Универзитет у Београду
Математички факултет

Стефан Петковић

Микросервисна платформа за симболичко израчунавање

Мастер теза

Ментор: **др Филип Марић**

Београд,
2015.

Ментор:

др Филип Марић
Математички факултет
Универзитет у Београду

Чланови комисије:

проф. др Зорица Станимировић
Математички факултет
Универзитет у Београду

мр Јелена Хаци Пурић
Математички факултет
Универзитет у Београду

Датум одбране:

Садржај

Глава 1

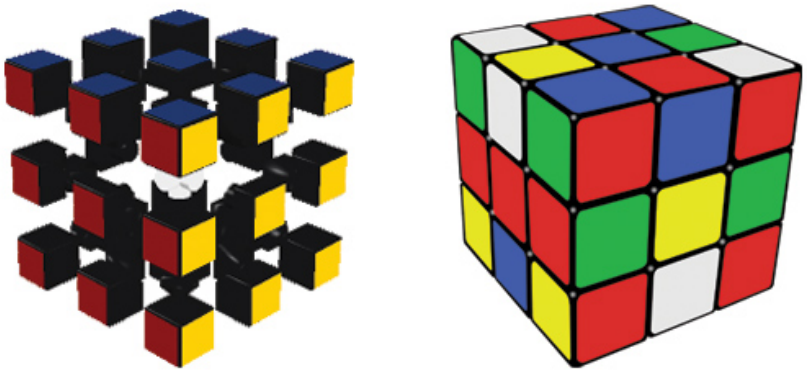
Увод

Микросервисна архитектура *представља софтверско архитектурално готово решење (енг. software architecture design pattern) у ком су комплексне апликације састављене из низа мањих, независних процеса који раде и комуницирају заједно.* ***Рачунарска алгебра*** *је научна област која се односи на проучавање, развој алгоритама и система који баратају математичким изразима.*
Главни циљ рада је представљање микросервисне архитектуре и конкретна имплементације овог приступа на платформи за симболичко израчунавање.

Термин „**микросервисна архитектура**" (енг. microservices) се појавио пре неколико година да би описао посебан начин пројектовања софтверских апликација као скуп сервиса који се могу независно користити. Иако не постоје прецизне дефиниције овакве архитектуре, постоје одређене заједничке карактеристике у погледу: организације, пословне способности, аутоматизације активности, интелигенције и децентрализоване контроле језика и података [1].

Микросервисна архитектура представља приступ развоју једне апликације као скупа мањих сервиса, где сваки ради у засебном процесу и комуницира једноставним механизама, обично коришћењем HTTP протокола, реда порука (7.1) или преко агента порука (7.1). Ови сервиси се изграђују око пословних активности, а потпуно аутоматизована машинерија их може употребљавати независно један од другог. Ови сервиси могу да се развијају у различитим програмским језицима и могу користити разне технологије за складиштење података. Сликовитији приказ микросервисне архитектуре може се видети на слици 1.1, на примеру Рубикове коцке, слика лево

показује низ сервиса издељених у мање сервисе, и слика десно приказује спој свих сервиса који чине једну јединствену целину.



Слика 1.1: Сликовити приказ микросервисне архитектуре у деловима и целини

Рачунарска алгебра *(енг. computer algebra), у математици и информатици позната као* ***симболичко израчунавање*** *(енг. symbolic computation), такође се назива и* ***алгебарско рачунање*** *(енг. algebraic computation).*

У науци и инжењерству, моделирање нас обично доводи до извесних једначина које треба решити. Постоји више типова таквих једначина: диференцијалне, линеарне, полиномијалне, рекурентне итд. Постоје два начина за њихово решавање: приближно и егзактно. Нумеричка анализа је област која пружа веома успешне математичке методе које представљају основ за развој софтвера за добијање приближних решења. *Рачунарска алгебра* је област рачунарства где се математичким методама и рачунарским софтвером долази до егзактних решења.

Зашто уопште користити приближно а не егзактно решење? Одговор лежи у томе да у многим случајевима није могуће наћи егзактно решење, нпр. за неке диференцијалне једначине се може доказати да егзактно решење није могуће одредити.

Питање ефикасности је још важније, нпр. систем линеарних једначина са рационалним коефицијентима се може решити егзактно, у пракси је тешко наћи егзактно решење, посебно у случају система великих димензија. На пример, у метеорологији, нуклеарној физици, геологији и другим наукама често се јављају системи великих димензија чије егзактно решавање захтева значајне ресурсе у временском или меморијском смислу, те се користи приближно решење које се релативно брзо може добити са задовољавајућом тачношћу.

Међутим, у домену егзактних решења, рачунарска алгебра пружа интересантније одговоре за разлику од традиционалних нумеричких метода. Уколико имамо диференцијалну једначину или систем линеарних једначина са параметром t, много више

закључака ћемо извести из оваквог записа него са записом који садржи конкретне вредности параметра t.
Многи не знају да је логаритмар(клизни лењир) био неопходан алат научницима све до 1960-их када су их у кратком временском периоду џепни калкулатори учинили застарелим. У наредним годинама системи рачунарске алгебре ће на сличан начин заменити калкулаторе. Иако су гломазни и скупи, овакви системи могу једноставно да изводе тачну (или произвољну прецизност) аритметику бројевима, матрицама, полиномима итд. Они ће постати незаменљив алат почев од студената, научника и инжињера па до свих радних места. Овакви системи сада постају део разних нумеричких пакета, CAD/CAM(Computer-Aided-Design, Computer-Aided-Manufacturing) система и као неизоставни алат система рачунарске графике [2].

У овом раду је представљена примена микросервиса на платформи за рачунарску алгебру, развијени су сервиси за специфична израчунавања који су писани у различитим програмским језицима који међусобно комуницирају преко посредника (енг. message broker 7.1) као и REST API-а (енг. Representational State Transfer, Application Programming Interface 7.1) за комуникацију са спољним светом. Овај рад илуструје један од начина на који се више различитих технологија и програмских језика могу уклопити у једну целину и да притом раде на веб-у.

Глава 2

Микросервисна архитектура

Микросервисни начин развијања и пројектовања није иновативан; корени оваквог стила почивају на принципима дизајна UNIX система.
Филозофија UNIX-а је следећа:[3]

- **Писати програме који раде једну ствар и раде је добро.**
- **Писати програме који раде заједно.**
- **Писати програме који рукују текстуалним токовима, јер то је универзалан интерфејс(спрега,веза).**

Информатичари који су покренули микросервисну причу су **Мартин Фаулер**[1] и **Џејмс Луис**[2].
Мартин Фаулер је информатичар који је дао велики допринос софтверском инжењерству написавши неколико књига попут: "Рефакторисање" [4], "Архитектурални шаблони за пословне апликације" (енг. Patterns of Enterprise Application Architecture [5]) и један је од аутора агилног манифеста [6]. Мартин Фаулер је заједно са Џејмсом Луисом радио на конкретној имплементацији микросервиса у компанији ThoughtWorks па су своја знања, запажања и искуства о микросервисима исписали у чланку [1].

[1]Мартин Фаулер (енг. Martin Fowler) - рођен 1963, британски софтверски инжењер, аутор и интернационални јавни говорник о развоју софтвера, специјалиста за објектно орјентисани дизајн, УМЛ, агилни развој софтвера и велики заговорник микросервисне идеје.

[2]Џејмс Луис (енг. James Lewis) - главни консултат фирме ThoughtWorks и члан саветодавног одбора Technology Advisory Board. Његова интересовања су углавном изградња софтвера сачињеног од малих компоненти које међусобно сарађују. Радио је на низу пројеката примењујући микросервисну архитектуру и активан је учесник у микросервисној заједници.

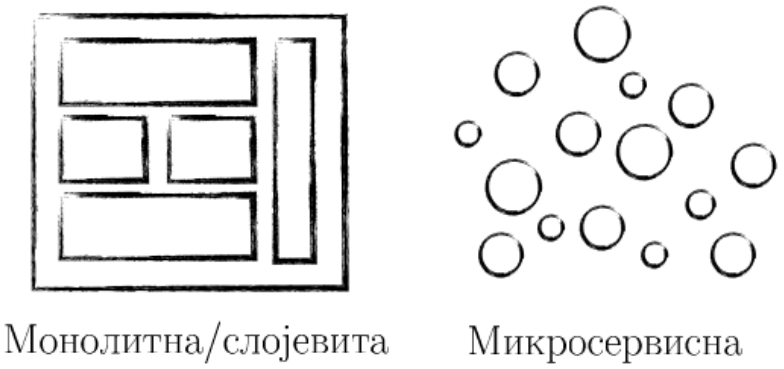
У раду на микросервисној архитектури прикључио им се и **Сем Њуман**[3] који је у фебруару 2015 године издао књигу о микросервисима [7]. Велики део овог поглавља преузет је из [1] укључујући слике и графиконе.

2.1 Поређење монолитне и микросервисне архитектуре

Пословне апликације (енг. Enterprise applications) се често састоје из три главна дела: *корисничког интерфејса* (оно што је на страни клијента), *базе података* и *серверског дела*. Серверска страна прихвата HTTP захтеве, извршава операције над базом података и након тога се генеришу веб-странице писане на језику HTML. Овакав серверски део је *монолитан* - једна логичка извршна целина. Било каква промена овакве целине укључује поновну изградњу (енг. building) и постављање (7.1) нове верзије. Овакав монолитни сервер је природни начин приступања развоја неког оваквог система. Сва логика која извршава захтеве одвија се у једном процесу, дозвољавајући нам да искористимо основне карактеристике језика да рашчланимо апликацију у класе, функције и именске просторе (енг. namespaces).
Аутори Мартин Фаулер и Џејмс Луис сматрају да монолитне апликације могу бити успешне, али и да могу донети пуно проблема пошто се данас све више апликација поставља на облак (енг. cloud). Мале промене доводе до поновне изградње (енг. rebuilding) и постављања читаве монолитне апликације, поготово што промена једног модула унутар апликације може довести и до промене осталих модула који су зависни од промењеног.

Уместо скалирања (7.1) појединих делова који имају потребу за бољом искоришћеношћу, овде се врши скалирање читаве апликације. Овакви проблеми довели су до развоја микросервисне архитектуре, која представља израду апликације као скупа неколико сервиса. Ти сервиси се независно могу развијати и скалирати, а уз то различити сервиси могу бити писани различитим програмским језицима. На сликама 2.1 и 2.2 можемо детаљније видети разлику ове две архитектуре. У наставку рада биће приказана конкретна примена и развој микросервисне архитектуре.

[3] Сем Њуман (енг. Sam Newman) технички консултант у компанији ThoughtWorks



Слика 2.1: Монолитна и микросервисна архитектура

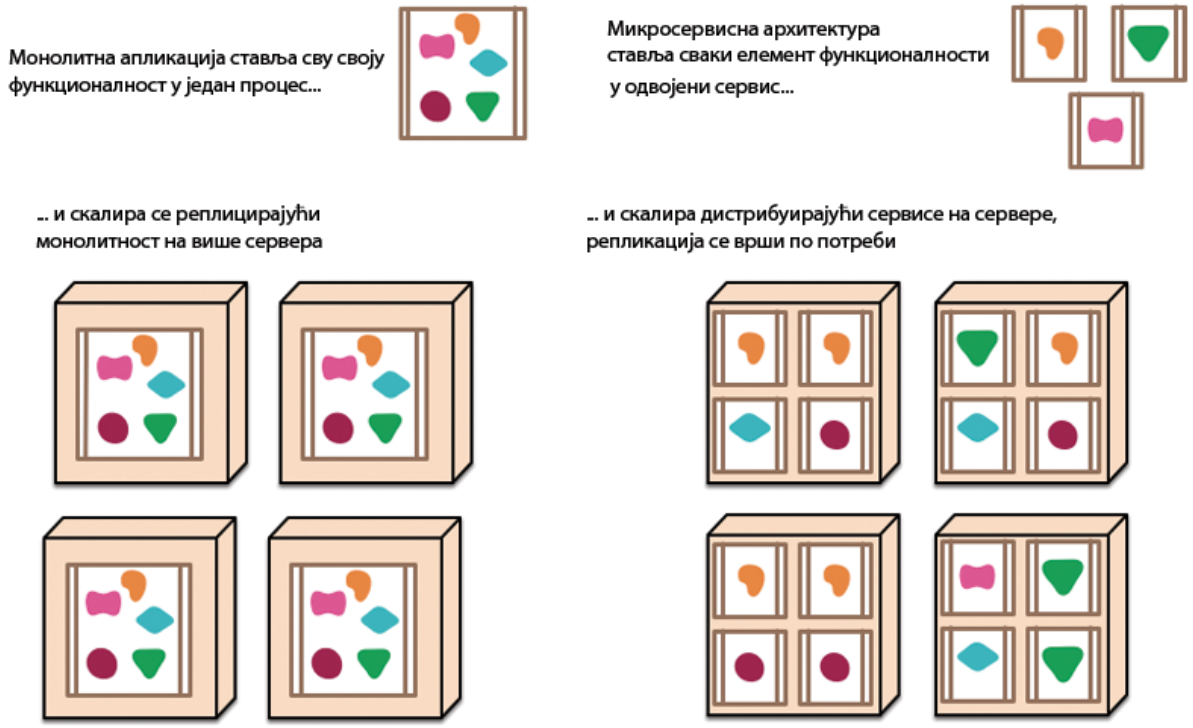
2.2 Карактеристике микросервисне архитектуре

У овом одељку су описане неке од карактеристика које **Мартин Фаулер** и **Џејмс Луис** као заговорници оваквог развоја сматрају најбитнијима, биће истакнута и нека запажања аутора овог рада.

2.2.1 Компонентизација преко сервиса

Компонента је софтверска јединица која је независно заменљива и надоградива. Компонентизација је процес разлагања ресурса у засебне пакете који се могу изнова користити (енг. reusable).
Микросервисне архитектуре користе библиотеке, али примарни начин на који компонентизују софтвер је разлагање у мање сервисе. **Библиотеке** (енг. libraries) дефинишемо као компонете које су повезане у програм и позивају се коришћењем унутар меморијских (енг. in-memory) позива као што су нпр. jar-ови у Јави или gem-ови у Рубију. **Сервиси** ванпроцесне компоненте које комуницирају преко механизама као што су веб-сервисни захтеви или позиви удаљених процедура (енг. remote procedure call).

У платформи за симболичко израчунавање заснованој на микросервисној архитектури која представља основу овог рада као посредник порука (7.1) користи се Apache Kafka (описана у одељку 4.2.3.1), која прима корисничке поруке и прослеђује их сервисима.
Један од главних разлога за коришћење сервиса као компоненти јесте чињеница да се они могу независно користити, односно постављати (енг. independently deployable).



Слика 2.2: Монолитна и микросервисна архитектура

На пример, код апликације која се састоји од више библиотека, у једном процесу било која промена условљава целокупно изграђивање и поновно постављање апликације. Међутим, ако користимо апликацију која је декомпонована у више сервиса и желимо да променимо неке њихове функционалности, довољно је да извршимо промену само једног сервиса и његово постављање. Пракса је да се прво изврши репликација постојећег сервиса, потом изврши исправка и након тога укључивање у систем. Након ових поступака, постојећи сервис се искључује из система.

Циљ добро организоване микросервисне архитектуре је да се минимизација промена кроз кохезивна сервисна ограничења и еволуцију механизама у сервисним уговорима (енг. service contracts).

2.2.2 Организација око пословних процеса

Ако размотримо организацију компанија које се баве развојем софтвера, можемо приметити да архитектура њихових производа умногоме одсликава организацију тимова у оквиру компаније. Компаније се састоје од: специјалиста за кориснички интерфејс (енг. User Interface specialists), логичких специјалиста (енг. middleware specialists) и администратора база података (енг. database administrators). Као последица овакве организације, често долази до тога да разни тимови имплементирају исте компоненте. Из овакве организације је формулисан својеврсни „Конвејев закон" [8]:

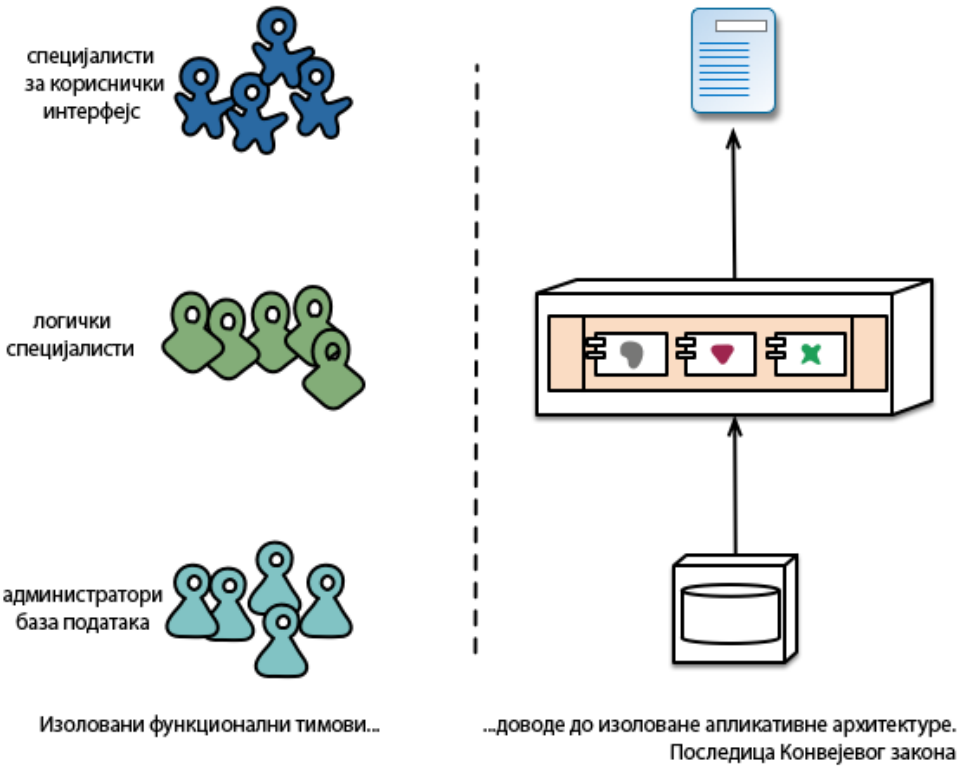
> "*Свака организација која дизајнира неки систем ће произвести дизајн чија структура представља копију комуникацијске структуре те компаније.*"

На основу ових искустава и сазнања, микросервисни приступ поделе задатака је другачији. Он раздваја задатке у сервисе који се организују око пословних способности[4]. Приказ овакве организације приказан је на слици 2.3.

Микросервисни приступ је такав да се тимови организују попут сервиса: велики тимови ће бити замењени мањим сервисним тимовима чији ће састав чинити по неколико чланова специјалиста за графички кориснички интерфејс, логичких специјалиста, администратора база података и др. У зависности од потребе тимове не морају да чине запослени свих специјалности. Тим се, на пример, може састојати од логичких специјалиста и администратора база података. Организација и структура тимова зависи од потреба пројекта и израде сервиса. Слика 2.4 приказује организацију микросервисних тимова.
Унакрсно функционални тимови су одговорни за израду и функционисање сваког производа. Сваки производ се дели у низ индивидуалних сервиса који комуницирају преко посредника.

Низ сервиса може да комуницира преко посредника, као што и корисник може да комуницира са сервисима преко REST API-ја који комунцира са посредником.

[4](енг. bussiness capability) начин репрезентовања високог концептуалног погледа на велике апликације



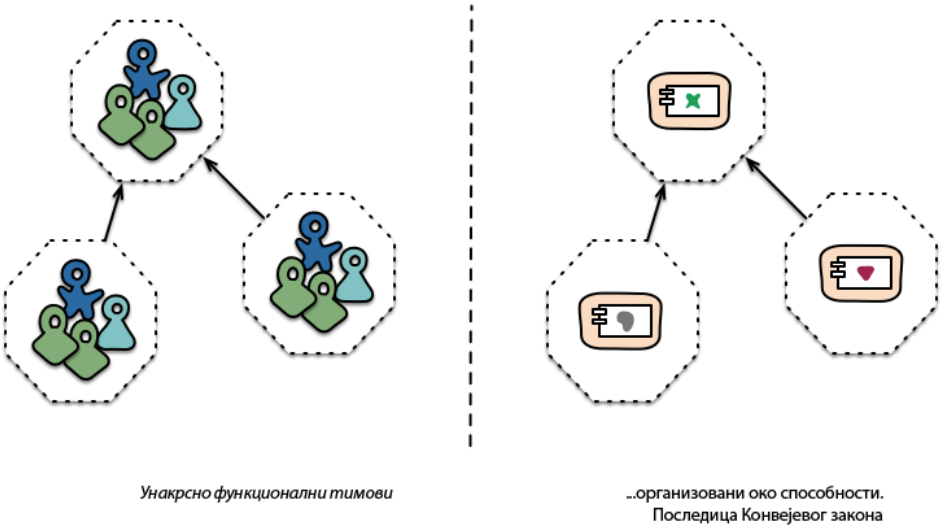
Слика 2.3: Пример огранизације компаније и архитектуре њених производа

2.2.3 Производи, не пројекти

За разлику од стандардних пројектних модела чији је циљ да испоруче неки део софтвера за који се сматра да је комплетиран и након завршетка бива предат на одржавање, заговорници микросервиса теже да избегну овакав модел. Они предлажу да тим треба да буде власник пројекта током његовог циклуса постојања. Инспирација за овако нешто је проистекла из појма компаније Амазон *„ти изграђујеш, ти покрећеш”*, где развојни тим преузима пуну одговорност за софтвер који је постављен (7.1) на сервере и који се може користити.

Овакав приступ пружа развојном тиму свакодневни контакт, као и сазнања о томе како се њихов софтвер понаша у продукцији, и повећава контакте са њиховим корисницима, јер на себе морају преузети један део терета који корисничка подршка носи са собом.

Својство софтверског производа је да се везује за пословне способности, уместо гледања на софтвер као на скуп функционалности које треба да буду завршене, у самом току постоји однос где се стално поставља питање како софтвер може да помогне својим корисницима у побољшавању њихових пословних способности. Овакав однос



Слика 2.4: Организација компаније у микросервисном стилу

се може применити и на монолитну архитектуру. Међутим, мања грануларност сервиса може утицати на лакше креирање везе између програмера са једне и корисника са друге стране.

2.2.4 Међусервисна комуникација

У многим производима, приликом изградње комуникацијске структуре између различитих процеса, углавном се користи приступ да се одређена логика поставља у сами комуникациони механизам. Добар пример оваквог приступа је ESB (енг. Enterpise Service Bus 7.1), чији модели често укључују софистициране механизме за рутирање порука, трансформацију и примену пословних правила.

Микросервисна заједница фаворизује алтернативни приступ : „паметни крајеви, глупи канали". Апликације које се изграђују у микросервисном стилу теже да буду рашчлањене и кохезивне у што је могуће већој мери. Свака апликација има карактеристична, одређена својства која се односе на примање захтева, примењивање логике на одређени начин и производње захтева. Ово се организује коришћењем једноставног REST протокола. Два протокола која се најчешће примењују су HTTP одговор-захтев са

удаљеним API-ијем као ресурсом и методом једноставнх порука. Најбољу дефиницију овог приступа налазимо у речима Ијана Робинсона: [5] „*Будите на вебу, не иза веба*". Микросервисни тимови користе принципе и протоколе на којима је веб заснован.

