

**Технически университет – София**

**Факултет по приложна математика и информатика**

**Катедра информатика**

**Архитектурен проект на тема:**

Система за споделяне на кулинарни рецепти

ТУ Кулинариум

Проектна група: 15

Антонио Димитров фак.№ 471221119

Кристина Симова фак.№ 471218081

Стелиан Тодоров фак.№ 471219057

Стоян Стоянов фак.№ 471219076

Теодора Маноилов фак.№ 473219001

София, 2021

**Съдържание:**

1. Въведение
2. Предназначение на избраната тема

2.1. Обхват на проекта

2.2. Избрани актьори

1. Основни цели на архитектурата - описание на няколко ключови изисквания и ограничения на разработвания софтуерен продукт, които оказват влияние на избраната архитектура
2. Архитектурен обзор

4.1. Описание на основни сценарии (use-case) и актьори (actors), които са важни за разработваната архитектура, чрез use-case диаграми

4.2. Описание на логическия изглед на архитектурата. Клас диаграма за илюстрация връзките между архитектурно значими класове, подсистеми, пакети и слоеве. Диаграма на състоянието за илюстрация конкретни процеси на определени роли.

4.3. Изглед на процесите. Описание на отделните изпълними процеси (подсистеми) и зависимостите между тях.

4.4. Изглед на внедряването (Deployment view)

4.5. Изглед за разработка (Implementation view)

1. Обосновка на това как избраната архитектура осигурява адекватна реализация на поставените нефункционални изисквания
2. Използвани термини и символи
3. Използвани източници

# **Въведениe**

Когато живеем в заобикалящия ни свят всеки ден ние се сблъскваме с хора. Постоянният обмен на информация в ежедневието поражда както своите предимства, така и недостатъци, като повечето пъти не осъзнаваме, колко изморително е цялото това обработване на информация от нашия организъм. Колкото повече помним информация, толкова повече създаваме предпоставката да забравим нещо. Дали то ще е изпусната възможност, когато забравим за среща, която може да ни развие чрез нейната тема или нещо по-важно, като забравена включена ютия, дете в градината или задача от работа, не е важното толкова какво е забравено, колкото причината, поради която се е случило. Нещото, което генерира много информация в главите ни, то това е готвенето. Дали защото си мислим какво ще трябва да готвим, или пък какво да напазаруваме, или пък когато сме гладни и сме все още на работа все си мислим за това какво ще ядем.

След като прекараме различните сценарии, свързани с ястия, опираме пак до термина информация. За това тази архитектура насочва фокуса си върху създаването на система, чийто основен фокус са рецептите – тяхното създаване, менажиране, съхранение и т.н.. Чрез поставянето на съществените функционални и нефункционални изисквания и тяхното стриктно изпълнение, ще се реши проблемът със създаването на система за споделяне на кулинарни рецепти.

# **Предназначение на избраната тема**

В настоящият документ се представя основна информация, нужна за разработката на система за споделяне на кулинарни рецепти. 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 на хора с различни кулинарни интереси. Също така ще се създаде среда за развитие на бизнесите и кулинарната култура на различни хора и ще се реши проблемът с липсата на точна и ясна информация за приготвянето на дадено ястие.

## **Обхват на проекта**

Проектът е насочен към всички крайни потребители без ограничение относно възраст, етнос, пол, финансов статут и други, като така гарантираме обществената му достъпност. По този начин всеки потребител ще може да разгледа каталог от множество разнообразни рецепти и да избере следващото ястие, което да приготви, по един много лесен и достъпен начин. Също така, чрез добавяне на продукти към кошницата за пазаруване рискът от забравяне на продукт, нужен за приготвянето на дадено ястие, многократно се намалява. Фазите които проекта ще обхваща са следните

* Иницииране
* Планиране
* Изпълнение
* Контролиране

В момента на писането на конкретния документ, сме преминали фазата на инициализация и сме на фаза планиране. В момента, в който се финализира документът, ще започне фазата на изпълнение, а именно създаването на база с данни, създаването на сървъри и писането на код.

