

**Univerzitet Singidunum**

**Master rad**

**Servisno-orijentisani razvoj aplikacije za predikciju cene zlata**

**Mihailo Šević**

Beograd, 2021.



**Univerzitet Singidunum**

**Departman za poslediplomske studije**

Master akademski program

Softversko i informaciono inženjerstvo

**Master rad**

**Servisno-orijentisani razvoj aplikacije za predikciju cene zlata**

|  |  |
| --- | --- |
| **Mentor**: | **Student**: |
| prof. dr Angelina Njeguš | Mihailo Šević, 2018440018 |
|  |  |

Beograd, 2021. godine

**Apstrakt**

*Poslednjih godina razvojem tehnologije se sve više uviđa potreba za mašinskim učenjem i primenom mašinskog učenja u različitim oblastima poslovanja. Mašinsko učenje postaje sve važnije za različite predikcije dešavanja, posebno na finansijskom tržištu koje je veoma osetljivo na promene, kako prirodne, tako i društvene. Kriza Covid-19 virusa je dovela do toga da ljudi „beže“ u štednju u nečemu opipljivijem od svetskih valuta, a to je zlato. Ta vrsta premeštanja finansijkih sredstava je dovela do naglog poskupljenja cene zlata i sve predikcije rađene na osnovu modela linearne regresije su pale u vodu. Stoga, da bi se pomoglo u donošenju finansijskih odluka u smislu ulaganja u različite valute, osmišljena je aplikacija koja će da davati predikciju sutrašnje cene zlata. Značaj ove aplikacije je lakoća korišćenja za ljude iz netehničkih nauka koji nisu imali priliku da se sreću sa kreiranjem i odabirom modela za predikcije, koji sada jednostavno mogu doći do informacije o predviđenoj ceni samo uz par klikova. Takođe, sekundarna funkcionalnost ove aplikacije je u tome što se ne mora raditi samo predikcija cene zlata, nego i predikcije cena bilo kojeg finansijskog instrumenta, derivata, robe... Praktično se može izvršiti bilo koja predikcija ukoliko postoje odgovarajući ulazni podaci. Za kreiranje ovakve aplikacije korišćena je servisno orijentisana arhitektura unutar koje se nalaze četiri servisa: broker koji predstavlja sponu između ostalih servisa, frontend servis koji prikazuje podatke korisnicima aplikacije, prediktor servis koji priprema, čuva i dostavlja podatke vezane za samu predikciju i modelator servis koji se bavi samim mašinskim učenjem i predikcijama.*

**Ključne reči:** SOA, mašinsko učenje, predikcija cena

**Abstract**

*In recent years, the development of technology has increasingly recognized the need for machine learning and the application of machine learning in various areas of business. Machine learning is becoming increasingly important for various predictions of events, especially in the financial market, which is very sensitive to changes, both natural and social. The crisis of the Covid-19 virus has led to people to save in something more tangible than world currencies, and that is gold. This type of transfer of funds led to a sharp rise in the price of gold and all predictions made on the basis of the linear regression model failled. Therefore, in order to help make financial decisions in terms of investing in different currencies, an application has been designed that will give a prediction of tomorrow's gold price. The significance of this application is the ease of use for people from non-technical backgrounds who have not had the opportunity to encounter the creation and selection of models for predictions, who can now easily get information about the estimated price with just a few clicks. Also, the secondary functionality of this application is various price predictions can be done, not only the gold price prediction, but also the prediction of price of any financial instrument, derivatives, commodities ... Practically any prediction can be made if there is appropriate data input. To create such an application, a service-oriented architecture was used within which there are four services: a broker that represents a link between other services, a frontend service that displays data to application users, a predictor service that prepares, stores and delivers data related to prediction and a modeler service that deals with machine learning and predictions themselves.*

**Key words:** SOA, machine learning, price prediction

**Sadržaj**

[1. Uvod 4](#_Toc69741519)

[2. Servisno-orijentisani razvoj aplikacija 5](#_Toc69741520)

[2.1. Servisno-orijentisana arhitektura 5](#_Toc69741521)

[2.2. Servisi 9](#_Toc69741522)

[2.3. Komunikacija, protokoli i hosting 10](#_Toc69741523)

[3. Algoritmi mašinskog učenja u analizi finansijskih pokazatelja 11](#_Toc69741524)

[4. Analiza naučnih radova i modela za predikciju cene zlata 16](#_Toc69741525)

[4.1. Predikcija cene zlata 16](#_Toc69741526)

[4.2. Analiza naučnih radova 17](#_Toc69741527)

[5. Razvoj aplikacije za previđanje cene zlata primenom mašinskog učenja 21](#_Toc69741528)

[5.1. Osnovne funkcionalnosti aplikacije 21](#_Toc69741529)

[5.2. Korišćene tehnologije 21](#_Toc69741530)

[5.2.1. Docker 22](#_Toc69741531)

[5.2.2. React 23](#_Toc69741532)

[5.2.3. Node.js i Express 25](#_Toc69741533)

[5.2.4. Socket.io 26](#_Toc69741534)

[5.2.5. Got 26](#_Toc69741535)

[5.2.6. Cors 27](#_Toc69741536)

[5.2.7. Flask 27](#_Toc69741537)

[5.2.8. Apscheduler 28](#_Toc69741538)

[5.2.9. Pandas 29](#_Toc69741539)

[5.2.10. Scikit-learn 29](#_Toc69741540)

[5.3. Korišćeni servisi 29](#_Toc69741541)

[5.3.1. Broker 30](#_Toc69741542)

[5.3.2. Frontend 31](#_Toc69741543)

[5.3.3. Prediktor 31](#_Toc69741544)

[5.3.4. Modelator 32](#_Toc69741545)

[5.4. Arhitektura aplikacije 33](#_Toc69741546)

[6. Zaključak 35](#_Toc69741547)

[Literatura 38](#_Toc69741548)

[Prilozi 41](#_Toc69741549)

[Prilog A: Github i struktura fajlova 41](#_Toc69741550)

[Prilog B: Van servisni fajlovi 42](#_Toc69741551)

[Prilog C: Fajlovi Broker Servisa 44](#_Toc69741552)

[Prilog D: Fajlovi Frontend servisa 48](#_Toc69741553)

[Prilog E: Fajlovi Modelator servisa 51](#_Toc69741554)

[Prilog F: Fajlovi Predictor servisa 57](#_Toc69741555)

# Uvod

Ovaj rad govori o primeni servisno orijentisane arhitekture na mašinsko učenje i finansijske predikcije. Svaka aplikacija, softver, ima unutrašnju organizaciju. Ta organizacija naziva se softverskom arhitekturom i najčešće se dešava da krajnji korisnici aplikacije nisu ni svesni postojanja te arhitekture ili bilo kojeg procesa koji se dešava u pozadini same aplikacije.

U ovom radu je za izradu aplikacije za predikciju kretanja cena zlata korišćena servisno orijentisana arhitektura, što znači da su komponente aplikacije organizovane u zasebne servise koji bi mogli da se tretiraju kao zasebne aplikacije. Za realizaciju aplikacije su korišćena četiri servisa/celine:

- Broker (servis koji predstavlja sponu između svih ostalih servisa),

- Prediktor (kontrola toka aplikacije, proračuna),

- Modelator (servis za mašinsko učenje),

- Frontend (prezentacioni sloj).

Same celine su vrlo nezavisne jedna od druge, a međusobno imaju interakciju preko API-ja kroz htttp protokol. Ovakva nezavisnost međusobna komunikacija omogućavaju da se servisi mogu nezavisno koristiti, bez ili sa jako malim unutrašnjim izmenama. Takođe, ovakvom arhitekturom se omogućava da se servisi nalaze na različitim fizičkim lokacijama i da imaju različite oblike hostinga (*serverless*).

Aplikacija predstavlja alat za predikciju sutrašnje cene zlata. Kreiranje ove aplikacije je odabrano jer predstavlja koristan alat za pomoć u donošenju odluka vezanih za ponašanje na finansijskom tržištu. Takođe, odabrana je jer je ovakve odluke veoma teško donositi i u normalnim uslovima, a još teže u periodima velikih nestabilnosti poput trenutne situacije sa COVID-19 virusom. Primena mašinskog učenja za predikciju cena zlata gde se uzimaju u obzir parametri COVID-19 virusa je već proverena sa izuzetno dobrim rezultatima sa preko 95% tačnosti u predikciji (Stokanović Šević, 2020).

Ovakve metode predikcije, iako su vrlo precizne, nisu baš jednostavne za upotrebu jer je potrebno znanje programiranja da bi se došlo do rezultata. Aplikacija koja je izrađena u ovom radu otvara mogućnosti onima koji znaju kakva vrsta predikcija im treba u smislu odabira parametara i finansijskih derivata za predikcije, ali nisu vični programiranju. Ona olakšava pristup različitim modelima mašinskog učenja, sa tim da daje i prikaz rada modela na primeru predikcije sutrašnje cene zlata.

# Servisno-orijentisani razvoj aplikacija

## Servisno-orijentisana arhitektura

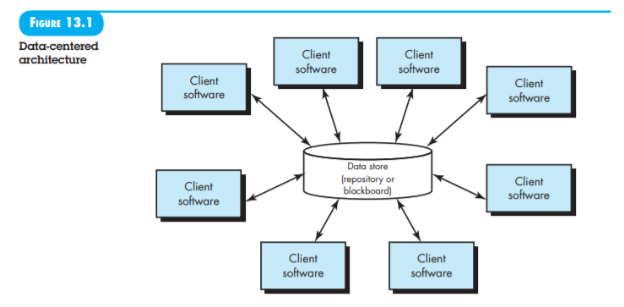
Softverska arhitektura je fundamentalna struktura koja diktira razvoj softvera. Svaka softverska arhitektura se sastoji od elemenata i njihovih međusobnih odnosa. (Clements и остали, 2010)

Značaj arhitekture je u tome da ona predstavlja nacrt željenog softvera. Kao i nacrt za izgradnju kuće, na osnovu arhitekture mogu se odrediti karakteristike i pre nego što sama izgradnja aplikacije počne. Prilikom dizajniranja arhitekture mogu se videti potencijalne poteškoće u budućnosti i na osnovu toga se dizajn treba dodatno usavršiti da bi se predupredile problematične situacije ili bar da se pripremi adekvatna reakcija kad dođe do njih.

Softverska arhitektura kao pojam se prvi put pominje šezdesetih godina prošlog veka, a sporadično do devedesetih. U tom periodu akcenat u rešavanju kompleksnih problema je bio na strukturama podataka i različitim algoritmima. Začeci softverske arhitekture se mogu naći u istraživanjima Edsger Dajkstra 1968 i Davida Parnasa 1970 kada je ustanovljeno da je struktura softvera bitna i da je izuzetno bitno da je ta struktura dobro definisana. (Garlan & Shaw, 1993)

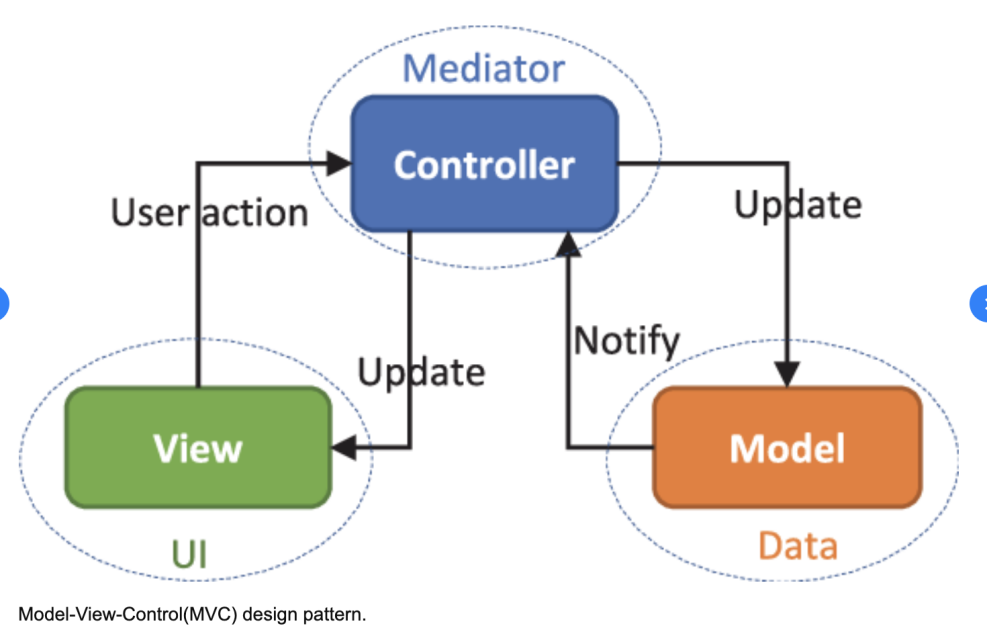
Devedesetih godina prošlog veka dolazi do buma u razvoju softvera sa *dot com* balonom. Sa ubrzanim razvojom softvera dolazi do diferenciranja nekoliko karakterističnih arhitektura.

Jedna od prvih arhtektura je arhitektura orijentisana ka podacima. To znači da su podaci bili skladišteni na jednom mestu, a različiti klijenti su mogli da vrše operacije nad tim podacima. Ova arhitektura pripada periodu *mainframe-ova* i skladišta podataka. (Pressman & Maxim, 2014)

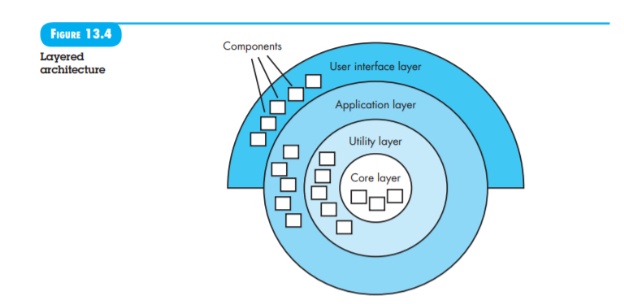


Slika 1. Prikaz arhitekture orijentisane podacima (Pressman & Maxim, 2014)

Najpopularniji arhitektonski pristup, koji je i dalje izuzetno popularan u malim do srednjim softerima je MVC (*model, view, controller*) šablon. Ova troslojna arhitektura je postala popularna iz nekoliko razloga među kojima su objektno orijentisano programiranje i brza postavka inicijalnog rešenja. Brza postavka je omogućena raznim frejmvorcima koji imaju već predefinisane sržne komponente. Objektno orijentisano programiranje dozvoljava nasleđivanje sržnih elemenata što dalje smanjuje vreme izrade i povećava kvalitet samog softvera. (Pressman & Maxim, 2014)



1. MVC - odnos elementa

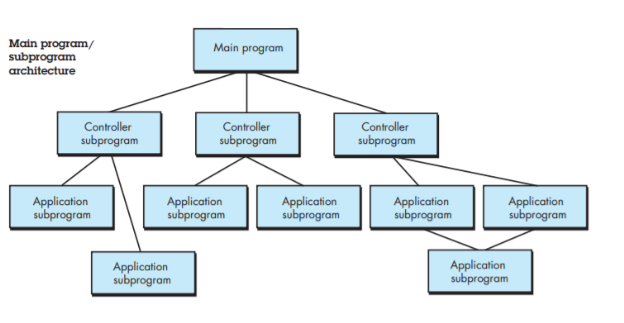


1. MVC - prikaz slojevitosti

Slika 2. Prikaz MVC arhitekture. a) odnos elementa; b) prikaz slojevitosti

(Pressman & Maxim, 2014)

Nešto pre MVC arhitekture korišćena je pozivna arhitektura gde glavni program poziva druge podprograme radi izvršavanja zadataka. Ova arhitektura se može u neku ruku smatrati pretečom servisno orijentisane arhitekture jer je svaki podprogram specijalizovan za određenu radnju, slično kao što svaki servis obavlja zadatke određene poslovne celine. (Pressman & Maxim, 2014)



Slika 3. Prikaz podprogramske arhitekture (Pressman & Maxim, 2014)

Servisno orijentisana arhitektura je vrlo mlad vid arhitekture koji se pojavljuje krajem prve decenije dvadeset i prvog veka. Arhitektura je nastala razvojem *web* servisa, koji su postali upotrebljivi oko 2010e godine sa unapređenjem internet brzina.

