# Implementacija sustava za automatsku naplatu parkiranja

**Gal, Stela**

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

Stela Gal

# Implementacija sustava za automatsku naplatu parkiranja

Diplomski rad

Pula, lipanj, 2024.godine

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

Stela Gal

# Implementacija sustava za automatsku naplatu parkiranja

Diplomski rad

JMBAG: 0303082675
Studijski smjer: Informatika
Predmet: Razvoj IT rješenja
Znanstveno područje: Društvene znanosti
Mentor: Prof.dr.sc. Nikola Tanković

Pula, lipanj, 2024.godine

**IZJAVA O AKADEMSKOJ ČESTITOSTI**

Ja, dolje potpisana Stela Gal, kandidat za magistra informatike ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljeni način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____

**IZJAVA O KORIŠTENJU AUTORSKOG DJELA**

Ja, Stela Gal dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom

Implementacija sustava za automatsku naplatu parkiranja

koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, _____

Potpis

_____

# Sadržaj

# Sažetak

Ovaj diplomski rad bavi se izradom sustava pametnog parkirališta koristeći suvremene tehnologije kontejnerizacije i orkestracije kontejnera. Sustav omogućuje automatizirano prepoznavanje registarskih oznaka vozila, upravljanje ulaznim i izlaznim rampama te vođenje evidencije parkirnih transakcija, čime se optimizira upravljanje parkiralištima i poboljšava korisničko iskustvo.

U radu su detaljno opisani procesi dizajna, implementacije i evaluacije sustava, uključujući analizu potrebne opreme i softvera te pružanje sveobuhvatnih uputa za implementaciju. Primjena kontejnerizacije osigurava izolaciju i neovisnost aplikacija, dok orkestracija kontejnera omogućava efikasno upravljanje distribuiranim aplikacijama, čime se postiže skalabilnost i jednostavnost upravljanja.

Svi detalji i izvori implementacije  poslužiteljskog sloja, korisničkog sloja i uklopnog računala Raspberry Pi mogu se pronaći na: https://github.com/stela88/smart-parking-cloud , https://github.com/stela88/smart-parking-frontend i https://github.com/stela88/smart-parking-raspberry-pi-app

# 1. Uvod

Pametna parkirališta predstavljaju inovativno rješenje koje koristi modernu tehnologiju za optimizaciju upravljanja parkiralištima, smanjujući gužve i poboljšavajući korisničko iskustvo. Ovaj sustav automatizira prepoznavanje registarskih oznaka vozila, upravljanje ulaznim i izlaznim rampama te evidenciju parkirnih transakcija, omogućujući efikasniji i korisnicima prilagođeniji način parkiranja.

Kako bi se osigurala skalabilnost i jednostavno upravljanje aplikacijom, ovaj rad prikazuje upotrebu kontejnerizacije i orkestracije kontejnera za implementaciju sustava. Kontejnerizacija omogućuje izolaciju i neovisnost aplikacija unutar kontejnera, dok orkestracija kontejnera osigurava jednostavno i učinkovito upravljanje distribuiranim aplikacijama.

Cilj ovog rada je prikazati proces izrade pametnog parkirališta koristeći moderne tehnologije. Rad uključuje analizu potrebne opreme i softvera, detaljne upute za implementaciju te evaluaciju performansi sustava, pružajući sveobuhvatan uvid u korake potrebne za realizaciju ovog inovativnog rješenja.

# 2. Dizajn i arhitektura sustava

Pametno parkiralište koje se razmatra u ovom radu temelji se na integraciji više tehnologija koje zajedno omogućuju automatizaciju i optimizaciju parkirališnih operacija. Dizajn i arhitektura rješenja osmišljeni su kako bi se postigla visoka skalabilnost, fleksibilnost i pouzdanost sustava. Sustav se sastoji od nekoliko ključnih komponenti koje su međusobno povezane i surađuju kako bi omogućile učinkovito upravljanje parkiralištem.

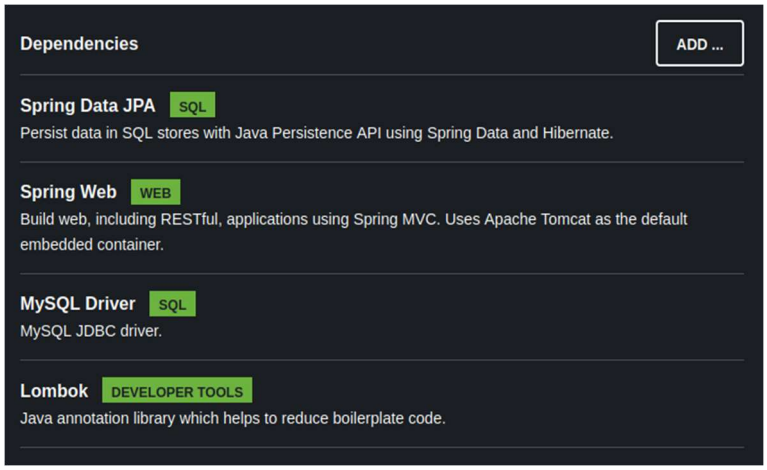
Komponente poslužiteljskog sloja su razvijene korištenjem Spring Boot-a i služi kao središnji dio sustava. Sustava koji je zadužen za obradu podataka, upravljanje bazom podataka i komunikaciju s drugim komponentama sustava. Komponente korisničkog sloja, razvijene korištenjem Vue.js-a, pružaju korisničko sučelje za pregled parkirnih karata i transakcija. Za prikupljanje podataka sa kamera, senzora i ramp te upravljanje istima koristi se Raspberry Pi uređaj. Docker i Docker Compose koriste se za modularnost, omogućavajući jednostavno postavljanje i pokretanje aplikacija u izoliranim okruženjima. Za orkestraciju kontejnera koristi se K3s, lagana verzija Kubernetes sustava, koja omogućuje efikasno upravljanje distribuiranim aplikacijama i osigurava skalabilnost sustava.

Arhitektura pametnog parkirališta dizajnirana je kako bi osigurala optimalnu komunikaciju između svih komponenti. Komponente poslužiteljskog sloja su podijeljene na mikroservise, pri čemu svaki mikroservis ima specifičnu funkciju (upravljanje registracijama, obrada plaćanja, upravljanje parkirnim mjestima). Komunikacija između korisničkog i poslužiteljskog sloja odvija se putem *REST API*-ja. Sustav koristi relacijsku bazu podataka za pohranu podataka o korisnicima, transakcijama, parkirnim mjestima i drugim relevantnim informacijama. Baza podataka je optimizirana za brzi pristup i skalabilnost.

## 2.1. Komponente poslužiteljskog sloja

Razvoj poslužiteljskog sloja za pametno parkiralište korištenjem Spring Boot-a obuhvaća kreiranje robustnog i skalabilnog sustava koji može upravljati raznim funkcionalnostima, kao što su evidencija parkirnih karata i transakcija. Spring Boot je izabran kao glavni okvir za razvoj zbog svoje jednostavnosti i učinkovitosti u izradi *RESTful* web servisa u Java programskom jeziku. Za uspješan rad aplikacije potrebne su određene zavisnosti koje će biti definirane u *pom.xml* datoteci.

Slika 1: Zavisnosti




Izvor: Vlastita izrada

*Spring Data JPA* pojednostavljuje implementaciju *JPA*(*Java Persistence API*) baziranih repozitorija. Omogućava jednostavno i brzo pristupanje podacima u bazi podataka koristeći Java objekte. Podržava rad s različitim relacijskim bazama podataka kao što su MySQL, PostgreSQL, Oracle, itd.

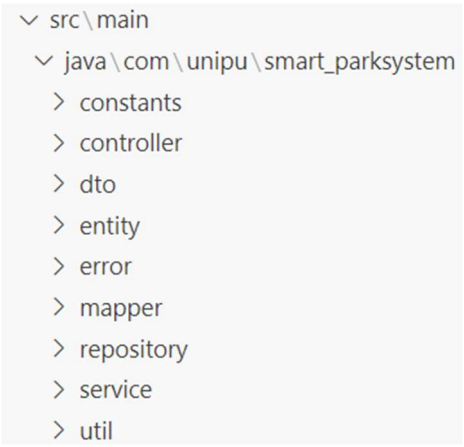
*Spring Web* omogućava razvoj web aplikacija. Omogućava razvoj *RESTful* (koristi anotacije kao što su *@RestController* i *@RequestMapping*) web servisa i integraciju s različitim web tehnologijama. Također, podržava *JSON*, *XML* i druge formate za razmjenu podataka između klijenta i servera.

*MySQL Driver* je *JDBC* (*Java Database Connectivity*) *driver* koji omogućava Java aplikacijama povezivanje i rad s MySQL bazom podataka. Omogućava izvršavanje *SQL* upita (*SELECT, INSERT, UPDATE, DELETE*) i manipulaciju nad podacima u MySQL bazi podataka.

*Lombok* je biblioteka koja pojednostavljuje pisanje Java koda eliminiranjem potrebe za pisanjem *boilerplate* koda kao što su *getter*, *setter*, *toString*, *equals* i *hashCode* metode. Automatski generira uobičajene metodekoristeći anotacije kao što su *@Getter*, *@Setter*, *@ToString*, *@EqualsAndHashCode*, *@NoArgsConstructor*, i *@AllArgsConstructor*.

Aplikacija je dizajnirana prema principima slojevite arhitekture, što omogućuje jasnu separaciju odgovornosti i lakše održavanje koda. Kontroleri, servisi, repozitoriji i *DTO (Data Transfer Object)* objekti čine glavne komponente ove aplikacije.

Slika 2: Pregled paketa




Izvor: Vlastita izrada

Paket *controller* sadrži kontrolere koji upravljaju *HTTP* zahtjevima. Kontroleri koriste anotacije poput *@RestController* za definiranje *RESTful API* krajnjih točaka. Sadrži dvije klase *TicketController* koja upravlja operacijama vezanim uz parkirne karte i *TransactionController* klase koja upravlja operacijama vezanim uz transakcije**.**

Paket *dto* sadrži *Data Transfer Objects (DTO)*, koji predstavljaju objekte za prijenos podataka između različitih slojeva aplikacije. *DTO*-ovi se koriste za enkapsulaciju podataka i olakšani prijenos podataka kroz mrežu.

Paket *repository* sadrži dvije klase, *TicketRepository* koja služi za pristup parkirnim kartama u bazi podataka i *TransactionRepository* klasa za pristup transakcijama.

*Service* paket sadrži servise koji sadrže poslovnu logiku. Logika je vezana uz upravljanje parkirnim kartama i transakcijama.

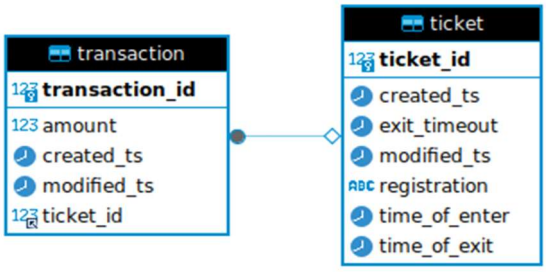
Postavke aplikacije spremljene su u datoteku a*pplication.properties* koja se u Spring Boot aplikacijama koristi za konfiguriranje različitih postavki aplikacije.

```
server.port = 8082
welcome.message = Welcome!!

spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/smart-parking
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql=true
```

*server.port* definira na kojem će *port*-u web poslužitelj slušati *HTTP* zahtjeve. Ovdje je određen *port* 8082. *Hibernate* automatski ažurira shemu baze podataka prema modelima entiteta. Zatim se definira *URL* veza s *MySQL* bazom podataka koja je na lokalnom računalu na *port*-u 3306. Definiraju se korisničko ime i lozinka potrebni za pristup bazi. Sljedeća naredba navodi ime *JDBC driver*-a koji se koristi za povezivanje s bazom, dok posljednja naredba (ukoliko je postavljeno na *true*) ispisuje generirane SQL upite u konzoli.

Slika 3: ER Diagram baze podataka




Izvor: Vlastita izrada

## 2.2. Komponenete korisničkog sloja

Komponente korisničkog sloja dizajnirane su koristeći Vue.js, što omogućava izgradnju korisničkog sučelja koje je responzivno i lako za korištenje. Vue.js je progresivni JavaScript okvir koji omogućava jednostavno povezivanje s poslužiteljskim servisima putem *REST API*-ja. Arhitektura Vue.js aplikacije je podijeljena na komponente koje olakšavaju održavanje koda i ponovno korištenje funkcionalnosti.

Aplikacija je podijeljena na manje, izolirane komponente koje imaju svoju logiku, stilove i predloške (*template*). To omogućava bolju organizaciju koda i olakšava testiranje i održavanje. Također, Vue.js koristi reaktivni sustav koji automatski ažurira prikaz kad se podaci promijene, što poboljšava korisničko iskustvo.

Za komunikaciju s poslužiteljskim *API*-jem koristi se *Axios*, *HTTP* klijent za slanje asinkronih zahtjeva. Ovaj pristup omogućava asinkrono preuzimanje i slanje podataka. Implementirani su servisi za rad s parkirnim kartama i transakcijama, kao što su *TicketService* i *TransactionService*, oni sadrže metode za preuzimanje i slanje podataka prema poslužiteljskom sloju. Projekt je organiziran u direktorije koji sadrže komponente, servise, statičke datoteke, itd. To uključuje *src/components* za Vue komponente i *src/services* za servise. Vue *Router* je korišten za upravljanje navigacijom unutar aplikacije. Omogućava definiranje ruta koje povezuju *URL*-ove s određenim komponentama.

Dizajn aplikacije je fokusiran na jednostavnost korištenja i intuitivno korisničko iskustvo. Korištene su moderne web tehnologije za postizanje responzivnog dizajna koji je prilagođen različitim uređajima. Korišten je *Bootstrap* za izgradnju responzivnih korisničkih sučelja, on omogućava korištenje predefiniranih stilova i komponenti koje su lako prilagodljive. Ovi elementi dizajna i arhitekture osiguravaju da komponente korisničkog sloja budu skalabilne, modularne i jednostavne za održavanje.

## 2.3. Uklopno računalo Raspberry Pi

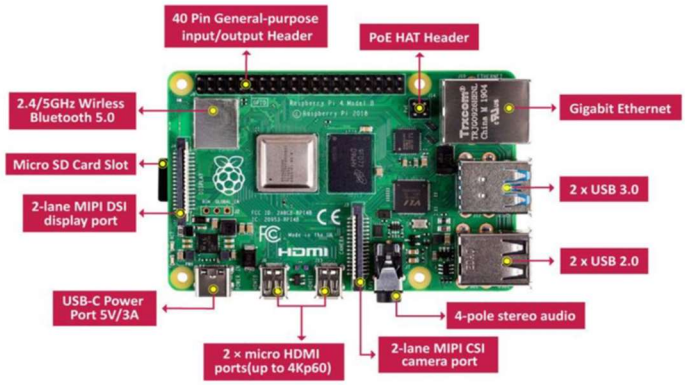
Raspberry Pi uređaj izvorno je stvoren 2012. godine od strane britanske računalne dobrotvorne organizacije Raspberry Pi Foundation. Raspberry Pi je jednoploćno računalo (*single-board computer*). To znači da je, fizički, Raspberry Pi prilično malen, sa svakom procesnom komponentom računala smještenoj na jednoj ploči.

