|  |
| --- |
| **ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**  **Факултет – „Компютърни системи и технологии“** |

**ДИПЛОМНА РАБОТА**

**„Наблюдение дейността на студентите по време на изпити провеждани в електронен вариант“**

**Студент: Иван Николаев Колев**

**фак. № 121213114**

**Специалност: КСИ**

**Дата: Научен ръководител:**

**маг. инж. Д. Андреев**

**София  
2017**

**СЪДЪРЖАНИЕ**

[Въведение 4](#_Toc493635782)

[Първа глава 5](#_Toc493635783)

[1. Постановка на дипломната работа. Цели и задачи. 5](#_Toc493635784)

[1.1. Поток от данни. 5](#_Toc493635785)

[1.2. Вид на събираните данни. 5](#_Toc493635786)

[1.3. Структура на събираните данни. 6](#_Toc493635787)

[1.4. Ресурсоемкост на приложението за събиране на данни. 6](#_Toc493635788)

[1.5. Надеждност и сигурност. 6](#_Toc493635789)

[1.5.1. Балансиране товара на сървърите (Load balancing) 7](#_Toc493635790)

[1.5.2. Управление на потребителските сесии 7](#_Toc493635791)

[1.5.3. Защита на вътрешните системи 7](#_Toc493635792)

[1.5.4. JSON Web Token 8](#_Toc493635793)

[1.6. Скалируемост на софтуерната система. 9](#_Toc493635794)

[1.7. Ресурсоемкост на софтуерната система за обработка и съхранение на данните. 9](#_Toc493635795)

[Втора глава 11](#_Toc493635796)

[2. Функционално описание на софтуерната система. 11](#_Toc493635797)

[2.1. Програмни езици и библиотеки. 11](#_Toc493635798)

[2.2. Бази за съхранение на данните (Database servers). 11](#_Toc493635799)

[2.3. Поточна линия за пренос на съобщения между системите (Message bus) 12](#_Toc493635800)

[2.3.1. Настройване производителността на Apache Kafka 13](#_Toc493635801)

[2.3.2. Замерване производителността на Apache Kafka 14](#_Toc493635802)

[2.4. Наблюдение на системите 15](#_Toc493635803)

[2.4.1. Наблюдение на логовете 15](#_Toc493635804)

[2.4.2. Наблюдение на метрики от системите. 17](#_Toc493635805)

[Трета глава 18](#_Toc493635806)

[3. Програмна реализация 18](#_Toc493635807)

[3.1 Софтуерна архитектура 20](#_Toc493635808)

[3.2. Структура на основните модели (модели за трансфер на данни и домейн модели). 21](#_Toc493635809)

[3.3. Структура на програмните интерфейси за събиране на данни от OS. 24](#_Toc493635810)

[3.4. Структура на десктоп приложението за събиране на данни 27](#_Toc493635811)

[3.5. Структура на front-end сървъра за получване на данни 32](#_Toc493635812)

[3.6. Структура на приложението за записване и обработка на получените данни 32](#_Toc493635813)

[3.7. Структура на уеб сървъра за визуализиране на получените данни в реално време 32](#_Toc493635814)

[Четвърта глава 33](#_Toc493635815)

[4. Указания за ползване 33](#_Toc493635816)

[Пета глава 34](#_Toc493635817)

[5. Експериментални данни 34](#_Toc493635818)

[5.1 Описание на експеримента и тестовата установка 34](#_Toc493635819)

[Заключение 35](#_Toc493635820)

[Източници 36](#_Toc493635821)

[Приложение 1 - речник 39](#_Toc493635822)

[Приложение 2 - изходен код 40](#_Toc493635823)

# Въведение

Една от основните цели на преподавателите в училища и университети е да осигуряват подходящи и равни условия за провеждане на изпити, контролни и упражнения. До голяма степен това се изразява в следене на студентите да не преписват от помощни материали, да работят самостоятелно по възложените задачи и да не се отклоняват от целите на упражнението или изпита. Това представлява трудна задача, особено при зали с капацитет повече от пет души, защото фокусът на преподавателя не може да бъде върху повече от един човек едновременно и винаги съществува възможността да изпусне моменти, които няма как да върне и види отново. Именно тези моменти студентите ползват, за да използват забранени материали, да препишат или да отворят сайта на любимата си социална мрежа, в която да прекарат времето си.

Изследвания проведени по темата за преписване показват стряскащи числа: процентът на студенти/ученици, които са преписвали поне веднъж започва от 17% и стига до 98% в различни части на света. Част от причините за това са, че за да останеш „конкурентоспособен“ си „длъжен“ да преписваш, защото други - по-слаби ученици, го правят и на резултати изглеждат по-способни от теб.

Компютрите и технологиите като цяло стават все по-достъпни и повече училища и университети започват да разчитат на тях за провеждане на упражнения, изпити и контролни. От една страна работата по следене дейността на цяла група студенти е непосилна за сам преподавател, но от друга страна, е сравнително лесно да се използват наличните технологии за събиране и анализ на данни свързани с дейностите, които дадено лице извърша на компютъра си.

Основният фокус на текущата дипломна работа е наблюдението дейността на студентите по време на изпити, контролни и упражнения, провеждани в електронен вариант с достъп до интернет, по време на които изпитваното лице трябва да спазва набор от правила и не трябва да използва забранени канали за комуникация, да търси или публикува въпроси във форуми и други вид информационни сайтове, да използва торент тракери и т.н.

Чрез използване на програмно приложение за Windows OS, студентите ще могат да се „регистрират“ в системата и стартират нова „работна сесия“. По време на работната сесия приложението ще събира информация за процесите, стартирани в операционната система за текущия фокусиран процес, снимки на дисплея, снимки от уеб камерата, натиснати клавиши, изпратени и получени мрежови пакети.

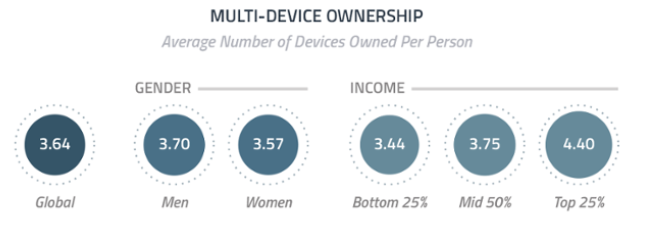
Събраната информация ще може да бъде разгледана и анализирана в реално време или в последващ период. С предложената имплементация се очаква като резултат, по-ефективно управление на учебните и изпитни процеси от страна на преподавателите, по-голяма прозрачност на действията и по-висока ангажираност на студентите.

# Първа глава

# 1. Постановка на дипломната работа. Цели и задачи.

## 1.1. Поток от данни.

За последните 20 години непрестанният технологичен възход промени значително начина, по който хората живеят, работят, комуникират и почиват. Масово притежаваме между 3 и 4 мобилни устройства, които ползваме активно през по-голямата част от денонощието[3]. Мобилните устройства могат да бъдат лаптопи, таблети, умни телефони, умни часовници, умни гривни, умни очила, електронни четци за книги, музикални устройства, камери, фотоапарати и т.н. Всяко едно от тези устройства в днешно време има опция за свързване към интернет, към умен телефон или към лаптоп. Това означава постоянен обмен на данни, данни които неусетно могат да станат публични и да бъдат използвани за изграждане на дигитален отпечатък на човека, притежаващ устройствата.



***Фигура 1.*** *Статистика за средния брой притежавани мобилни устройства в глобален мащаб.*

Потокът от данни включва, но не се ограничава до: уеб сайтовете, които посещаваме и времето, прекарано в тях; скоростта, с която натискаме последователно клавиши на клавиатурата; софтуерните приложения, които инсталираме и използваме; имейлите, които изпращаме; хората, с които сме приятели и познаваме; стилът музика, който предпочитаме и любимите ни песни; местата, които посещаваме и пътищата, по които минаваме (благодарение на вградени GPS системи и триангулация от страна на мобилните оператори към които сме абонирани) и т.н.

Всеки ден, използвайки преносими или стационарни устройства, средностатистическият човек може да генерира стотици гигабайти информация. Голяма част от тази информация вече се използва от маркетинг компании с цел продуктово позициониране, показване на подходящи реклами и промоции, целящи да подтикнат човека да закупи вещи, които са в сферата на неговите интереси. Този голям обем от данни може да бъде използван, както за добронамерени, така и за злонамерени цели. Затова е важно всеки потребител да използва технологиите с предпазливост и с ясното съзнание, че всеки момент може с действията си да стане уязвим.

## 1.2. Вид на събираните данни.

За целта на приложението, ще се събират данни свързани с начина, по който потребителят използва компютъра, а именно:

* Стартирани процеси/приложения (active processes).
* Текущите процеси на фокус (foreground processes).
* Изпратени и получени мрежови пакети (network traffic).
* Моментни снимки на дисплея, който потребителят вижда (display snapshots).
* Моментни снимки през уебкамерата, ако такава съществува (camera snapshots).
* Натиснати клавиши и скоростта на писане (captured keyboard keys).

## 1.3. Структура на събираните данни.

За целта на системата, данните ще следват един и същ шаблон. Всеки прихванат обект, ще принадлежи на определена „сесия“. Сесия, в контекста на системата, означава всяко ново стартиране на процес за наблюдение.

Всеки прихванат обект, ще се състои от следните 5 полета:

* Име на потребителя.
* Идентификационен номер на потребителя.
* Идентификационен номер на текущата сесия.
* Дата на стартиране на сесията.
* Обектът, който е прихванат от приложението (натиснат клавиш, стартиран процес, фокусиране на нов процес, снимка на дисплея, снимка на уебкамерата или мрежови пакет).

## 1.4. Ресурсоемкост на приложението за събиране на данни.

С оглед естеството на данните, които ще се събират, основната част от системата за наблюдение, а именно десктоп приложението, трябва да бъде конфигуруемо.

Следните параметри са от основно значение за ресурсоемкостта му:

* Период, през който ще се правят снимки на дисплея.
* Период, през който ще се правят снимки на уеб камерата.
* Период, през който ще се изпращат агрегираните данни на отдалечения сървър за съхранение.

Информация за активните процеси ще се прихваща само когато нов процес бива стартиран. Освен това такава за процеса на фокус ще се прихваща само когато се фокусира процес, различен от текущия. Към това информация за натиснатите клавиши ще се прихваща при всяко натискане на клавиш. От прихванатите мрежови пакети ще се записват само първите 24 байта информация, за да се избегне претоварване на мрежата или консумиране на прекалено много RAM памет за временното съхранение на пакетите.

С оглед предоставената информация само първите три изброени параметъра ще оказват голямо влияние на разхода на ресурси от приложението, затова те трябва да бъдат избрани внимателно, спрямо машината, на която ще работи продуктът. Конфигурациите ще бъдат четени от конфигурационен JSON файл и ще могат да бъдат променяни, спрямо нуждите на различните сесии/потребители/машини.

## 1.5. Надеждност и сигурност.

Първото нещо, за което трябва да мислим, когато работим с потребителска информация, е „Законът за защита на потребителските данни“. Всичко, което е чувствителна информация, или може да навреди по един или друг начин, ако попадне в злонамерени ръце, трябва да бъде съхранявано по сигурен начин. За целта всяко едно предаване на потребителски данни от компютъра домакин до сървърите, които ще обработват тази информация, трябва да се случва, като информацията се криптира, изпраща се на сървъра приемник, а той я декриптира и използва в оригиналния й вид. За жалост, все още не съществува 100% сигурен начин за предаване на информация между 2 отдалечени точки и методите, които ще използваме, могат единствено да затруднят по-опитните специалисти.

Второто нещо, което искаме да подсигурим, е интегритетът на данните и информацията, която постъпва в нашата система. Понеже ще градим решение, което може да скалира достатъчно добре за голям обем от потребители първата линия, която обработва всички потребителски заявки, е ферма от front-end уеб сървъри, чиято единствена цел е да получат данните, да идентифицират потребителя, който ги е изпратил, и след това да ги запишат в съответното хранилище, където да бъдат поети за обработка по поточната „тръба“.

