Online JVM monitoring sustav

Pavić, Marin

Undergraduate thesis / Završni rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Algebra University College / Visoko učilište Algebra**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:225:048853

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Algebra Univerity College - Repository of Algebra Univerity College





VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

Online JVM monitoring sustav

Marin Pavić

Zagreb, rujan 2018.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada".

U Zagrebu, 23.9.2018.

Marin Paric

Predgovor

Želio bih se zahvaliti roditeljima koji su me bodrili kroz cijeli proces pisanja, te svojemu mentoru Aleksandru Radovanu koji je uvijek odvajao vremena za mene unatoč mojim stalnim zapitkivanjima.

Također bi se htio zahvaliti svojem računalu i Mavenu što nisu stvarali previše neželjenih problema.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Cilj ovog rada je povećati mogućnosti upravljanja Java aplikacijama korištenjem *weba* kao modernijeg sučelja.

Danas u svijetu Java aplikacija postoje mnoga rješenja za praćenje performansi Java virtualne mašine na kojima se aplikacije pokreću. No sva ta rješenja su bazirana na činjenici da osoba ima pristup *hardveru* na kojem je pokrenuta aplikacija, ili da mora instalirati posebnu aplikaciju koja će te podatke pratiti.

Praktičnim dijelom ovog završnog rada je ponuđeno rješenje koje svu funkcionalnost koje nude standardne aplikacije prenosi na *web*. Eliminirajući potrebu za instalacijom posebnog softvera za praćenje performansi Java aplikacija. Korisnikova aplikacija će imati mogućnost davanja uvida u osnovne performanse samog računala: CPU opterećenje, količina iskorištene memorije itd. Te specifične funkcionalnosti koje se odnose na samu Java aplikaciju: koliko je niti pokrenuto, trenutno stanje memorije, broj učitanih klasa i sl.

Ključne riječi: performanse, praćenje performansi, web aplikacija, udaljena kontrola

Summary

The goal of this work is to increase the ways of moderating Java applications using the web as a more modern UX design.

Today in the Java world there are a few software solutions that enable Java virtual machine (JVM) monitoring. But most of these solutions require that either, the user has access to the running hardware or the user has a computer at hand. In order to install proprietary software to monitor the application on a server.

With the practical part of this finals work is offered a solution that shifts the entire functionality of these standard desktop apps to the web, Eliminating the need for third party proprietary desktop applications in order to monitor a Java application. The web app has to offer the ability to monitor the standard performance metrics such as: CPU load, memory usage etc. And also specific functionalities that have to do with the monitored app itself such as: The number of threads running and their state, the current memory map and the ability to execute methods from the web app on the monitored app.

Keywords: performance, monitoring performance, web aplication, remote control

Sadržaj

1. Uvod 1

2. Funkcionalnosti rada 2

3. Korištene tehnologije 3

3.1. Programski jezici 3

3.1.1. Java 3

3.1.2. TypeScript 3

3.2. Spring (Boot) 4

3.3. Angular 6

3.4. *Angular* material design 8

3.5. JMX API 9

3.5.1. Instrumentacijski sloj 9

3.5.2. Sloj agenta 10

3.5.3. Sloj distribucije 10

3.6. WebSockets 11

4. Arhitektura 12

4.1. API biblioteka 13

4.2. Web API 15

4.2.1. Baza Podataka 15

4.2.2. Poslovni sloj 18

4.2.3. JMX Manager Sloj 21

4.3. Web aplikacija 25

4.3.1. *Container* komponente 26

4.3.2. *Screen* komponente 27
4.3.3. Servisi 30
4.3.4. Ruter 31
4.3.5. JMX komunikacijski dio prezentacijskog sloja 31
5. Korisničko iskustvo 35
5.1. Administracija 35
5.2. Stvaranje udaljenog računala 36
5.3. JMX Podatci o performansama 37
Zaključak 38
Popis kratica 39
Popis slika 40
Popis tablica 41
Popis kôdova 42
Literatura 43
Prilog 44

1. Uvod

Glavna ideja za ovaj projekt je dobivena pri izradi drugog projekta za kolegij Java web programiranje. Naime, tijekom izrade tog projekta došlo je do potrebe testiranja performansi aplikacije i uočeno je da svako ponuđeno rješenje zahtijeva od korisnika da, ili ima pristup hardveru (u ovom slučaju serveru, koji je u Francuskoj), ili da na svojem računalu instalira posebne aplikacije koje će služiti toj svrsi. Posebna aplikacija mora konstantno raditi kako bi se podatci o performansama slali na računalo, ako osoba nije trenutno blizu računala, ne postoji mogućnost pristupa tim podatcima.

Radi ovih problema zamišljena je aplikacija koja će koristiti već pokrenutu korisnikovu Java aplikaciju na udaljenom računalu i korištenjem tehnologije Java Management Extensions (JMX) posluživati podatke o svojim performansama i podatke o performansama računala na kojem se odvija, te sve to prikazati u web sučelju, eliminirajući potrebu za korištenjem posebnih *desktop* aplikacija za tu svrhu.

Kroz ovaj rad je detaljno opisana arhitektura aplikacije nadgledanja udaljenim računalima na kojima se odvijaju Java aplikacije. Čitatelja se isprva upoznaje s korištenim tehnologijama, te nakon toga se prelazi na objašnjenje arhitekture sustava. Objašnjenje arhitekture započinje kompletom alata za pokretanje i modificiranje JMX biblioteke i izlaganjem podataka za performanse, gdje se nakon toga pogled prebacuje na Web aplikaciju koja se sastoji od Baze podataka, servisnog sloja, te same web aplikacije.

Osim opisa tehnologije bit će opisan i uzorak dizajna, te opis zašto je korišten određeni dizajn.

2. Funkcionalnosti rada

Finalni produkt ovog rada je samostalno funkcionalni API sustav, koji dopušta povezivanje udaljene aplikacije s API sustavom, te omogućava pristup podatcima o performansama te aplikacije. Korisnik ima mogućnosti korištenja krajnjih točaka API-a samostalno, ili putem *web* sučelja. Tijekom korištenja aplikacije korisnik može vidjeti podatke o udaljenom računalu (engl. *Managed Machine*) i o pokrenutoj udaljenoj aplikaciji na tom računalu, te koristeći te podatke saznati kako se njegova aplikacija ponaša na udaljenom računalu.

Sve počinje tako da korisnik zatraži izradu novog udaljenog računala, pri čemu dobije *randomizirani JmxKey* koji će na dalje predstavljati njegovu aplikaciju na udaljenom računalu. Korisnik nakon toga skida *JmxOnlineSDK* paket i dodaje ga u svoju aplikaciju koja će biti pokrenuta na udaljenom računalu, te unutar koda aplikacije, konfigurira servis pri čemu „lijepi“ *JmxKey* koji je prethodno dobio. Kada se korisnikova aplikacija na udaljenom računalu pokreće, prije definirani servis pokreće *JmxServer* na korisnikovoj aplikaciji, te generirani *serviceURL* i *JmxKey* šalje *JmxOnline* API sustavu. API sustav, koristeći *JmxKey* pronalazi registriranu aplikaciju u bazi podataka, te joj dodaje *serviceURL* kako bi korisnik mogao pristupiti podatcima. Korisnik nakon toga može pristupiti udaljenoj aplikaciji korištenjem krajnjih točaka API-a ili korištenjem web sučelja.

Prilikom korištenja sustava korisnik ima mogućnost birati koji vrstu podataka o performansama želi vidjeti, nakon čega sustav od udaljene aplikacije traži samo one podatke koje je korisnik odabrao. Također, prilikom korištenja *Web* sučelja korisnik ima mogućnost spremiti koje vrste podataka želi vidjeti za koji aplikaciju, tako da pri sljedećem prijavljivanju ti podatci su automatski učitani.

Svaki sloj *JmxOnline* sustava je izrađen tako da može funkcionirati samostalno ako korisnik ne želi koristiti sustav u cjelini, što cijelom projektu daje jako veliku fleksibilnost.

3. Korištene tehnologije

Zbog kompleksnosti rada korišteno je mnoštvo različitih tehnologija. U ovom odlomku su objašnjene sve tehnologije korištene u radu, činjenice o njima, te zašto su baš one izabrane.

3.1. Programski jezici

3.1.1. Java

Java programski jezik je objektno orijentirani jezik čiji su originalni autori: James Gosling, Mike Sheridan i Patrick Naughton. Prva verzija Java programskog jezika započela je 1991. gdje je njezino inicijalno ime bilo „*Oak*“, nazvana po stablu hrasta ispred Goslingovog ureda. Najnovija verzija Jave za vrijeme pisanja ovog rada je Java 10 koja je izašla 20.3.2018.

Java je dizajnirana s ciljem da ima što manje ovisnosti o operacijskom sustavu. Java slijedi princip WORA (engl. *Write Once, Run Anywhere*), što znači da Java kod napisan na npr. Windows platformi može se bez problema pokrenuti na Linux platformi bez potrebe mijenjanja koda ili ponovnog sastavljanja koda. Java kod se sastavlja u *Bytecode* koji se odvija u JVM-u (engl. *Java Virtual Machine*), te sam kod ima sintaksu koja naliči C/C++ stilu.[1]

Cijeli pozadinski dio aplikacije (engl. *Backend*), te SDK, je napisan u Java programskom jeziku, a sam jezik je izabran baš iz razloga WORA principa kako sam sustav ne bi ovisio o operacijskom sustavu na kojem se pokreće.

3.1.2. TypeScript

TypeScript je programski jezik otvorenog koda (engl. *Open Source*), stvoren i održavan od strane Microsofta. Objavljen je po prvi put u javnost 2012. godine s verzijom 0.8. nakon dvije godine internog razvoja od strane Microsofta. Najnovija verzija u vrijeme pisanja ovog rada je 3.0. koja je objavljena 30.7.2018.

[1] https://en.wikipedia.org/wiki/Java_(programming_language)

TypeScript je nad skup JavaScripta, te na sve funkcionalnosti JavaScripta dodaje i opcionalno statično provjeravanje tipova, te standardne objektno orijentirane funkcionalnosti kao što su klase, sučelja itd., s tim da i dalje ostaje kompatibilan s JavaScript kodom. TypeScript se zapravo prepisuje (engl. *Transcompile*) u JavaScript tijekom izgradnje koda (engl. *Build*).[2]

TypeScript je korišten pri izradi prezentacijskog dijela *JmxOnline* sustava najviše radi sigurnosti tipova prilikom pisanja, ali i same fleksibilnosti radi toga što je podržan od strane *Angular* radnog okvira, pa je sam razvoj bio jednostavniji.

3.2. Spring (Boot)

Spring je radni okvir čiji je originalni autor Rod Johnson, te je prva verzija radnog okvira objavljena 2002.

Spring je izrađen kako bi se smanjila kompleksnost *web* Java aplikacija. To postiže tako što je cijeli *Spring* radni okvir podijeljen u različite module, od kojih svaki daje određene usluge za različite tehnologije. Tablica 3.1. popisuje neke od popularnijih modula[3].