## **Избрани актьори**

Основните актьори при използването на системата, ще се представят в по-големи детайли, чрез use-case диаграмите по-надолу в архитектурния модел. За сега само ще приложим техните наименования:

* Администратор
* Потребител
* Посетител

## **Основни цели на архитектурата – описание на няколко ключови изисквания и ограничения на разработвания софтуерен продукт, които оказват влияние на избраната архитектура**

Уеб системата „ТУ Кулинариум“ предоставя платформа за споделяне на рецепти между различни потребители, както и посетители, предоставяйки следните функционалности:

* Преглеждане на рецептите от различните посетители
* Създаване на коментари под рецептите (операция за потребители)
* Създаване на рецепти (операция за потребители)
* Редактиране на рецепти (операция за потребители)
* Изтриване на рецепти (операция за потребители само върху техни рецепти, както и администратор върху всички)
* Одобряване на рецепти преди публикуването им (наличие на администратор)
* Преглеждане на профила на различните потребители, като така се дава достъп до рецептите, които са публикували

За постигането на целите на бакенд частта ние ще използваме комбинация от MVC и Microsevices шаблоните за разработка на софтуерни архитектурни решения. Да обосновем защо се налага използването на архитектура чрез микро-услуги за бакенда. Чрез използването на тази архитектура ще може да се реагира на променящият се трафик към системата. Също така важна е и възможността за скалируемост. В зависимост от необходимостта ще бъде приложена както хоризонтална - чрез добавяне на допълнителни компютърни единици и обединяването им чрез load-balancer, който да менажира натоварването между отделните сървърни инстанции, така и вертикална, когато бизнес логиката се усложни достатъчно откъм брой изчисления за единица време, или е необходимо пространство за съхранение на данни. Ще предоставим система, към която в бъдещето си развитие ще има голям наплив от потребители, ще има нужда от разпределител на натоварванията в случай на много заявки към основната система, което ще ни принуди да разширим системата си хоризонтално. В по-дългото си развитие, може да наложим автоматичен анализ на рецептите и автоматичното им одобрение, чрез използването на даден алгоритъм прилагащ изкуствен интелект, което ще увеличи времето за обработка на дадена заявка. Съхранението на отделните снимки към дадените рецепти също налага вертикалното разширяване на системат. Относно използването на MVC, ще предоставим по-лесното структуриране и впоследствие изпълнение на проекта. Използвайки тази архитектура ще се гарантират модели за нужните структури, които ще бъдат важни при взаимодействието на приложението със слоя за запаметяване на данните (базата с данни). Чрез контролерите ще може да изпълняваме бизнес логиката посредством взаимодействие с потребителите, като им връщаме подходящ отговор. 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. Друго важно нещо е ефективността, която чрез повечето MVC решения или по-често чувано framework-ове, зареждането на данните от няколко таблици няма да се осъществява, докато изрично не достъпим елемент от тях, както и кеширането на данни при повторно наложно използване в конкретен времеви диапазон. 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. От друга страна MVC, шаблона ни предоставя по-лесна възможност за разширяване на приложението ни, като промяна на модели, добавяне на нови, добавяне на ново поведение, без да влияе на другите (при контролерите). Единствен нейн недостатък е многото нужен код за постигане на малко, което не е толкова голям проблем, защото от друга страна е по-разбираем, което при по-дългосрочен план би помогнало при разбиране на дейността на отделните компоненти. Отделно комбинацията от двете архитектури, т.е. по-простото им разбиране, ще ни помогне при нови членове в даден екип разработващ определена услуга, да навлязат по-бързо в процеса. Другите плюсове от използването са следните: по-различни технологии, стига изходния формат да е еднакъв, по-лесно доставяне на следваща версия, дори да е поправка на бъг илитестване на услуга, защото една услуга отговаря за по-малко задачи.