За други приступ се користи размена преко једноставне магистрале порука (енг. message bus). Изабрана инфраструктура је значајно упрошћена; једноставне имплементације попут (RabbitMQ 7.1) или (ZeroMQ 7.1) углавном пружају само поуздану асинхрону фабрикацију порука, док је сва логика смештена унутар самих сервиса. Сличан принцип функционисања има и Apache Kafka која пружа више могућности, која је коришћена на "Микросервисној платформи за симболичко израчунавање" и о којој ће бити више речи у одељку 4.2.3.1.

2.2.5 Децентрализовано управљање

Раздвајањем монолитних компонената у сервисе добијамо различите могућности за изградњу сваког од њих. Сервиси могу бити писани у различитим програмским језицима. Овај рад показује да језици попут C-a, Рубија, Јаве, JavaScript-a и многих других могу заједно радити и чинити једну целину. Такође, сваки од сервиса може имати своју базу података, нпр. један сервис може користити MySQL, други неку од Oracle или Microsoft-ових база података. Тимови који раде на изградњи микросервиса воле другачији приступ стандардима. Уместо да користе дефинисани скуп стандарда који је записан негде на папиру, они преферирају да праве корисне алате који други тимови могу користити за решавање сличних проблема. Неки од оваквих алата попут Kafka-node, Librdkafka, Poseidon, Kafka-python који су развијени од стране микросервисне заједнице коришћени су на изградњи "Микросервисне платформе за симболичко израчунавање", о овим алатима биће више речи у поглављу 5. Netflix[6] је једна од компанија чија организација потпуно прати микросервисну филозофију, делећи своја сазнања, алате, библотеке и др.

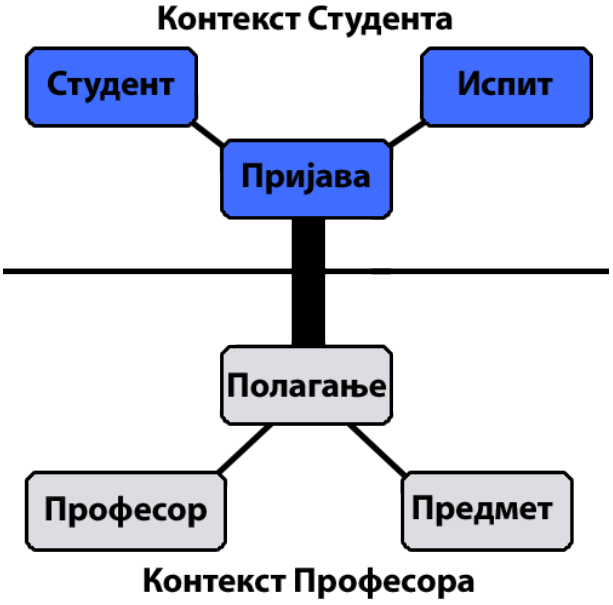
[5] Ијан Робинсон (енг. Ian Robinson) - директор корисничког успеха у компанији Neo Technology, која стоји иза компаније Neo4j, светске водеће компаније за отворени развој графовских база података, уједно је и аутор неколико уџбеника о REST-u.

[6] Netflix Inc. - компанија која пружа услуге приказивања видео садржаја на захтев

2.2.6 Децентрализовано управљање подацима

Децентрализовано управљање подацима може бити представљено на више различитих начина. Са становишта највишег апстрактног нивоа, то значи да ће се конецептуални поглед модела разликовати од система до система. Ово је чест проблем када се врши интеграција пословног система (енг. enterprise system) јер се поглед клијента се разликује од погледа тима који пружа подршку. Неке ствари које представљају клијенте са аспекта продајног тима се можда неће уопште појавити код тима за подршку. Сличан пример је сервис Математичког факултета - Hypatia[7] која има два различита погледа за студенте и професоре.

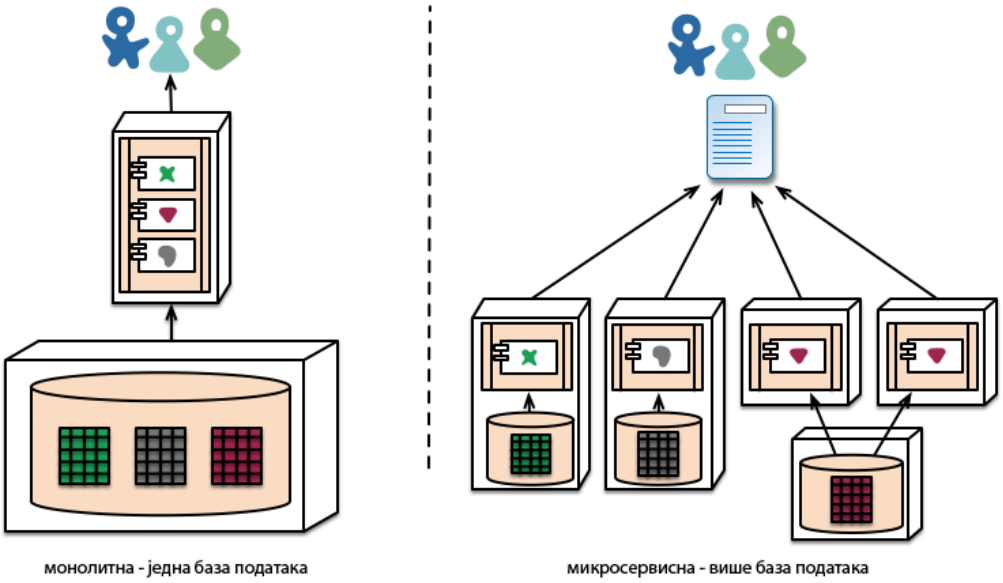
Овакав проблем се може појавити и унутар апликација, посебно када су апликације рашчлањене на одвојене компоненте. Користан начин за решавање оваквих проблема је принцип дизајна вођеног доменом (енг. Domain-Driven Design 7.1) као и појам Ограниченог контекста (енг. Bounded Context)[10]. Принцип дизајна вођеног доменом раздваја комплексан домен у више ограничених контекста и осликава односе између њих. Овај процес је користан како за монолитне, тако и за микросервисне архитектуре, али постоји природна корелација између сервиса и контекстних граница. Гранични контекст се управо односи на поделу погледа, само се односи на конкретну поделу модела у бази података. Више о Принципу дизајна вођеног доменом може се наћи у [11]. Осим што микросервиси децентрализују одлуке о концептуалном моделу,



Слика 2.5: Пример ограниченог контекста

[7] Hypatija - студентски сервис Математичког факултета[9]

они децентрализују и одлуке о складиштењу података. Док монолитне апликације преферирају рад са једном базом података, микросервисне могу радити са различитим базама (видети подсекцију 2.4). Микросервиси преферирају да сваки сервис управља својом базом података, било да су то различите инстанце исте технологије базе података, било да је то сасвим други систем базе података, овакав приступ се назива Polyglot Persistence. Илустрација разлика монолитне и микросервисне архитектуре дата је на слици 2.6.



Слика 2.6: Монолитна и микросервисна архитектура

2.2.7 Инфраструктурна аутоматизација

Технике инфраструктурне аутоматизације су остариле огроман напредак током последњих пар година. Еволуција облака (енг. cloud) и Амазонових Веб сервиса (енг. Amazon Web Service) је нарочито смањила број оперативних комплексности - изградње (енг. building), постављања (енг. deploy) и функционисања микросервиса. Многе производе и системе који се граде на основу микросервиса изграђују тимови са великим искуством у непрекидној испоруци (енг. Continous Delivery 7.1) и непрекидној интеграцији (енг. Continous Integration 7.1). Тимови који праве софтвер на овакав начин, широко употребљавају технике инфраструктурне аутоматизације, што је илустровано на слици 2.7. Више о сталној испоруци може се прочитати у [12]. С обзиром да желимо да будемо у највећој могућој мери сигурни да наш производ поуздано ради, изводимо бројне **аутоматске тестове**. Још једна област где тимови

Слика 2.7: Монолитна и микросервисна архитектура

користе обимну инфраструктурну аутоматизацију је управљање микросервисима у фази употребе.

2.2.7.1 Docker

Докер (енг. Docker) [13] је пројекат отвореног кода који аутоматизује постављање апликације унутар софтверских контејнера[8], пружајући додатни слој апстракције и аутоматизације виртуализационог нивоа оперативног система на Линуксу. Докер користи карактеристике ресурсне изолованости Линукс језгра попут cgroups и језгарних именских простора (енг. namespaces) како би дозволили независним контејнерима да се покрећу унутар једне Линкус инстанце, избегавајући тако додатне трошкове стартовања виртуелних машина. Именски простори Линукс језгра комплетно изолују поглед на оперативно окружење, укључујући процесна стабла, мрежу, корисичку идентификацију и прикључене системе датотека једне апликације. Док cgroup-е омогућавају изолацију ресурса, укључујући процесор, меморију, блокирање улаза/излаза и мреже. Од верзије 0.9 Докер укључује libcontainer библиотеку као његов сопствени начин за директно коришћење виртуелизационих могућности које обезбеђује Линукс језгро.

Докер имплементира API високог нивоа како би створио услове контејнерима да покрећу процесе у изолацији. Докер контејнер, за разлику од традиционалних виртуалних машина, не захтева или не укључује одвојени оперативни систем. Уместо

[8]Контејнери представљају ниво виртуализације оперативног система, а то је у ствари серверски виртуализациони метод где језгро оперативног система дозвољава извршавање више изоловнаних инстанци у корисничком простору, уместо извршавања само једне инстанце, у виду програма, процеса или сервиса. Контејнери су виртуализациони механизми (енг. virtualization engines), виртуелни приватни сервери (енг. virtual private servers) који могу изгледати као прави сервери од једне до друге тачке гледишта њихових власника и корисника.

тога, он се ослања на функционалност језгра и користи ресурсну изолацију и одвојене именске просторе како би комплетно изоловао поглед апликације на оперативни систем.

Докер приступа виртуализационим опцијама Линукс језгра директно преко обезбеђене libcontainer библиотеке или директно преко libvirt, LXC(Linux Containters) или systemnspawn.

Користећи контејнере, ресурси могу бити изоловани, сервиси ограничени и процеси снабдевени приватним погледом на оперативни систем преко сопствене процесне идентификације, структуре фајл система и мрежних интерфејса. Више контејнера могу да деле исто језгро, али сваки контејнер може бити ограничен на коришћење дефинисане количине ресурса попут: процесора, меморије и улаза/излаза.
Користећи Докер за креирање и управљање контејнерима, олакшава се стварање високо дистрибуираних система, при чему је дозвољено да се више апликација, радних задатака и других процеса покрећу аутономно на једној физичкој машини или преко спектра виртуелних машина. Ово омогућава постављање чворова као ресурса који су доступни или потребни, чиме се обезбеђује платформа као сервис (енг. Platform as as Service - PaaS 7.1) у постављању и скалирању за системе попут Апачи Касандре (eng. Apache Cassandra), MongoDB-a (7.1) или Riak-a (7.1) као и поједностављење стварања и рада радних или радно оптерећених редова и других дистрибуираних система.
Докер подржава интеграцију са разним инфраструктурним алатима, као што су: Amazon Web Service, Ansible, CFEngine, Chef, Google Cloud Platform-e, Jenkins-a, Microsoft Azure, OpenStack Nova, OpenSVC-a, Puppet-a, Salt-a, и Vagrant-a [14] [15].
Контејнерска технологија попут Докера пружа идеално окружење за рад са микросервисном архитектуром, креирању мањих сервиса и са потребом за лаким механизмима, који се могу независно постављати, скалирати и портовати.
Ури Коен[9] описује Докер као одлично окружење за микросервисе које изолује контејнере на један процес или сервис. Овакав приступ стављања процеса или сервиса у контејнер чини ове сервисе врло једноставним за коришћење и ажурирање. Зато и не изненађује чињеница да је Докер једна од најкоришћенијих платформи за развој. Kubernetes је пројекат отвореног кода који је дизајниран за развој микросервиса који проширује Докерове способности. Описивањем карактеристика слике Kubernetes може поставити и управљати са неколико Докер контејнера истог типа.

[9]Ури Коен (енг. Uri Cohen) - шеф одсека за управљање производима у компанији GigaSpaces

Празнину у функционалности која настаје коришћењем Kubernetes-а код управљања пројекта који има више нивоа може попунити TOSCA[10] шематски план [16].

2.2.8 Дизајн за неуспех

Последица коришћења сервиса као компоненти јесте да апликације морају бити дизајниране тако да могу да толеришу грешке и падове сервиса. Сваки сервисни позив може бити неуспешан услед грешке или недоступности сервиса и клијенту је потребно одговорити што пре уколико дође до оваквих грешака. Овај недостатак који микросервиси имају у односу на монолитну архитектуру је утицао на то да микросервисни тимови константно преиспитују како сервисни падови утичу на корисничка искуства. Пракса Netflix-а је да сами изазивају неуспехе на сервисима и центрима података (енг. datacenters) како би тестирали флексибилност и надгледање апликације у виду праћења рада сервиса и детектовања потенцијалних проблема.
Пошто је могуће да сервиси падну у било које време, веома је важно брзо детектовати кварове и аутоматски их отклонити. Микросервисне апликације стављају велики акценат на праћење апликације, проверавајући архитектуралне (колико захтева по секунди прима база података) и пословне елементе система(колико се добија наруџбина по минуту). Семантичко посматрање може да пружи рано упозорење да нешто није у реду и да алармира развојне тимове да погледају о каквој грешци се ради. Микросервисна архитектура пружа низ погодности за одбрану од разних врста падова. На пример, уколико неки сервис не ради из неких разлога, не морамо дубље залазити у проблем, већ примењујемо најједноставније могуће решење, односно гасимо постојећи сервис који прави проблем и дижемо нову сервисну инстанцу. Такође, посредници пружају разне врсте механизама за праћење система и опоравка од било каквих врста падова. Нпр. Apache Kafka има механизам репликације својих података на више сервера, тако да у случају пада једног сервера, на његово место долази други који преузима његов посао.

[10]TOSCA (Topology and Orchestration Specification for Cloud Applications) - је OASIS(Организација за напредно структуиране информационе стандарде) стандардни језик који описује топологију веб сервиса заснованих на облаку, њихових компоненти, односа и процеса који управљају њима.

2.2.9 Еволутивни дизајн

Сваки пут када покушамо да рашчланимо софтверски систем у компоненте, суочавамо се са питањем како да раздвојимо целину у делове, односно, који су то принципи на које можемо да се ослонимо како бисмо поделили нашу апликацију. Кључна особина компоненте је појам независне замене и надоградње, што доводи да се компонента погледа из другог угла како би уочили начин на који можемо презаписати компоненту без утицаја на остале сараднике(компоненте).

Сајт Гардијана[11] је добар пример апликације која је осмишљена и направљена као монолитна, али која је еволуирала у микросервисном смеру. Монолитни стил је и даље језгро вебсајта, али аутори веб-сајта су одлучили да додају нова својства градећи микросервисе који користе монолитни API. Овакав приступ је веома згодан за неке привремене опције попут специјализованих страница за праћење неког спортског догађаја. Овакав део вебсајта може брзо да се направи користећи језике који омогућавају бржи развој и може се брзо уклонити када се догађај заврши.

Овај нагласак на заменљивости је специјалан случај општег принципа модуларног дизајна, који води модуларност кроз шаблон промена[12]. Потребно је задржати ствари које се мењају у истом тренутку и у истом модулу. Делови система који се не мењају ретко треба да буду у другим сервисима од оних који се тренутно мењају и оних који трпе промене. Уколико у више наврата установимо да се два сервиса мењају заједно, то је знак да сервиси требају да буду спојени.

Постављање компоненте у сервисе пружа могућност за планирање и прављење учесталијих издања. Коришћење монолитне архитектуре, било која промена изискује пуно изграђивање и постављање целокупне апликације. Употребом микросервиса, потребно је само да разместимо сервисе које смо модификовали. Ово може да упрости и убрза процес поновне употребе. Лоша страна оваквог приступа је да морамо бринути да промена једног сервиса не утиче на корисника. Традиционални интеграциони приступ је да покушамо да користимо верзионирање, али наклоност у микросервисном свету је да верзионирање користимо само као последњу опцију, више о овоме можете наћи на [18]. Можемо да избегнемо много верзионирања дизајнирањем сервиса који су у највећој могућој мери толерантни у односу на промене добављача.

О искуствима великих компанија попут:

[11] Guardian - Британске националне дневне новине, theguardian.com

[12] Кент Бек - у свом делу *Implementation Patterns*[17] наглашава овај шаблон као један од својих главних принципа

Netflix, Amazon, The Guardian, the UK Government Digital Service, realestate.com.au, Forward и comparethemarket.com и примени микросервисне архитектуре на ниховим системима може се наћи на [19][15]. Детаљи о микросервисима могу се пронаћи у [1] [7].

Глава 3

Рачунарска алгебра - симболичко израчунавање

3.1 Појам

Математичари моделују природни феномен превођењем експерименталних резултата и теоријског концепта у математичке изразе који садрже бројеве, променљиве, функције и релације. Потом, користећи прихваћене методе математичког резоновања, овим изразима се пажљиво барата или се трансформишу у друге изразе који откривају нова сазнања о овом феномену. Овакав математички приступ разумевању света је важна компонента научних метода у физици који датирају још од времена Галилеја[1] и Декарта[2]. Пратећи кораке ових научника, Исак Њутн[3] користи овакав приступ за формулисање аксиоматског, квантитативног описа кретања објеката. Уз помоћ математичког резоновања, он је открио универзалан закон гравитације и извео још закона који описују кретање плима, осека и орбите планета. Тако је настала наука коју називамо механика, а техника за манипулисање и трансформисање математичких израза је чврсто успостављена као важно средство за откривање нових знања о физичком свету.

У протеклих педесет година рачунар је постао незаменљиво експериментално средство које умногоме увећава нашу способност у решавању математичких проблема.

[1]Галилео Галилеј (15.02.1564 - 08.01.1642)- италијански астроном, физичар, математичар и филозоф.

[2]Рене Декарт (31.03.1596 - 11.02.1650) - француски математичар, филозоф и научник

[3]Исак Њутн (04.01.1643 - 31.03.1527) - енглески физичар, математичар, астроном, алхемичар и филозоф

Математичари користе рачунаре за добијање нумеричких и графичких решења проблема које је исувише тешко или чак немогуће решити експлицитно. Али рачунар није само апарат за нумеричка израчунавања. У принципу, на најнижем нивоу рачунари баратају само нулама и јединицама према добро дефинисаним правилима. Природно је запитати се који су још делови (процеса) математичког резоновања погодни за имплементацију. Наравно, не можемо очекивати од рачунара да формулише аксиоме механике као Њутн и да ни из чега изведе важне теоријске закључке. Међутим, један део процеса математичког резоновања, механичке манипулације и анализе математичких израза је алгоритамски. Данас постоји низ рачунарских програма који рутински поједностављују алгебарске изразе, интегришу компликоване функције, проналазе тачна решења диференцијалних једначина и врше мноштво других операција на које се наилази у примењеној математици, науци и инжењерству [2][20][21].

Рачунарски алгебарски систем (енг. Computer Algebra System, скраћено CAS) је рачунарски програм који врши симболичка израчунавања. Наша платформа је базирана на симболичким библиотекама SymPy и Symja које су имплементиране у сервисе као делове микросервисне архитектуралне платформе. Детаљније о овим библиотекама можете прочитати у поглављу 4, као и у [22][23][24].

Системи рачунарске алгебре се деле у две класе: специјализоване и оне опште намене. Специјализовни су посвећени посебним деловима математике, попут *теорије бројева, теорије група* или *подучавања основа математике*. Системи рачунарске алгебре опште намене имају за циљ да буду корисни за рад и примену у различитим научним областима где се манипулише математичким изразима [25]. Систем рачунарске алгебре опште намене мора садржати различите елеменате као што су:

- *кориснички интерфејс* - унос и приказ математичких израза;
- *програмски језик* и *преводилац* - резултат израчунавања обично има непредвидљиву форму и величину, стога је корисничка интервенција често неопходна;
- *упрошћивач* - систем за презаписивање и упрошћавање математичких израза;
- *систем за управљање меморијом* који укључује сакупљање смећа (енг. garbage collector) - потребан због величине привремених података који се могу појавити током израчунавања;
- *произвољно прецизна аритметика (енг. arbitrary-precision arithmetic)* - потребна због великих целих бројева који се могу појавити током израчунавања;

- *велика библиотека математичких алгоритама.*

3.2 Преглед постојећих алата

Један од најпознатијих програмских пакета за нумеричка и симболичка израчунавања јесте MATLAB. MATLAB се махом користи у инжењерству, за нумеричка израчунавања, помаже при различитим симулацијама система и помоћу њега се лако добијају графички прикази резултата. Међутим, овај алат није бесплатан, његова цена у зависности од потреба може достићи износ од две, три хиљаде долара, стога су се развили и други алати попут следећих софтвера:

- Mathematica - софтвер за израчунавање који се користи у научним, инжењерским, математичким и рачунарским пољима, заснован на симболичкој математици. Написан у програмским језицима Wolfram, C/C++, Java и Mathematica.
- GNU Octave је програмски језик високог нивоа примарно намењен за нумеричка израчунавања. Пружа интерфејс преко командне линије за решавање линеарних и нелинеарних математичких проблема, извођење других нумеричких експеримената који су углавном компатибилни са MATLAB-ом. Написан комплетно у C++-у.
- Scilab - је платформа високог нивоа, отвореног кода за нумеричка израчунавања, процесирање сигнала, статистичку анализу, обраду дигиталних фотографија, симулацију динамике флуида, нумеричку оптимизацију, моделирање, симулацију експлицитних или имплицитних динамичких система и симболичку манипулацију. Написан у C, C++, Java и Scilab-у.
- Maxima - систем рачунарске алгебре, писан у Лисп-у, доступан за све платформе.
- Maple - комерцијални систем рачунарске алгебре, написан у језицима C и Java. Архитектура Maple-а је базирана на малом језгу који је имплементиран у C-у, а који пружа језик Maple-а. Већина функционалности долази из библиотека која долазе из различитих извора, а већина библиотека имплементирана је у Maple језику, док су стандардни интерфејси и калкулаторски интерфејс имплементирани у Java-и. Развијен од стране компаније Maplesoft.

- Sage - (енг. System for Algebra and Geometry Experimentation), математички софтвер са карактеристикама које покривају много аспеката математике, укључујући алгебру, нумеричку математику, теорију бројева и инфинитезималног рачуна. Написана у Пајтону.
- Singular - систем рачунарске алгебре за полиномијална израчунавања са посебним нагласком на потребе комутативне алгебре и теорије прстена, теорије сингуларитета и алгебарске геометрије. Развијен на Универзитету у Кајзерслаутерну, написан у C++-у
- Yacas - Yet Another Computer Algebra System, програм са симболичку манипулацију математичких израза. Потпуно написан у C++-у.

Поред ових набројаних постоји још алата који се баве симболички израчунавањем, библиотека SymPy која је описана у одељку 4.2.4.2 као и библиотека Symja која је описана у одељку 4.2.4.4. Оно што је заједничко за овакве алате и платформе јесте да је њихова архитектура монолитна, у одељку 2.1, дали смо аргументе зашто је овакав вид архитектуре умногоме не толико добар. Наведени аргументи су били један од главних мотива за изградњу "микросервисне платформе за симболичко израчунавање".