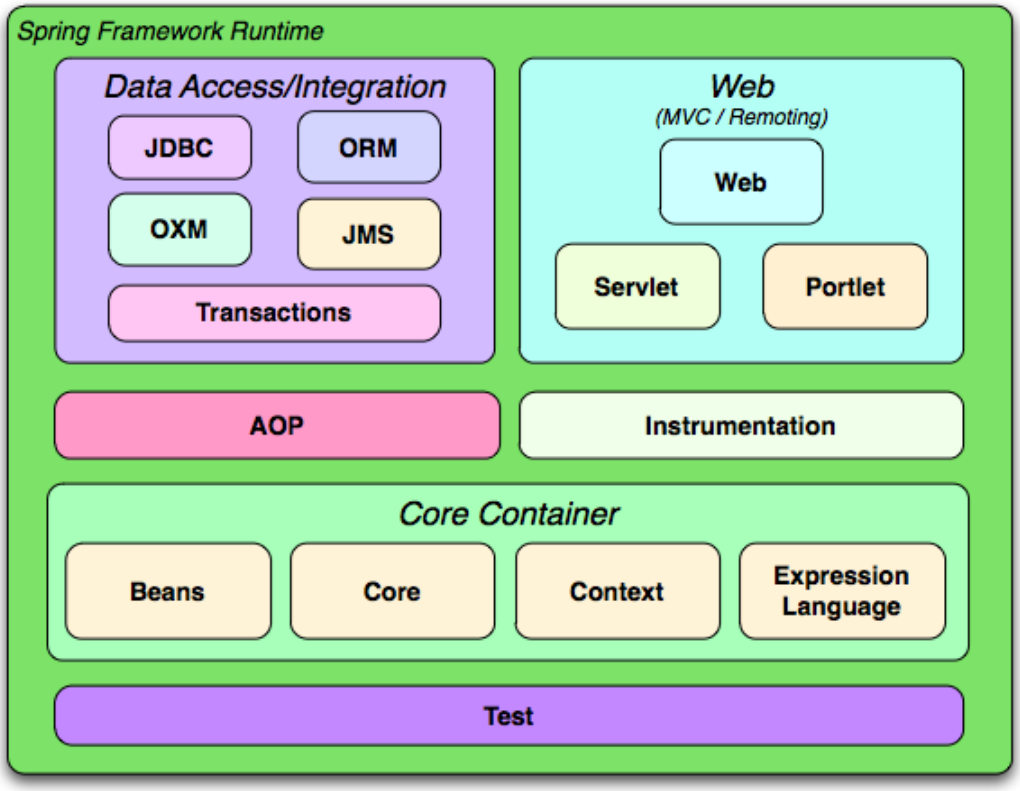
Tablica 3.1. Moduli *Spring* radnog okvira

Spring Core Container	Osnovni modul *Spring* radnog okvira koji daje usluge kontejnera.
Aspect Oriented Programming	Daje usluge smanjivanja ovisnosti unutar aplikacije.
Autentikacija i autorizacija	Daje osnovnu implementaciju autentikacije i autorizacije korisnika u aplikaciji.
Data access	Daje usluge pristupanja bazi podataka preko raznih biblioteka kao što je JDBC ili razni ORM alati.
Model-view-controller	Daje osnovnu okvirnu podržku za MVC uzorak dizajna.

[2] https://en.wikipedia.org/wiki/TypeScript
[3] https://en.wikipedia.org/wiki/Spring_Framework

Uz to, *Spring* radni okvir postiže jednostavnost tako da se konfiguracija sustava radi pomoću POJO-a (engl. *Plain Old Java Object*). Također, *Spring* moduli su labavo povezani (engl. *loosely coupled*) što daje mogućnosti odvajanja funkcionalnosti u svoje klase bez da se mora brinuti o ovisnosti između njih. To je postignuto korištenjem kontejnera za sve aplikacijske objekte unutar aplikacije. *Spring* kontejner se brine o svim objektima od trenutka kada su kreirani, do trenutka kada su izbrisani. Radi ovakvog dizajna dobija se mogućnost referenciranja bilo kojeg *Beana* unutar drugog *Beana*, bez potrebe konstruktora ili *setter* i *gettera*. Sve te informacije se čuvaju unutar konteksta aplikacije (engl. *Application context*), te pri konfiguraciji *Springa* postoji mogućnost biranja između konfiguracije u XML formatu ili konfiguracije preko Java anotacija. U ovom radu su korištene Java anotacije jer su puno „čišće“ za čitanje i pisanje.

Spring Boot je verzija *Springa* koja ne zahtijeva ručnu konfiguraciju svih stavki, već svaka funkcionalnost dolazi u „Starter paketima“ koji su već konfigurirani za određene funkcionalnosti. Svaka funkcionalnost ima mogućnost mijenjanja konfiguracije ako je to potrebno. Slika 3.1. prikazuje neke od najčešće korištenih modula unutar jednog sustava u *Spring* radnom okviru.



Slika 3.1. Najčešće korišteni moduli *Spring* radnog okvira

https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch01s02.html

3.3. Angular

Angular je platforma otvorenog koda za izradu dinamičkih web stranica, a razvijena od strane Google tima. *Angular* je platforma za izradu web stranica korištenjem *HTML*-a i TypeScripta. *Angular* platforma je prvobitno bila zvana *AngularJS,* te je originalno objavljena 2010. Nakon verzije 2.0. platforma gubi sufiks „JS“, cijela platforma izrađena ispočetka, te preimenovana u „*Angular*“, iako interno još uvijek dodaju sufiks verzije, npr. *Angular* 5. Najnovija verzija *Angular* platforme u vrijeme pisanja ovog rada je 6.0., te objavljena je 3.5.2018.

Cijela arhitektura platforme je podijeljena u module od kojih svaki daje određene funkcionalnosti. Također *Angular* podržava i standardne *JavaScript* biblioteke koji se dodaju na isti način kao u sami *Angular* moduli. O paketima brigu vode upravitelji paketa (engl. *Package Manager*) kao što su NPM (engl. *Node Package manager*) ili Yarn. [4]

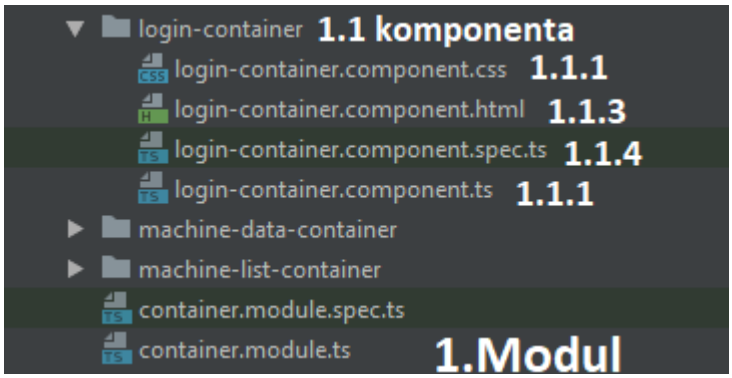
Angular je dizajniran za izradu web aplikacija s „Jednom stranicom“, Što znači da od trenutka kada korisnik posjeti web stranicu do trenutka kada ode s nje, korisnik tehnički gleda samo jednu *HTML* stranicu koja se većinom zove „index.html“. *Angular* ovu funkcionalnost postiže tako da inicijalno pokaže „index.html“, te po potrebi „ubrizga“ *HTML* elemente zapakirane u komponente unutar „index.html“ stranice. S korisnikove strane web aplikacija se ponaša kao i svaka druga, URL se mijenja, različiti elementi se pojavljuju, ali u stvarnosti korisnik je i dalje na „index.html“ stranici. Ovakav pristup web stranicama daje odlične performanse, te sam razvoj ovakvih web stranica je puno ugodniji za same programere jer je razvoj sličan kao razvijanje desktop aplikacija.

Angular koristi sljedeću hijerarhiju podatka:

1. Modul
    1.1. Komponenta
        1.1.1. *TypeScript* datoteka
        1.1.2. *CSS* datoteka
        1.1.3. *HTML* datoteka
        1.1.4. Opcionalna *TypeScript* test datoteka

Slika 3.2. pokazuje dijelove jednog modula unutar *Angular* projekta.

[4] https://en.wikipedia.org/wiki/Angular_(application_platform)



Slika 3.2. Dijelovi *Angular* modula

Angular aplikacija može imati više modula, te svaki modul može deklarirati više komponenti. Modul može odlučiti koje će komponente izložiti van, a koje će koristiti samo interno gdje onda samo komponente koje su u istom modulu mogu koristiti tu komponentu. Modul također može referencirati druge module gdje se onda sve komponente koje referencirani moduli izlaže van mogu koristiti unutar tog modula.

Komponenta se sastoji od četiri datoteke koje skupa opisuju funkcionalnosti komponente: *TypeScript* datoteka, CSS datoteka, *HTML* datoteka te opcionalna *TypeScript* test datoteka.

TypeScript datoteka služi za obavljanje pozadinske logike i događaja tijekom prikazivanja i interakcije komponente s korisnikom: klikovi gumba, povlačenje klizača, unos teksta i sl.

CSS datoteka služi za stiliziranje te komponente, s tim da je važno napomenuti da *Angular* platforma sadrži i globalni „styles.css“ koji je dostupan svim komponentama.

HTML datoteka služi za prikaz samih elemenata na ekranu, s tim da je važno napomenuti da komponenta može referencirati drugu komponentu u kojem slučaju se onda *HTML* datoteka te komponente ubrizgava (engl. *injects*) unutar na mjesto referenciranja, ako joj može pristupiti.

Kod 3.1. opisuje jedan primjer *HTML* datoteke *Angular* komponente.

```
<div fxLayout="column" fxFlexAlign="center center">
  <mat-card>
    <mat-card-title>Machine list</mat-card-title>
    <mat-card-content>
      <mat-list >
        <div *ngFor="let machine of machineList">
          <mat-list-item >
            <app-machine-list-item
                [managedMachine]="machine">
          </app-machine-list-item>
          </mat-list-item>
          <mat-divider></mat-divider>
```

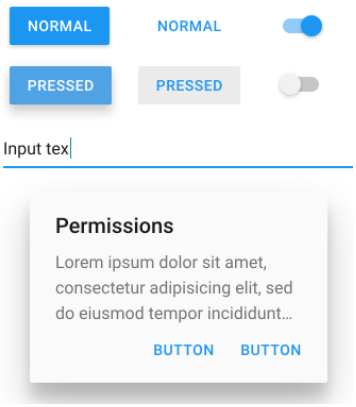
```
        </div>
      </mat-list>
    </mat-card-content>
  </mat-card>
</div>
```

Kôd 3.1. Primjer *HTML* datoteke *Angular* komponente

Opcionalna *TypeScript* datoteka za testiranje služi za automatsko testiranje korištenjem *angular-cli* alata komandne linije.

3.4. *Angular* material design

Material Design je jezik za opisivanje vizualnog prikaza web stranica. Skup pravila koji web stranicama daje izgled sličan izgledu na Android mobitelima, te se aplikacija automatski prilagođava veličini ekrana. *Material Design* je osmislila tvrtka Google, te je inicijalno objavljena za Android mobilne uređaje. No kasnije je objavljena i za web o obliku CSS tema ili u slučaju ovog rada dodatna knjižnica komponenti koji imaju primijenjen *Material Design*. Slika 3.3. prikazuje ekran izrađen po *Material Design* principima.



Slika 3.3. Ekran izrađen po *Material Design* principima

https://en.wikipedia.org/wiki/Material_Design

3.5. JMX API

Java Management Extensions (*JMX*) je tehnologija programskog jezika Java koji daje pristup objektima/resursima unutar aplikacije preko mreže. JMX arhitektura se sastoji od tri sloja:

- Instrumentacijski sloj
- Sloj agenata
- Sloj distribucije

3.5.1. Instrumentacijski sloj

Instrumentacijski sloj je najbliži podatcima i sastoji se od *MBean* objekata registriranih na agenta. *MBean* je obični *JavaBean* kojemu je dodano ubrizgavanje zavisnosti (engl. *Dependency Injection*). *MBean* dopušta upravljanje s resursom kroz JMX agenta, a sastoji se od dijela funkcionalnosti resursa kojeg upravlja. Sam resurs ne mora cijelu svoju funkcionalnost izlagati vani, niti mora „pričati“ Java programski jezik. *MBean* svu funkcionalnost prevodi u Java jezik bez obzira na resurs. Postoje tri vrste *MBean*ova:

Standardni *MBean* je namijenjen na situacije gdje postoji novi resurs ili resurs s dobro poznatim i statičkim sučeljem. Ova vrsta *MBeana* sama kreira sučelje za pristup resursu koristeći sam resurs kao okvir.