Kao što je već napomenuto, svaki softver ima dizajn koji može koristiti jednu ili više arhitektura, a u konkretnom slučaju odabrana je servisno orijentisana arhitektura zbog svoje fleksibilnosti.

Servisno orijentisana arhitektura je nezavisna od tehnologija, proizvoda i dobavljača (*vendor*).

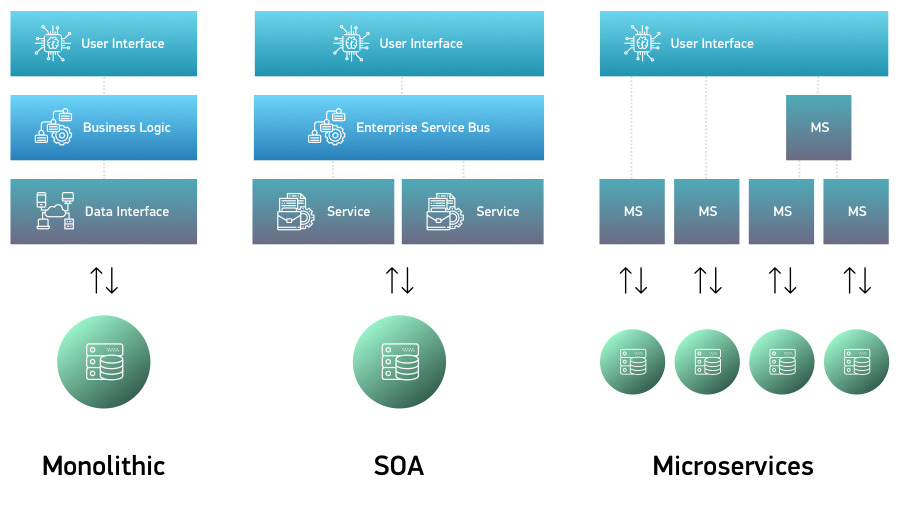
Ova aritektura predstavlja takav međusobni odnos logičkih/funkcionalnih celina da odlično prati poslovnu logiku očekivanog rezultata aplikacije. (Bell, 2008)

Svaka celina se naziva servisom i obavlja svoju ulogu ili nezavisno ili delegirajući određene zadatke drugim servisima koji su sposobni za izvršenje takvih zadataka. Prednosti ovakve arhitekture su konciznost koda i mogućnost testiranja pojedinačnih elemenata(servisa), laka skalabilnost, diverzitet izbora tehnologija servisa kao i fleksibilna distributivnost. (Dragoni и остали, 2017)

Mana ovakve arhitekture je u tome što se lako dolazi u situaciju preteranog broja servisa koje je teško upratiti što vodi do servisa koji rade iste stvari i međusobnog preplitanjima u izvršavanju. Takođe, mana može biti sporost sistema u slučaju da je komunikacija između sistema otežana.

Prilikom dizajniranja servisno orijentisane aplikacije/sistema poželjno je voditi se SOA manifestom (*SOA Manifesto*, 2009) koji se sastoji od šest postulata:

* Poslovna vrednost je bitnija od tehničke strategije
* Strateški ciljevi su bitniji od projektno specifičnih benefita
* Sopstvena interoperabilnost nad specifičnom integracijom
* Deljeni servisi su bitniji od specifičnih servisa
* Fleksibilnost je bitnija od optimizacije
* Evolutivno usavršavanje je bitnije nego insistiranje inicijalnoj perfekciji.



Slika 4. Razlika između monolitne arhitekture, SOA i mikro servisa (Velvetech, 2021)

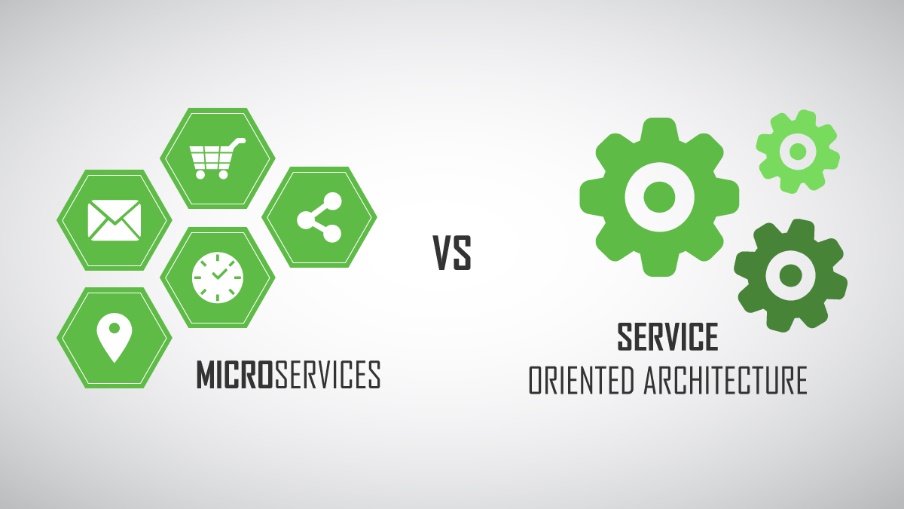
Svaki servis predstavlja jednu zaokruženu poslovnu celinu i može igrati jednu od tri uloge (Rotem-Gal-Oz, 2012):

* Servis provajder, predstavlja sam servis i dužnost mu je da definiše koji će interfejs biti dostupan brokeru i pod kakvim uslovima(odobrenjima).
* Broker ili registar servisa predstavlja komponentu preko koje se vrši interakcija sa servisima. Javni broker je dostupan svuda i svima , dok interni broker vrši komunikaciju između servisa. Ova dva brokera mogu biti spojeni u jedan broker opšteg tipa.
* Potraživač ili korisnik servisa, predstavlja komponentu koja od brokera zahteva pristup određenom servisu.

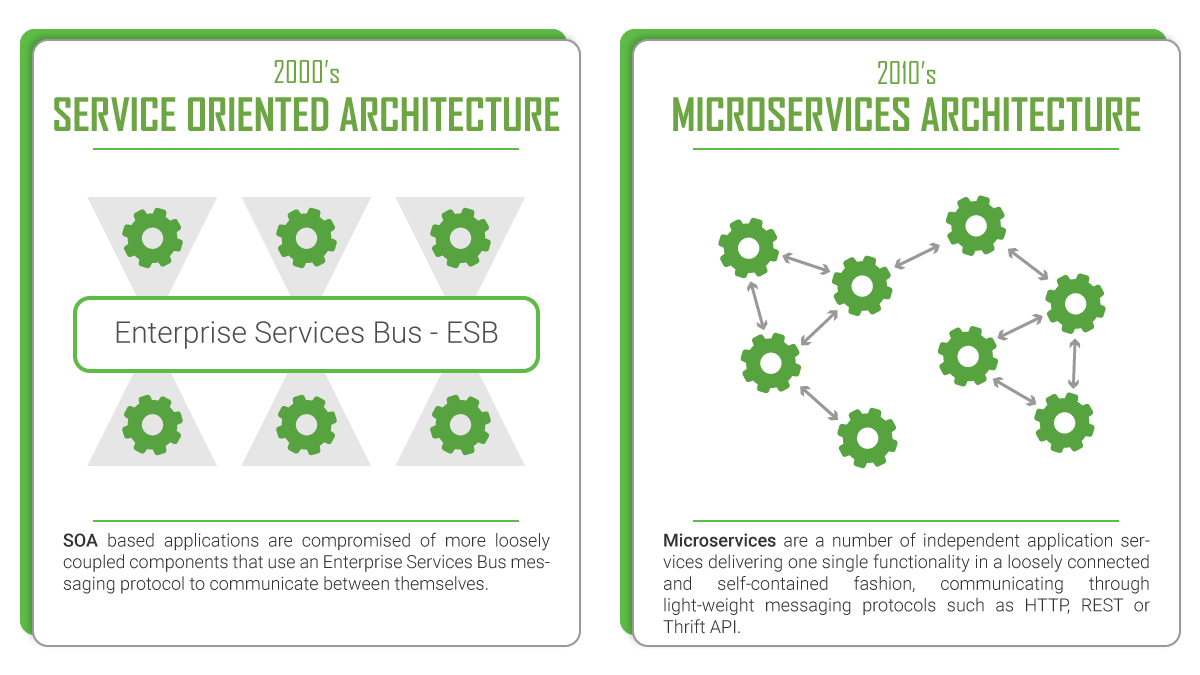
## Servisi

Servisi su mikroaplikacije koje obavljaju specifične zadatke. Njihova interna organizacija može biti raznovrsna, od jednostavnih funkcija do kompleksnih podprograma koji se sastoje od više slojeva. Takođe, servisi se ponašaju kao crne kutije sa predefinisanim interfejsima što nam pomaže da ne moramo previše razmišljati o samoj strukturi servisa.

Prvi servisi su nastali nezavisno od arhitekture, to su takozvani web servisi. U pitanju su web aplikacije koje umesto korisničkog interfejsa vraćaju podatke u vidu XML ili JSON stringova.



1. Mikro servis u poređenju sa web servisom



1. SOA arhitektura u odnosu na mikroservisnu arhitekturu

Slika 5. Web servisi u odnosu na mikro servise (Despodovski, 2021)

Sama tehnologija u kojoj je servis realizovan je potpuno nebitna dokle god ima jasno definisan interfejs i brzo i kvalitetno izvršava svoje zadatke.

Servisi imaju mnogih prednosti zbog svoje atomičnosti. To znači da jedan servis u sebi sadrži sve što je neophodno da bi se zadaci izvršavali. Jedan servis je moguće koristiti od strane više različitih aplikacija istovremeno čime se smanjuje vreme razvoja tih aplikacija, ali se smanjuje i vreme potrebno za održavanje jer je potrebno održavati samo jedan servis.

Pošto su servisi pravljeni da prate poslovnu logiku i sami servisi mogu predstavljati uslugu koju firma nudi svojim klijentima što nam omogućava da napravimo aplikaciju u kojoj se dodatne mogućnosti dodatno naplaćuju. (Bell, 2008)

## Komunikacija, protokoli i hosting

Komunikacija između servisa se izvršava preko preko mreže koristeći http protokol. Http je skraćenica za *Hypertext Transfer Protocol*, dakle to je protokol za razmenu jasno strukturisanih poruka. (Leach и остали, 1999)

Najčešća komunikaciona arhitektura je REST ili *Representational state transfer*. REST arhitektura koristi podset http protokola i detaljno je definisana. Pod detaljnijim definisanjem se misli na dodatno definisanje strukture samog tela poruka, puna upotreba http metoda (GET, POST, DELETE...) kao i odgovora u skladu sa promenom stanja što je najčešće prikazano http statusnom porukom. (Lange, 2016)

REST arhitektura je dobila na popularnosti zbog svoje jednostavnosti i brzine. Prethodna, SOAP arhitektura je deskriptivnija i koristi XML format za razmenu poruka, što je u suštini i ključna razlika između SOAP i REST arhitektura. SOAP pristup sa slanjem poruka u XML formatu je previše deskriptivan i time zauzima previše resursa, pogotovo kad se komunikacija vrši preko ograničenih mreža poput mobilne. Iz tog razloga je REST uzla maha u poslednje dve decenije. Iako REST predstavlja veliki napredak u odnosu na SOAP i dalje postoje problemi vezani za REST. Najveća dva problema su Over i Under fetching. Overfetching znači da se od servera na zahtev dobija više informacije nego što je potrebno, a Underfetching znači da je potrebno napraviti više zahteva da bi se prikazala celokupna slika.

Kao rešenje za prethodna dva probelma je 2015 napravljen GraphQL koji kroz jedan zahtev dobija sve potrebne informacije, a opet ne koristi komplikovane strukture poprut XML-a. Kao i svaki protokol i GraphQL ima svojih mana, prevashodno što ne postoji način da se naglasi verzija. (DA-14, 2018)

Komunikacija ne mora da bude preko http protokola pogotovo kad je potrebna uzajamna asinhrona komunikacija između servera i klijenta. Ako je potrebna ovakva komunikacija može se koristiti *web-socket* protokol koji predstavlja *full duplex* komunikacioni kanal između dve strane. *Web soket* protokol je standardizovan 2011e i podržan je od strane većine novijih *web* pregledača.

Prednost ovog protokola je u tome što se klijetu mogu slati informacije iako ih nije tražio. Ova mogućnost je od ključnog značaja za *chat* aplikacije/servise i vrlo korisna za *real-time* obaveštenja. *Web soket* protokol se može koristiti za komunikaciju između klijenta i servisa kao i između samih servisa, isto kao i http. (Lubbers & Greco, 2020)

Pošto je komunikacija između servisa standardizovana i koristi standardne protokole, informacije se prenose preko mreže između nezavisnih mašina. Ova karakteristika omogućava veliku fleksibilnost u hostingu tako da statički elementi sistema poput frontenda mogu biti hostovani na CDN (*content delivery network*) koji čuvaju kopije blizu klijenata tako da se smanjuje inicijalno vreme učitavanja aplikacije. Logički elementi se mogu izvršavati na zasebnim serverima ili kao servisi na *serverless* sistemima. Odabir same organizacije hostinga i organizacije lokacije servisa su kompleksna pitanja koja dosta zavise od same aplikacije i potražnje određenih servisa aplikacije. (Laws & Goodfellow, 2018)

Servisno orijentisana arhitektura se uglavnom koristi za kompleksne sisteme (aplikacije), ali se može koristiti delom u manjim projektima gde se određeni servisi samo inkorporiraju spram potreba.

# Algoritmi mašinskog učenja u analizi finansijskih pokazatelja

Sama definicija veštačke inteligencije je nepostojeća iako je sam termin definisan još 1956 godine od strane Džona Mekarntija tako da se može reći da je veštačka inteligencija disciplina računarstva koja se bavi razvojem softvera koji bi mogli biti okarakterisani kao inteligentni. (Russell & Norvig, 2009)

Primena sistema sa karakteristikama veštačke inteligencije je sve popularnija poslednjih godina jer tehnologije dozvoljava izvršavanje tako kompleksnih algoritama. Najčešće se upotrebljava poddisciplina mašinsko učenje. Mašinsko učenje se bavi konstrukcijom i primenom algoritama koji su u stanju da se adaptiraju i uče na osnovu svog dosadašnjeg iskustva.

Postoje tri pristupa obuke algoritama mašinskog učenja:

- Nadgledano učenje.

- Nenadgledano učenje

- Ojačavajuće učenje (*Reinforcement learning*)

Proces učenja kod nadgledanog pristupa je takav da se algoritmu daje set podataka sa željenim odgovorima, na osnovu takvog dataseta algoritam uči kakav odgovor treba da da. Ovakav pristup učenju je koristan za predikcije i zato je korišćen za izradu aplikacije ovog rada.

Nenadgledano učenje je učenje nad podacima bez datog tačnog odgovora. Ovakvim pristupom kao rezultat se dobija struktura međusobnih odnosa podataka. Pristup nenadgledanog učenja je koristan za bolje razumevanje samih podataka.

Ojačavajuće učenje je učenje na osnovu nagrada i kazni. Alogritmu su pruženi stimulusi na koje on treba da odgovori, za svaki odgovor dobija nagradne poene ili mu se poeni oduzimaju. Cilj algoritma je da skupi što veći broj poena. Ovakva vrsta učenja se koristi u sistemima samohodnih vozila, za veštačke inteligencije koje igraju različite igre i za modeliranje i simuliranje različitih događaja i odnosa.

