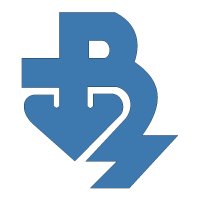
ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - ВАРНА

ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ

Катедра „Компютърни науки и технологии“

ДИПЛОМНА РАБОТА

за придобиване

на ОКС „Бакалавър”

**Тема:**

„Разработване на обучителни системи на базата на микросервисна архитектура“

**Изготвил:** Георги Карло Паскуали

**Специалност:** Софтуерни и интернет технологии

**Факултетен номер:** 61562152

ТУ-Варна Ръководител: ас. Ивайло Пенев

2019 г.

Съдържание

[**I Глава:** Въведение и цел на проекта 3](#_Toc11631934)

[**II Глава:** Микросервисна архитектура и използвани технологии 4](#_Toc11631935)

[1. Какво представлява архитектурата от „микросервиси“? 5](#_Toc11631936)

[2. Защо микроуслугите са подходящи за „облачните“ хостинг услуги? 7](#_Toc11631937)

[3. Какво е Docker и каква цел има той в системите от микро услуги. 9](#_Toc11631938)

[4. Docker терминология 10](#_Toc11631939)

[5. Kubernetes и как той взаимодейства с Doker 12](#_Toc11631940)

[6. Angular: Изграждане на уеб клиент за връзка с системата от микро услуги. 13](#_Toc11631941)

[7. NodeJS: Перфектното средство за бърза и „лека“ микро услуга. 13](#_Toc11631942)

[8. Dotnet Core: Стремежът на Microsoft да се наложат на „облачните“ пространства. 14](#_Toc11631943)

[9. PostgreSQL 15](#_Toc11631944)

[10. MongoDB: Добро решение за архитектура от микроуслуги 15](#_Toc11631945)

[**III Глава:** Проектиране на системата 16](#_Toc11631946)

[1. Изграждане на софтуерно задание 16](#_Toc11631947)

[2. Планиране на реализацията на проекта: Как проектирането на система от микроуслуги се различава от монолитната архитектура? 19](#_Toc11631948)

[3. Използване на услуга вход към програмния интерфейс: API Gateway Pattern 21](#_Toc11631949)

[4. Бази данни и модели на съхранение при системи от микросервиси 22](#_Toc11631950)

[**IV Глава:** Софтуерна Реализация 23](#_Toc11631951)

[1. Как се изолират отделните контейнери от „микросервиси“ използвайки Docker 24](#_Toc11631952)

[2. Създаване на API Gateway микроуслуга и как тя е свързана с останалите компоненти на сървърната архитектура 24](#_Toc11631953)

[Създаване на “Dockerfile” и настройка на изолираната среда. 24](#_Toc11631954)

[Конфигуриране на микроуслугата. 25](#_Toc11631955)

[Създаване на скрипт за стартиране на сървъра 28](#_Toc11631956)

[3. Създаване на микросервис за съхраняване на списъци: Topics API 30](#_Toc11631957)

[Връзка с базата данни 30](#_Toc11631958)

[Контролери и обработване на заявки 31](#_Toc11631959)

[4. Как се съхраняват статиите и парчетата код, които се демонстрират на потребителите (algorithms service)? 32](#_Toc11631960)

[5. Услуга за интерпретиране на примерен код 32](#_Toc11631961)

[6. Как е съединена системата от микроуслуги? 35](#_Toc11631962)

[Всеки отделен „service” има определена роля. 38](#_Toc11631963)

[7. Потребителски интерфейс и връзка с архитектурата от микроуслуги 38](#_Toc11631964)

[V Глава: Тестване на системата от микро услуги. 42](#_Toc11631965)

[1. Използване на системата за обучение 42](#_Toc11631966)

[2. Как пишем тестове за малки сървъри 42](#_Toc11631967)

[VI Глава: Заключение и бъдещо развитие 42](#_Toc11631968)

[1. Защо е лесно и безопасно да разширяваме системата ни от микроуслуги? 42](#_Toc11631969)

[2. Използвана литература 42](#_Toc11631970)