Dinamični *MBean* koristi meta-podatke klase kako bi opisao sučelje za upravljanje. S ovakvim pristupom programer odlučuje koliko i što će izlagati vani, ali zahtijeva više tipkanja.

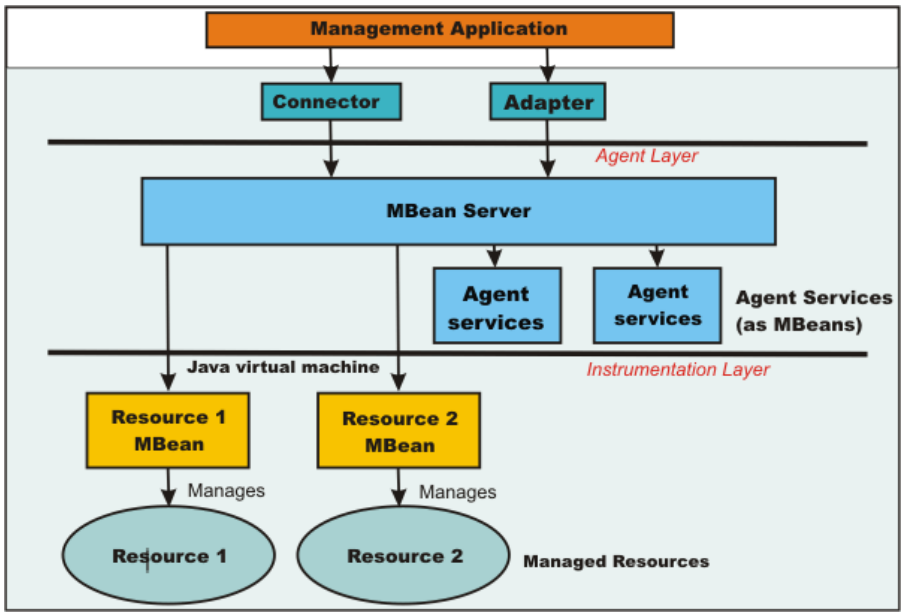
Model MBean je posebna vrsta dinamičnog *MBeana* gdje je kreacija meta-podataka omogućena tijekom rada aplikacije (engl. *Runtime*). To je postignuto tako da je sam *ModelMBean* već dio JMX Agenta. *ModelMBean* u sebe prima *ModelMBeanInfoSupport* koji daje dodatne informacije Agentu kako napraviti *MBean* resurs prema klasi koju predstavlja, to je postignuto korištenjem Java Reflection API-a koji omogućuje čitanje i manipuliranje objekata tijekom rada aplikacije.

3.5.2. Sloj agenta

MBeanServer je glavna komponenta sloja agenta, te po retrospektivi cijelog JMX API-a. *MBeanServer* je Java objekt koji služi kao registar svim *MBeanovima*, ali nikad ne izlaže referencu na sam objekt, već samo sučelje za upravljanje s tim objektom. Također *MBeanServer* tretira sve *MBeanove* jednako, bez obzira koji resurs oni predstavljaju.

3.5.3. Sloj distribucije

Sloj distribucije je rubni sloj JMX API-a. Ovaj sloj se brine o izlaganju agenata prema vanjskoj mreži. To postiže koristeći dvije vrste objekata: adapteri (engl. *Adapters*) i konektori (engl. *connectors*). Adapteri izlažu *MBean*ove preko različitih protokola kao što su HTTP i SMTP, a adapteri izlažu *MBeanove* preko distribucijskih tehnologija kao što su Java RMI (engl. *Remote Method Invocation*). U ovom radu je korištena druga opcija s adapterima, s tim da je važno napomenuti da je cijeli JMX API modularan, te da je moguće imati više Adaptera i/ili više Konektora unutar jedne aplikacije. Slika 3.4. opisuje slojeve JMX API sustava.



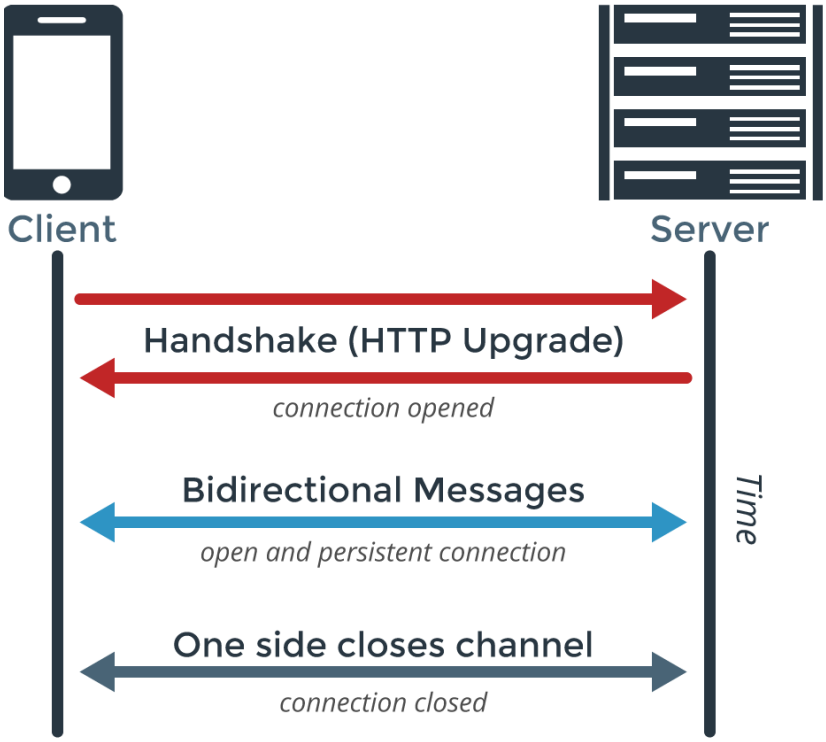
Slika 3.4. Slojevi JMX API sustava

https://www.ibm.com/support/knowledgecenter/en/SSEQTP_9.0.0/com.ibm.websphere.base.doc/ae/cxml_javamanagementx.html

3.6. WebSockets

WebSocket je komunikacijski protokol autora Ian Hickson i Michael Carter. Prva verzija protokola je objavljena krajem 2009., te Google Chrome 4 je prvi preglednik koji ga je podržavao. Protokol daje mogućnost komunikacije u oba smjera, daje mogućnost konstantne komunikacije, te je generalno dosta stabilniji za konstantnu komunikaciju naspram HTTP komunikacije gdje se pri svakom slanju i primanju događa rukovanje klijenta i servera.

Kako bi se uspostavila *WebSocket* konekcija, klijent pošalje zahtjev rukovanja. Klijent šalje obični HTTP zahtjev, ali koristeći HTTP-ov *upgrade header* zahtijeva promjenu protokola na *WebSocket*. Server odgovara s HTTP porukom „101 mijenjanje protokola.“, nakon čega se otvara novi *WebSocket*, te server i klijent mogu slati podatke ili tekst okvire u oba smjera. [5]Slika 3.5. objašnjava ovaj proces.



Slika 3.5. Komunikacija u oba smjera pomoću *websocketa*

https://www.pubnub.com/wp-content/uploads/2014/09/WebSockets-Diagram.png

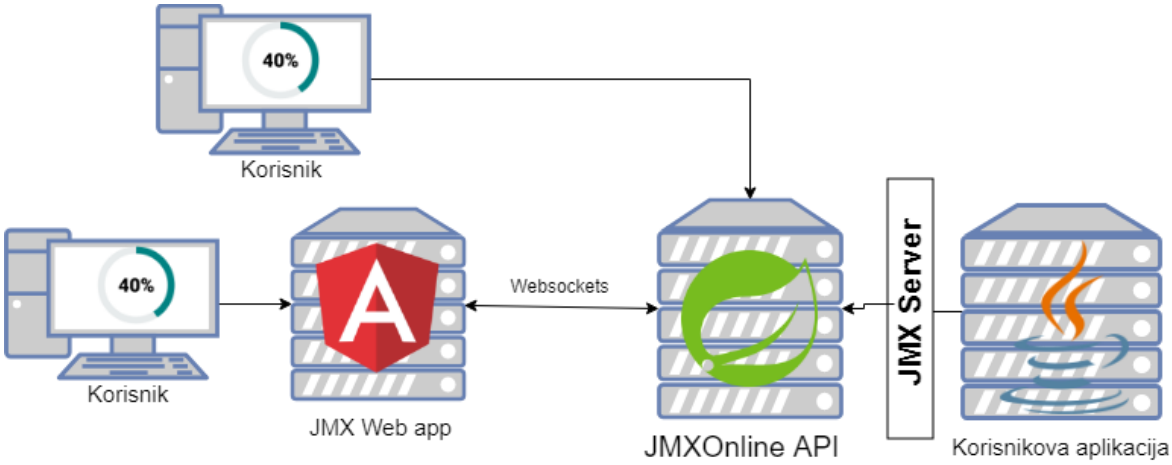
[5] https://en.wikipedia.org/wiki/WebSocket

4. Arhitektura

Praktični dio ovog rada je podijeljen u 3 dijela gdje svaki dio ima cjeline:

- API biblioteka
- Web API
    - Baza podataka
    - Poslovni sloj (poslovna logika)
    - API Endpoint
- Web Aplikacija
    - Sloj za komunikaciju s API-em
    - Prezentacijski sloj

Ovakva podjela daje sustavu prenosivost i neovisnost o daljini, također je sam razvoj ovakvog sustava puno jednostavniji. API je strukturiran tako da upravljana aplikacija ne mora brinuti o protokolu prijenosa podatka prema web serveru. Ta je komunikacija određena u pozadini i u slučaju promjene načina komunikacije funkcionalnosti se ne mijenjaju. Slika 4.1. ilustrira okvirnu arhitekturu sustava.



Slika 4.1. Arhitektura sustava

Također web aplikacija slijedi uzorak dizajna MVC koji joj daje fleksibilnost pri održavanju i unaprjeđenju. Svaki sloj se brine o svojim podatcima i svaki sloj se može individualno mijenjati, bez velikih utjecaja na ostatak sustava.

Korištenjem web aplikacije kao klijentskog sučelja dobije se neovisnost o uređaju na kojem se pristupa podatcima jer aplikacija koristi *materijal design* kako bi se sučelje prilagodilo uređaju s kojeg se pristupa stranici.

4.1. API biblioteka

API biblioteka je srce sustava, te pruža sve podatke za prikaz i upravljanje performansama udaljene aplikacije. Biblioteka koristi Java Management Extensions API kako bi izložila podatke o računalu na kojem se odvija.

