom, kao što je prikazano na sljedećoj slici.



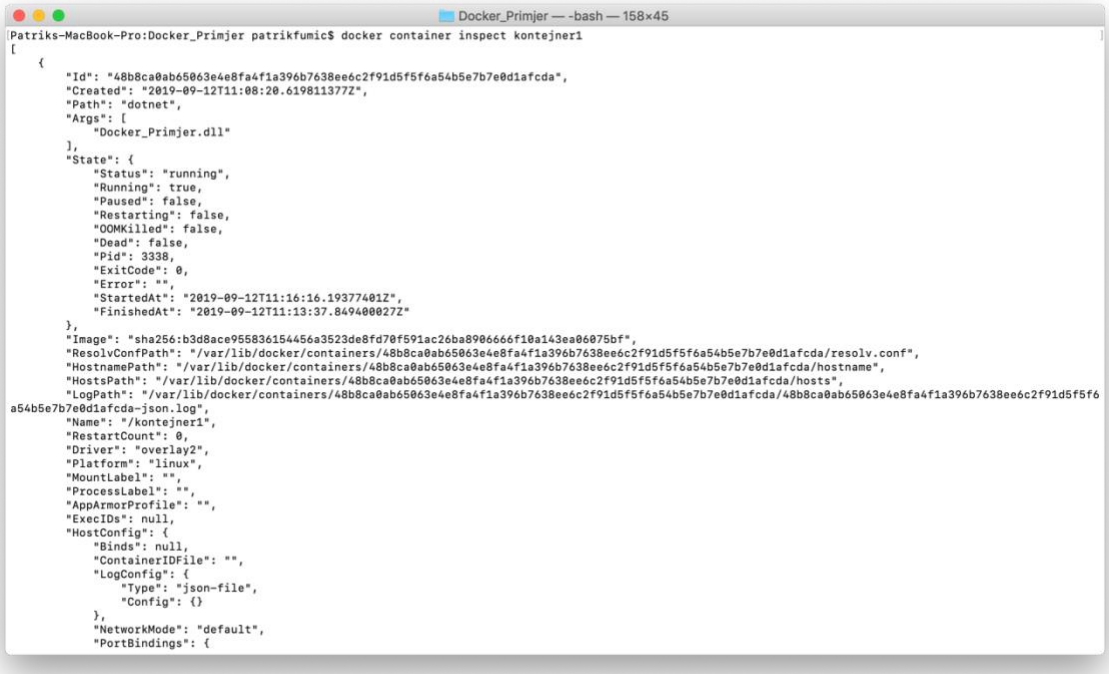

Slika 25. Interaktivno pokretanje kontejnera

Također je moguće pratiti uživo statistike danog kontejnera kako bi bili upoznati s informacijama kao što su zauzeće prostora za pohranu podataka, memorijskog prostora, procesorske i mrežne propusnosti.




Slika 26. Statistike kontejnera

Najiscrpnijem izvoru podataka vezanom za dani kontejner pristupamo na način koji je prikazan na sljedećoj slici. Neki od podataka koji su prikazani su podaci o stanju, prostorni podaci, mrežni podaci i slično.




Slika 27. Inspekcija Docker kontejnera

## 5.3. Pokretanje više kontejnera iz iste slike kontejnera

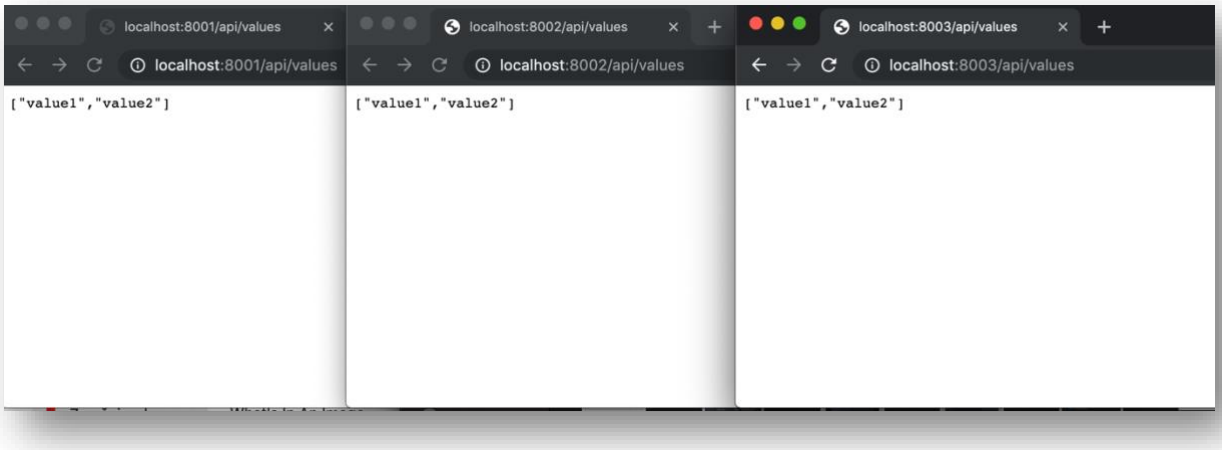
Kao što je već navedeno u prethodnim poglavljima, docker kontejneri su iznimno pogodan način isporuke aplikacija koje moraju biti skalabilne i zahtijevaju mnogo instanca pojedinih aplikacija bez obzira radi li se o standardnim monolitskim aplikacijama ili modulima unutar arhitekture mikroservisa. U ovom odjeljku ćemo prikazati pokretanje više kontejnera iz iste slike kontejnera koja kasnije možemo koristiti kod velikih opterećenja tako da ih stavimo iza sloja koji će upravljati opterećenjem i ravnomjerno usmjeravati zahtjeve. Docker kontejneri su iznimno laki što se tiče načina isporuke, nemaju velikih zahtijevanja i upravo iz tog razloga su pogodni kod velikog broja instanci.