За постигане на целите на мобилното приложение ще използваме MVVM и Coordinator шаблоните. Защо MVC шаблонът не е подходящ за целите на мобилното приложение? MVVM означава Model (прев. от английски модел), View (прев. от английски изглед) и ViewModel (прев. от английски изглед на модела). Ролята на модела е да структурира необходимите данни за изгледа, на изгледа – да визуализира тези данни, а на изгледа на модела – да управлява цялата бизнес логика и да обновява изгледа. Основната идея на този шаблон е да се изолира логиката за показването на изгледите от контролера в изгледа на модела. По този начин се скриват самата подготовка на данните, преди те да бъдат показани и асинхроните заявки. Това няма как да се постигне с MVC, тъй като там цялата тази логика се намира в контролерите. Цялостното разделение на ролите води до по-добра способност да се тества кодът, както и до по-добра визия за това как комуникират отделните модули помежду си. Координатор шаблонът ще се използва с цел оптимизиране на навигацията в приложението.

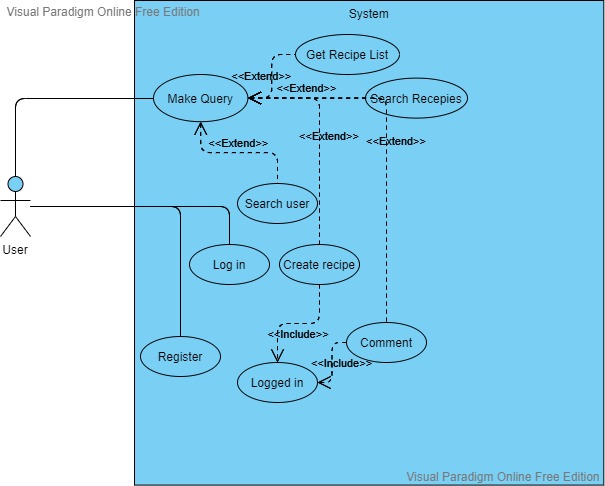
## **Архитектурен обзор**

Водейки се от избраният архитектурен шаблон (MVC) 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 основни сценарии (use-case) и значението на актьорите**

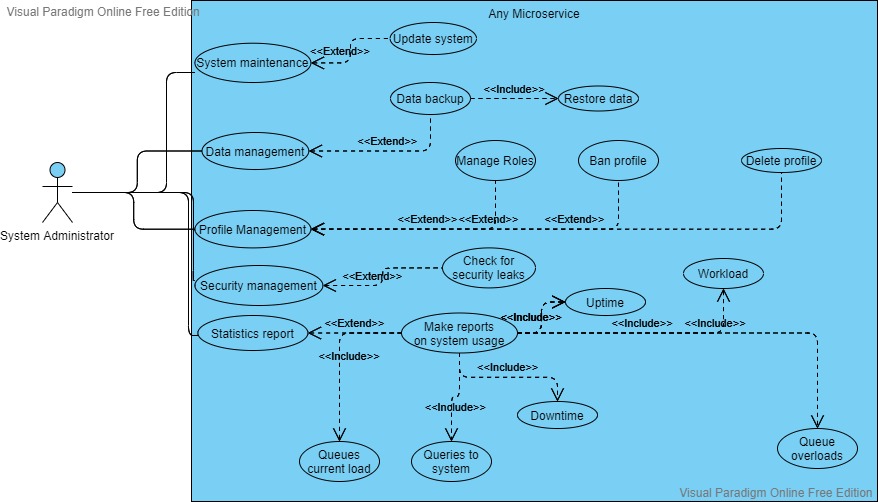
За представянето на системата или по-точно описването на действията, които се извършват в нея, ще използваме сценарии, в който ще имаме 3 различни актьора, като за всеки един от тях ще има набор от действия, които може да осъществява. Първи тип актьор е посетителя, в зависимост от това дали се вписал в системата, той има вариращи привилегии. Друг обособен в отделен клас от системата актьор е системният администратор, чиято задача е да се грижи за системата, като я обслужва. Целта на архитектурата е абстрактен поглед над градивните неща на дадена система, затова няма да изпадаме в подробности, за конкретна услуга дали има специален системен администратор, или относно потрбителя, защото с развиването на архитектурата, както и по-нататъшен дизайн, тези неща ще се определят.

