

**Технически университет – София**

**Факултет по приложна математика и информатика**

**Катедра информатика**

**Архитектурен проект на тема:**

Дистрибутиран Load-Balancer (Active-Active режим на възлите)

Стоян Стоянов фак.№ 471219076

София, 2022

Contents

[1.) Въведение 3](#_Toc124306217)

[2.) Предназначение на избраната тема 3](#_Toc124306218)

[2.1) Обхват на проекта 4](#_Toc124306219)

[2.2) Избрани актьори 5](#_Toc124306220)

[3.) Основни цели на архитектурата 5](#_Toc124306221)

[4.) Архитектурен обзор 8](#_Toc124306222)

[4.1) Описание на основните сценарии (use-cases) и значението на актьорите 9](#_Toc124306223)

[4.2) Описание на логическия изглед чрез класови (class) диаграми 12](#_Toc124306224)

[4.3) Изглед на процесите 13](#_Toc124306225)

[4.4) Изглед на внедряването 16](#_Toc124306226)

[5.) Обосновка защо архитектурата осигурява адекватно решение на проблема 17](#_Toc124306227)

[6.) Източници 19](#_Toc124306228)

# Въведение

В модерните приложения все повече и повече се налага използването на методологията за дистрибитурани приложения или услуги. По този начин може лесно да се удовлетворят нуждите на доставчиците на дадена услуги относно скалирането и достъпа им до тях. Когато дадена инстанция на приложение изнемогва на ресурси, доставчика може лесно да увеличи инстанциите му, като така може да разпредели натоварването, също така може лесно да се доставят и тестват новите свойства към приложението. Докато това ново влияние се разпространява, заедно с новите технологии съпътсващи го като Kubernetes, Docker и друг SaaS, IaaS доставчици, то остава проблема с хората имащи и желаещи да могат лесно да си направят достъпността до своите дистрибутирани услуги, като има и възможността да се възползват от услуги използвани по подобен начин в средите представени по-горе.

Целта на този проект е именно такава, да предостави услуга, която може да бъде използвана от клиент, за да направи достъп до услугите си от централно-децентрализирано място, като при по-голямо натоварване или неочаквано събитие с някоя от машините, да може все пак да запази достъпа до своето приложение.

# Предназначение на избраната тема

Конкретния архитектурен документ описва целите, възможностите и структурата за изграждане на дистрибутиран `***Load Balancer***` в Active-Active режим. Чрез представяне на основите на архитектурното решение, както и шаблоните, приложени към него, ще се улесни разработването, както и ще може да съберем различни мнения за подобряване на архитектурата, преди да е почнало самото разработване на софтуерният продукт. Чрез предоставяне на основната идея в по-абстрактен вид, ще се създаде дизайн, който максимално да съответства на изискванията към продукта. Колкото по-добър дизайн успеем да постигнем, толкова по-лесно ще ориентираме разработчиците към проекта, като разбира се това включва и избиране на подходящ архитектурен шаблон.

Чрез изпълнението на архитектурата, впоследствие и изпълнението на крайният продукт, ще се даде територия за комуникация със заинтересованите страни, които лесно може да представят нуждите си и на база тях може да определим, какво може да се подобри по дизайна, какво ново може да се заложи и обхвата на проекта ни.

## 2.1) Обхват на проекта

Проектът е насочен към всички потребители имащи трудни за менажиране на достъпа услуги и искащ да има постоянен достъп до свойте услуги, при възможен проблем на хост-а на някоя от инстанциите ни на Load Balancer-а. Фазите който ще обхваща следният документ са:

* Иницииране
* Планиране
* Изпълнение
* Контролиране

В конкретния момента на писането на документа сме преминали следните фази: Иницииране – поставени са основните цели на проекта, Планиране – планиран е начин по който да се процедира, какъв проблем решаваме и как точно да го решим, Изпълнение – имплементирали сме най-важната част нужна за оперирането на приложението, като така лесно може да се даде приложението към тестови потребител с цел предоставяне на обратна връзка.