Kao što je već napomenuto nadgledano mašinsko učenje je odlično za različite predikcije i postoje dva statistička modela koja se koriste za predikcije, klasifikacija i regresija. Klasifikacija je statistički problem dodeljivanja novih podataka postojećim grupama, ovakav pristup je koristan nad nenumeričkim podacima i predikciji klase na osnovu inputa. Regresioni modeli određuju međusobne odnose ulaznih parametara i na osnovu novih ulaznih parametara daju predikciju rezultata. Ovakvi modeli za razliku od klasifikacionih modela koji daju isključivo cele brojeve, kao rezultate daju racionalne brojeve što je odlično za predikcije cena i generalno finansije. (Bishop, 2006)

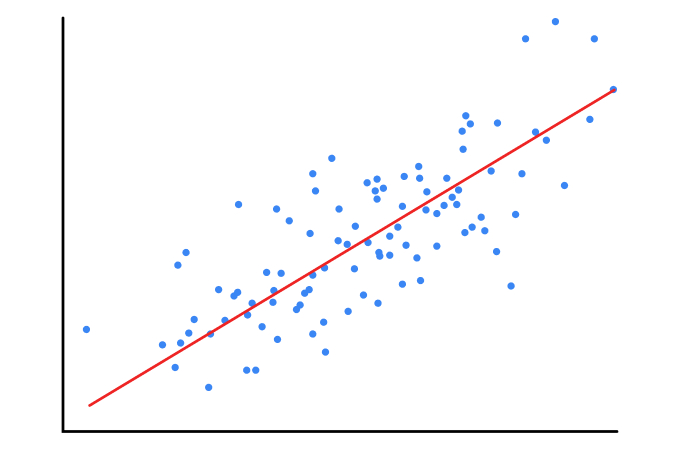
Za samu predikciju cene zlata u aplikaciji je korišćen *decision tree* regresioni model jer je on u prethodnim istraživanjima davao najbolje rezultate. Aplikacija takođe raspolaže i sa drugim regresivnim modelima i to sa:

- Linearnim regresorom

- *K nearest neighbor*

- *Support vector machine*

Linearna regresija pokušava da najtačnije moguće prikaže vrednosti zavisne promenljive u odnosu na nezavisnu promenljivu linearnom funkcijom. Čest i uobičajen način predikcije vrednosti finansijskih instrumenata, pogotovu ukoliko su vrednosti inertne.



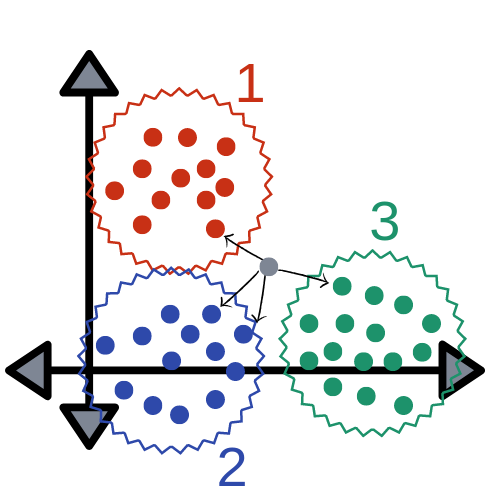
Slika 6. Primer linearne regresije (Thiebaut, 2019)

Drvo odlučivanja je jedan od najboljih i najčešće korišćenih algoritama za klasifikaciju jer pored toga što nudi veliku tačnost predviđanja i jasnoću, lako mapira i nelinearne veze. Lako rešava probleme i regresije i klasifikacije.



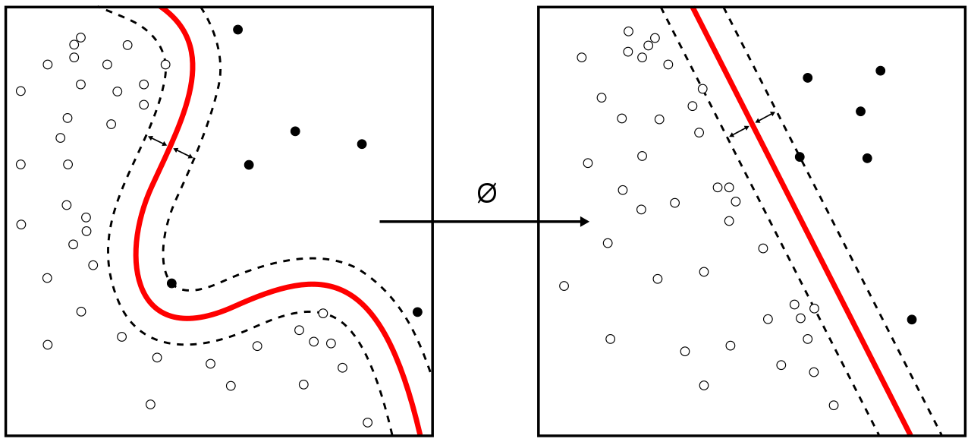
Slika 7. Primer algoritma “drvo odlučivanja” (Jacobi, 2017)

*K-nearest neighbor* (kNN) se može koristiti za probleme klasifikacije i regresije. Model je koji klasifikuje tačke podataka na osnovu tačaka koje su mu najsličnije. Koristi podatke testa da bi „učio nagađanje“ o tome na šta treba svrstati neklasifikovanu tačku (Schott, 2020)



Slika 8. Primer modela kNN (Schott, 2019)

*Support vector machine* (SVM) je model koji podseća na napredniju verziju linearne regresije. Ovaj model podatke predstavlja kao tačke u prostoru koje on klasifikuje u dve kategorije između kojih postoji razmak. Da model ne bi bio model linearne regresije koristi se takozvani kernel trik koji podrazumeva posmatranje pojedinačnih zona, a ne seta u celosti.



Slika 9. Primer SVM modela (Zirguezi & Alisneaky, 2011)

Izvršavanjem nekoliko iteracija ovih modela i poređenjem rezultata svim iteracija, odlučeno je da najtačnije rezultate daje model *decision tree* i to u slučaju kada modele trenirano na podacima od sedam radnih dana berze i na taj način izbegavamo nedostajuće vikend podatke.

U tabeli u nastavku se vide detaljne informacije tačnosti i uspešnosti modela i jasno je da model drveta odlučivanja daje najbolje rećultate po svim kategorijama stoga je rešeno da samo taj model bude uključen u frontend prikaz predikcije cene zlata.

Tabela 1. Komparacija tačnosti modela

|  |  |  |
| --- | --- | --- |
|  | **Linearna regresija** | **Drvo odlučivanja** |
| explained\_variance\_score | -0.6391 | 0.9616 |
| max\_error | 143.1799 | 10.7200 |
| mean\_absolute\_error | 6.6431 | 1.8054 |
| mean\_squared\_error | 274.9183 | 6.4203 |
| mean\_squared\_log\_error | 0.1156 | 0.0002 |
| median\_absolute\_error | 2.8622 | 1.3300 |
| r2\_score | **-0.6412** | **0.9616** |
|  | **K-nearest neighbor** | **Supp. vector machine** |
| explained\_variance\_score | 0.9386 | 0.9307 |
| max\_error | 13.4921 | 11.3136 |
| mean\_absolute\_error | 2.4035 | 2.5986 |
| mean\_squared\_error | 10.7402 | 12.1284 |
| mean\_squared\_log\_error | 0.0003 | 0.0004 |
| median\_absolute\_error | 1.8060 | 2.0386 |
| r2\_score | 0.9358 | 0.9275 |

# Analiza naučnih radova i modela za predikciju cene zlata

Mašinsko učenje i predikcije su vrlo popularne tematike koje nalaze sve više oblasti primene. Iako imaju veliku primenjivost, mali je broj ljudi u interesnim oblastima kojima je jasno i koji imaju umeća kako da primene metode mašinskog učenja. Ovaj problem je naročito izražen u ekonomskim disciplinama. Uglavnom ljudi koji se bave ekonomijom slabo razumeju programiranje, a mašinsko učenje još manje. Ovaj razlog je primarni pokretač za pravljenje aplikacije ovog master rada.

Aplikacija je napravljena tako da omogući lakši pristup mašinskom učenju osobama koje nemaju umeća da sami naprave predikcije. Takođe, pokazuje primenjivost na konkretnom problemu predikcije cene zlata što je značajan podatak za osobe koje se bave investiranjem poput brokera i investicionih savetnika.

Najbitniji aspekt prilikom izrade aplikacije je bila višestruka primena servisa. Svaki servis je napravljen ili da ima univerzalnu funkciju poput brokera i modelatora ili da se lako može proširiti novim mogućnostima poput prediktora. Na ovaj način napravljena je aplikacija da uz odgovarajuć zahtev svako može da dođe do rezultata predikcije bez obzira na poreklo i sadržaj podataka i da, ako se javi potreba, se proširi predikcija na druge finansijske derivate,a ne samo zlato.

Iz želje da se napravi aplikacija koja se može lako menjati spram potreba klijenata i koja može da pruži uslugu koja će biti primenjiva na raznovrsnim podacima, odlučeno je da se koristi servisno orijentisana arhitektura jer se postulati poklapaju sa planom aplikacije.

## Predikcija cene zlata

Zlato kao plemeniti metal je oduvek imalo vrednost koju je bilo moguće zameniti za robu ili uslugu. Preferirano i likvidno je zato sto je relativno lako prenosivo, ne gubi svoje karkteristike vremenom (sjaj, masu, oblik), nije široko dostupno jer je potrebno dosta truda, a i sreće da bi se u prirodi našle i najmanje količine. Zahvaljujući svojim karakteristikama našlo je primenu u prvom novcu u vidu kovanica. Kasnije je uveden novac u papiru koji je garantovao donosiocu novčanice da će dobiti količinu zlata istaknutu na istoj.

Shodno zlatnim rezervama, države su štampale novčanice. Uzimajući u obzir da su šanse da sve novčanice budu zamenjene za zlato u isto vreme izuzetno male, drzave su često štampale više novca nego što imaju rezervi. To nije predstavljalo problem jer je odnos bio uglavnom konstantan što je i omogućavalo stabilost valuta, tj. stabilnost međusobnih odnosa između valuta. Problem počinje sa Prvim Svetskim ratom gde su države taj odnos ozbiljno poremetile štampajuci sve veće količne novca stvarajući inflaciju valuta i time finansirajući rat. Posle rata zemlje koje su profitirale od rata su se vratile na stari sistem i odnos količine novca i zlata ali na globalnom tržistu se sve manje trgovalo u zlatu, koje je u to doba više korišteno za ratne reparacije. (Warwick-Ching, 1993)

Drugi svetski rat je doveo do jos veće inflacije cena i posle rata je većina valuta prešla na Bretonvudski sporazum gde se više razmena valuta nije vršila kroz njihov odnos prema zlatu, nego prema dolaru, što je u to vreme bilo skoro isto jer je dolar imao konstantu vrednost prema zlatu. Ovom sistemu je došao kraj Niksonovim šokom 1971. kad je dolar postao Fiat valuta i njegova vrednost je prestala da bude fiksna u odnosu na zlato, nego je pušteno da tržiste diktira vrednost. Od tog trenutka predikcija cene zlata postaje zanimljiva tematika. Iako je većina valuta u svetu fiat, centralne banke i dalje drže zlatne rezerve radi likvidnosti. Takođe, u kriznim situacijama kao što je situacija sa pandemijom COVID-19 virusa ljudi pokazuju da imaju veće poverenje u zlato kao čuvara vrednosti nego u valute.(Lewis, 1976)



Slika 10. istorijski prikaz cene zlata u dolarima. Crvena linija predstavlja Niksonov šok avgusta 1971 (macrotrends, 2021)

## Analiza naučnih radova

Potreba za predikcijom cene zlata je nastala od kad je američki dolar postao fiat moneta.

Za predikcije velike tačnosti je bilo dovoljno raditi lineranu regresiju jer je samo tržiste bilo vrlo stabilno. Ali usled pandemije virusa COVID-19 ljudi su pokazali da je poverenje i dalje u zlatu, što se može videti naglim skokom cene usled povećane potražnje. Pošto je došlo do nestabilnosti na tržistu, modeli linearne regresije više ne prate trend dovoljno dobro, a u vreme nestabilnosti dobar model je bitniji nego ikad jer nestabilnost predstavlja šansu za zaradu trgovanjem. Usled pomenute potrebe različiti istraživači su isprobavali koji modeli mašinskog učenja najbolje vrše predikciju i ustanovili su da ARIMA model daje najbolje rezultate, ali primenom modela poput Decision tree na kotrljajući prozor se dobijaju još bolji rezultati jer se smanjenjem dataseta smanjuje inertnost modela i preciznost modela dostiže i do 96% u tačnosti predikcije vrednosti za sutrašnji dan. Primer takvog modela je obrađen u master radu Jovane Stokanović Šević, a dodatna poboljšanja su možda moguća ako se podaci o broju novozaraženih i umrlih o koroni dodatno klasifikuju po geografskim regijama, jer je pretpostavka da situacija pandemije u razlicitim delovima sveta, za početak po kontinentima, drugačije utiče na kretanje cene zlata.(Stokanović Šević, 2020)

Tabela 2. Prikaz prethodnih istraživanja zlata kao finansijskog sredstva i modela mašinskog učenja.

|  |  |  |  |
| --- | --- | --- | --- |
| **Naziv autora** | **Naziv rada** | **Opis istraživanja** | **Rezultat istraživanja** |
| (Azzutti, A., 2016) | Forecasting Gold Price  A Comparative Study | Evaluacija različitih tehnika za predviđanje cene zlata | Iako različiti modeli daju dobre rezultate ARIMA model daje najpreciznije rezultate |
| (Brownlee, 2016) | How To Backtest Machine Learning Models for Time Series Forecasting | Primer kako se može raditi sa vremenski zavisnim podacima | Hronoloski sled podataka, prilikom predikcija vrednosti na osnovu istorijskih podataka je vrlo bitan |
| (Brownlee, 2017) | How to Create an ARIMA Model for Time Series Forecasting in Python | Prikaz ARIMA modela i način kretanja kroz vremenski zavisan dataset | Pokazano je da je u redu koristiti klizni dataset za predikciju. |
| (Riazuddin, 2020) | Predicting Gold Prices Using Machine Learning | Komparacija modela na istom datasetu za predikciju dobiti za 22 dana | Iz ovog rada se može zaključiti da je veoma bitan odabir dataseta i da parametri nevezani za zlato mogu pomoći u boljoj predikciji |
| (Jansen, 2020) | MACHINE LEARNING FOR ALGORITHM TRADING : Master as a PRO applied artificial intelligence and Python for predict systematic strategies for options and stocks. Learn data-driven finance using keras | Knjiga o primeni mašinskog učenja u trgovanju finansijskim istrumentima |  |
| **Naziv autora** | **Naziv rada** | **Opis istraživanja** | **Rezultat istraživanja** |
| (Warwick-Ching, 1993) | The International Gold Trade | Knjiga o zlatu kao finansijskom instrumentu. |  |
| (Yousef & Shehadeh, 2020) | The Impact of COVID-19 on Gold Price Volatility | Rad o korelaciji između skoka cene zlata i povećanja broja obolelih od virusa | Rad zaključuje da postoji korelacija I da utiče na finansijski sistem |
| (Bingol и остали, 2020) | Gold price prediction in times of financial and geopolitical uncertainty: A machine learning approach | Nalaženje veze između cene zlata i varijabli koje bi mogle predstavljati finansijsku i geopolitičku krizu | ARIMA model najbolje predviđa buduće cene zlata na osnovu varijabli nestabilnosti |
| (Weng и остали, 2020) | Gold price forecasting research based on an improved online extreme learning machine algorithm | Predikcija cene zlata različitim modelima | GA-ROSELM model ima najpreciznije rezultate |
| (Yerra, 2019) | Predicting Tomorrows Gold Price | Predikcija cene zlata je složen problem s obzirom na to da kolebanje cena zlata nije u potpunosti zasnovano na ponudi i potražnji, već zavisi i od mnoštva geopolitičkih i finansijskih faktora. | Kada se vrše predikcije treba voditi računa o odabiru modela, ali i veličini dataseta i hronološkom rasporedu podataka |
| (Altig и остали, 2020) | Economic uncertainty in the wake of the COVID-19 pandemic | Provera parametara nesigurnosti za vreme Covid pandemije | Povećana nesigurnost preduzeća i potrošače generalno čini opreznim za ulaganja |

Na osnovu brojnih prethodnih istaživanja vezanih za predikciju cena finansijskih derivata može se zaključiti da postoji tražnja, a samim tim i potreba za takvim predikcijama. Primetno je u istraživanjima da se ispituje primena modela i uticaj dodatnih informacija.