Като за начало ще опишем потребителя и неговите основни функции, като за тази цел ще използваме следната use-case диаграма показана във фиг. 1.1. В нея представяме потребителя на нашата система, като той може да изпълнява следните задачи - да се впише, да се регистрира, да направи заявка, като заявката е обединяваща единица за всички възможни действия които може да се изпълнят, както и нуждите за изпълнението и. За да се регистрира потребителя ще трябва да предостави данни, които са уникални, като потребителско име, ако не представи такива, той няма да може да завърши регистрацията и ще му бъде предложено да повтори процеса по регистриране. Ако не е регистриран потребителят, той няма да може да използва опцията да се впише във системата, а след като не се е вписал неговите права ще бъдат ограничени до това да може да преглежда рецепти, както и да търси потребители. За да успее да влезне в профила си потребителят трябва да предостави правилните данни съответстващи за него, а именно потребителско име и парола съответстваща на потребителското име. Ако процесът не е успешен, системата ще се обърне към потребителя, казвайки му че в представените от него данни има грешка. Ако пък този процес е успешен, потребителят бива вписан и взима токен-а, чрез който ще се идентифицира като такъв и по този начин ще му бъдат представени конкретните права - за потребител или администратор. След като се е вписал той ще може пак да изпълнява заявките, присъщи за посетител, като надгражда техните и вече може да създава коментари под рецептите, както и да създава рецепти, чрез попълването на форма за рецепта.



Фиг. 1.1. Use-case диаграма на обикновен потребител

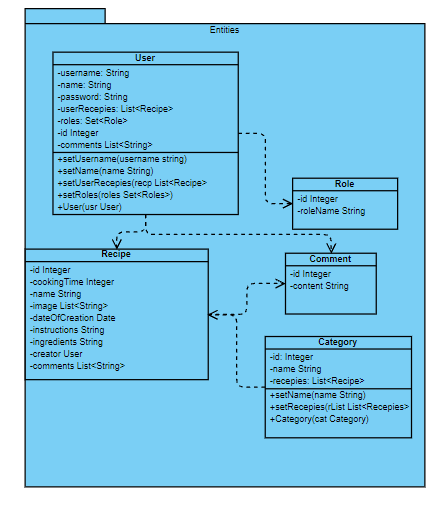
Остана да представим другия сценарии, включващ взаимодействието между системния администратор и системата. Той е актьорът, който ще поддържа системата ни и заради това заслужава подробното описание на дейностите му. За визуалното представяне на възможните сценарии, ще се използва use-case диаграмата от фиг. 1.2. Основните задачи на системния администратор са разделени в 5 по-големи класа от задачи. Те са следните: поддръжка на системата, управление на данните, управление на профилите, управление на сигурността и извличане на статистически данни, генерирани от работата на системата. При поддръжката на системата той ще може да обновява версията на софтуера. Друга важна негова задача е да прави копие на данните от базите с данни на услугите, за които е отговорен, както и при срив в системите, при който данните се изгубват или се наруши тяхната цялост, той ще трябва да ги възстанови, до възможно най-скорошно състояние. Той също ще трябва да може да администрира акаунтите, да може да създава, да променя състоянието, както и да премахва даден потребител, като също да одобрява рецептите, които са добавени. Важна негова задача е следенето на системите, чрез извличането на статистическите данни от работата на услугите той ще може да прецени броя на заявките към дадената услуга и подробни данни, нужни за правилната преценка за мащабиране на специфична част от системата. За пример ще дадем две възможни ситуации: когато има много заявки, в опашките средното време до началото на обслужване на дадена заявка ще е голяма, възможен позитив от статистическите данни е проследяването им, като така администраторът може да заяви системата да се мащабира хоризонтално. Втори случай е когато средното време за обработване е прекалено голямо, тогава може да се разпореди вертикално разширяване на системата. И последно, но най-важно откъм ежедневната работа на системата, е сигурността. Той ще може да следи дали някой по някакъв начин не се опитва да достъпи системата и да я използва не по предназначение, това може да се следи от журнала за който споменахме отгоре, като се записва информация за IP и т.н. на този който се опитва да достъпи системата.