## 2.2) Избрани актьори

Основните актьори налични при използването на системата, ще се представят в по-големи детайли, чрез use-case диаграмите и тяхното описание по-надолу в архитектурния модел. За сега ще са ни нужни само техните наименования:

* Потребител на приложенията на доставичика на услуги (SCustomer)
* Доставчик на услуги или администратор (SDeliver)

# Основни цели на архитектурата

Услугата дистрибутиран Load-Balancer представлява услуга дефинираща се в разпределено разпространение на Load-Balancer-и, с цел постигането на голямо покритие (High-Availability), с цел навалянето на времето на недостъпността на услугите на потребителя ни, при възможни проблеми с инстанцията на някой единичен Load Balancer. Той предоставя следните функционалности:

* Задаване на основна конфигурация
* Добавяне на нови услуги, който да се използват за препращане на заявките към
* Достъп до съществуващи услуги
* Модифицирането на съществуващи услуги
* Изтриването на съществуващи услуги
* Механизъм за синхронизиране на отделните инстанции с главния сървър
* Инстанция използвана за балансиране на заявките

За постигането на целите на проекта ни ще реализираме client-server шаблона, като в случая клиента ще ни е самата инстанция, която прави Load Balancing-а между предоставените услуги и сървър частта, за да можем лесно да централизираме конфигурацията на едно място, където да е достъпна само от клиент ни и инстанциите. За да обосновем нуждата от използването на този шаблон, нека първо засегнем имплементациите на такива услуги за балансиране на постъпващ трафик. Повечето нови услуги имат конфигурации, като за конфигурирането на такъв тип балансър с високо покритие, тези конфигурации трябва да са синхронизирани. Това синхронизиране се постига чрез централизиране на файловете от конфигурацията на едно място като Git хранилище, или някакво централно файлово хранилище като FTP сървър и т.н., като за прилагането им се използват продукти като: ArgoCD, Ansible, Terraform и други, като чрез тях лесно може да се конфигурират много машини чрез конфигурация предоставена на едно място. Но с тези технологии идват и следните проблеми: какво правим ако машината може да достъпи клиентските услуги без прокси връзка, а друга може (тогава не може лесно да се адресира с тези централизирани конфигурационни файлове, защото идеята им е да не специфични за хоста, а специфични за целия набор от хостове), какво става ако само една инстанция не е успяла да се синхронизира с конфигурациите (това би донесло допълнителна недостъпност до нашите услуги при промяна на синхронизиране), какво става при загуба на данни в даден конфигурация и как ще гарантираме че същата конфигурация ще бъде изпълнена (продуктите като Ansible, Terraform и т.н., предоставят хубава среда за менажиране на ресурсите, но не предоставят начин на следене за промени между различните инстанции на тези файлове, те само предават, а какво се случва на машината в последствие след изпълнение им не е тяхна работа), като за последно нека не забравяме и лесното добавяне на нови инстанции които да балансираме, без спирането на достъпа до услугите ни.

Именно така се роди и идеята за client-server шаблона, като така много лесно може да адресираме следните проблеми. Сървър частта държи основната конфигурация, като тя се задава при стартирането и живее в паметта му, така лесно може да гарантираме, че ако искаме промени лесно може да рестартира сървъра с нова конфигурация, без да се налага писане по бази данни и т.н., също така сървър частта поддържа REST endpoint-и за взимане, добавяне, модифицирането и изтриването на конфигурации, като при изпълнението им се използват `event`-и, за синхронизирането на отделните клиентски инстанции, изпълнявайки важния за микро-услугите шаблон CQRS (Command and Query Responsibility Segregation), като така тотално махаме зависимостта отделните инстанции да имат достъп до базата и да забавят клиентските заявки към `backend`-a. Идеята зад използването на `event`-и е лесното синхронизиране на инстанциите, като спрямо типа му лесно може да се избере стратегията която инстанцията да изпълни за регистрирането на правилната конфигурация в инстанцията. Откъм клиентската част лесно може да се имплементира този механизъм за обработка на събитията от backend частта, като няма да се налага четене от базата с данни, също така този клиент ще инициализира кеш памет, за да пази потребителските сесии със съответните хостове, при режим на `sticky-connections`.