Poanta Raspberry Pi hardvera bila je stvoriti idealno okruženje za učenje za ljude koji se tek

upoznaju s računalstvom i programiranjem. Raspberry Pi također ima IoT (*Internet of Things*) tehnologiju, što mu omogućuje komunikaciju s drugim uređajima spojenim na istu mrežu. Unatoč jednostavnosti Raspberry Pi-ja, to je nevjerojatno svestrano računalo koje može kreirati jednostavne ulazno/izlazne naredbe, omogućujući iznenađujuću razinu automatizacije.

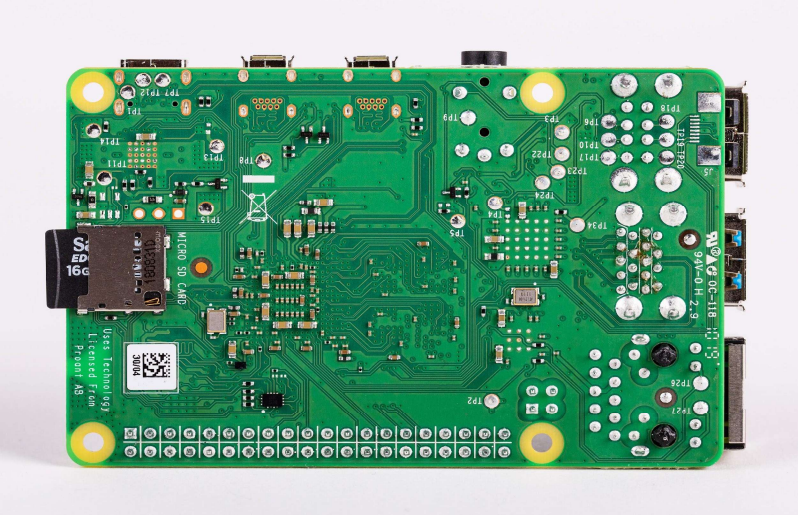
Raspberry Pi koristi integrirani krug koji objedinjuje *CPU (Central Processing Unit)* i *GPU (Graphics Processing Unit)* na jednoj ploči. Ovo čini Raspberry Pi ekonomičnim i energetski učinkovitim računalom. Na ploči se također nalaze RAM memorija i utor za SD karticu koji se koristi kao primarni uređaj za pohranu, koristi za držanje operativnog sustava i potencijalno još nekih datoteka. Što se tiče ulaza i izlaza (*I/O*) na ploči su prisutni razni priključci kao što su USB priključci, *GPIO (General Purpose Input/Output)* pinovi, HDMI izlaz za povezivanje s monitorom, i Ethernet port za mrežnu povezanost. *GPIO* pinovi omogućuju povezivanje raznih senzora, motora i drugih elektroničkih komponenti, što Raspberry Pi čini idealnim za projekte u područjima robotike, automatizacije i *IoT (Internet of Things)*. Raspberry Pi se napaja putem USB-C priključka s naponom od najmanje 5V i 800 mA struje. Nedovoljno napajanje može uzrokovati nepravilno funkcioniranje uređaja. Optimalno napajanje je ključno za stabilan rad cijelog sustava. Što se tiče pohrane, SD kartica je primarni medij za pohranu operativnog sustava i korisničkih podataka. Preporučuje se korištenje SD kartice s kapacitetom od najmanje 32 GB kako bi se osigurala dovoljna memorija za rad i pohranu podataka.

Slika 4: Gornja strana uređaja




Izvor: https://www.researchgate.net/figure/A-well-labelled-diagram-of-the-Raspberry-Pi-4-Board-7_fig1_378746679

Slika 5: Donja strana uređaja



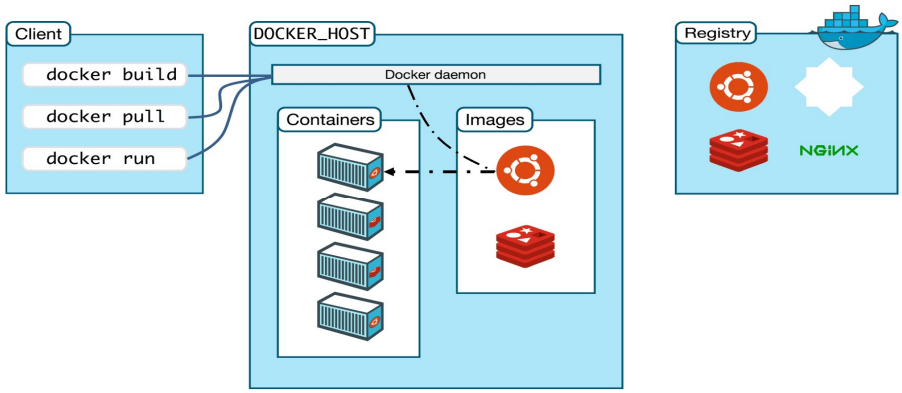
Izvor: https://www.cnx-software.com/2019/06/24/raspberry-pi-4-features-broadcom-bcm2711-processor-up-to-4gb-ram/

Na uređaj će biti priključeni ultrazvučni senzori, kamere i servo motori. Ultrazvučni senzori koriste zvučne valove za mjerenje udaljenosti do objekata. Emitiraju zvučne impulse i mjere vrijeme potrebno za povratak eha. Na temelju brzine zvuka i vremena povratka signala, senzori izračunavaju udaljenost do objekta. Senzori su povezani na *GPIO* pinove na Raspberry Pi-ju. Svaki senzor koristi nekoliko pinova za napajanje (*VCC* i *GND*) i komunikaciju (*TRIG* i *ECHO*). Ultrazvučni senzori obično rade na 5V, što se može osigurati iz Raspberry Pi-jevog napajanja ili vanjskog izvora ako je potrebno. Svaki senzor je dizajniran kao samostalna jedinica koja se može lako zamijeniti ili nadograditi bez utjecaja na ostatak sustava. Servo motor koristi *PWM (Pulse Width Modulation)* signal za kontrolu položaja osovine. Trajanje pulsa određuje kut na koji će se motor postaviti. Servo motor ima tri žice: napajanje (*VCC*), uzemljenje (*GND*) i signal (*PWM*). Signalna žica se povezuje na *GPIO* pin koji podržava *PWM* izlaz na Raspberry Pi-ju. Servo motori se obično napajaju s 5V, a za stabilan rad često koriste vanjski izvor napajanja kako bi se izbjeglo opterećenje Raspberry Pi-jevih resursa. Dizajnirani kao zamjenjive jedinice koje se lako mogu ukloniti i zamijeniti. Korištenje *PWM* signala omogućava se precizna kontrola položaja servo motora, što je ključno za točne mehaničke operacije. Kamere koriste senzore slike za snimanje vizualnih podataka i njihovo prenošenje Raspberry Pi-ju za daljnju obradu. Mogu biti povezane putem *CSI (Camera Serial Interface)* ili USB sučelja. U ovom radu koristit će se kamere povezane putem USB sučelja. Spajaju se na jedan od USB portova na Raspberry Pi-ju. Ova vrsta kamere je fleksibilnija jer može koristiti standardne USB *driver*-e.

## 2.4. Docker

Docker je softverska platforma koja omogućuje brzu izradu, testiranje i implementaciju aplikacija. Docker pakira softver u standardizirane jedinice koje se nazivaju kontejneri (*containers*) koji imaju sve što je softveru potrebno za pokretanje, uključujući biblioteke, sistemske alate, kod i vrijeme izvođenja. Izolacija i sigurnost omogućuju pokretanje više spremnika istovremeno na određenom *host*-u. Kontejneri se mogu dijeliti tako da svi oni s kojima ih dijelimo dobivaju isti kontejner koji radi na isti način.

Slika 6: Docker arhitektura




Izvor: https://k21academy.com/docker-kubernetes/docker-architecture-docker-engine-components-container-lifecycle/

Docker koristi arhitekturu klijent-poslužitelj. Docker klijent razgovara s Docker daemon-om, koji obavlja težak posao izgradnje, pokretanja i distribucije Docker kontejnera. Docker klijent i daemon mogu se izvoditi na istom sustavu ili se Docker klijent može povezati s udaljenim Docker daemon-om. Docker klijent i daemon komuniciraju pomoću *REST API*-ja, preko *UNIX socket*-a ili mrežnog sučelja. Još jedan Docker klijent je Docker Compose, koji omogućuje rad s aplikacijama koje se sastoje od više kontejnera.

Docker daemon (*dockerd*) sluša Docker *API* zahtjeve i upravlja Docker objektima kao što su slike, spremnici, mreže i volumeni. Daemon također može komunicirati s drugim daemonima za upravljanje Docker uslugama.

Docker klijent (*docker*) primarni je način na koji mnogi Docker korisnici komuniciraju s Docker-om. Kada se koriste naredbe kao što je *docker run*, klijent šalje te naredbe *dockerd-u*, koji ih izvršava. Naredba *docker* koristi Docker *API*.

Docker registar pohranjuje Docker slike (*images*). Docker Hub je javni registar koji svatko može koristiti, a Docker prema zadanim postavkama traži slike na Docker Hub-u.

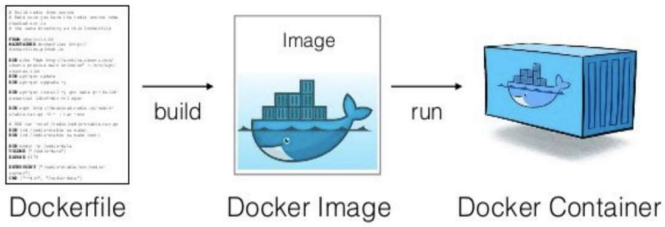
Postoji mogućnost pokretanja vlastitog privatnog registra. Kada se koriste naredbe *docker pull* ili *docker run*, Docker izvlači potrebne slike iz vašeg konfiguriranog registra. Kada se koristi naredba *docker push*, Docker gura  sliku u konfigurirani registar.

### 2.4.1. Razlika između Docker slike i kontejnera

Docker slike i kontejneri su tehnologije za implementaciju aplikacija. Kontejneri omogućuju programerima da pakiraju softver za izvođenje na bilo kojem ciljnom sustavu.
Docker kontejner je *runtime* okruženje sa svim potrebnim komponentama poput koda, ovisnosti i biblioteka potrebnih za pokretanje aplikacijskog koda bez korištenja ovisnosti glavnog stroja. Kontejneri se pokreću na poslužiteljskoj mašini, stroju ili instanci oblaka. Mašina pokreće više kontejnera ovisno o dostupnim temeljnim resursima. Docker slika ili slika kontejnera je samostalna, izvršna datoteka koja se koristi za stvaranje kontejnera. Slika sadrži sve biblioteke, ovisnosti i datoteke koje spremnik treba pokrenuti. Docker slika se može dijeliti i prenositi, tako da možete implementirati istu sliku na više lokacija odjednom - slično kao softverska binarna datoteka. Slike se mogu pohraniti u register kako bi se pratile složene softverske arhitekture, projekti, poslovni segment… Na primjer, javni registar Docker Hub-a sadrži slike kao što su operativni sustavi, okviri programskih jezika, baze podataka i uređivači koda.
Docker kontejner je samostalna softverska aplikacija ili usluga koja se može pokrenuti. S druge strane, Docker slika je predložak učitan u kontejner za njegovo pokretanje, poput skupa uputa. Slike se pohranjuju za dijeljenje i ponovnu upotrebu, a kontejneri se stvaraju i uništavaju tijekom životnog ciklusa aplikacije. Docker slika se stvara iz Dockerfile-a, tekstualne datoteke čitljive ljudima slične konfiguracijskoj datoteci. Dockerfile sadrži sve upute za izradu slike. Nasuprot tome, Docker kontejneri se stvaraju izravno iz Docker slike. Docker slikovna datoteka sastoji se od slojeva slike kako bi veličina datoteke bila mala. Svaki sloj predstavlja promjenu napravljenu na slici. Slojevi su samo za čitanje i mogu se dijeliti između više spremnika. Docker kontejner, budući da je instanca slike, također sadrži slojeve. Međutim, na vrhu ima dodatni sloj za pisanje, poznat kao sloj kontejnera. Sloj kontejnera omogućuje pristup za čitanje i pisanje. Također omogućuje da se sve promjene napravljene unutar kontejnera izoliraju od drugih kontejnera na temelju iste slike. Docker slike su nepromjenjive, što znači da se ne mogu mijenjati nakon što su stvorene. Ako je potrebno napraviti promjene na slici, mora se stvoriti nova sliku sa željenim izmjenama.

Nasuprot tome, kontejneri su promjenjivi i dopuštaju izmjene tijekom vremena izvođenja. Promjene napravljene unutar kontejnera su izolirane za taj određeni kontejner i ne utječu na pridruženu sliku. Neki primjeri promjene su kada se pišu nove datoteke, instaliraju softver ili mijenjaju konfiguracije.

Slika 7: Docker slika i kontejner

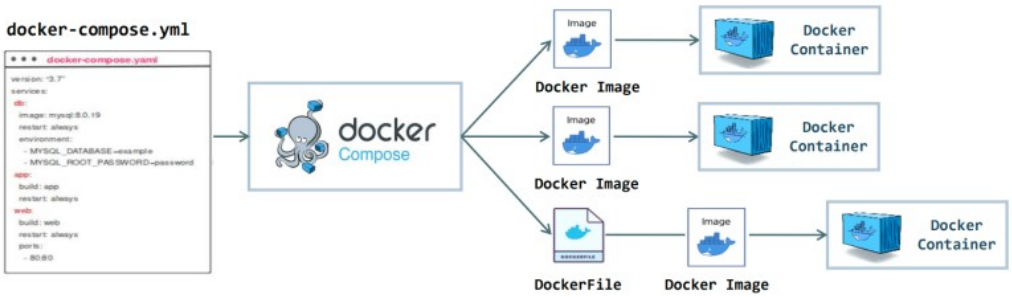



Izvor: https://cto.ai/blog/docker-image-vs-container-vs-dockerfile/

### 2.4.2. Docker-compose

Docker Compose je alat za definiranje i pokretanje aplikacija s više kontejnera. Korištenje Docker Compose-a nudi nekoliko prednosti koje pojednostavljuju razvoj, implementaciju i upravljanje kontejnerskim aplikacijama. Docker Compose omogućuje definiranje aplikacija s više kontejnera i upravljanje njima u jednoj *YAML* datoteci. To pojednostavljuje složeni zadatak orkestriranja i koordinacije različitih usluga, olakšavajući upravljanje i repliciranje aplikacijskog okruženja. Docker Compose konfiguracijske datoteke lako se dijele, olakšavajući suradnju među programerima, operativnim timovima i drugim sudionicima. Ovaj suradnički pristup dovodi do glatkih radnih procesa, bržeg rješavanja problema i povećane ukupne učinkovitosti. Compose predmemorira konfiguraciju korištenu za stvaranje kontejnera. Kada se ponovno pokrene usluga koja se nije promijenila, Compose ponovno koristi postojeće kontejnere. Ponovno korištenje kontejnera znači da se može vrlo brzo i lako promijeniti okruženje. Compose podržava varijable u Compose datoteci. Varijable se mogu koristiti da bi se sustav prilagodio za različita okruženja ili različite korisnike.