Фиг. 1.2. Use-case диаграма на системен администратор

## **4.2 Описание на логически изглед чрез class диаграми**

Описанието чрез use-case диаграмите ще стане в няколко стъпки. Понеже използваме MVC архитектурата за бакенд частта, ще разделим проекта на 3 части: Controllers – пакет съдържащ различните контролери, Repositories – пакет, който ще съдържа функциите със заявките към базата с данни, и Entities – представящ полетата от базата с данни под формата на данни, които може да ползваме в бизнес логиката. В момента може да се видят основните заченки на MVC архитектурата, а именно имаме модели - Entity-тата, имаме контролери – Controllers, които освен че са отговорни за взаимодействието на бизнес логиката и потребителя, също така генерират изгледи – View-та, обработените данни спрямо бизнес модела.

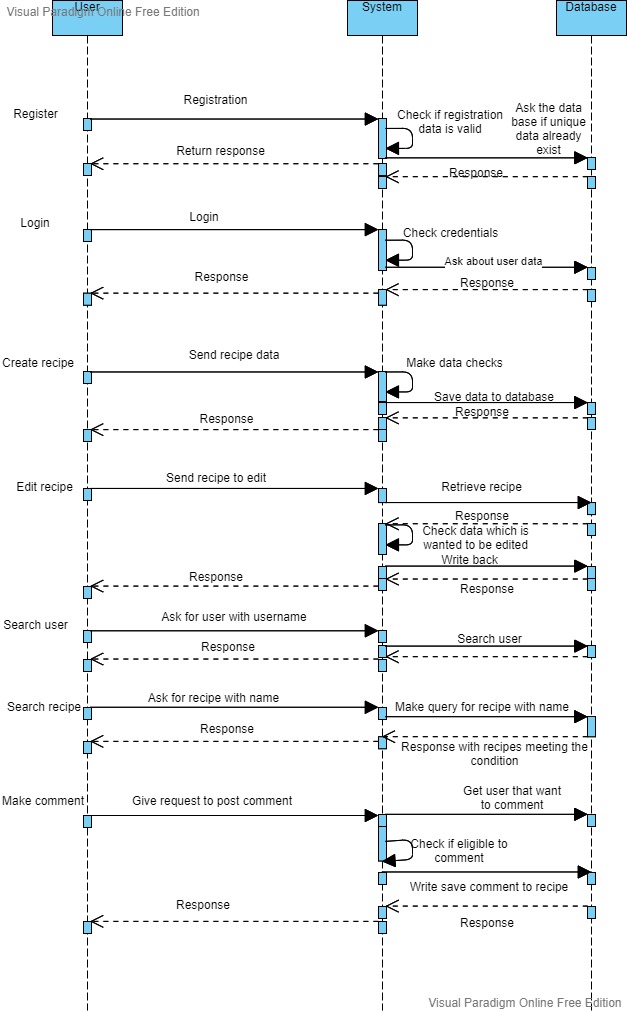
Фиг. 2.1. Модел на базата данни

На фиг. 2.1. е описан основният пакет – този на моделите или в случая кръстен Entity. Чрез него ще можем да адресираме полетата от базата с данни към тези от програмата, изпълняваща бизнес логиката. В нея имаме потребител, чиито полета са потребителско име и идентификационен номер, които трябва да се уникални, парола, която ще представлява кодиран низ от символи, както и лист от рецепти, представящ рецептите на даден потребител, сет от ролите на дадения потребител, както и коментарите публикувани от даден потребител от тип лист от коментари, с което изчерпва потребителският модел. Следва модела, които ще се асоциира с ролите, той ще съдържа данните за дадена роля, като основна информация ще се ползва цяло число за идентификационен номер, както и низ от символи за име на ролята. Друг неизбежен модел са рецептите, които ще имат уникален номер от тип цяло число, време за приготвяне от тип цяло число, име на рецептата от тип низ от символи, изображения от тип лист от низ от символи, тези низове ще реферират с линк към изображение съществуващо на сървър, който ще играе роля на CDN – content delivery network (система за доставка на съдържание), като при даден линк ще ни