Имплементирането на client-server частта ни носи едно от основните неща нужни ни в конкретната имплементация, а именно централизиране на конфигурациите и лесното им синхронизиране с отделните клиенти. Също така CQRS разделя операциите за четене и актуализиране за хранилище на данни. Внедряването на CQRS в нашето приложение може да увеличи максимално неговата производителност, мащабируемост и сигурност. Гъвкавостта, създадена чрез използването на CQRS пред директното използване отделните компоненти на приложението ни, позволява на системата да се развива по-добре с времето и предотвратява командите за актуализиране да причиняват конфликти при сливане на ниво домейн. В конкретната имплементация на CQRS шаблона ние няма да се сблъскаме с един от проблемите му, а именно поддържане на различни технологии за бази данни. Като това бива така, защото: клиентите (отделните инстанции на Load Balancer-а) държат както основната така и конфигурацията за наличните хостове в своята памет, докато единствения достъп до единствената база се осъществява само от сървъра, от който чрез `REST endpoint`-и и `event`-и се осъществява достъп до каквито и да е данни.

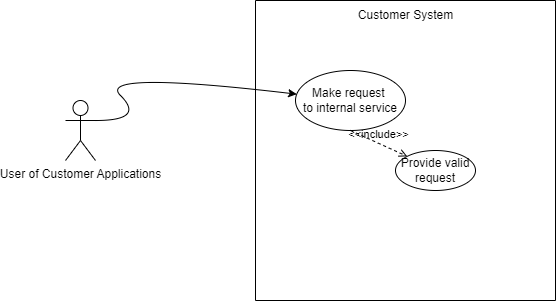
Следвайки тези стъпки и архитектурни решения, ние гарантираме също и няколко не по-малко важни нефункционални изисквания, които играят важна роля в бъдещото развитие на продукта. Именно производителността ще бъде оптимално разпределена, като и заради възможното проследяване на броя на чакащите заявки на опашките ще можем да разпределяме дали нова машина, или повече ресурси, за забързването на средното време, нужно за обработка. Както по-горе споменахме друго, което ще се постигне е скалируемостта, когато говорехме за повече товар при едни услуги и по-малко при други. Също споменахме централизирането на данните, което доставя защита на данните, от външните мрежи.

# Архитектурен обзор

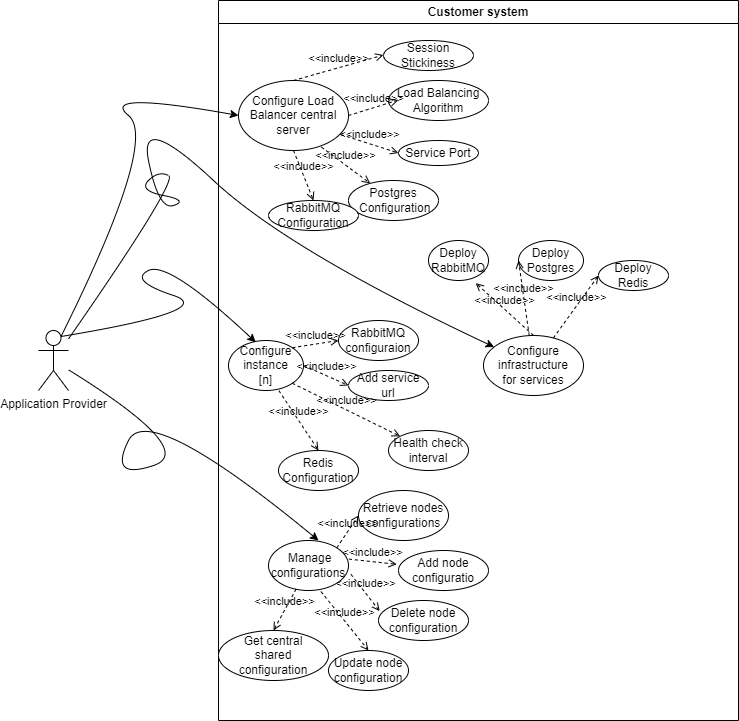
Водейки се от избраният архитектурен шаблон (Client-Server with CQRS) за бакенд частта, в дадената точка ще се опишат някой от основните изгледи, нужни на разработчиците, за изграждането на ясна представа за начина на функциониране на системата, както и участниците, играещи роля при изпълнението на операции с нея.

## 4.1) Описание на основните сценарии (use-cases) и значението на актьорите

За представянето на системата ни ще използваме два типа актьори. Първият тип е потребител на доставчика на услуги, а вторият е администратора на доставчика на услуги. Докато задачата на първия тип актьор е тривиална за нашата архитектура, понеже самия потребител е агностик относно това как заявките му да бъдат разпределени, а какво ще подаде на системата и нейния резултат. От друга страна втория актьор а именно администратора на доставчика на услуги е по-важен за нашето приложение.