Slika 8: Docker-compose

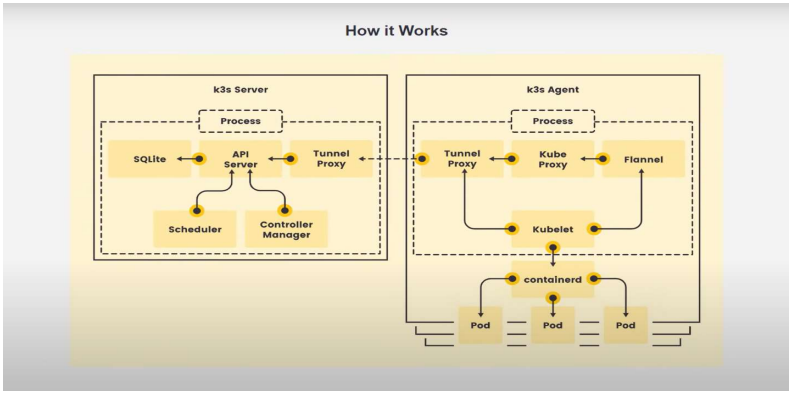



Izvor: https://www.linkedin.com/pulse/swapnil-singh-ltdqf/

## 2.5. K3s

K3s je Kubernetes distribucija koju je izradio Rancher Labs, a u potpunosti je certificirana od strane *Cloud Native Computing Foundation (CNCF)*. K3s je vrlo dostupan i spreman za proizvodnju. K3s je binarna datoteka koju je lako instalirati i konfigurirati. Binarna datoteka teži između 50 i 100 MB te se raspakira kako bi uključila sve potrebne komponente za pokretanje Kubernetes-a na kontrolnoj ploči i radnim čvorovima (*worker nodes*). Sadrži balanser opterećenja usluge koji povezuje Kubernetes usluge s *IP*-em glavnog računala, što ga čini prikladnim za klastere s jednim čvorom. Čvorovi kontrolne ploče pokreću sve Kubernetes u manje od 512 MB RAM-a, a radni čvorovi pokreću svoje komponente u manje od 50 MB RAM-a.

Slika 9: Arhitektura k3s-a




Izvor: https://nerc-project.github.io/nerc-docs/other-tools/kubernetes/k3s/k3s/

### 2.5.1. Usporedba K3s-a i Kubernetes-a

Što se tiče veličine i učinkovitosti, K3s je značajno manji, kako u pogledu prostora na disku tako i u korištenju memorije, što ga čini idealnim za okruženja s ograničenim resursima. K3s zauzima samo oko 40 MB prostora na disku i koristi puno manje memorije u usporedbi s punim Kubernetesom (K8s). Manja veličina omogućuje brže pokretanje i smanjenje troškova infrastrukture.

K3s pojednostavljuje postavljanje klastera s jednom binarnom instalacijom, što ga čini dostupnim za manje timove ili projekte s ograničenom stručnošću u Kubernetesu. Instalacija je krajnje jednostavna, zahtijeva minimalne resurse i može se izvesti s jednim naredbenim retkom, što značajno smanjuje složenost u usporedbi s tradicionalnim K8s klasterom.

Jedan od ključnih aspekata K3s-a je uklanjanje određenih značajki i dodataka koji su često nepotrebni za njegove ciljane slučajeve upotrebe. Fokusira se na temeljne funkcije potrebne za rubno računalstvo, IoT i CI/CD cjevovode, što omogućuje lakšu i bržu implementaciju te upravljanje aplikacijama u ovim specifičnim okruženjima. K3s dolazi s uklonjenim komponentama poput obrade logova i alata za praćenje performansi, koje nisu kritične za male ili rubne implementacije.

Osim toga, K3s uključuje dodatne komponente kao što su pružatelj lokalne pohrane, balanser opterećenja usluge, kontroler *Helm* i pojednostavljena mrežna konfiguracija. Pružatelj lokalne pohrane omogućuje lako postavljanje trajnih volumena, dok balanser opterećenja pomaže u distribuciji mrežnog prometa između različitih servisa. *Helm* kontroler omogućuje jednostavno upravljanje paketima aplikacija, a pojednostavljena mrežna konfiguracija smanjuje potrebu za složenim mrežnim postavkama, što sve zajedno pomaže pojednostaviti operacije u predviđenim okruženjima.

Odabir između K3s i K8s ovisi o veličini projekta, složenosti i dostupnosti resursa. K8s je bez premca u svojoj skalabilnosti i fleksibilnosti za velike, složene aplikacije koje zahtijevaju napredne značajke kao što su horizontalno skaliranje, automatsko otkrivanje servisa i napredna mrežna politika. K8s je dizajniran za upravljanje velikim klasterima koji mogu uključivati stotine ili tisuće čvorova, pružajući robusne mehanizme za visoku dostupnost, otkrivanje kvarova i oporavak.

Nasuprot tome, K3s nudi pojednostavljenu, učinkovitu alternativu za rubno računalstvo, IoT i scenarije u kojima su jednostavnost i očuvanje resursa najvažniji. Primjene uključuju male uređaje poput Raspberry Pi-ja, gdje su resursi ograničeni, ali je potrebna funkcionalnost kontejnerizacije za aplikacije kao što su pametna kućanstva, industrijska automatizacija ili

distribuirani senzorski sustavi. K3s je također izuzetno koristan u okruženjima gdje je potrebna brza implementacija i učestalo mijenjanje aplikacija, kao što su razvojna i testna okruženja, te kontinuirane integracije i isporuke (*CI/CD*).
U konačnici, i K3s i K8s imaju svoje mjesto u modernim IT infrastrukturama, a odabir pravog rješenja ovisi o specifičnim zahtjevima projekta. Dok K8s ostaje zlatni standard za velike korporativne implementacije, K3s pruža prilagodljivu i laganu opciju za manje, brže i manje složene projekte.

# 3. Implementacija rješenja

Implementacija poslužiteljskog sloja uključuje nekoliko ključnih komponenti koje omogućuju funkcionalnost aplikacije, kao što su servisi, repozitoriji i *DTO* objekti.
Servisi implementiraju poslovnu logiku aplikacije i služe kao posrednici između kontrolera i repozitorija. *TicketingService* i *TransactionService* su primjeri servisa koji obrađuju logiku vezanu uz izdavanje parkirnih karata i evidenciju transakcija. Repozitoriji koriste *Spring Data JPA* za pristup podacima u bazi podataka, omogućujući jednostavan i intuitivan način rada s podacima poput spremanja, dohvaćanja i ažuriranja informacija. *TicketRepository* i *TransactionRepository* ključni su repozitoriji u ovoj aplikaciji. *DTO* objekti koriste se za prijenos podataka između različitih slojeva aplikacije, omogućujući enkapsulaciju podataka i prijenos samo potrebnih informacija, čime se poboljšava sigurnost i učinkovitost sustava.

```
public interface TicketingService {

    TicketDto saveTicket(String registration);
```

Sučelje *TicketingService* definira metodu *saveTicket*, koja prima registraciju vozila i vraća *DTO* povezan s parkirnom kartom.

Implementacija *TicketingServiceImpl* pruža konkretne detalje kako će se metoda *saveTicket* izvršiti. U ovom slučaju, sprema novu kartu u bazu podataka. Prije spremanja potrbne su određene provjere. Provjerava se postoji li već aktivna parkirna karta za istu registraciju i je li kapacitet garaže pun. Ako su te provjere uspješne, stvara kartu i sprema ju u bazu podataka.

Kontroler *TicketingController* prima *HTTP* zahtjeve i prosljeđuje ih odgovarajućoj usluzi (*TicketingService*). *HTTP POST* zahtjev za spremanje karte prima se putem endpointa */api/tickets*. Metoda *saveTicket* provjerava je li registracija prisutna u *DTO*-u i zatim prosljeđuje registraciju metodi *saveTicket* usluge *TicketingService*.

Ukupno je 11 poziva vezanih za parkirne karte: gore navedeno spremanje karte u sustav, dohvaćanje liste svih karata, dohvaćanje po registraciji vozila, provjera može li vozilo napustiti parking, označavanje izlaza s parkirališta, dohvaćanje po *ID*-ju, dohvaćanje računa, brisanje i ažuriranje po *ID*-ju, dohvaćanje svih aktivnih karata I dohvaćanje svih aktivnih karata po registracijskoj oznaci. Istom logikom građeni su *TransactionService*, *TransactionServiceImpl*, *TransactionController*. Ukupno je 4 poziva koji se vežu na transakcije: spremanje nove transakcije, dohvaćanje liste svih transakcija, dohvaćanje po *ID*-ju i brisanje transakcije po *ID*-ju.

Komponente korisničkog sloja osiguravaju korisničko sučelje za interakciju sa poslužiteljskim *API*-jem. Korisnički sloj je razvijen koristeći Vue.js, što omogućuje dinamično i responzivno korisničko sučelje. Vue komponente komuniciraju sa poslužiteljskim *API*-jem putem *axios* biblioteke, osiguravajući efikasno dohvaćanje i prikaz podataka korisnicima.

```
import axios from 'axios';

const TICKET_API_BASE_URL = 'http://172.16.1.18:8083/api/tickets';
const TICKET_BY_REGISTRATION =
'http://172.16.1.18:8083/api/tickets/registration/';

class TicketService {
 getTickets() {
  return axios.get(TICKET_API_BASE_URL);
 }
 getTicketByRegistration(registration) {
  return axios.get(`${TICKET_BY_REGISTRATION}${registration}`);
 }
 getReciep(id){
  return axios.get(`${TICKET_API_BASE_URL}/${id}/receipt`);
 }
}
export default new TicketService();
```

Servis *TicketService* zadužen je za rad sa parkirnim kartama. Metoda *getTickets* preuzima sve parkirne karte koje se trnutno nalaze u bazi podataka. Metoda *getTicketByRegistration* dohvaća parkirnu kartu na osnovu registracije vozila, koristi *axios.get* za slanje *GET* zahtjeva na definirani *URL*. Posljednja metoda ovog servisa je *getReceipt* koja dohvaća račun na osnovu *ID*-ja karte.
*TransactionService* je servis zadužen za provođenje transakcija. U ovom servisu nalazi se *POST* metoda *postTransaction* koja na *API* šalje novu transaciju sa podacima o transakciji (*transactionData*).

```
<script>
import TicketService from '../services/TicketService'
import moment from 'moment';

export default{
  name: "Tickets",
  data(){
      return{
         tickets : []
      }
  },
  methods: {
      getTickets(){
         TicketService.getTickets().then((response) => {
            this.tickets = response.data;
         })
      }
  },
  created(){
      this.getTickets()
  },
  filters: {
  formatDate(value) {
    if (value) {
       return moment.utc(new Date(value).toString()).format('MM/DD/YYYY
       hh:mm A');
    }
  }
  }
}
</script>
```

TicketService uvozi se u *Vue* komponentu *Ticket.vue.* Komponenta koristi *data* objekt koji sadrži *tickets* koji će biti ispunjen podacima iz *API*-ja. Poziva se metoda *getTickets* koja, kada

odgovor stigne, ažurira *tickets* čiji se podaci onda prikazuju u tablici koja je vidljiva korisniku. *Created* se poziva kada je komponenta kreirana, automatski poziva *getTickets* da preuzme podatke. *TicketByRegistration* je *Vue* komponenta koja omogućava korisnicima da pretražuju i prikazuju parkirne karte na temelju registracije vozila, provjere koliki je iznos računa te da obave transakciju kako bi mogli napustiti parking.

```
<script>
import TicketService from '../services/TicketService';
import TransactionService from '../services/TransactionService';
import moment from 'moment';

export default {
  name: 'TicketByRegistration',
  data() {
  return {
    tickets: [],
    registration: '',
    ticketId: '',
    receipt: null,
    price: '',
    popupMessage: null,
    refreshIntervalId: null
  };
},
```

Uvoze se servisi koji se koriste za komunikaciju sa poslužiteljskim *API*-jem. *Data* vraća objekt koji sadrži sve podatke za ovu *Vue* komponentu. *Registration* je string koji predstavlja registracijsku oznaku koju korisnik unosi za pretragu karata. *TicketId* predstavlja *ID* trenutno odabrane karte te se koristi prilikom preuzimanja računa i obavljanja transakcija. *Receipt* sadrži informacije o računu, ovaj podatak se popunjava kada korisnik preuzme račun za svoju parkirnu kartu. *Price* predstavlja cijenu parkirne karte. *PopupMessage* sadrži poruku za obavještavanje korisnika (npr. uspješna transakcija), a *refreshIntervalId* koriti se za preiodično osvježavanje tablice.

Metoda *getTicketByRegistration* pretražuje karte na osnovu registracijske oznake. Ako nema registracije briše trenutne podatke, a ako pronađe podatke postavlja ih u reaktivni niz *tickets* i ažurira *ticketId*. *GetReceip* metoda preuzima račun za danu kartu i sprema podatke u *receipt,* dok metoda *postTransaction* šalje transakciju za trenutno odabranu kartu. Ako je transakcija uspješna prikazuje poruku o uspjehu, u suprotnom pokazuje poruku o grešci.

Nakon razvoja komponenti korisničkog sloja, na redu je Raspberry Pi i njegove komponente.

Ultrazvučni senzori igraju ključnu ulogu u sustavu za prepoznavanje objekata i udaljenosti, što je esencijalno za funkcionalnost aplikacije. Ultrazvučni senzori koriste zvučne valove za mjerenje udaljenosti do objekata. U ovom projektu, oni su ključni za detekciju prisutnosti vozila u parkirnim mjestima, pružajući podatke koji se dalje koriste za upravljanje parking sustavom.

U zraku se zvuk širi brzinom od 343 metra u sekundi. Ultrazvučni senzor udaljenosti šalje impulse ultrazvuka koji su nečujni za ljude i detektira jeku koja se šalje natrag kada se zvuk odbija od obližnjeg objekta. Zatim koristi brzinu zvuka za izračunavanje udaljenosti od objekta. U ovom radu za rad senzora koristit će se dva ultrazvučna senzora za mjerenje udaljenosti uz pomoć biblioteke *RPi.GPIO*. *RPi.GPIO* je popularna Python biblioteka koja omogućuje kontrolu *GPIO* pinova na Raspberry Pi uređaju. Ova biblioteka se koristi za postavljanje pinova, čitanje ulaznih i postavljanje izlaznih signala.

```
servo1 = AngularServo(12, min_angle=-90, max_angle=90)
servo2 = AngularServo(25, min_angle=-90, max_angle=90)
GPIO.setmode(GPIO.BCM)
TRIG1, ECHO1, TRIG2, ECHO2 = 23, 24, 21, 20
GPIO.setup([TRIG1, TRIG2], GPIO.OUT)
GPIO.setup([ECHO1, ECHO2], GPIO.IN)

GPIO.output(TRIG1, False)
print("Waiting For Sensor 1 To Settle")
time.sleep(2)
GPIO.output(TRIG2, False)
print("Waiting For Sensor 2 To Settle")
time.sleep(2)
```

Senzori su spojeni na pinove 23, 24, 21 i 20, postavljeni su kao ulazni i izlazni. *BCM* se odnosi na *Broadcom SOC channel number*, što je numeracija unutar čipa koji se koristi na Raspberry Pi.

```
def get_distance(TRIG, ECHO):
    GPIO.output(TRIG, True)
    time.sleep(0.1)
    GPIO.output(TRIG, False)
    pulse_start, pulse_end = 0, 0

    while GPIO.input(ECHO) == 0:
        pulse_start = time.time()
    while GPIO.input(ECHO) == 1:
        pulse_end = time.time()

    pulse_duration = pulse_end - pulse_start
    return round(pulse_duration * 17150, 2)
```

Ultrazvučni senzor radi tako što emitira zvučni signal putem *TRIG* pina i mjeri vrijeme potrebno da se signal vrati nazad putem *ECHO* pina. Vrijeme se mjeri od trenutka kada *ECHO* pin postane visoko *(pulse_start)* do trenutka kada ECHO pin postane nisko *(pulse_end)*. Na kraju ovih petlji, imamo početno i završno vrijeme pulsa, što nam omogućuje da izračunamo ukupno trajanje pulsa. To je bitno za izračunavanje udaljenosti objekta od senzora. Formula po kojoj se računa udaljenost jest ***udaljenost = trajanje pulsa * 17150***.