Slika 28. Pokretanje više kontejnera iz iste slike kontejnera

Na prethodnoj slici vidimo kako smo uz pomoć iste slike kontejnera 'mojimage' pokrenuli dva kontejnera, s imenima kontejner2 i kontejner3, na zasebnim portovima 8002 i 8003. Važno je primijetiti kako je drugi port i dalje ostao 89, zato što se odnosi na pojedini kontejner koji je pokrenut, a svaki od tih kontejnera je instanca sama za sebe.


Slika 29. Prikaz više pokrenutih kontejnera iz iste slike

Uz pomoć naredbe prikazane na prethodnoj slici možemo se uvjeriti kako smo uspješno pokrenuli još dodatna dva kontejnera od kojih se oba temelje na istoj slici. Sada je samo preostalo dokazati da i preostala dva kontejnera paralelno rade i izvršavaju našu aplikaciju. Vraćamo se unutar preglednika i unosimo adrese s pripadajućim portovima preostalih kontejnera.

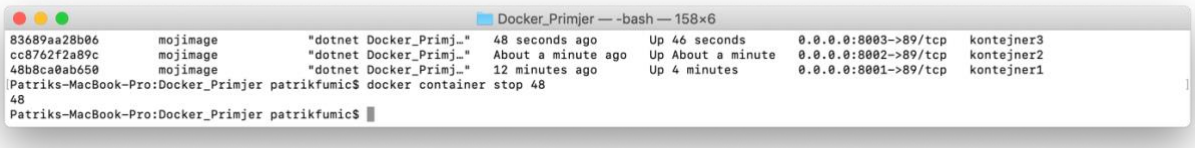



Slika 30. Prikaz paralelnog izvršavanja više kontejnera

## 5.4. Upravljanje kontejnerima

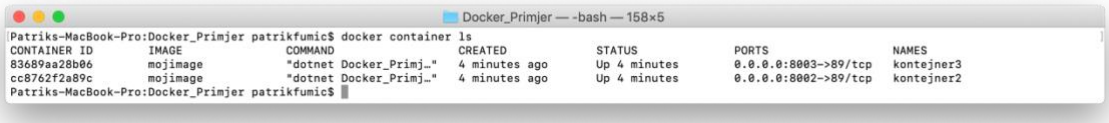
Ovaj odjeljak je namijenjen upravljanju kontejnerima. Prikazat ćemo kako pokrenuti, zaustaviti, obrisati određene kontejnere i kako prikazati sve kontejnere koji su spremni biti pokrenuti.

Jednom kada smo kreirali kontejner nije nužno da taj isti kontejner cijelo vrijeme radi. Isto tako jednog kada ga zaustavimo s radom to ne znači da je potrebno ponovno kreirati takav kontejner, već je moguće samo ponovno pokrenuti već postojeći.



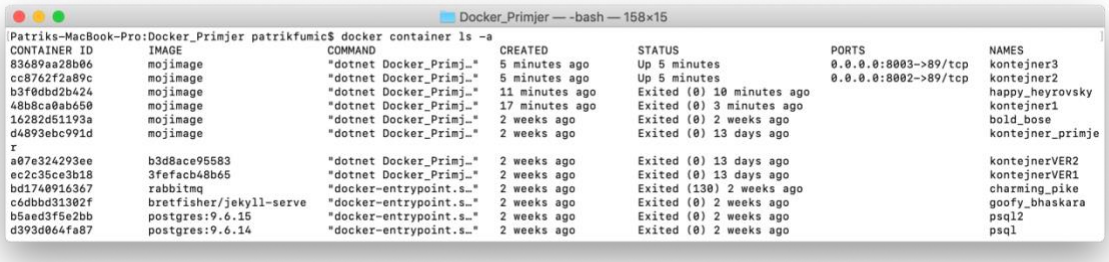

Slika 31. Zaustavljanje rada kontejnera

Prethodna slika prikazuje zaustavljanje kontejnera s nazivom kontejner1, međutim iskoristili smo njegov atribut koji ga jedinstveno identificira, potrebno je navesti samo prvih nekoliko znakova, dovoljno da ne postoji takav skup prvih znakova kod drugih kontejnera.




Slika 32. Pregled pokrenutih kontejnera bez zaustavljenih

Na prethodnoj slici izvršavamo naredbu za pregled aktivnih kontejnera i kontejner1 više nije prisutan na prikazanom popisu. Kako bi vidjeli sve kontejnere koji su kreirani i spremni za izvršavanje pozivamo istu naredbu kao na prethodnoj slici samo dodajemo atribut '-a'.




Slika 33. Pregled svih kreiranih kontejnera

Kontejner1 više se ne izvršava i više mu ne možemo pristupiti s lokalnog računala kako bi prikazali našu aplikaciju. Kako bi ponovno pokrenuli kontejner izvršavamo naredbu na sljedećoj slici.