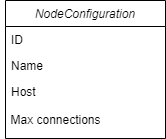
 фиг. 4.1

Като за начало, нека опишем първия ни актьор (за повече информация погледнете фиг. 4.1), който е крайния потребител на нашето балансиране. За него е важно заявката му да стига максимално бързо до определена инстанция на услугата, също така важно е той да бъде балансиран по правилния начин от системата за балансира, което се изразява в пазене на сесия (ако е специфицирано от администратора), ако трябва една и съща инстанция да балансира заявките на определен потребител. За него също така е важно да подаде правилна заявка, ако при първата му заявка услугата ни му подаде бисквитка с токен, който да се използва за следващи заявки, то ако той не я подаде тогава вината ще е негова при неналична идемпотентност между две еднакви заявки (при услуги изискващи да бъдат изпълнявани на една инстанция).

 фиг. 4.2

След като описахме основния потребител нека опишем и системния администратор, който ще се грижи за конфигуриране на системата и добавяне на нови услуги към услугата за балансиране. Една от неговите основни задача е конфигурирането на инстанцията, която ще подсигурява синхронизиране между балансиращите такива. За целта той трябва да предостави първоначална конфигурация, която включва какъв алгоритъм да се ползва за балансирането между инстанциите и дали да се използват `sticky-connections`[[1]](#footnote-1). Също така важна част от конфигурацията е предоставяне на порт, който да се отвори услугата на. След като е предоставил основната конфигурация, то остава да предостави и конфигурацията за запазване на данните, както и такава за комуникация с останалите балансиращи инстанции. За целта той трябва да предостави първо конфигурацията за Postgres базата данни, като за целта към приложението трябва да подаде първо хост-а на който базата може да се открие, а след това да зададе преди стартирането следните променливи на средата: PG\_USER – потребителя с който да се оторизира пред базата, PG\_PASS – паролата към потребителя с който ще се оторизира, PG\_DBNAME – името на базата, която иска да ползва за приложението. След като сме задали конфигурацията за базата с данни остана да подаде конфигурацията за RabbitMQ, което ще се използва за синхронизация между отделните услуги, като за целта се подава следния низ: ”amqp://<user>:<password>@<rabbit\_hostname>:<rabbit\_host\_port>“, където amqp е протоколът който RabbitMQ имплементира за пренасянето на съобщенипта, user & password – потребителя и паролата с които да се оторизираме пред сървъра предоствящ message queue, rabbit\_hostname – хоста на който се намира queue-то, rabbit\_host\_port – порт-а на който е отоворено queue-то. Преди да сетъпне дали сървър или инстанция, администратора, трябва да се е погрижил да е пусна средата, като това може да се направи с гореспоменатите инструменти за менажиране на хостове, като пускането на средата се изразява именно с пускането на инстанциите със следните услуги: RabbitMQ, Postgres и Redis, като администратора, също така трябва да осигури, че има създадена база, поради факта, че не се създава от услугата, която служи за синхронизация на конфигурациите. След като средата е сетъп-ната, услугата за синхронизация пусната, то той може да продължи с пускането на отделна инстанция за балансиране, като нужните конфигурации който трябва да зададе там са препокриващи се с част от предишните, като например RabbitMQ и Redis конфигурацията, като зададе допълни настройки като: Интервал на който да се прави health-check на регистрираните node-ове за балансиране, както и URL към главната услуга, която ползваме за синхронизация. Следващото важно нещо което администратора може да прави е добавянето на конфигурации с услуги, който да се балансират. Това става посредством REST заявки, като възможностите са: Да вземе всички конфигурации, Да добави конфигурация, Да изтрие конфигурация, Да обнови конфигурация, Да вземе глобалната конфигурация, която отделните услуги споделят.