Poređenje modela u istraživanjima je uglavnom nad celim datasetom koji je podeljen u test i train poddataset u odnosu 80%-20% sa nasumičnom raspodelom podataka između dva poddataseta, takvo poređenje govori samo o kvalitetu algoritama, ali ne uzima u obzir lokalne trendove varijacije vrednosti u datasetu, tako da se može reći da su ovakvi modeli pretrenirani i inertni. Primena pomenutih modela iz prethodnih istraživanja ima smisla za dugogodišnje predikcije. Problem kontinuiteta podataka i predikcije je pokušan da se reši ARIMA modelom, ali kao i svi ostali modeli, ARIMA model uzima u obzir celokupan dataset, odnosno train deo dataseta. Iako daje najbolje rezultate, značajno povećanje preciznosti se može postići smanjenjem train i test set. Da bi se manji i train i test setovi primenili na celokupnom datasetu potrebno je proći *frameom* preko dataseta i svaki *frame* tretirati kao inicijalni dataset. Primena kliznog prozora kao manu ima to što je kalkulacijski intenzivnija od standardnog modeliranja, ali pošto je u pitanju primena funkcije nad nizom paralelizacija procesa je lako primenljiva.

Prethodna istraživanja koja su se uglavnom bavila uticajem ulaznih podataka su ispitivala da li postoji uticaj dodatnih podataka, a ne koliki je uticaj tih dodatnih podataka. Nedostatak informacija o veličini uticaja može lako dovesti do preopterećenja nepotrebnim informacijama u slučaju da uticaj postoji, ali je suviše mali da bi bio značajan. Ovakav pristup ima smisla kad su dopunske informacije nepotpune i nepouzdanog kontinuiteta u budućnosti, što se i pokazalo tačnim jer je izvor dodatnih podataka za COVID-19 prestao sa unosom svežih podataka u januaru 2021 godine.

# Razvoj aplikacije za previđanje cene zlata primenom mašinskog učenja

## Osnovne funkcionalnosti aplikacije

Ova aplikacija ima dve funckionalnosti. Primarna funkcionalnost je predikcija cene zlata za sutrašnji dan, sekundarna funkcionalnost je primena nadgledanom mašinskog učenja nad željenim podatcime preko API-ja aplikacije.

Sama predikcija cene zlata je dostupna na mnogim internet sajtovima, ali sa njima priča isključivo o pokazateljima koji potkrepljuju njihove predikcije, i nigde se ne pominju algoritmi koji su korišćeni za predikciju. Iz tog aspekta ova aplikacija, iako za sad nema velik predikcioni period, trenutno jedan dan, ima dostupne i objašnjene pristupe unutar ovog master rada, a i na samom repozitorijumu aplikacije gde je i kod dostupan. Ovakva transparentnos je štetna po špekulativnu industriju trgovanja jer precizan, jasan i pouzdan model ide u korist stabilizacije tržišta i smanjenja prostora za neočekivane skokove vrednosti.

Broj online aplikacije koje pružaju mogućnost mašinskog učenja komunikacijom preko API-ja je velik. Neke od aplikacija imaju već predodređene modele koji se mogu podešavati i trenirati, dok se neke ponašaju kao platforme na kojima je moguće izvršavati lični kod. Funkcionalnost aplikacije ovog rada je jedinstvena u tome sto ima predefinisane modele sa dodatkom kliznog prozora nad datasetom. Ovaj efekat je moguće izvesti i na već postojećim servisima, ali bi se za svaki frejm data seta morao slati nov zahtev. Takvo intenzivno slanje zahteva je daleko od optimalnog, čak se i kod nekih servisa može smatrati napadom, tako da bi onda servis postao nedostupan.

## Korišćene tehnologije

S obzirom da se svaki servis ponaša kao crna kutija može se smatrati da je svaki servis nezavisna aplikacija za sebe. Zbog raznovrsnosti zadataka koje servisi izvršavaju raznovrsne su i tehnologije kojie su primenjene.

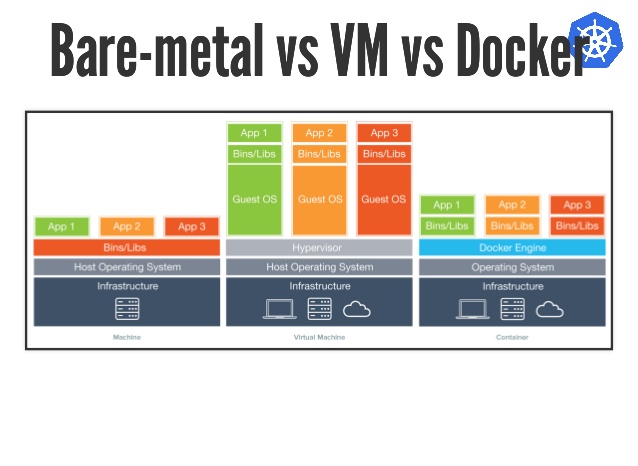
Za lokalno pokretanje korišćen je Docker, za servise su korišćeni programski jezici Python i JavaScript. Frontend servis je realizovan uz pomoć React biblioteke. Broker je Node.js sa Express frejmvorkom i socekt.io dodatkom za web-sockete i got paket za pozive ka drugim servisima. Prediktor i Modelator su napravljeni u Flask frejmworku u python programskom jeziku i oba koriste pandas paket za manipulaciju podacima. Prediktor koristi dodatni paket za zakazivenje egzekucije koda, apscheduler, i paket za pravljenje zahteva ka drugim servisima, requests. Za modelator servis karakterističan je scikit-learn koji je moćan alat za mašinsko učenje. Dodatno je potrebno napomenuti da je za lokalni deployment potrebno omogućiti CORS (objašnjeno u okviru 5.2.6. poglavlja ovog rada).

### Docker

Za pokretanje aplikacije na lokalu je korišćen Docker i Docker-Compose. Alternativna rešenja su da se servisi pokreću direktno na računaru na kom se radi ili unutar jedne ili više virtualnih mašina.

Tabela 3. Komparacija rešenja za postavku lokalne razvojne instance. (Kominos et al, 2017)

|  |  |  |  |
| --- | --- | --- | --- |
| **Karakteristika** | **Računar** | **Virtualna mašina** | **Docker** |
| Postavka lokalne razvojne instance | Vrlo komplikovano | Inicijalno komplikovano | Jednostavno |
| Kompatibilost sa različitim platformama | Svaki servis se mora podešavati od početka za svaku postavku | Kompatibilno rešenje za sve operativne sisteme, jer se postavlja dodatni operativni sistem unutar virtualne mašine. | Kompatibilno sa UNIX-like operativnim sistemima. Jedini bitan ne unix-like operativni sistem je Windows, ali uz pomoć WSL omogućen je pristup celokupnom Linux kernelu |
| Performanse  Brzina egzekucije | Servisi se izvršavaju najbrže moguće.. | Najsporije rešenje zbog dodatnih slojeva. | Malo sporije rešenje od pokretanja direktno na računaru. |
| Performanse  Bezbednost | Najnebezbedniji pristup. dodatne biblioteke imaju pun pristup računaru, ograničavajući mehanizmi se moraju dodatno implementirati | Najbezbednije rešenje jer je celokupna egzekucija izolovana u virtualnu mašinu. | Bezbednost zavisi od podešavanja kontejnera i dozvoljenoj interakciji sa ostatkom sistema. |
| Performase  Zauzimanje memorije | Najmanje jer svi servisi imaju pristup resursima, tako da ne postoji dupliranje resursa. | najviše, a pogotovo ako se za svki servis koristi zasebna virtualna mašina. | Skoro kao direktno na računaru, zavisi od optimizacije kontejnera, jer je moguće da određeni kontejneri unutar sebe sadrže iste resurse. |
| Deployment | Slično kao i na lokalu, Vrlo komplikovano pogotovo ako se produkciona platforma razlikuje od razvojne. | Vrlo jednostavno dokle god je produkciona platforma ista kao virtualna mašina. | Vrlo raznovrsan. MOguća deployment rešenja variraju od podešavanja produkcione platforme od nule, preko pokretanja kontejnera na produkcionoj mašini do korišćenja servisa koji pokreću samo kontejnere u skaldu sa potražnjom servisa. |



Slika 11. Komparacija platformnih rešenja (Kominos *et al*, 2017)

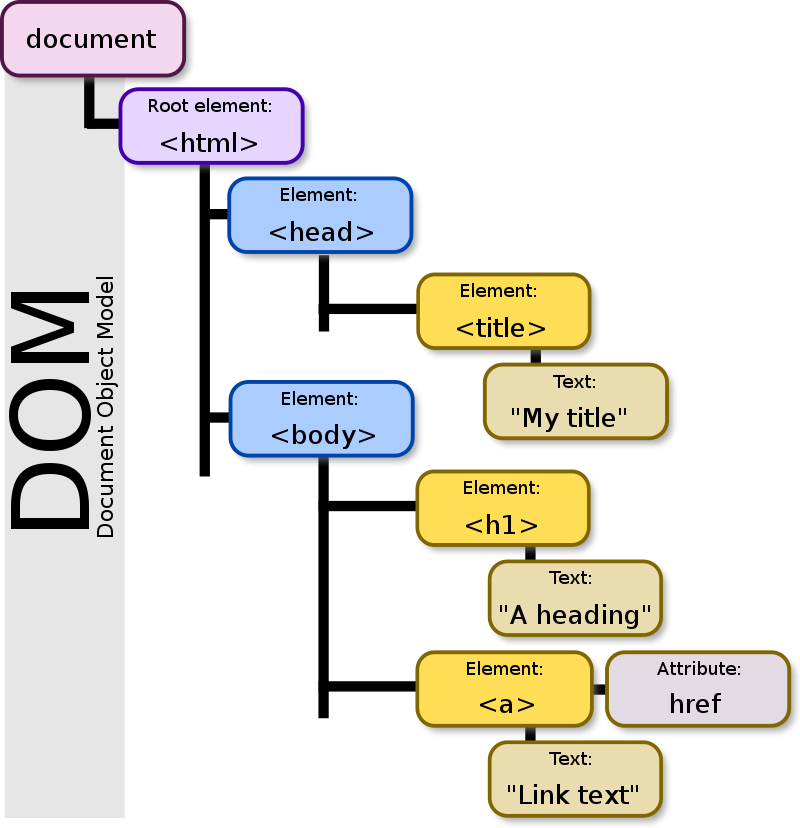
Docker i DockerCompose su odabrani jer je to jedan od najlakših načina da se pokrene aplikacija na lokalu, a pogotovo kad je aplikacija multi servisna. Docker predstavlja set alata za pokretanje softvera zapakovanih u kontejnere koji se pokreću u virtualnim okruženjima koja su u nivou operativnog sisitema. Pošto kontejneri koriste servise kernela, nema potrebe za dodatnim alociranjem resursa, stoga je ovaj pristup lokalnog deploymenta značajno lakši i efikasniji od standarnih rešenja sa virtualnim mašinama poput vagranta.

Svaki servis je zapakovan u svoj kontejner tako da je moguće raditi na pojedinačnim servisima. Pošto je u pitanju više servisni-kontejnerski sistem, za lokalni depoloj je korišćen docker-compose alat. Docker-compose za instrukcije deploya koristi YAML fajl u kome su definisani koji su kontejneri potrebni, da li je potrebno predpakovanje kontejnera pre pokretanja, koji su međusobni odnosi kontejnera i kako komuniciraju sa spoljnim svetom. (*Docker Frequently Asked Questions (FAQ)*, 2021)

Još jedan od razloga zasto je Docker odabran je zato što radi na svim popularnim operativnim sistemima, tako da nema poblema u kompatibilnosti i takozvanog „Kod mene je radilo“ problema. (*How Docker containers solve No more works on my machine*, 2020)

### React

Klijentska prezentacija se sastoji od tri elementa, HTML Strukture, CSS Stilova i Javascript logike. HTML ili *Hyper text marcup language* predstavlja varijaciju XML-a i njime je definisana struktura web stranice. HTML predstavlja kostur DOM-a (Document Object Model) na koji se potom dodaju različite osobine. HTML zajdno sa CSS-om (Cascade Style Sheats) predstavljaju osnovne fajlove interneta, početkom devedesetih su u svrhu interneta i stvoreni u CERN-u. Da bi internet bio upotrebljiv bio je potreban pregledač, prvi pregledaj je bio Mosaic 1993 godine i prikazivao je samo statične stranice. Odmah je bilo primećeno da bi interakcija sa korisnicima bila vrlo poželjna zato 1995 Netscape pregledač uvodi novi skriptni jezik JavaScript koji omoguća iterakciju klijenta i stranice.