Глава 4

Технологије

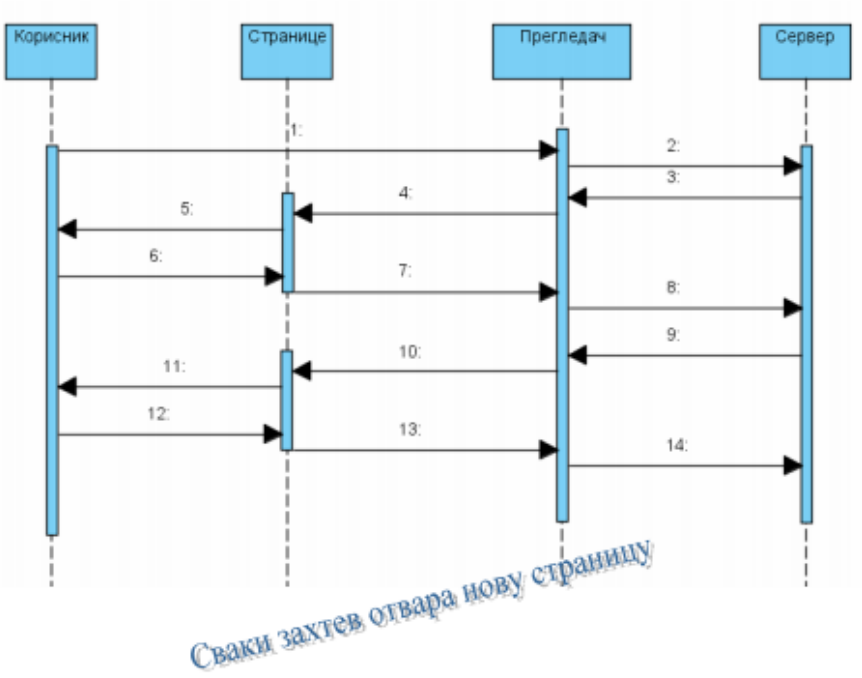
У овом поглављу биће представљене све технологије које су коришћене за изградњу микросервисне платформе. Сама архитектура и начин рада ових технологија биће представљени у поглављу 5.

4.1 Клијентска страна

Клијентска страна се односи на операције које врши клијент у односу на серверску страну у оквиру рачунарске мреже. Клијент је у овом случају прегледач преко ког корисник врши слање захтева ка серверској страни. Прегледач се покреће на локалном рачунару или радној станици клијента и повезује на сервер по потреби. Операције се могу извршавати на клијентској страни јер захтевају приступ информацијама или функционалностима које су доступне на клијенту, али не на серверу. Разлог је јер корисник треба да их посматра или унесе податке или је серверска страна слабија и није у могућности да одговори на све клијентске захтеве благовремено. Осим тога, ако се операције могу изводити на клијентској страни, без слања података преко мреже, могу одузети мање времена, користити мање протока и сносити мањи безбедносни ризик. Клијентску страну имплементиране платформе чине: AngularJS, Bootstrap и Katex, о којима ће детаљиније бити речи у следећим одељцима.

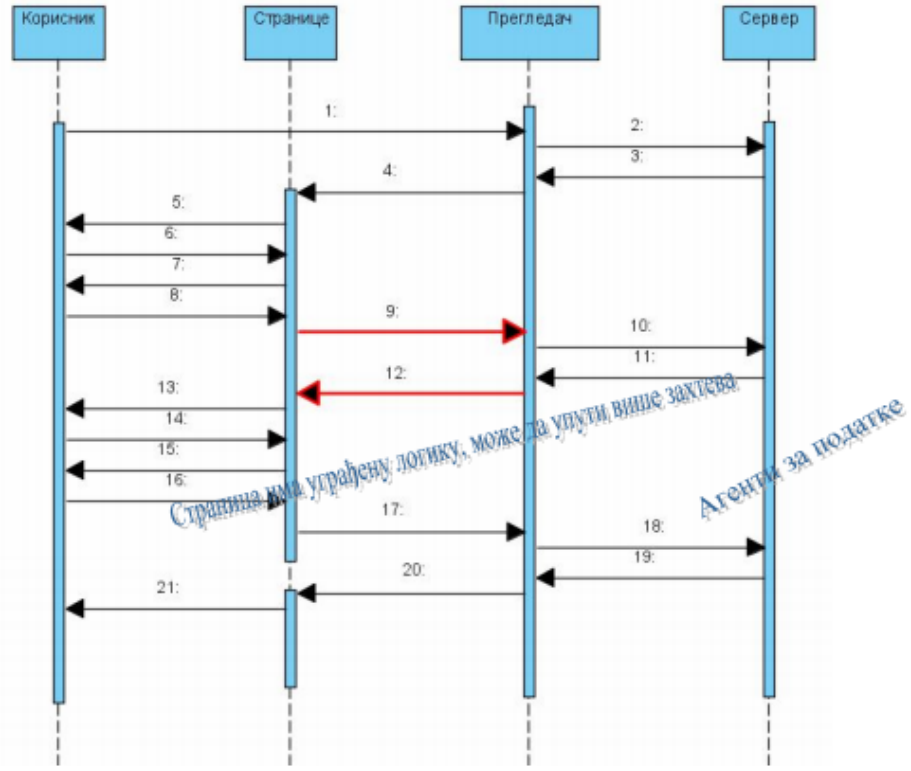
Клијент-сервер архитектура Клијент-сервер архитектура ради тако што сервер на основу података и метаподатака припрема и шаље клијенту презентацију (припремљену веб страницу). Пример овакве архитектуре представља прављење странице

на страни сервера (ASP.NET или PHP) или прослеђивање аплета и скриптова од стране сервера. Карактеристике овакве архитектуре су високо оптерећење сервера (припреме презентације се врши на серверу и свака активност корисника завршава на серверу) и ниско оптерећење клијента (идеално за танке клијенте и практично без неке велике обраде података). Елементи ове архитектуре су: веб прегледач, страница, веб-сервер и генератор садржаја (различити програми на страни клијента) [26].



Слика 4.1: УМЛ приказ рада Клијент-сервер архитектуре

Архитектура танких и тешких клијената Сервер шаље клијенту податке и метаподатке и препушта му да самостално припреми презентацију (сервер обично шаље и програме, али независно од конкретних података). Пример овакве архитектуре су веб локације које интензивно користе JavaScript и plug-in прегледача, нпр. PDF. Неке од карактеристика су значајно смањено оптерећење сервера (сервер је ослобођен од послова припреме презентације) и велико оптерећење клијената (програми за припрему презентације се извршавају на страни клијента, најчешће у облику извршавања скриптова у прегледачу). Елементи овакве архитектуре су: веб прегледач, страница, веб-сервер, агент за податке. Програми који праве динамичке садржаје на основу расположивих података и програми који изводе одговарајуће трансакције на расположивим подацима [26].



Слика 4.2: УМЛ приказ рада архитектуре танких и тешких клијената

4.1.1 AngularJS

AngularJS је платформа за развој веб апликација отвореног кода, који одржава компанија Гугл и заједница индивидуалних програмера и корпорација који решавају многе изазове на које се наилази у развијању једностраничне апликације. Има за циљ да упрости развој и тестирање апликација пружајући платформу за клијентску страну модел-поглед-контролер (енг. model-view-controller) архитектуре, упоредно са компонентама које се користе у развоју богатих интернет апликација.

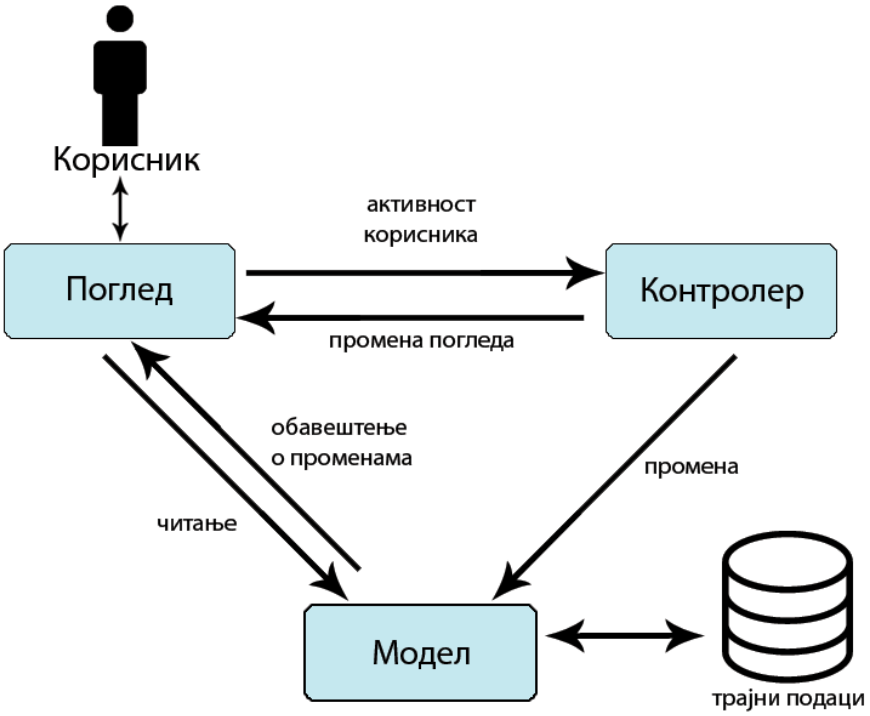
Библиотека ради тако што прво учитава HTML страну, у којој су елементи обележени додатним атрибутима. Овакви атрибути се интерпретирају као директиве које налажу Angular-у да веже улазне или излазне делове стране у модел који се репрезентује стандардним JavaScript променљивама. Вредности ових JavaScript променљивих могу се ручно поставити у оквиру кода или добити из статичног или динамичног JSON[1] ресурса [27].

[1]JSON (Java Script Object Notation) - отворени стандардни формат који користи лако читљив текст за слање објеката који се састоје од парова кључ-вредност.

4.1.1.1 Модел-поглед-контролер

AngularJS је заснован на архитектури Модел-поглед-контролер (енг. Model–view –controller, скр. MVC), која представља специјализацију раздвајања презентације и садржаја, тзв „кружно раздвајање".
Концепт модел-поглед-контролер састоји се из следећих елемената:

- **Модел** је интерна репрезентација података и пословне логике. Он обухвата објекте и податке домена, не зна ништа о корисничком интерфејсу јер тежи раздвајању презентације и садржаја. Захтеве за промене прима од контролера и обавештава поглед о променама, док захтеве за читање прима од погледа.
- **Поглед** је имплементација корисничког интерфејса која обухвата све елементе интерфејса и може непосредно да чита податке из модела.
- **Контролер** је логичка компонента корисничког интерфејса који прима обавештења о активностима корисника од погледа и иницира активност на нивоу модела и промене на погледу [28][5]. Приказ МПК дат је на слици 4.3.



Слика 4.3: Архитектура Модел-поглед-контролер

Више о МПК архитектури може се пронаћи у [28][5][29][17].

Модел-поглед-контролер AngularJS-а представљају:

- **модел** (подаци - својства JavaScript објеката),
- **поглед** (HTML стране/DOM дрво),
- **контролер** (JavaScript функције).

4.1.1.2 Појмовни преглед

У овом одељку укратко ћемо се осврнути на важне појмове AngluarJS-а.

- **Шаблон** - .html страна у којој је направљен простор за уметање података из модела, обично су ови делови обележени са {{ }} или ng-bind.
- **Директиве** - функције које извршавамо над DOM дрветом да бисмо реализовали неку функционалност.
  - ng-app: указује Angular-у о ком делу DOM дрвета треба да води рачуна; када се страна учита Angular обилази DOM дрво почевши од чвора који садржи ову директиву и надаље придружује функционалности свим чворовима за које пронађе **ng** директиве унутар њега
  - ng-controller: JavaScript функције задужене за поједине делове DOM дрвета
  - ng-model: омогућава везивање између шаблона и података
  - ng-repeat: омогућава понављање делова DOM дрвета
  - ng-init: омогућава иницијализацију
  - ng-class, ng-class-even, ng-class-odd, ng-show: неке од директива за контролу CSS својства
  - ng-click, ng-mouseup, ng-change, ng-focus, ng-blur, ng-keyup: својства JavaScript догађаја
  - ng-required, ng-change, ng-maxlength, ng-form, ng-submit: неке од директива које се користе у раду са формама
- **Модел** - представља податке који се приказују кориснику у погледу и са којима корисник врши интеракцију

- **Домен (енг. Scope)** - представља механизам комуникације између компоненти, {{ }} је начин везивања променљиве у шаблону (погледу) или ng-bind-y
- **Изрази** - приступају променљивама и функцијама из домена
- **Филтери** - форматирају вредност израза за приказивање како би био приказан кориснику
- **Поглед** - представља оно што види корисник, DOM
- **Везивање података (енг. Data Binding)** - врши синхронизацију података између модела и погледа
- **Контролер** - представља пословну логику иза погледа
- **Убризгавање зависности** - 7.1 креира и везује објекте и функције
- **Убризгач** - контејнер за убризгавање зависности
- **Модул** - контејнер за различите делове једне апликације укључујући контролере, сервисе, филтере, директиве који конфигурише убризгач
- **Сервис** - пословна логика независна од погледа, која се изнова може користити

4.1.2 Здраво свете

Пример „Здраво свете" коришћењем AngularJS-a.

Пример кода који врши преусмеравање на неки од шаблона (енг. templates) и врши везивање шаблона и контролера приказан је у скрипти 4.1.2. Под везивањем подразумева се да html страница везује своје делове за одређени контролер који има неку своју одређену функцију нпр. учитавање података са сервера или као резултата неких израчунавања.

```
var helloWorldApp = angular.module('helloWorldApp', [
    'ngRoute',
    'helloWorldControllers',
    'helloWorldServices'
]);

helloWorldApp.config(['$routeProvider',
    function ($routeProvider) {
```

```
        $routeProvider.
                when('/home', {
                    templateUrl: 'templates/helloWorld.html',
                    controller: 'navCtrl'
                }).otherwise({
                    redirectTo: '/'
                });
    }]).run(['$rootScope', '$location', '$http', 'RequestApi',
        function ($rootScope, $location, $http, RequestApi) {
    }]);
```

Скрипта 4.1: Контролерска класа

У следећој контролерској скрипти, налази се дефиниција контролера као и тело контролера које дефинише шта контролер тачно ради. У овом примеру контролер се везује за фабрику (енг. Factory) која је описана у 4.1.2. И у телу контролера може се написати логика коју ради као и извршавање серверских захтева преко $http. Такође врши се уписивање резултата у $scope.results који се у html делу везује са приказивањем.

```
/* Controllers */
var helloWorldControllers = angular.module('helloWorldControllers', []);
//GET HelloWorld
helloWorldApp.controller('helloWorldController', ['$scope', '
   getHelloWorldFactory',
    function ($scope, getHelloWorldFactory) {
        $scope.results = getHelloWorldFactory.result();
    }]);
```

Скрипта 4.2: Контролерска класа

У скрипти 4.1.2 дата је дефиниција фабрике. Њено својство на овом примеру је извршавање GET захтева ка серверу.

```
var helloWorldServices = angular.module('helloWorldServices', ['
   ngResource']);

//get factory
helloWorldServices.factory('getHelloWorldFactory', ['$resource',
    function ($resource) {
        return $resource('../server/server.php/helloworld', {}, {
            result: {method: 'GET', isArray: true}
        });
```

```
    }]);
```

Скрипта 4.3: Фабрика

У 4.1.2 делу кода врши се везивање контролера за дугме. Приказ резултата у HTML-у врши се преко res.result.

```
<form>
    <div class="row">
        <div class="col-lg-12">
            <div class="form-group">
                <label class="control-label">Enter expression</label>
                <div class="input-group">
                    <span class="input-group-addon">=</span>
                    <input class="form-control" type="text" id="
   helloWorld" name="result" ng-model="expression" ng-required="true">
                    <span class="input-group-btn">
                        <button class="btn btn-default" type="button" ng
   -click="helloWorldController()">Hello World</button>
                    </span>
                </div>
            </div>
        </div>

    <div>
    <h2>Results</h2>
    <div class="list-group" ng-repeat="res in results">
        <a class="list-group-item">
            <h4 class="list-group-item-heading">{{res.result}}
            </h4>
        </a>
    </div>

</div>
</form>
```

Скрипта 4.4: Скрипта која врши везивање контролера и његових података за HTML

4.1.2 део кода се уграђује у почетну страницу у делу који је предвиђен за то, а он се одређује преко ng-view-a. За почетну страницу врши се дефинисање коришћења AngularJS преко ng-app="helloWorldApp".

```
<!DOCTYPE html>
<html lang="en" ng-app="helloWorldApp">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale
   =1.0">
        <title>Kafkalator</title>

        <link rel="stylesheet" href="css/bootstrap.css" type="text/css"/
   >
        <!--<script src="js/angular/angular.min.js"></script> -->
        <script src="js/angular/angular.js"></script>
        <script src="js/angular/angular-resource.js"></script>
        <script src="js/angular/angular-route.js"></script>

        <script src="js/app.js"></script>
        <script src="js/services.js"></script>
        <script src="js/controllers.js"></script>
    </head>
    <body>
        <div id="wrapper">
            <div ng-view ></div>
        </div>
        <script src="js/jquery/jquery-1.11.1.js"></script>
        <script src="js/bootstrap.js"></script>
    </body>
</html>
```

Скрипта 4.5: Почетна страница

Више о AngularJS-у може се пронаћи у [27][30][31][32][33][34].

4.1.3 Bootstrap

Bootstrap је платформа за развој предњег дела (корисничког интерфејса) (енг.front-end framework) која представља бесплатну колекцију алата за израду веб-сајтова и веб апликација. Bootstrap је производ отвореног кода који су развили **Марк Ото**[2]

[2]Mark Oto (енг. Mark Otto)

и **Јакоб Торнтон**[3] који су у тренутку када је пласиран, били запослени у Твитеру[4]. Постојала је потреба за стандардизацијом скупа инжењерских алата предњег плана у оквиру компаније.
Након што је покренут у августу 2011. године, доживео је велику популарност. Еволуирао је од потпуно CSS (енг. Cascading Style Sheets) до тога да садржи низ (JavaScript) додатака и икона које иду руку под руку са формама и дугмадима. У основи он подржава прилагодљив веб дизајн, има мрежи коју чини 12 робусних колона чија је ширина 940 пиксела. Коришћење Bootstrap-а има својих предности и мана, у следећа два одељка осврнућемо се на неке од њих [35][36][37].

4.1.3.1 Добре стране

- **Више прегледачка (енг. Cross-browser) подршка**: Bootstrap ради на свим новијим десктоп и мобилним прегледачима. Док старији прегледачи могу другачије приказати Bootstrap када је реч о стиловима, он је и даље потпуно функционалан у старијем прегледачу попут Internet explorer-а 8.

- **Лако прилагодљив**: Bootstrap се лако прилагођава, посебно коришћењем LESS-а[5]. Могуће је изоставити делове који нису потребни, наиме, могуће је користити мрежу, а изоставити компоненте или обрнуто.

- **Подстиче на употребу LESS-а**: Bootstrap је писан у LESS-у, LESS се преводи у CSS, а то му пружа пуно флексибилности. То се може искористити уколико корисник употребљава LESS, како би написао сопствене стилове.

- **Подржава коришћење корисних jQuerry[6] додатака**: Bootstrap долази уз мноштво корисних jQuerry додатака који могу добро доћи у разним ситуацијама.

- **Доступност многих корисничких jQuerry додатака**: Bootstrap укључује широк спектар jQuerry додатака који проширују његову функционалност, као

[3] Јакоб Торнтон (енг. Jacob Thornton)
[4] Твитер (енг. Twitter)је бесплатна онлајн друштвена мрежа и микро-блог услуга која омогућава својим корисницима да шаљу своје и читају туђе микро-текстуалне уносе, тзв. твитове.
[5] LESS - чини језгро Bootstrap-а, динамички језик за опис стилова
[6] jQuerry - javaScript библиотека за писање скрипти клијентског HTML-а

што су:
X-editable[7], Wysihtnml5 [8] и jQuerry слање датотека на сервер.

- **Преносивист на првом месту**: Bootstrap је преносив од верзије 3.0. Ово значи да се мрежа уз помоћ испитивања о каквом екрану је реч, шири или скупља [36].

4.1.3.2 Лоше стране

- **jQuerry додатке је тешко прилагодити**: jQuerry додаци који долазе уз Bootstrap се тешко прилагођавају, многи расправљају о томе да нису написани по узору на најбољу праксу, па може бити изазовно радити на изворном коду. Обично додаци раде у већини стандардних случајева, али умеју да закажу уколико покушамо да их прилагодимо за сопствене потребе.
- **Мноштво Bootstrap страница личе једна на другу**: Нажалост, мноштво сајтова који су изграђени уз помоћ Bootstrap-а су потпуно слични, али ово се може избећи креирањем сопствених тема [36].

У овом следећем одељку видећемо једноставан пример коришћења Bootstrap-а.