Cijela komunikacija se odvija preko jednog URI-a (engl. *Uniform Resource Identifier*) koji se generira pri pokretanju upravljane aplikacije, te API ključa koji služi za identificiranje više udaljenih aplikacija na jednom računalu. Način komunikacije između JMX API-a i web servera odvija se preko Java RMI API-a (engl. *Remote Method Invocation*). Taj način komunikacije odabran jer nema preveliku ovisnost o okruženju u kojem se nalazi tako da je idealan za ovakav sustav. API je zamišljen tako da bez obzira koja se vrsta Java aplikacije odvija na serveru, API će u najmanju ruku moći iznijeti krajnje točke za pristup osnovnim podatcima za praćenje performansi aplikacije i servera na kojem se odvija, i to sve bez potrebe ikakve promjene izvornog koda udaljene aplikacije. Cijeli JMX sustav se pokreće sa samo 4 linije koda (kod 4.1.):

```
JOnlineService service = JOnlineFactory.getInstance();
service.setUrl("localhost",1234);
service.setApiKey("dfgdfsg-342fds-242df-2342ds");
service.start();
```

Kôd 4.1. *Snippet* za pokretanje JMX servisa na udaljenoj aplikaciji

`JOnlineService` je glavno sučelje preko kojeg se modificira, zaustavlja i pokreće *JmxOnline* sustav. Korisnik sustava ima potpunu kontrolu što njegova aplikacija izlaže vani i što trenutno radi ili ne radi.

Metoda `setUrl(String url,int port)` uzima parametar *hostnamea* i *porta*, te s njima konstruira JMX URI preko kojeg će web aplikacija moći pristupiti podatcima u kodu Kod 4.1 bi generirao URI koji bi izgledao ovako:

`service:jmx:rmi://localhost/jndi/rmi://localhost:1234/jmxrmi`

Metoda `setApiKey(String key)` ima ulogu prijavljivanja aplikacije serveru prilikom pokretanja udaljene aplikacije. Ideja je da će korisnik *JmxOnlinea* prije nego što pokrene

svoju aplikaciju, otići na web stranicu *JmxOnlinea* i registrirati aplikaciju, pri čemu će dobiti jedinstveni JmxKey koji od trenutka kreiranja pa do promjene predstavlja upravljanu aplikaciju kao njegovu i nitko drugi ne može vidjeti performanse njegove aplikacije. Korisnik bi nakon toga kopirao `ApiKey` i „zalijepio“ ga u metodu `setApiKey`. `JOnlineService` koristi taj API ključ pri prvom pokretanju aplikacije kako bi se pravilno registrirao na server.

Nakon što su svi parametri namješteni, metoda `start()` pokreće postupak pokretanja JMX servisa i registriranja istog na u JMX Online API što se odvija u nekoliko koraka (kod 4.2.):

```
// Korak 1.
LocateRegistry.createRegistry(port);
MBeanServer mbs = ManagementFactory
                .getPlatformMBeanServer();

String jmxUrl = createUrl();

// Korak 2.
if(registerMachine(jmxUrl)){

     // Korak 3.
     JMXServiceURL url = new JMXServiceURL(jmxUrl);
     JMXConnectorServer svr = JMXConnectorServerFactory
                    .newJMXConnectorServer(url, null, mbs);

            svr.start();
}else{
throw new RegistrationException(
     "Invalid Jmx registration");
     }
```

Kôd 4.2. Koraci pri pokretanju JMX servisa na udaljenoj aplikaciji

Korak 1. stvara novi `MBeanServer`, te generira `jmxUrl` na kojem će izložiti sučelje servera. *MBeanServer* je glavno sučelje preko kojeg JMX API izlaže klase prema vanjskoj mreži.

Korak 2. Registrira trenutno računalo JMX Online API-u koristeći svoj JmxKey, te API-u predaje `url` generiran u Koraku 1. Ovo se obavlja koristeći obični HTTP POST zahtjev.

Korak 3. Nakon što se SDK uspješno registrirao na JMX Online API, SDK na udaljenoj aplikaciji pokreće `JMXConnectorServer` koristeći `jmxUrl` generiran u Koraku 1.

Kod 4.3. opisuje proces registracije udaljene aplikacije na JmxOnline API.

```
private boolean registerMachine(String jmxUrl) throws IOException
{
  ClientRegistration registration = new ClientRegistration();
        registration.setJmxKey(apiKey);
        registration.setJmxUrl(jmxUrl);
        Gson = new Gson();
        String apiUrl =
 "http://localhost:8080/jmxOnlineApi/managedMachines/register";

        CloseableHttpClient client = HttpClients.createDefault();
        HttpPost httopost = new HttpPost(apiUrl);

 StringEntity entity = new
                StringEntity(gson.toJson(registration));

        httopost.setEntity(entity);
        httopost.setHeader("Accept","application/json");
        httopost.setHeader("Content-type","application/json");

       CloseableHttpResponse response = client.execute(httopost);
        int responseCode =
                  response.getStatusLine().getStatusCode();
        if(responseCode == 200){
            client.close();
            return true;

        }else{
            client.close();
            return false;
        }
    }
```

Kôd 4.3. Proces registracije udaljene aplikacije na JmxOnline API

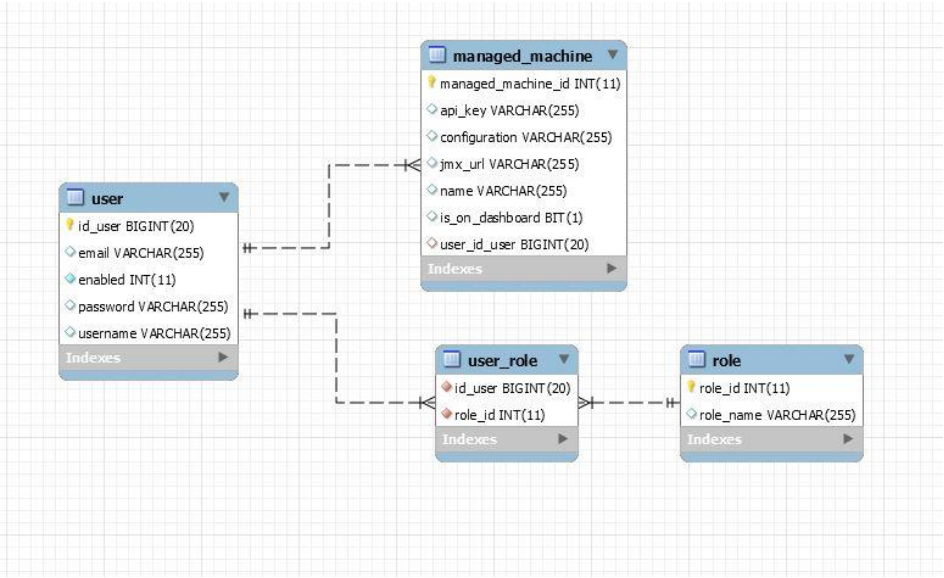
4.2. Web API

Kao što je cijeli praktični rad podijeljen na tri dijela radi lakšeg održavanja, web API je također odvojen u logičke dijelove. Arhitektura sustava započinje s bazom podataka, te preko poslovne logike i krajnjih točaka, pruža podatke na prezentacijski sloj. Također se arhitektura sastoji i od *JmxManager* sloja koji je ukomponiran kao poslovna logika, te služi za komunikaciju s upravljanom aplikacijom.

4.2.1. Baza Podataka

Za bazu podatka odabran je My SQL *server* jer ima najbolju podršku sa Spring radnim okvirom, na kojem je web aplikacija bazirana. Za pristup bazi je korišten JPA API u kombinaciji s *Hibernate* radnim okvirom koji je jedan od poznatih ORM-ova (engl. *Object Relational Mapping*). *Hibernate* daje mogućnost manipulacije tablica na lakši i sigurniji

način mapiranjem tablica i procedura u objekte nad kojima se mogu izvoditi promjene koje reflektiraju promjenama u bazi, što sprječava stupanj ljudske greške i ubrzava vrijeme izrade baze. Na slici 4.2. se može vidjeti dijagram baze podataka:



Slika 4.2. Dijagram baze podataka

Baza podataka nije suvišno kompliciranа jer zapravo *JmxOnline* sustavu nije potrebno puno podataka o samom korisniku ili računalu kojeg upravlja. Jedino što treba znati je *JmxKey* i *JmxUrl*.

Također je u bazu ubačena i autorizacija preko tablice „role“ i „user_role“ kako bi korisnik u sustavu mogao imati različite uloge. To znači da je moguće imati *admin* korisnika koji ima pristup svim upravljanim aplikacijama ili da je u budućnosti moguće naplaćivati uslugu i imati npr. „*Premium*“ korisnika koji će imati pristup većem broju funkcionalnosti nego obični korisnik.

Baza nije izrađena ručno (koristeći SQL jezik), već je generirana iz modela koristeći *Hibernate* radni okvir. Ovakav način izrade baze daje sigurnost da model koji je definiran u kodu definitivno predstavlja podatke u bazi. Primjer jednog modela tablice u kodu se može vidjeti u kodu 4.4.

```
@Entity
@Table(name = "ManagedMachine")
@Getter
@Setter
public class ManagedMachine {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="managed_machine_id")
    private int id;
    @Column(name="apiKey")
    private String apiKey;
    @Column(name = "jmxUrl")
    private String jmxUrl;
    @Column(name = "name")
    private String name;
    @ManyToOne(cascade = CascadeType.ALL,targetEntity =
AppUser.class)
    private AppUser user;
    @Column(name = "configuration")
    private String configuration;
    @Column(name="is_on_dashboard")
    private boolean onDashboard;
}
```

Kôd 4.4. Primjer modela tablice `managed_machine`

U primjeru iznad tablica u pitanju je `managed_machine` iz slike 4.2. i koristeći ovu klasu *Hibernate* radni okvir može napraviti reprezentaciju klase u bazi koristeći anotacije (npr. `@Column`) kao dodatne informacije.

Također se može vidjeti na predzadnjoj liniji da je moguće izraziti i odnose između tablica. U ovom slučaju izraženo je da „*ManagedMachine*“ može pripadati jednom korisniku, a da jedan korisnik može imati više „*ManagedMachine*“ objekata. Ovo daje mogućnost ako je u budućnosti potreba izbrisati korisnika da je moguće kaskadno izbrisati sva upravljana računala koje ta osoba ima.

Komunikacija poslovnog sloja i baze podataka riješena je preko *JPARepository* API-a. *JPARepository* je sustav koji može sam generirati CRUD (*CREATE, READ, UPDATE, DELETE*) operacije nad bazom i čvrsto je integriran s *Hibernate* radnim okvirom, na kojem

je bazirana baza. Također daje mogućnost generiranja SQL procedura bez potrebe pisanja SQL koda jer koristi ime metode u Javi kao predložak za generiranje SQL upita u pozadini. Npr. metoda:

```
public ManagedMachine findByApiKey(String apiKey);
```

bi bila ekvivalentna SQL kodu:

```
CREATE PROCEDURE findByApiKey
(IN apiKey CHAR(30))
BEGIN
      SELECT * FROM managed_machine
      WHERE api_key=apiKey
END
DELIMITER ;
```

Radi ovakvog uzorka web aplikacija je vrlo lagana za održavanje jer ako npr. postoji potreba promjene baze iz My SQL-a na, recimo, Microsoft SQL Server, jedino što bi se trebalo promijeniti je URL u *properties* datoteci i možda par SQL upita, ako postoje, ostatak sustava ostaje ne promijenjen.

4.2.2. Poslovni sloj

Pod poslovni sloj spadaju kontroleri (engl. *controller*) za otkrivanje krajnjih točaka, te servisi za komunikaciju s repozitorijima. Korisnik komunicira s ovim kontrolerima preko HTTP poziva kako bi pokretao akcije na stranici kao npr.: navigacija i prijava u sustav itd.