## 4.2) Описание на логическия изглед чрез класови (class) диаграми

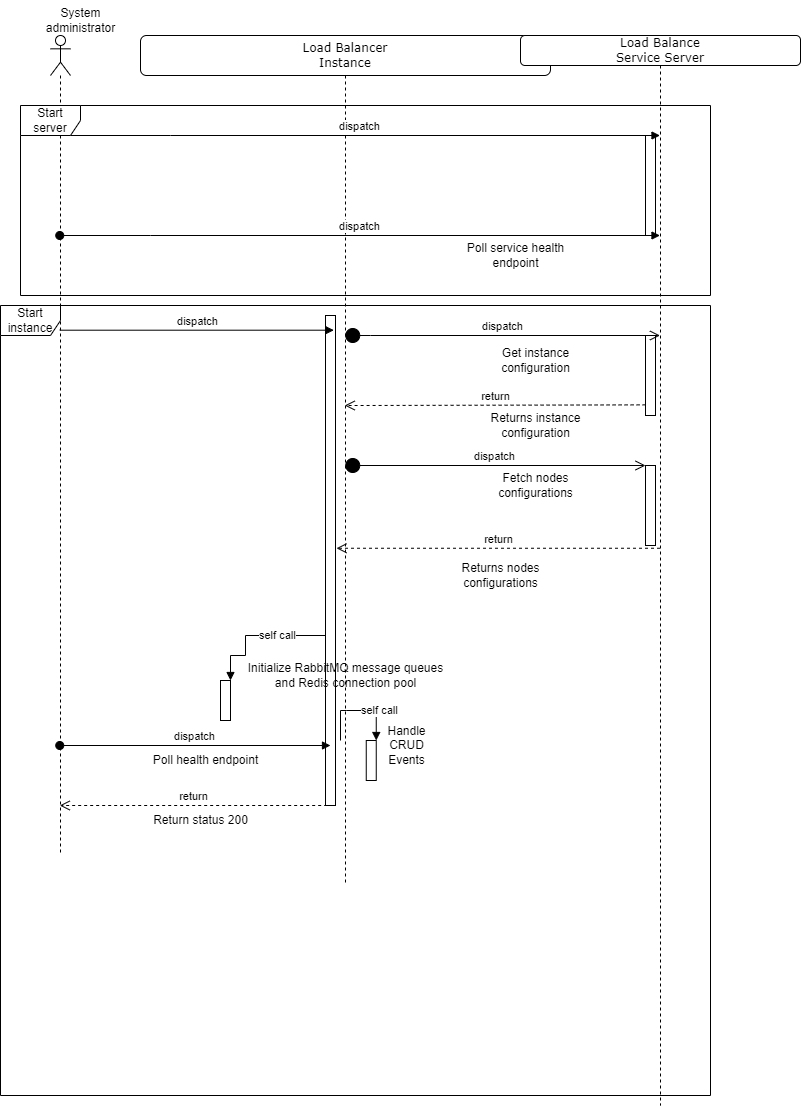
 фиг 4.2.1

На фигура 4.2.1, може да видим единствения клас, чрез който ще оперираме, като в него се специфира ID, което ни индекса чрез който може да търсим и да правим допълнителни операции с даден хост в базата. Второто ни поле е името на хост-а, то се генерира чрез UUID генератор, също така в базата се мигрира като uniqueIndex[[2]](#footnote-2). Следващото ни поле е Host, което представлява хост-а на който трябва да се пращат заявките, той също се идентифицира с uniqueIndex ограничението, като важно е да се специфицира, че схемата трябва да присъства в хост-а, като за пример за валиден хост би бил: <http://google.com>, където http ни е схемата, а невалиден би бил: `google.com`. Като последно пропърти имаме Max Connections, което специфицира, колко максимални връзки, може да имаме към даден хост в даден момент, като това ще го използваме, като теглови параметър за балансирането в случая на Round-Robin алгоритъма.

Важно да се отбележи че при създаването или модифицирането на конфигурация, полетата като ID и Host, ще бъдат игнорирани, т.е. няма да могат да се променят, защото са генерирани от системата, като ни дава достъп да модифицираме Host-a и максималния брой връзки който позволява, като се има предвид, че ако модифицираме хост, към хост съществуващ вече в базата, то базата ще ни върне грешка че не може да направи промяната.