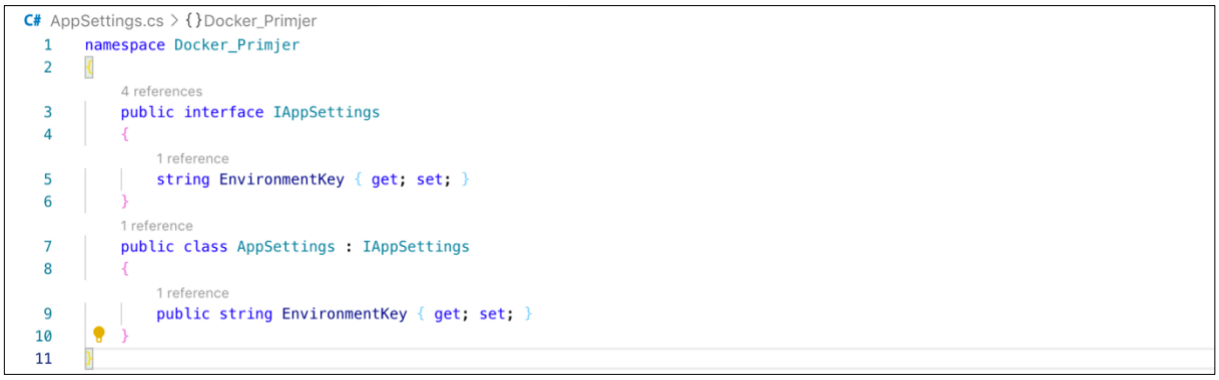

Slika 34. Pokretanje zaustavljenog kontejnera

U ovom slučaju, za razliku od zaustavljanja kontejnera, koristimo njegovo puno ime kako bi ga identificirali, na ovaj način se pokazuju oba načina referenciranja.

## 5.5. Pokretanje kontejnera u različitim okolinama

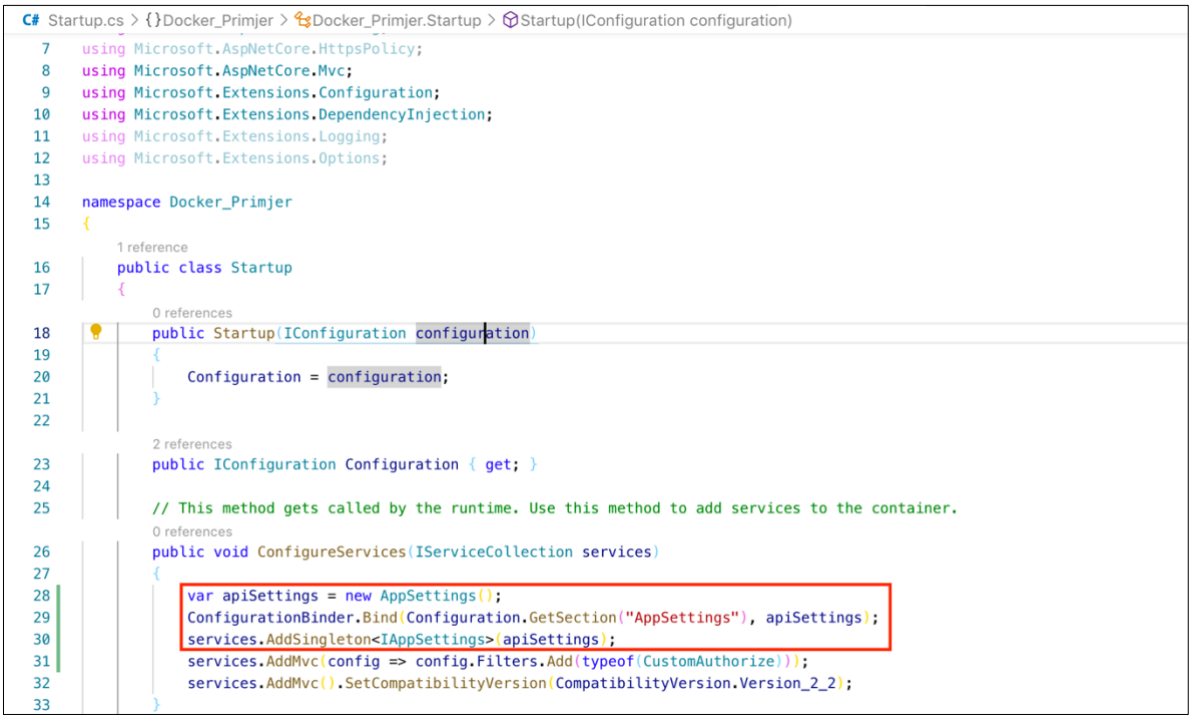
Kod razvoja aplikacija potrebno je osposobiti više radnih okolina od kojih će svaka biti prilagođena traženoj svrsi. Najčešće su to razvojna okolina, testna okolina i produkcijska okolina. Cilj ovog odjeljka je prikazati mogućnost kreiranja različitih okolina uz pomoć Docker kontejnera.

Prvi korak je kreiranje prilagođene klase kojom ćemo definirati varijablu na temelju koje ćemo pokretati određenu okolinu.




Slika 35. AppSettings klasa

U idućem koraku u našoj predefiniranoj klasi Startup uključujemo prethodno kreiranu klasu unutar ConfigureServices metode.

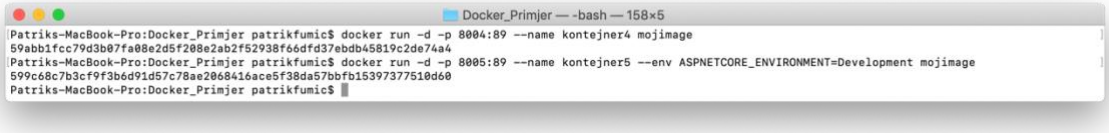



Slika 36. Uključivanje AppSettings klase

U zadnjem koraku kreiramo dvije zasebne json datoteke, od kojih će svaka sadržavati sve podatke za izvršavanje određene okoline. Za potrebe ovog rada navedene datoteke će samo sadržavati izjavu koju ćemo kasnije uključiti u naš kontroler i koja će nam potvrditi da smo pokrenuli aplikaciju u željenoj okolini.