Prijava u sustav je riješena koristeći *JSON Web Token* (JWT). JWT je standard baziran na JSON formatu koji služi za kreiranje jedinstvenih *tokena* koji u sebi sadrže razne tvrdnje. Glavna ideja je da korisnik svoje korisničino ime i lozinku upisuje samo jednom, te nakon što je autenticiran i autoriziran, dobije jedinstveni *token* koji koristi u daljnjoj komunikaciji s API-em. Zaštita *tokena* je riješena tako da server potpiše *token* prilikom njegovog kreiranja sa svojim privatnim ključem. U daljnjoj komunikaciji pri svakom pozivu server ima mogućnost provjeriti da li je *token* valjan, te pročitati iz *tokena* o kojem je korisniku riječ. JMX Online API unutar *tokena* sprema korisničko ime korisnika kako bi mogao dobiti daljnje informacije o korisniku, kao što su uloga ili osobne informacije, nakon što je dokazano da je *token* valjan. Kao password *encoder* je korišten *BCryptPasswordEncoder*. To je Springova implementacija *BCrypt hashing* algoritma kako bi se spriječilo spremanje passworda u obliku čistog teksta u bazu.

O JWT-u se brinu tri klase koje se integriraju sa *Springovom* konfiguracijom sigurnosti:

- *JWTLoginFilter*
- *JWTAuthenticationFilter*
- *JWTTokenUtil*

`JWTLoginFilter` je unutar *Springove* konfiguracije vezan za `/login` putanju i ovaj se filter okida svaki put kada netko pokuša poslati POST zahtjev na `~/jmxOnlineApi/login`. Ovaj filter dobije korisničko ime i lozinku, te pomoću *Springovog* `AuthenticationManager` objekta provjerava ih naspram baze. U slučaju da su korisničko ime i lozinka točni, stvara novi JWT, te ga sprema u *header* objekt `HttpResponse` objekta.

`JWTAuthentication` filter je unutar *Springove* konfiguracije vezan za svaki zahtjev prema serveru koji mora biti zaštićen. Njegova zadaća je da provjerava da li postoji JWT unutar *request* objekta, te ako postoji, da li je JWT ispravan, da li je istekao i sl. U slučaju da nešto nije uredu s JWT-om *request* se odbacuje sa status porukom 401 *Unauthorized*. Ako je sve uredu `JWTAuthenticationFilter` dodaje u *Springov security context* novu autentikaciju koju je dobio od metode `getAuthentication()` `JWTTokenUtil` klase.

`JWTTokenUtil` je klasa koja se brine o radu s JWT-om. Koriste ga `JWTLoginFilter`, `JWTAuthenticationFilter` za generaciju JWT-a. `JWTTokenUtil` koristi `jjwt` knjižnicu klasa kako bi stvorio JWT u provjerio da li je valjan. Kod 4.5. opisuje `JwtTokenUtil` klasu.

```
public class JwtTokenUtil implements Serializable {
     private static final String SECRET ="##" ;
     private static final String HEADER_STRING = "Authorization" ;
     private static final String TOKEN_PREFIX = "Bearer";

static void addAuthentication(HttpServletRequest req,
                              HttpServletResponse res,
                              String username,
                              UserServices userService){
     Calendar timeTillExpr = Calendar.getInstance();
      timeTillExpr
           .setTime(new Date(System.currentTimeMillis()));
      timeTillExpr.add(Calendar.YEAR,1);
```

```
            String token = Jwts.builder()
                              .setSubject(username)
                              .setExpiration(timeTillExpr.getTime())
                              .signWith(SignatureAlgorithm.HS512,SECRET)
                              .compact();
            res.addHeader(HEADER_STRING,TOKEN_PREFIX + " " +token);

      }
static Authentication getAuthentication(HttpServletRequest req,
                                        UserServices userService){
            String token = req.getHeader(HEADER_STRING);
            if(token == null){
                 token = req.getParameter("key");
            }
            String username;
            if(token != null){

                try{
                  username =Jwts.parser().setSigningKey(SECRET)
                     .parseClaimsJws(token.replace(TOKEN_PREFIX,""))
                     .getBody()
                     .getSubject();
                }catch(SignatureException ex){
                  username = null;
                }
                if(username != null){

                   AppUser user = userService
                                 .getUserByUsername(username);
                   List<GrantedAuthority> grantedAuthorityList=
                                                    new ArrayList<>();

                   for (Role: user.getRoles()) {
                         grantedAuthorityList
                                      .add(
                                       new SimpleGrantedAuthority(
                                       role.getRoleName()));
                   }
                  return new UsernamePasswordAuthenticationToken(
                                       user.getUsername(),
                                       null,
                                       grantedAuthorityList);
                 }
            }
                  return new UsernamePasswordAuthenticationToken(
                                       null,
                                       null,
                                       null);
    }
```

Kôd 4.5. `JwtTokenUtil` klasa

Metoda `addAuthentication` stvara novi JWT token, koristeći korisničko ime kao subjekt *tokena*. Metoda `getAuthentication` provjerava valjanost *tokena,* te vraća odgovarajuću autentikaciju.

Komunikacija API-a s vanjskom mrežom je riješena koristeći REST princip. REST (engl. *Representational State Transfer*) je stil arhitekture koji definira pravila izrade web usluga. Usluge koje odgovaraju REST principu imaju mogućnost interoperabilnost s drugim sustavima na internetu. Aplikacije koje su napisane po pravilima REST-a se nazivaju RESTful, te im se radi toga poboljšava iskoristivost. Korisnici i druge aplikacije koji koriste RESTful servise mogu na vrlo lak način razumjeti i koristiti krajnje točke API sustava, jer su rađene po određenom „standardu".

Kao dodatni sloj apstrakcije poslovnog sloja od baze, uveden je u *Service* sloj kao uzorak dizajna, kako bi se svi JPA repozitoriji mogli logički organizirati. Taj dodatni sloj apstrakcije daje poslovnom sloju neovisnost o implementaciji repozitorija ako se npr. u budućnosti promijeni iz JPA-a u nešto drugo. Sloj apstrakcije je postignut tako da *kontroleri* ne uključuju repozitorij u svoj kod, već uključuju sučelje servisa. Sam *kontroler* nije svjestan kakvu je implementaciju sučelja dobio, niti zna koji se repozitorij krije u implementaciji, samo zna da ima servis, koji ima određene metode. Ovaj sloj pomaže i pri rješavanju potencijalnih iznimki koje se mogu dogoditi prilikom dohvaćanja podataka. Na ovaj način se pogreške ne propagiraju na kontrolere, već su riješene u samom servisu, a kontroler samo serijalizira i šalje poruke van.

4.2.3. JMX Manager Sloj

Iako *JMXManager* sloj službeno pripada poslovnom sloju, njegove funkcionalnosti se jako razlikuju od drugih klasa/servisa u poslovnom sloju. Stoga je odlučeno da će se JMX manager sloj opisati u vlastitom odlomku.

JMX *manager* sloj je glavna funkcionalnost ovog cijelog rada i bez njega rad ne bi imao funkciju. Komunikacija između JMX sloja i web aplikacije riješena je pomoću *WebSocketa*.

WebSocket je komunikacijski protokol koji dopušta komunikaciju između dvije točke u oba smjera koristeći jednu TCP konekciju. To daje mogućnost da klijent ne mora stalno otvarati novu konekciju prema JMX podatcima, već će otvoriti jednu konekciju i svakih nekoliko sekundi primati svježe podatke.

Nakon što je SDK pokrenut na udaljenoj aplikaciju i korisnik se uspješno spojio s *WebSocketom*, komunikacija s JMX Slojem API-a počinje sa slanjem GET *requesta* na *URL:* `-/jmx/connections/{type}/{jmxKey}` gdje `type` predstavlja vrstu željenih podataka performansi, a `jmxKey` predstavlja ključ udaljenog računala. Nakon ovog poziva `JMXService` stvara novu konekciju preko JMX API-a, te stvara novi *Workload* koji će dobiti podatke preko JMX API-a, te ih poslati prema otvorenom *soketu* koristeći `SimpleMessagingTemplate`.

JMX Online API podržava dohvaćanje nekoliko vrsta performansi, kod 4.6. i tablica 4.1. prikazuju i objašnjavaju sve vrste performansi koje se mogu zatražiti od JMX Online API-a.

```
public enum WorkloadType {
    BASIC_PERFORMANCE,
    THREAD_DATA,
    CLASS_LOADER_INFO,}
```

Kôd 4.6. Vrste performansi

Tablica 4.1. Objašnjenje vrsta performansi

`BASIC_PERFORMANCE`	Daje osnovne informacije o samom računalu na kojem se udaljena aplikacija pokreće kao što su CPU opterećenje, količina slobodne memorije, te ime operacijskog sustava i arhitektura.
`THREAD_DATA`	Daje popis svih niti koji se trenutno odvijaju u JVM-u aplikacije.
`CLASS_LOADER_INFO`	Daje popis svih uključenih i isključenih klasa od trenutka pokretanja JVM-a.

Svaka vrsta podataka performansi po korisniku je predstavljena sa svojim `Workload` objektom. Svaki `Workload` objekt je ujedno i `Runnable` objekt gdje se njegova `run` metoda može pokretati unutar nove niti. Ova funkcionalnost je pojednostavljena tako da svaki `Workload` ima referencu na vlastitu nit, te sam sebe može zaustaviti i pokrenuti ovisno o određenim postavljenim `boolean` zastavama. Kod 4.7. prikazuje sučelje `Workload`.

```
public interface Workload extends Runnable {
    void addConnection(SocketConnection connection);
    void setOff();
    void setOn();
    void pause();
    void resume();
    WorkloadType getType();}
```

Kôd 4.7. Sučelje `Workload`

Implementacija samog `Workload` sučelja je poprilično velika tako da je opisana samo glavna petlja same implementacije u kodu 4.8.

```
        {...}
        @Override
public void run() {
     while (on){
          if (paused == false && connection != null) {
             if (connection.getKeepAlive() < MAX_TIMEOUT) {
                   try {
                      sendAndConvert(type, connection);
                   } catch (IOException e) {
                      e.printStackTrace();
                      on = false;
                   }
             }
          }
          try {
             Thread.sleep(SLEEP_TIME);
          } catch (InterruptedException e) {
             e.printStackTrace();
             on= false;
          }
     }
}
private void sendAndConvert(WorkloadType type,
                            SocketConnection connection)
                            throws IOException {
      JmxData data = factory.getJmxData(connection,type);
      String url = constructURL(data.getUrl(),connection);

      template.convertAndSend(url,data.getData());
       }{...}
```

Kôd 4.8. `Workload` implementacija

Svaki `Workload` objekt je ujedno i nit koja je u beskonačnoj petlji i prilikom svakog prolaza `Workload` pregledava sve zastave i postavlja svoje stanje prema njima, te pokreće metodu `sendAndConvert()`. Ova metoda dohvaća podatke preko JMX API-a prema tipu

koji je poslan, te ih šalje pomoću `template` objekta tipa `SimpMessagingTemplate` na URL koji se kreira prema vrsti podataka performansi i `jmxKey` podatku.