U glavnom programu to izgleda ovako:

```
try:
    while True:
        # First sensor
        distance_first_sensor = get_distance(TRIG1, ECHO1)
        print("Distance sensor 1:", distance_first_sensor, "cm")

        # Second sensor
        distance_second_sensor = get_distance(TRIG2, ECHO2)
        print("Distance sensor 2:", distance_second_sensor, "cm")

        # Camera 1
        if distance_first_sensor <= 8:
            main_process(distance_first_sensor, cv2.VideoCapture(0))
```

Udaljenost se mjeri za prvi i drugi senzor. Ovisno koji senzor je očitao udaljenost manju od 8 cm uključuje kameru na ulazu (veže se na prvi senzor) ili na izlazu (veže se na drugi senzor).

Na redu je implementacija servo motora koji se koriste za upravljanje mehaničkim komponentama kao što su barijere ili indikatori u parkirnom sustavu. Njihova sposobnost za precizno pozicioniranje omogućuje pouzdanu i efikasnu kontrolu fizičkih elemenata sustava. U ovom radu koriste se dva servo motora koji služe za podizanje i spuštanje rampe. Koriste se pomoću biblioteke *gpiozero*.

```
servo1 = AngularServo(12, min_angle=-90, max_angle=90)
servo2 = AngularServo(25, min_angle=-90, max_angle=90)
```

Servo motori inicijalizirani su na pin 12 i pin 25. Raspon kretanja motora postavljen je od -90 do 90 stupnjeva. Servo motori postavljeni su na ulaz i izlaz. Na ulazu se rampa diže kada je uspješno spremljena registracijska oznaka automobila, a spušta se kada je automobil udaljen minimalno 20 cm od senzora. Servo na izlazu radi na način da se prvo radi provjera je li korisnik platio vrijeme provedeno na parkingu. Ukoliko je sve plaćeno rampa se podiže te ponovno spušta nakon što se automobil udalji 20 cm.

Slijedeća na redu je implementacija kamera. Kamere su esencijalne za prepoznavanje registarskih oznaka i nadzor parkinga. Integracija s računalnim vidom omogućuje automatsku identifikaciju vozila i poboljšava sigurnost i efikasnost parkirnog sustava. U ovom radu koristit će se dvije USB kamere koje će raditi uz pomoć OpenCV-a (*Open Source Computer Vision Library*). OpenCV, poznata kao cv2 u Pythonu, je moćna biblioteka za računalni vid i obradu slika. OpenCV je razvijen kako bi programerima omogućio pristup alatima za obradu slika i video analiza.

```
def capture_images(camera):
    print("Taking 10 pictures...")
    save_dir = '/home/sgal/images/pics/'
    os.makedirs(save_dir, exist_ok=True)

    for i in range(10):
        ret, frame = camera.read()
        if ret:
            cv2.imwrite(os.path.join(save_dir, f'image_{i}.jpg'), frame)
    return glob.glob(save_dir + '*.jpg')
```

Funkcija *capture_images(camera)* provjerava postoji li direktorij za spremanje slika, ako ne postoji kreira ga. Zatim snima 10 slika koje sprema u prethodno definirani direktorij. Finalno, vraća listu putanja do svih snimljenih slika.

```
def process_image(image_paths):
    for image_path in image_paths:
        image = cv2.imread(image_path)
        success, image_jpg = cv2.imencode('.jpg', image)
        files = {'upload': image_jpg.tobytes()}
        time.sleep(1)

        response = requests.post(
            'https://api.platerecognizer.com/v1/plate-reader/',
            headers={'Authorization': 'Token
            397cefc199536f232215b12cc3a5651c5d8847f3'},
            files=files
        )
        response_data = response.json()
        results = response_data.get('results', [])
        for result in results:
            plate_number = result.get('plate')
            print("License plate number:", plate_number)
            registration_list.append(plate_number)
```

Funkcija *process_image(image_paths)* obrađuje snimljene slike te pretvara kodirane podatke u bajtove kako bi se mogli poslati kao dio *HTTP* zahtjeva. Koristeći biblioteku *requests*, šalje se *POST* zahtjev na *API* s odgovarajućim datoetkama i zaglavljima (moraju sadržavati autorizacijski token). Nakon slanja zahtjeva, *API* vraća odgovor koji sadrži rezultate prepoznavanja tablica. Iz odgovora se izvlače prepoznate registarske tablice i dodaju u *registration_list*.

```
def main_process(distance_sensor, camera):
    image_paths = capture_images(camera)
    process_image(image_paths)
```

Funkcija *main_process(distance_sensor, camera)* povezuje prethodne dvije funkcije i upravlja cijelim procesom rada s kamerama.

Sada kada je implementiran poslužiteljski sloj, korisnički sloj, baza podataka i Raspberry Pi u projket se uključuje Docker-compose. On omogućuje orkestraciju višestrukih Docker kontejnera za efikasno upravljanje aplikacijom. Pomoću jedne konfiguracijske datoteke, svi dijelovi aplikacije mogu se pokrenuti u izoliranim kontejnerima, čime se olakšava razvoj, testiranje i proizvodno okruženje.

```yaml
version: '3.8'
services:
  mysql:
    image: mysql:latest
    restart: always
    ports:
      - '3306:3306'
    environment:
      MYSQL_DATABASE: smart-parking
      MYSQL_ROOT_PASSWORD: password
    volumes:
      - mysql_data:/var/lib/mysql
      - ./mysql-init-scripts:/docker-entrypoint-initdb.d
  backend:
    build: ./smart_park-system
    ports:
      - '8082:8082'
    depends_on:
      - mysql
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/smart-parking
      SPRING_DATASOURCE_USERNAME: root
      SPRING_DATASOURCE_PASSWORD: password
  frontend:
    build: ./smart-parking-frontend
    ports:
      - '8080:8080'
    depends_on:
      - backend
volumes:
  mysql_data:
```

Datoteka docker-compose.yaml definira konfiguraciju za pokretanje više usluga koristeći Docker Compose. Konkretno, ova datoteka postavlja tri usluge: *mysql*, *backend* i *frontend*. Servis *mysql* koristi posljednju dostupnu verziju slike *mysql* i uvijek ponovno pokreće kontejner ako se zaustavi. Port 3306 na *host*-u mapira na port 3306 u kontejneru. U *environment*-u se postavlja okruženje za MySQL – kreira se baza podataka pod nazivom *smart-parking* i postavlja *root* lozinka na *password.* Postavlja se trajni volumen za podatke te se lokalna mapa *mysql-init-scripts* mapira na odgovarajuću mapu u kontejneru.
*Backend* servis gradi Docker sliku iz lokalne mape *./smart_park-syste,* dok port mapira na 8082. *Depends_on* osigurava da se MySQL usluga pokrene prije ovog servisa, a *environment* postavlja okruženje za Spring Boot aplikaciju uključujući *URL* baze podataka.
S druge strane, *frontend* servis pokreće se nakon *backend* servisa, mapira se na port 8080, a sliku gradi iz lokalne mape *./smart-parking-frontend*.

Finalno, u projekt se dodaje K3s. K3s omogućuje skalabilno upravljanje aplikacijom, automatizaciju implementacije, skaliranja i operacija kontejnera, osiguravajući pouzdanost i fleksibilnost sustava. Koristeći K3s, aplikacija može lako skalirati kako bi zadovoljila rastuće zahtjeve korisnika i upravljala distribuiranim resursima učinkovito.

```
FROM francoisgervais/opencv-python:latest

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    build-essential

RUN pip install --no-binary numpy numpy

RUN pip install RPi.GPIO gpiozero requests

COPY sensor-app.py ./

CMD ["python", "./sensor-app.py"]
```

Ovaj *Dockerfile* se koristi za izgradnju Docker kontejnera koji će pokretati Python aplikaciju na Raspberry Pi uređaju koja koristi *OpenCV* biblioteku, kao i druge Python module. *Francoisgervais/opencv-python:latest* je bazna Docker slika koji se koristi za OpenCV i Python, a verzija *latest* označava da se koristi najnovija dostupna verzija. Instalira se *numpy* paket s izvornim kodom umjesto prekompajliranih binarnih datoteka. Dodatni python moduli koji se instaliraju jesu *Rpi.GPIO* I *gpiozero* koji se koriste za interakciju s pinovima, dok se *requests* instalira kako bi bilo omogućeno slanje HTTP zahtjeva. Datoteka *sensor-app.py* se kopira unutar Docker kontejnera, a posljednja naredba pokreće skriptu kada se kontejner pokrene. Sljedeća naredba koju pozivamo je ***docker build -t sgal88/sgalk3s:1.0 -f Dockerfile .*** Kada se ova naredba izvrši, Docker će pronaći *Dockerfile* u trenutnom radnom direktoriju, slijediti upute u *Dockerfile*-u za izgradnju kontejnera, te će rezultirajuća Docker slika biti označena kao *sgal88/sgalk3s:1.0.*

Sljedeće što je potrebno jest izraditi račun na Docker Hub-u i stvoriti repozitorij pod imenom *sgalk3s* te poslati sliku sa naredbom ***docker push sgal88/sgalks3:1.0.***
Kada se ova naredba izvrši, Docker klijent će tražiti Docker sliku s navedenim imenom i verzijom u lokalnom Docker registru. Ako pronađe odgovarajuću sliku poslat će na odredišni

Docker Hub pod navedenim imenom i verzijom. Docker slika bit će dostupana na Docker Hub-u pod specificiranom lokacijom i verzijom, u ovom slučaju kao *sgal88/sgalk3s:1.0.*

Sljedeće je izgradnja *yml* datoteke koja sadrži definicije Kubernetes resursa koji će se stvoriti na klasteru.

Slika 10: raspberry.yaml (Igress i Servise)

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: sensor-ingress
spec:
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: sensor-service
                port:
                  number: 84
---
apiVersion: v1
kind: Service
metadata:
  name: sensor-service
spec:
  selector:
    app: sensor-app
  ports:
    - protocol: TCP
      port: 84
      targetPort: 8084
```

Izvor: Vlastita izrada

Slika 11: raspberry.yaml (Deployment)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sensor-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sensor-app
  template:
    metadata:
      labels:
        app: sensor-app
    spec:
      containers:
        - name: sensor-container
          image: sgal88/sgalk3s:1.0
          imagePullPolicy: Always
          securityContext:
            privileged: true
          env:
            - name: NODE_NAME
              valueFrom:
                fieldRef:
                  fieldPath: spec.nodeName
          ports:
            - containerPort: 8084
          volumeMounts:
            - mountPath: /dev/mem
              name: mem
            - mountPath: /dev/gpiomem
              name: gpiomem
      volumes:
        - name: mem
          hostPath:
            path: /dev/mem
        - name: gpiomem
          hostPath:
            path: /dev/gpiomem
```

Izvor: Vlastita izrada

Ova *YAML* datoteka sadrži definicije tri različita Kubernetes resursa: *Ingress*, *Service*, i *Deployment*.

*Ingerss* definira pravila za preusmjeravanje *HTTP* zahtjeva na odgovarajuće servise unutar klastera. Ime *Ingress* resursa je *sensor-ingress*. Specifikacija (*spec*) definira pravila za preusmjeravanje *HTTP* zahtjeva na temelju staza. Svaki put (*paths*) definira *URL* stazu koja će biti preusmjerena, a *backend* određuje koji servis će obraditi preusmjerene zahtjeve.

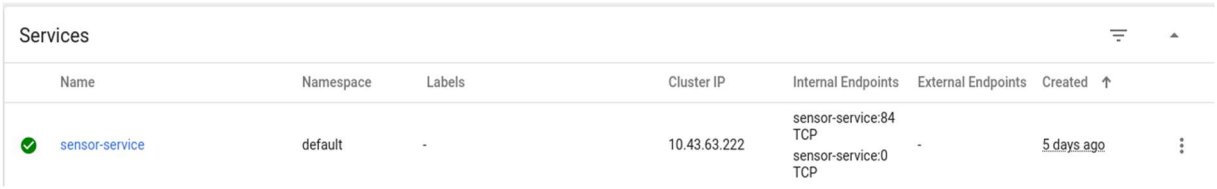
*Service* definira servis unutar klastera koji omogućava pristup aplikaciji, a ime mu je postavljeno na *sensor-service*. Specifikacija (*spec*) definira selektore za pronalaženje odgovarajućih *pod*-ova i konfiguraciju port-ova na kojima će servis slušati.

*Deployment* definira skup *pod*-ova (instanci aplikacije) koji se izvršavaju unutar klastera. Ime *deployment*-a je *sensor-deployment*. Specifikacija (*spec*) definira broj replika (instanci) aplikacije, selektore za pronalaženje odgovarajućih *pod*-ova i konfiguraciju kontejnera unutar

*pod*-ova. Obrazac (*template*) definira kako će se kreirati novi *pod*-ovi, uključujući oznake i definicije kontejnera. Kontejneri (*containers*) definiraju Docker slike, portove i volumene koje će koristiti *pod*-ovi.

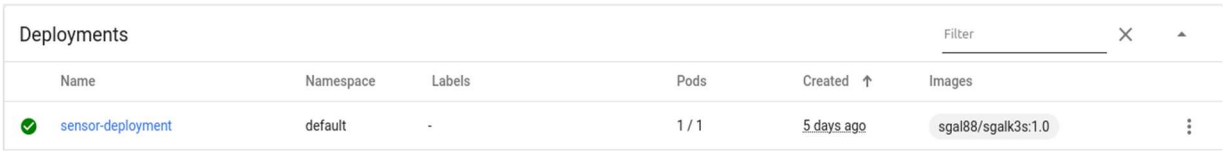
Naredba ***kubectl create -f raspberry.yaml*** koristi se za stvaranje Kubernetes resursa u klasteru na temelju definicija navedenih u YAML datoteci *raspberry.yaml (Ingress, Service, i Deployment).*

Slika 12: Sensor-service




Izvor. Vlastita izrada

Slika 13: Sensor-deployment




Izvor: Vlastita izrada

Sljedeće na redu je izgradnja *YAML* datoteke za bazu podataka (MySQL).