Slika 12. Prikaz DOM-a

Virutalni DOM je js reprezentacija već pomenutog DOM-a. Ovakva implementacija ima performansnih benefita u odnosu na realni DOM jer prilikom izmena elemenata nište ne mora da renderuje na ekranu ako ne mora, čak i izmene izgleda samih elemenata zahtevaju renderovanje samo tih elemenata, a ne cele stranice. (Rauschmayer, 2014)

React je frontend biblioteka za manipulaciju virtualnog DOM-a. Veoma je pogodna za pravljenje *single-page* frontend aplikacija. Pod *single page* se smatra da kad se jednom stranica učita sav sadržaj stranice se programski menja iz JavaSkripta. Ovakav pristup je odličan za kompleksne korisničke interfejse. Početno učitavanje stranice zavisi od količine koda, ali pošto je moguće koristiti isti element više puta pogotovo kod stranica sa repetitivnim sadržajem je brže učitati šablon i primeniti ga kroz petlju umesto da se celokupan sadržaj stranice šalje sa servera. Mana ovakvog pristupa je u tome što je u pitanju jedna stranica, što otežava vidljivost u pretragama. Ovaj problem *single page* aplikacije web pretraživači počinju da rešavaju jer je sve više i više single page aplikacija

Popularnost Reacta je zasnovana na tome što je vrlo lagana biblioteka u smislu veličine i brzine same biblioteke, za razliku od drugih popularnih frejmworka poput Angulara i Vue.js. Osim veličine i brzine bitna je i stabilnost biblioteke, o čemu govori broj korisnika među kojima su i velike firme poput *Facebook*-a koji je i kreirao i održava biblioteku. (*React – A JavaScript Library for Building User Interfaces*, 2021)

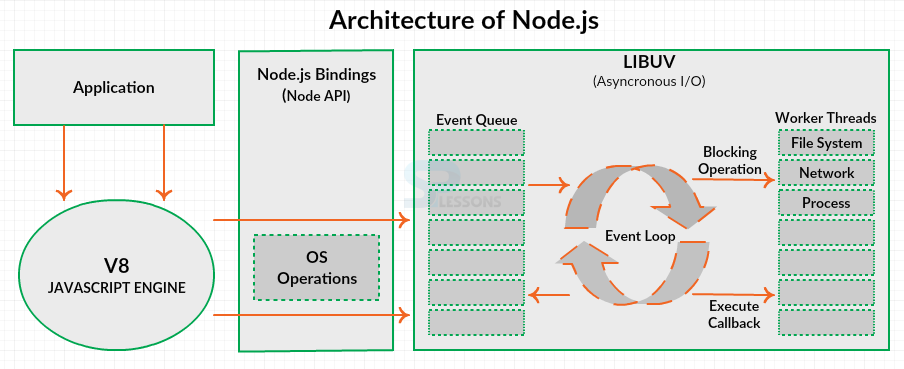
React takođe ima pandan za kreiranje *cross platform* aplikacija, *React-native*, tako da je moguće uz manje promene koda web aplikaciju pretvoriti u mobile aplikaciju za Android i/ili iOS. (*React Native · Learn Once, Write Anywhere*, 2021)

### Node.js i Express

Postojeće implementacije interpretera za JavaScript su 2008 bile unapređene verzije inicijalnih interpretera od pre 12 godina. S obzirom da je u pitanju skriptni jezik, egzekucija koda je bila vrlo spora jer se kod prevodio u mašinski kod prilikom svake operacije. Google za potrebe svojih servisa poput *GoogleMaps* je razvio svoj pregledač *Chrome* sa V8 interpreterom. V8 interpreter je u odnosu na konkurenciju bio stabilniji i brži. Stabilnost je postignuta *sandboxovanjem* procesa, svaki tab je bio proces za sebe, tako da ako dođe do greške na jednom tabu, ostali tabovi uglavnom ne bi bili pogođeni. Ovakva implementaicja kao manu imala povećano zauzimanje ram memorije, što je i dalje jedan od najvećih problema. Brzina egzekucije je u poređenu sa konkurencijom 2008 bila neverovatna, u poređenju sa I*nternet Explorerom Chrome* je bio 8 do 9 puta brži. Ovakve performanse su postignute sa JIT(*Just in time)* kompilacijom gde je celokupan kod pretvoren u mašinski kod odmah po učitavanju, a ne kao što je do tad bio slučaj sa svakim pokretanjem događaja.

Godinu dana kasnije V8 nalazi i upotrebu sa serverske strane u okviru Node.js okruženja. Node.js nije prvo serversko rešenje za javaskript, ali sa novim interpreterom upotreba javaskripta na serveru postaje primamljivo rešenje. Autor Node.js-a Rayan Dahl je kombinovao V8 interpreter, sa eventloop-om i sistemskim I/O apijem i time omogućio asinhrono programiranje vođeno događajima u javascriptu na serveru. (Teixeira, 2012)

Sa obzirom da Node ima pristup sistemskom I/O API-ju moguće je direktno komunicirati preko http protokola. Pošto je u pitanju osnovni sistemski API sama iterakcija je malo komplikovanija za kodiranje, zato postoji veliki broj biblioteka koje olakšavaju interakciju sa API-jem praveći pojednostavljenije interfejse. Neke od takvih biblioteka su Express, Socket.io i Got.



Slika13. Arhitektura Node.js

Express je frejmwork za Node.js koji je specijalizovan za kreiranje web aplikacija, ali je poznat kao server frejmwork za Node.js. Ovakav opis se savršeno uklapa u opis Brokera. Prednost Express-a za Broker servis je u tome što se zahtevi obrađuju asinhrono tako da ne dolazi do zagušenja prilikom čekanja na odgovor, a pritom se sve izvršava u jednoj niti, tako da ne dolazi do zagušenja na procesoru prilikom preskakanja sa niti na nit. (*Express - Node.Js Web Application Framework*, 2021)

Generalna mana Node.js-a, a samim tim i Expressa, je da je izuzetno spor na aritmetičkim operacijama i interakcijama sa relacionim bazama. Međutim, pošto Broker servis u suštini samo preusmerava zahteve na ostale servise ova mana ne predstavlja problem. (Kayode, 2020)

### Socket.io

Socet.io je biblioteka za Node.js koja omogućava dvosmernu komunikaciju između klijenta i servera u realnom vremenu upotrebom WebSocket protokola. Kao i sam Node,js biblioteka je asinhrona i vođena događajima. Primenom ovakve konekcije je moguće da se obveštenja i informacije šalju klijentu tu renutka kad su informacije dostupne. Ovime se realizuju notifikacije u realnom vremenu, a može se i realizovati čet aplikacija. (Arrachequesne, 2021)

Ova biblioteka je dodata radi budućeg razvoja aplikacije. Pošto Modelator servis radi isključivo sa modelima mašinskog učenja, za određene kalkulacije i kalibracije je potreban duži vremenski period. Tako da ako klijent pokrene dugotrajnu predikciju može doći do odbijanja konekcije usled predugačkog vremena za odgovor na zahtev. Ovaj problem se može rešiti websocket protokolom. Kad klijent posalje zahtev dobija odogvor da je kalkulacija započeta, a onda se rezultat šalje klijentu web socketom na događaj završetka proračuna ili ako klijent nije dostupan rezultat se čuva uz nalog klijenta i onda se na događaj konekcije na server šalju informacije.

### Got

Kad expres primi zahtev taj zahtev se dalje prosleđuje odgovarajućem servisu tako što sam broker šalje novi zahtev servisu. Inicijalni plan je bio da se koristi request biblioteka, ali tu biblioteku više niko ne održava, pa samim tim predstavlja potencijalan bezbednosni problem u budućnosti. Alternative request biblioteci su got, node-fetch, axios(iskorišćen unutar frontend servisa) i super agent. Got je odabran kao najbogatija biblioteka koja ima i mogućnosti koje prevazilaze mogućnosti request biblioteke. Te dodatne mogućnosti se trenutno ne koriste ali bi mogle biti značajne u budućnosti. Neke od dodatnih mogućnosti su ponovno slanje zahteva u slušaju greške prilikom inicijalnog zahteva. Ova mogućnost doprinosi stabilnosti sistema u slučaju da je otežana komunikacija između servisa, ova mogućnost nije dostupna u axios biblioteci koja je korišćena u frontendu. Takođe jedna od korisnih mogućnosti je prihvatanje grešaka sa meta podacima što moze biti korisno u rešavanju bagova u toku rada.(Liz Parody, 2019)

### Cors

Zbog bezbednosnih razloga internet pregledači sa standardnim podešavanjima ne dozvoljavaju ajax zahteve ka različitim domenima od domena stranice odakle se zahtev šalje. Ovakva vrsta komunikacije se naziva CORS *Cross Origin Resource Sharing*. Da bi se ovakva komunikacija omogućila servis od kog se zahtevaju informacije u hederu odgovora mora da ima setovanu Access-Control-Allow-Origin vrednost. Ta vrednost treba da bude domen sa kog dolazi zahtev ili ako je servis dostupan bilo kom potražiocu servisa.

Sa obzirom da je svaki servis u aplikaicji ima svoj domen problem nastaje kad frontend servis zatraži podatke od broker servisa. Pošto ova dva servisa nisu istog domena, potrebno je omogućiti na broker servisu da u hederu odgovora ima eksplicitno naglašeno da dozvoljava CORS. Ovo se najlakše rešava dodatnom bibliotekom “cors”. Express koristi ovu biblioteku da se hederu setuje “Access-Control-Allow-Origin” vrednost na “\*” i time da se omogući pristup sa bilo kog domena. (Hevia, 2020)

### Flask

Flask je mikro framwork za *Python*. Mikro frejmwork je zato što ne zavisi od dodatnih biblioteka i zato što nema dodatnih mogućnosti koje inače većina frejmworka ima, poput autorizacije i sloja za apsrakciju baze podataka. Sve dodatne mogućnosti su dostupne u vidu plugina ili se mogu ručno realizovati u skladu sa potrebama aplikacije.(*Foreword — Flask Documentation (1.1.x)*, 2021). Alternativno je mogao biti korišćen veoma popularan frejmvork Đango(*Django*).

Tabela 4. Komparacja Flask i Đango frejmworka (*Flask vs Django- The Hot Debate of Python Development Section*, 2019)

|  |  |  |
| --- | --- | --- |
| **Karakteristike** | **Đango** | **Flask** |
| Tip frejmworka | Full Stack Web Framework | Web server interface gateway frejmwork koji u osnovi samo moze da prihvata i odgovara na zahteve |
| Fleksibilnost | Fleksibilnost zavisi od brojnih dodataka | Potpuna fleksibilnost |
| Mapiranje baze podataka | Ugrađen ORM | SQLAlchemy ili bilo koja druga biblioteka za konekciju na bazu podataka |
| Dizajn | Kompletan, sadrži sve sto je potrebno, a i dosta toga što verovatno i nije | Minimalistički. Predstavlja samo kostur na koji se sve mora ručno dodati |
| Stil aplikacja | Monolitan | Opcije su mnoge, zavisi samo od programera |
| Upotreba | Popularan u industrijama i projektima sa kratkim rokovima | Pogodan za manje projekte, servise i projekte u kojima je bitna kontrola nad svakim delom aplikacije. |

Flask i Đango su dva vrlo različita frejmworka u svakom pogledu. Fleksibilnost i minimalnost Flask-a su korisni za manje aplikacije ili za aplikacije u kojima su preformanse vrlo bitne, popularan je kod početnika zbog svoje jednostavnosti, ali ima vrlo strmu krivu učenja jer je neophodno poznavati svaki deo aplikaicje. Sa druge strane Đango ima mnoštvo mogućnosti koje olakšavaju razvoj, stoga je popularan u industrijskim okruženjima gde je brzina razvoja bitna.

Mogućnosti Đanga predstavljaju nepotrebne resurse i nepotrebno opterećenje za jednostavne servise stoga je odabran Flask. Zbog svoje minimalnosi frejmworka Flask je idealan za servise i zato se koristi u dva servisa aplikacije ovog master rada, Predictor i Modelator. (*Flask vs Django- The Hot Debate of Python Development Section*, 2019)

Ovaj frejmwork se takođe ponaša kao server, slično kao i Express za Node.js tako da se putanje direktno definišu u kodu i nema potrebe za dodatnim slojevima za realizaciju servisa. Flask je odabran za servise modelator i prediktor zbog svoje jednostavnosti i slobode koju pruža, tako da su servisi realizovani u funkcionalnim programiranjem.

### Apscheduler

Za realizaciju Predictor servisa se pojavila potreba za zakazivanjem zadataka radi dobijanja najsvežije predikcije. Da je implementaicja servisa na realnoj mašini bilo bi lako podesti izvršavanje zadataka uz pomoc cron servisa, ali pošto se lokalni deploy radi kroz docker compose, pitanje je kako bi se svaki put podešavao cron. Tako da je odlučeno da se ovom problemu pristupi kroz sam servis (kod).

Zadati se mogu zakazivati tako da se izvrše samo jednom ili da se izvršavaju periodično ili da se izvršavaju na zadate datume. Prednosti ovakovog pristupa u odnosu na cron servis su pristupačnost, razmena i čuvanje podešavanja i nezavisnost od platforme. Pod pristupačnosti se misli na to što je moguće podesiti zadatke direktno u kodu ili čak napraviti interfejs za zakazivanje zadataka sa daljine. Mogućnost čuvanja događaja u bazi omogućava da se ne pogube prilikom restartovanja servisa, a omogućava i deljenje zadataka prilikom promene sistema koji poziva zadatke. Nezavisnost od platforme je možda najbitnij aspekt ove biblioteke jer omogućava da servis može biti implementiran na bilo kojoj platformi koja podržava python i ima pristup mreži. (Grönholm, 2021)

### Pandas

Pandas je biblioteka koja omogućava manipulaciju i analizu podataka u pythonu. Biblioteka omogućava strukture za čuvanje i manupulacije serija podataka poput tabela i podataka sa vremenskom komponentom.

Glavni objekat koji pruža biblioteka i nad kojim se vrši manupulacija se zove dataframe. Pandas biblioteka ima mogućnost da se podaci učitavaju u dataframe objekte iz raznovrsnih izvora poput: csv fajlova, eksel fajlova, JSON stringova, ili sql-om iz različitih baza. Pandas dozvoljava različite operacije nad podacima poput spajanja, pretrage, prečišćavanja i konverzije iz sirovih podataka. (McKinney, 2011) Pandas biblioteka je korišćena zbog brzine i jednostavnosti rada sa podacima i zato što Scikit-learn biblioteka u svojim operacijama radisa dataframe objektima.

### Scikit-learn

Sickit-learn je python biblioteka za mašinsko učenje. Ona u sebi sadrži različite algoritme za klasifikacije, regresije i klasterovanje. U konkretnom slučaju predikcije se koristi decisiont tree regresor, a u modelatoru su implementirani još sledeći regresori: linerni regresor, knn, random forest i SVM (support vector machine) pored već pomenutog decision tree algoritma. (Pedregosa и остали, 2011)

Ova biblioteka otvara mogućnosti mašinskog učenja većem auditorijumu nego što je ranije bilo dostupno. Iako u velikom olakšava upotrebu mašinskog učenja i dalje je potrebno znanje programiranja u pythonu što otežava primenu mašinskog učenja u već postojećim sistemima i zahteva izdavajanje vremena i truda da se nauči dodatni programski jezik, za šta mnogi nemaju vremena i kapaciteta. Baš je ovaj problem je jedan od glavnih motiva za izradu aplikacije ovog master rada.

## Korišćeni servisi

U duhu arhitekture aplikacija se sastoji iz nekoliko servisa koji su zaduženi za određene delove poslovne logike. Aplikacija ima servise za sve tri uloge koje mogu postojati u servisno orijentisanoj arhitekturi. Postoje dva provajdera servisa Prediktor i Modelator, sa tim što Prediktor igra i ulogu potraživača servisa i to Modelator servisa. Drugi potraživač je Frontend servis, u pitanju je statički servis koji klijentu vraća frontend aplikaciju u React-u prilikom učitavanja aplikacije servis sa klijentske mašine potražuje rezultate od prediktora da bi ih prikazao. I jedan servis da ih sve okupi i u aplikaciju spoji, Broker. Sva komunikacija između servisa se vrši preko brokera. Iako bi servisi mogli međusobno da se povezuju odlučeno je da se to ne radi jer je na ovaj način potrebno buduće validacije staviti samo na jedno mesto, a i lakše se prati međusobna interakcija servisa i sprečava najveća boljka servisno orijentisanih aplikacija, pravljenje gomile servisa za koje niko nije baš siguran šta i sa čime rade.

Ova četiri servisa i njihovi međusobni odnosi predstavljaju servisno orijentisanu aplikaciju za predviđanje cene zlata.

### Broker

Borker je servis preko kog idu svi zahtevi ka drugim servisima. Ovaj servis je realizovan u Node.js uz upotrebu Express frejmvorka i Socket.io biblioteke i Got biblioteke. Sam servis se sastoji iz dva dela, prijemni i otpremni deo. Prijemni deo se sastoji od dva paralelna interfejsa. Deo za REST zahteve je realizovn u Express-u, a deo za Web Soket je realizovan uz pomoć Socket.io biblioteke. Ova dva dela potom u skladu sa zahtevima koje dobijaju pozivaju odgovajarujći interfejs ka zahtevanom servisu. Interfejsi za pozivanje servisa se nalaze u zasebim fajlovima, tako da svaki servis ima svoj fajl za interakciju sa njim. Ovako izdvojeni, fajlovi za interakciju sa servisima omogućavaju uvođenje generisanja koda na osnovu specifikacije API-ja servisa koji se dodaje brokeru. Za opšte provere zahteva proveru treba dodati u Prijemni deo Brokera (validacija tokena, provera IP adresa potraživača…) a unutar Otpremnog dela se mogu staviti dodatne provere specifične za same servise (nedovoljan broj parametara za pozivanje servisa…).