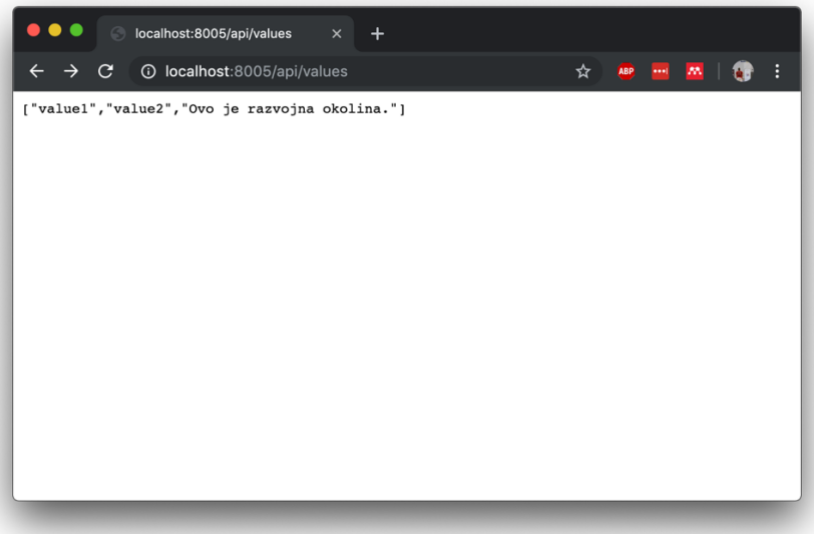

Slika 37. Primjer json datoteke

S obzirom da smo napravili promjene unutar našeg .NET Core projekta moramo ponoviti postupak kreiranja Docker slike, kao što je napravljeno u početku ovog poglavlja, kako bi promjene bile aktivne i unutar Docker kontejnera. Jednom kada je slika kontejnera ponovno kreirana pokrenut ćemo Docker kontejner namijenjen za razvojnu okolinu na način da ćemo proslijediti traženi parametar kod pokretanja samog Docker kontejnera kao što je prikazano na sljedećoj slici.



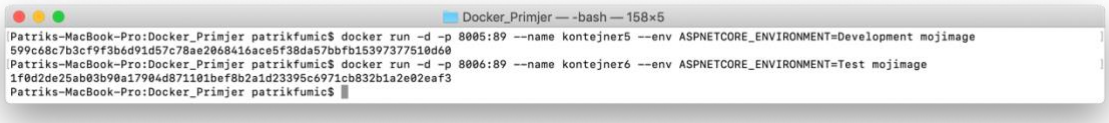

Slika 38. Pokretanje Docker kontejnera u razvojnoj okolini

Pristupamo upravo kreiranom kontejneru preko preglednika i možemo uočiti kako se trenutno nalazimo u razvojnoj okolini.

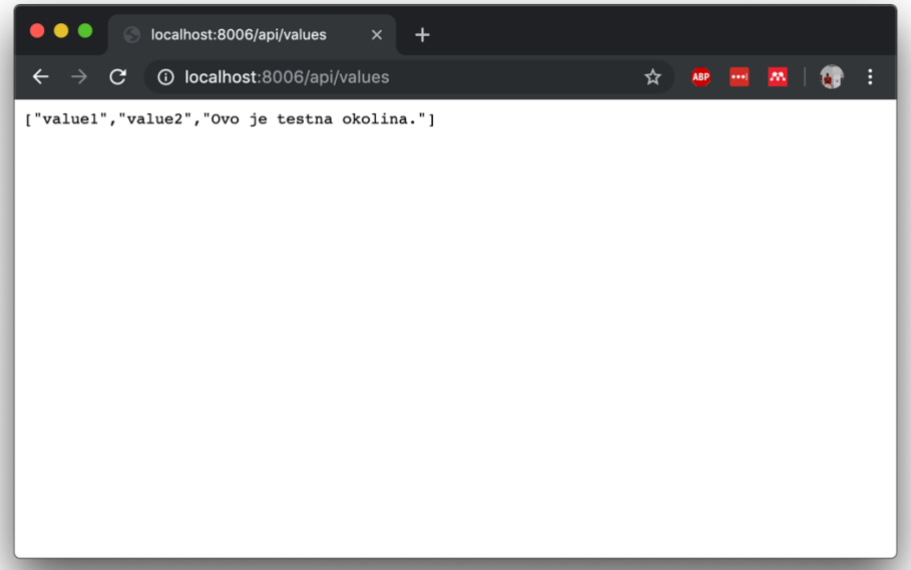



Slika 39. Prikaz razvojne okoline u pregledniku

Na isti način pokrećemo i drugi kontejner koji ćemo pokrenuti na portu 8006 i koji će biti namijenjen za testu okolinu. U ovom slučaju prosljeđujemo vrijednost 'Test' za 'ASPNETCORE_ENVIRONMENT' parametar.




Slika 40. Pokretanje Docker kontejnera u testnoj okolini




Slika 41. Prikaz testne okoline u pregledniku

Na ovaj način smo prikazali kako uz pomoć Docker kontejnera možemo realizirati sve potrebne okoline koje su dio postupka razvoja aplikacijskih rješenja. Ukoliko pogledamo trenutno aktivne kontejnere uočiti ćemo da imamo tri pokrenuta kontejnera od kojih je svaki pokrenut u različitoj okolini.