Slika 14: mysql.yml (PersistentVolume I PersistentVolumeClaim)

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /var/lib/mysql
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Izvor: Vlastita izrada

Slika 15: mysql.yml (Service)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  selector:
    app: mysql
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
      nodePort: 30000
  type: NodePort
```

Izvor: Vlastita izrada

Prvi dio *mysql.yml* datoteke definira dva Kubernetes resursa: *PersistentVolume* (*PV*) i *PersistentVolumeClaim* (*PVC*). Ovi resursi se koriste za upravljanje trajnijim skladištem podataka za MySQL bazu podataka.

Kada se *PersistentVolume* resurs kreira, Kubernetes dodaje definirano skladište u klaster. U ovom slučaju, to je lokalni direktorij */var/lib/mysql* na *host*-u s kapacitetom od 1 *GiB*.

Kada se *PersistentVolumeClaim* resurs kreira, Kubernetes traži dostupni *PersistentVolume* koji zadovoljava zahtjeve *PVC*-a (kapacitet i način pristupa). Ako nađe odgovarajući *PV* (u ovom slučaju *mysql-pv*), *PVC* će biti vezan za taj *PV*.

*Service* izlaže MySQL bazu na *port*-u 3306 unutar klastera. Koristi *NodePort* tip *Service*-a za izlaganje MySQL baze na portu 30000 svakog čvora u klasteru, omogućavajući pristup izvana (izvan klastera) putem *IP* adrese bilo kojeg čvora i specificiranog porta (30000).

Slika 16: mysql.ym (Deployment)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:latest
          ports:
            - containerPort: 3306
          env:
            - name: MYSQL_DATABASE
              value: smart-parking
            - name: MYSQL_ROOT_PASSWORD
              value: password
          volumeMounts:
            - name: mysql-persistent-storage
              mountPath: /var/lib/mysql
            - name: mysql-init-scripts
              mountPath: /docker-entrypoint-initdb.d
      volumes:
        - name: mysql-persistent-storage
          persistentVolumeClaim:
            claimName: mysql-pvc
        - name: mysql-init-scripts
          hostPath:
            path: /docker-entrypoint-initdb.d
```

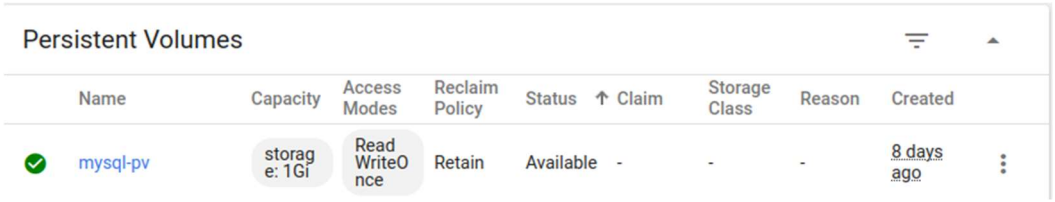
Izvor: Vlastita izrada

Kontejner *mysql* za pokretanje koristi sliku *mysql:latest*. *Port* na kojem MySQL baza podataka sluša unutar kontejnera je postavljen na 3306. Zatim se postavljaju varijable okruženja. Varijabli *MYSQL_DATABASE* pridružuje se vrijednost *smart-parking*, dok se varijabli *MYSQL_ROOT_PASSWORD* pridružuje vrijednost *password*. *VolumeMounts* montira trajno skladište (*mysql-persistent-storage*) na */var/lib/mysql* unutar kontejnera, što osigurava da podaci MySQL baze ostanu trajni čak i nakon ponovnog pokretanja kontejnera. *Volumes* montira *mysql-init-scripts* na */docker-entrypoint-initdb.d* omogućavajući prilagodbu MySQL inicijalizacije.

***kubectl apply -f mysql.yml***

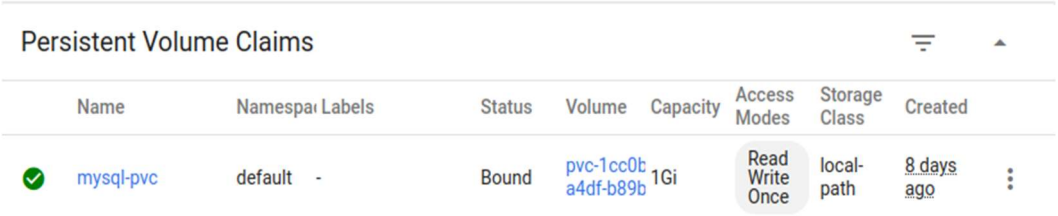
Nakon izvršetka ove naredbe svi resursi definirani u *mysql.yml* datoteci su kreirani u Kubernetes klasteru, omogućujući postavljanje i korištenje MySQL baze podataka.

Slika 17: mysql.yml (Persistent Volume)



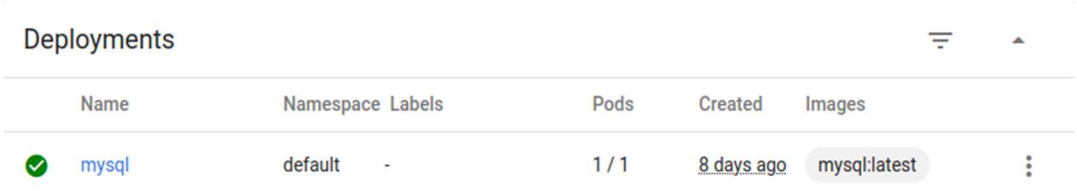

Izvor: Vlastita izrada

Slika 18: mysql.yml (Persistent Volume Claim)



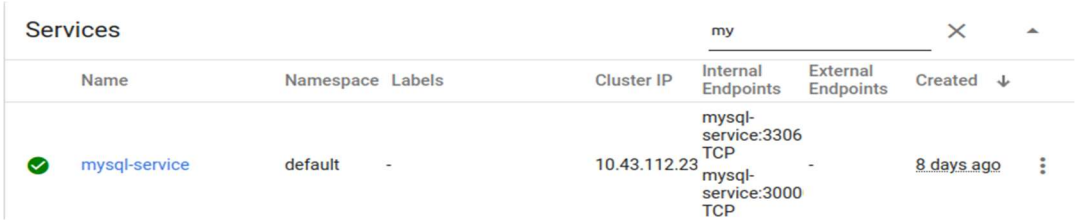

Izvor: Vlastita izrada

Slika 19: mysql.yml (Deployment)




Izvor: Vlastita izrada

Slika 20: mysql.yml (Service)




Izvor: Vlastita izrada

Slijedi izgradnja Dockerfile-a i YAML datoteke za komponente poslužiteljskog sloja.

```
FROM arm32v7/ubuntu:latest

ENV JAVA_HOME /user/lib/jvm/java-11-openjdk-armhf
ENV PATH $JAVA_HOME/bin:$PATH
ENV MAVEN_HOME /usr/share/MAVEN_HOME
ENV PATH $MAVEN_HOME/bin:$PATH

RUN apt-get update && \
    apt-get install -y openjdk-11-jdk

RUN apt-get install -y maven

RUN apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

WORKDIR /app

CMD ["bash"]
```

Ovaj Dockerfile određuje *arm32v7/ubuntu* kao baznu sliku koja će s koristiti za izgradnju nove Docker slike koja će biti spremljena kao *sgal88/backend:1.0.* Koristeći *arm32v7* kao baznu sliku u Dockerfile-u osigurava se da je izgrađena Docker slika kompatibilna s *ARM* arhitekturom procesora koji se nalaze u Raspberry Pi uređajima.
*JAVA_HOME* je varijabla okruženja koja ukazuje na direktorij gdje je instalirana JDK 11 verzija za ARM arhitekturu. Zatim se dodaje *bin* direktorij iz *JAVA_HOME* varijable u *PATH*, što omogućava izvršavanje Java naredbi bez potrebe za potpunom putanjom. *MAVEN_HOME* ukazuje na direktorij gdje je instaliran Maven i nakon toga se dodaje *bin* direktorij iz *MAVEN_HOME* varijable u *PATH*, što omogućava izvršavanje Maven naredbi. Ažuriraju se paketne liste (*apt-get update*) i instalira JDK 11 (*openjdk-11-jdk*). Instalira se Maven alat pomoću *RUN apt-get install -y maven.* Sljedeća linija čisti privremene datoteke koje su preostale nakon instalacije paketa kako bi se smanjila veličina Docker slike. Posljednja linija definira zadanu naredbu koja će se izvršiti kada se kontejner pokrene. U ovom slučaju, to je *bash* ljuska, što znači da će se kontejner pokrenuti s interaktivnom *bash* ljuskom.

***docker build -t sgal88/backend:1.0 -f Dockerfile .***
Kada se ova naredba izvrši, Docker će pročitati *Dockerfile*, prikupiti sve potrebne datoteke iz konteksta izgradnje i izgraditi Docker sliku prema uputama navedenim u *Dockerfile*-u. Nakon uspješne izgradnje, slika će biti označena imenom *sgal88/backend* i verzijom 1.0.

***docker push sgal88/backend:1.0***

Nakon uspješnog izvršenja naredbe, slika s*gal88/backend:1.0* bit će dostupna na registru, odakle ju drugi korisnici mogu preuzeti (*pull*) i koristiti.

```
FROM sgal88/backend:1.0

COPY target/smart_park-system-0.0.1-SNAPSHOT.jar /app/smart_park-system.jar

CMD ["java", "-jar", "smart_park-system.jar"]
```

Ovaj *Dockerfile* koristi već postojeću Docker sliku kao bazu i nadograđuje ju dodavanjem novog Java arhiva (*JAR*) te postavlja zadanu naredbu za pokretanje aplikacije.

Slika 21: Backend.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: smart-park-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: smart-park-system
  template:
    metadata:
      labels:
        app: smart-park-system
    spec:
      containers:
        - name: smart-park-system
          image: sgal88/backend:3.0
          ports:
            - containerPort: 8082
          env:
            - name: SPRING_DATASOURCE_URL
              value: jdbc:mysql://172.16.1.18:30000/smart-parking
            - name: SPRING_DATASOURCE_USERNAME
              value: root
            - name: SPRING_DATASOURCE_PASSWORD
              value: password
            - name: SPRING_JPA_HIBERNATE_DDL_AUTO
              value: update
            - name: SPRING_JPA_SHOW_SQL
              value: "true"
---
apiVersion: v1
kind: Service
metadata:
  name: smart-park-system
spec:
  type: LoadBalancer
  ports:
    - port: 8082
      targetPort: 8082
  selector:
    app: smart-park-system
```

Izvor: Vlastita izrada

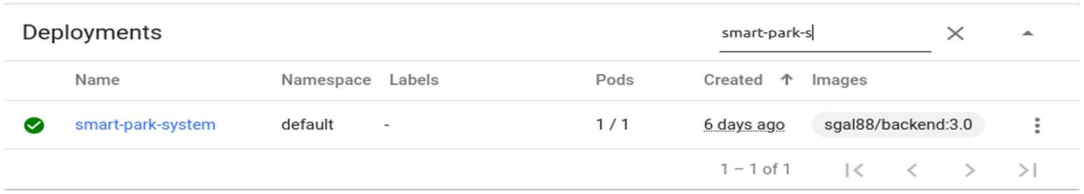
Gore navedena *backend.yml* datoteka definira dva Kubernetes resursa: *Deployment* i *Service* za aplikaciju pod nazivom *smart-park-system*. Definira se *API* verzija koja se koristi za stvaranje ovog resursa. *apps/v1* je standardna verzija za *Deployment* resurse.

Kao ime *Deployment*-a se postavlja *smart-park-system.* Broj replika (*pod*-ova) koji će se koristiti se postavlja na 1. *Selector* određuje kako će se identificirati *pod*-ovi kojima upravlja ovaj *Deployment*, dok *matchLabels* predstavlja labelu koju *pod*-ovi moraju imati da bi se njima moglo upravljati. Kao ime kontejnera postavlja se *smart-park-system.* Docker slika koja će se koristiti za pokretanje kontejnera je *sgal88/backend:3.0* koja je građena na ranije navedenoj slici *sgal88/backend:1.0*. *Port* na kojem aplikacija sluša je 8082. Sljedeće na redu je definiranje varijabli okruženja za konfiguraciju aplikacije. Postavlja se *URL* baze podataka, korisničko ime i lozinka, postavka *Hibernate DDL* automatskog ažuriranja i postavka za prikaz *SQL* upita u konzoli.
*ApiVersion* definira *API* verziju koja se koristi za stvaranje ovog resursa. *V1* je standardna verzija za *Service* resurse. *LoadBalancer* omogućuje da *Service* dobije vanjsku *IP* adresu. *Port:8082* predtavlja *port* na kojem će *Service* biti dostupan, dok *targetPort:8082* predstavlja *port* na koji će *Service* prosljeđivati promet unutar *pod*-ova.

***kubectl apply -f backend.yml***

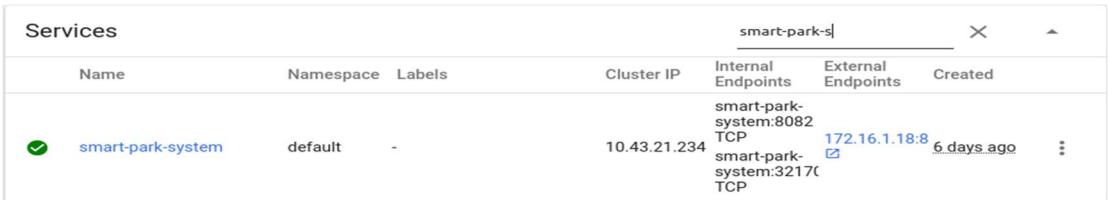
Rezultat izvršavanja ove naredbe je da će Kubernetes klaster biti konfiguriran tako da pokreće aplikaciju prema definicijama u *backend.yml* datoteci, osiguravajući da aplikacija bude dostupna putem mreže.

Slika 22: Backend-deployment




Izvor: Vlastita izrada

Slika 23: Backend-services




Izvor: Vlastita izrada

Nakon što je postavljen Deployment i Service za komponente poslužiteljskog sloja potrebno je izraditi Dockerfile i YAML datoteku za Vue aplikaciju.

```
FROM arm32v7/node:latest

WORKDIR /app

COPY package*.json ./

RUN npm install

RUN npm install cors

COPY . .