Ovaj servis je prilikom lokalnog deploymenta preko docker-a dostupan na adresi [http://localhost:8000](http://localhost:8000/) ako se zatraži GET metodom bez dodatnih parametara Broker će vratiti Frontend servis.

Prijemnik

Otpremnik

Express

Socketio

Frontend

Prediktor

Modelator

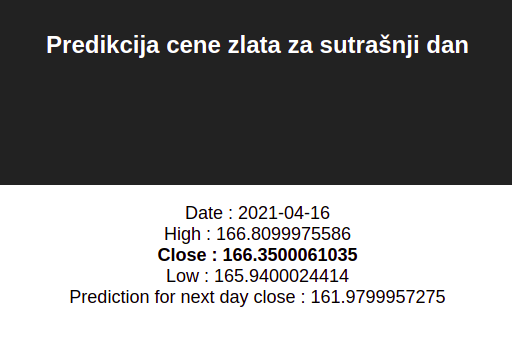
Broker

Slika 14. Unutrašnja strukture Broker servisa

### Frontend

Frontend je servis koji predstavlja korisnički interfejs. Korisnički interfejs je realizovan kao single page aplikacjia realizovana sa React bibliotekom. Prilikom učitavanja stranice klijent šalje zahtev brokeru da dohvati informacije od prediktora o predviđenim cenama zlata za idući berzanski dan. Ovaj zahtev se realizuje uz pomoć axios biblioteke. Ovaj servis prati razvoj ostalih servisa jer on predstavlja prozor za iterakciju klijentima, ali nije nužan za rad same aplikacije jer je moguće koristiti aplikaciju direktno preko zahteva što je naročito korisno ako se aplikacija koristi kao dodatni servis za već postojeća rešenja.

Na lokalnom deplojmentu ovaj servis je dostupan na adresi [http://localhost:3000](http://localhost:3000/) ili preko brokera GET metodom bez dodatnih parametara.



Slika 15. Prikaz predikcije klijentu.

### Prediktor

Prediktor je servis koji priprema sve što je potrebno za samu predikciju cene zlata. Prilikom prvog pokretanja pokreće se Apscheduler da na određeni interval poziva funkciju za pravljenje predikcije i da rezultate te funkcije upiše u globalnu varijablu koja sadrži kompletan dataframe predikcije. Na svaki interval funkcija za predikciju će biti ponovo pozvana i novi rezultati će pregaziti stare rezultate.

Funkcija za predikciju poziva biblioteku yahoo finance odakle dobija dataframe sa informacijama o ceni zalata za prethodnih godinu dana. Tom datafrejmu potom dodaje jos jednu kolonu i u nju upisuje vrednosti Close cene zlata od sutrašnjeg dana, ova vrednost će nam predstavljati y dataset prilikom treniranja modela.

Takav dataset se potom pretvara u csv i zajedno sa dodatnim informacijama o kolonama koje treba izostaviti iz predikcije, kolonama koje predstavljaju y data set i metodu koji treba da se primeni na datasetu se šalje modelatoru na obradu. Kad modelator vrati rezultat, taj rezultat se pretvara u dataframe i vraća se u globalnu vrijablu dataseta.

Potražnja podataka se vrši kroz drugi deo aplikacije koja je realizovana kroz flask mikro frejmwork. Kad prediktor servis dobije zahtev za informacijama vraća se određeni deo datafrejma iz globalne varijable koji se potom pretvara u json objekat radi lakše manipulacije od strane korisnika.

### Modelator

Modelator predstavlja srce aplikacije i odgovor na problem pristupačnosti mašinskom učenju. Modelator je takođe napravljen u flask mikro frejmworku i shodno zahevu poziva određeni model. Ceo servis je napravljen da se ponaša kao poziv funkciji što omogućava da se servis hostuje kao serverless funkcija. Ovakav vid hostinga predstavlja dobar izbor jer Modelator nije pod stalnim opterećenjem, a kad jeste, veći resursi mogu pomoći značajno. Ako se hosting plaća samo kad se koristi time se potencijalno smanjuju troškovi jer nam ne treba snažan server koji je uvek dostupan, a potencijalno dobar deo vremena ne radi ništa.

Modelator na raspolaganje daje pet regresivnih alogritama za predikciju mašinskim učenjem:

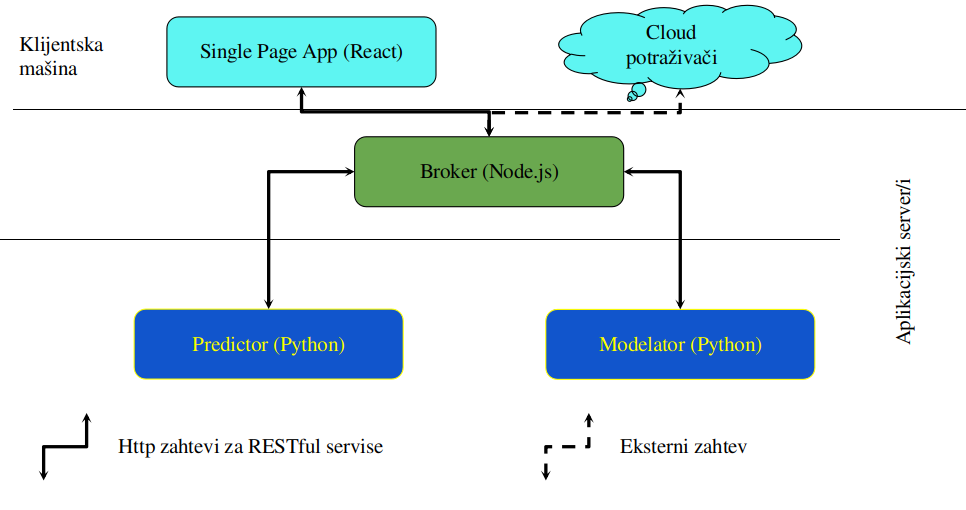
* Decision Tree
* Random Forest
* Linear regresor
* KNN
* SVM

Ovih pet algoritama su dostupni sa standardnom obukom sa podelom datafrejmana 80-20% delove za train i test dataset i ovaj odnos je moguće definisati prilikom potraživanja servisa. Ovakva predikcija je često korišćena u radovima da se pokaze da mašinsko učenje može da radi predikciju cene zlata, takav pristup ima velikih propusta, jer pogađa nasumičnih 20% dataseta i u obzir uzima skoro ceo dataset gde u suštini pokušava samo da što bolje zakrpi 20% „rupa“ u podacima.

Zbog prethodno navedenih problema modelator ima i mogućnost rolling window predikcije gde je moguće definisati veličinu prozora koji treba uzeti u obzir za treniranje modela pre nego što se testira na prvom sledećem unosu. Ovakav pristup se pokazao vrlo uspešnim u istraživanju Jovane Stokanović Šević.

## Arhitektura aplikacije

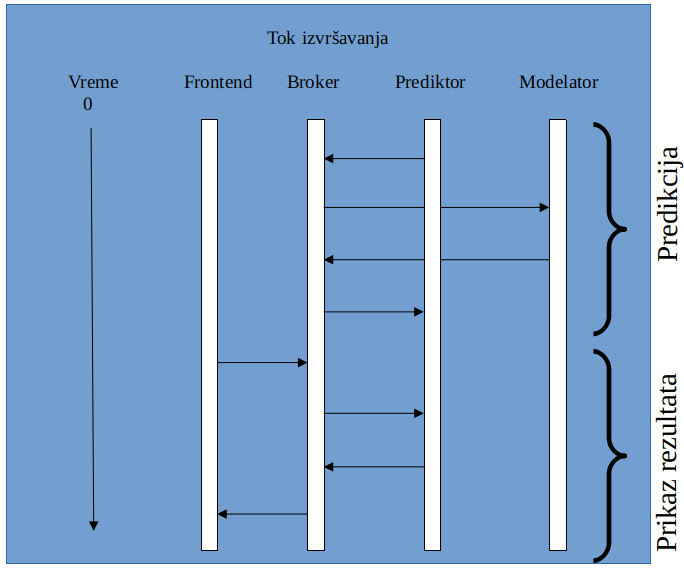
Komunikacija između servisa je isključivo preko Broker servisa, i to bez obzira da li je uloga servisa provajder ili potraživač, da li se servis izvršava na serveru ili kod klijenta. Struktura aplikacije se moze prikazati sledećom šemom. Osim komunikacije preko korisničkog interfejsa moguće je direktno pristupiti brokeru za ostale servise, ovakvim pristupom je moguće pozivati direktno modelator što predstavlja zanimljivu uslugu za integraciju mašinskog učenja u projekte koji ne mogu lako da inkorporiraju python.



Slika 16. Strukturna organizacija aplikacije.

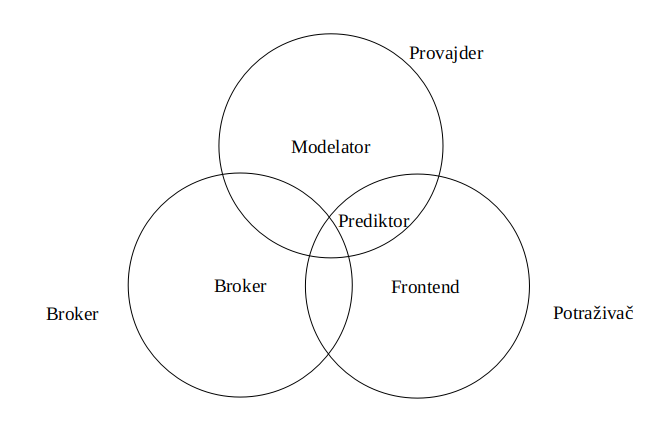
Sa pokretanjem instance aplikacije prvu operaciju započinje Prediktor pozivanjem Modelator servisa preko Brokera da bi dobio prvobitnu predikciju cene zlata. Kad prediktor dobije odgovor od Modelatora dataset predikcija se čuva unutar Prediktora.

Odlaskom na naslovnu stranicu aplikacije zahtev dolazi do Brokera koji vraća Frontend servis. Trenutno je moguće tražiti frontend servis direktno jer web socket nije implementiran do kraja, tako da nema potrebe za vođenjem evidencije konekija. Kad klijent dobije Frontend servis servis se pokreće u okviru pregledača, sa pokretanjem frontend servisa poziva se broker da pozove Prediktor da vrati vrednost predikcije za sledeći investicioni dan, kad Prediktor vrati odgovor i kad odgovor dođe do Frontend servisa, menja se stanje Frontend servisa i prikazuju se informacije klijentu.



Slika 17. Prikaz toka aplikacije

Po ulogama koje servisi imaju dele se u tri grupe servisa: Provajder, Broker i Potraživač. U aplikaciji postoje sevisi za svaku od uloga, a jedan od servisa ima i dvojnu ulogu tako da je i potraživač i povajder.



Slika 18. Prikaz uloga servisa

# Zaključak

Ovom aplikacijom su približene mogućnosti mašinskog učenja široj javnosti. Aplikacija svojom servisno orijentisanom arhitekturom rešava dva problema.

Prvi problem je problem pristupačnosti mašinskog učenja na već postojećim sistemima i platformama koje ne podržavaju kompleksne procese poput mašinskog učenja. Drugi problem je predikcija finansijskih derivata poput zlata. Aplikacija rešava prvi problem servisom za modeliranje, Modelator, u kome je primenjena *scikit-learn* biblioteka koja sadrži različite algoritme mašinskog učenja. Problem predikcije cena finansijskih derivata je rešen prediktor servisom koji prikuplja podatke i priprema ih za predikciju, i nakon uspešne predikcije čuva podatke i čini ih dostupnim korisnicima servisa. Prediktor servis se može posmatrati i kao demonstracija potencijala Modelator servisa

Aplikacija ima potencijala za dalji razvoj, a predstavlja i dobru početnu tačku za buduće projekte.

Prateći poslednji postulat servisno orijentisane arhitekture koji kaže da je evolutivno usavršavanje bitnije od inicijalne savršenosti u ovom poglavlju se navode smernice za dalje usavršavanje ove aplikacije.

Aplikacija je potvrdila da je ideja o aplikaciji koja će se baviti predikcijom moguća. da bi dalji razvoj bio sa što manje poteškoća i da bi se obezbedio kvalitet aplikacije potrebno je uvesti testiranje.

Testiranje može biti organizovano na nivou komponenata unutar samih servisa, na nivou servisa i funkcionalnosti i na nivou celokupne aplikacije i međusobnih odnosa među servisima.

Automatsko testiranje unutar servisa se vrši *Unit* testovima koji proveravaju izvršavanje pojedinačnih metoda/funkcija. Filozofija ovih testova je da ako i najmanji deo sistema radi kako treba da će i sam sistem raditi kako treba. Ovako posmatranje u realnim uslovima nije najtačnije jer logičke greške lako prolaze neopaženo, pogotovo ako su greške u široj slici, bez obzira na to vrlo je korisno imati ovakve testove i uhvatiti potencijalno lako rešive bagove i pre nego što promene budu bilo gde primenjene.

Automatsko testiranje samih servisa je slično unit testovima, i u nekim slučajevima se i koriste unit testovi u tu svrhu. Ovo testiranje se može podeliti na dve vrste testiranja, testiranje grafičkih interfejsa i testiranje API-a.

Testiranje grafičkih interfejsa je vrlo zahtevan proces zato što su mogućnosti koje pruža grafički interfejs često ogromne, a onda na to je još bitan i redosled izvršavanja komandi. S obzirom na obimnost i monotonost posla ovo testiranje se u poslednje vreme prebacuje na veštačku inteligenciju kojoj se daju granični uslovi i traži se da reprodukuje željeni rezultat.

Testiranje API-ja je znatno jednostavnije jer sekvenca izvršavanja nije naručito bitna jer je moguće postaviti prethodno sačuvano stanje pre testiranja ili izvršiti komande koje su potrebne prethodno testiranju. Testiranje celokupnog sistema je potrebno izvršiti sa bezbednosnog, funkcionalnog i performansnog aspekta.

Bezbednoso testiranje je testiranje sistema na različite napade i pokušaje korupcije sistema. Za ovakvo testiranje se često unajmljuju eksterne specijalizovane kompanije zbog objektivnosti.

Funkcionalno testiranje je slično testiranju samih servisa, ponekad je dobra ideja pustiti beta verziju aplikacije i tražiti od korisnika da prijave probleme u radu.

Testiranje performansni za zadatak ima da nađe koji delovi aplikacije i celokupnog sistema zajedno sa serverima i internet provajderima imaju problema sa preformansama. Ovi testovi su takođe poznati kao testovi opterećenja.

Testiranje se treba vršiti sa svakom promenom na kodu, preporučljivo je da se promene ne mogu primeniti u slučaju da nemaju uspešno izvršeno testiranjee. Kad se ovakav sistem testiranja primeni može se nastaviti sa daljim razvojem i za svaku novu funkcionalnost treba napisati i prateće testove.

Prvo, i možda najznačajnije unapređenje ove aplikacije bi bilo uvođenje *User* servisa. Ovaj servis bi bio zadužen za kontrolu pristupa aplikaciji. Ovaj servis bi bio od ključnog značaja ako bi aplikacija prešla u produkcijsku fazu, jer je modelator servis jako lako preopteretiti do neupotrebljivosti.

Sladeći korak u razvoju bi trebalo da bude servis za čuvanje rezultata modela klijenata. Sa ovim servisom bi implementirana *web socket* tehnologija mogla da pokaže svoj pun potencijal tako što klijent dobije rezultate predikcija koje su se izračunale dok je klijent bio odsutan. Predlaže se da se servis za čuvanje odvoji od *User* servisa da ne bi došlo do preopterećenja user servisa i da se smanji mogućnost za pravljenje grešaka unutar *User* servisa što bi moglo da dovede do ozbiljnih bezbednosnih problema.