Ако при ферма от сървъри за управление на потребителските сесии, разчитаме на това състояние да бъде пазено на сървъра, ще си създадем потенциални проблеми и неудобства за самите потребители.

### 1.5.1. Балансиране товара на сървърите (Load balancing)

Има няколко техники за маршрутизиране на потребителските заявки при наличие на ферма от сървъри:

* “Sticky session” е стратегия, при която пред фермата стои load balancer, който прихваща потребителска заявка и винаги за дадения потребител, маршрутизира заявката към един и същ уеб сървър. Създава се таблица с връзки „user” – “server”. По този начин се гарантира, че потребителят няма да има проблеми със сесиите си, както ако бъде маршрутизиран към друг сървър. Тази стратегия не е много надеждна, защото при непланирано спиране на съответния сървър, всички потребителски сесии ще бъдат прекъснати, а load-balancer-ът ще пренасочи потребителите към нов активен сървър;
* “Round robin” е стратегия, при която пред фермата отново стои load balancer, който избира следващия сървър, който да обработи заявката на ротационен принцип. Това е най-семплият начин за балансиране на натоварването между уеб сървърите, но отново води до проблеми при управление на потребителските сесии, ако те се съхраняват на ниво сървър.

### 1.5.2. Управление на потребителските сесии

Фазата на идентификация е деликатен момент при наличието на ферма от дистрибутирани уеб сървъри. За решаване на този проблем ще използваме RESTful архитектура, чиято основна цел е да не пазим потребителско състояние/сесия на сървъра, а цялата необходима информация да се предава всеки път, криптирана в бисквитка на браузъра (browser cookie) или като част от тялото на заявката, която ще се декриптира с машинен ключ на ниво уеб сървър и ще се проверява за автентичност. След като е решен проблемът с идентификацията на потребителите, трябва да подсигурим, че всички „входни“ точки в нашите уеб сървъри изискват съответното ниво на автентикация преди да допуснат каквито и да било данни, напред по поточната линия. Входните точки още се наричат “endpoint”-и [mark] на уеб сървъра. Това са адресите, към които изпращаме заявки за извличане или създаване на информация.

### 1.5.3. Защита на вътрешните системи

Друга съществена част при защитаване на софтуерната система е подсигуряване комуникацията между отделните модули и валидиране на автентичността на всяка заявка вътрешно в системата.

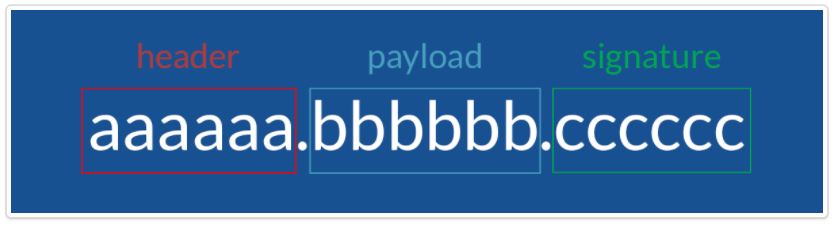
За решаване на този проблем стандартно се ползват две стратегии:

* Конфигуриране на защитна стена (firewall), която да допуска заявки само от определени IP адреси.
* Изпращане на идентификационен токен, който да съдържа идентичността на системата, която прави заявката.

За идентификацията между системите ще използваме много семпла и надеждна концепция, а именно по-долу представената “JWT” (JSON Web Token) [mark].

### 1.5.4. JSON Web Token

Накратко JWT представлява отворен стандарт (**RFC 7519**), който дефинира компактен начин за сигурно предаване на информация (payload) между различни страни под формата на JSON обект. Предадената информация може да бъде проверена посредством дигитален подпис. JWT токените могат да бъдат подписани, използвайки „тайна” ключова дума (за **HMAC** алгоритъм) или двойка публичен/частен ключ (за **RSA** алгоритъм)



***Фигура 2.*** *Структура на base64-encoded JWT.*

**JWT е подходящ за следните случаи**:

* **Автентикация** – това е най-честият случай за използване на JWT. Веднъж щом потребителят се идентифицира в системата, всяка последваща заявка ще включва токена, позволявайки на потребителят да достъпва пътища, услуги и ресурси, които изискват съответното ниво за достъп. Single Sign On функционалността широко използва JWT в днешно време, заради малкия му размер, бързодействието което осигурява и възможността да се споделя лесно между различни домейни.
* **Предаване на съобщения** – JSON Web Token-ите са добър начин за сигурен пренос на информация между две или повече страни. Поради факта, че тези токени са подписани, можем да сме почти сигурни, че страната, изпращаща информацията, е тази, за която се представя. В допълнение, подписът се генерира, използвайки хедърите и самото съобщение, съответно можем да валидираме, че съдържанието не е фалшифицирано.

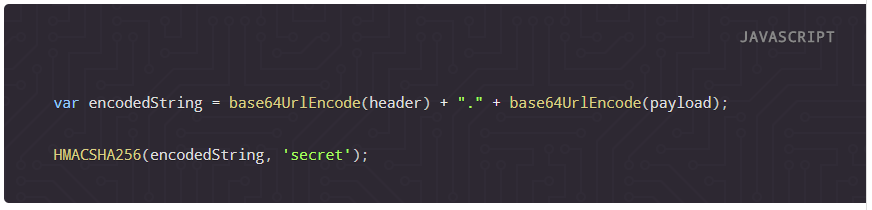


***Фигура 3****. Примерен Header на JWT.*

За надеждна и сигурна работа, използвайки този тип токени, ключът, който се използва за подписването им трябва да бъде ръчно сложен в конфигурационния файл на системата, на скрито, непублично място. Добра практика е през определен период от време да се подменя ключът за подписване.



***Фигура 4.*** *Примерен Payload на JWT.*



***Фигура 5.*** *Примерно подписване (signature) на JWT посредством JavaScript.*

## 1.6. Скалируемост на софтуерната система.

Тъй като трябва да създадем надеждна система, която да продължи да работи дори при внезапно спиране на някой от съставните й модули, товa означава, че трябва да мислим за разпределение и мултипликиране на отделните сървъри. Не искаме всичките ни ресурси да бъдат на едно централизирано място, защото това означава, че при най-малък проблем като спиране на захранването или дефект в компютъра домакин, цялата ни система ще преустанови работата си.

За целта, трябва да изберем подходящи технологии и стратегии за изграждане на системата по начин, който ще я направи независима от местоположение, устойчива и скалируема. Под скалируемост се има предвид системата да може да се разширява хоризонтално (вдигане на инстанции върху нови физически или виртуални машини), за да може да поеме по-голям работен товар, когато това е наложително. Например, ако имаме университет с 30 000 студенти, които всеки ден провеждат поне едно упражнение, включващо употребата на компютри, това означава, че нашата система ще трябва да може да поеме голям обем входни данни, които съответно трябва да обработи в реално време, без да има осезаемо забавяне или срив.

## 1.7. Ресурсоемкост на софтуерната система за обработка и съхранение на данните.

Със скалирането на сървърите разходите за ползване на хардуерни ресурси могат да скочат драстично, ако изборът на технологии и дизайнът на системата не са добре обмислени. За да ограничим разходите до разумен минимум, ще целим да използваме основно Linux-базирани операционни системи и системи с отворен код, които са безплатни за ползване с комерсиални цели, и които се характеризират с ниска цена за притежание (low-cost of ownership).

Наборът от системи ще включва:

* Apache Zookeeper (Linux).
* Apache Kafka (Linux).
* ElasticSearch (Linux).
* Kibana (Linux).
* Grafana (Linux).
* Logstash (Linux).
* MongoDB (Linux).
* ASP.NET Web Api (Windows, но може да бъде пренесен на Linux операционна система, ако приложението е разработено върху .NET Core платформата).
* SQL Server (Windows).

Аргументация за избора на технологичен стек ще бъде представена в следващата глава.

# Втора глава

## 2. Функционално описание на софтуерната система.

## 2.1. Програмни езици и библиотеки.

Основният програмен език, който ще използваме, е **C#** - той е създаден да се изпълнява от .NET Framework [mark]. Езикът е разработен от Microsoft, а общността, която го използва е много широка. Съществуват огромен брой библиотеки, примери с код и решения на често срещани проблеми, които могат лесно да бъдат използвани в други проекти.

Езикът е обектно-ориентиран със силна типизация и мощна поддръжка на Generic класове и функции, което позволява гъвкавост, надеждност и изграждане на добра абстракция. Друг плюс е, че зад него стои корпорация, чиито продукти са базирани на него, което автоматично означава, че езикът ще продължи да се поддържа и развива, докато Microsoft съществува.

Друг плюс е, че с пускането на **.NET Core**, C# става **мултиплатформен** и може да работи под Линукс-базирани операционни системи, като например **Ubuntu, Fedora, Red Hat Enterprise, Debian, CentOS, SUSE/openSUSE**. Това на свой ред води до факта, че системи, разработени на C#, ще могат да бъдат инсталирани на безплатен Linux сървър, вместо на платен Windows сървър, а това автоматично редуцира TCO[mark]. От 2 години насам, езикът и целият фреймуърк под него стават с отворен код и всеки, който иска, може да допринесе за развитието му или за поправянето на открити проблеми като направи заявка в официалното хранилище [mark], където е публикуван кодът.

Технологичният стек, който ще използваме за потребителския интерфейс, включва HTML, CSS и JavaScript, като за манипулации различните елементи (бутони, текстови полета, изображения и т.н.) - ще ползваме библиотека, която е изградена на база JavaScript езика, а именно JQuery. Тя ни предоставя програмен интерфейс, през който лесно можем да обработваме, трием и създаваме елементи от потребителския такъв. Към това посредством HTML ще бъде изградена само и единствено структурата на дадения документ, а използвайки CSS – ще бъдат стилизирани елементите и техния външен вид.

За комуникация с всички останали модули и системи, ще използваме драйвери, изградени специално за езика C#.

* Apache Kafka – Official Confluent Kafka .NET Driver [mark].
* MongoDB – Official MongoDB .NET Driver [mark].
* Microsoft SQL Server – Entity Framework.

## 2.2. Бази за съхранение на данните (Database servers).

Изключително важен е изборът на database server за съхранение на различните видове данни и операции, които ще запазва и извършва софтуерната система. В случая избираме стратегия отново с ясната идея, че трябва да можем да скалираме хоризонтално (добавяне на повече машини и инстанции на дадена услуга), а не вертикално (добавяне на повече хардуерни ресурси в една единствена машина).

Стратегията, която ще използваме е следната:

* За всички важни потребителски данни, които трябва да бъдат съхранявани по сигурен и консистентен начин, използваме SQL Server. Причината за това е, че по подразбиране, SQL Server-ите носят със себе си множество функционалности, свързани със сигурността на достъп и интегритета на данните, като например User Authentication[mark]/Windows Authentication, различни роли за контролиране нивата на достъп до съответни функционалности, криптиране на данните, трансакции и т.н.;
* За всички функционалности, свързани с търсене на логове и метрики от различните системи, използваме ElasticSearch engine, който представлява индекс силно оптимизиран за търсене;
* За всички останали оперативни заявки, свързани със записване, четене и обработка на данните свързани с дейността на студентите, използваме MongoDB, заради гъвкавостта и скоростта която предоставя;

За оперативните данни ще се използва NoSQL база, заради предимствата, които има в случая пред релационните SQL бази. Релационните бази се сблъскват със следните предизвикателства:

* Не са удачни при работа с големи обеми от данни и широк набор от типове на данните (изображения, видеа, текст и т.н.);
* Не могат да скалират вертикално, защото една машина е ограничена от обема хардуерни ресурси, които може да управлява ефективно (предимно RAM и CPU);
* Не могат да скалират хоризонтално, защото са ограничени от кеш-зависими операции за четене и писане;
* Не могат да бъдат раздробявани и разпръсквани на различни машини (nodes);
* Сложен релационен модел, който прави трудна промяната на схемите (добавяне на нови колони, промяна или махане на стари и т.н.);

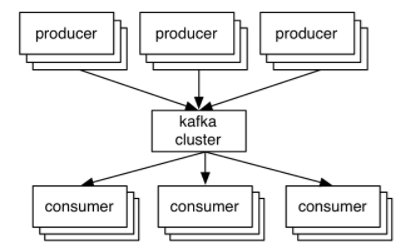
Нерелационните бази (NoSQL) атакуват проблемите, които релационните бази срещат по следния начин:

* Създадени с мисъл за хоризонтално скалиране и нулева зависимост между отделните колекции, документи и т.н. Възможност да работят на огромен брой разпределени възли (nodes);
* Незаключващ документите контрол при конкурентни операции, за да могат операции, свързани с четене, да не бъдат блокирани от операции, свързани с писане/промяна на документа;
* Скалиране на репликациите и висока дистрибутивност (могат да достигнат хиляди машини);
* Архитектура, която предоставя по-висока производителност на машина отколкото при релационните бази;
* Модел на данните, който не се контролира от схема/шаблон, което позволява голяма гъвкавост при работа с различни данни;

В повечето случаи производителността на NoSQL базите е ограничена от скоростта на мрежата (network-bound) по която минава заявката. При добра инфраструктура, това означава между 2 и 10 милисекунди на заявка. Като обобщение – изборът на база трябва да зависи изцяло от конкретния случай на ползване и да бъде добре обмислен.