EXPOSE 8080

CMD ["npm", "run", "serve"]
```

U ovom *Dockerfile*-u se koristi *arm32v7/node:latest*, što znači da se koristi najnovija verzija Node.js slike optimizirana za arhitekturu *ARMv7* koji je kompatibilan s *ARM* arhitekturom procesora koji se nalaze u Raspberry Pi uređajima. Radni direktorij u kontejneru se postavlja na */app.* To je mjesto gdje će se kopirati svi izvorni kodovi aplikacije i gdje će se izvršavati naredbe vezane uz izgradnju i pokretanje aplikacije. Kopiraju se *package.json* i p*ackage-lock.json* datoteke iz lokalnog direktorija u trenutni radni direktorij u kontejneru (*/app*). Izvršava se *npm install* unutar kontejnera kako bi se instalirali svi potrebni paketi navedeni u *package.json* datoteci. Instalira se i *cors* paket koji omogućuje kontrolu pristupa resursima iz različitih izvora. Zatim se kopira sav preostali aplikacijski kod (uključujući JavaScript datoteke, konfiguracijske datoteke, druge datoteke i direktorije) iz lokalnog direktorija u trenutni radni direktorij u kontejneru (*/app*). Aplikacija će unutar kontejnera biti dostupna na *port*-u 8080. Posljednja naredba pokreće razvojni server za aplikaciju.

***docker build -t sgal88/frontend:4.0 -f Dockerfile .***

Nakon izvršenja ove naredbe, bit će dostupana Docker slika *sgal88/frontend:4.0* spremana za pokretanje kontejnera.

***docker push sgal88/frontend:4.0***

Kada se ova naredba izvrši, Docker Engine će poslati Docker sliku s*gal88/frontend:4.0* na registriranu udaljenu lokaciju, omogućavajući drugim korisnicima ili sustavima da preuzmu tu sliku i pokrenu kontejnere temeljene na njoj.

Slika 24: frontend.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: smart-parking-frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: smart-parking-frontend
  template:
    metadata:
      labels:
        app: smart-parking-frontend
    spec:
      containers:
        - name: frontend
          image: sgal88/frontend:4.0
          ports:
            - containerPort: 8080

---
apiVersion: v1
kind: Service
metadata:
  name: smart-parking-frontend
spec:
  selector:
    app: smart-parking-frontend
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  type: LoadBalancer
```

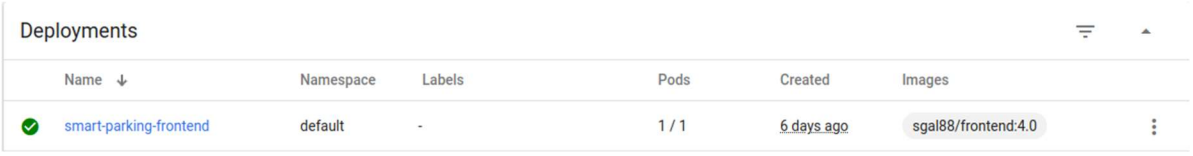
Izvor: Vlastita izrada

API verzija koja se koristi za ovaj objekt je *app/v1*. *Metadata* sadrži podatke o *Deployment*-u, uključujući ime *smart-parking-frontend. Spec* sadrži specifikaciju gdje se broj replika postavlja na 1, a *matchLabels* definira da se *Deployment* odnosi na *pod*-ove koji imaju oznaku *smart-parking-frontend.* Kontejner pod nazivom *frontend* koristi prethodno izgrađenu sliku *sgal88/frontend:4.0* i biti dostupan na *port*-u 8080.
*Service* dio definira *v1* kao verziju *API*-ja koja se koristi za ovaj objekt. *Service* omogućava vanjski pristup *frontend* aplikaciji putem *LoadBalancer*-a.

***kubectl apply -f frontend.yml***

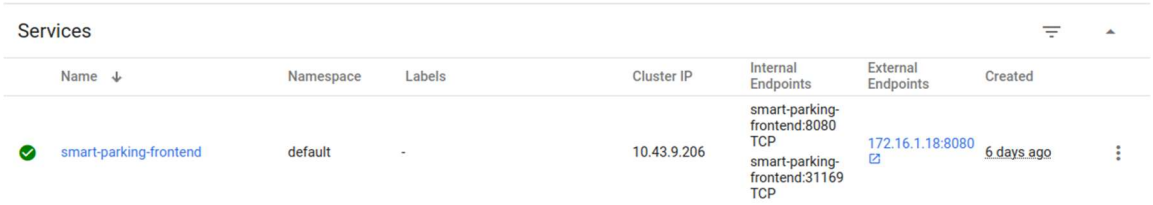
U ovom slučaju, naredba će stvoriti *Deployment* i *Service* za *frontend* aplikaciju s nazivom *smart-parking-frontend* unutar Kubernetes klastera, omogućujući pokretanje i pristup *frontend* aplikaciji. Nakon izvršenja naredbe, *frontend* aplikacija će biti dostupna unutar klastera.

Slika 25: Frontend-deployment




Izvor: Vlastita izrada

Slika 26: Frontend-service




Izvor: Vlastita izrada

Finalno kako bi sva komunikacija bila moguća potreban je NGINX.

NGINX je softver web poslužitelja otvorenog koda koji se koristi za obrnuti proxy (*reverse proxy*), balansiranje opterećenja i predmemoriju. Pruža mogućnosti HTTPS poslužitelja i uglavnom je dizajniran za maksimalnu izvedbu i stabilnost. Također funkcionira kao proxy poslužitelj za komunikacijske protokole e-pošte, kao što su IMAP, POP3 i SMTP.
Korištenje NGINX-a donosi nekoliko prednosti. Smanjuje vrijeme čekanja za učitavanje web stranice. Nije potrbno brinuti o visokoj latenciji na web stranicama. Ubrzava radne karakteristike usmjeravanjem prometa na web poslužitelje na način koji povećava ukupnu brzinu.
Djeluje kao jeftin i robustan balanser opterećenja, nudi skalabilnost i mogućnost obrade istodobnih zahtjeva I omogućuje nadogradnju u hodu bez prekida rada.

```
events {}
http {
  server {
    listen 8083 default_server reuseport;
    listen [::]:8083 default_server reuseport;
    client_max_body_size 100M;
    server_name localhost;
    location / {
      proxy_pass http://172.16.1.18:8080;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      client_max_body_size 100M;
      proxy_set_header X-Forwarded-Proto $scheme;
      proxy_buffering off;
    }
    location /api {
      proxy_pass http://172.16.1.18:8082;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      client_max_body_size 100M;
      proxy_set_header X-Forwarded-Proto $scheme;
      proxy_buffering off;
    }
  }
}
```

Ovaj kod predstavlja konfiguracijsku datoteku za NGINX web poslužitelj. *Events* definira globalne postavke događaja za NGINX. To je obavezni blok u konfiguracijskim datotekama NGINX-a, ali u ovom slučaju je prazan. *Http* blok definira konfiguraciju za *HTTP* poslužitelj. Linija *listen 8083 default_server reuseport* definira da *HTTP* poslužitelj sluša na *port*-u 8083. Opcija *default_server* označava da je ovo zadan poslužitelj za obradu zahtjeva koji ne odgovaraju drugim poslužiteljima. Opcija *reuseport* omogućava višestruko korištenje istog *port*-a, što može poboljšati radne karakteristike u određenim situacijama. Linija nakon definira da *HTTP* poslužitelj sluša na *port*-u 8083.

*Location /* definira pravila za rukovanje zahtjevima upućenim na osnovni *URL* (/). Svi zahtjevi koji odgovaraju ovom *URL*-u bit će proslijeđeni na adresu *http://172.16.1.18:8080* putem *proxy*-ja. Ostale linije definiraju postavke *proxy*-ja za proslijeđene zahtjeve, poput dodavanja određenih zaglavlja (*X-Forwarded-For*, *Host*, *X-Forwarded-Proto*) i kontrolu veličine tijela zahtjeva.

*Location /api* definira pravila za rukovanje zahtjevima upućenim na *URL*-u */api*. Svi zahtjevi koji odgovaraju ovom *URL*-u bit će proslijeđeni na drugu adresu putem *proxy*-ja, u ovom slučaju na *http://172.16.1.18:8082.* Kao i u prethodnom bloku, ostale linije definiraju postavke *proxy*-ja za proslijeđene zahtjeve.

Slika 27: nginx.ym (Deployment)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25.2-alpine
          volumeMounts:
            - name: nginx-config
              mountPath: /etc/nginx/nginx.conf
      volumes:
        - name: nginx-config
          hostPath:
            path: /home/sgal/nginx/nginx.conf
            type: File
```

Izvor: Vlastita izrada

Slika 28: nginx.ym (Service)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - port: 8083
      targetPort: 8083
  type: LoadBalancer
```

Izvor: Vlastita izrada

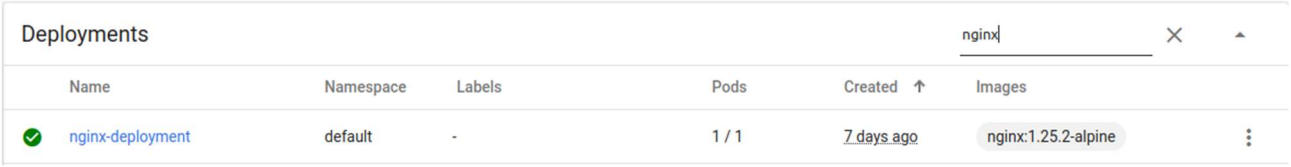
*Metadata* sadrži metapodatke o *Deployment*-u, uključujući ime *nginx-deployment.* U specifikaciji se definira jedna replika, dok *matchLabels: app: nginx* definira da se *Deployment* odnosi na *pod*-ove koji imaju oznaku *app: nginx*. Kontejner koji će se pokrenuti nazvan je *nginx* a sliku koju će koristiti definiramo na *nginx:1.25.2-alpine*. Putanja na kojoj će se volumen postaviti unutar kontejnera je */etc/nginx/nginx.conf. Volumens* s druge strane definira volumen koji će biti postavljen, *hostPath* specificira stvarnu putanju do datoteke na *host* sustavu koja će biti korištena kao konfiguracijska datoteka unutar kontejnera.

Service unutar *metadata* sadrži metapodatke o *Service*-u, uključujući ime *nginx-service. Selector* specificira kriterije za odabir *pod*-ova na koje će ovaj *Service* usmjeravati promet, u ovom slučaju to su *pod*-ovi s oznakom *nginx*. *Port* označava da *Service* sluša na *port*-u 8083, a *targetPort* označava da će promet koji stigne na *port* 8083 biti usmjeren na isti port na *pod*-ovima. *LoadBalancer* označava da će ovaj *Service* biti izložen izvan klastera kao *LoadBalancer*, omogućujući vanjski pristup NGINX poslužitelju.

***kubectl apply -f nginx.yml***

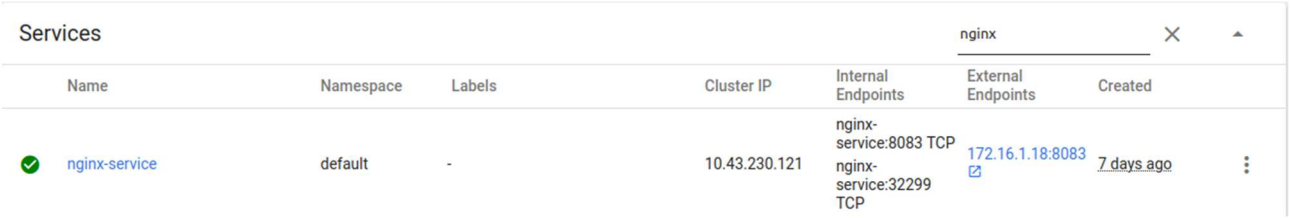
Nakon izvršetka ove naredbe konfiguracija definirana u datoteci *nginx.yml* se analizira. Drugim riječima, kubernetes klaster primjenjuje definiranu konfiguraciju tako što kreira resurse (*Deployment* i *Service*) kako je specificirano.

Slika 29: Nginx-deployment




Izvor: Vlastita izrada

Slika 30: Nginx-service




Izvor: Vlastita izrada

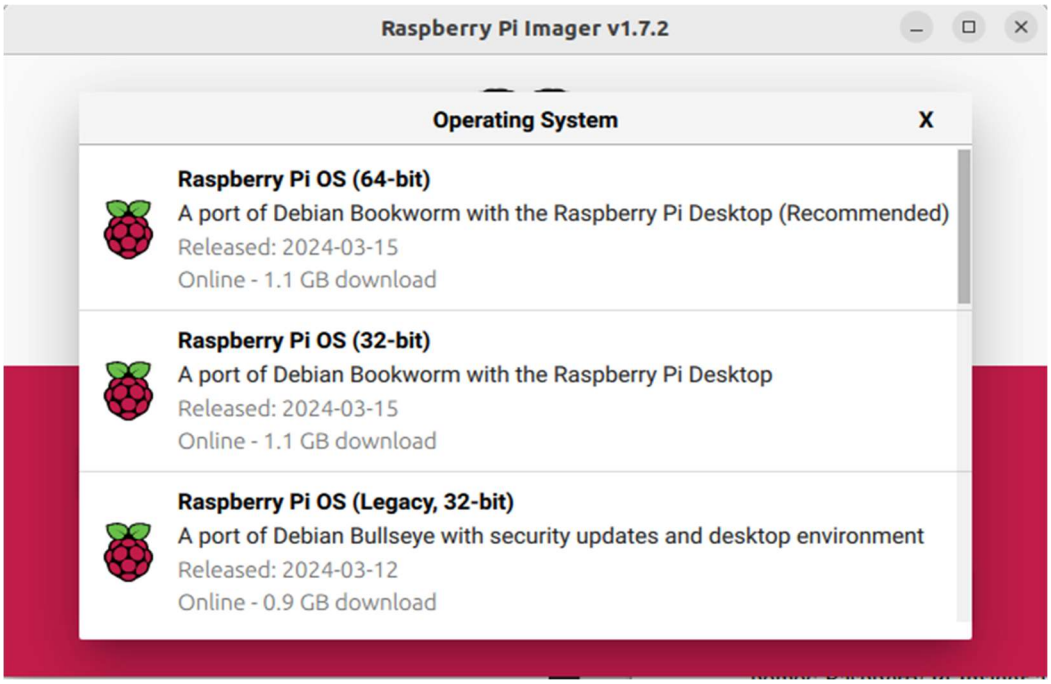
# 4. Korisnička dokumentacija

## 4.1. Raspberry Pi i komponente

Da bi se započelo s postavljanjem Raspberry Pi  uređaja bit će potrebno sljedeće: napajanje i medij za pokretanje (microSD kartica s dovoljno prostora za pohranu). Raspberry Pi se može postaviti kao interaktivno računalo sa radnom površinom ili kao bezglavo (*headless*) računalo dostupno samo preko mreže. Da bi bezglavo postavili svoj Raspberry Pi, nisu potrebni nikakvi dodatni periferni uređaji. Unaprijed konfiguriramo ime domaćina (*hostname*), korisnički račun, mrežnu vezu i *SSH* konekciju. Ukoliko želimo izravno koristiti Raspberry Pi potreban je sljedeći pribor: zaslon, tipkovnica i miš. U ovom radu Raspberry Pi postavljen je bezglavno.

Za napajanje se koristi Raspberry Pi 15W USB-C, napona barem 5V, uz 800 mA struje i snage 4 W.  Nedovoljna snaga uzrokuje nepravilan rad računala i čudno ponašanje uređaja. Računalo se priključuje Ethernet kabelom u Ethernet ulaz koji se nalazi pored *USB* priključaka. Za početak, potrebno je umetnuti *SD* karticu u utor za *SD* karticu na računalu, ako računalo nema utor, moguće je koristiti *SD* čitač kartice (*card reader*). Preporučuje se korištenje *SD* kartice s najmanje 32 GB a najviše 2TB prostora za pohranu. Kapaciteti iznad 2TB trenutno nisu podržani zbog ograničenja u *MBR*-u.
Da bismo koristili uređaj trebat će instalirati operativni sustav. Prema zadanim postavkama Raspberry Pi provjerava operativni sustav na bilo kojoj *SD* kartici umetnutoj u utor za *SD* karticu. U ovom radu kao operativni sustav koristit će se Raspberry Pi OS (*Debian*) koji će biti instaliran uz pomoć *Raspberry Pi Imager*-a.