4.1.3.3 Здраво свете

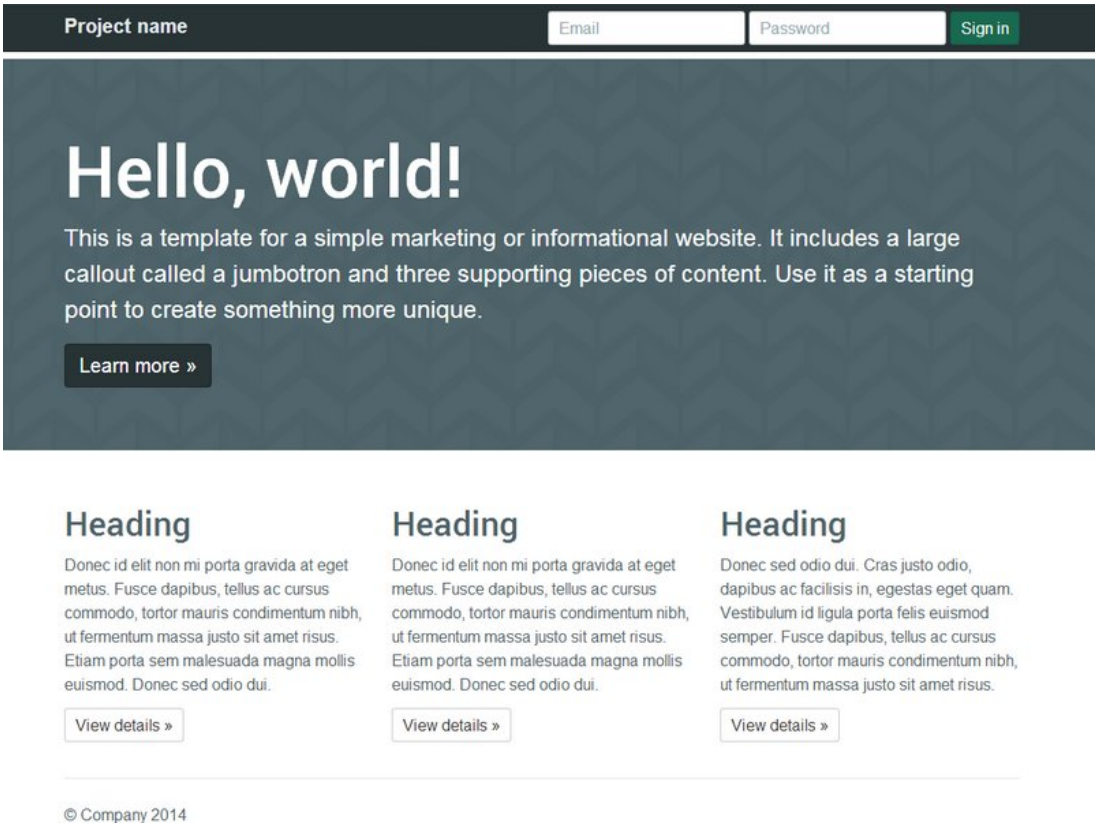
Пример (4.1.3.3) „Здраво свете" коришћењем Bootstrap-а:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,initial-scale
   =1.0">
        <link rel="stylesheet" href="css/bootstrap.css" type="text/css"/
   >
        <title>Bootstrap - Zdravo svete</title>
    </head>
    <body>
```

[7] X-editable - библиотека која дозвољава креирање променљивих елемената на страници
[8] Wysihtnml5 - JavaScript додатак који омогућава креирање једноставних wysiwyg(what you see is what you get) програма за уређење текста (енг. editor)

```
        <div class="container">
            <h1>Zdravo svete!</h1>
        </div>
        <script src="js/jquery/jquery-1.11.1.js"></script>
        <script src="js/bootstrap.js"></script>
    </body>
</html>
```

Скрипта 4.6: Здраво Свете



Слика 4.4: Пример Bootstrap странице

Више о Bootstrap-у може се пронаћи у [36][35][37][38].

4.1.4 KaTeX

KaTeX је брза, лака за коришћење JavaScript библиотека за TeX, математички приказ на веб-у.

Особине:

- **Брзина**: KaTeX синхроно исцртава математичке симболе и нема потребу за променом форматирања странице
- **Квалитет штампе**: KaTeX-ов распоред је базиран на TeX-у Доналда Кнута[9]
- **Самосталан**: није зависан и може лако да се интегрише са ресурсима веб-сајт-а
- **Серверски преводљив**: производи исти резултат у било ком прегледачу или окружењу, тако је могуће исцртати изразе на серверу уз помоћ NodeJS-a, и послати клијенту као HTML.

Пример позивања KaTeX-а са изразом и DOM елементом за исцртавање:

```
katex.render("c = sqrt{a^2 + b^2}", element);
```

За генерисање HTML-а на клијенту, користимо:

```
var html = katex.renderToString("c = sqrt{a^2 + b^2}");
```

Више о KaTeX-у може се пронаћи у [39][40].

4.2 Серверска страна

Серверска страна односи се на операције које се изводе на страни сервера у односу клијент-сервер у оквиру рачунарске мреже. Типично, сервер је рачунарски програм, као што је веб-сервер, који се извршава на удаљеном серверу који је доступан из корисничке рачунарске мреже или радне станице. Операције се изводе на серверској страни зато што захтевају приступ информацијама или функционалностима које нису доступне клијенту или захтевају неко специфично понашање које је непоуздано када се извршава на страни клијента [41]. Више о архитектурама Веб апликација, принципима и протоколима може се пронаћи у [42].

[9] Доналд Кнут (енг. Donald Knuth) - амерички информатичар, математичар и пензионисани професор на универзитету Стенфорд, често називан „оцем алгоритама”.

4.2.1 Преношење репрезентације ресурса (енг. REST, Representational State Transfer)

Преношење репрезентације ресурса REST је софтверски архитектурални стил који се састоји од смерница и најбољих пракси за израду скалабилних веб сервиса. REST је координисан скуп ограничења примењен на дизајн компоненти система за дистрибуирање хипермедија (7.1) који може да доведе до бољих перформанси и одрживе архитектуре. Основна идеја REST-а као основног архитектуралног принципа веба је први пут је изложена у докторској дисертацији Роја Филдинга[10]. Основну идеју и мотивацију REST-а налазимо у речима Роја Филдинига:[43]

"Мотивација за развој REST-а је прављење модела архитектуре који описује како би Веб требало да ради, таквог да може да послужи као оријентир при дефинисању стандарда протокола за Веб."

"REST би требало да подсећа на понашање добро пројектоване Веб апликације: мрежа страница (виртуални коначни аутомат), код које корисник пролази кроз апликацију бирањем веза (преласци стања), што резултује преношењем нове странице (наредно стање апликације) кориснику и њеним припремањем за употребу"

Филдинг је дефинисао основе REST-а кроз шест ограничења, а то су:

- **Униформан интерфејс**: дефинише интерфејс између клијента и сервера. Упрошћава и рашчлањује архитектуру, што омогућава независан развој сваког дела. Четири водећа принципа јединственог интерфејса су:

  1. **Заснован на ресурсима** - индивидуални ресурси су идентификовали у захтевима користећи URI-е као идентификаторе ресурса. Сами ресурси су концептуално одвојени од репрезентације која се враћа клијенту. На пример, сервер након примања захтева не шаље читаву базу података већ HTML, XML или JSON који репрезентују тражене податке из базе података. Нпр. уколико тражимо члана са идентификационим бројем 12, наш URI може бити **GET: /member/12**.

[10]Роj Филдинг (енг. Roy Fielding) - амерички информатичар, један од главних аутора HTTP спецификације, суоснивач Apache HTTP Server пројекта

2. **Манипулација ресурса кроз репрезентацију** - када клијент има репрезентацију ресурса, укључујући прикључене метаподатке. Клијент са тим има довољно информација да промени, обрише или унесе ресурс, уколико му наравно привилегије на серверу то дозвољавају. На пример уколико желимо да унесемо новог члана, вршимо **POST: /member** са телом поруке приказанаом у делу 2. На овом примеру наводи се JSON репрезентација ресурса.

```
{
"name": "Stefan",
"surname": "Petkovic"
}
```

Скрипта 4.7: JSON ресурс

3. **Самоописујуће поруке** - свака порука укључује довољно информација које описују како да се обради та порука. Нпр. Internet media type (некада познат под називом MIME 7.1) може садржати информације о томе који од парсера требамо позвати. Одговори експлицитно означавају да ли имају способност кеширања.

4. **Хипермедија као механизам апликативног ресурса (енг. HATEOAS)** - клијенти испоручују стање у овиру садржаја тела, параметара упита-стринга, заглавља захтева и ресурсног URI-а (име ресурса). Сервиси испоручују стање клијентима преко садржаја тела, кодова одговора и заглавља захтева. Ово се технички односи на хипермедију (хиперлинкове унутар хипертекста). HATEOAS (енг. Hypermedia as the Engine of Application State) означава где је неопходно да се линкови (везе) садрже унутар враћеног одговора (или заглавља) како би снабдели URI за повраћај објеката или објеката који су везани за тражени објекат. Униформан интерфејс (раздвајање класе од имплементације или њених зависности) који сваки REST сервис мора пружити је фундаменталана основа за његов дизајн [44].

- **Нема успостављања стања (енг. Stateles)** - како је REST акроним за **RE**presentational **S**tate **T**ransfer, Stateless је кључан појам. У суштини, ово значи да је неопходан ресурс који се обрађује садржан у самом захтеву, било као део URI-а, параметар упит стринга, тела или као део заглавља. URI јединствено идентификује ресурс и тело садржи стање или промену стања тог

ресурса. Након што сервер изврши процесирање, одговарајуће стање или делови стања који су битни, шаљу се назад клијенту преко заглавља у ком се налазе статус и тело одговора који се потом саопштава клијенту.

- **Кеширање** - као и на вебу, клијенти могу да кеширају(складиште) одговоре. Одговори морају бити имплицитно или експлицитно дефинисани као они који могу да се кеширају или не, како би спречили клијенте да поновно користе застареле или неприкладне податке као одговор на даље захтеве. Добро управљање кешом делимично или потпуно елиминише неке од клијент-сервер интеракција и даље побољшавање проширивости и боље перформансе.

- **Клијент-сервер** - униформан интерфејс одваја клијенте од сервера. Клијенти се не баве складиштењем података, то остаје интерно на сваком серверу, тако да се преносивост клијентског кода побољшава. Сервери се не баве корисничким интерфејсом или стањем корисника, тако да могу бити једноставнији и скалабилнији. Серверски и клијентски део може бити замењен или развијен самостално, докле год се интерфејс не мења.

- **Слојевит систем** - у нормалним ситуацијама клијент не може знати да ли је директно повезан са крајњим сервером или је повезан на посредник. Посреднички сервери могу повећати скалабилност система омогућујући балансирање оптерећења и обезбеђивањем заједничог кеша. Слојеви такође могу спроводити неке безбедносне политике.

- **Код на захтев (енг. Code on demand)(изборни)** - сервери су у могућности да привремено прошире или прилагоде функционалност пребацивањем логике на клијента како би је самостално извршио. Примери оваквог пребацивања су Јава аплети или JavaScript скриптови [44].

Један од циљева REST-а јесте да сложене протоколе попут: CORBA-е[11], RPC-а[12] и SOAP-а[13] замени једноставним HTTP или HTTPS.
Апликације које подржавају стил архитектуре REST називају се RESTful апликације.

[11]CORBA (Common Object Request Broker Architecture) - стандард дефинисан од стране Object Management Grop-е дизајниран да олакша комуникацију система који су распоређени на различитим платформама

[12]RPC (Remote Procedure Call) - унутар процесна комуникација која дозвољава једном програму да позива потпрограме или процедуре на извршавање у другом адресном простору.

[13]SOAP (Simple Object Acess) - представља спецификацију протокола за размену структуираних информација у имплементацији веб сервиса у оквиру рачунарске мреже

- За све операције ПЧМБ (Пиши, Читај, Мењај, Бриши) (енг. CRUD) апликација користи услуге сервиса путем одговарајућих протокола и имена
- Све операције средњег слоја се дефинишу као једноставне операције (ПЧМБ) на потенцијално сложеним објектима

Као што смо навели, ресурс коме се приступа се одређује URI-ем, а операције се одређују методом протокола HTTP:

- **PUT** - мењање ресурса
- **GET** - читање ресурса
- **POST** - прављење новог ресурса
- **DELETE** - брисање ресурса.

Више о REST-у и RESTful веб сервисима може се наћи у [45][46][47][48].

4.2.2 PHP

PHP је серверски скрипт језик дизајниран за веб-развој али се такође користи као програмски језик опште намене. Од јануара 2013, PHP је инсталиран на више од 240 милиона веб-сајт-ова и 2.1 милиона веб-сервера. Креирао га је Расмус Лердорф[14] 1994 године а референтну имплементацију PHP-а (чији је погон Zend Engine[15]) данас се производи од стране PHP групе.
PHP код се може једноставно комбиновати са HTML кодом или се може користити у комбинацији са разним тематским механизмима (енг. templating engines) и веб оквирима. PHP код обично обрађује PHP интерпретер, који је обично имплементиран као основни модул веб-сервера или се извршава преко општег интерфејса за мрежни пролаз (CGI - Common Gateway Interface). Након што је PHP код интерпретиран и извршен, веб-сервер шаље излазне резултате клијенту, обично у форми дела генерисане веб странице или као део генерисане веб странице. На пример, PHP код може генерисати HTML код, слике или неке друге податке веб странице.

[14]Расмус Лердорф (енг. Rasmus Lerdorf) - гренландско-канадски програмер, креатор PHP-а
[15] Zend Engine - скрипт механизам отвореног кода који интерпретира PHP

PHP је императиван, функционалан, објектно-оријентисан, процедурални и рефлективан језик који је писан примарно у програмском језику **C**, док су неке компоненте писане у **C++**-у.[49]

Више о PHP-у може се наћи у [50][51][52][53].

4.2.2.1 Laravel платформа за развој

Ларавел је бесплатана, PHP веб апликативна платформа за развој отвореног кода, дизајнирана за развој модел-поглед-контролер МПК (4.1.1.1) веб апликација. Ларавел је издата под MIT лиценцом и њен изворни код је доступан на складишту (7.1) на GitHub-у [54].
Програмери најчешће интерагују са Ларавелом преко кодмандне линије као услужног програма који генерише и руководи Ларавел пројектом. Ларавел садржи алат командне линије под називом Занатлија (енг. Artisan), који може да се користи као генератор скелетног кода и чланова шеме базе података. Занатлија управља свима, почевши од померања шеме базе података до елемената (поставке) и управљања конфигурацијом.

Компоненте Ларавела

Типична Ларавел апликација се састоји од компоненти МПК, као што се може видети на слици 4.5.
Када интерагује са Ларавел апликацијом, прегледач шаље захтев, који се прима од стране веб-сервера и прослеђује Ларавел рутирајућем механизму. Ларавел рутер прима захтев и преусмерава одговарајућој контролерској методи класе заснованој на рутирајућем URL шаблону.
Контролерска класа потом преузима захтев. У неким случајевима, контролер ће одмах приказати поглед, који представља шаблон који се конвертује у HTML и враћа натраг прегледачу. Код динамичких сајтова, чест је случај да контролер интерагује са моделом, који је PHP објекат који репрезентује један елемент апликације (попут корисника или неког текста) и он је одговоран за комуникацију са базом података. Након што је добио одговор од модела, контролер припрема финални поглед (HTML, CSS) и враћа комплетну веб страницу корисничком прегледачу.
Ларавел се залаже за концепт да модели, погледи и контролери треба да буду сасвим одвојени тако што ће се кодови за сваки од ових елемената складиштити као посебни фајлови у одвојеним директоријумима. На овом месту структура Ларавел

Слика 4.5: Архитектура Модел-поглед-контролер

директоријума долази до изражаја.
Готова решења попут МПК су креирана како би посао програмера био олакшан. Ово је још један од разлога зашто је Ларавел бољи од обичног PHP-а.
Детаљи о Ларавел платформи могу се пронаћи у [55][56].

4.2.3 Посредник порука

Посредник порука (енг. Message broker) је програм који врши превођење са једног системског језика на други глобално прихватљив језик путем телекомуникационог медија.
Посредник порука је архитектурални шаблон за валидацију, трансформацију и усмеравање порука. Он посредује комуникацији међу апликацијама, сводећи на минимум заједничку свест коју би апликације требало да имају једне о другима како би могле да размењују поруке, уз ефикасно спровођење раздвајања.

Сврха посредника је да прими поруке које стижу од апликација (сервиса) и изврши неку акцију над њима. Неке од акција које се могу предузети од стране брокера су:

- Рутирање порука на једну или више могућих дестинација
- Трансформација порука на неку алтернативну репрезентацију
- Агрегација порука, раздвајање порука на више мањих и њихово слање на одређено одредиште, преформулисање одговора у једну поруку која ће бити послата кориснику.
- Интеракција са екстерним спремиштима у циљу проширења порука или складиштења
- Позивање Веб сервиса за преузимање података
- Одоговор на грешке или догађаје
- Обезбеђује садржај и рутирање тематски заснованих порука помоћу шаблона објавити-претплатити [57]

4.2.3.1 Apache Kafka

Апачи Кафка (енг. Apache Kafka) је посредник порука и дистрибуирани објави-претплати систем порука отвореног кода. Развила га је LinkedIn[16] корпорације, а касније је постао део Апачи пројекта. Кафка је:

- *Брз* - један Кафка посредник (агент, брокер) може носити стотине мегабајта читања и писања по секунди од хиљаду клијената
- *Скалабилан* - Дизајн Кафке омогућава једном кластеру да служи као централна окосница података за неке велике организације. Може се еластично и транспарентно проширити без застоја. Токови података су подељени и простиру се преко кластер машина како би дозволили токове података веће од способности сваке појединачне машине и дозволили кластере координисаних потрошача
- *Дуготрајан* - поруке су трајне на диску и реплицирају се у оквиру кластера како би се спречио губитак података. Сваки брокер може носити терабајте порука без утицаја на перформансе

[16]LinkedIn - пословно орјентисана друштвена мрежа

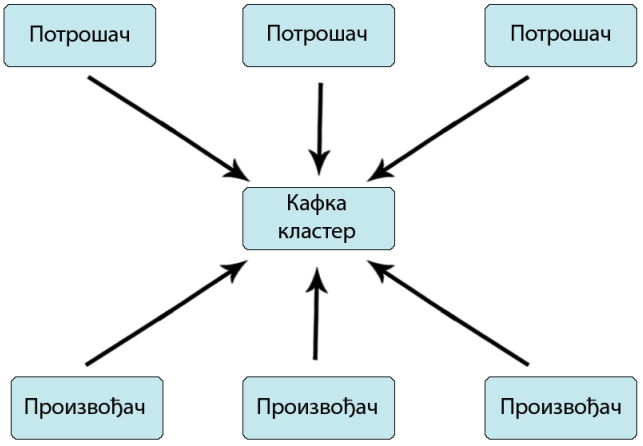
- *Дистрибуиран по дизајну* - Модеран дизајн фокусиран на извршавање на кластеру који пружа јаку издржљивост и гарантну отпорност на грешке [58][59].

Развила га је Апачи софтверска фондација а писан је на Скали[17].

Архитектура

Основна терминологија

- Ток порука одређеног типа дефинише се као **Тема** (енг.**topic**). **Порука** је дефинисана као скуп корисних података који носе низ бајтова, а **тема** је категорија или извор снабдевања на који се поруке објављују.
- **Произвођач** може бити свако ко објављује поруке на неку тему.
- Објављене поруке се потом чувају на скупу сервера који се називају **посредници Кафка кластера**.
- **Потрошач** може да се претплати на једну или више Тема и конзумира објављене Поруке узимајући податке од агента.



Слика 4.6: Кафка Потрошачко, Произвођачко и Посредничко окружење

Комуникација између клијента и сервера се изводи простим, једноставним, TCP протоколом, језичним агностиком[18] високих перформанси. Произвођач може изабрати

[17] Скала (енг. Scala) - објектно-функционални програмски језик

[18] језични агностик - описује развојну софтверску парадигму где се бира посебан језик због своје прикладности за посебан задатак.

било који метод серијализације за кодирање садржаја поруке. Ради ефикасности, произвођач може послати скуп порука у једном захтеву за објављивање. Следећи Јава код показује како се прави Потрошач који шаље поруке.

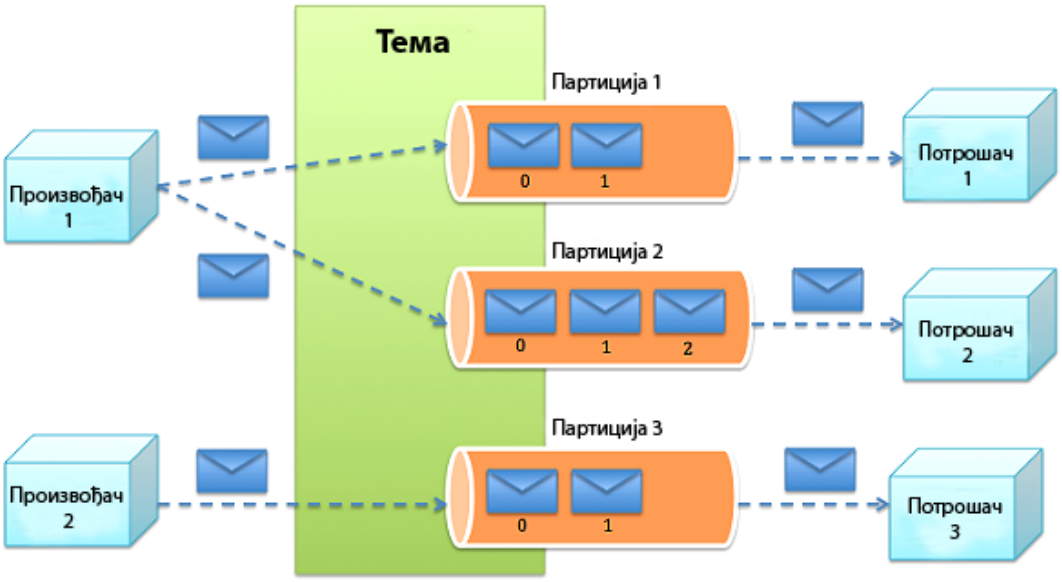
```
producer = new Producer();
message = new Message(''tekstualna poruka''.getBytes());
set = new MessageSet(message);
producer.send(''topic1'', set);
```

За пријављивање на тему, потрошач прво ствара један или више токова порука за тему. Поруке које су објављење на ту тему биће равномерно распоређење на токове. Сваки ток порука пружа итераторски интерфејс преко континуалног тока порука које су произведене. Потрошач затим пролази кроз сваку поруку у току и обрађује исту. За разлику од традиционалних итератора, итератор тока порука никада не долази до завршетка, увек се извршава. Ако тренутно нема нових порука, итератор се блокира док се нова порука не објави на теми. Кафка подржава како режим доставе од тачке до тачке у којем више потрошача заједнички користе једну копију поруке у низу, тако и модел објави-претплати у ком више потрошача преузимају своју копију поруке. Следећи Јава код показује како се врши примање порука.