## 2.3. Поточна линия за пренос на съобщения между системите (Message bus)

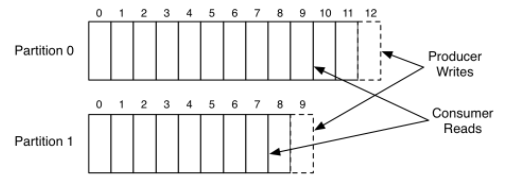
За предаване на съобщения между отделните системи, ще използваме Apache Kafka, която е известна със способността си за приемане на стотици хиляди съобщения в секунда, което перфектно пасва на нуждите ни от буфер, който да може своевременно да съхранява всички данни, които се събират от студентите.



***Фигура 6.*** *Основна схема на работа на Apache Kafka*

Кафка имплементира publish – subscribe модела. Той представлява начин да се абонираме към дадена тема (topic) и в момента, в който някой публикува нещо, свързано с тази тема, автоматично да получим известие/съобщение. Процесът на работа с Кафка, е сравнително прост: Производител (Producer[mark]) изпраща съобщение относно дадена тема (Topic[mark]) до Кафка, която на свой ред записва съобщението на твърд диск и след това го препраща към всички абонирани за тази тема потребители (Consumers[mark]).

Ключовата абстракция при Кафка е темата. Тя представлява разделен регистрационен дневник (sharded write-ahead log). Производителите добавят записи към тези дневници, а потребителите се абонират за промени. Всеки запис е свързана двойка ключ/стойност. Ключът се използва за записване на стойността(съобщението) в съответстващ регистрационен дял (log partition).



***Фигура 7****. Пример за производител и консуматор, които четат и пишат в двудялова тема.*

Всеки запис се асоциира с входящ номер, който наричаме „отстояние“ (offset). Това отстояние се използва от консуматорите и указва позицията, до която са стигнали във всеки един от логовете, които четат. Регистрационните дялове, могат да бъдат разпръснати по различни машини в рамките на един Кафка-клъстер (съвкупност от Кафка-брокери), което позволява една тема да съхранява повече данни, отколкото могат да се поберат на единична машина.

Важно е да се отбележи, че за разлика от други системи за предаване на съобщения, Кафка-логовете винаги са персистентни. Съобщенията се записват на файловата система в момента на получаването им. Съобщенията не се изтриват, когато биват прочетени, а се запазват за определен срок от време (който може да бъде конфигуриран, спрямо нуждите на системата, например две седмици). Това покрива случаи, когато бихме искали повторно да проиграем (обработим) последователност от съобщения. Също така, Кафка позволява репликиране на логовете между множество различни сървъри с цел устойчивост на грешки. Например, ако един от Кафка-брокерите спре, консуматорите ще могат да бъдат прикачени към друг жив брокер и ще продължат нормален работен процес.

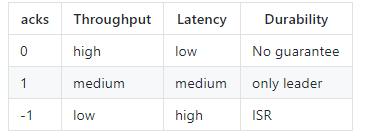
## 2.3.1. Настройване производителността на Apache Kafka

Кафка идва с конфигурация по подразбиране, която работи изключително бързо в повечето от случаите и не се налага допълнително конфигуриране на отделните модули, но е важно да се отбележи, че има няколко елемента от значителна важност за бързодействието:

* Размер на твърдия диск (по-голям размер означава по-голям период на задържане на съобщенията);
* Скорост на писане и четене от твърдия диск (по-висока скорост означава по-голямо бързодействие на Производителите);
* Капацитет на мрежата (по-голям капацитет означава по-висока пропускливост на съобщения);
* Размер на енергозависимата памет (повече RAM означава по-висока производителност на Консуматорите);

Друг важен фактор е дали записваните съобщения ще бъдат компресирани. Кафка предоставя няколко различни алгоритми за компресия, като всеки от тях си има плюсове и минуси в зависимост от структурата и размера на съобщенията. Наличието на компресия ще редуцира значително размера на използваното дисково пространство, но и ще намали малко пропускливостта на клъстера.

Последният фактор, който значително влияе на бързодействието на Кафка- производителя, е така наречената „надеждност“ (durability) при предаване на съобщенията. Нивото на надеждност се настройва от конфигурация, която се казва „request.required.acks”, като спрямо зададеното ниво се променят параметрите: пропускливост (throughput); забавяне (latency); надеждност (durability).



***Фигура 8.*** *Таблица на зависимостите при различни стойности на „request.required.acks”.*

### 2.3.2. Замерване производителността на Apache Kafka

Параметри на физическата машина, на която са проведени тестовете[mark]:

* + - процесор Intel Xeon 2.5 GHz с 6 ядра.
    - шест твърди диска 7200 RPM SATA.
    - 32 GB RAM памет.
    - 1Gb мрежова карта.
    - Размер на записваното съобщение – 100 bytes.
* При един единствен Кафка-производител, записващ в една тема с 6 дяла и без репликация на съобщенията, резултатът е: **821,557 записа в секунда** (78.3 MB/s).
* При един единствен Кафка-производител, 3x асинхронна репликация на съобщенията, резултатът е: **786,980 записа в секунда** (75.1 MB/s). Асинхронна репликация означава, че брокерът, който получава съобщението, връща веднага потвърждение без да чака резултат от останалите два брокера, към които е пратил заявката за репликация.
* При един единствен Кафка-производител, 3x синхронна репликация на съобщенията, резултатът е: **421,823 записа в секунда** (40.2 MB/s). Синхронна репликация означава, че брокерът, който получава съобщението, връща потвърждение, чак след като е получил потвърждение от останалите два брокера, към които е пратил заявката за репликация.
* При един единствен Кафка консуматор, абониран за една тема, резултатите са: **940,521 прочетени записа в секунда** (89.7 MB/s).
* При три Кафка-консуматора, абонирани за една и съща тема, резултатите са: **2,615,968 прочетени записа в секунда** (249.5 MB/s).
* При един Кафка-консуматор и един Кафка-производител, които работят едновременно, резултатите са: **795,064 записа в секунда** (75.8 MB/s).

Забавянето от момента на записване на съобщението до момента на прочитане на съобщението е: **2ms** за **98%** от записите; **3ms** за **1.9%** от записите; **14ms** за **0.1%** от записите. От тук нататък няма причина да продължаваме с тестовете, защото тези метрики покриват нуждите ни предостатъчно.

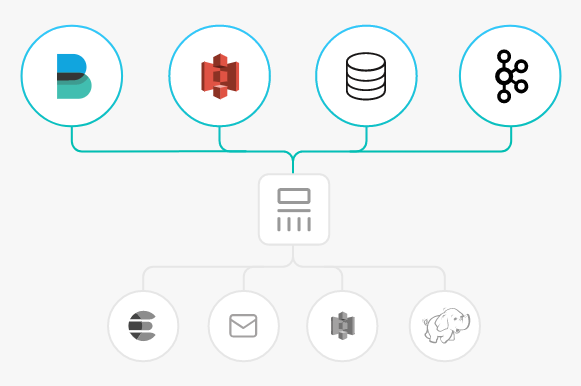
## 2.4. Наблюдение на системите

При система, състояща се от множество отделни модули, които трябва да работят в синхрон, е от огромно значение да имаме начин за наблюдение на процесите, които се случват в реално време. Това означава да знаем кога нова заявка е достигнала до сървъра ни, кога е била обработена, какъв е резултатът от обработката на тази заявка, от кой потребител е дошла, за колко време е минала през поточната линия и т.н. Също така е полезно да имаме достъп до всяка една грешка или изключение, което е възникнало по време на работата на нашата система, например: връзка между уеб сървър и база от данни е прекъсната по време на трансфер на данни или просто не може да бъде установена; изпратена заявка до друг уеб сървър е неуспешна и е върнат код за грешка; входни данни не са в очаквания формат и не могат да бъдат трансформирани и т.н. Възможността за записване и проследяване на всяко едно подобно събитие е ключът към ефективната поддръжка на една софтуерна система.

### 2.4.1. Наблюдение на логовете

За наблюдение и на логовете и лесно търсене измежду милионите записи, ще използваме стандартен ELK стек (ElasticSearch, Logstash & Kibana). Комбинацията от тези три системи ни дава невероятна гъвкавост и лекота при работата с логове.

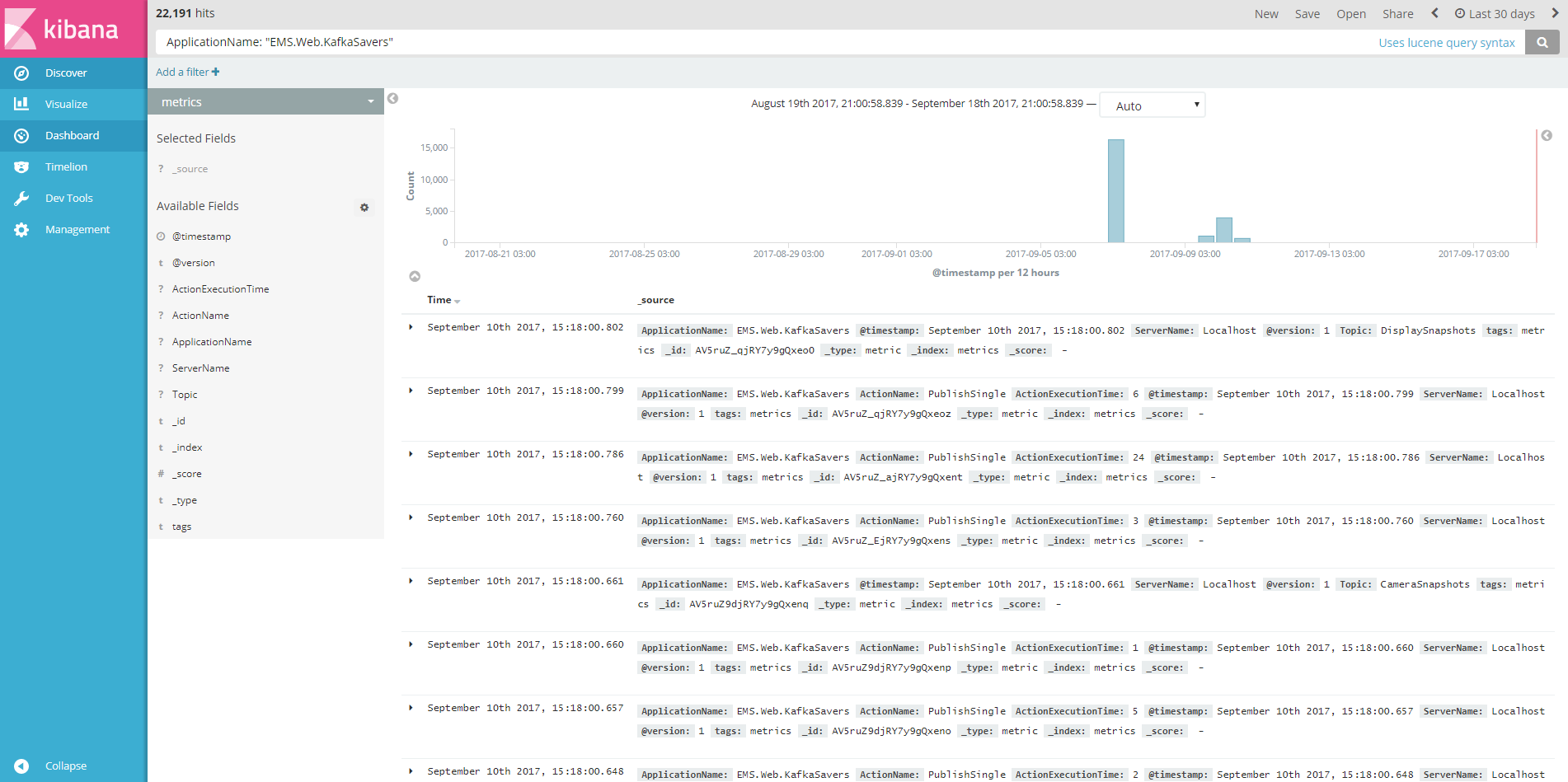
**Logstash** [mark] представлява поточна линия с вградени функционалности за четене, обработка и записване на входни данни от и към множество източници. Системата е реализирана на Ruby и използва Ruby Gems за разширяване на съществуващите функционалности. Общността, която подпомага развитието на Logstash, е разработила десетки плъгини, които можем да ползваме наготово за нашите цели. Също така, авторите са отворили кода публично в Github [mark] и той е достъпен за преглед и модификация от всеки, който има интерес.