Kad su ova dva produkcijski nepohodna servisa dodata sledeći koraci se mogu dodavati bilo kojim redom.

Jedan od najlakših koraka bi bio da se Modelator servisu doda mogućnost primene i ostalih algoritama tako da bi zaista predstavljao jedan potpun servis za mašinsko učenje.

Osim modelatora Prediktoru bi mogle da se dodaju nove mogućnosti poput različitih finansijskih derivata, čuvanja rezultata u bazi, računanje tačnosti predikcija, predikcije za više od jednog dana u napred i predikcija celog dataseta samo ako dataset nema već predikciju, a ako ima da se radi samo predikcija za dane koji nemaju predikciju. Sa ovim unapređenjima Prediktor servis bi bio značajan finansijski/investicioni alat.

Unapređenja Modelatora i Prediktora bi trebalo da prate i izmene u Frontend servisu jer su sve izmene nebitne ako nisu primenljive. Frontend servis posle izmena na modelatoru i prediktoru mogao bi da se podeli na dva dela, Forntend za predkiciju finansijskih derivata i Frontend za mašinsko učenje i time bi ova aplikacija mogla da se predstavi kao dve odvojene aplikacije iako koriste iste servise.

Broker unapređenja GraphQL, generisanje koda za konekcije

Ova aplikaicja predstavlja dobru odskočnu dasku za druge aplikacije jer se funkionalnost može u potpunosti promeniti izmenom samo par servisa. Čak bi i bilo moguće dodati još servisa uz već postojeće i onda samo *user* interfejsom (frontendom) izdvojiti različite aplikacije iako u pozadini dele iste servise.

Konkretno ova aplikacija ima dosta prostora za usavršavanje funkcionalnosti. Unutar prediktor servisa može se dodati još finansijskih derivata poput investicionog srebra i akcija sa berze. Takođe, može se dati da ako klijent traži neku vrednost koja ranije nije bila korišćena da se uradi predikcija za tu vrednost i da se rezultati sačuvaju tako da budu dostupni i ostalim klijentima. Može se dodati da se na *update* predikcije šalje zahtev brokeru da obavesti frontend svih klijenata preko web socketa, iz tog razloga broker ima socket.io biblioteku primenjenu. Modelator servisu dodati mogućnost slanja dodatnih opcija koje modeli pirhvataju, tako da je moguće još detaljnije podesiti same modele. Modelator je takođe dobar kandidat za *serverless deployment* jer se ponaša kao funkcija koja uvek vraća isti rezultat za iste ulazne podatke, takođe moguće je da je nekim modelima potrebno dosta vremena da se izvrše što bi moglo da dovede do pretrpavanja klasičnih *deployment* rešenja, a sa *serverless* pristupom se naplaćuje samo vreme egzekucije i moguće je da neograničeno traje. Ako se doda jos *User* servis moguće je i naplaćivati korisnicima upotrebu aplikacije, tako da bi se i sami troškovi *serverless* izvršavanja odmah prebacivali na krajnje korisnike. Frontend bi trebalo da prati sva pomenuta dalja usavršavanja, tako da je potrebno napraviti i interfejs za direktnu upotrebu modela, i interfejs za odabir i dodavanje finansijskih derivata.

# Literatura

1. Altig, D. E., Baker, S., Barrero, J. M., Bloom, N., Bunn, P., Chen, S., Davis, S., Leather, J., Meyer, B., Mihaylov, E., Mizen, P., Parker, N., Renault, T., Smietanka, P., & Thwaites, G. (2020). Economic uncertainty in the wake of the COVID-19 pandemic. *VoxEU.org*. https://voxeu.org/article/economic-uncertainty-wake-covid-19-pandemic
2. Arrachequesne, D. (2021). *Socket.IO*. Socket.IO. https://socket.io/index.html
3. Azzutti, A. (2016). *Forecasting Gold Price A Comapative Study*. University of Florence.
4. Bell, M. (2008). *Service-oriented modeling: Service analysis, design, and architecture*. Hoboken, N.J. : John Wiley & Sons. http://archive.org/details/serviceorientedm00bell
5. Bingol, S., Sadik, S., Abu Bakar, R. M., Mouhcine, H., Albaf, J., Chaabene, S., & Auta, A. (2020). *Gold price prediction in times of financial and geopolitical uncertainty: A machine learning approach*. https://doi.org/10.13140/RG.2.2.31050.39366
6. Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer-Verlag. https://www.springer.com/gp/book/9780387310732
7. Brownlee, J. (2016). How To Backtest Machine Learning Models for Time Series Forecasting. *Machine Learning Mastery*. https://machinelearningmastery.com/backtest-machine-learning-models-time-series-forecasting/
8. Brownlee, J. (2017). How to Create an ARIMA Model for Time Series Forecasting in Python. *Machine Learning Mastery*. https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/
9. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond* (2nd edition). Addison-Wesley Professional.
10. DA-14. (2018). *The Ultimate Guide to API Architecture: REST, SOAP or GraphQL? | DA-14*. <https://da-14.com/blog/ultimate-guide-api-architecture-rest-soap-or-graphql>
11. Despodovski, R. (2021) “Microservices vs. SOA – Is There Any Difference at All?”. Dostupno na: <https://dzone.com/articles/microservices-vs-soa-is-there-any-difference-at-al> (14/4/2021)
12. Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. У M. Mazzara & B. Meyer (Ур.), *Present and Ulterior Software Engineering* (стр. 195–216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4\_12
13. *Express—Node.js web application framework*. (2021). https://expressjs.com/
14. *Flask vs Django- The Hot Debate of Python Development Section*. (2019, Септембар 7). DataFlair. https://data-flair.training/blogs/flask-vs-django/
15. *Foreword—Flask Documentation (1.1.x)*. (2021). https://flask.palletsprojects.com/en/1.1.x/foreword/
16. Garlan, D., & Shaw, M. (1993). AN INTRODUCTION TO SOFTWARE ARCHITECTURE. У V. Ambriola & G. Tortora, *Series on Software Engineering and Knowledge Engineering* (Том 2, стр. 1–39). WORLD SCIENTIFIC. https://doi.org/10.1142/9789812798039\_0001
17. Grönholm, A. (2021)“Advanced Python Scheduler—APScheduler 3.7.0 documentation”. APScheduler. Dostupno na: <https://apscheduler.readthedocs.io/en/stable/> (14/4/2021)
18. Hevia, A. (2020). *Using CORS in Express*. Medium. https://medium.com/zero-equals-false/using-cors-in-express-cac7e29b005b
19. Jacobi, J. (2017). *How to Build a Decision Tree | Bloomreach Developers*. Bloomreach.Com. https://developers.bloomreach.com/blog/2017/how-to-build-a-decision-tree-in-bloomreach-experience.html
20. Jansen, S. (2020). *Machine Learning for Algorithmic Trading: Predictive models to extract signals from market and alternative data for systematic trading strategies with Python, 2nd Edition*. Packt Publishing.
21. Kayode, B. (2020). *Running CPU Intensive task in Nodejs*. DEV Community. https://dev.to/kayslaycode/running-cpu-intensive-task-in-nodejs-201k
22. Kominos, C. G., Seyvet, N., & Vandikas, K. (2017). *Bare-metal, Virtual Machines and Containers in OpenStack*. 8.
23. Lange, K. (2016). *The Little Book on REST Services*. 31.
24. Laws, M. P., & Goodfellow, M. G. (2018). *Choosing a Web Host: How to Choose the Web Hosting Service that is Best for You* (1st edition). CreateSpace Independent Publishing Platform.
25. Leach, P. J., Berners-Lee, T., Mogul, J. C., Masinter, L., Fielding, R. T., & Gettys, J. (1999). *Hypertext Transfer Protocol—HTTP/1.1*. https://tools.ietf.org/html/rfc2616
26. Lewis, P. (1976). Nixon’s Economic Policies Return to Haunt the G. O. P. *The New York Times*. https://www.nytimes.com/1976/08/15/archives/nixons-economic-policies-return-to-haunt-the-gop-nixons-economic.html
27. Liz Parody. (2019, Април 3). *Request is going into maintenance mode, this is what you need to know*. The NodeSource Blog - Node.Js Tutorials, Guides, and Updates. http://nodesource.com/blog/express-going-into-maintenance-mode/
28. Lubbers, P., & Greco, F. (2020). *HTML5 WebSocket—A Quantum Leap in Scalability for the Web*. http://www.websocket.org/quantum.html
29. macrotrends. (2021, Април 14). *Historical Gold price graph*. https://www.macrotrends.net/1333/historical-gold-prices-100-year-chart
30. McKinney, W. (2011). *pandas: A Foundational Python Library for Data Analysis and Statistics*. 9.
31. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*(85), 2825–2830.
32. Pressman, R. S., & Maxim, B. (2014). *Software Engineering: A Practitioner’s Approach* (8th edition). McGraw-Hill Education.
33. Rauschmayer, A. (2014). *Speaking JavaScript: An In-Depth Guide for Programmers* (1st edition). O’Reilly Media.
34. *React – A JavaScript library for building user interfaces*. (2021). https://reactjs.org/
35. *React Native · Learn once, write anywhere*. (2021). https://reactnative.dev/
36. Riazuddin, M. (2020, Април 20). *Machine Learning to Predict Gold Price Returns*. Medium. https://towardsdatascience.com/machine-learning-to-predict-gold-price-returns-4bdb0506b132
37. Rotem-Gal-Oz, A. (2012). *SOA Patterns* (1st edition). Manning Publications.
38. Russell, S., & Norvig, P. (2009). *Artificial Intelligence: A Modern Approach* (3rd edition). Pearson.
39. Schott, M. (2020). *K-Nearest Neighbors (KNN) Algorithm for Machine Learning*. Medium. https://medium.com/capital-one-tech/k-nearest-neighbors-knn-algorithm-for-machine-learning-e883219c8f26
40. *SOA Manifesto*. (2009). http://www.soa-manifesto.org/
41. Stokanović Šević, J. (2020). *Model predikcije kretanja cena zlata uzimajući u obzir broj zaraženih od Covid-19 virusa*. Singidunum University.
42. Teixeira, P. (2012). *Professional Node.js: Building Javascript Based Scalable Software*. John Wiley & Sons.
43. Thiebaut, C. (2019). *Gradient descent for linear regression using Golang*. Backlog. <https://backlog.com/blog/gradient-descent-linear-regression-using-golang/>
44. Velvetech (2021) Microservices vs Web services. Dostupno na: <https://www.velvetech.com/blog/microservices-vs-web-services/> (19/4/2021)
45. Warwick-Ching, T. (1993). *The International Gold Trade*. Woodhead Publishing.
46. Weng, F., Yinghao, C., Wang, Z., Hou, M., Luo, J., & Tian, Z. (2020). Gold price forecasting research based on an improved online extreme learning machine algorithm. *Journal of Ambient Intelligence and Humanized Computing*, *11*. https://doi.org/10.1007/s12652-020-01682-z
47. Yerra, B. (2019). *Predicting Tomorrows Gold Price*. https://mlbhanuyerra.github.io/2019-10-12-Predicting-Tomorrows-Gold-Price/
48. Yousef, I., & Shehadeh, E. (2020). The Impact of COVID-19 on Gold Price Volatility. У *International Journal of Economics and Business Administration: Том VIII*. https://doi.org/10.35808/ijeba/592
49. Zirguezi, & Alisneaky. (2011). *English: Kernel machines are used to compute non-linearly separable functions into a higher dimension linearly separable function.* Own work. https://commons.wikimedia.org/wiki/File:Kernel\_Machine.svg

# Prilozi

## Prilog A: Github i struktura fajlova

**Github:** Kod projekta je dostupan na repozitorijumu <https://github.com/Msevic/master>.

**Struktura fajlova**

|  |  |
| --- | --- |
|  |  |
| Struktura celog projekta | Struktura Broker servisa |
|  |  |
| Struktura Frontend servisa | Struktura Modelator servisa |
|  |  |
| Struktura Prediktor Servisa |  |

## Prilog B: Van servisni fajlovi

**README.md**

1. *# Master*
2. Application for predicting gold price
3. Created for the needs of master thesis
4. Use docker compose to run this software

**LICENSE**

1. GNU GENERAL PUBLIC LICENSE
2. Version 3, 29 June 2007

**docker-compose.yml**

1. Version: "3.4"
2. services:
3. broker:
4. build:
5. context: ./broker
6. network: host
7. command: npm run dev
8. volumes:
9. - ./broker:/usr/src/app/
10. ports:
11. - "8000:8000"
12. frontend:
13. build: frontend
14. command: npm start
15. volumes:
16. - ./frontend:/app/
17. ports:
18. - "3000:3000"
19. environment:
20. - CHOKIDAR\_USEPOLLING=true
21. predictor:
22. build:
23. context: ./predictor
24. network: host
25. volumes:
26. - ./predictor/:/usr/src/app/
27. ports:
28. - 5000:5000
29. environment:
30. PORT: 5000
31. FLASK\_DEBUG: 1
32. depends\_on:
33. - modelator\_d3\_rw
34. modelator\_d3\_rw:
35. build:
36. context: ./modelator\_d3\_rw
37. network: host
38. volumes:
39. - ./modelator\_d3\_rw/:/usr/src/app/
40. ports:
41. - 6001:6001
42. environment:
43. PORT: 6001
44. FLASK\_DEBUG: 1

**gitignore**

1. .idea

## Prilog C: Fajlovi Broker Servisa

**broker/sevices/forntend/service.js**

1. *frontend* = async (app) => {
2. app.get('/', function (req, res) {
3. res.redirect('[http://localhost:3000](http://localhost:3000/)')
4. });
5. };
6. module.exports = *frontend*

**broker/sevices/modelator\_d3\_rw/service.js**

1. const got = require('got');
2. const proxy = require('express-http-proxy');
3. *md3rw* = (app) => {
4. app.get('/md3rw', function (req, res) {
5. got('http://modelator\_d3\_rw:6001').then(response => {
6. res.send(response.body);
7. }).catch(error => {
8. res.send(error.response.body);
9. });
10. });
11. app.use('/md3rw/rw/d3', proxy('modelator\_d3\_rw:6001', {
12. reqBodyEncoding: null,
13. preserveHostHdr: true,
14. proxyReqPathResolver: function(req){
15. return "/rw/d3";
16. }
17. }));
18. }
19. module.exports = *md3rw*

**broker/sevices/predictor/service.js**

1. const got = require('got');
2. *predictor* = (app) => {
3. app.get('/predictor', function (req, res) {
4. got('http://predictor:5000/').then(response => {
5. res.send(response.body);
6. }).catch(error => {
7. console.log(error.response.body);
8. });
9. });
10. app.get('/predictor/predictions/:commodity?/:count?', (req, res) => {
11. var url = '<http://predictor:5000/predictions/>';
12. if( req.params.commodity){
13. var count = req.params.count || 0;
14. url += req.params.commodity + "/" +count
15. }
16. got(url).then(response => {
17. res.send(response.body);
18. }).catch(error => {
19. res.send(error.response.body);
20. });
21. })
22. }
23. module.exports = *predictor*

**broker/index.js**

1. var express = require('express');
2. var app = express();
3. var http = require('http').Server(app);
4. var io = require('socket.io')(http);
5. var *frontend* = require('./sevices/forntend/service')
6. var *predictor* = require('./sevices/predictor/service')
7. var *md3rw* = require('./sevices/modelator\_d3\_rw/service')
8. *frontend*(app)
9. *md3rw*(app)
10. app.get('/md3rw', function (req, res) {
11. *md3rw*(res)
12. });
13. *predictor*(app)
14. io.on('connection', function (socket) {
15. console.log("New Connection")
16. });
17. http.listen(8000, function () {
18. console.log('listening on \*:8000');
19. });

**broker/Dockerfile**

**broker/package.json**

1. {
2. "name": "broker",
3. "version": "1.0.0",
4. "description": "Node.js Broker",
5. "author": "Mihailo Sevic",
6. "main": "index.js",
7. "scripts": {
8. "start": "node index.js",
9. "dev": "nodemon index.js"
10. },
11. "dependencies": {
12. "express": "^4.16.1",
13. "socket.io": "^2.2.0",
14. "nodemon": "^1.18.7"
15. },
16. "devDependencies": {
17. "nodemon": "^1.18.7"
18. }
19. }

## Prilog D: Fajlovi Frontend servisa

**Fajlovi**

Fajlovi frontend servisa su autogenersani na osnovu package.json fajla. Jedino su promenjeni App.js, Dockerfile i package.json fajl.