```
streams[] = Consumer.createMessageStreams(''topic1'', 1)
for (message : streams[0]) {
    bytes = message.payload();
    // obraditi bajtove
}
```

Целокупна архитектура Кафке је приказана на слици 4.7. Пошто је Кафка дистрибуираног карактера, Кафка кластер се обично састоји од више агената. Како би се успоставила равнотежа у раду, тема је подељена на више партиција и сваки агент чува једну или више таквих партиција. Више произвођача и потрошача могу истовремено објавити и преузети поруке.

Меморија Кафке Кафка има врло једноставан план складиштења. Свакој партицији теме одговара логички дневник. Физички, дневник је имплементиран као

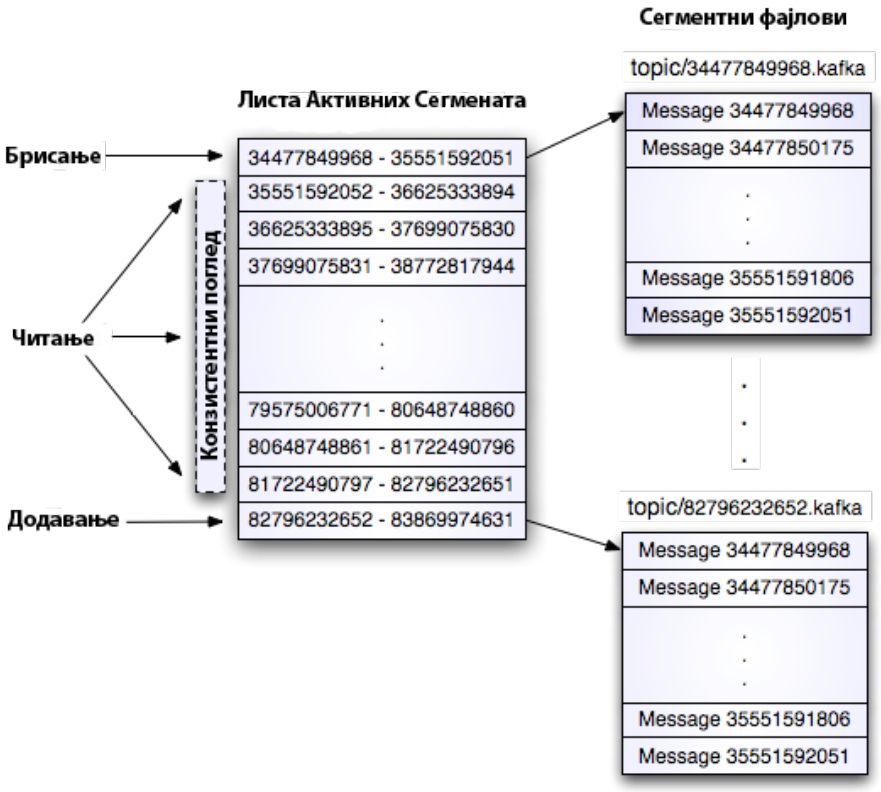


Слика 4.7: Архитектура Кафке

скуп сегментних фајлова једнаких димензија. Сваки пут када произвођач објави поруку на партицији, агент једноставно додаје поруку последњем сегментном фајлу. Сегментни фајл се „гура" на диск након што се конфигурисани број порука објави или након што прође неко одређено време. Поруке су изложене потрошачу након што буду „гурнуте".

За разлику од традиционалних система порука, поруке које се чувају у Кафка систему немају одређену експлицитну идентификацију. Логичким померајем у дневнику излажу се поруке. Овим се избегавају додатни трошкови одржавања помоћних индексних структура које мапирају идентификацију поруке на стварну локацију поруке. Идентификација порука је инкрементална, али није узастопна. Израчунавање идентификације следеће поруке се врши тако што дужину тренутне поруке додамо на логички померај. Потрошач увек преузима поруке са одређене партиције секвенцијално и ако потрошач „потврди" посебни померај поруке, то подразумева да је потрошач примио све претходне поруке. Потрошач шаље асинхрони захтев за повлачење посреднику који садржи померај поруке која се преузима. Кафка користи API послатог фајла за ефикасно испоруку бајтова у дневник од посредника потрошачу.

Кафка Посредник За разлику од других система за посредовање порукама, Кафка посредници немају стање. Ово значи да потрошач води рачуна о томе колико је порука преузео. Потрошач се самостално одржава и агент нема потребе за



Слика 4.8: Архитектура меморије Кафке

било каквим радом. Овакав дизајн је иновативан, али, са друге стране, носи одређене потешкоће.

- Врло је незгодно брисати поруке из агента с обзиром на то да агент не зна да ли је потрошач преузео поруку или није. Кафка иновативно решава овај проблем тако што користи једноставан временски заснован SLA[19] 7.1 за политику задржавања. Порука се аутоматски брише ако је задржана од стране брокера дуже од неког одређеног периода.

- Овакав иновативни дизајн има огромну корист, потрошач може планирано, циљано да се врати на стари померај и поновно преузме податке. Ово крши заједнички договор о реду, али доказује се да је ово за многе потрошаче јако корисна карактеристика јер могу да изврше враћање на неке од порука које нису обрадили или нису примили.

[19] SLA (Service-level agreement) - је део сервисног договора где је сервис формално дефинисан

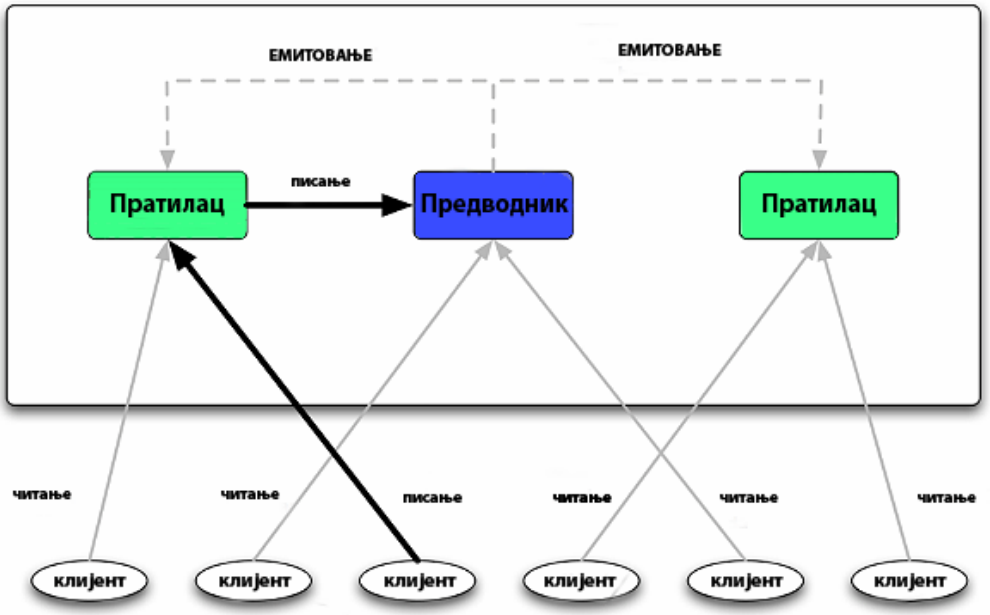
ZooKeeper и Кафка Размотримо дистрибуиран систем са више сервера, од којих је сваки одговоран за држање података и вршење операција над тим подацима. Неки потенцијални примери су дистрибуирани претраживач, дистрибуиран систем изградње или познати системи попут Апачи Хадупа[20]. Један од уобичајених проблема код свих ових дистрибурираних система је како утврдити који сервис је функционалан и оперативан у било ком временском тренутку. Овде се поставља питање како поздано поступити када се суочимо са потешкоћама дистрибуиране обраде попут мрежних падова, ограничења пропусне ширине, разних кашњења на вези, сигурносних и осталих проблема који могу настати у мрежном окружењу и у центрима за обраду података. Ове врсте питања су фокус Апачи Zookeeper-а, који је брз, веома доступан, отпоран на грешке, дистрибуиран сервис за координирање. Користећи ZooKeeper можемо градити поуздане, дистрибуиране структуре података за групно чланство, бирање предводника, координираног тока рада и конфигурационих сервиса, као и генерализованих дистрибуираних структура података попут брава, редова, баријера и копчи. Многи познати и успешни пројекти се ослањају на ZooKeeper, као што су: HBase, Hadoop 2.0, Solr Cloud, Neo4J, Apache Blur и Accumulo.

ZooKeeper је дистрибуиран, хијерархијски систем фајлова који олакшава лабаво повезивање између клијената и пружа доследан поглед на своје з-чворове (регистре података, енг. z-nodes), који су попут фајлова и директоријума у традиционалном фајл систему. Он пружа основне операције као што су креирање, брисање и провера постојања регистара података, као и модел вођен догађајима у ком клијенти могу посматрати промене специфичних з-чворова, као на пример када се дете прикључи на постојећи з-чвор.

ZooKeeper постиже високу доступност користећи више ZooKeeper сервера, који се називају **ансамбли**, где сваки сервер држи копију дистрибуираног фајл система унутар меморије како би одговорио на клијентов захтев за учитавањем/читањем. Слика 4.9 приказује типичан ZooKeeper ансамбл у којем се један сервер понаша као предводник а сви остали као његови пратиоци. На почетку ансамбла, предводник се бира први, потом сви пратиоци врше копирање својих стања са стањем предводника. Сви захтеви за писање су усмерени ка предвонику, а промене се емитују свим пратиоцима. Промена емитовања се назива **атомичко емитовање** (енг. atomic broadcast).

Употреба ZooKeeper-а у Кафки: У погледу координације и олакшавања дистрибуираног система, ZooKeeper се користи из истог разлога као и Кафка. ZooKeeper

[20]Апачи Хадуп (енг. Apache Hadoop) - је скуп алгоритама за дистрибуирано складиштење и дистрибуирано процесирање веома великих количина података (енг. Big Data) на кластерима рачунара изграђеним од commodity computing (коришћење великог броја доступних компоненти са паралелним израчунавњем како би добили што већи број корисних израчунавања по ниској цени)



Слика 4.9: Ансамбалска ZooKeeper архитектура

се користи за управљање и кординисање Кафка посредника. Сваки Кафка посредник кординише са осталим Кафка посредницима користећи ZooKeeper. Произвођач и потрошач су обавештени ZooKeeper-овим сервисом о присуству новог брокера или о неуспеху брокера у Кафка систему.

Према обавештењу које ZooKeeper добија у вези са присуством или пада посредничког потрошача или произвођача, Zookeeper доноси решење и почиње са координисаним радом са неким од других брокера. Генерална системска архитектура Кафке је приказана на слици 4.10.

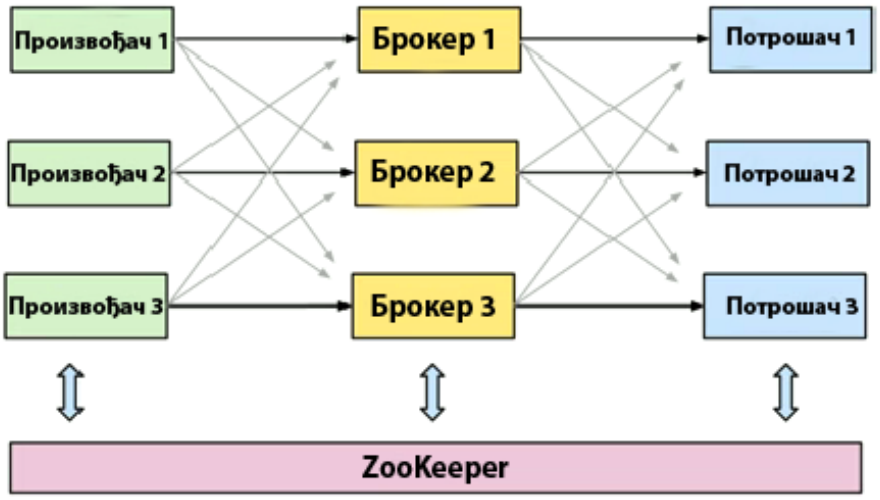
Упутство за постављање Кафке

За почетак, потребно је преузети Кафку са [60]. Као што смо написали у претходном делу да Кафка користи ZooKeeper, потребно је прво да покренемо ZooKeeper. То постижемо покретањем команде:

```
:bin/zookeeper-server-start.sh config/zookeeper.properties
```

Zookeeer.properties представља конфигурациони фајл. Сада стартујемо Кафку покретањем команде:

```
:bin/kafka-server-start.sh config/server.properties
```



Слика 4.10: Генерална архитектура Кафке као дистрибуираног система

Креирање теме се постиже извршавањем команде:

```
:bin/kafka-topics.sh --create --zookeeper localhost:2181
    --replication-factor 1 --partitions 1 --topic nazivTeme
```

Индикатори попут *replication-factor* означавају на колико различитих сервера се реплицира тема, partitions означава број партиција које ће поседовати направљена тема и topic означава именовање теме. Излиставање свих постојећих тема врши се командом:

```
:bin/kafka-topics.sh --list --zookeeper localhost:2181
```

Покретање конзолног произвођача за слање порука врши се командом:

```
:bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
   nazivTeme
```

А покретање конзолног потрошача врши се командом:

```
:bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic
   nazivTeme   --from-beginning
```

Индикатор *–from-beginning* означава да се ишчитају све поруке које су пристигле на дату тему.

Више речи о конкретној примени Кафке биће у одељеку 5. Више информација о самој Кафки и ZooKeeper-у може се пронаћи на [58][59].

4.2.4 Језици сервиса

Сервиси могу бити писани у разним програмским језицима. У овом одељку биће представљени језици који су коришћени на платформи, неки од њих се обрађују на наставним јединицама у оквиру курсева на Математичком факултету.

4.2.4.1 Програмски језик C

Програмски језик C је програмски језик опште намене који је 1972. године развио Денис Ричи[21] у Беловим телефонским лабораторијама (енг. Bell Telephone Laboratories) у САД. Име C долази од чињенице да је језик настао као наследник језика B (а који је био поједностављена верзија језика BCPL). C је језик који је био намењен превасходно писању системског софтвера и то у оквиру оперативног система Unix. Међутим, временом је почео да се користи и за писање апликативног софтвера на великом броју других платформи. C је данас присутан на широком спектру платформи - од микроконтролера до суперрачунара. Језик C је утицао и на развој других програмских језика (најзначајнији од њих је језик C++ који се може сматрати проширењем језика C).
Језик C спада у групу императивних (процедуралних) програмских језика. Како је изворно био намењен за системско програмирање, програмерима нуди прилично директан приступ меморији и конструкције језика су тако осмишљене да се једноставно

[21] Денис Ричи (енг. Dennis Ritchie) (1941–2011), амерички информатичар, добитник Тјурингове награде 1983. године.

преводе на машински језик. Захваљујући томе, у C-у се релативно једноставно креирају програми који су раније углавном писани на асемблерском језику. Језик је креиран у минималистичком духу — има мали број кључних речи, а додатна функционалност програмерима се нуди углавном кроз коришћење (стандардизованих) библиотечких функција. На пример, не постоје наредбе језика којима би се учитавали подаци са тастатуре рачунара или исписивали на екран, већ се ови задаци извршавају позивањем функција из стандардне библиотеке.
У развоју језика C се од самог почетка инсистирало на стандардизацији и преносивости кода (тзв. портабилности), захваљујући чему се исти програми на C-у могу користити (тј. преводити) на различитим платформама [61].
Више о програмском језику C можете наћи у [61] [62].

Lex. Lex је рачунарски програм који генерише лексички анализатор. Lex се обично користи са yacc-ом. Lex, који су написали Ерик Шмит[22] и Мајк Леск[23] а описан 1975. постао је стандардни генератор за лексички анализатор на многим Unix системима, а понашање lex-а прецизирано је деловима POSIX стандарда. Lex чита улазни ток и враћа код који имплементира lexer u C-у [63]. Структура lex програма је намерно направљена да буде слична структури yacc програма. Програми су подељени на три дела, одвојена линијама које садрже само два процентна знака (%%), тј.:

Део за дефиниције
%%
Део за правила
%%
Део за C код

Yacc. Рачунарски програм yacc је генератор парсера, који је развио Стивен Џонсон[24] из компаније AT&T за оперативни систем Unix. Име представља скраћеницу која значи још један компилатор компилатора (енг. Yet Another Compiler Compiler). Он генерише парсер (део компилатора који покушава да синтаксно обради изворни

[22]Ерик Шмит (енг. Eric Schmidt) - амерички софтверски инжењер, привредник и извршни директор Гугл-а
[23]Мајк Леск (енг. Mike Lesk) - амерички информатичар и професор универзитета.
[24]Стивен Џонсон (енг. Steven Johnson) - амерички математичар и информатичар творац yacc-а, lint-а, spell-а и портабилног C компајлера

код) на основу формалне граматике записане у форми сличној BNF. Yacc генерише код за парсер у програмском језику C.
Рад уасс-а заснива се на методи LALR(1)(lookahed LR parser)-анализе. Улаз у систем уасс представља изворна датотека која садржи опис правила контекстно слободне граматике. Описи правила граматике проширују се семантичким акцијама које представљају програмски код у C-у који се извршава када се препозна одређени део улаза. Резултат рада уасс-а је датотека која се обично назива y.tab.c. Њу је потребно превести неким C преводиоцем и повезати са остатком програма који пишемо да бисмо добили извршну верзију програма [64].

4.2.4.2 Python

Пајтон (енг. Python) је програмски језик високог нивоа опште намене. Подржава, у првом реду императивни, објектно-оријентисан и функционални стил програмирања. Синтакса језика Пајтон омогућава писање веома прегледних програма. Језик се брзо и лако учи. Програми писани у Пајтон језику се најчешће интерпретирају. Уз интерпретатор се обично испоручује и веома развијена стандардна библиотека модула. Аутор овог језика је Гвидо ван Росум[25] са Универзитета Стичинг у Холандији [65]. Покретање интерпретера за пајтон програм врши се куцањем следеће наредбе у командној линији:

```
python mojProgram.py
```

Више о Пајтону може се наћи у [66][67][68][69][70][71][72][73].

SymPy. SymPy је Пајтон библиотека за симболичко израчунавање, која пружа све функционалности рачунарске алгебре о којој је било речи у одељку 3.
SymPy садржи карактеристике почевши од симболичке аритметике до инфинитезималног рачуна (основе математичке анализе), алгебре, дискретне математике, геометрије, статистике и квантне физике [74].

[25] Гвидо ван Росум, (енг. Guido van Rossum) - холандски програмер, добитник награде за унапређење слободног софтвера, биши запослен Гугла-а, тренутно радни у Dropbox-у.

Додавање SymPy библиотеке врши се командом:

```
from sympy import (* ili navedena oblast)
```

Више о SymPy-ју може се наћи у [22] [23].

4.2.4.3 Ruby

Руби (енг. Ruby) је објектно оријентисани програмски језик. У себи комбинује синтаксу инспирисану језицима Перл[26] и неких команди оперативног система Ада [27], са објектно оријентисаним особинама налик језику Смолток[28]), а дели и неке особине са језицима Пајтон, Лисп[29], Дилан[30] и CLU[31]. Руби је једнопролазни интерпретирани језик. Његова главна имплементација је слободни софтвер под лиценцом отвореног кода.
Јукихиро Мацумото[32] је са развојем овог језика почео у фебруару 1993. године. Први пут је објављен 1995. године, а тренутно је актуелна стабилна верзија 2.2.0. Руби следи принцип „најмањег изненађења", чиме се мисли да је тај језик ослобођен свих замки и контрадикторности познатих из других језика.[75][76][77]
Особине језика:

- једноставна и читљива синтакса
- "чисто" објектно оријентисани језик
  - Наслеђивање од модула, уместо вишеструког наслеђивања
  - Уникатне методе (Singleton)

[26]Перл (енг. Perl) - слободни, независни од платформе и интерпретирани програмски језик. Настао је као синтеза програмског језика C, неких команди оперативног система Unix и других елемената.
[27]Ада (енг. Ada) - програмски језик високог нивоа, који је заснован на Паскалу.
[28]Смолток (енг. Smalltalk) - је објектно-оријентисани, динамички, рефлективни програмски језик.
[29]Лисп (енг. Lisp) - динамички програмски језик који је пројектовао Џон Макарти крајем 1950-тих. Иако је Лисп општенаменски програмски језик, обично се каже да је језик вештачке интелигенције.
[30]Дилан (енг. Dylan) - мулти парадигматски опрограмски језик који укључује подршку за функционално и објектно орјентисано програмирање, уједно је динамички и рефлективан када пружа програмски модел дизајниран за подршку ефиксног генерисања машинског кода.
[31] CLU - објектно орјентисани али не потпуно, процедурални програмски језик, развијен на универзитету MIT од стране професора Барбаре Лисков и њених студената
[32]Јукихиро Мацумото (енг. Yukihiro Matsumoto - јапански информатичар, творац Рубија

  - Динамична промена имена и надоградња класа приликом извршења програма
  - Итератори
  - Преписивање оператора
  - Рефлексија

- нетипизоване варијабле
- аутоматско ослобађање непотребно заузете меморије (garbage collection)
- обрада изузетака
- подршка на више оперативних система
- подршка Перлових регуларних израза
- јединствен интерфејс за приступ базама података
- могућност и функционалног и процедуралног програмирања
- аутоматска документација (слично javadoc-у)
- либерална лиценца

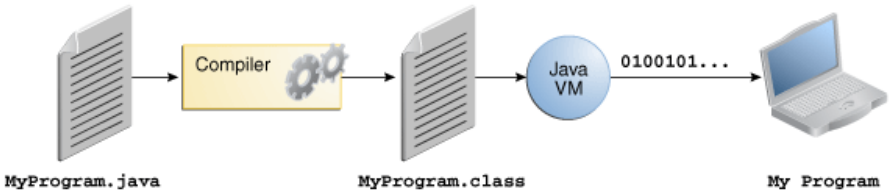
Више о програмском језику Руби можете наћи у [76][77][78].

4.2.4.4 Java

Јава (енг. Java) је програмски језик опште намене који је конкурентан, класно заснован, објектно-оријентисан и функционалан (од Јаве 8) програмски језик, специјално дизајниран да има што је могуће мање имплементационе зависности. Намењен је да дозволи програмерима апликација могућност „пиши једном, покрени било где", што значи да компајлирани Јава код може да се покреће на свим платформама које подржавају Јаву без потребе за рекомпилацијом. Јава апликације се преводе на бајткод који се покреће на било којој Јава виртуелној машини (енг. JVM) без обзира на архитектуру рачунара. Од 2015, Јава је један од најпопуларнијих програмских језика у примени, нарочито за клијент-сервер веб апликације, са пријављених 9 милиона програмера. Јаву је првобитно развио Џејмса Гослинг [33] у компанији Sun Microsystems

[33]Џејмс Гослинг (енг. James Gosling - канадски информатичар, сматра се оцем Јаве.

(која је од тада постала део Оракл корпорације) и издата 1995 као основна компонента Sun Microsystems-ске Јава Платформе. језик великим делом своје синтаксе потиче из C-а и C++-а, али има објекте ниског нивоа за разлику од ова два језика.



Слика 4.11: Преглед процеса развоја софтвера

Гугл и Андроид су изабрали да користе Јаву као кључни стуб за креирање Андроид оперативног система, мобилног оперативног система отвореног кода. Иако је Андроид оперативни систем изграђен на Линкус језгру, већим делом је написан у C-у; Андроид софтверски развојни алат (енг. SDK) користи Јаву као базу за Андроид апликације. Међутим, Андроид не користи Јава виртуелну машину, уместо да користе Јава бајткод као прелазни корак он ултимативно циља Андроидову сопствену виртуелну машину под називом Далвик (енг. Dalvik) [79].

Више о Јава програмском језику може се наћи у [80][81].

Symja. Symja је Јава библиотека за симболичко израчунавање. Неке од њених карактеристика су:

- Користи целе бројеве произвољне прецизности, рационални и комплексни бројеви
- Омогућава диференцирање и интеграцијију
- Рад са полиномима
- Користи обрасце за подударање - проверавање да ли дати низ токена садржи делове неког обрасца
- Решавање проблема линеарне алгебре

Symja користи:

- Пакет опште математике - Апачи општу математичку библиотеку

- Apfloat - библиотеку високих перформанси за произвољно прецизну аритметику
- JAS - Java Algebra System - објектно оријентисани, вишенитни приступ рачунарској алгебри. JAS има добро осмишљену софтверску библиотеку која користи генеричке типове за алгебарске прорачуне реализоване у Јави, коришћењем JVM покретачке инфраструктуре. Библиотека може бити коришћена у било ком Јава софтверском пакету, интерактивно или интерпретирана преко jython (Java Python) или jruby (Java Ruby) предњег краја (енг. front end). Постоји и Андроид апликација базирана на Rubot-у (jruby за Андроид). Фокус JAS-а у овом тренутку је на: комутативним, решивим и нерешивим полиномима, временским серијама, Гребнеровим базама, факторизацији, реалним и комплексним коренима. JAS користи вишепроцесорску снагу где је год то могуће. Имплементиран је као 64-битни систем. JAS је развијен на универзитету у Манхајму [82] [83].

Пример израчунавања израза помоћу Symja библиотеке:

```
import static org.matheclipse.core.expression.F.*;

import org.matheclipse.core.basic.Config;
import org.matheclipse.core.eval.EvalUtilities;
import org.matheclipse.core.interfaces.IAST;
import org.matheclipse.core.interfaces.IExpr;
import org.matheclipse.parser.client.SyntaxError;
import org.matheclipse.parser.client.math.MathException;

public class PrimerIzracunavanja {
  public static void main(String[] args) {
    try {
      // postavljanje flag-a koji parsira sve pozive funkcija
      //bilo da su napisane velikim ili malim slovom
      Config.PARSER_USE_LOWERCASE_SYMBOLS = true;

      EvalUtilities util = new EvalUtilities(false, true);
      // Prikazivanje izraza u Java formi:
      String javaForm = util.toJavaForm("D(sin(x)*cos(x),x)");
      // stampa: D(Times(Sin(x),Cos(x)),x)
      System.out.println(javaForm.toString());

      // Koriscenje Java forme za kreiranje
```

```
      //izraza sa F.* statickim metodama:
      IAST function = D(Times(Sin(x), Cos(x)), x);
      IExpr result = util.evaluate(function);
      // stampa: -Sin(x)^2+Cos(x)^2
      System.out.println(result.toString());

      // izracunavanje direktno iz stringa
      result = util.evaluate("diff(sin(x)*cos(x),x)");
      // stampa: -Sin(x)^2+Cos(x)^2
      System.out.println(result.toString());
    } catch (SyntaxError e) {
      // greske koje proizvodi Symja parser
      System.out.println(e.getMessage());
    } catch (MathException me) {
      //hvatanje Symja matematickih gresaka
      System.out.println(me.getMessage());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Скрипта 4.8: Symja пример израчунавања

4.2.4.5 NodeJS

NodeJS је вишеплатформско окружење отвореног кода које служи за креирање брзих и скалабилних мрежних апликација које се извршавају на страни сервера. NodeJS апликације су писане у JavaScript-у, а могу се покренути у овкиру NodeJS-а на окружењима попут OS X, Microsoft Windows, Linux, FreeBSD и IBM i.