Slika 42. Prikaz pokrenutih kontejnera u različitim okolinama

## 5.6. Pokretanje različitih verzija slika kontejnera

Kod razvoja većih projekata uvijek je poželjno verzionirati rad na način da se uvijek možemo vratiti na prethodnu verziju, ukoliko je potrebno ili održavati različite verzije ukoliko moramo isporučiti već spomenute različite verzije različitim korisnicima. Ovaj odjeljak je namijenjen upravo verzioniranju slika kontejnera. Kada kreiramo sliku kontejnera bez da specificiramo verziju zadano se verzija sama postavlja na vrijednost 'latest'. Kod svakog kreiranja slike s istim imenom, prehodna slika kontejnera se zamjeni s onom koja se trenutno kreira i zato je potrebno ručno verzionirati slike kontejnera ukoliko ih želimo sačuvati. Za potrebe ovog rada dodat ćemo potrebu za uključivanjem određenog zaglavlja u zahtjevu prema aplikaciji kako bi ona bila dostupna i tu promjenu ćemo primijeniti u verziji 2.0 naše slike kontejnera dok ćemo u verziji 1.0 ostaviti trenutno stanje u kojemu se nalazi slika kontejnera.

Prvo što želimo napraviti je dodijeliti verziju 1.0 postojećoj slici kontejnera 'mojimage' kako je prikazano na sljedećoj slici.


Slika 43. Dodjeljivanje verzije postojećoj slici kontejnera

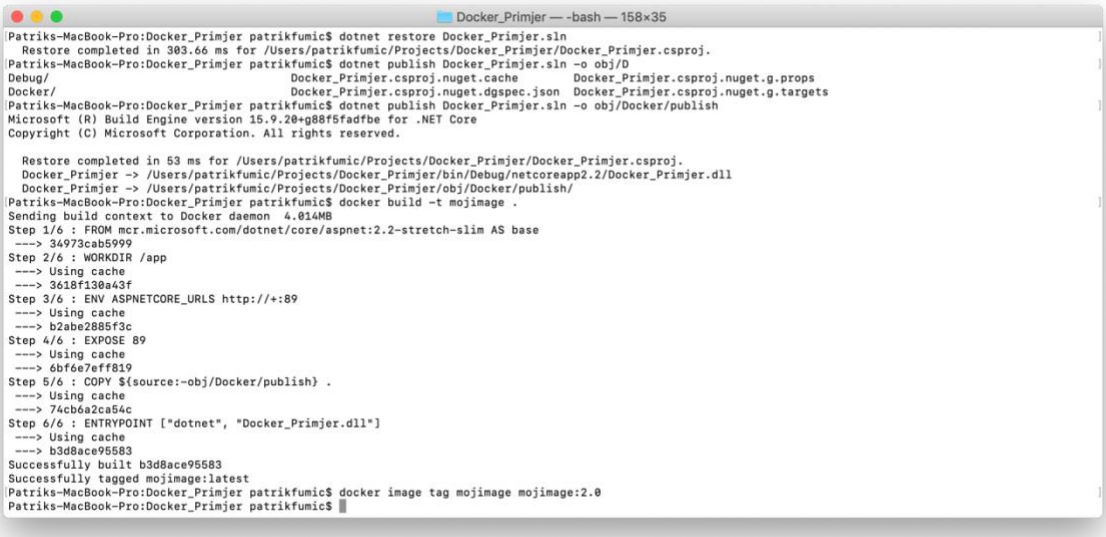
Odmah nakon dodjeljivanje verzije izvršili smo i naredbu za prikaz postojećih slika kontejnera, kao što je prikazano na slici i možemo vidjeti kako sada imamo dvije slike kontejnera s nazivom 'mojimage' i jedna od njih sada sadrži verziju 1.0.

Kao što je već navedeno u ovom odjeljku kreirana je klasa koja će definirati prilagođenu autorizaciju koja će zahtijevati prosljeđivanje određenog zaglavlja kako bi zahtjev bio odobren. Na iduće dvije slike je prikazana sama prilagođena klasa i njezino uključivanje u metodi ConfigureServices.


Slika 44. CustomAuthorize klasa

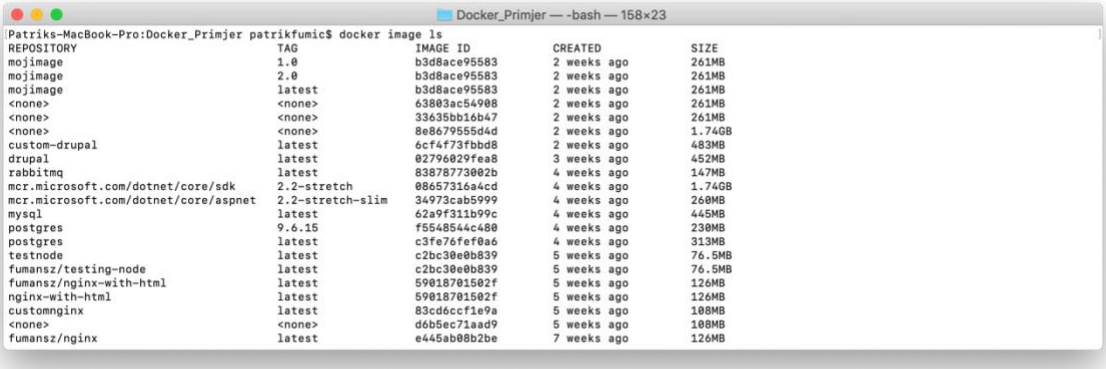

Slika 45. Uključivanje prilagođene autorizacije

Zbog izmjena unutar .NET Core projekta ponovno moramo ponoviti postupak objavljivanja projekta i kreiranja slike kontejnera kako bi uključili provedene izmjene unutar istog. Na idućoj slici je prikazan postupak ponovnog kreiranja slike kontejnera i dodjeljivanje verzije 2.0 kreiranoj slici kontejnera.