Slika 31: Odabir operativnog sustava

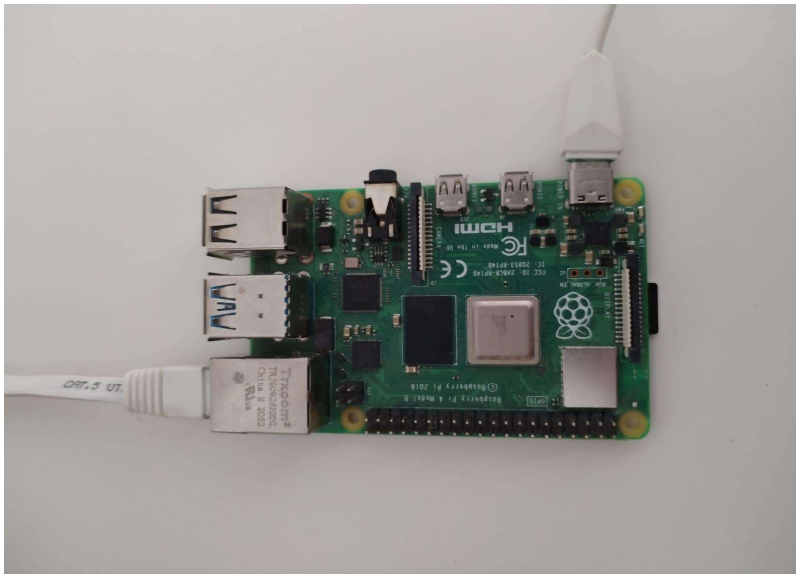



Izvor: Vlastita izrada

*Raspberry Pi OS* je besplatni operativni sustav temeljen na *Debian*-u, optimiziran za Raspberry Pi hardver i preporučeni je operativni sustav za normalnu upotrebu na Raspberry Pi-ja. Kada se operativni sustav uspješno instalirao na *SD* karticu dodajemo još jednu vrlo bitnu datoteku koju spremamo kao *ssh. SSH* je mrežni protokol koji korisnicima omogućuje

uspostavu sigurnog komunikacijskog kanala između dva računala putem računalne mreže. *SD* kartica je spremna i možemo ju umetnuti u utor za *SD* karticu na Raspberry Pi uređaju. Zatim spajamo uređaj na internet putem Ethernet kabla. Finalno spajamo napajanje na priključak za napajanje Raspberry Pi uređaja.

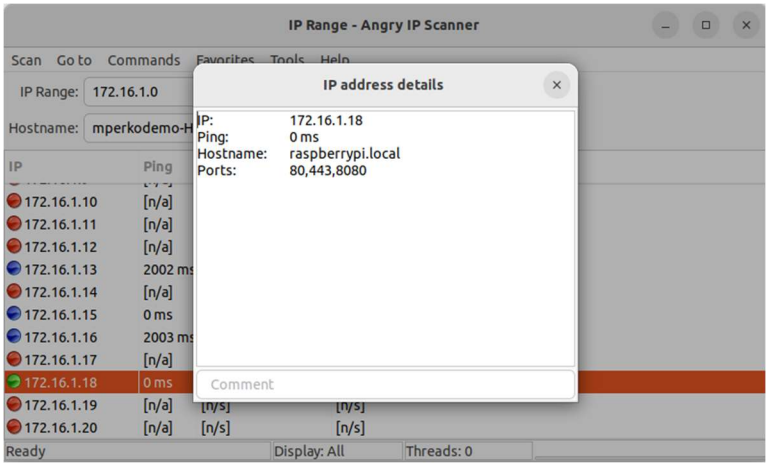
Slika 32: Spajanje Ethernet-a i napajanja



Izvor: Vlastita izrada

Zatim, pomoću IP skenera (*Angry IP Scanner*) pronalazimo *IP* adresu Raspberry Pi uređaja.

Slika 33: Pronalaženje IP adrese




Izvor: Vlastita izrada

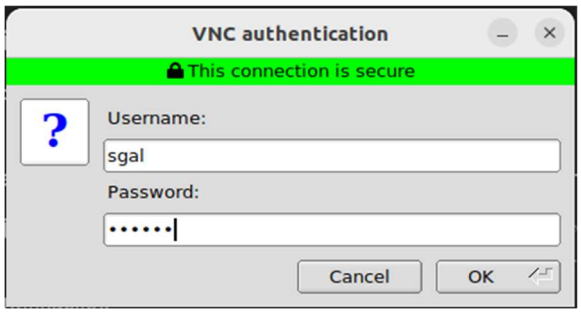
Sljedeći korak je povezivanje preko *ssh* sesije. Korisničko ime je postavljeno na *sgal*.

Slika 34: Uspješna SSH konekcija


Izvor: Vlastita izrada

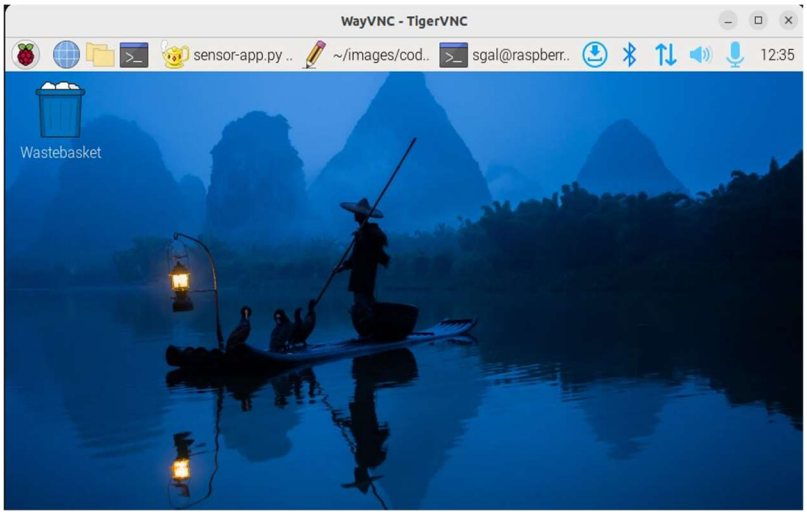
Drugi način, kako bismo pristupili grafičkom korisničkom sučelju preko mreže, koristit će se TigerVNC Viewer. TigerVNC Viewer je klijentski softver za pristupanje udaljenim računalima putem Virtual Network Computing (*VNC*) protokola.

Slika 35: VNC autentifikacija




Izvor: Vlastita izrada

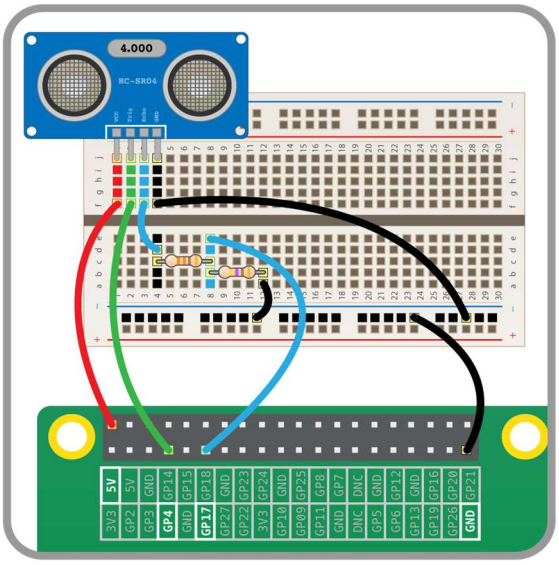
Slika 36: Prikaz grafičkog sučelja




Izvor: Vlastita izrada

Nakon uspješnog povezivanja na Raspberry Pi na redu je osposobljavanje senzora i servo motora koji će upravljati ulazom i izlazom pametnog parkirališta.

Slika 37: Povezivanje senzora

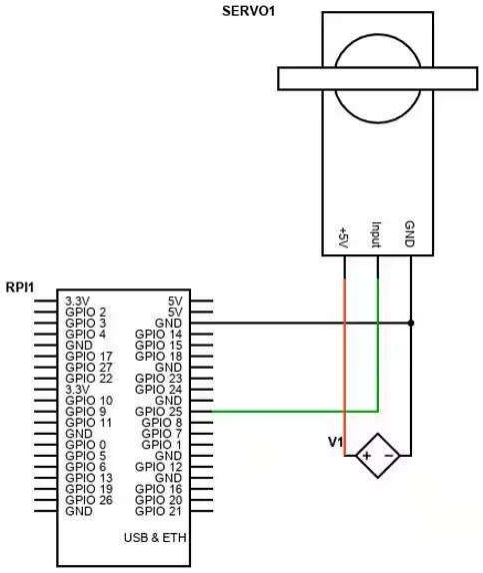



Izvor: https://projects.raspberrypi.org/en/projects/physical-computing/12

Krug se spaja na dva GPIO pina (jedan za *echo*, jedan za *trigger*), pin za uzemljenje i 5V pin. Potrebno je koristiti I dva otpornika (330Ω i 470Ω) budući da ultrazvučni sensor ima 5 volti otpornici smanjuju na 3,3 volta Koliko i podržavaju GPIO pinovi.

Nakon što su senzori postavljeni, postavljaju se i instaliraju servo motori.
Obično servo motori imaju tri ulaza, a dva od njih napajaju motor smješten unutar plastičnog tijela. Treći ulaz kontrolira koliko se servo okreće.

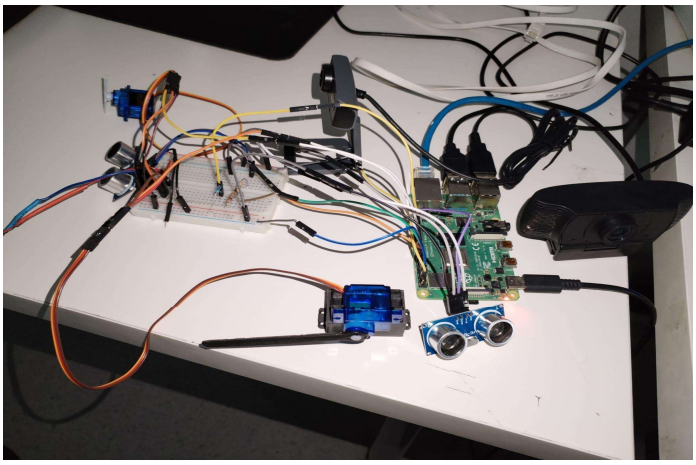
Slika 38: Povezivanje servo motora




Izvor: https://digikey.com/en/maker/tutorials/2021/how-to-control-servo-motors-with-a-raspberry-pi

Da bi Raspberry Pi upravljao servo motorom, potrbno je spojiti +5 V i GND vodove servo uređaja na vanjsko napajanje, a preostalu signalnu žicu na bilo koji I/O pin Raspberry Pi-ja. U ovom radu kao vanjsko napajanje korišten je USB kabal.

Slika 39: Pametno parkiralište sa svim komponentama

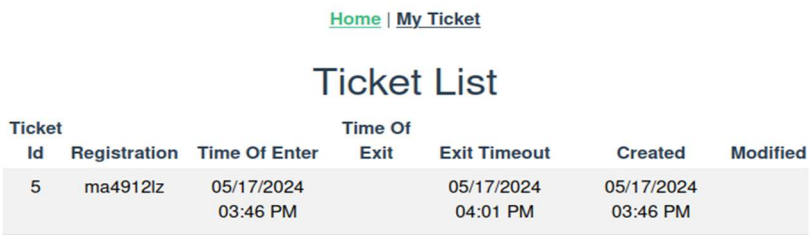


Izvor: Vlastita izrada

Finalno rad izgleda ovako. Sve komponente su spojene uz pomoć eksperimentalne ploče (*Breadboard*) i raspodjeljene na dva dijela: ulaz i izlaz pametnog parkirališta.

## 4.2. Korisničko sučelje

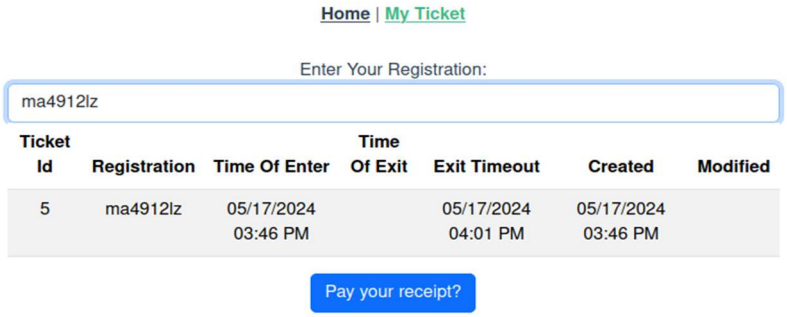
Slika 40: Home




Izvor: Vlastita izrada

Prikaz *Home* rute na kojoj se izlistavaju sve parkirne karte spremljene u bazu podataka. Trenutno se u bazi nalazi samo jedna. *Exit Timeout* postavljen je na 15 minuta, odnosno korisniku je omogućen besplatan izlaz nakon maksimalno 15 minuta provedenih na parkingu.

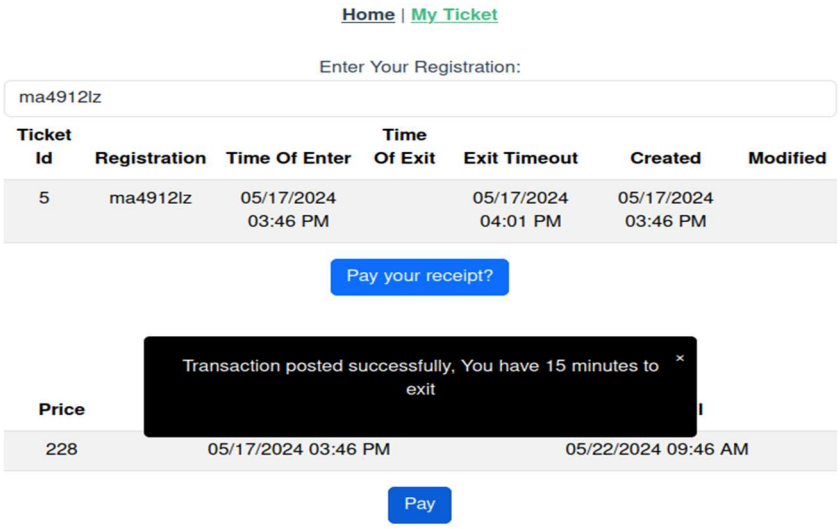
Slika 41: My Ticket




Izvor: Vlastita izrada

Na ruti *My Ticket* korisnik u polje za pretragu upisuje svoju registracijsku oznaku. Ukoliko želi napustiti parking prvo mora platiti sate provedene na parkingu.