[Приложение – програмен код 43](#_Toc11631971)

# **I Глава:** Въведение и цел на проекта

В дипломната работа е представена архитектура за създаване на WEB базирани приложения, която представлява съвкупност от множество сървъри, изпълняващи само една цел. Тези малки компоненти се наричат „микросервиси“ или микроуслуги. Тази парадигма за разработка на софтуер е подходяща за създаването на големи по размер проекти [1]. Този подход за структуриране на една софтуерна разработка е значително нов и все повече навлиза сред софтуерните гиганти на пазара. Нетфликс, Гугъл, Амазон, Туитър , Пейпал и много други вече са превърнали техните монолитни приложения, в системи от микросервиси. Главната причина за тази реформа е, че тази нова архитектура е по-удачна да просъществува на „cloud“ инфраструктурата (хостинг с наето време или по заявка).

Програмата, която представлява система за обучение има за цел да помогне на начинаещи и професионалисти в сферата на компютърните технологии. Трябва всеки потребител да има възможността лесно да разглежда голямо количество от данни, като лесно се ориентира през тях. Също така, системата ще дава възможност да се изпълняват различни примерни програмни фрагменти, за да може по лесно да се илюстрират дадени концепции и похвати.

Архитектурата, която се използва е подходяща за такъв тип системи, защото тя позволява лесно да се разширява функционалността на приложението, без рискове за вече съществуващите модули. Друго предимство е, че когато изпълняваме код, написан от потребителя на нашият сървър, ако се случи нещо на контейнера-интерпретатор, останалата част на системата остава работеща.

# **II Глава:** Микросервисна архитектура и използвани технологии

## Какво представлява архитектурата от „микросервиси“?

Системата от микросервиси представлява стил за структуриране на приложение като набор от самостоятелни софтуерни приложения изпълняващи различни услуги [1, 4]:. Най-големите предимства на този подход са:

* Отделните услуги се поддържат по лесно и могат да бъдат тествани поотделно.
* Работата на различните контейнери е напълно самостоятелна.
* Всяко различно парче от системата може да бъде качено на хостинг само по себе си.
* По-лесно се оптимизира работата на големи приложения

Микросервисната архитектура позволява поддръжка в реално време на големи, комплексни системи. Също така позволява на една фирма да използва множество различни технологии за разработката на една система. Това е възможно, защото всеки неин компонент е изолиран от останалите (фиг.1):



Фиг.1. Пример за система от микроуслуги

Системата от микросервиси има много предимства, но също така въвежда редица проблеми, които трябва да бъдат решавани. Преди да се започне работа по проект, използвайки тази методология е добре да се реши дали това си заслужава. Ако приложението не е планирано за голяма група от хора, или не е много сложно, няма смисъл да отделяме време да планираме, изолираме и инсталираме отделни изолирани контейнери за всяка услуга. При разработката на микросервисна архитектура се крият рискове:

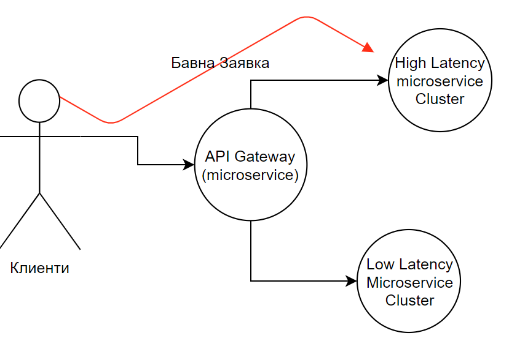
* Въвежда се допълнителна сложност по време на разработка.
* Нарастващият брой на микро услугите може да доведе до информационни бариери.
* Когато системата се разрасне може да стане трудно проследяването на проблеми.
* Трябва да се разработва и комуникацията между различните услуги.