връща изображение. Също така рецептата ще съдържа дата на създаване от тип дата, както и инструкции и подправки, които ще се представят като низ от символи, създател на дадената рецепта от тип потребител, както и коментари които са лист от модела коментари. След като обобщихме съществуването на рецептите е хубаво да поясним въпроса за техните категории. Моделът, обвързан с категорията, ще съдържа идентификационен номер от тип цяло число, име на категорията от тип низ от символи, както и лист от рецепти. Последният важен модел за предоставянето на функционалните изисквания е този обвързан с коментарите, той ще съдържа идентификационен номер съдържащ цяло число и съдържание съдържащ низ от символи.

Върху тези основни класове ще се гради цялата бизнес логика. Ще се създадат контролери, отговорни да манипулират моделите, спрямо бизнес логиката на приложението. Такива контролери ще са контролер свързан с потребителския модел, като този тип контролери ще предоставят опцията на потребителя да взаимодейства с модела на потребителя, като за пример такива операции биват създаването на потребител, търсене на потребител, премахване на потребител (спрямо това дали има права, подателя на заявката), премахване на рецепта, редактиране на потребител и т.н.. Чрез контролера за рецептите, ще управляваме модела на рецептите, ще достъпваме рецепта спрямо нейния идентификационен номер, премахване на рецепта, създаване на рецепта и т.н.. Нуждата от контролер за ролите не е от такава наложителност на началния етап на разработка, тъй като засега е планирано да има само две роли, в бъдеще може да се добави контролер който е отговорен за добавянето на нова роля. Друг важен пакет е repository, отговорен за взаимодействието с базата с данни, писането на SQL заявки, които да са свързани с определен метод и т.н., като в този пакет ще бъдат само интерфейси. Това го съобразяваме спрямо изискванията на фреймуърка, чрез който ще разработваме системата.

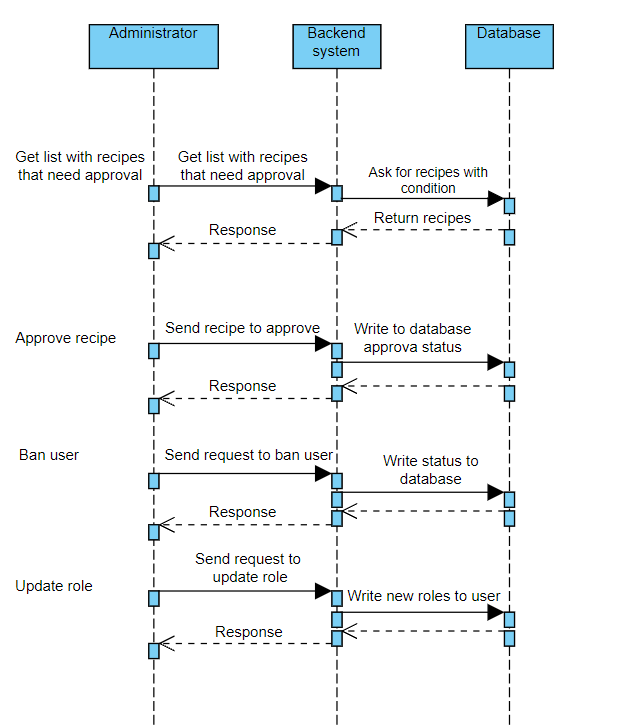
## **4.3 Изглед на процесите. Описание на отделните процеси и системи**