**frontend/package.json**

1. {
2. "name": "frontend",
3. "version": "0.1.0",
4. "private": true,
5. "dependencies": {
6. "axios": "^0.21.1",
7. "react": "^17.0.1",
8. "react-dom": "^17.0.1",
9. "react-scripts": "1.1.5"
10. },
11. "scripts": {
12. "start": "react-scripts start",
13. "build": "react-scripts build",
14. "test": "react-scripts test –env=jsdom",
15. "eject": "react-scripts eject"
16. }
17. }

**frontend/Dockerfile**

1. FROM node:14
2. *# set working directory*
3. WORKDIR /app
4. *# add `/app/node\_modules/.bin` to $PATH*
5. ENV *PATH* /app/node\_modules/.bin:$*PATH*
6. *# install app dependencies*
7. COPY package.json ./
8. COPY package-lock.json ./
9. RUN npm install –silent
10. RUN npm install react-scripts@3.4.1 -g –silent
11. *# add app*
12. COPY . ./
13. *# start app*
14. CMD ["npm", "start"]

**frontend/src/App.js**

1. import React, {Component} from 'react';
2. import logo from './logo.svg';
3. import './App.css';
4. import axios from 'axios'
5. class App extends Component {
6. state = {
7. today: {
8. Date: "",
9. CLose: 0,
10. High: 1,
11. Low: 0,
12. Prediction: 0
13. }
14. }
15. componentDidMount() {
16. axios.get('<http://localhost:8000/predictor/predictions/GLD/1>')
17. .then(response => {
18. var data = response.data
19. var keys = Object.keys(data)
20. var thekey = keys.reduce(function (p, v) {
21. return (p > v ? p : v);
22. });
23. this.setState({
24. today: {
25. Date: response.data[thekey].Date,
26. Close: response.data[thekey].Close,
27. High: response.data[thekey].High,
28. Low: response.data[thekey].Low,
29. Prediction: response.data[thekey].prediction\_d3
30. }
31. })
32. })
33. }
34. render() {
35. return (
36. <div className="App">
37. <header className="App-header">
38. <h1 className="App-title">Predikcija cene zlata za sutrašnji dan</h1>
39. </header>
40. <p className="App-intro">
41. Date : {this.state.today.Date} <br/>
42. High : {this.state.today.High} <br/>
43. <b>Close : {this.state.today.Close}</b> <br/>
44. Low : {this.state.today.Low} <br/>
45. Prediction for next day close : {this.state.today.Prediction} <br/>
46. </p>
47. </div>
48. );
49. }
50. }
51. export default App;

## Prilog E: Fajlovi Modelator servisa

**modelator\_d3\_rw/app.py**

1. import os
2. from flask import Flask, jsonify, request
3. from modelator import modelator
4. app = Flask(\_\_name\_\_)
5. @app.route(**'/'**)
6. def welcome():
7. return jsonify({**'status'**: **'modelator working'**})
8. @app.route(**'/rw/<model>'**, methods=[**'POST'**])
9. def rw(model=False):
10. if request.method != **'POST'**:
11. return 404
12. if not model:
13. return jsonify({**'available models'**: [**'d3'**, **'lin\_r'**, **'r\_forest'**, **'knn'**, **'svm\_model'**]})
14. split = True
15. rolling\_window = True
16. *# dataframes*
17. test = False
18. if **'test.csv'** in request.files:
19. split = False
20. test = request.files.get(**'test.csv'**)
21. train = request.files[**'train.csv'**]
22. frame\_size = 10
23. test\_size = 0.8
24. if rolling\_window:
25. frame\_size = int(request.form[**'frame\_size'**])
26. if frame\_size < 3:
27. frame\_size = 10
28. else:
29. test\_size = request.form[**'test\_size'**]
30. target\_column = request.form[**'target\_column'**]
31. return modelator(model, split, rolling\_window, train, test, target\_column, frame\_size, test\_size)
32. if \_\_name\_\_ == **'\_\_main\_\_'**:
33. app.run(host=**'0.0.0.0'**, port=os.getenv(**'PORT'**))

**modelator\_d3\_rw/modelator.py**

1. from rolling\_window import roller
2. from standard\_model import standard\_model
3. def modelator(model, split, rolling\_window, train, test, target\_column, frame\_size=10,

test\_size=0.8):

1. if rolling\_window:
2. return roller(model, train, frame\_size, target\_column)
3. else:
4. return standard\_model(model, train, target\_column, split, test)

**modelator\_d3\_rw/models.py**

1. from sklearn import svm
2. from sklearn.ensemble import RandomForestRegressor
3. from sklearn.neighbors import KneighborsRegressor
4. from sklearn.tree import DecisionTreeRegressor
5. from sklearn.linear\_model import LinearRegression
6. def lin\_r(x\_train, y\_train, x\_test):
7. lin = LinearRegression()
8. lin.fit(x\_train, y\_train)
9. lin\_pred = lin.predict(x\_test)
10. return lin\_pred
11. def r\_forest(x\_train, y\_train, x\_test):
12. rfr = RandomForestRegressor(n\_estimators=100, min\_samples\_leaf=10,

random\_state=1)

1. rfr.fit(x\_train, y\_train)
2. predictions = rfr.predict(x\_test)
3. return predictions
4. def knn(x\_train, y\_train, x\_test):
5. n\_neighbors = 2
6. knn\_m = KNeighborsRegressor(n\_neighbors, weights=**'distance'**)
7. knn\_m.fit(x\_train, y\_train)
8. pred\_knn = knn\_m.predict(x\_test)
9. return pred\_knn
10. def d3(x\_train, y\_train, x\_test):
11. dtr = DecisionTreeRegressor()
12. dtr = dtr.fit(x\_train, y\_train)
13. pred\_dtc = dtr.predict(x\_test)
14. return pred\_dtc
15. def svm\_model(x\_train, y\_train, x\_test):
16. svm\_model = svm.SVR(gamma=**'scale'**)
17. svm\_model = svm\_model.fit(x\_train, y\_train)
18. pred\_svm = svm\_model.predict(x\_test)
19. return pred\_svm

**modelator\_d3\_rw/responder.py**

1. from flask import Response
2. def respond(df):
3. return Response(
4. df.to\_csv(),
5. mimetype=**"text/csv"**,
6. headers={**"Content-disposition"**: **"attachment; filename=y\_test.csv"**, **"Content-Type"**: **"text/csv"**})

**modelator\_d3\_rw/rolling\_window.py**

1. import models
2. from responder import respond
3. import pandas as pd
4. import numpy as np
5. from splitter import splitter\_rolling
6. def roller(model, train, frame\_size, target\_column):
7. train = pd.read\_csv(train, index\_col=0)
8. train = train[~train.isin([np.nan, np.inf, -np.inf]).any(1)]
9. method\_to\_call = getattr(models, model)
10. if method\_to\_call:
11. i = frame\_size
12. exclude\_column = **'prediction\_'** + str(model)
13. exclude\_columns = [exclude\_column, **"Date"**]
14. train[exclude\_column] = 0
15. while i < (len(train.index) – 1):
16. i += 1
17. x\_train, y\_train, x\_test = splitter\_rolling(train, frame\_size, i, target\_column, exclude\_columns)
18. train.loc[[i], **'prediction\_'** + str(model)] = method\_to\_call(x\_train, y\_train, x\_test)[0]
19. return respond(train)
20. else:
21. return jsonify({**'error'**: **'selected model not available'**})

**modelator\_d3\_rw/splitter.py**

1. from sklearn.model\_selection import train\_test\_split
2. def splitter(df, target\_column, test\_size):
3. columns = df.columns.tolist()
4. columns = [c for c in columns if c not in [target\_column]]
5. X = df[columns]
6. y = df[target\_column]
7. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=test\_size, random\_state=123)
8. return X\_train, X\_test, y\_train, y\_test
9. def splitter\_rolling(df, frame\_size, start\_index, target\_column, exclude\_columns):
10. start\_index = start\_index - frame\_size – 1
11. end\_index = start\_index + frame\_size
12. columns = df.columns.tolist()
13. excludeArray = [target\_column] + exclude\_columns
14. columns = [c for c in columns if c not in excludeArray]
15. x = df[columns]
16. y = df[target\_column]
17. *# Delimo podatke na train i test set*
18. x\_test = x.iloc[[end\_index + 1]]
19. x\_train = x.iloc[[start\_index, end\_index]]
20. y\_train = y.iloc[[start\_index, end\_index]]
21. return x\_train, y\_train, x\_test

**modelator\_d3\_rw/standard\_model.py**

1. import pandas as pd
2. import models
3. from responder import respond
4. from splitter import splitter
5. def standard\_model(model, train, target\_column, split, test):
6. train = pd.read\_csv(train)
7. if split:
8. x\_train, x\_test, y\_train, y\_test = splitter(df, target\_column, test\_size)
9. else:
10. test = df.read\_csv(test)
11. y\_train = train[target\_column]
12. x\_train = train.drop(target\_column, axis=1)
13. x\_test = test.drop(target\_column, axis=1)
14. method\_to\_call = getattr(models, model)
15. y\_test = method\_to\_call(x\_train, y\_train, x\_test)
16. return respond(y\_test)

**requirements.txt**

1. Flask
2. pandas
3. numpy
4. sklearn

**Dockerfile**

1. FROM python:3.6
2. RUN mkdir /usr/src/app/
3. COPY . /usr/src/app/
4. WORKDIR /usr/src/app/
5. EXPOSE 6001
6. RUN pip install -r requirements.txt
7. CMD ["python", "app.py"]

## Prilog F: Fajlovi Predictor servisa

**predictor/app.py**

1. import os
2. import time
3. import atexit
4. import vars
5. from flask import Flask, jsonify
6. from apscheduler.schedulers.background import BackgroundScheduler
7. from predictor import make\_predictions, get\_prediction\_for, update\_predictions
8. vars.init()
9. scheduler = BackgroundScheduler()
10. *# comment for production*
11. scheduler.add\_job(func=update\_predictions, trigger=**"interval"**, hours=0.5)
12. *# uncomment for production*
13. *# scheduler.add\_job(func=make\_predictions, trigger="cron", hour="4")*
14. scheduler.start()
15. *# Shut down the scheduler when exiting the app*
16. atexit.register(lambda: scheduler.shutdown())
17. app = Flask(\_\_name\_\_)
18. @app.route(**'/'**)
19. def welcome():
20. *# return a json*
21. return jsonify({**'status'**: **'predictor working'**})
22. @app.route(**'/predictions/<commodity>/<rows>'**, methods=[**'GET'**])
23. def get\_latest\_prediction(commodity, rows = 0):
24. if commodity not in [**'GLD'**]:
25. return jsonify({**'avalable predictions'**: [**'GLD'**]})
26. return get\_prediction\_for(commodity, rows)
27. if \_\_name\_\_ == **'\_\_main\_\_'**:
28. *# define the localhost ip and the port that is going to be used*
29. *# in some future article, we are going to use an env variable instead a hardcoded port*
30. app.run(host=**'0.0.0.0'**, port=os.getenv(**'PORT'**))

**predictor/predictor.py**

1. import pandas as pd
2. import yfinance as yf
3. from datetime import date, timedelta
4. import requests
5. import vars
6. import io
7. def predict(df, target\_column, window):
8. url = **"**<http://broker:8000/md3rw/rw/d3>**"**
9. *# url = "*<http://modelator_d3_rw:6001/rw/d3>*"*
10. params = {**'frame\_size'**: window,
11. **'target\_column'**: target\_column}
12. files = {**'train.csv'**: io.StringIO(df.to\_csv())}
13. response = requests.post(url, data=params, files=files)
14. return response.content.decode(**'utf-8'**)
15. def get\_covid():
16. df = pd.read\_csv(**'**<https://opendata.ecdc.europa.eu/covid19/casedistribution/csv>**'**)
17. df[**'Date'**] = df[**'dateRep'**]
18. df[**'Date'**] = pd.to\_datetime(df.dateRep, format=**'%d/%m/%Y'**)
19. df = df[[**'Date'**, **'deaths'**, **'cases'**]].groupby(**'Date'**).sum()
20. df[**'C\_7'**] = df[**'cases'**].rolling(window=7).mean()
21. df[**'D\_7'**] = df[**'deaths'**].rolling(window=7).mean()
22. return df
23. def get\_gold(commodity, from\_date, to\_date):
24. dst = yf.download(commodity, from\_date, to\_date)
25. dst = dst[[**'Close'**, **'High'**, **'Low'**, **'Volume'**]]
26. dst = dst.dropna()
27. dst[**'S\_3'**] = dst[**'Close'**].rolling(window=3).mean()
28. dst[**'S\_9'**] = dst[**'Close'**].rolling(window=9).mean()
29. dst[**'c\_to\_guess'**] = dst[**'Close'**].shift(-1)
30. return dst
31. def get\_data(commodity, from\_date, to\_date):
32. df = get\_gold(commodity, from\_date, to\_date)
33. *# covid = get\_covid()*
34. *# df = pd.merge(gold, covid, how='left', on='Date').drop\_duplicates()*
35. df.loc[[df.tail(1).index.item()], **'c\_to\_guess'**] = 0
36. df = df.dropna()
37. df[**'Date'**] = df.index
38. df = df.reset\_index(drop=True, inplace=False)
39. return df
40. def make\_predictions(commodity=**'GLD'**):
41. to\_date = date.today().strftime(**"%Y-%m-%d"**)
42. from\_date = **'2020-01-01'**
43. df = get\_data(commodity, from\_date, to\_date)
44. window = 10
45. target\_column = **"c\_to\_guess"**
46. predicted\_df = predict(df, target\_column, window)
47. predicted\_df = io.StringIO(predicted\_df)
48. result = pd.read\_csv(predicted\_df, index\_col=0)
49. return result
50. def get\_prediction\_for(commodity, rows):
51. rsp = getattr(vars, **"df"** + commodity)
52. rows = int(rows)
53. if rows != 0:
54. rsp = rsp.tail(rows)
55. *# return rsp.to\_csv()*
56. return rsp.to\_json(orient=**'index'**)
57. def update\_predictions():
58. vars.update()
59. *# notify frontend with latest data*

**predictor/vars.py**

1. from predictor import make\_predictions
2. def init():
3. global dfGLD
4. dfGLD = make\_predictions()
5. *# dfGLD = 1*
6. def update():
7. global dfGLD
8. dfGLD = make\_predictions()
9. *# dfGLD += 1*

**predictor/Dockerfile**

1. FROM python:3.6
2. RUN mkdir /usr/src/app/
3. COPY . /usr/src/app/
4. WORKDIR /usr/src/app/
5. EXPOSE 5000
6. RUN pip install -r requirements.txt
7. CMD ["python", "-u", "app.py"]

**predictor/requirements.txt**

1. Flaskapscheduler
2. pandas
3. yfinance
4. requests