NodeJS обезбеђује архитектуру вођену догађајима не блокирајући улазно/излазни API који оптимизује пропусност и проширивост апликације. Ове технологије се обично користе за веб апликације које се извршавају у реалном времену.

NodeJS користи Google V8 JavaScript механизам за извршавање кода, и велики проценат базних модула је оригинално написан у JavaScript-у. NodeJS садржи уграђену библиотеку која дозвољава апликацијама да се понашају као веб-сервери без софтвера као што је Apache HTTP сервер.

NodeJS се полако усваја као серверска платформа, користе је неке од великих компанија као што су: Groupon, SAP, LinkedIn, Microsoft, Yahoo!, Walmart, Rakuten, PayPal, Voxer и GoDaddy [84]. Више о NodeJS-у може се наћи у [85][86].

4.2.5 Redis

Редис (енг. Redis) је унутар меморијска даљинска база података, која пружа високе перформансе, репликацију и јединствени модел података за креирање платформе за решавање проблема. Подржавајући пет различитих типова структура података, Редис прилагођава разноврсне проблеме који се природно могу мапирати у оно што Редис нуди, омогућавајући решења проблема без концептуалне гимнастике коју захтевају друге базе података. Додатне функције, као што су репликација, постојаност, и sharding[34]), на страни клијента омогућавају Редис-у погодну проширивост ка прототипу система, све до неколико стотина гигабајта података и милиона захтева у секунди. Многи програмски језици пружају могућност везивања на Редис, попут: ActionScript, C, C++, C#, Clojure, Common Lisp, Dart, Erlang, Go, Haskell, Haxe, Io, Java, JavaScript (Node.js), Lua, Objective-C, Perl, PHP, Pure Data, Python, R, Ruby, Scala, Smalltalk и Tcl-a.
Типови података:
Редис мапира кључеве ка многим вредносним типовима. Кључна разлика између Редиса и других структуираних складиштних система је да Редис не подржава само стрингове већ и апстрактне типове попут:

- Листе стрингова
- Скупова стрингова (колекције непонављајућих и несортираних елемената)
- Сортираних скупова стрингова (сортиране по децималном броју који се назива поен)
- Хеш табела код које су кључеви и вредности стрингови.

[34] Sharding - метод којим партиционишемо наше податке различитим комадима. У овом случају, ми партиционишемо наше податке на основу идентификатора уграђеног у кључу, заснованог на хешу кључева или некој комбинацији ова два. Партиционисањем података, можемо да складиштимо и довлачимо податке са више машина, које могу да дозволе линеарну сразмеру у перформансама за одређене домене проблема.

- Битмапе (скуп битских операција на стринговима) и Хипер-лог-логови (структура података базирана на вероватноћи која се користи за пребројавање јединствених ствари, углавном се ово односи на процењивање кардиналности скупа).

Више о Редису може се наћи у [87][88] и на [89].

Глава 5

Архитектура

У овом одељку биће представљена замишљена архитектура на дистрибуираном систему са више кафка инстанци, независних сервиса који се налазе на више машина и више Редис база података. Такође, биће представљена имплементирана архитектура која се извршава на једној машини.
Архитектуру можемо поделити на неколико делова:

- Клијентски део 4.1:
  - AngularJS 4.1.1
  - Bootstrap 4.1.3
  - KaTeX 4.1.4
- Серверски део 4.2
  - REST API 4.2.1
    - PHP 4.2.2
    - Laravel 4.2.2.1
  - Посредник порука 4.2.3
    - Apache Kafka 4.2.3.1
- Сервиси 4.2.4
  - C 4.2.4.1
  - Python 4.2.4.2

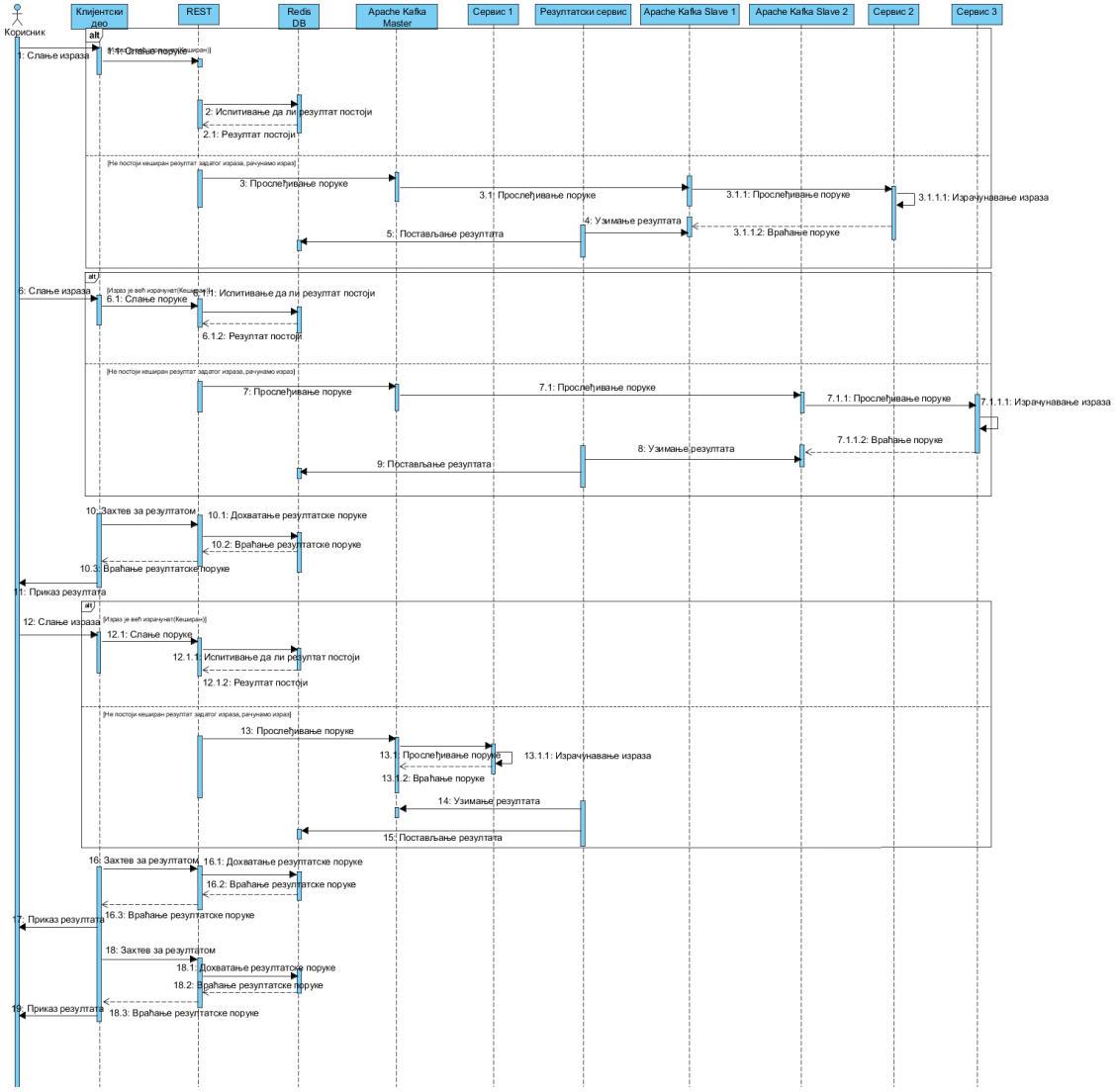
- Ruby 4.2.4.3
- Java 4.2.4.4
- NodeJS (JavaScript) 4.2.4.5

5.1 Дистрибуирана архитектура

Одабиром овакве архитектуре као и Кафке, чија је главна особина дистрибуираност, стекли смо све услове да се имплементирана платформа може извршавати на више различитих машина.
Извршавање на више различитих машина подразумева репликацију Кафка сервера. Сама Кафка инстанца, тј. његов предводник, емитоваће пристигле поруке на пратиоце и тако ће вршити синхронизацију свих чворова на свим машинама. Сервиси који су закачени на различите чворове радиће несметано и несметано примати поруке са одређеног Кафка сервера. Уколико дође до претрпавања захтевима, покрећемо више инстанци сервиса који ће паралелно обрађивати све пристигле поруке.

5.1.1 Ток извршавања

Корисник приступа страници и уноси свој израз, AngularJS креира поруку и шаље асинхрони POST захтев на серверски REST API. Серверски REST, Ларавел део врши испитивање пристиглог израза и након утврђивања на коју Кафка тему треба проследити поруку, врши слање захтева свим Кафка серверима (чворовима). Кафка чворови прихватају поруку, прослеђују поруку неком од слободних сервиса који врше обраду таквих израза и резултат прослеђују назад на Кафку. Потом сервис који је задужен за преузимање порука са теме са резултатима врши постављање истих на Редис базу. AngularJS врши нови GET захтев, добија резултат са Редиса, позива KaTeX и врши приказ резултата. Секвенцијални дијаграм који описује ток извршавања можемо видети на слици 5.1.



Слика 5.1: Секвенцијални дијаграм, тока извршавања

5.1.2 Дијаграм постављања

На дијаграму постављања 5.2 дат је графички приказ свих компонената на примеру дистрибуираног система са два Кафка сервера (чвора) и више различитих инстанци сервиса који раде на различитим машинама.

5.1.3 Репликација Редис-а

Репликација Редиса је веома једноставна за коришћење и конфигурисање главно-подређене (енг. master-slave) репликације која дозвољава подређеном Редис чвору да буде тачна копија мастер чвора. Неке од важних чињеница Редис репликације су:

- Редис користи асинхрону репликацију.
- Један мастер чвор може имати више подређених чворова.
- Подређени чворови могу прихватати везе других подређених чворова. Поред тога што се више подређених чворова може повезати на један главни сервер, они могу бити повезани и на остале подређене графолике структуре.
- Редис репликација је неблокирајућа на подређеној страни. Ово значи да ће главни сервер наставити да управља упитима када један или више подређенх чворова изводе иницијалну синхронизацију.
- Репликација је такође неблокирајућа на страни подређеног чвора. Док подређени чвор изводи иницијалну синхронизацију, може управљати упитима користећи старије верзије скупова података, претпостављајући да је Редис сервер конфигурисан за тако нешто у конфигурационом фајлу redis.conf. Иначе, можемо конфигурисати подређене чворове Редиса да враћају грешку клијенту уколико репликација пође по злу. Међутим, након иницијалне синхронизације, најстарији скуп података мора бити обрисан док најновији мора бити учитан. Подређени чвор ће блокирати надолазећа повезивања током ових операција.
- Репликација се може користити за скалабилност у смислу да имамо више подређених чворова само за читајуће упите (нпр. тешке SORT операције могу бити искључене на подређеним серверима) или просто у смислу редундантности података.
- Могуће је користити репликацију како бисмо избегли цену да имамо тежак главни сервер који уписује целокупне скупове података на диск.

Сигурна репликација када главни чвор има искључену перзистенцију. У подешавањима где се користи Редис репликација, строго се препоручује да је опција перзистентности укључена на главном серверу или када то није могуће. На пример, услед кашњења инстанце би требало конфигурисати да избегавају аутоматско ресетовање. Како би се боље разумело зашто је опасно да главни сервери са искљученом перзистенцијом имају укључен мод аутоматског ресетовања, представићемо следећи пример где су подаци избрисани са главног сервера и његових подређених сервера:

1. Имамо подешавање где чвор А има улогу главног сервера, са искљученом перзистенцијом, и чворове Б и Ц који реплицирају чвор А.

2. Дешава се пад, међутим постоји систем аутоматског ресетовања који ресетује процес. Међутим, пошто је перзистенција искључена, чвор се рестартује са празним скупом података.

3. Чворови Б и Ц реплицирају А, који је такође празан, тако да се дешава да сви подаци бивају уништени.

Када се користи Redis Sentinel[1] за високу доступност, такође се искључује перзистенција на главном чвору, заједно са аутоматским ресетовањем процеса то може бити опасно. Нпр. главни чвор се може рестартовати јако брзо тако да Sentinel неће ни детектовати пад, као што је описана претходна ситуација. Сваки пут када је безбедност података битна и када се користи репликација, главни сервер се конфигурише без перзистенције, а аутоматско ресетовање инстанци требало би да буде угашено.

Како се врши репликација. Ако подесимо пратиоца, по повезивању он шаље синхронизациону команду. Нема везе да ли се дата повезује први пут или врши поновну конекцију. Главни чвор стартује позадинско снимање и врши баферовање свих пристиглих команди које врше неку модификацију скупа података. Када се заврши позадинско снимање, главни чвор премешта фајл базе података на пратиоца који све то снима на диск и потом учитава у меморију. Главни чвор потом свим пратиоцима шаље све бафероване команде. Ово се извршава као низ команди и у истом формату као сам Редис протокол.

Делимична ресинхронизација. ради тако што се креира унутар меморијски заостатак репликационог тока на главном чвору. Главни и пратећи чворови договарају се око репликационог офсета и идентификације главног чвора, тако да када веза падне, помоћник се поново конектује и пита главни чвор за настављање репликације. Претпостављајући да је идентификација главног чвора остала непромењена и да је договорени офсет доступан у репликационом заостатку, репликација се наставља од тачке где се стало. Ако ниједан од ових услова није испуњен, врши се пуна ресинхронизација.

Репликација без диска. У нормалним условима пуна ресинхронизација тражи да се направи RDB фајл (бинарна репрезентација унутар меморијског складиштења),

[1]Redis Sentinel - систем дизајниран за помоћ у управљању Редис инстанцама.

потом поновно учитавање RDB фајла са диска у циљу достављања података осталим подређеним чворовима. Са спорим дисковима ово може бити јако захтевна операција за главни чвор. Редис верзија 2.8.18 ће бити прва која ће имати експерименталну подршку репликације без диска. У овој поставци, дечији процеси директно шаљу RDB фајлове преко мреже подређених чворова, без коришћења диска као посредног складишта. Ова карактеристика се тренутно сматра експерименталном [90].

5.2 Имплементирана архитектура

Имплементирана архитектура је јако слична описаној дистрибуираној архитектури, једини изузетак је што нема репликације Кафка сервера, већ постоји само један, такође се извршавање свих сервиса врши на истој машини.
Дијаграм постављања на слици 5.3 појашњава начин организације имплементиране архитектуре.

Ток извршавања. Корисник на почетку врши захтев за учитавањем странице, након тога уноси израз и исти шаље клијентском делу. Клијентски део узима израз, врши припрему поруке и шаље на серверски REST API. Порука која се шаље серверском делу је у JSON формату, пример поруке:

```
{
    "expression":"integrate(x**15+ x + 1, x)"
}
```

Оваква порука се на сервер шаље POST **server.php/expression/** захтевом. Слање поруке се врши асинхроно, након слања POST захтева, шаље се:
GET **server.php/expression/integrate**$(x**15 + x + 1, x)$ захтев. Након што серверски REST део прими поруку, врши се проверавање да ли је можда такав израз већ израчунат. Уколико постоји одмах се враћа резултат, без потребе за било каквим израчунавањем. Уколико резултат није до сада израчунат врши се испитивање израза тј. одређивање на који од Кафка тема је потребно поставити израз. Испитивање послатог израза вршимо регуларним изразима (енг. Regular Expression). Након утврђивања на коју од тема је потребно послати поруку, вршимо слање поруке на Кафку.

Ово се ради из PHP-а, помоћу Кафка PHP клијента доступног на [91], овакве алате спомињали смо у 2.2.5. Након што порука бива послата на Кафку, Кафка прослеђује свим сервисима који су претплаћени на ту тему. Сервиси примају поруку, десеријализују (7.1) JSON поруку, врше израчунавање израза, додају резултат у поруку, серијализују (7.1) поруку и враћају на резултатску тему. Сервис који је претплаћен на резултатску тему, врши преузимање порука и постављање на Редис. Потом наведени GET захтев врши приступање Редису и преузимање поруке са резултатом. Овај процес је описан секвенцијалним дијаграмом на слици 5.4.

5.2.1 Клијентска страна

Као што смо описали у одељку 4.1, клијентски део се састоји из три дела. Bootstrap користимо за графички приказ, док AngularJS користимо за слање, прихватање, исписивање порука и израза. Пример AngularJS контролера који врши слање захтева приказан је у делу кода 5.2.1.

```
kafkalatorApp.controller('expressionController', ['$scope', '
   postExpressionFactory',
'getExpressionFactory', '$location', '$q', '$http',
    function ($scope, postExpressionFactory, getExpressionFactory,
   $location, $q, $http)
    {
        $scope.results = [];
        $scope.postNewExpression = function () {
            $http.post('../server.php/expression', {expression: $scope.
   expression})
            .success(function (e) {
                        if (e.hasOwnProperty('expressionId')) {
                            //wait funkcija
                            $http.get('../server.php/expression/' + e.
   expression)
                            .success(function (e) {
                                $scope.results.push(e);
                            });
                        }
                        else {
                            $scope.results.push(e);
                        }
                    });
        };
```

```
    }]);
```

Скрипта 5.1: AngularJS контролер

Након слања захтева приказивање се врши кодом:

```
<h2>Results</h2>
    <div class="list-group" ng-repeat="res in results">
        <a class="list-group-item">
            Expression:<div katex>res.expression</div></br>
                Result:<div katex>res.resultLatex</div>
            <h4 class="list-group-item-heading">{{res.expression}}</h4>
            <div>
                <h4 class="list-group-item-heading" ng-click="showme =
   true">
                {{res.result}}</h4>
                <div class="wrapper">
                    <h4>Latex code: </h4>
                    <button ng-click="showme = false">Show</button>
                    <button ng-click="showme = true">Hide</button>
                    <div class="well well-sm" ng-show="showme" ng-hide="
   showme">
                            {{res.expressionLatex}}
                    </div>
                    <div class="well well-sm" ng-show="showme" ng-hide="
   showme">
                            {{res.resultLatex}}
                    </div>
                </div>
            </div>
        </a>
    </div>
```

Приказивање израза помоћу KaTeX-a остварује се модулом приказаним у скрипти 5.2.1.

```
angular.module('katex-module', []).value('mathDefaults', {
    center: true,
    fallback: true
}).directive('katex', ['mathDefaults', function (mathDefaults) {
        function render(katex, text, element) {
```

```
            try {
                katex.render(text, element[0]);
            }
            catch (err) {
                // MathJax rezerva
                if (mathDefaults.fallback) {
                    require(['mathjax'], function (mathjax) {
                        if (text.substring(0, 15) === '\\displaystyle {'
)
                            text = text.substring(15, text.length - 1);
                        element.append(text);
                        mathjax.Hub.Queue(["Typeset", mathjax.Hub,
element[0]]);
                    });
                } else
                    element.html("<div class='alert alert-danger' role='
alert'>"
                    + err + "</div>");
            }
        }
    }]);
```

Скрипта 5.2: Приказ резултата

Модул врши преузимање израза и прављење LaTeX репрезентације задатог израза и резултата израза, уколико KaTex није у могућности да прикаже израз, посеже се за сличном али споријом библиотеком - MathJax. Секвенцијални дијаграм на слици 5.5 приказује начин рада клијентског дела.

5.2.2 Серверска страна

Контролер на серверској страни преузима поруку, и регуларним изразима испитује на коју од тема поруку треба послати. Фрагмент кода који ради испитивање да ли је неки израз аритметички и врши слање на аритметичку тему приказан је у 5.2.2.

```
class ExpressionController extends BaseController {
    //provera da li je izraz aritmeticki
    private function arithmeticCheck($expression) {
            $re =   "/^(?<expr>(?:\\d++|\\((?&expr)\\))(?:[-+*\\/](?&
expr))*)$/mx";
        if (preg_match($re, $expression)) {
```

```
                return true;
            } else {
                return false;
            }
        }
        //slanje izraza na aritmeticku temu
        if ($this->arithmeticCheck($json->expression)) {
            $kafka = new MessageProducer($message->encode(), 'arithmetic
    ');
            $kafka->sendArithmeticMessage();
            $response = Response::make($message->encode(), 200);
            $response->header('Content-Type','application/json');
            return $response;
        }
```

Скрипта 5.3: Серверски контролер израза

Класни дијаграм на слици 5.6 приказује серверску организацију у оквиру Ларавел оквира.

5.2.3 Сервиси

Сервиси су имплементирани на језицима који су описани у одељку 4.2.4. У овом одељку приказаћемо неке од њих.