Za stvaranje i brisanje ovih `Workload` objekata se brine `JmxManager` klasa koja za svaku novu konekciju pokreće novi `Workload`, a ima mogućnost pauzirati, nastaviti, te zaustaviti svaki `Workload`. Kod 4.8. prikazuje implementaciju `JmxManager` klase.

```
public class JmxManager {

    Map<String, List<Workload>> workloads = new HashMap<>();

  public void addConnection(WorkloadType type,
                            SocketConnection connection,
                            SimpMessagingTemplate template) {
      boolean match = false;
      List<Workload> w = workloads.get(connection.getApiKey());
      if (w != null) {
         match = w.stream()
                  .anyMatch(
                  p -> p.getType().equals(type));
      }
      if (!match) {
         Workload = createNewWorkload(type, connection, template);
         if (workloads.containsKey(connection.getApiKey())) {
               workloads.get(connection.getApiKey())
                        .add(workload);
         } else {
               List<Workload> tmp = new ArrayList<>();
               tmp.add(workload);
               workloads.put(connection.getApiKey(), tmp);
         }
      }
  }
    public void removeConnection(WorkloadType type,
                                 String apikey){
         List<Workload> workload = workloads.get(type);

         for (Workload w : workload) {
              if (w.getType().equals(type)) {
                  w.pause();
                  w.setOff();
                  w.resume();
              }
         }

    }
   createNewWorkload(WorkloadType type,
                     SocketConnection connection,
                     SimpMessagingTemplate template) {
          WorkloadImpl workload = new WorkloadImpl(template);
          workload.setType(type);
```

```
                workload.addConnection(connection);
                workload.setOn();
                return workload;
        }
    }
```

Kôd 4.9. Implementacija `JMXManager` klase

`JmxManager` klasa je glavna klasa unutar JMX sloja, te sadrži reference na sve niti trenutno pokrenute na sustavu kako bi mogla upravljati s njima, te da ih ne bi *garbage collector* pokupio. Također `JmxManager` ima mogućnost pokretanja nove niti, te zaustavljanja postojećih niti.

Radi prirode rada sustava putem *WebSocketa*, sama `workload` nit ne zna da li je netko sluša na drugoj strani komunikacijskog kanala. Stoga svaka nit u sebi ima interni brojač koji se povećava za jedan nakon svakog prelaska petlje, a resetira se kada korisnik pošalje `ping` prema toj niti. Ako brojač unutar niti pređe zadani limit, nit zaustavlja samu sebe. Svaku nit je moguće postaviti da nikad samu sebe ne zaustavi, u slučaju da korisnik treba 24/7 pristup podatcima o performansi, čak i kada nije nužno na aplikaciji. No ova funkcionalnost trenutno nije dostupna preko web sučelja. U trenutku kada korisnik više ne gleda podatke svoje aplikacije, sve niti te aplikacije se gase i brišu.

`JmxManager` klasa periodično diže svoju nit nazvanu "The Warden", koja skenira sve niti unutar `workloads` mape, te briše one koje su zaustavljene. Ova je odluka donesena kako bi se smanjilo opterećenje na sam API sustav.

4.3. Web aplikacija

Web aplikacija predstavlja vizualni modul JMX Online API-a, te je rađena u *Angular* 6 okviru. Takav pristup odvaja web prezentacijski sloj od pozadinskog sloja, te daje pozadinskom dijelu mogućnost samostalnog rada. Korisnici JMX Online sustava nisu prisiljeni koristiti web stranicu ako to ne žele. Na API se mogu spojiti preko URL-a dobiti podatke u JSON formatu i prikazati ih kako god žele u svojim aplikacijama. Cijela web aplikacija je podijeljena u 4 dijela, te svaki dio obavlja određenu funkciju. Tablica 4.2. objašnjava svaki od modula.

Tablica 4.2. Moduli Web aplikacije

Container modul	Ovaj modul se bavi s kontaktiranjem API servisa, te slanjem podatka prema *screen* modulu.
Screen modul	Ovaj modul se bavi prikazom dobivenih podataka od strane *Container* modula.
Model modul	Model modul sadrži objektnu reprezentaciju JSON podataka koje dobije od strane API servisa.
Shared modul	Unutar *shared* modula se nalaze sve komponente koje generalno koristi svaki drugi modul unutar aplikacije. Kao što su servisi, navigacija i sl.

4.3.1. *Container* komponente

```
export class MachineDataContainerComponent implements OnInit {
      machine: ManagedMachine;
      navigationSubscription: any;
  constructor(private currentUserService: CurrentUserService,
              private active: ActivatedRoute,
              private machineSocketService:MachineSocketService,
              private machineService: ManagedMachineService,
              private router: Router) {}
  ngOnInit() {
      this.initComponent();
  }
  private initComponent() {
      if (this.currentUserService.isUserLoggedIn()) {
          this.active.params.subscribe(params => {
             this.machineService.getMachineById(params['id'])
                 .subscribe((data) => {
                  this.machine = data;
                });
          });
      } else {
        this.router.navigate(['/login'])
      }
  }
  saveConfiguration(machine: ManagedMachine) {
      this.machineService.saveConfiguration(machine)
                  .subscribe((data) => {});
}}
```

Kôd 4.10. Implementacija *Container* komponente

Kod 4.10. prikazuje implementaciju *Container* komponente za dohvaćanje podataka u udaljenoj aplikaciji i računalu. *Container* komponenta pri svojoj inicijalizaciji dohvaća podatke s API servisa potrebne za prikaz pogleda, te nakon toga referencira *screen* komponentu iz modula *screen*, te mu predaje potrebne podatke i postavlja *callback* metode za bilo kakve akcije koje korisnik može raditi na pogledu. Sama *container* komponenta po potrebi može i namjestiti određene CSS stilove za cijeli *container*, u kojem će se ubrizgati *screen* komponenta, kao npr.: margine, font i sl.

4.3.2. *Screen* komponente

Screen komponente služe za prikazivanje podataka u vizualnom obliku, te za komunikaciju s korisnikom aplikacije. Sva komunikacija se odvija pomoću događaja (engl. *Event*). Korisnik pritisne gumb na stranici i događaj je okinut unutar TypeScript datoteke, te komponente. Komponenta zatim može pravilno reagirati na događaj, kao što je npr. navigacija, animacija i sl. U slučaju da događaj zahtijeva komunikaciju s API-em *screen* komponenta šalje događaj jedan nivo iznad u *container* komponentu, gdje se onda kontaktira API servis, nakon čega se šalju novi podatci *screen* komponenti. *Angular* platforma sama ponovno učitava novo dijete u slučaju da roditelj promijeni podatke koji su predani djetetu na korištenje.

```
<div class="outer" fxLayout="row" fxLayoutAlign="center center">
    <form(ngSubmit)="submitForm($event)"
         [formGroup]="registerFormGroup">
       <mat-card>
         <mat-card-header class="header">
              Register
         </mat-card-header>
            <mat-card-content fxLayout="column">
              <mat-form-field>
                <input [errorStateMatcher]="matcher"
                       formControlName="email"
                        matInput placeholder="email">
                <mat-error *ngIf="email.hasError('email')
                          && !email.hasError('required')">
                  Please enter a valid email address
                </mat-error>
              </mat-form-field>
              <mat-form-field>
                <input formControlName="username" matInput
                           placeholder="Username">
                <mat-error *ngIf="username.hasError('required')">
                 Username is required
                </mat-error>
              </mat-form-field>
```

```
              <mat-form-field>
                <input [errorStateMatcher]="matcher"
                           formControlName="password"
                           matInput placeholder="Password">
                <mat-error *ngIf="password.hasError('minlength')
                        && !password.hasError('required')">
                  Number of characters is too short,please add
                    {{minlength - password.value.length}} more
                </mat-error>
              </mat-form-field>
            </mat-card-content>
            <mat-card-actions>
              <button mat-raised-button
                      color="primary">
                  Register
              </button>
              <button [routerLink]="['/login']"
                       mat-raised-button color="accent">
                    Cancel
              </button>
             </mat-card-actions>
        </mat-card>
    </form>
 </div>
```

Kôd 4.11. *HTML* datoteka *screen* komponente

```
@Component({
  selector: 'app-register-screen', // definiranje selektora
                                                za referenciranje
  templateUrl: './register-screen.component.html'//html datoteka
  styleUrls: ['./register-screen.component.css']//css datoteka
})
export class RegisterScreenComponent implements OnInit {

  // događaj za okidanje container komponente
  @Output()
  register: EventEmitter<AppUser> = new EventEmitter<AppUser>();
  minlength = 5;
  constructor(private _fb: FormBuilder) {}
  registerFormGroup: FormGroup;
  email;
  username;
  password;
  matcher = new PasswordStateMatcher();

  ngOnInit() {
      this.registerFormGroup =
      this._fb.group({
        email: ['', Validators.compose(
               [Validators.required,Validators.email])],
          username: ['', Validators.required],
          password: ['', Validators.compose(
                    [Validators.required,
```

```
                    Validators.minLength(this.minlength)]],
        });

        this.email = this.registerFormGroup.get('email');
        this.username = this.registerFormGroup.get('username');
        this.password = this.registerFormGroup.get('password');
    }
    submitForm(e) {
        e.preventDefault();
        const email = this.registerFormGroup
                          .controls.username.value;
        const username = this.registerFormGroup
                              .controls.username.value;
        const password = this.registerFormGroup
                              .controls.password.value;
           if (this.registerFormGroup.valid) {

               const user: AppUser = new AppUser();
               user.email = email;
               user.username = username;
               user.password = password;

               this.register.emit(user); //okidanje događaja za
                          registraciju ako je forma validna
            }
    }
}
```

Kôd 4.12. *TypeScript* datoteka *screen* komponente

Kod 4.11. i 4.12. opisuju jednu *screen* komponentu, zajedno s TypeScript datotekom i *HTML* datotekom. Kod 4.12 predstavlja vizualni dio koji će korisnik vidjeti. Uz osnovne HTML oznake poput `<div>` i `<input>`, Unutar HTML datoteke su dodane u druge oznake poput `<mat-card>` i `<mat-card-header>`. Ove oznake predstavljaju komponente koje na sebi imaju primijenjen *material design*. `<form>` oznaci su dodana dva svojstva. `(ngSubmit)` oznaka registrira pretplatu na podnošenje ove forme te predaje ime *callback* metode koja će rukovati s događajem (u ovom slučaju, registrirati korisnika). `[formGroup]` svojstvo registrira objekt koji će predstavljati formu unutar TypeScript datoteke (kod 4.12). U ovom slučaju to je objekt `registerFormGroup`. Također svaki ulazni parametar unutar (`<input>` oznaka) forme je mapiran na određeno svojstvo unutar `formGroup` objekta korištenjem svojstva `formControlName`. Na taj način je moguće prenositi unesene podatke na HTML formi unutar TypeScript datoteke. Ovaj način se također koristi i za provjeravanje forme, gdje se razni *validatori* koji su mapirani na svojstva

`registerFormGroup` objekta, propagiraju na HTML formu koja može prikazati određene validacijske poruke ako ih ima.

4.3.3. Servisi

Angular aplikacija kontaktira određene dijelove API-a pomoću servisa. Ovaj pristup je korišten iz razloga lakšeg rukovanja potencijalnih grešaka, te lakšeg upravljanja stanjem aplikacije. Svaki servis je dio *shared* modula, te komunicira sa samo jednim dijelom API sustava. S takvim pristupom servis ispunjava samo određenu uslugu kao npr. `loginRegister` servis komunicira s krajnjim točkama za prijavu i registraciju korisnika, te s time daje mogućnosti korisnicima da se prijave i registriraju u aplikaciju. *Login screen* i *container* komponenta se ne mora brinuti o kontaktiranju API sustava kako bi korisnika prijavili, već pitaju servis da to učini, a za ostale detalje se ne brinu.

Servisi koriste *Angularov* `HttpClient` servis za komunikaciju s API-em koji, koristeći modele automatski mapira dobiveni JSON u objektnu reprezentaciju podataka.