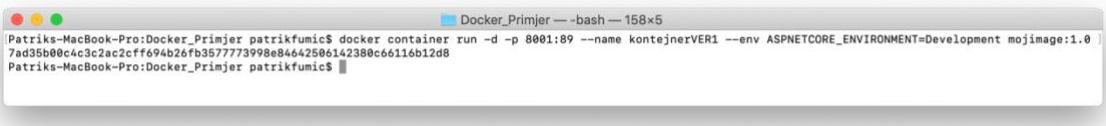

Slika 46. Kreiranje verzije 2.0 slike kontejnera

Sada kada pregledamo postojeće slike kontejnera možemo vidjeti da raspolažemo s dvije različite verzije slike kontejnera. Prisutne su verzije 1.0 i 2.0 'mojimage' slike kontejnera kao i 'latest' verzija iz koje smo kreirali verziju 2.0.



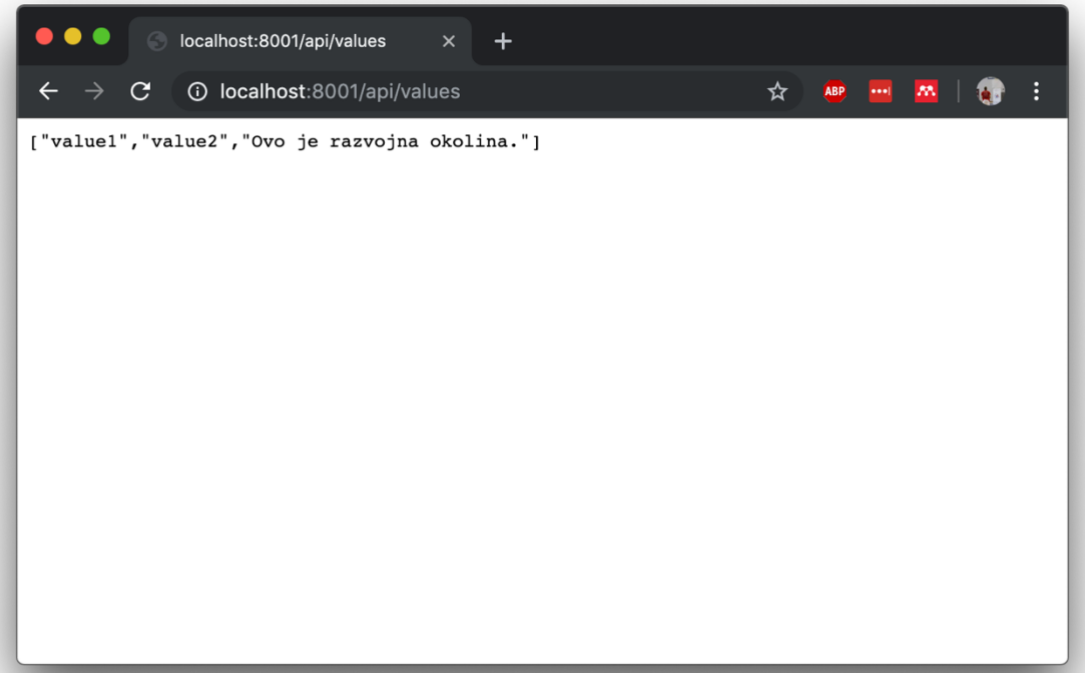

Slika 47. Prikaz kreiranih različitih verzija slika kontejnera

Prvo pokrećemo kontejner koji će se temeljiti na verziji 1.0 i pristupit ćemo mu na uobičajeni način preko preglednika bez slanja zaglavlja.




Slika 48. Pokretanje verzije 1.0 kontejnera

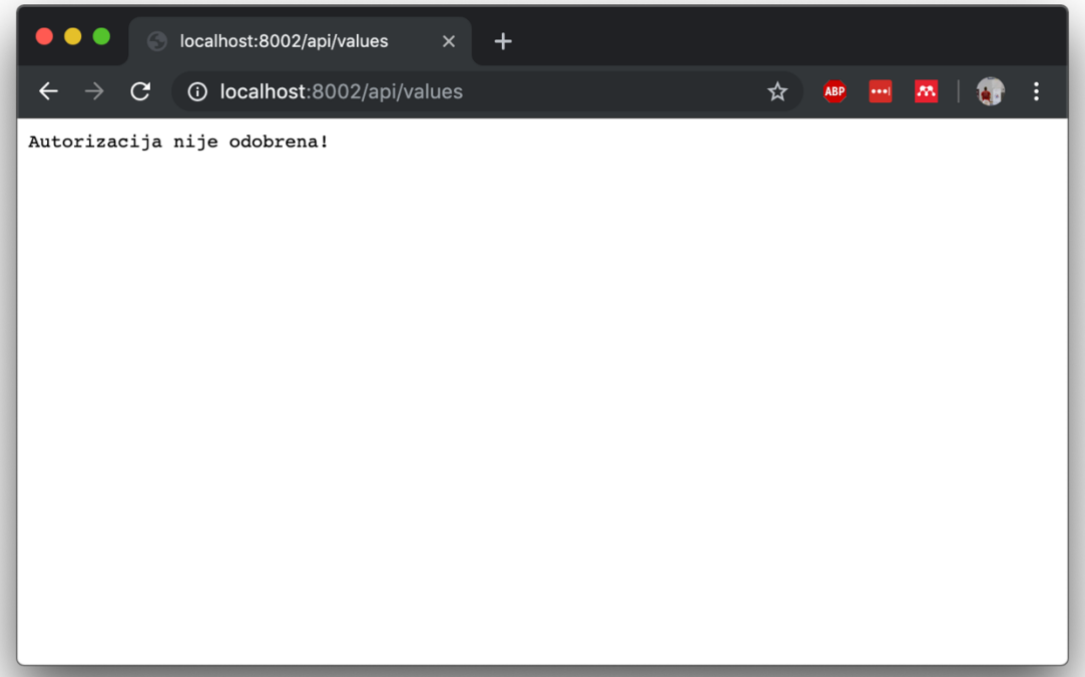
Sljedeća slika prikazuje da je pristup kontejneru preko preglednika u potpunosti omogućen kao i do sada, zato što se radi o verziji 1.0 kojoj smo već prije pristupali na ovaj način.