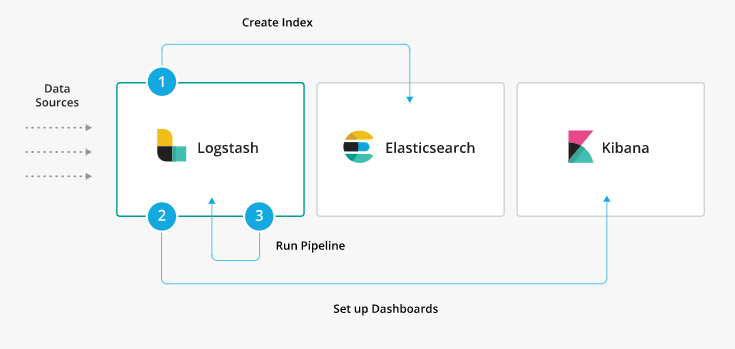
***Фигура 9.*** *Архитектура на Logstash – Схема за консумиране на данни от множество входни точки, обработване и записване в множество изходни точки.*

**ElasticSearch** [mark] ще бъде основната ни база за търсене на логове и метрики. Той представлява документ-базирана система за търсене, която е изградена върху Lucene engine[mark]. Към ElasticSearch можем да изпращаме заявки за търсене по име на поле, стойност на което и да е поле от документа, стойност на точно определено поле, и т.н. Също така е създаден с идеята за скалируемост и устойчивост на грешки, а това е един от основните приоритети на нашата система. Основната абстракция в Elastic са индексите. Индекс представлява колекция от еднородни документи, измежду които ще търсим данни.

**Kibana** [mark] представлява потребителски интерфейс върху ElasticSearch. Тя предоставя удобен модел за визуализиране на записаните документи. Носи със себе си възможност за рендериране на различни графики, топлинни карти и т.н. Основно се използва за бързо навигиране в големи обеми от данни.



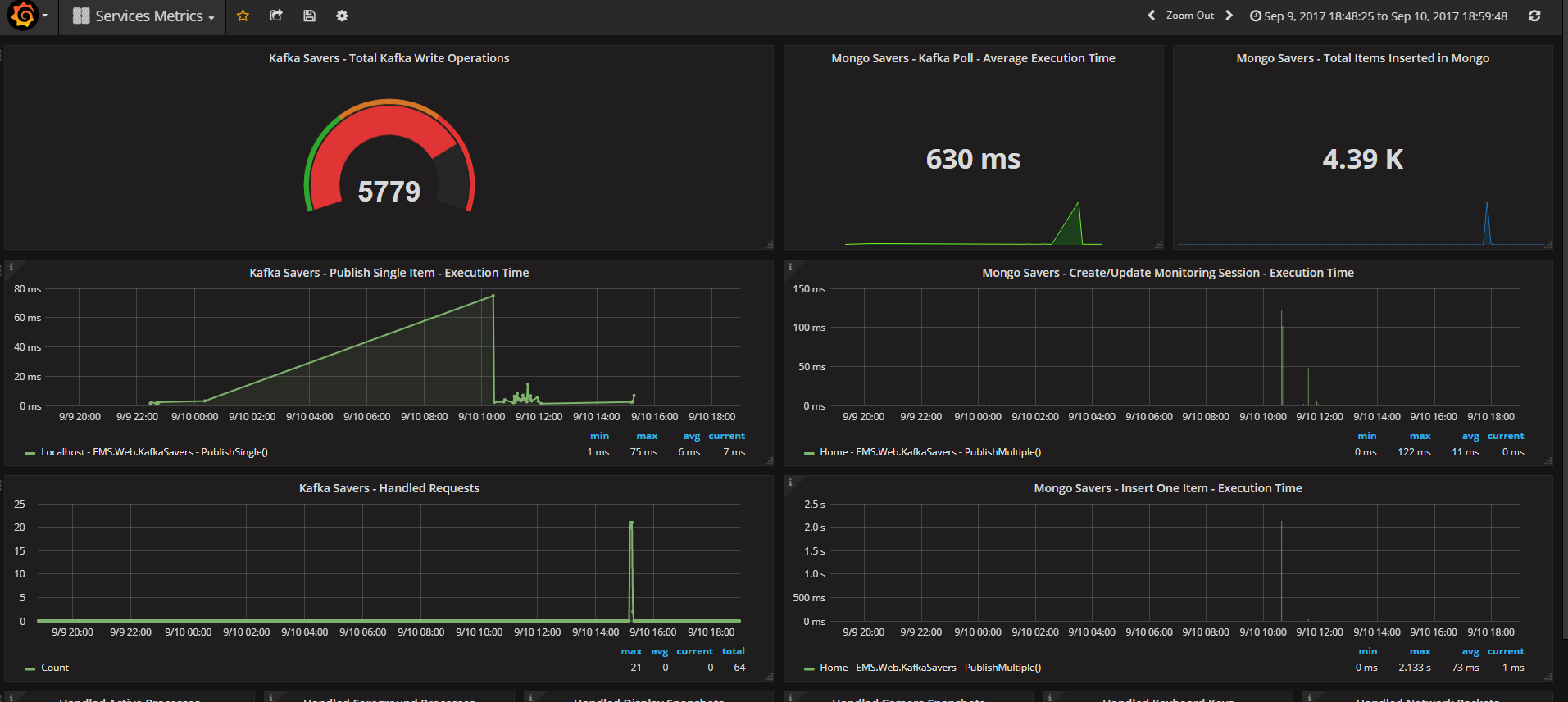
***Фигура 10****. Примерна страница в Kibanа, която визуализиране данните от даден индекс в ElasticSearch.*



***Фигура 11.*** *Поток на данните между отделните системи в ELK стека.*

### 2.4.2. Наблюдение на метрики от системите.

За следене на метрики, свързани с работата на системите и комуникацията между тях, можем да използваме вече инсталираната Kibana, но факт е, че нейният основен фокус, е визуализиране и търсене измежду големи обеми данни, а не рендериране на графики. Затова ще включим още един инструмент в нашата система, а именно Grafana [mark].

**

***Фигура 12****. Примерно табло с множество различни панели в Grafana.*

За началния етап на системата ни, ще следим метрики свързани с времето за обработка на дадена заявка от различните сървъри, времето за записване на документ в MongoDB, времето за записване на документ в Kafka, броя записани документи в MongoDB и Kafka, като може да разширим графиките да показват и ресурсите използвани от приложенията, като например процесорно време (CPU) и памет (RAM), за да може да реагираме, ако някоя машина е критично натоварена.

Като обобщение, имаме за цел да композираме надеждна система за следене поведението на студентите по време на различни занятия. Искаме да имаме добра видимост върху работата на отделните модули в системата и да можем да скалираме лесно при необходимост.

# Трета глава

## 3. Програмна реализация

Всеки опитен софтуерен инженер знае, че колкото повече редове код съдържа проекта, толкова по-труден става за поддръжка в бъдеще. Писането на гъвкав, лесно разширим и тестваем код е трудно постижима задача, а при постоянно гонене на срокове, става почти невъзможна.

За да минимизираме техническият дълг, който ще се натрупа в проекта ни с течение на времето, ще се фокусираме върху използване на възможно най-голям брой готови модули, библиотеки и системи, които имат силна подкрепа от обществото зад тях и изглеждат устойчиви в бъдеще.

Това ни дава възможността да се възползваме от човешки ресурси, външни за нашата организация, които да се грижат за еволюцията и поддръжката на кода, който ползваме.

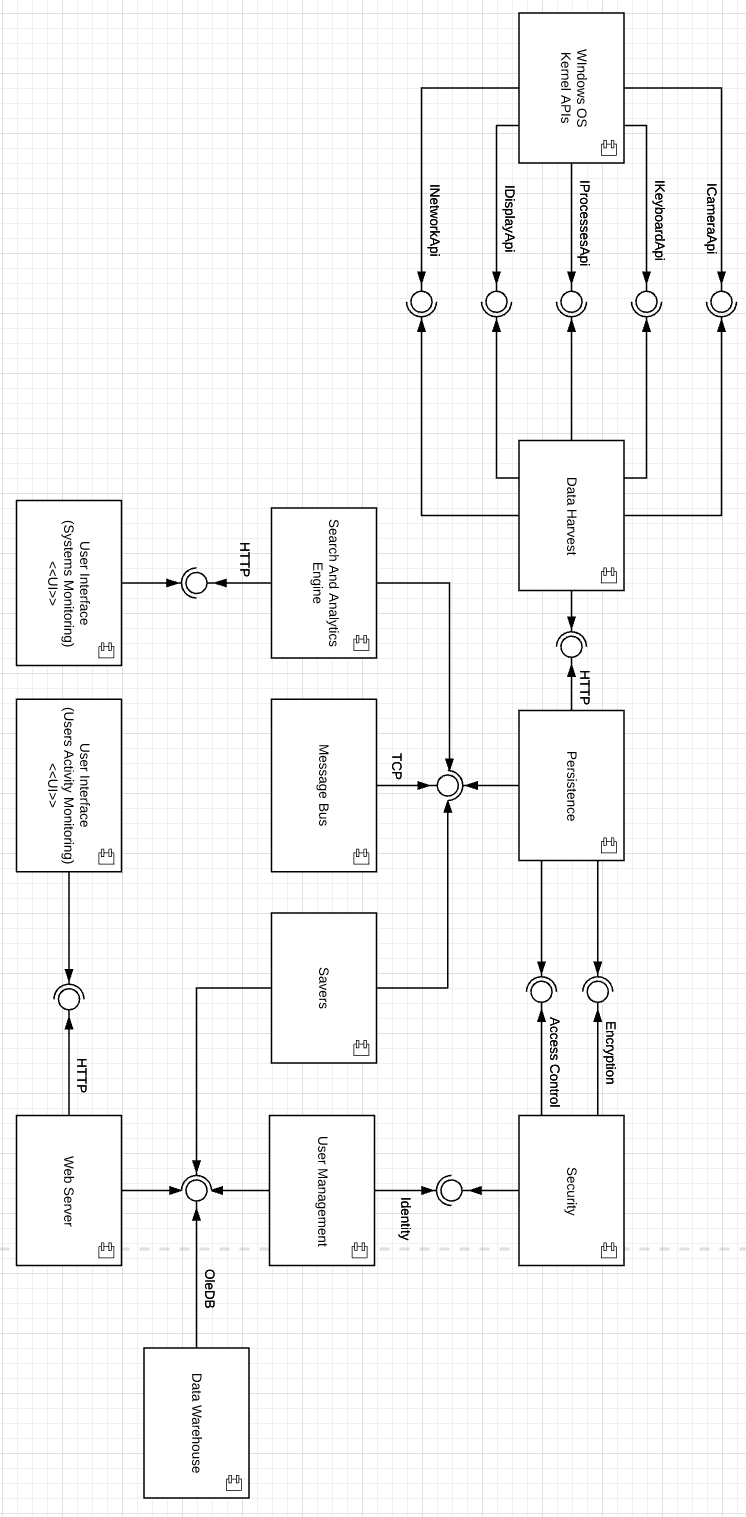
За нас остава само частта с интеграцията между отделните системи и разработка на основните функционалности, които са строго индивидуални за нашия продукт и няма от къде да си набавим готови.