На фиг. 3.1. е показана диаграма за последователност на един обикновен потребител. Както по-горе споменахме, всеки потребител ще може да се регистрира и вписва в системата. Въвеждат се данните, прави се съответната заявка и след няколко проверки системата връща отговор дали е направена успешна регистрация или съответно успешно логване. Основната функционалност е създаването на рецепти, като когато се създаде такава тя първо трябва да бъде проверена от администратор, за да подсигури надеждността на информацията и след това бива запазена в базата данни. На същият принцип е и функционалността да се променя дадена рецепта. Един потребител също така може да търси друг потребител, като по този начин може да се достъпва списък с рецепти на даден човек. Освен потребител ще може да се търси и конкретна рецепта, както и да се коментира такава. Това дава възможността да се подобри дадена рецепта чрез конструктивен фийдбак.



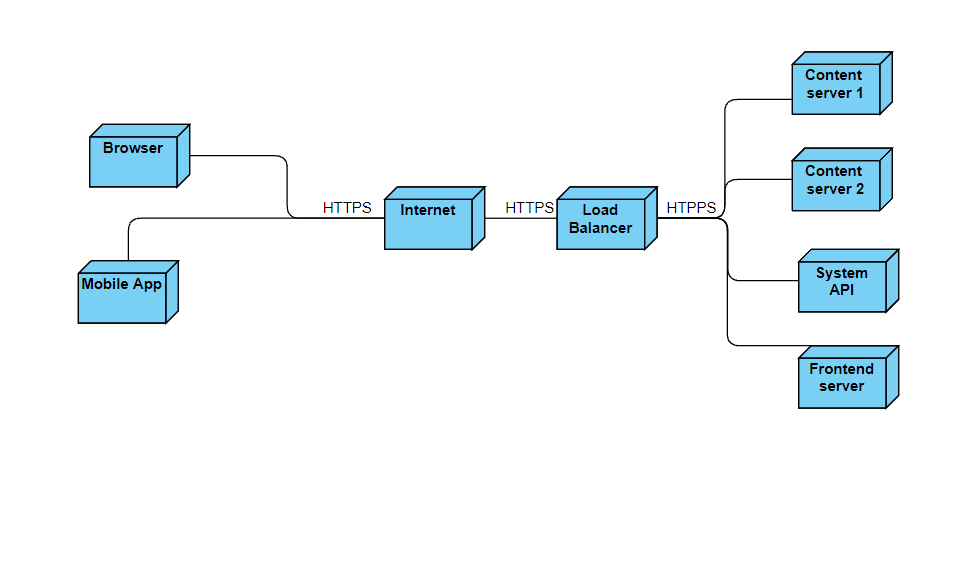
Фиг. 3.1. Диаграма на последователност на обикновен потребител

Не по-малко важна е диаграмата за последователност на системния администратор, тъй като той има доста повече функционалности от обикновен потребител. Тя е представена на фиг. 3.2. Както вече споменахме, всяка рецепта ще бъде качена след като е проверена от администратор. Когато той се логне в системата може да направи заявка, за да вземе листа с рецепти, които очакват неговото одобрение или да търси конкретна таква.След като я прегледа може да я одобри или не, като следователно тя бива качена или изтрита (например, поради невярна, неточна информация). Също така системният администратор е единственият, който може да спира достъпа на даден потребител до системата (да го банне) и само той управлява ролите.