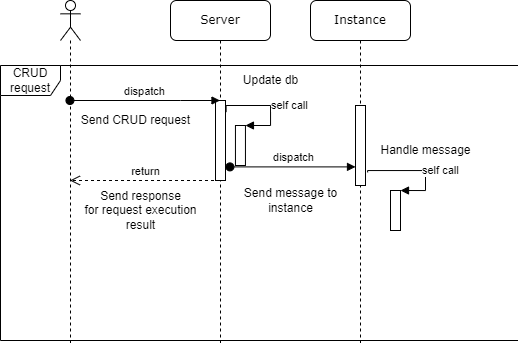
## 4.3) Изглед на процесите

Процесите в апликацията ни основно се базират около администратора ни, поради природата му или по-точно, при всяко негово действие ние караме системата да проверява статуса си и при наличност на нови данни, да го обновява спрямо тях. Нека затова представим няколко ситуации, като ги групираме, спрямо момент на изпълнение.



Фиг. 4.3.1

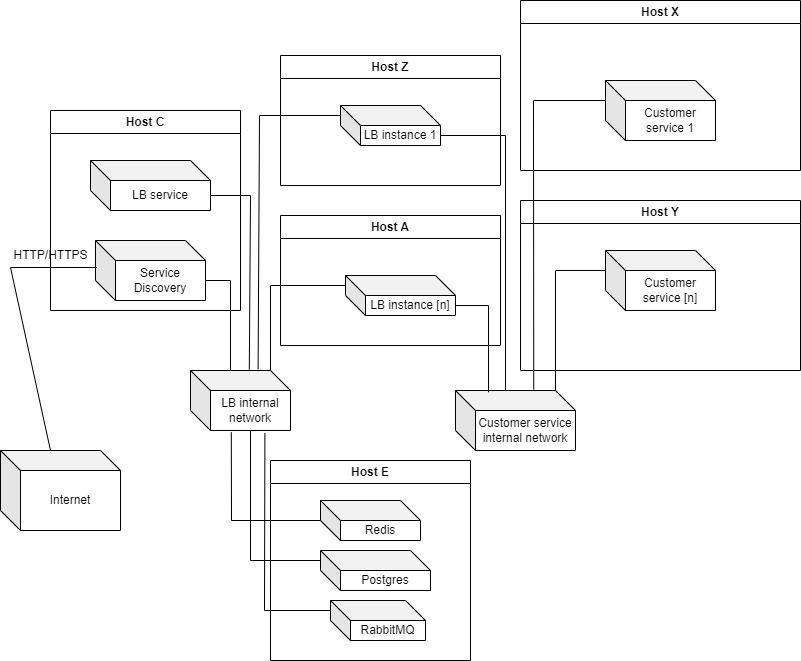
На фиг. 4.3.1 са групирани ситуациите, където администратора, стартира, както сървърната инстанция, така и инстанция за балансиране на заявките. Нека за първи случай разгледаме стартирането на сървъра. След като задава правилната конфигурация, като алгоритъм с който да балансира, параметри за базата с данни и т.н., администратора почва да следи дали сървъра се е пуснал, като за това нещо има endpoint от REST API-то ни, който ни връща статус код, когато се е стартирал, именно чрез него администратора може постоянно да праща заявки и да следи статуса на сървъра. След като е успешно пусна сървъра, то той може да продължи със стартирането на инстанцията, като я стартира с нужните параметри за конфигурация, след което подобно на сървъра, следи за статуса от `health endpoint`-a, за да може да види, дали инстанцията е готова да приема нови заявки. Самата инстанция се опитва да вземе основната конфигурация от главния ни сървър, след което се опитва да вземе възлите върху който ще балансира заявките (паралелно), като на този етап е възможно да има такива, но е възможно и противното. След като се е опитало да ги вземе, стартира мониториране на хостовете (паралелно), както и инициализира `Redis Connection Pool`-а, RabbitMQ връзката, като започва да следи дали не са се появили нови съобщение с CRUD операция върху конфигурация за даден възел (паралелно). След което продължава своето изпълнение, което ще опишем по-долу.



Фиг. 4.3.2

На фигура 4.3.2, може да се наблюдава следващия важен за приложението процес, а именно когато администратора прави CRUD заявка към сървъра ни като например: промени, изтри, добави конфигурация. Така след като направи заявка към бекенд-а ни, той прави нужните промени по базата с данни, после праща съобщение към всички инстанции, с променената конфигурация на възела, като после връща статуса от заявката на администратора. Паралелно на този процес се осъществява обработването на съобщението, за да може да се обработи сървъра добавя хендър към съобщението, който казва каква операция се е извършила, тогава щом инстанцията прочете съобщението, ще може да избере стратегия която да изпълни за конкретния евент, обновявайки данните с които работи, правейки проверки за хостовете, които са променени дали са активни и т.н..