Slika 49. Prikaz pristupanja kontejneru verzije 1.0

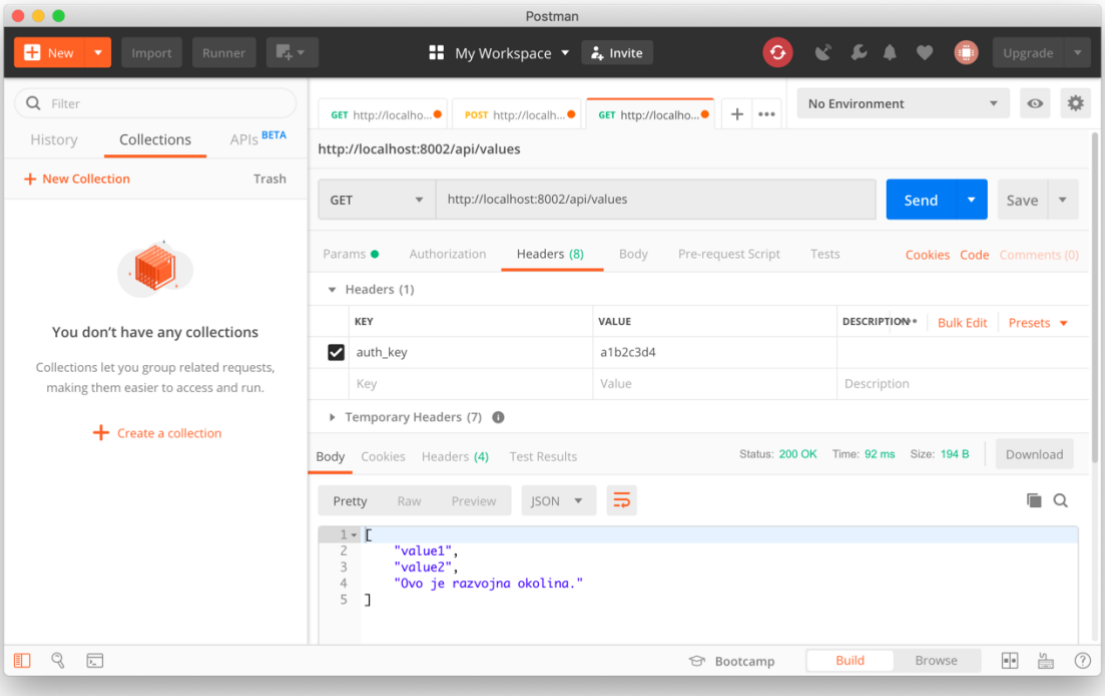
Sada ćemo pokrenuti verziju 2.0 u koju smo uključili potrebu za autorizacijom pri slanju zahtjeva i pokušat ćemo na isti način pristupiti kreiranom kontejneru uz pomoć preglednika kako je prikazano na sljedećim slikama.


Slika 50. Pokretanje verzije 2.0 kontejnera




Slika 51. Prikaz pristupanja kontejneru verzije 2.0

Vidimo da je prikazana prilagođena poruka kod neodobrene autorizacije koju smo definirali u klasi CustomAuthorize. Ovo se dogodilo zato što se radi o verziji 2.0 koja uključuje autorizaciju i za pristupanje ovoj aplikaciji potrebno je uključiti tražena zaglavlja u zahtjev kako bi isti bio odobren. Za ovaj postupak ćemo koristiti alat Postman kojim će nam omogućiti da definiramo potrebne parametre kako bi dobili pristup aplikaciji kako je prikazano na slici u nastavku.




Slika 52. Prikaz pristupanja kontejneru verzije 2.0 s potrebnim zaglavljem

Kao i u pregledniku unosimo localhost adresu s dodijeljenim portom za kontejner verzije 2.0, ali ovaj put uključujemo zaglavlje čiji je ključ 'auth_key', a vrijednost je jedinstveni ključ koji smo definirali u klasi CustomAuthorize. Možemo vidjeti pri dnu Postman aplikacije kako smo dobili očekivani odgovor i dokazali na ovaj način da se radi o kontejneru koji pokreće verziju 2.0.

# 6. Zaključak

Primarna svrha Dockera kao sredstva isporuke softvera je ubrzati postojeće procese i kao rezultat toga omogućiti brži dolazak do željenih rezultata unutar IT područja. Ono što Docker omogućava je brži razvoj, isporuku, testiranje, ažuriranje i oporavak. Način na koji to Docker kontejneri omogućuju je to što održavaju konzistentnost projekta u pitanju, bez obzira na vanjske faktore kao što je operativni sustav na kojemu ga pokrećemo. Dozvoljava nam da pokrećemo i testiramo projekt u jednakim uvjetima bez nepotrebnog trošenja vremena na postavljanje okoline i svih potrebnih zavisnosti kako bi mogli pokrenuti već spomenuti projekt i na taj način pruža razvojnim programerima više vremena za razvoj novih značajki.

Dodavanje same Docker podrške nije problem bez obzira radi li se o projektu koji tek započinje svoj životni ciklus ili se radi o projekti koji već postoji i želi migrirati svoje aplikacija na Docker tehnologiju i prepakirati ih u obliku Docker kontejnera. Ne postoji potreba da se ponovno razvija aplikacija kako bi bila kompatibilna s Dockerom nego je dovoljno zapakirati postojeću i raspodijeliti je po potrebama unutar dane infrastrukture.