Още преди създаване на проект в средата ни за разработка, трябва много добре да сме проучили сферата (domain) с която е свързан софтуера, който ще създаваме. Това ще ни позволи да проектираме системата с много повече внимание към детайлите и спецификите на конкретната сфера и ще намали до минимум нуждата от промяна или нагласяне на модулите спрямо изискванията, които не са били ясни в началото.

Като основни правила при писане на код, ще следваме принципа за „обръщане на контрола“ (Inversion-of-control), който гласи, че даден клас/обект/функция не трябва да се грижи за инициализиране на модулите от които зависи (dependencies). Това означава, че всички елементи, които се явяват зависимости на друг елемент, трябва да бъдат създадени и инициализирани извън неговите рамки. Той не трябва да се грижи за менажиране на живота на зависимостите, които има.

Друго основно правило, което ще следваме е, всяка зависимост да се консумира през интерфейс, а не през конкретен тип. Нивото на абстракция трябва да е такова, че да позволява лесна подмяна на използваните модули, лесно автоматизирано тестване и лесна поддръжка. Всички повтарящи се функционалности трябва да бъдат изнесени в базов клас, който да бъде наследен. Така ще избегнем повтаряем код и всеки път когато правим промяна, тя ще бъде само на едно място.

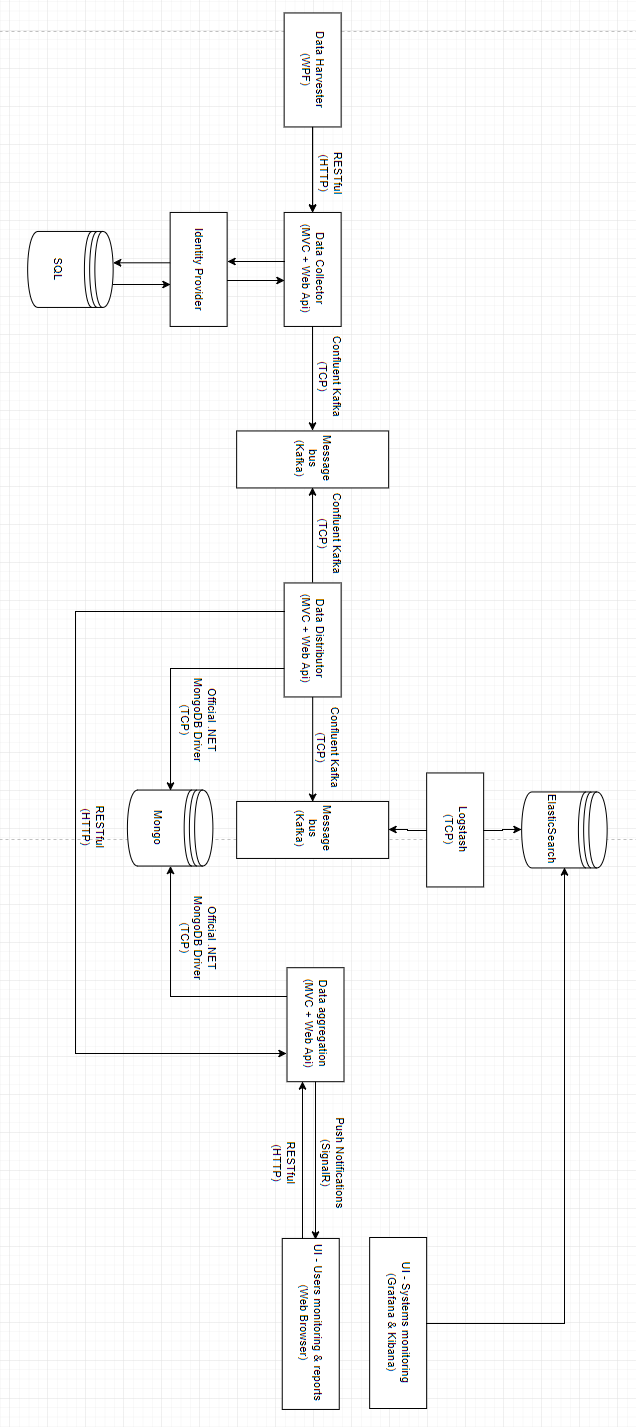
Това са много семпли и мощни правила, върху които ще се стремим да изградим качествена код-база.



Фигура 13. Диаграма на компонентите на цялата софтуерна система.

## 3.1 Софтуерна архитектура

Имплементацията ни ще бъде определена от предварително проектирана архитектура от високо ниво, която има за цел да предостави видимост върху софтуерната система като цяло. Това помага за по-лесното анализиране и набелязване на потенциални проблеми с производителността или пропуски в самата логика на приложението, преди то да бъде разработено.



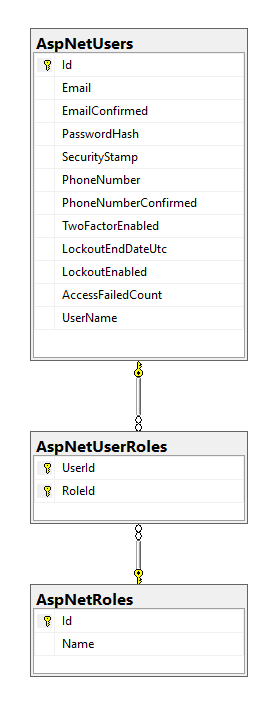
***Фигура 14.*** *Цялостна архитектура на софтуерната система*

## 3.2. Структура на основните модели (модели за трансфер на данни и домейн модели).

Основните ни домейн модели, ще бъдат свързани с потребителите на системата – администратори и студенти, както и с дейностите, които следим за студентите. Искаме потребителите ни да имат и роли, за да можем да ограничим достъпа на един регистриран студент до администраторските панели.

В SQL базата ще съхраняваме информацията за потребителите на системата и техните роли, а в MongoDB базата, ще съхраняваме информацията свързана с наблюдаваната дейност на студентите.

Примерна ER диаграма на един потребител в SQL ще изглежда по следния начин:



***Фигура 15.*** *ER диаграма на потребителския домейн модел.*

При регистрация, за потребителско име може да се ползва факултетния номер на студента или друг вид идентификационен символен низ, в зависимост от избраните конвенции.

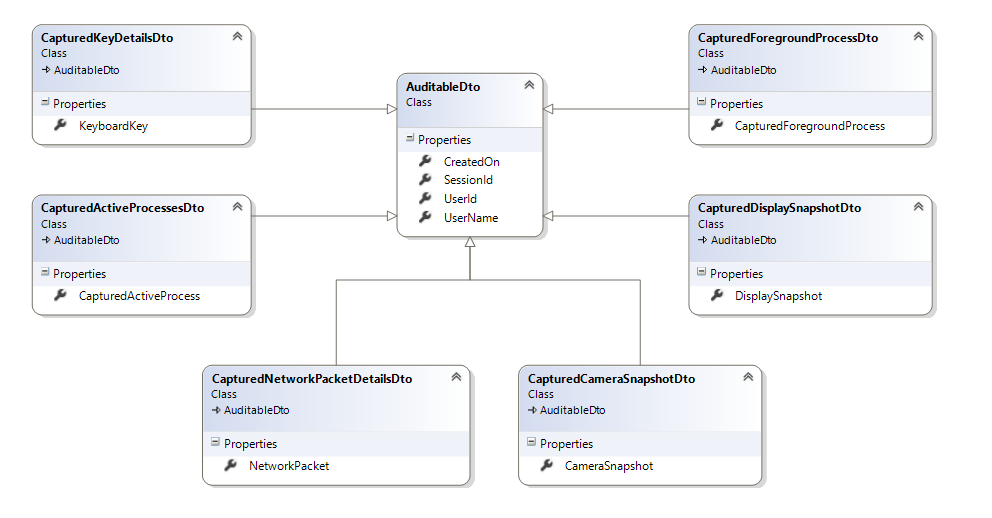
За целите на нашата система, този семпъл модел е достатъчен, защото единственото което ни интересува е да имаме начин да идентифицираме различните сесии от кой потребител са създадени и дали този потребител е валиден за нашата система.

Другият вид основен модел ще бъдат моделите за трансфер на данни, които ще съдържат четири базови полета служещи за идентифициране на потребителят, за който са събрани данни, сесията, която е стартирал (един потребител може да стартира много сесии), и датата на която са били събрани съответните данни.

Ще имаме един основен модел, който ще бъде наследен от останалите 6 модела, които ще указват типа на събраните данни:

* Активни процеси (CapturedActiveProcessesDto)
* Процеси на фокус (CapturedForegroundProcessDto)
* Снимка на дисплея (CapturedDisplaySnapshotDto)
* Снимка на камерата (CapturedCameraSnapshotDto)
* Натиснати клавиши (CapturedKeyboardKeysDto)
* Мрежови пакети [[19]](#B19) (CapturedNetworkPacketDto)

Примерна клас диаграма на моделите за трансфер на данни (DTOs) ще изглежда по следния начин:

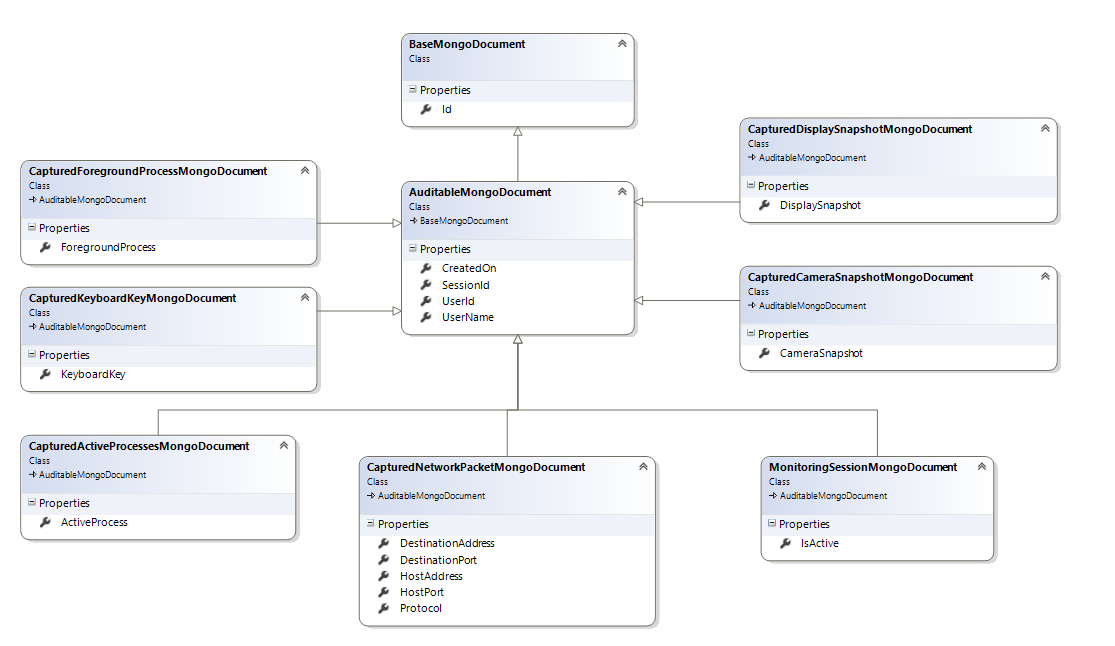


***Фигура 16.*** *Клас диаграма на основните модели за трансфер на данни*

Тези модели ще се различават много малко от основните домейн модели, в които ще съхраняваме събраните данни, по това че ще съдържат информацията в суров вид. Друг вариант, за да избегнем дублиране на моделите за трансфер на данни с домейн моделите е да пренесем повече отговорност на десктоп приложението, което ще събира данните и да го накараме да ги изпраща във формат, готов за съхранение.