5.2.3.1 Python

Сервис користи SymPy библиотеку за израчунавање израза. Израчунавање израза врши се командом $S(izraz)$.

```
from sympy import *
from kafka import *
import json
import sys

#inicijalizacija kafka klijenta
kafka = KafkaClient("localhost:9092")
#inicijalizacija proizvodjaca za zadatu temu
consumer = SimpleConsumer(kafka,"sympy","test1")
#inicijalizacija potrosaca
```

```
producer = SimpleProducer(kafka, async = True)
for message in consumer:
    #preuzimanje poruke
  json_message = message.message.value;
  #deserijalizacija poruke
  json_encoded = json.loads(json_message)
  exp = json_encoded["expression"]
  try:
      #pozivanje SymPy-a
    result = S(exp)
    #umetanje rezultata u poruku
    json_encoded['result'] = str(result)
    #serijalizacija poruke
    json_message = json.dumps(json_encoded)
    #slanje poruke na rezultatsku temu
    producer.send_messages("result",json_message)
    print json_message
  except:
      #obrada greske
    print "Error parsing expresion:" + exp
    print sys.exc_info()
kafka.close()
```

Скрипта 5.4: Пајтон сервис

5.2.3.2 Python - Redis

Овај сервис врши уписивање резултата на Редис као и LaTex репрезентацију свих задатих израза и резултата.

```
from kafka import *
import redis
import json
from sympy import *
from sympy.abc import *

#konvertovanje zaraza u LaTex reprezentaciju
def convertToLatex(expression):
        a = sympify(expression)
        return latex(a)

def addFields(json_message):
```

```
        json_encoded = json.loads(json_message)
  if 'function' in json_encoded['expression']:
                exp = json_encoded['expression']['function'] +
                "(" + json_encoded['expression']['a'] + ","
                + json_encoded['expression']['b'] + ")"
                json_encoded['expression'] = exp
                json_encoded['expressionLatex'] = exp
                json_encoded['resultLatex'] = json_encoded["result"]
                return json.dumps(json_encoded)
  if "complex(" in json_encoded['expression']:
                json_encoded['expressionLatex'] = json_encoded["
   expression"]
                json_encoded['resultLatex'] = json_encoded["result"]
                return json.dumps(json_encoded)
        json_encoded['expressionLatex'] = convertToLatex(json_encoded["
   expression"])
        json_encoded['resultLatex'] = convertToLatex(json_encoded["
   result"])
        return json.dumps(json_encoded)

r_server = redis.Redis('localhost')
kafka = KafkaClient("localhost:9092")
consumer = SimpleConsumer(kafka,"sympy","result")

for message in consumer:
  json_message = message.message.value
  try:
    json_message = addFields(json_message)
    json_encoded = json.loads(json_message)
    #uzimanje izraza
    exp = json_encoded["expression"]
    #uzimanje rezultata
    result = json_encoded["result"]
    #postavljanje rezultatskodId-a kao kljuca poruke
    r_server.hset('results',expId,json_message)
    #postavljanje poruke kao vrednosti i izraza kao kljuca
    r_server.hset('cache',exp,json_message)
    print json_message
  except:
    print json_message
kafka.close()
```

Скрипта 5.5: Пајтон-Редис сервис

5.2.3.3 Ruby

Овај сервис користи Poseidon клијент за повезивање на Кафку, који је доступан на [92], као и библиотеку за израчунавање математичких израза. Овај сервис нам конкретно служи за израчунавање свих тригонометријских израза. Класа која врши израчунавање приказана је у скрипти 5.2.3.3.

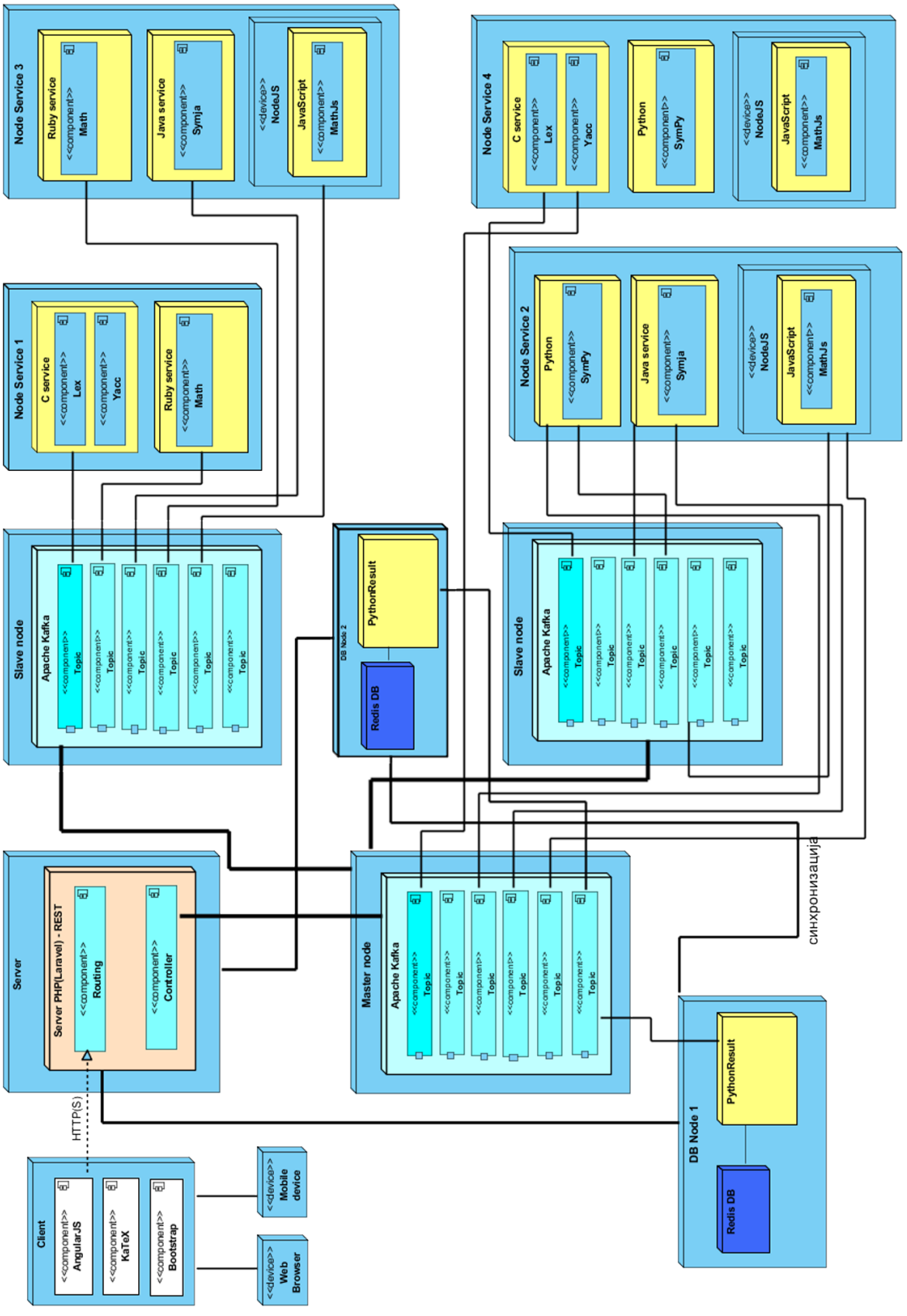
```
include Math
require 'json'
#definisanje Trigonometrijske klase koja vr i izra unavanje Math.
    komandi
class Trigonometry
    #konstruktor
  def initialize(command)
    @command = 'Math.' + command
    begin
      #izvrsavanje komande
      @result = eval(@command)
    rescue
      #slucaj greske
      @result = "error"
    end
  end
    #rezultatski getter
  def result
    @result
  end
    #funkcija za stampanje rezultata
  def print_result
    puts @result
  end
end
```

Скрипта 5.6: Руби сервис

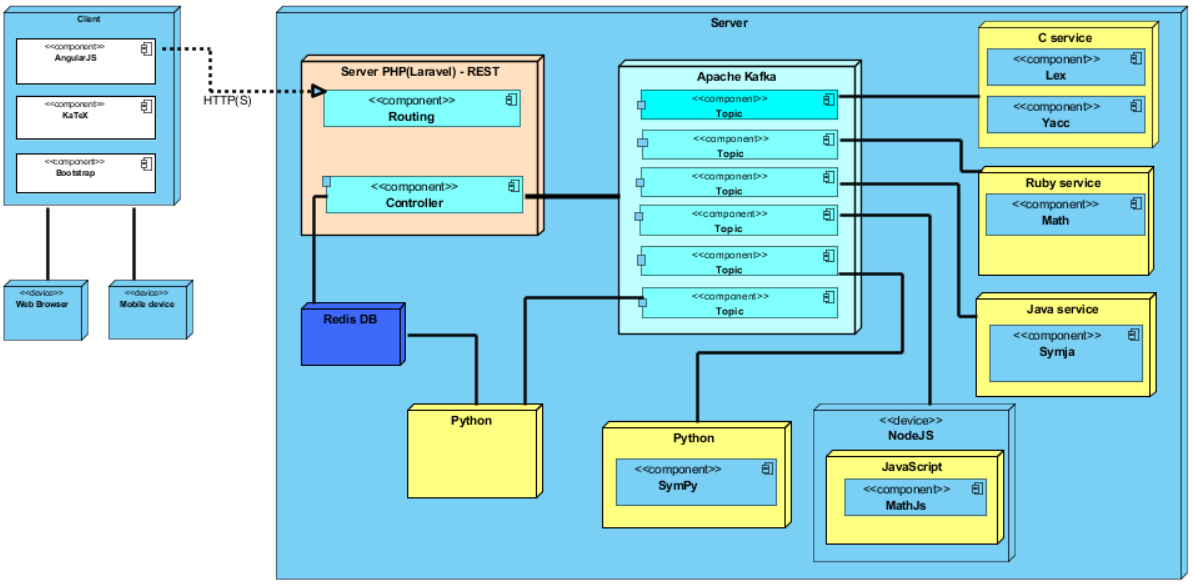
Повезивање на Кафку и слање порука:

```
require 'poseidon'
require 'json'
load 'Trigonometry.rb'

#inicijalizacija potrosaca, postavljanje njegovih argumenata
i povezivanje na odre enu temu
consumer = Poseidon::PartitionConsumer.new("ruby_consumer", "localhost",
    9092,
"trigonometry", 0, :earliest_offset)
#inicijalizacija proizvodjaca
producer = Poseidon::Producer.new(["localhost:9092"], "ruby_producer")

loop do
  begin
    #preuzimanje poruka
    messages = consumer.fetch
    messages.each do |m|
      #preuzimanje poruke
      a = JSON.parse(m.value)
      #izracunavanje izraza
      t = Trigonometry.new(a['expression'])
      #postavljanje rezultata
      a['result'] = t.result
      #serijalizacija
      m = a.to_json
      #slanje poruke na rezultatsku temu
      producer.send_messages([Poseidon::MessageToSend.new("result", m)])
    end
  rescue Poseidon::Errors::UnknownTopicOrPartition
    #hvatanje greske
    puts "Topic does not exist yet"
  end

  sleep 1
end
```

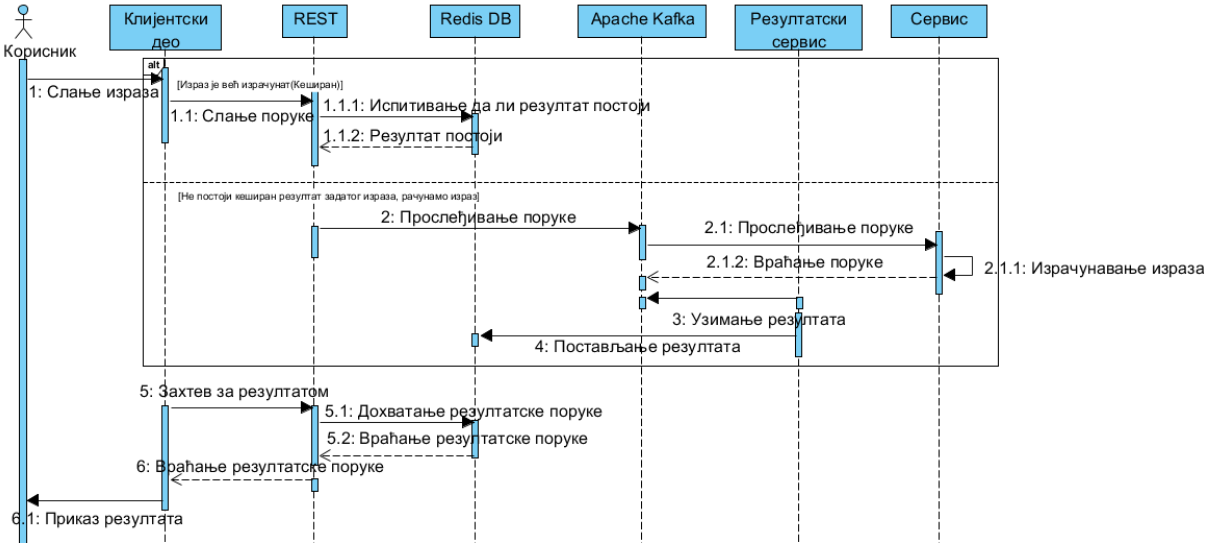
Скрипта 5.7: Руби сервис - слање порука



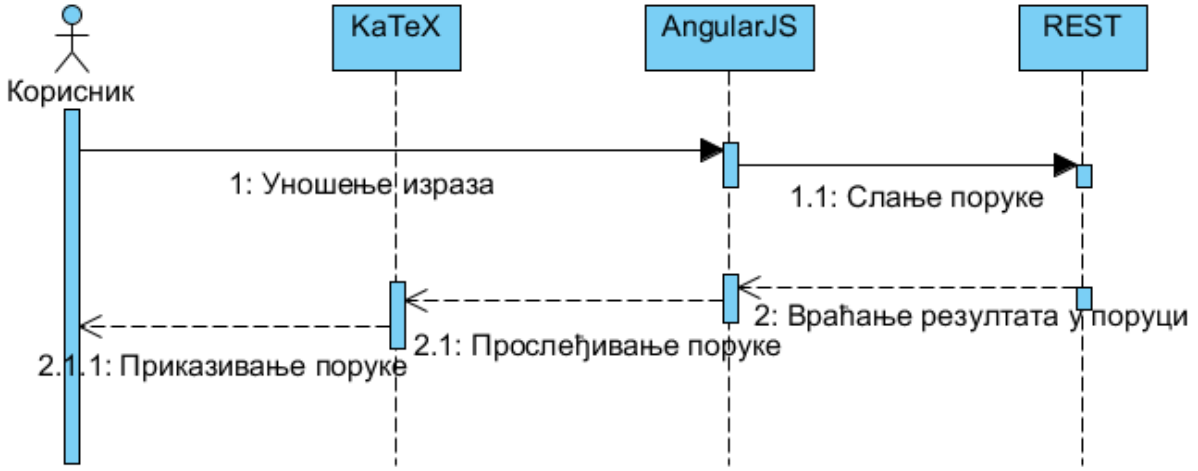
Слика 5.2: Дијатрам постављања, приказ дистрибуиране архитектуре



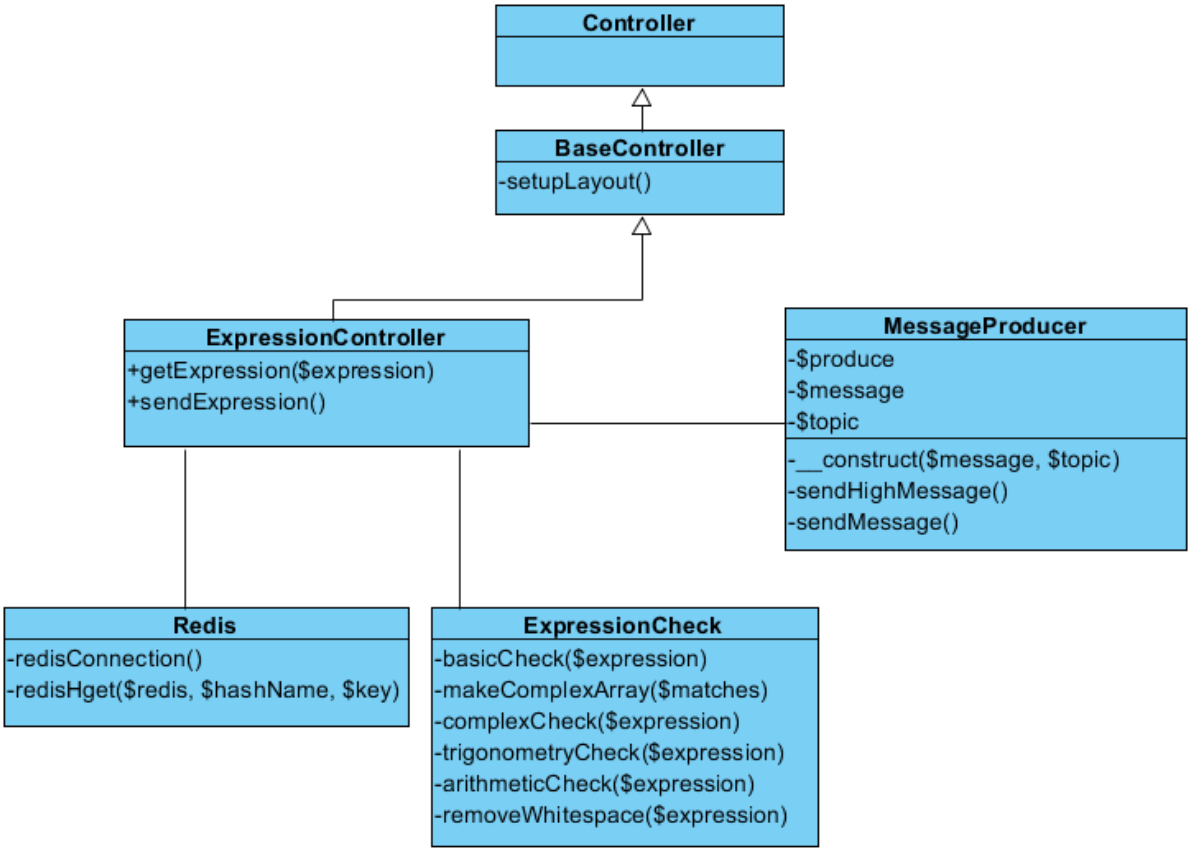
Слика 5.3: Дијаграм постављања имплементиране архитектуре



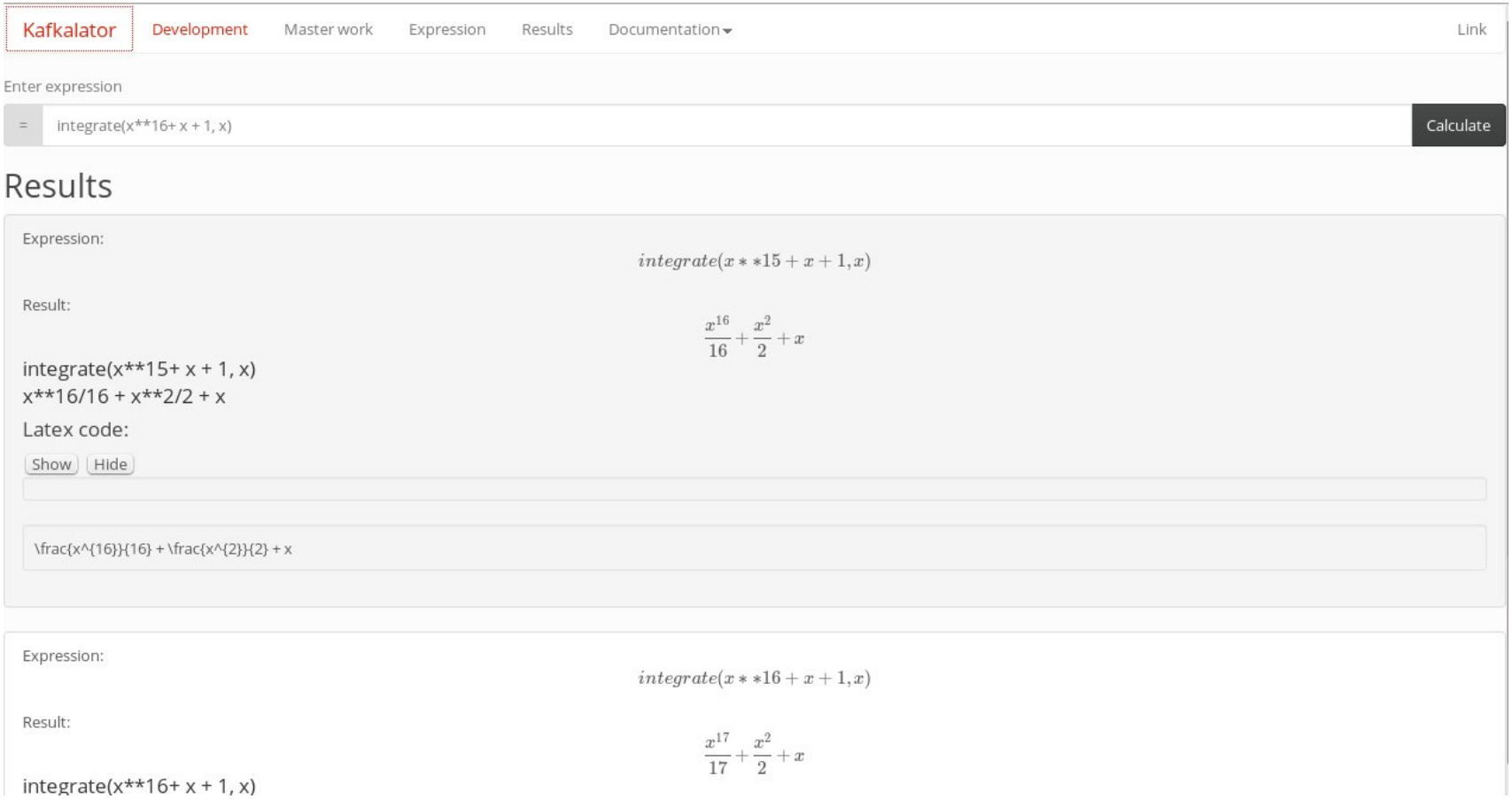
Слика 5.4: Секвенцијални дијаграм, тока извршавања



Слика 5.5: Секвенцијални дијаграм, тока извршавања на клијентској страни



Слика 5.6: Класни дијаграм



Слика 5.7: Пример рада платформе

Глава 6

Закључак

Примарни циљ овог рада је представљање идеје и принципа микросервиса.
Искуства и радови Мартина Фаулера, Џејмса Луиса и великих светских компанија попут Netflix-а, Амазона, Гардијана-а само потврђују да микросервиси имају светлу будућност, поготово у развоју великих (енг. enterprise) апликација. У овом раду имплементиран је један такав систем који нам је помогао да дођемо до истог закључка. При имплементацији није било никаквих проблема или сумње у овакав архитектурални приступ. Наравно да микросервисна архитектура не представља универзално решење за све могуће проблеме, али се показује да је свакако стабилна, рашчлањена, одржива и хоризонтално скалабилна архитектура која лако може да одговори свим захтевима и која ће временом вероватно доказати своје предности у односу на монолитни архитектурални стил. Све веће интересовање и примена у пракси микросервисног архитектуралног приступа показује да овај приступ постаје устаљен код већих софтверских компанија.
Акценат овог рада је представљање једног оваквог архитектуралног приступа, као и низа технологија и библиотека које су коришћење за баратање симболичких израза. Такође, рад показује како различите технологије, програмски језици и решења функционишу кохерентно. У имплементираној платформи коришћена је Кафка као посредник. Наравно, микросервиси не морају бити засновани на њој, већ то може бити било који други вид посредника, тока података или неког другог приступа.