Kod 4.13. pokazuje primjer servisa.

```
export default class JmxService {
  constructor(private _http: HttpClient,
              private apiService: ApiUrlService,
              private currentUserService: CurrentUserService) {}
  startConnection(type: string, apiKey: string):Observable<any> {
      const headers = this.currentUserService.getAuthHeader();
      return this._http.get<any>(this.apiService
                                      .jmxConnections +
                                       type + '/' + apiKey,
                                  {observe: 'response',
                                   headers: headers })
                 .pipe(catchError(
                       this.handleError<any>('jmxConnection')));
  }
  private handleError<T> (operation = 'operation', result?: T) {
      return (error: any): Observable<T> => {
          console.error(error); // log to console instead
          return of(result as T);
       };
  }
}
```

Kôd 4.13. Implementacija servisa `JMXService`

4.3.4. Ruter

```
const appRoutes: Routes = [
      {path: 'login', component:
            LoginContainerComponent},
      {path: 'dashboard', component:
            DashboardContainerComponent},
      {path: 'machineList', component:
            MachineListContainerComponent},
      {path: 'machineDetails/:id', component:
            MachineDataContainerComponent},
      {path: 'getStarted', component:
            GetStartedContainerComponent},
      {path: '', component:
            MachineListContainerComponent},
      ];
      @NgModule({
        imports: [
          CommonModule,
          RouterModule.forRoot(appRoutes)
        ],
        declarations: [],
        exports: [RouterModule]
      })
```

Kôd 4.14. Implementacija rutera web aplikacije

Glavna navigacija kroz aplikaciju odvija se u *routing* modulu koji se nalazi unutar *shared* modula, a opisan je u kodu 4.14. *Routing* modul daje korisniku iluziju navigacije po web aplikaciji. URL se mijenja, novi pogledi se učitavaju, ali u pozadini se samo *HTML* komponente ubrizgavaju u „index.html“ stranicu.

Svaka ruta je mapirana na određenu *container* komponentu koja se ubrizgava u „index.html“ u trenutku kada se promijeni.

4.3.5. JMX komunikacijski dio prezentacijskog sloja

Radi toga što podatci o performansi moraju biti konstantno najnoviji taj dio web aplikacije ne prati *container-screen*, već postoji samo jedna *container* komponenta u kojoj se referencira jedna *screen* komponenta, koja dinamično ubrizgava ostale komponente. Cijela komunikacija s JMX slojem API-a odvija se preko *WebSocketa*, specifičnije *StompJS* biblioteke za *WebSockete*, te počinje s `MachineDataContainer` komponentom gdje se

stvara inicijalno rukovanje s *WebSocket* poslužiteljem API-a, te se otvara novi kanal za komunikaciju. Nakon toga *container* komponenta referencira *screen* komponentu koja, prema izboru korisnika dinamički ubrizgava određenu *screen* komponentu za prikaz podataka. To postiže koristeći `ConfigurationService` servis koji je opisan u kodu 4.15. `ConfigurationService` ima dvije metode, `assembleConfiguration`, i `addNewItem`. `assembleConfiguration` čita ulančani `string`, te po njemu, koristeći `addNewItem` metodu, dinamički dodaje komponente unutar pogleda koji mu je zadan (u ovom slučaju `viewContainerRow` i `viewContainerFullscreen` pogledi).

```
      @Injectable()
export default class ConfigurationService {
   constructor(private _componentFactoryResolver:
                        ComponentFactoryResolver) {}
   assembleConfiguration(machine: ManagedMachine,
                         viewContainerRow: ViewContainerRef,
                         viewContainerFullScreen: ViewContainerRef){

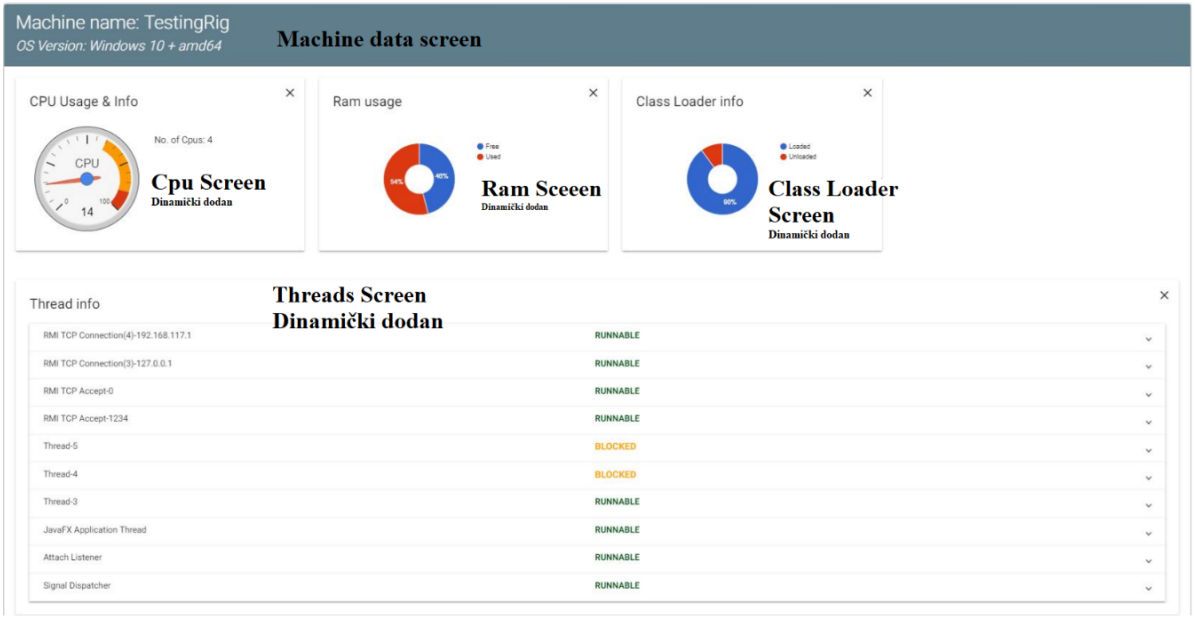
        const config = machine.configuration.split(';');
        const children: ComponentRef<any>[] = new Array();
        config.forEach((element, index) => {
           const child = this.addNewItem(element,
                                         viewContainerRow,
                                         viewContainerFullScreen,
                                         machine.apiKey);
           if (child) {
              children.push(child);
           }
         });
          return children;
   }


   addNewItem(itemType: string,
              viewContainer: ViewContainerRef,
              viewContainerFullscreen: ViewContainerRef,
              apiKey: string) {
        if (itemType === 'viewCPU') {
            const componentFactory = this._componentFactoryResolver
                                         .resolveComponentFactory(
                                          CPUComponent);
            const cpuChild =viewContainer.createComponent(
                                          componentFactory);
           cpuChild.instance.apiKey = apiKey;
           cpuChild.instance.selfRef = cpuChild;
           cpuChild.instance.name = 'viewCPU';
           return cpuChild;
```

```
        } else if (itemType === 'viewMemory') {
           const componentFactory = this._componentFactoryResolver
                                        .resolveComponentFactory(
                                         RAMComponent);
           const ramChild = viewContainer.createComponent(
                                                 componentFactory);
           ramChild.instance.apiKey = apiKey;
           ramChild.instance.selfRef = ramChild;
           ramChild.instance.name = 'viewMemory';
           return ramChild;
        } else if (itemType === 'viewThreads') {
            const componentFactory = this._componentFactoryResolver
                                        .resolveComponentFactory(
                                         ThreadsComponent);
           const threadChild = viewContainerFullscreen.createComponent(
                                                    componentFactory);
           threadChild.instance.apiKey = apiKey;
           threadChild.instance.selfRef = threadChild;
           threadChild.instance.name = 'viewThreads';
           return threadChild;
        } else if (itemType === 'viewClassInfo') {
           const componentFactory = this._componentFactoryResolver
                                        .resolveComponentFactory(
                                         ClassLoaderComponent);
           const classChild = viewContainer.createComponent(
                                            componentFactory);
           classChild.instance.apiKey = apiKey;
           classChild.instance.selfRef = classChild;
           classChild.instance.name = 'viewThreads';
           return classChild;
        }
        return false;
  }
}
```

Kôd 4.15. Implementacija servisa `ConfigurationService`

Svaka *screen* komponenta za prikaz podataka pretplaćuje se na određeni put (engl. *path*) kako bi dobila podatke koji joj trebaju za prikaz. Slika 4.3 ilustrira dinamičko učitavanje ekrana:

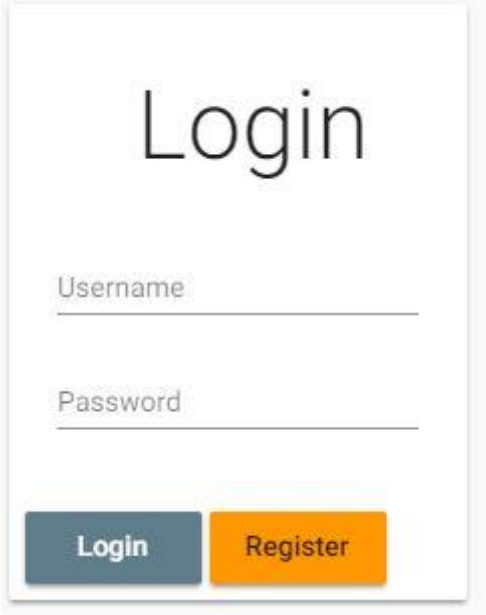


Slika 4.3. Dinamičko učitavanje ekrana

5. Korisničko iskustvo

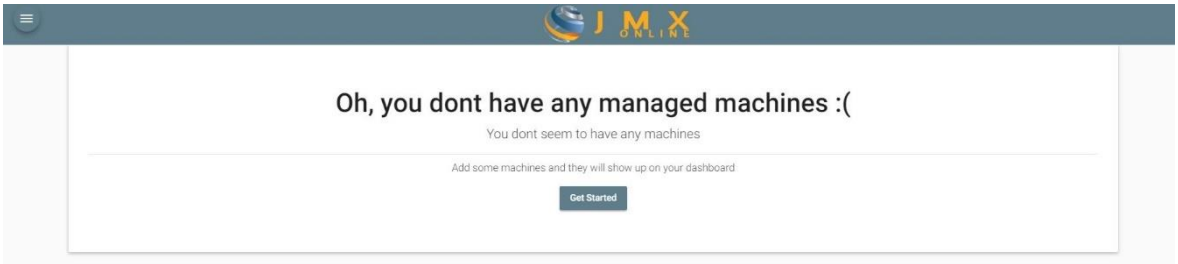
5.1. Administracija

Kada korisnik prvi puta otvori web aplikaciju, prezentiran je s login sučeljem (Slika 5.1.).



Slika 5.1. Login sučelje

Nakon toga, korisnika se vodi na početnu stranicu gdje, ako nema niti jedan udaljeno računalo, dobije obavijest u kojoj može pokrenuti proces izrade novog udaljenog računala (slika 5.2.).