Все пак е добра практика да имаме ясно разделение между моделите, които използваме за трансфер на данни и тези, които описват даден домейн модел, дори за текущия момент да изглеждат еднакви, контекстът е винаги от значение и може да се промени бързо, а това да ни донесе проблеми в бъдеще.

Домейн моделите свързани със прихванатата информация за потребителските дейности, които ще бъдат съхранявани в MongoDB, ще имат следната йерархия и структура:



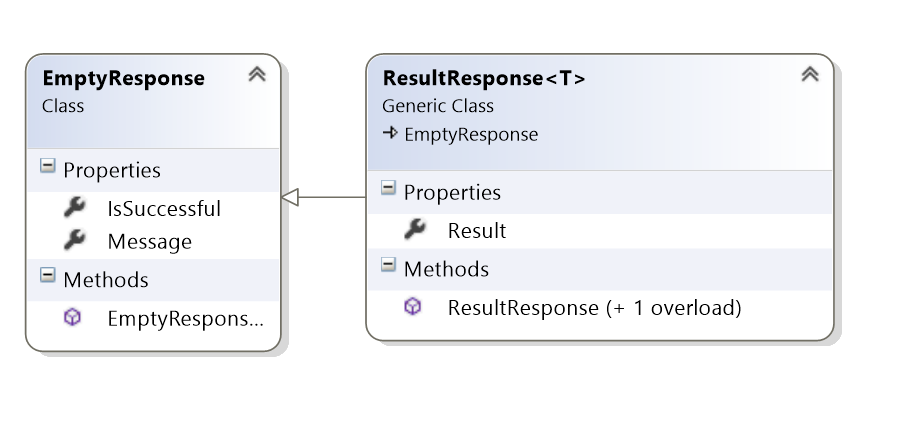
***Фигура 17.*** *Клас диаграма на MongoDB домейн моделите.*

Структурата на тези модели много прилича на тази на моделите за трансфер на данни, но се различава по това, че към основният базов модел има добавено още едно поле “Id”, което представлява уникален идентификатор за документа записан в MongoDB, както и по структурата на един прихванат мрежови пакет. В домейн моделът виждаме вече извлечена информацията от байт масива, който сме прихванали, а именно Адрес и Порт на домакина (от който е изпратен пакета), Адрес и Порт на получателя, вид на протокола [[18]](#B18) .

Снимките на дисплея и на камерата ще записваме като масив от байтове. Информацията за активните процеси ще записваме като комплексен обект от тип “SlimProcess”, който ще съдържа следните полета:

* Идентификационен номер на процеса (в операционната система)
* Име на процеса
* Име на машината домакин
* Заделена виртуална памет
* Заделена физическа памет
* Дата на стартиране на процеса
* Използвано процесорно време

За записване и изпращане на резултата при комуникация с отдалечени приложно-програмни интерфейси[[20]](#B20) (APIs) ще използваме два общи класа – празен отговор (EmptyResponse) и отговор с резултат (ResultResponse).



***Фигура 18.*** *Клас диаграма на моделите за комуникация с отдалечени приложно-програмни интерфейси.*

Празният отговор ще съдържа 2 полета:

* Булева стойност описваща дали заявката е обработена успешно (IsSuccess)
* Съобщение, обясняващо резултата от заявката (Message)

Отговорът с резултат ще съдържа 1 допълнително поле, което съдържа резултата от заявката:

* Резултат (Result)

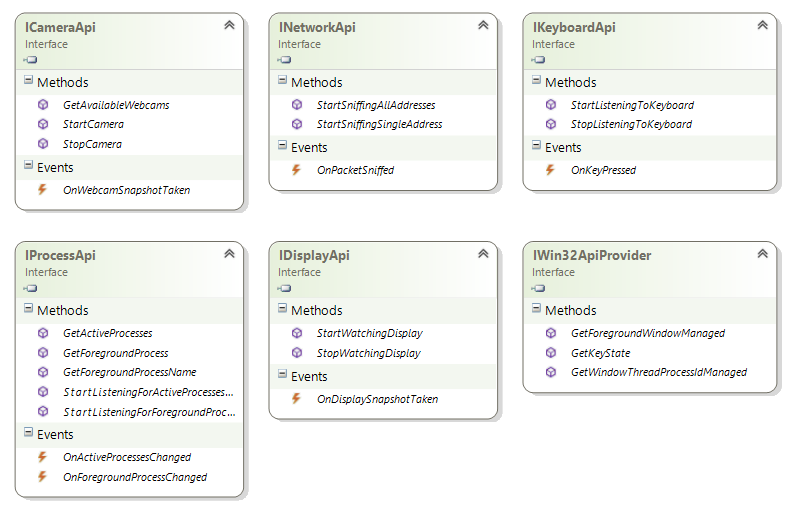
## 3.3. Структура на програмните интерфейси за събиране на данни от OS.

За достъп до приложно-програмния интерфейс на операционната система, ще извлечем само функционалностите, от които имаме нужда, и ще ги консумираме през строго дефинирани интерфейси.

Използването на интерфейси позволява на нашите класове да бъдат тотално разкачени от конкретни имплементационни детайли. Това ни предоставя възможност да тестваме лесно отделните модули на нашето приложение и ни дава предимство, ако се наложи в бъдеще да подменим основата на нашето приложение (версията на операционната система или цялата операционна система).

Интерфейсите ще следват един и същ шаблон. Ще предоставят методи за следене на системна информация[21], която ни интересува, както и събития, които ще бъдат извиквани, когато има нови данни за консумиране.

Десктоп приложението ни ще се абонира за тези събития и ще препраща всяко получено съобщение към front-end сървърите, като ще му добавя информация за потребителя, който генерира данните.



Фигура 19. Клас диаграма на изнесените интерфейси от приложно-програмния интерфейс на Windows операционната система.

Гореописаните интерфейси, ще бъдат имплементирани от конкретни класове, а именно:

* + CameraApi (implements ICameraApi)
  + DisplayApi (implements IDisplayApi)
  + KeyboardApi (implements IKeyboardApi)
  + NetworkApi (implements INetworkApi)
  + ProcessApi (implements IProcessApi)

Основните функции на “CameraApi” класът, ще бъдат следните:

* + Извличане на списък с всички налични камери, които са инсталирани на компютъра. Списъкът се състои от елементи (WebcamDetails), които съдържат името на камерата и уникален идентификационен номер.
  + Стартиране на определена камера по идентификационен номер и правене на снимка през определен период от време, който може да бъде конфигуриран.
  + Спиране на определена камера по идентификационен номер.
  + Извикване на събитие при записване на нова снимка.

Основните функции на “DisplayApi” класът, ще бъдат следните:

* + Стартиране наблюдението на дисплея и правене на снимка през определен интервал от време.
  + Спиране наблюдението на дисплея.
  + Извикване на събитие при записване на нова снимка.

Основните функции на “KeyboardApi” класът, ще бъдат следните:

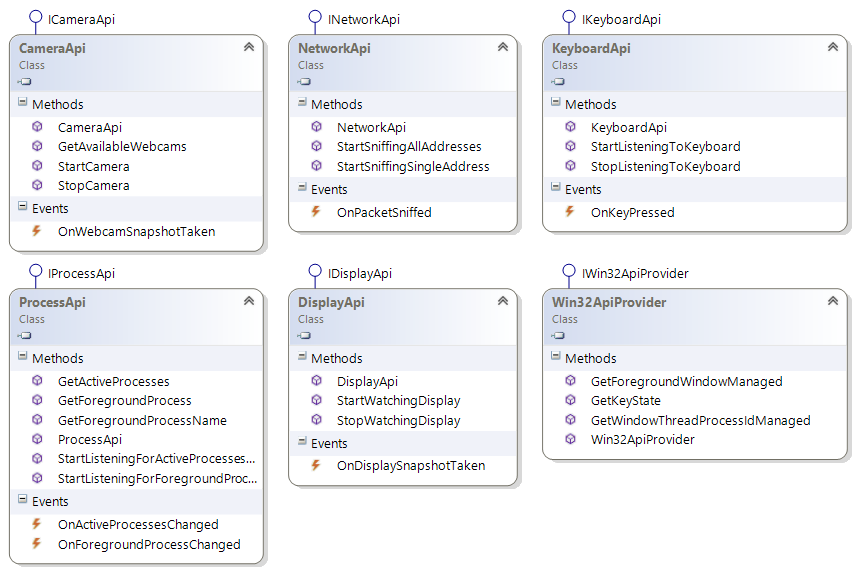
* + Стартиране слушането за натиснати клавиши и записване на всеки натиснат клавиш на клавиатурата в опашка (за да подсигурим редът).
  + Спиране слушането на клавиши.
  + Извикване на събитие при достигане на лимит от записани клавиши (ако изпращаме всеки натиснат клавиш до сървъра колектор, броят на заявките ще скочи експоненциално с броя следени потребители, затова избираме да буферираме определен брой натиснати клавиши и след като е пълен буфера, да ги изпратим наведнъж на сървъра).

Основните функции на “NetworkApi” класът, ще бъдат следните:

* + Стартиране слушането за мрежови пакети, изпращани и получавани от машината домакин.
  + Спиране слушането за мрежови пакети.
  + Извикване на събитие при достигане на лимит от записани мрежови пакети (поради същата причина, която обяснихме по-горе за KeyboardApi класът, а именно огромният брой заявки, които потенциално ще обсипят front-end сървърите)

Основните функции на “ProcessApi” класът, ще бъдат следните:

* + Извличане на информация за текущият процес на фокус (Foreground process)
  + Извличане на информация за активните процеси.
  + Извикване на събитие при стартиране на нов процес.
  + Извикване на събитие при промяна на процеса на фокус



***Фигура 20.*** *Клас диаграма на приложно-програмните интерфейси, които сме изградили върху функции на Windows операционната система.*

## 3.4. Структура на десктоп приложението за събиране на данни

Десктоп приложението, което ще играе роля на колектор на данни за дейността на потребителите, ще бъде реализирано изцяло на WPF[26] (Windows Presentation Foundation).

Ще се състои от 2 страници: една за вход (LoginPage.xaml) и една за регистрация (RegisterPage.xaml). Като над тях за основа ще стои една главна страница за навигация (MainNavigationWindow.xaml).

Потребителският интерфейс в WPF приложенията се описва чрез XAML[25] (Extensible Markup Language), който представлява декларативен маркъп език използван за иницализиране на структурирани стойности и обекти. Базиран е на XML и е създаден от Microsoft с цел опростяване създаването на потребителски интерфейси (UI) за .NET приложения.