## 4.4) Изглед на внедряването

 фиг. 4.4.1

За да покажем системата, как ще изглежда в продуктивна среда, нека да покажем следната диаграма на внедряване. В нея заявката ще постъпва през интернет от даден потребител, през едно от определените места, което се нарича Service Discovery, което само по себе си ще избира кой load balancer ще използваме и ни дава достъп до мрежата, чрез която може да установим връзка с конкретния балансър. В конкретния случай дефинираме две мрежи: LB internal network – която се ползва за комуникация между услугите нужни ни за load balancer-a и load balancer услугата, като комуникационни канали, база данни, кеш и т.н. и Customer service internal network – нужна за комуникацията между load balancer-ите и услугите които предоставя клиента ни. Така лесно постигаме голяма степен на изолация на мрежите, като същевременно намаляваме достъпа за външна атака от злонамерено лице.

# Обосновка защо архитектурата осигурява адекватно решение на проблема

С конкретната архитектурна имплементация се осигурява по-лесно конфигуриране на мрежата от услуги на даден клиент, като се има в предвид, че тези услуги не работят на IaaS услуга, като Kubernetes, Docker и т.н., ами са смесица, което ще рече, че част от тях може да работят на гореспоменатите среди, но също така може и да работа и на Bare Metal[[3]](#footnote-3), като тази нехомогенност от среди за изпълнение на услугите, прави задачата за интегриране на вече съществуващ Load Balancer, значително по-сложна.

Друга важна цел на конкретната архитектура бе да изпълни възможността за висока наличност (High Availability) на нашите услуги, или по-просто казано да имаме много инстанции, който да може да балансират постъпилите към тях заявки, като така лесно се осигурява балансъри от тип Active-Active, с централизирана конфигурация. Също така предоставянето на начин за комуникация при CRUD[[4]](#footnote-4) операциите върху главната инстанцния, посредством CQRS шаблона, споменат по-горе, чрез message queue-та осигурява евентуална консистенция на данните в всяка инстанция по всяко време. Друг важен аспект е запазването на сесиите (където е нужно), именно чрез Redis кеш, осигурява споделен статус между отделните инстанции с цел ако заявката попадне в различна от първоначалния балансър, то другия да я препрати към правилния хост, запазвайки консистентността.

# Източници

* <https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/>
* https://www.nginx.com/resources/glossary/load-balancing/
* <https://fideloper.com/golang-single-host-reverse-proxy>
* <https://github.com/sohamkamani/go-session-auth-example/blob/master/handlers.go>
* <https://kasvith.me/posts/lets-create-a-simple-lb-go/>
* <https://github.com/sohamkamani/go-session-auth-example>
* <https://www.sohamkamani.com/golang/session-cookie-authentication/>
* <https://github.com/gomodule/redigo>
* <https://gorm.io/>
* <https://www.rabbitmq.com/documentation.html>
* https://pkg.go.dev/net/http/httputil

1. Sticky-connection – запазва сесията/ip адреса на даден потребител, за да може при последваща негова заявка да го преведе към същия хост, който за първоначалната е бил избран. Ползва се за идемпотентност между заявките, при системи, които пазят статуса си локално. [↑](#footnote-ref-1)
2. uniqueIndex – това свойство, което се задава на поле в базата гарантира, че ако се опитаме да запишем елемент който е с същия индекс, то базата няма да направи такава операция и ще даде грешка. Също така се използва за ускорение при изпълнението на заявки, ако търсим по него. [↑](#footnote-ref-2)
3. Bare Metal – стандартна машина или виртуална машина, която изпълнява операционна система, върху която си изпълняваме приложението. В такава среда, ние не можем да гарантираме изолация на процесите, също така и мрежова изолация между отделните ни услуги, като така при повече услуги, има нужда от повече конфигурация и внимание с цел да не си пречат помежду си. [↑](#footnote-ref-3)
4. CRUD – Create, Remove, Update, Delete, са едни от основните операции, при опериране с база данни или ресурси. [↑](#footnote-ref-4)