Docker je iznimno lagana tehnologija u pogledu zauzeća hardvera. Na konkretnom primjeru tvrtke MetLife, jedne od najvećih svjetskih kompanija koja se specijalizira u području osiguranja, možemo vidjeti rezultat njihovog prelaska na Docker tehnologiju. Prelazak na Docker tehnologiju unutar navedene tvrtke je rezultirao u 70 postotnom smanjenju troškova na virtualne strojeve, 67 postotnom smanjenju korištenja središnjih procesorskih jedinica i na kraju 10 puta boljem iskorištenju dostupne procesorske snage što je na kraju rezultiralo u 66 postotnim smanjenjima troškova u odnosu na stanje prije implementacije Docker tehnologije (Junod, 2017).

Kod projekata kod kojih očekujemo veliko opterećenje, pokretanjem većeg broja kontejnera koje ćemo postaviti iza sloja koji će ravnomjerno raspodjeljivati zahtjeve između istih možemo razriješiti problem u pitanju. Zbog malih zahtjeva samih Docker kontejnera njihova primjena je idealna u ovakvim situacijama zato što su dodatni troškovi Docker kontejnera mnogo manji od onih koje nameću virtualni strojevi.

Uz sve pozitivne aspekte Docker tehnologije važno je osvrnuti se i na određene negativne stvari vezane za spomenutu Docker tehnologiju. Kod velikih projekata sama integracija ili prelazak na pakiranje aplikativnih rješenja unutar Docker kontejnera može biti iznimno kompleksan pothvat koji ponekada nije potreban kao takav. Osobe koje su dosada radile na projektu i nakon prelaska mogu nastaviti raditi i dalje na isti način uz manju obuku vezanu za kontejnere ukoliko je prelazak ispravno odrađen, ali je nužno, pogotovo na velikim projektima, imati osobe koje su specijalizirane u Docker tehnologiji i koje će biti zadužene za

održavanje cijelog Docker dijela sustava i pripremanje aplikativnih rješenja kako bi ostali zaposlenici mogli raditi na istima.

Smatram da je Docker tehnologija budućnost informacijskog sektora što se i može uvidjeti prelaskom većih kompanija na način isporuke svojih aplikativnih rješenja unutar Docker kontejnera. Svim svojim značajkama oni ubrzavaju i olakšavaju razvojni ciklus i na taj način omogućavaju razvojnim programerima više vremena za rad na novim značajkama koje će pomaknuti cijeli informacijski sektor naprijed.

# Popis literature


Bauer, R. (2018). What's the Diff: VMs vs Containers. Retrieved August 7, 2019, from https://www.backblaze.com/blog/vm-vs-containers/

Black, A. P. (2013). Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation*. https://doi.org/10.1016/j.ic.2013.08.002

Cleverism. (2019). Docker. Retrieved August 7, 2019, from https://www.cleverism.com/company/docker/

Dietrich, E. (n.d.). CONTAINER AS A SERVICE: WHAT DEVS NEED TO UNDERSTAND. Retrieved September 12, 2019, from https://www.scalyr.com/blog/container-as-a-service-what-devs-need-to-understand/

Docker. (2019a). Docker Overview. Retrieved August 10, 2019, from https://docs.docker.com/engine/docker-overview/

Docker. (2019b). What is a Container? Retrieved August 10, 2019, from https://www.docker.com/resources/what-container

Fong, J. (2017). Visa Inc. Gains Speed and Operational Efficiency with Docker Enterprise Edition. Retrieved from https://blog.docker.com/2017/04/visa-inc-gains-speed-operational-efficiency-docker-enterprise-edition/

Golden, B. (2008). *Virtualization For Dummies*. Hoboken: Wiley Publishing, Inc.

Gray, J. (1981). *The Transaction Concept: Virtues and Limitations*.

Grey, J., & Reuter, A. (n.d.). *Transaction Processing: Concepts and Techniques*. Retrieved from https://books.google.hr/books?hl=en&lr=&id=VFKbCgAAQBAJ&oi=fnd&pg=PP1&dq=transaction&ots=2vEZmnEmQR&sig=DIRpGvIY_CM_fW7HyTxUKrRtc3Q&redir_esc=y#v=onepage&q=transaction&f=false

Junod, B. (2017). MetLife Uses Docker Enterprise Edition to Self Fund Containerization. Retrieved August 28, 2019, from https://blog.docker.com/2017/10/metlife-docker-enterprise-edition-self-funded-containerization/

Lewis, J., & Fowler, M. (2014). Microservices. Retrieved August 17, 2019, from https://martinfowler.com/articles/microservices.html

Manturewicz, M. (2019). What is CI/CD - all you need to know. Retrieved August 17, 2019, from https://codilime.com/what-is-ci-cd-all-you-need-to-know/

Powers, G. K. (2000). *Information Processes and Technology HSC Course* (1st ed.). Pearson Australia.

Reuters. (2019). Visa Inc (V.N). Retrieved August 16, 2019, from https://www.reuters.com/finance/stocks/company-profile/V.N

Smith, J. E., & Nair, R. (2005). The architecture of virtual machines. *Computer*, *38*(5), 32–38. https://doi.org/10.1109/MC.2005.173

Tuli, S. (n.d.). Learn How to Set Up a CI/CD Pipeline From Scratch. Retrieved August 16, 2019, from https://dzone.com/articles/learn-how-to-setup-a-cicd-pipeline-from-scratch

Turnbull, J. (2019). *The Docker Book*. Creative Commons.

Wright, C. P. (2004). *Kernel Korner - Unionfs: Bringing Filesystems Together*. Retrieved from https://www.linuxjournal.com/article/7714

# Popis slika