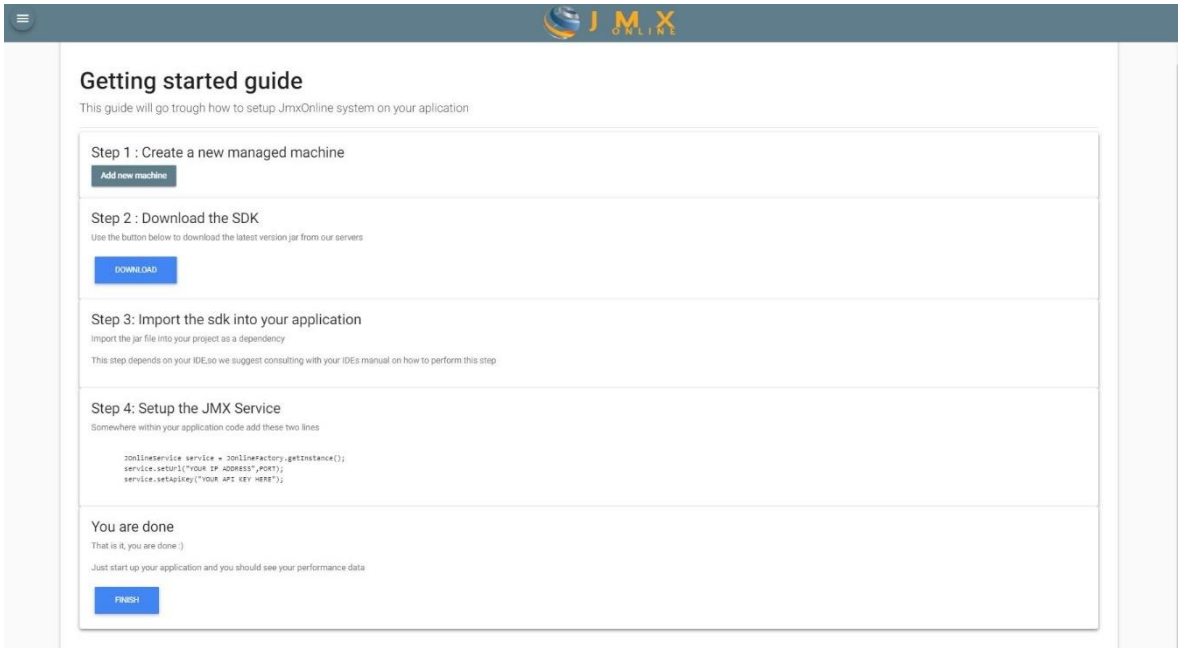


Slika 5.2. Početna stranica kada korisnik nema udaljenih računala

5.2. Stvaranje udaljenog računala

Udaljeno računalo se može stvoriti na dva načina.

Jedan je da korisnik napravi novo računalo kroz početni vodič (slika 5.3.) gdje ga se korak po korak vodi kroz postupak izrade udaljenog računala. Od preuzimanja .jar SDK datoteke, do samog generiranja ključa, te pokretanja aplikacije.

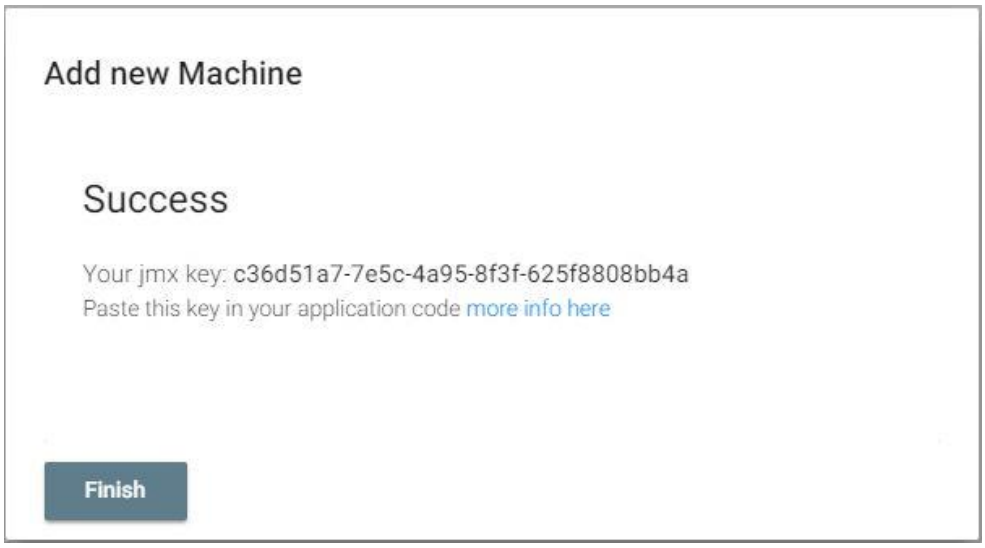


Slika 5.3. Vodič za stvaranje novog udaljenog računala

Drugi način je da korisnik radi dodatne udaljene aplikacije gdje se onda preskače uvodni vodič, već se otvara dijalog u kojem se prolazi samo kroz proces imenovanja i generiranja *jmxKeya*, a preskače se dio s .jar datotekom (slika 5.4. i 5.5.).



Slika 5.4. Dijalog za stvaranje udaljenog računala (korak 1)



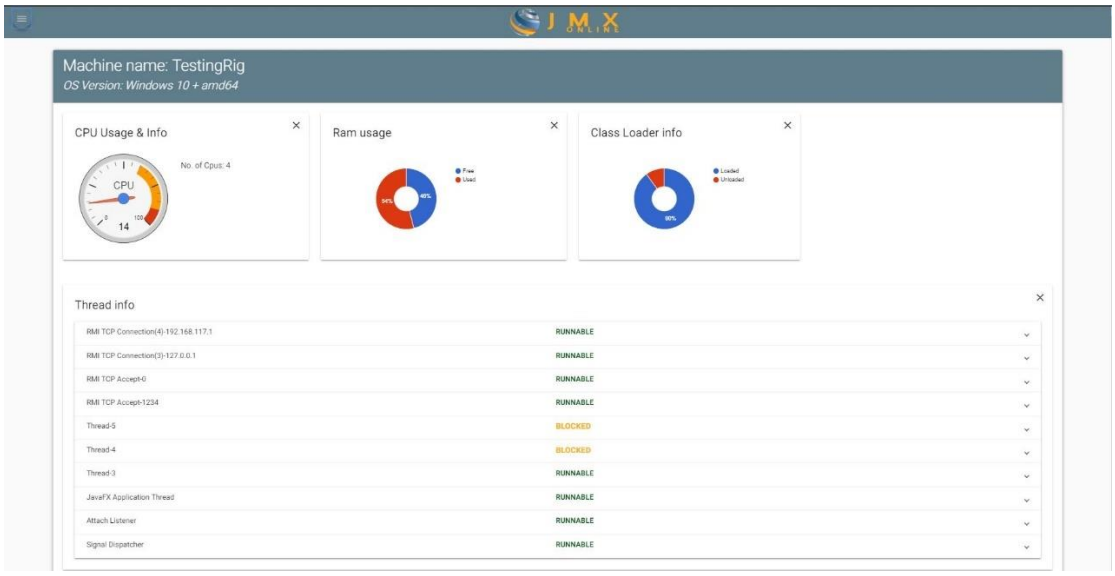
Slika 5.5. Dijalog za stvaranje udaljenog računala (korak 2)

5.3. JMX Podatci o performansama

JMX podatci o performansama aplikacije i virtualne mašine na udaljenom računalu se prikazuju na dva mjesta unutar aplikacije.

Jedan je ako korisnik ima neka udaljena računala postavljena kao favorit. Tada se na *dashboard* ekranu pokreće dohvaćanje podataka, te se prikazuju podatci.

Podatci se prikazuju automatski samo ako korisnik ima spremljenu kombinaciju podataka koje želi vidjeti, u protivnom dobije obavijest da prvo treba dodati bar jednu vrstu podataka proje nego što će vidjeti rezultate. Slika 5.6. pokazuje aplikaciju u funkciji kada je sve uredu.



Slika 5.6. Sustav u funkciji

Zaključak

Tijekom razvoja, testiranja aplikacije, te kod već *deployanih* aplikacija, jako često se postavlja pitanje, „zašto sad aplikacija radi sporo“, „zašto se server ruši“. Zašto se, nakon nekoliko sati „kopanja“ po logoima dobije objašnjenje : „Server nije imao dovoljno memorije“ ili je aplikacija imala problem istjecanja memorije i sl.

U ovome radu se postigla aplikacija koja kroz par linija koda na korisnikovoj strani, dobije mnoštvo podataka o trenutnom stanu aplikacije, te o podatcima trenutnog stanja resursa u aplikaciji.

Aplikaciju je moguće integrirati s udaljenom aplikacijom koja je tek u fazi razvoja, ali i s aplikacijom koja je već negdje postavljena, zbog toga što koristi standardizirani sustav dohvaćanja podataka putem JMX-a.

Budućnosti aplikacije je proširivanje tipova podataka koje može prikazivati (čak i izvan JMX okvira) poput lokacije računala, GPU komponente itd. Radi toga što je aplikacija odvojena u tri sloja sve buduće nadogradnje ne bi trebale biti komplicirane za napraviti, te se mogu ekstenzivno testirati unutar jednog sloja prije nego što se koriste u drugom.

Popis kratica

API	*Application programming interface*	Aplikacijsko programsko sučelje
GPU	*Graphics processing unit*	Grafička kartica
HTML	*HyperText Markup Language*	Jezik za izradu izgleda web stranica
JMX	*Java Management Extensions*	API za udaljeno upravljanje resursima
JPA	*Java Persistance API*	API za upravljanje bazom
ORM	*Object relational mapping*	Mapiranje tablica baze u objekte
REST	*Representational State Transfer*	Stil arhitekture API sustava
SDK	*Software Development Kit*	Skupina alata za razvoj aplikacija
URL	*Uniform Resoruce Locator*	Usklađeni lokator sadržaja

Popis slika

Slika 3.1. Najčešće korišteni moduli *Spring* radnog okvira 5

Slika 3.2. Dijelovi *Angular* modula 7

Slika 3.3. Ekran izrađen po *Material Design* principima 8

Slika 3.4. Slojevi JMX API sustava 10

Slika 3.5. Komunikacija u oba smjera pomoću *websocketa* 11

Slika 4.1. Arhitektura sustava 12

Slika 4.2. Dijagram baze podataka 16

Slika 4.3. Dinamičko učitavanje ekrana 34

Slika 5.1. Login sučelje 35

Slika 5.2. Početna stranica kada korisnik nema udaljenih računala 35

Slika 5.3. Vodič za stvaranje novog udaljenog računala 36

Slika 5.4. Dijalog za stvaranje udaljenog računala (korak 1) 36

Slika 5.5. Dijalog za stvaranje udaljenog računala (korak 2) 37

Slika 5.6. Sustav u funkciji 37

Popis tablica

Tablica 3.1. Moduli *Spring* radnog okvira 4

Tablica 4.1. Objašnjenje vrsta performansi 22

Tablica 4.2. Moduli Web aplikacije 26

Popis kôdova

Kôd 3.1. Primjer *HTML* datoteke *Angular* komponente 8

Kôd 4.1. *Snippet* za pokretanje JMX servisa na udaljenoj aplikaciji 13

Kôd 4.2. Koraci pri pokretanju JMX servisa na udaljenoj aplikaciji 14

Kôd 4.3. Proces registracije udaljene aplikacije na JmxOnline API 15

Kôd 4.4. Primjer modela tablice `managed_machine` 17

Kôd 4.5. `JwtTokenUtil` klasa 20

Kôd 4.6. Vrste performansi 22

Kôd 4.7. Sučelje `Workload` 23

Kôd 4.8. `Workload` implementacija 23

Kôd 4.9. Implementacija `JMXManager` klase 25

Kôd 4.10. Implementacija *Container* komponente 26

Kôd 4.11. *HTML* datoteka *screen* komponente 28

Kôd 4.12. *TypeScript* datoteka *screen* komponente 29

Kôd 4.13. Implementacija servisa `JMXService` 30

Kôd 4.14. Implementacija rutera web aplikacije 31

Kôd 4.15. Implementacija servisa `ConfigurationService` 33

Literatura

[1] WALLS, C, Spring Boot in Action: Manning, Prosinac 2015.

[2] MURRAY, N, COUNRY,F, LERNER A, TABORDA, C, *ngbook* : FULLSTACK.io, 2017.

[3] SULLINS, B , WHIPPLE, Jmx in Action: Manning, 2003.

[4] WALLS, C, Spring in Action:Manning, 2015.

Prilog

Uz Priloženi CD s kodom rada, kod se može pronaći i na:

https://gitlab.com/JmxOnline/JmxOnlineWebApp.git

https://gitlab.com/JmxOnline/JmxOnlineWeb.git

https://gitlab.com/JmxOnline/JmxOnlineSDK.git