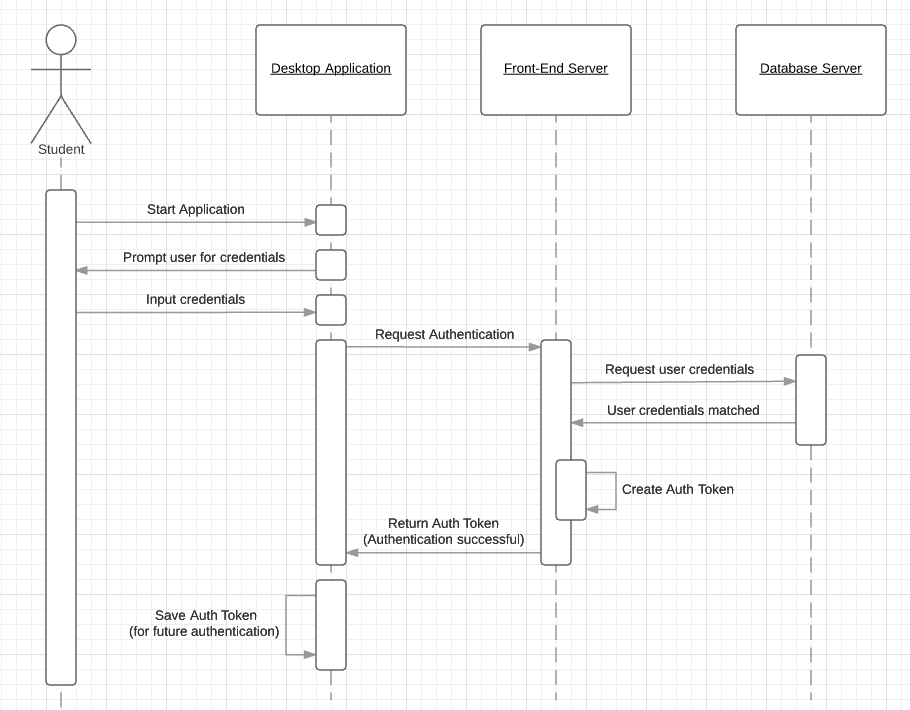
Езикът е представен през 2006 г. заедно с WPF в .NET 3.0. WPF е графична подсистема за рендериране на потребителски интерфейс в Windows-базирани приложения. Основните предимства на XAML са че разделя UI от бизнес логиката и че позволява едновременно да са работи върху логиката и интерфейса на една програма с различни инструменти (Visual Studio и Microsoft Expression Blend). Елементите на XAML са директно свързани със CLR[24] обекти, а атрибутите – със CLR свойства и събития върху тези обекти.

Потребителският интерфейс на страницата за вход ще бъде описан от следния XAML:



***Фигура 21.*** *XAML описание на страницата за вход в десктоп приложението.*

Страницата за вход ще се състои от 2 текстови полета за въвеждане на потребителско име и парола, и 2 бутона – един за вход и един за навигиране към страницата за регистрация, като полето за парола ще използва WPF контролата “PasswordBox”[27], която предоставя готова имплементация за сигурно въвеждане на чувствителна информация. Всеки символ от PasswordBox контролата е маскиран с предварително дефиниран символ, който в повечето случаи е звезда „\*“. Също така, паролата се съхранява в криптиран масив от байтове (“SecureString”[28]), с цел предпазване от хакерски атаки насочени към паметта на компютъра.



***Фигура 22****. Диаграма на последователностите на страницата за вход в системата (Login).*

Както виждаме от диаграмата на последователностите, след като потребителят въведе потребителско име и парола, информацията бива изпратена към сървъра за автентикация, който при успешно удостоверяване, ни връща токен за автентикация, който можем да запазим и използваме при всяка следваща заявка, докато не изтече срокът му на годност и трябва да се автентикираме наново.

Страницата за регистрация, наподобява тази за вход, но съдържа няколко допълнителни текстови полета за въвеждане на данни за студента, като имейл и факултетен номер, както и поле за потвърждение на паролата.

XAML описанието на потребителския интерфейс на тази страница е следното:

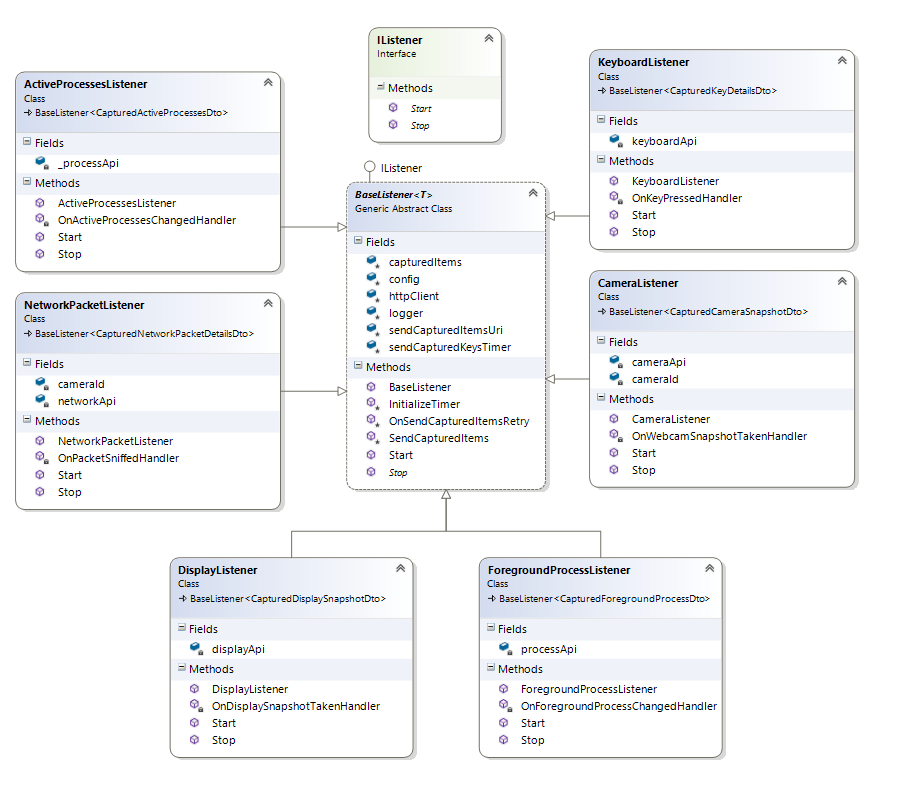


***Фигура 23.*** *XAML описание на страницата за регистрация.*

Диаграмата на последователностите за страницата на регистрация е почти същата като тази за вход, като премахнем частта с връщане на автентикационен токен.

Друга основна част от WPF приложението ще бъдат „слушателите“ за промени (Listeners), които ще използват приложно-програмните интерфейси и модули за комуникация с операционната система. Слушателите следят за конкретни събития. Всеки слушател си има определен тип събития, които регистрира и изпраща на front-end сървърите.

* + CameraListener слуша за снимки на камерата.
  + DisplayListener слуша за снимки на дисплея.
  + ActiveProcessesListener слуша за нови стартирани процеси.
  + ForegroundProcessesListener слуша за смяна на процеса на фокус.
  + KeyboardListener слуша за натиснати клавиши.
  + NetworkPacketsListener слуша за изпратени и получени мрежови пакети.



Фигура 24. Клас диаграма на слушателите.

Всички слушатели имплементират общ интерфейс IListener, който предоставя два метода – Start() и Stop(), и наследяват общ базов клас BaseListener, който се грижи за изпращане на прихванатите обекти към front-end сървърите и имплементиране на “Retry” стратегии[29] за повторно изпращане, ако първите заявки не са били успешни.

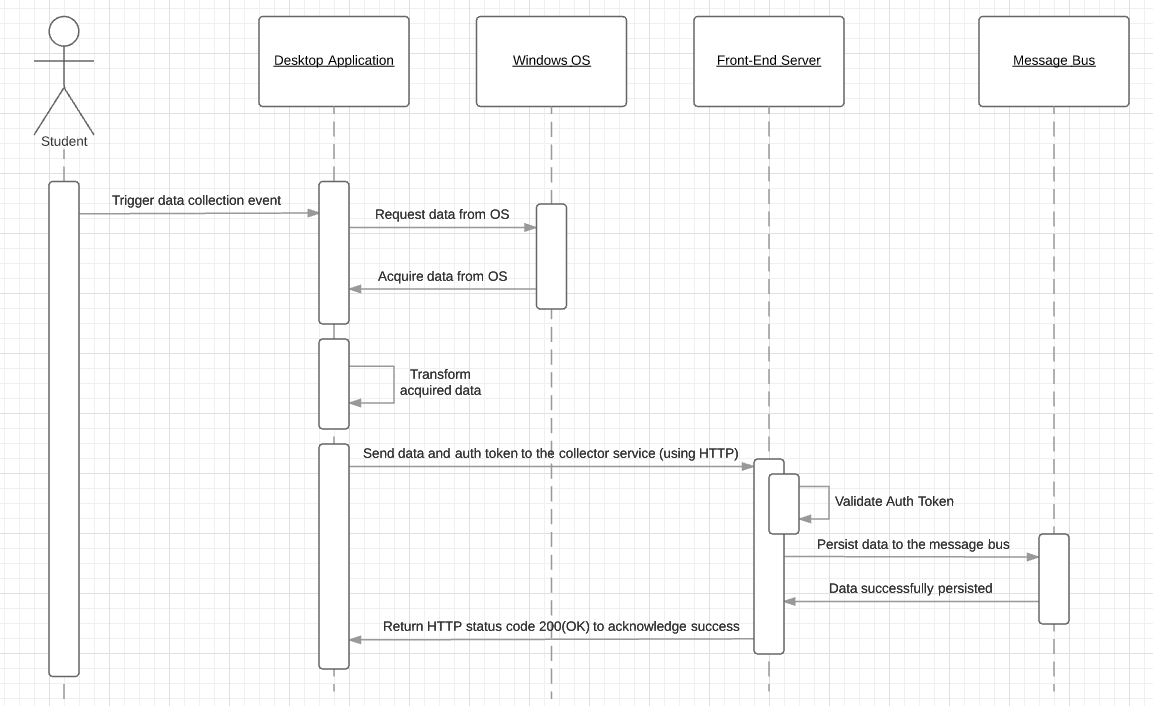
Ще използваме „Polly“[30] библиотеката, за устойчивост към грешки (fault-tolerance) при извикване на отдалечени приложно-програмни интерфейси. Ще имплементираме стандартна Wait-and-Retry (изчакай и повтори) стратегия за всички външни заявки, като интервала за изчакване и броя повторения ще може да бъде конфигуриран от **config.json** файл, който се зарежда при стартиране на десктоп приложението.

Примерна имплементация на „fault-tolerance policy“ с „Polly“:

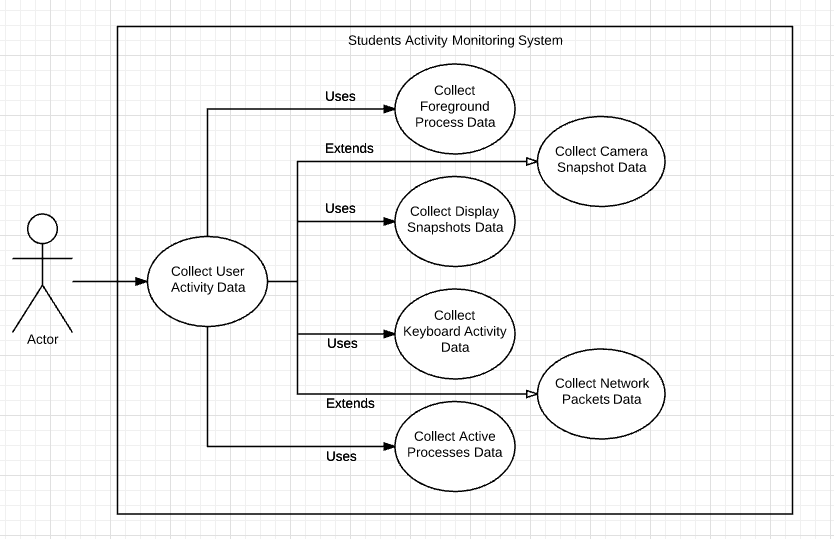


***Фигура 25.*** *Примерна „Retry“ стратегия с изчакване и повтаряне.*

Тази библиотека ни позволява да се справим успешно с частни случаи, като например: за момент връзка към даден сървър е загубена; интернетът е изключен или просто заявката не може да бъде обработена веднага и връща таймаут. Чрез изчакване и повтаряне на заявката гарантираме по-висок процент успеваемост.



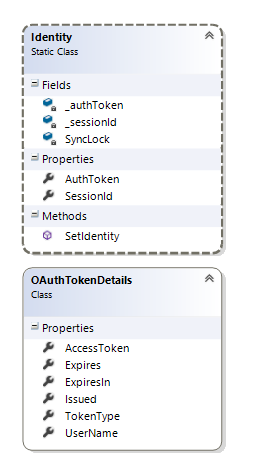
***Фигура 26.*** *Диаграма на последователностите при изпращане на данни относно потребителската активност*.