фиг. 3.2. Диаграма на последователност на системен администратор

## **4.4 Изглед на внедряването:**

Фиг. 4.1. Изглед на внедряването

Изгледът на внедряването е показан на фиг. 4.1. Достъпът до системата ще се осъществява чрез мобилно iOS приложение, което чрез защитената версия на HTTP – HTTPS, осигурява криптирано транспортиране на данните в пакетите през интернет, което подобрява защитата от подслушване на пакетите и трафика и крадене на информация. От там той достъпва разпределителя на натоварване (чиято имплементация ще бъде когато продукта достигне до повече потребители), от там спрямо заявката ще бъде пренасочен до сървъра, които ще я обслужи - дали API-то, изпълняващо бизнес логиката на приложението, дали фронт-енд сървъра или сървъра предоставящ файлово съдържание, като картини и т.н.. Нужно е да бъде казано, че чрез сървъра за балансиране на натоварването, ще можем да следим натоварването на отделните сървъри и ще избираме оптималния за изпълнението на заявката ни. Също така тези услуги може да са по няколко, както на схемата е показано чрез сървъра за предоставяне на съдържание. Също така при отказ на някоя от системите, балансьорът може да се включи и да пренасочва трафика към другите активни, като за тази цел ще трябва да следи състоянията на отделните сървъри.

# **Обосновка защо архитектурата осигурява адекватно решение на проблема**

Нефункционалните изисквания, които системата трябва да покрие са производителността – системата трябва да извършва както най-тежките, така и най-леките операции за по-малко от 2 – 3 секунди, което според стандартите днес трябва да е целево време за зареждането на едно уеб приложение, независимо от броя на потребителите едновременно ползващи системата. Чрез използването на архитектура базирана на микро-услуги, може да гарантираме подобряване на компонентите, които имат завишена натовареност. Подобрението, както е написано по-рано има два вида: хоризонтално и вертикално, то лесно може да бъде постигнато за отделните компоненти на системата и всеки компонент може да се скалира, спрямо вида ресурси, които са му нужни, дали повече система, които да обработват заявките или повече място за съхранение на файлове, или пък повече процесорна мощност. Друго предимство на микро-услугите e по-лесното вкарване на нови софтуерни разработчици в проекта. Също така предимство е и и наличието на среди като **Docker**, които предоставят лесна контейнеризация на отделните услуги, което пък от своя страна прави по-лесен процеса на **CI/CD**, съответно и при наличието на бъг, по-бърз процеса по отстраняването му.

С прилагането на **OOP** и **S.O.L.I.D**. принципите, ще гарантираме разширяемостта му, като при поява на нова функция на приложението ще дава възможността за по-лесното и имплементиране, чрез абстракциите, които са създадени.

Друго важно нефункционално изискване е сигурността, като за него основна грижа трябва да полага, както дизайна на системата, който трябва да спазва най-добрите практики за сигурност, като пренасяне на данните под форма на HTTPS, а не по-простия HTTP протокол, както и чрез строгото определяне кой може да дава администраторски роли в приложението, и достъпа до чувствителни данни да имат само администраторите.

Последно, но не по важност е че трябва да осигурим системата да бъде устойчива, при какъвто и да е срив, трябва да се правят записи на състоянията на системата и тези записи трябва да са максимално актуални, така че при определен срив да може бързо да се пусне алтернативен сървър, който да поеме потребителските заявки. Възможно е използването и на облачни услуги като тези на Амазон, Гугъл и Майкрософт, като те гарантират бързото възстановяване на сървъра при отказ, както и актуални снапшоти на системата, чрез които да я възстановим при отказ.

# **Използвани термини и символи:**

* CI/CD - continuous integration / continuous delivery
* OOP – Object-Oriented Programming
* SOLID – Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle
* MVVM – Model-View-ViewModel
* HTTP – hypertext transfer protocol
* HTTPS - hypertext transfer protocol secure

# **Използвани източници:**

* Лекции и презентации за UML диаграмите, архитектурни стилове и атрибути за качество от курса “Софтуерни архитектури”
* Знания от предишни курсове (ООП, СУБД, СА, ИБТ)
* <https://www.clariontech.com/blog/mvvm-in-ios-a-quick-walkthrough> - MVVM in iOS