6.1 Будући рад

Позитивна страна оваквог микросервисног приступа је што даљи рад на овако здраво постављеној архитектури неће изискивати пуно посла или изненадних проблема. Имплементација нових сервиса у оваквом приступу тече јако брзо. У даљем раду планира се имплементација:

- симболичких алгоритама у неким од скрипт језика као и премештање ових скрипти на клијентски део, тј. селидба израчунавања на клијентску страну
- сервиса који управља дељењем преузетих клијентских сервиса. Дистрибуција сервиса на клијенте довешће до тога да умрежени клијенти могу међусобно да деле преузете сервисе и функционишу попут чворова
- писање неке врсте балансера који ради на управљању процеса са аспекта паралелизације. Нпр. уколико стигне n захтева за израчунавањем на једном сервису, балансер ради на активирању n инстанци једног истог или више сервиса који преузимају извршавање у константном времену
- графичког приказивања резултата на клијентској страни (помоћу jqPlot библиотеке)
- матичне апликације за мобилне платформе која се везује на постојећи API
- тестирање брзина сервиса који нуде сличну функционалност потом распоређивање задатих израза сервисима који брже раде
- сервиса за динамичку геометрију

Глава 7

Речник термина

7.1 Речник термина

Ред порука (енг. Message queue)

Софтверско-инжењерска компонента која се користи за међупроцесну комуникацију (енг. inter process communication) или међунитну (енг. inter-thread) комуникацију у оквиру истог процеса. Међупроцесна комуникација представља размену порука, дељење ресурса и синхронизацију унутар једног рачунарског система, док међунитна комуникација такође представља размену порука али унутар једног процеса.

Агент порука (енг. Message broker)

Архитектурални шаблон за валидацију, трансформацију и рутирање порука. Он прима поруке са разних страна, одређује тачно одредиште и путању кроз коју потом шаље поруку.

Преношење репрезентације ресурса (енг. REST, Representational State Transfer)

REST је координисан скуп ограничења примењен на дизајнирању компоненти у дистривуираном систему хипермедија (7.1) који може довести до перформативније и одрживије архитектуре. Више о REST-у у поглављу 4.2.1.

Хипермедија (енг. Hypermedia)

Хипермедија је нелинеарно средство за представљање информација које укључују аудио и видео садржаје као и текстове и хиперлинкове (7.1).

Хиперлинкови (енг. Hyperlinks)

Хиперлинкови представљају референце (везе) ка неким другим ресурсима.

RabbitMQ

RabbitMQ је софтверски агент порука (енг. message broker), у неким случајевима се назива и посредник оријентисан на порукама (енг. Message-oriented middleware) отвореног кода који имплементира протокол напредних редова порука (енг. Advanced Message Queuing Protocol, скр. AMQP). RabbitMQ сервер је написан у Ерланг програмском језику и изграђен је на Open Telecom Platform, платформи за кластеровање и опоравке од падова. Клијентске библиотеке за рад са агентом су доступне у скоро свим програмским језицима.

ZeroMQ

ZeroMQ је високо перформантана асинхрона библиотека која се користи за развој скалабилних, дистрибуираних или конкурентних апликација. Обезбеђује ред порука, али за разлику од посредника оријентисаног на порукама, ZeroMQ може да функционише без одређеног агента.

Постављање софтвера (енг. Software Deployment)

Постављање софтвера представља све активности које чине софтверски систем употребљивим. Генерални процес постављања састоји се од неколико активности које су у међусобном односу са могући променама између њих. Ове активности се могу појавити на производној и потрошачкој страни или на обе. Пошто је сваки софтверски систем јединствен, прецизни процеси или процедуре унутар сваке активности се тешко могу дефинисати. Стога, постављање се интерпретира као генерални процес који мора бити прилагођен специфичним захтевима и карактеристикама.
Активности постављања су:

1. **пуштање** (енг. release) - операције које се спорводе у припреми система за монтажу и пренос клијенту
2. **инсталирање и активирање**
3. **деактивирање** - гашење система
4. **прилагођавање** - процес модификовања софтверског система који је претходно инсталиран
5. **ажурирање**
6. **уграђивање** - механизми за инсталацију исправки
7. **праћење верзија** - модули који помажу кориснику да пронађе и инсталира исправке свог софтверског система **деинсталација**
8. **повлачење** (енг. retire) - софтверски систем је означен као застарео и подршка од стране произвођача је повучена, ово уједно означава и крај животног циклуса софтверског система.

Скалирање, скалабилност (енг. Scalability)

Способост система, мреже или процеса да рукује све већем обиму посла или способност да се увећа при таквом расту захтева. Систем чије перформансе расту након додавања новог хардвера, пропорционално потребном капацитету назива се скалабилан систем. Методе за додавање нових ресурса се деле на: **хоризонтално**(додавање

нових чворова у систем, попут додавања нових рачунара) **вертикално**(додавање нових ресурса једном чвору, попут додавања новог процесора, меморије и сл). Скалирање сервиса у микросервисној архитектури подразумева покретање више инстанци сервиса за одрађивање више послова истовремено.

Континуирана испорука (енг. Continous delivery)

Континуирана испорука је приступ у софтверском инжењерству у ком тимови одржавају продукцију вредног софтвера у кратким циклусима и притом дају сигурност да се софтвер може поуздано пустити у сваком тренутку.

Континуирана интеграција (енг. Continous integration)

Континуирана интеграција у софтверском инжењерству представља праксу да се неколико пута дневно врши спајање свих кодова које пишу програмери у оквиру тима или компаније. Овај термин је усвојен као део екстремног програмирања који подржава и спајање неколико десетина пута дневно.

Платфома као сервис (енг. Platform as a service)

Платформа као сервис спада у категорију сервиса рачунарства у облаку (енг. cloud computing services) која пружа платформу која дозвољава корисницима да постављају, покрећу и управљају веб апликацијама без потешкоћа у изградњи и одржавању инфраструктуре која се обично повезује са развојем и лансирањем (покретањем) апликације.

Монго база података (енг. MongoDB)

Представља једну од водећих NoSQL база података. Написана у језику C++, MongoDB чува податке као JSON документе са динамичким шемама. Чини интеграцију података у многим апликацијама једноставнијом и брзом, прихваћена је као софтвер задњег плана од стране бројних веб-сајтова и сервиса попут eBay, Foursquare, SourceForge и New York Times-a.

Riak

Дистрибуирана, кључ-вредност, NoSQL база података, која пружа високу доступност, отпорност на грешке, оперативну једноставност и скалабилност. Поред верзије отвореног кода, долази у подржаној пословној верзији и као верзија за складиштење на облаку која је идеална за окружења у овиру рачунарства у облаку. Riak имплементира неке од принципа Амазоновог Dynamo (систем за складиштење) система са огромним утицајем теореме постављене од стране др. Ерика Брувера (енг. Eric Brewer) названом Бруверовом теоремом или CAP теоремом. Riak је написан у програмском језику Ерланг. Има способност репликација које су отпорне на падове и аутоматску дистрибуцију података преко чворова у циљу постизања веће еластичности и више перформанси.

Бруверова теорема (енг. CAP theorem)

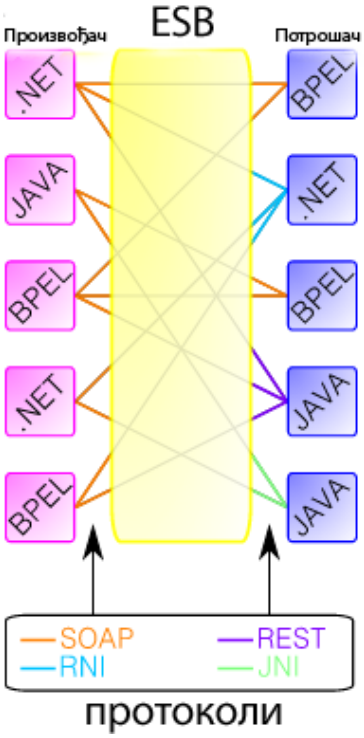
Бруверова теорема наводи да је немогуће да један дистрибуиран рачунарски систем пружи све три следеће гаранције:

- **Конзистентност (енг. Consistency)** - сви чворови виде исте податке у истом временском тренутку
- **Доступност (енг. Availability)** - гаранција да сваки захтев добија одговор о томе да ли је послати захтев био успешан или не
- **Толерантне партиције (енг. Partition tolerance)** - систем наставља да ради упркос арбитражном губитку порука, паду целог или неког од делова система.

Пословно сервисна магистрала (енг. Enterpise Service Bus)

Пословно сервисна магистрала је архитектурални модел који се користи при дизајнирању и имплементацији комуникације између софтверских апликација које међусобно интерагују у сервисно орјентисаној архитектури (енг. Service Oriented Architecture). Мотивација за дизајнирање оваквог модела је пронаћи стандардан, структуиран и опште наменски концепт за описивање имплементације лабаво повезаних софтверских компоненти (сервиса), за које се очекује да буду независно постављени, покретљиви,

хомогени, диспарантни (неједнаки, нескладни) унутар мреже. Пословно сервисна магистрала је усвојена на вебу као суштински дизајн за мрежу и уобичајен имплементациони шаблон сервисно-оријентисане архитектуре (енг. SOA).



Слика 7.1: Преглед ESB-а

Сервисно оријентисана архитектура (енг. SOA)

COA (енг. Service Oriented Architecture) је архитектурални стил који промовише коришћење лабаво повезаних сервиса ради обезбеђивања високе флексибилности на интероперабилан и технолошки независан начин. Флексибилност архитектуре представља њену способност да се као целина прилагођава променама у окружењу и примењује у новим околностима, као и способност елемената архитектуре да се прилагођавају променама у окружењу, примењују у новим околностима и примењују у другим новим пројектима. Флексибилност је блиска поновној употребљивости. COA је архитектурална парадигма која се примарно односи на дизајн софтвера (а не на технологију) [93].

Принцип дизајна вођеног доменом (енг. Domain-Driven Design)

Принцип дизајна вођеног доменом је приступ развоју софтвера за комплексне потребе повезујући имплементацију на еволутивни модел. Премисе принципа дизајна вођеног доменом су:

- Примарни фокус пројекта је постављање фокуса на домене језгра и логике.
- Модел домена се темељи на сложеном дизајну.
- Покретање креативне сарадње између техничких стручњака и стручњака домена на непрестаном детаљисању о концептуалном домену који се бави доменским проблемима.

Термин принципа дизајна вођеног доменом је увео Ерик Еванс у истоименом делу [11].

Убризгавање зависност (енг. Dependency injection) Софтверски шематски план који имплементира контролну инверзију (енг. inversion of control) за софтверске библиотеке.

Споразум на нивоу услуга (енг. Service-level agreement) Споразум на нивоу услуга представља део сервисног договора где је сервис формално дефинисан. Посебни аспекти сервиса су: обим, квалитет и одговорност. Ови аспекти се договарају између пружаоца услуга и корисника услуга. Једна од заједничких карактеристика SLA је уговорено време испоруке.

Складиште - Репозиторијум (енг. Repository) Представља концепт у оквиру контроле ревизија у виду структуре података, најчешће смештене на серверу, која, између осталог, садржи: скуп датотека и директоријума, историју промена у спремишту(репозиторијуму), историју промена у спремишту(репозиторијуму) и скуп комитованих објеката.

Договор на сервисном нивоу (енг. Service-level agreement) Представља део сервисног договора где је сервис формално дефинисан. Посебни аспекти сервиса као што су домен, квалитет, задужења се уговарају између сервисног снабдевача и корисника сервиса. Заједничка карактеристика једног сервисног договора (SLA) је уговорено време испоруке, тј. време које је потребно сервису да испоручи одговор.

Тип интернет медија (енг. Interned media type) Представља дводелни идентификатор за формат документа на Интернету. Изворно, идентификатори су дефинисани у оквиру RFC 2046 стандарда за Интернет пошту у оквиру SMTP (енг. Simple Mail Transfer Protocol) протокола, али је њихово коришћење проширено и на друге протоколе као што су: HTTP (енг.Hypertext Transfer Protocol), RTP (енг. Real-time Transport Protocol) и SIP (енг. Session Initiation Protocol).

Серијализација Процес превођења структура података или стања објекта у формат који се може сачувати (нпр. у фајл или меморијски бафер, може бити послат преко мреже) а потом касније реконструисан на истом или другом окружењу.

Десеријализација Представља инверзни процес серијализације, екстраховање структуре података из низа бајтова.

Библиографија

[1] Џејмс Луис Мартин Фаулер. Microservices. 2014. URL http://martinfowler.com/articles/microservices.html.

[2] Jurgen Gerhard Joackim von Zurathen. *Modern Computer Algebra*. Cambridge University Press, New York, 2013.

[3] Rob Pike Brian W. Kernighan. *The Unix Programming Environment*. Prentice-Hall, Upper Saddle River, New Jersey, 1983.

[4] Мартин Фаулер. *Рефакторисање*. Addison-Wesley, Boston, 2003.

[5] Мартин Фаулер. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, 2002.

[6] Манифест Агилног Развоја Софтвера, Јун 19, 2015. URL http://agilemanifesto.org/iso/sr/.

[7] Sam Newman. *Building Microservices*. O'Reilly, Sebastopol, California, 2015.

[8] Melvin E. Conway. How do committees invent? *Datamation magazine*, 62(1):1–20, April 1968. URL http://www.melconway.com/research/committees.html.

[9] Хипатија: Студентски сервис Математичког факултета, Јун 15, 2015. URL hypatia.matf.bg.ac.rs:10333/StudInfo/scripts/studenti/prijavljivanjeFormular.

[10] Мартин Фаулер. Boundedcontext. 2014. URL http://martinfowler.com/bliki/BoundedContext.html.

[11] Eric Evans. *Domain-Driven Design:Tackling Complexity in the Heart of Software*. Prentice Hall, Upper Saddle River, New Jersey, 2003.

[12] David Farley Jez Humble. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, Boston, 2010.

[13] Docker. Docker, Март 04, 2015. URL https://www.docker.com.

[14] Wiki:. Docker, Март 04, 2015. URL https://en.wikipedia.org/wiki/Docker_(software).

[15] Docker. What is docker?, Март 04, 2015. URL https://www.docker.com/whatisdocker.

[16] InfoQ. Microservices, containers and docker, Март 04, 2015. URL http://www.infoq.com/news/2014/12/microservices-docker.

[17] Kent Beck. *Implementation Patterns*. Addison-Wesley, Boston, 2007.

[18] Brandon Byars. Enterprise integration using rest. Март 04, 2013. URL http://martinfowler.com/articles/enterpriseREST.html#versioning.

[19] InfoQ. *InfoQ eMag: Microservices*. InfoQ, Март 04, 2014. URL http://www.infoq.com/minibooks/emag-microservices.

[20] Joel S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. A K Peters, Ltd., Natick, Massachusetts, 2003.

[21] Joel S. Cohen. *Computer Algebra and Symbolic Computation: Elementary Algorithms*. A K Peters, Ltd., Natick, Massachusetts, 2002.

[22] SymPy Documentation. Март 04, 2015. URL http://docs.sympy.org/latest/index.html.

[23] Ronan Lamy. *Instant SymPy Starter*. Packt, Birmingham, UK, 2013.

[24] Symja, Март 05, 2015. URL https://bitbucket.org/axelclk/symja_android_library/wiki/Home.

[25] Wiki: Computer algebra system, Март 04, 2015. URL https://en.wikipedia.org/wiki/Computer_algebra_system.

[26] Саша Малков. Програмирање за веб: Архитектура веб апликација, Април 02, 2015. URL http://poincare.matf.bg.ac.rs/~smalkov/files/pveb.r338.2014/public/predavanja/PVeb.2014.04%20-%20arh.apl,veb.2.0.pdf.

[27] Wiki: Angularjs, Март 04, 2015. URL https://en.wikipedia.org/wiki/AngularJS.

[28] Саша Малков. Програмирање за Веб: Модел-поглед-контролер, Март 04, 2015. URL http://poincare.matf.bg.ac.rs/~smalkov/files/pveb.r338.2014/public/predavanja/PVeb.2014.04%20-%20arh.apl,veb.2.0.pdf.

[29] Hans Rohnert Peter Sommerlad Michael Stal Frank Buschmann, Regine Meunier. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, Hoboken, New Jersey, 1996.

[30] Анђелка Зечевић. Програмирање за Веб: Angularjs, Март 04, 2015. URL http://poincare.matf.bg.ac.rs/~andjelkaz/pzv/cas_angularJS/angularJS.pdf.

[31] Angularjs документација, Март 04, 2015. URL https://docs.angularjs.org/api.

[32] Shyam Seshadri Brad Green. *AngularJS*. O'Reilly, Sebastopol, California, 2013.

[33] Sandeep Panda. *AngularJS: Novice to Ninja*. Sitepoint, Melbourne, Australia, 2014.

[34] Ari Lerner. *ng-book - The Complete Book on AngularJS*. Fullstack, 2013.

[35] Jake Spurlock. *Bootstrap*. O'Reilly, Sebastopol, California, 2013.

[36] Christoffer Niska. *Extending Bootstrap*. Packt, Birmingham, UK, 2014.

[37] Bootstrap документација, Март 04, 2015. URL http://getbootstrap.com/.

[38] David Cochran. *Bootstrap Site Blueprints*. Packt, Birmingham, UK, 2014.

[39] Github katex, Март 04, 2015. URL https://github.com/Khan/KaTeX/blob/master/README.md.

[40] Katex документација, Март 04, 2015. URL https://khan.github.io/KaTeX/.

[41] Wiki: Server-side, Март 04, 2015. URL https://en.wikipedia.org/wiki/Server-side.

[42] Richard Rosen Leon Shklar. *Web Application Architecture: Principles, Protocols and Practices*. Wiley, Hoboken, New Jersey, 2009.

[43] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures, Докторска дисертација.* University of California, Irvine, 2000. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[44] Todd Friedrich. *RESTful Service Best Practices, Recommendations for Creating Web Services.* Pearson eCollege, London, United Kingdom, 2012. URL http://toddfredrich.com/restful-best-practices-v1-1.html.

[45] Саша Малков. Програмирање за Веб: Rest сервиси, Март 04, 2015. URL http://poincare.matf.bg.ac.rs/~smalkov/files/pveb.r338.2014/public/predavanja/PVeb.2014.05%20-%20soa,%20rest.pdf.

[46] Ian Robinson Jim Webber, Savas Parastatidis. *REST in Practice: Hypermedia and Systems Architecture.* O'Reilly, Sebastopol, California, 2010.

[47] Subbu Allamaraju. *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity.* O'Reilly, Sebastopol, California, 2010.

[48] David Heinemeier Hansson Leonard Richardson, Sam Ruby. *RESTful Web Services.* O'Reilly, Sebastopol, California, 2007.

[49] Wiki: Php, Март 04, 2015. URL https://en.wikipedia.org/wiki/PHP.

[50] Peter MacIntyre. *PHP: The Good Parts.* O'Reilly, Sebastopol, California, 2010.

[51] Lorna Jane Mitchell. *PHP Web Services: APIs for the Modern Web.* O'Reilly, Sebastopol, California, 2013.

[52] Matthew Turland Davey Shafik, Lorna Jane Mitchell. *PHP Master: Write Cutting Edge Code.* Sitepoint, Melbourne, Australia, 2011.

[53] Adam Trachtenberg David Sklar. *PHP Cookbook.* O'Reilly, Sebastopol, California, 2002.

[54] Github laravel, Март 04, 2015. URL https://github.com/laravel/laravel.

[55] Laravel book, Март 04, 2015. URL http://laravelbook.com/.

[56] Laravel документација, Март 04, 2015. URL http://laravel.com/docs/5.0.

[57] Wiki:. Message broker, Април 02, 2015. URL https://en.wikipedia.org/wiki/Message_broker.

[58] Apache kafka. URL https://kafka.apache.org/.

[59] Apache kafka infoq, Март 04, 2015. URL http://www.infoq.com/articles/apache-kafka.

[60] Apache kafka download, Март 04, 2015. URL https://kafka.apache.org/downloads.html.

[61] Предраг Јаничић Филип Марић. *Основе програмирања кроз програмски језик C*. Март 04, 2015. URL http://poincare.matf.bg.ac.rs/~janicic//courses/p1.pdf.

[62] Dennis M. Ritchie Brian W. Kernighan. *The C Programming Language, 2nd Edition*. Prentice Hall, Upper Saddle River, New Jersey, 1988.

[63] Wiki: Lex, Март 04, 2015. URL https://en.wikipedia.org/wiki/Lex_(software).

[64] Wiki: Yacc. URL https://sr.wikipedia.org/wiki/Yacc.

[65] Wiki: Python, Март 04, 2015. URL https://en.wikipedia.org/wiki/Python_(programming_language).

[66] Python документација, Март 04, 2015. URL https://www.python.org/doc/.

[67] Mark Pilgrim. *Dive Into Python*. Apress Media LLC, New York, 2004.

[68] Magnus Lie Hetland. *Beginning Python: From Novice to Professional, 2nd Edition*. Apress Media LLC, New York, 2009.

[69] Charles Dierbach. *Introduction to Computer Science Using Python: A Computational Problem-Solving Focus*. Wiley, Hoboken, New Jersey, 2012.

[70] Jason Montojo Paul Gries, Jennifer Campbell. *Practical Programming: An Introduction to Computer Science Using Python 3*. The Pragmatic Bookshelf, Raleigh, North Carolina, 2013.

[71] Mark Lutz. *Programming Python*. O'Reilly, Sebastopol, California, 2006.

[72] Alex Martelli. *Python in a Nutshell*. O'Reilly, Sebastopol, California, 2003.

[73] Brian K. Jones David Beazley. *Python Cookbook, Third edition*. O'Reilly, Sebastopol, California, 2013.

[74] Wiki: Sympy, Март 04, 2015. URL https://en.wikipedia.org/wiki/SymPy.

[75] Wiki: Ruby, Март 04, 2015. URL https://en.wikipedia.org/wiki/Ruby_(programming_language).

[76] Yukihiro Matsumoto David Flanagan. *The Ruby Programming Language*. O'Reilly, Sebastopol, California, 2008.

[77] Yukihiro Matsumoto. *Ruby In A Nutshell*. O'Reilly, Sebastopol, California, 2001.

[78] Gregory T Brown. *Ruby Best Practices*. O'Reilly, Sebastopol, California, 2009.

[79] Wiki: Java, Март 04, 2015. URL https://en.wikipedia.org/wiki/Java_(programming_language).

[80] Joshua Bloch. *Effective Java (2nd Edition)*. Addison-Wesley, Boston, 2008.

[81] Java tuttorial. URL http://docs.oracle.com/javase/tutorial/.

[82] JAS. Java algebra system (jas) project, Март 04, 2015. URL http://krum.rz.uni-mannheim.de/jas/.

[83] Heinz Kredel. Evaluation of a java computer algebra system, Март 04, 2015. URL http://krum.rz.uni-mannheim.de/kredel/jas-ascm2007.pdf.

[84] Wiki: Nodejs, Март 04, 2015. URL https://en.wikipedia.org/wiki/Node.js.

[85] Azat Mardan. *Practical Node.js: Building Real-World Scalable Web Apps*. Apress Media LLC, New York, 2014.

[86] Mike Cantelonn. *Node.js in Action*. Manning, Greenwich, Connecticut, 2013.

[87] Redis документација, Март 04, 2015. URL http://redis.io).

[88] Josiah L. Carlson. *Redis in Action*. Manning, Greenwich, Connecticut, 2013.

[89] Tiago Macedo. *Redis Cookbook*. O'Reilly, Sebastopol, California, 2011.

[90] Redis replication, Март 05, 2015. URL http://redis.io/topics/replication.

[91] Kafka php client, Март 04, 2015. URL https://github.com/quipo/kafka-php.

[92] Kafka ruby client, Март 04, 2015. URL https://github.com/bpot/poseidon.

[93] Саша Малков. Информациони системи: Архитектуре оријентисане према сервисима – СОА, Април 01, 2015. URL http://poincare.matf.bg.ac.rs/~smalkov//files/is.r271.2014/public/predavanja/IS.cas.2014.10.SOA.pdf.