***Фигура 27.*** *Use-case диаграма на десктоп приложението.*

WPF приложението ще съдържа и статичен клас, който ще пази информация свързана с идентичността на потребителя, текущата сесия за наблюдение и токените за автентикация. Този клас ще има един основен метод – SetIdentity(), който ще се вика след успешен опит за вход в системата.

При всяка следваща заявка към front-end сървърите, от Identity класът ще се взима информация за автентикационният токен – “Identity.AuthToken.AccessToken” и ще се изпраща като HTTP Request Header[31].



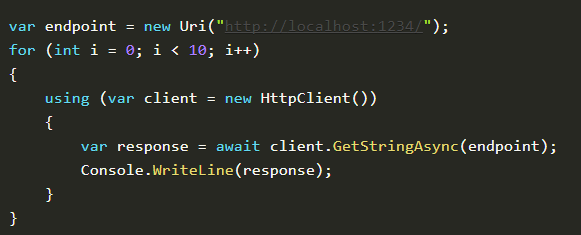
***Фигура 28.*** *Клас-диаграма на Identity класа и автентикационния токен.*

Тъй като комуникацията с отдалечени сървъри ще се извършва по HTTP протокол, трябва да изберем подходящ HTTP клиент за езикът C#. Имаме няколко опции за избор:

* + HttpWebRequest
  + WebClient
  + HttpClient (.NET 4.5+)
  + RestClient (Safe and Simple HttpClient wrapper)

Първите два клиента са от стари версии на .NET Framework и не са добре оптимизирани за изпращане на HTTP заявки, затова ги изключваме автоматично.

HttpClient класът е абстракция, която предоставя лесни за използване методи за комуникация с HTTP сървър изцяло асинхронно. Всичко това звучи добре, но има някой тънкости при работата с този клас, които трябва да знаем, за да не изпаднем в тежки проблеми с производителността[32] на нашите приложения. Наблюдавайки дефиницията на HttpClient класа, виждаме, че имплементира IDisposable интерфейса. Това на пръв поглед ни изглежда нормално и се надяваме, че в момента в който извикаме методът “Dispose()”, всички заделени ресурси свързани с клиента, ще бъдат зачистени.



***Фигура 29.*** *Примерна употреба на HttpClient класа със зачистване на ресурси.*

Но това не е съвсем така. Поради начина, по който е създаден да работи TCP/IP стека, установените връзки не се затварят мигновенно, за да могат да позволят на мрежовите пакети да пристигнат извън нормалния ред или да бъдат изпратени наново, след като връзката е затворена. Стандартното време за изчакване преди да бъде затворена една TCP/IP връзка е 4 минути, като може да бъде променено от настройките[33] на операционната система.

За да избегнем изцеждане на портовете на физическата ни машина при натоварено състояние на приложението, трябва просто да държим една единствена инстанция на HttpClient-а по време на целия живот на нашето приложение. Това ни гарантира, че при всяка нова HTTP заявка, ще се преизползват вече отворени връзки и няма да изтощаваме поголовно ресурсите си.

За целта ще използваме RestClient класа, който е семпла имплементация на HTTP клиент, използваща .NET HttpClient класа по правилен начин.

## 3.5. Структура на front-end сървъра за получване на данни

Front-end сървърът за получаване на данни свързани с активността на студентите, ще бъде имплементиран използвайки ASP.NET Web API v2.0. Web API представлява прост уеб сървър, който по подразбиране поддържа работа с JSON и XML данни, като използва вградени сериализатори за трансформация на данни получени в суров вид, към C# обекти и обратно.

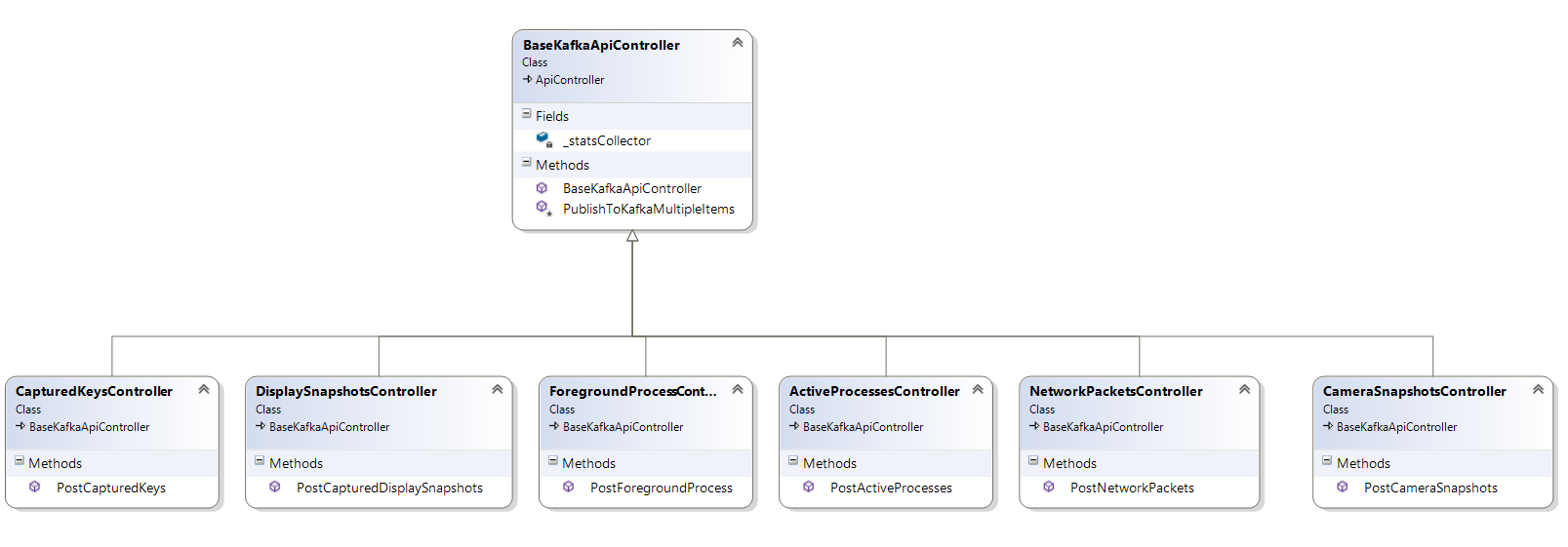
Също така, той ще се грижи за удостоверяване на потребителските заявки, а именно ще проверява дали даден потребител е регистриран в системата и има право да записва информация. Това се осъществява благодарение на ASP.NET Identity 2.0 системата, която се грижи за менажиране на потребителите и техните сесии.

Основната работна единица на Web API е контролерът. Той е обикновен C# клас, чиито методи играят ролята на адреси, които могат да прихващат заявки, да ги обработват и да връщат резултат. Резултатите могат да бъдат файлове в различни формати (JSON, XML, BSON), прости символни низове, числа, изображения, и т.н.



Фигура 31. Примерен Web API контролер.

Използвайки Identity системата, слагайки един единствен атрибут върху метод или контролер, който



***Фигура 30.*** *Клас-диаграма на front-end server контролерите.*

## 3.6. Структура на приложението за записване и обработка на получените данни

## 3.7. Структура на уеб сървъра за визуализиране на получените данни в реално време

# Четвърта глава

## 

## 4. Указания за ползване

4.1 Основни сценарии за използване

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

# Пета глава

## 5. Експериментални данни

## 5.1 Описание на експеримента и тестовата установка

# Заключение

# Източници

[1] Academic Integrity, Number of university students who admitted cheating.

<http://www.academicintegrity.org/icai/integrity-3.php>

[2] Kessler International, Percentage of students cheating using mobile devices.

<http://www.cleveland.com/metro/index.ssf/2017/02/cheating_in_college_has_become.html>

[3] Global Web Index, Average count of owned mobile devices.

<http://blog.globalwebindex.net/chart-of-the-day/digital-consumers-own-3-64-connected-devices/>

[4] JSON Web Token, Introduction and Anatomy.

<https://jwt.io/introduction/>

[5] MongoDB Licensing.

<https://www.mongodb.com/community/licensing>

[6] Apache Licensing (used by Apache Kafka, Grafana, ElasticSearch, Kibana, Logstash etc)

<https://www.apache.org/licenses/LICENSE-2.0>

[7] Microsoft, Official .NET open-source repository

<https://github.com/dotnet>

[8] Confluent, Official .NET driver for Apache Kafka

<https://github.com/confluentinc/confluent-kafka-dotnet>

[9] MongoDB, Official .NET driver for MongoDB

<https://github.com/mongodb/mongo-csharp-driver>

[10] LinkedIn, Benchmarking Apache Kafka

<https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>

[11] Logstash, Official Website

<https://www.elastic.co/products/logstash>

[12] Logstash, Official Documentation

<https://www.elastic.co/guide/en/logstash/current/index.html>

[13] ElasticSearch, Official open-source repository

<https://github.com/elastic/logstash>

[14] ElasticSearch, Official Website

<https://www.elastic.co/products/elasticsearch>

[15] Grafana, Offical documentation

<http://docs.grafana.org/>

[16] Grafana, Official open-source repository

<https://github.com/grafana/grafana>

[17] Wikipedia, List of network protocols

<https://en.wikipedia.org/wiki/Lists_of_network_protocols>

[18] Wikipedia, List of protocol numbers

https://en.wikipedia.org/wiki/List\_of\_IP\_protocol\_numbers

[19] Wikipedia, Network packet structure

<https://en.wikipedia.org/wiki/Network_packet>

[20] Wikipedia, Application Programming Interface

<https://en.wikipedia.org/wiki/Application_programming_interface>

[21] MSDN, System Diagnostics, Process class

<https://msdn.microsoft.com/en-us/library/system.diagnostics.process(v=vs.110).aspx>

[22] MSDN, Keyboard Input Functions, Get async key state

<https://msdn.microsoft.com/en-us/library/windows/desktop/ms646293(v=vs.85).aspx>

[23] MSDN, Window Functions, Get window thread process id

<https://msdn.microsoft.com/en-us/library/windows/desktop/ms633522(v=vs.85).aspx>

[24] MSDN, Common Language Runtime (CLR)

<https://docs.microsoft.com/en-us/dotnet/standard/clr>

[25] Wikipedia, XAML

<https://bg.wikipedia.org/wiki/XAML>

[26] MSDN, Windows Presentation Foundation

<https://docs.microsoft.com/en-us/dotnet/framework/wpf/>

[27] MSDN, System Windows, PasswordBox

<https://msdn.microsoft.com/en-us/library/system.windows.controls.passwordbox(v=vs.110).aspx>

[28] MSDN, System Security, Secure string

<https://msdn.microsoft.com/en-us/library/system.security.securestring(v=vs.110).aspx>

[29] Polly, .NET resilience and transient fault-handling library

<http://www.thepollyproject.org/>

[30] Polly, Github repository

<https://github.com/App-vNext/Polly>

[31] Wikipedia, HTTP Headers

<https://en.wikipedia.org/wiki/List_of_HTTP_header_fields>

[32] Nimaara, Beware of the .NET HttpClient

<http://www.nimaara.com/2016/11/01/beware-of-the-net-httpclient/>

[33] MSDN, Tuning TCP/IP for Performance

<https://support.microsoft.com/en-us/help/93444/tuning-tcp-ip-for-performance>

# 

# 

# 

# Приложение 1 - речник

Front-end server – сървър, който стои на фронта на една софтуерна система и се грижи за първична обработка на заявки. Най-често пред системата стои ферма от front-end сървъри, които пренасочват заявките към поточна линия за съобщения (message bus), които в последствие биват обработени от останалата част от системата. Идеята е да се изгради устойчива архитектура, която да може да поема огромни обеми заявки, без това да влияе на всички останали модули (например да забавя работата на базата за съхранение на данни).

Framework – работна рамка; съвкупност от базови библиотеки и функции, които са често използвани от всяко едно приложение и улесняват работата на софтуерния разработчик, като му предоставят вече разработени и тествани модули или шаблони.

TCO (Total-cost of ownership) – тотална цена за притежание; сумата, която ще ни коства поддръжката и работата на системата.

# Приложение 2 - изходен код