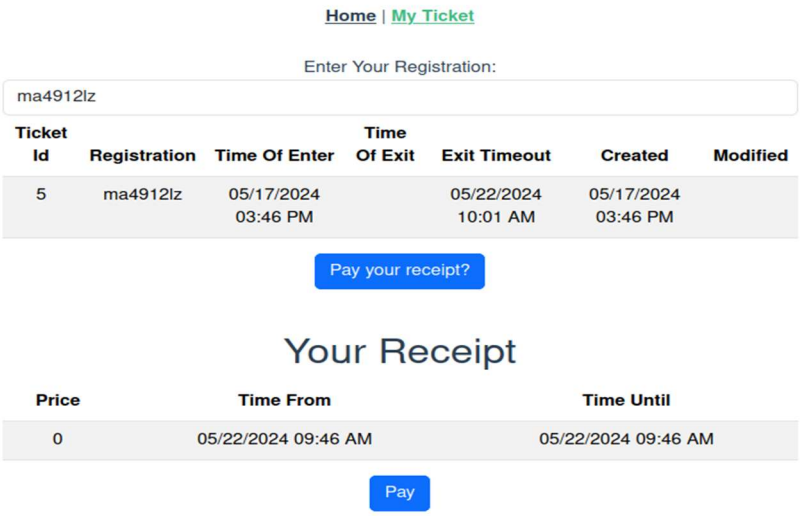
Slika 42: Plaćanje računa




Izvor: Vlastita izrada

Korisnik je na parkingu od ulaska do sada proveo 114 sati (cijena po satu je konstantna i iznosi 2) što daje ukupan iznos od 228. Korisnik nakon plaćanja može vidjeti je li plaćanje bilo uspješno ili nije. U primjeru gore navedeno je uspješno plaćanje te se korisniku automatski daje 15 minuta za izlaz sa parkirališta.

Slika 43: Novi Exit Timeout




Izvor: Vlastita izrada

Nakon plaćanja računa korisnik dobiva novo vrijeme (*Exit Timeout*) do kada može napustiti parkirng. Kada korisnik dođe do izlaza kamera ponovo snima registracijske oznake te se radi provjera može li korisnik izaći, ukoliko je platio račun povratna vrijednost bit će *true.* Rampa se podiže i korisniku omogućava izlaz te se šalje POST metoda koja sprema vrijeme izlaza (*Time Of Exit*).

## 4.3. Priprema sustava K3s

Standardne verzije Raspberry Pi sustava zadano *cgroup* imaju isključen. Potrebno ga je uključiti iz razloga što K3s treba *cgroups* za pokretanje *systemd* servisa. Potrebno je urediti datoteku *cmdline.txt* sljedećom naredbom: ***sudo nano /boot/firmware/cmdline.txt***
Na kraj reda ove datoteke dodajemo ***cgroup_memory=1 cgroup_enable=memory.***

Operacijski sustav na Raspberry Pi uređaju prema zadanim postavkama koristi *nftables* umjesto *iptables*. K3S mrežne značajke zahtijevaju *iptables*, stoga se podešava njegova konfiguracija.
Potrebno je ponovno pokretanje sustava kako bi konfiguracija bila uspješno postavljena. Za to koristimo naredbu ***sudo reboot***.

Da bi se jednostavno i brzo instalirao i pokrenio sustav K3s koristi se sljedeća naredba:
***curl -sfL https://get.k3s.io | sh --***

Ova skripta je instalirala K3s binarnu datoteku, kreirala potrebne simboličke poveznice, dodala skripte za upravljanje K3s, konfigurirala *environment* datoteku, postavila *systemd* servisnu datoteku i omogućila K3s servis za automatsko pokretanje. Sve ključne komponente su postavljene u */usr/local/bin* za izvršne datoteke i /*etc/systemd/system* za *systemd* konfiguracijske datoteke.
Kada je naredba završila već postoji postavljen i pokrenut klaster sa jednim čvorom što se može provjeriti naredbom ***sudo kubectl get node***.

Slika 44: Prikaz čvora

```
sgal@raspberrypi:~ $ sudo kubectl get nodes
NAME          STATUS   ROLES                  AGE   VERSION
raspberrypi   Ready    control-plane,master   17d   v1.29.4+k3s1
```

Izvor: Vlastita izrada

Ukoliko lokalno želimo pregledati ili modificirati klaster, svaki put je potrebna *ssh* konekcija što baš i nije praktično. Ono što je potrebno jest prenjeti *kubectl* na lokalno računalo. Da bi pristupili informacijama prvo je potrebno povezati se preko *ssh* konekcije. Uzimaju se informacije o konfiguraciji glavnoga čvora. Informacijama se pristupa naredbom
***sudo cat /etc/rancher/k3s/k3s.yaml***

Slika 45: Konfiguracija kubectl


Izvor: Vlastita izrada

Zatim na lokalnom računalu izrađujemo direktorij za konfiguraciju s imenom *.kube*. Kopirana konfiguracija se sprema kao */.kube/config*. Datoteku je potrbno urediti tako da se redak *server: https://localhost:6443* promjeni u *https://172.16.1.18:6443*.

Slika 46: kube/config

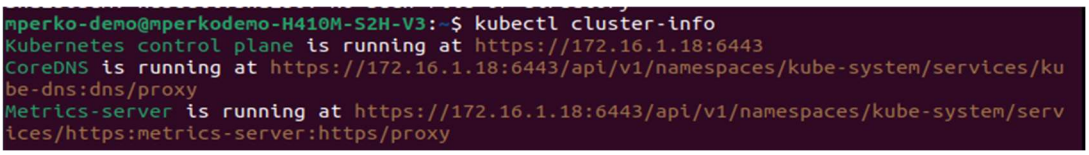

Izvor: Vlastita izrada

Nakon toga porebna je instalacija *kubectl*-a na lokalno računalo. Potrebo je preuzimanje kubectl binarne datoteke što omogućava sljedeća komanda:

***curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"***

Kako bismo provjerili je li *kubectl* ispravno konfiguriran pokreće se naredba ***kubectl cluster-info***, time dobivamo stanje klastera.

Slika 47: Stanje klastera




Izvor: Vlastita izrada

Kontrolne ploče nisu potrebne, ali su jednostavne za dobivanje informacija o Kubernetes resursima. Kada postoje veliki klasteri gdje se mora voditi računa o resursima, tada je kontrolna ploča iznimno koristan alat. U ovom radu kao kontrolna ploča korist će se *Kubernetes dashboard*.

Slika 48: Postavljanje kontrolne ploče

```
sgal@raspberrypi:~ $ GITHUB_URL=https://github.com/kubernetes/dashboard/releases VERSION_KUBE_DASHBOARD=$(c
url -w '%{url_effective}' -I -L -s -S ${GITHUB_URL}/latest -o /dev/null | sed -e 's|.*/||')
sgal@raspberrypi:~ $ echo $VERSION_KUBE_DASHBOARD
kubernetes-dashboard-7.4.0
sgal@raspberrypi:~ $ sudo kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/ai
o/deploy/recommended.yaml
namespace/kubernetes-dashboard unchanged
serviceaccount/kubernetes-dashboard unchanged
service/kubernetes-dashboard unchanged
secret/kubernetes-dashboard-certs unchanged
secret/kubernetes-dashboard-csrf configured
Warning: resource secrets/kubernetes-dashboard-key-holder is missing the kubectl.kubernetes.io/last-applied
-configuration annotation which is required by kubectl apply. kubectl apply should only be used on resource
s created declaratively by either kubectl create --save-config or kubectl apply. The missing annotation wil
l be patched automatically.
secret/kubernetes-dashboard-key-holder configured
configmap/kubernetes-dashboard-settings unchanged
role.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
deployment.apps/kubernetes-dashboard unchanged
service/dashboard-metrics-scraper unchanged
deployment.apps/dashboard-metrics-scraper unchanged
```

Izvor: Vlastita izrada

Naredbe koje su se izvršile postavile su varijablu verzije za Kubernetes Dashboard, prikazale tu verziju, a zatim primijenile preporučenu *YAML* konfiguraciju za Kubernetes Dashboard na klasteru.

Poželjno je postaviti prava pristupa kontrolnoj ploči iz sigurnosnih razloga, preporučena konfiguracija daje kontrolnoj ploči *ServiceAccount*, ograničen pristup Kubernetes resursima. To može spriječiti otkrivanje osjetljivih podataka klastera kao što su tajne ili certifikati.

Da bi se iskotristile sve funkcionalnosti web korisničkog sučelja bit će potrban račun usluge s administrativnim posvlasticama. To će biti račun usluge s ulogom administratora klastera.

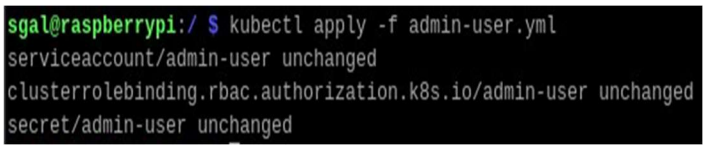
Slika 49: admin-user.yml

```
sgal@raspberrypi:/ $ cat admin-user.yml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
---
apiVersion: v1
kind: Secret
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
  annotations:
    kubernetes.io/service-account.name: "admin-user"
type: kubernetes.io/service-account-token
```

Izvor: Vlastita izrada

Kada je datoteka uređena, novi račun usluge izrađuje se putem naredbe:
***kubectl apply -f admin-user.yml***

Slika 50: Kreiranje administratora




Izvor: Vlastita izrada

Preporučeni način za pristup Kubernetes nadzornoj ploči je putem tokena nositelja. To omogućuje veliku fleksibilnost u upravljanju dopuštenjima. Da bi došli do tokena nositelja administratorskog korisnika ServiceAccount, pokreće se sljedeća naredba:

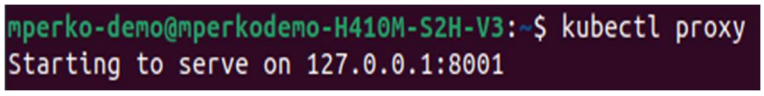
***kubectl get secret -n kubernetes-dashboard $(kubectl get serviceaccount admin-user -n kubernetes-dashboard -o jsonpath="{.secrets[0].name}") -o jsonpath="{.data.token}" | base64 –decode***

Slika 51: Ispis tokena


Izvor: Vlastita izrada

Token će biti potreban za prijavu u Kubernetes kontrolnu ploču. Kako bi se pristupilo ploči potrebna je sljedeća nareba na lokalnom računalu: ***kubectl proxy***

Slika 52: kubectl proxy

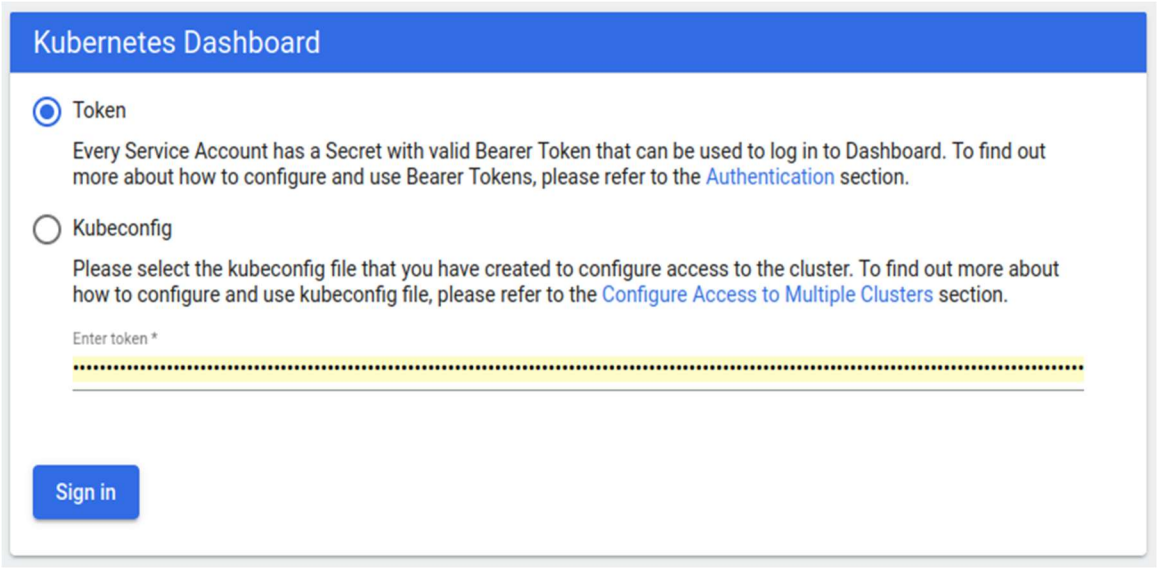



Izvor: Vlastita izrada

Nakon izvršetka naredbe prikazuje se da je *WebUI* dostupan na *127.0.0.1:8001*. Nakon što je *proxy* aktiviran, web sučelju se može pristupiti preko sljedeće adrese:

***http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy***

Slika 53: Unos tokena za prijavu




Izvor: Vlastita izrada

Nakon unosa ispravnog tokena omogućen je pristup Kubernetes nadzornoj ploči.

# Zaključak

Provedena implementacija pametnog parkirališta pokazala je da je moguće izgraditi učinkovit i funkcionalan sustav koristeći pristupačne tehnologije poput Raspberry Pi uređaja i otvorenog softvera. Kombinacija Python koda za obradu slika i prepoznavanje registarskih oznaka, Spring Boot za logiku poslužiteljskog sloja te Vue.js za korisničko sučelje, zajedno s Dockerom i k3s za orkestraciju, omogućila je kreiranje skalabilnog i pouzdanog rješenja. Rezultati pokazuju da sustav može uspješno prepoznati registarske oznake i upravljati rampama u stvarnom vremenu, što potvrđuje njegovu primjenjivost u stvarnim scenarijima. Korištenje ultrazvučnih senzora za mjerenje udaljenosti dodatno povećava preciznost i efikasnost sustava, omogućujući automatsko upravljanje rampama na temelju stvarnog stanja na terenu.

Modularnost pomoću Dockera i Docker Composea te orkestracija s k3s sustavom osigurali su jednostavno postavljanje, upravljanje i skaliranje aplikacije. Ovo rješenje pokazalo je da se kontejnerizacija i orkestracija mogu uspješno primijeniti i na manje, lokalne sustave poput ovog, što otvara mogućnosti za daljnja istraživanja i primjene u sličnim projektima.

Daljnji rad može uključivati dodatna poboljšanja u preciznosti prepoznavanja registarskih oznaka, optimizaciju performansi sustava te integraciju s drugim sustavima za pametno upravljanje gradovima. Također, istraživanje novih senzorskih tehnologija i metoda analize podataka može pridonijeti razvoju još naprednijih rješenja za pametna parkirališta.

Ovaj projekt pokazuje da je uz korištenje modernih tehnologija moguće izgraditi učinkovit sustav koji može značajno unaprijediti upravljanje parkiralištima, pružajući bolje korisničko iskustvo i veću efikasnost.

# Popis literature

https://nordvpn.com/blog/what-is-raspberry-pi/
https://www.raspberrypi.com/documentation/computers/getting-started.html
https://www.raspberrypi.com/documentation/computers/os.html
https://opencv.org/about/
https://guides.platerecognizer.com/docs/snapshot/getting-started/
https://projects.raspberrypi.org/en/projects/physical-computing/12
https://pi4j.com/getting-started/understanding-the-pins/
https://thepihut.com/blogs/raspberry-pi-tutorials/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi
https://www.digikey.com/en/maker/tutorials/2021/how-to-control-servo-motors-with-a-raspberry-pi
https://www.cloudflare.com/learning/security/api/what-is-api-call/
https://aws.amazon.com/docker/
https://docs.docker.com/get-started/overview/
https://circleci.com/blog/docker-image-vs-container/
https://docs.docker.com/compose/
https://traefik.io/glossary/k3s-explained/
https://www.finout.io/blog/k3s-vs-k8s
https://medium.com/@stevenhoang/step-by-step-guide-installing-k3s-on-a-raspberry-pi-4-cluster-8c12243800b9
https://www.papertrail.com/solution/guides/nginx/