## Защо микроуслугите са подходящи за „облачните“ хостинг услуги?

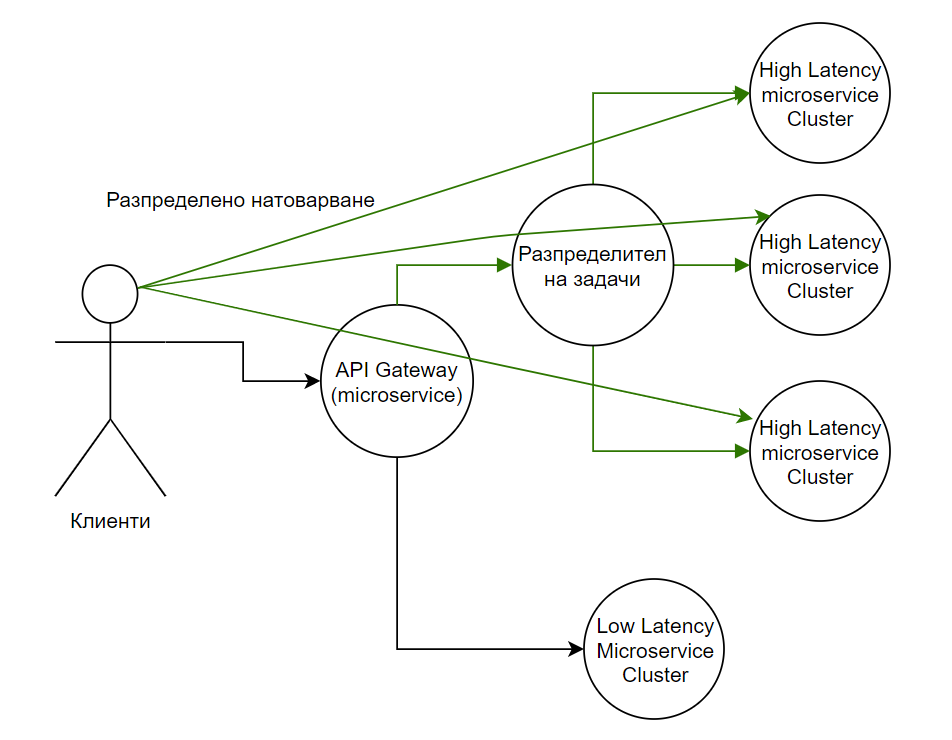
Когато могат да се разпределят на воля ресурсите на използвания хостинг, е добре да има възможност да се използват точно толкова ресурси, от колкото има нужда. [1, 2, 3]

Пример за следния проблем може да се наблюдава на диаграмата, представена на фиг. 1, където имаме забавяне, поради голямо натоварване:

Фиг. 1. Бавна заявка при голямо натоварване



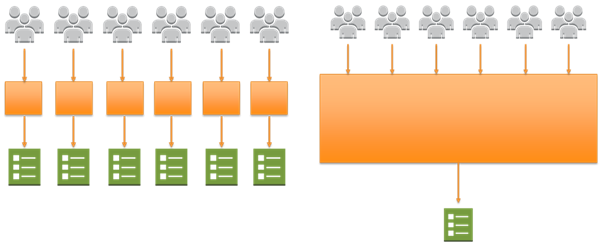
Ако се използва обикновена монолитна архитектура, в този случай щеше да се наложи да се правят промени по хардуера на сървъра или да се стартира нов, на който да се копира **цялото** приложение, за да може да се облекчи трафика. Това коства много ненужни ресурси. В архитектурата от микро услуги могат да се присвоят повече клъстери само на проблемния контейнер, което намалява драстично разходите по балансирането на трафика(фиг.2):



Фиг.2. Балансиране на трафика при микросервиси (Load Balancing)

В този случай се заделя повече хардуерен ресурс, но само за модулът, който изпитва забавяния.

Друго предимство на тази архитектура е гъвкавостта. Когато модулите не зависят един от друг, това означава, че множество екипи могат да работят по различни системи без да влияят на работата на другите значително по-лесно. За това този начин на разработка се предпочита от екипи, следващи методологията AGILE (Фиг.3).



Фиг.3. Разлика между монолитна(дясно) и микросервисна(ляво) архитектура

Заради разделението на услугите в различни контейнери екипите, работещи към една обща кауза могат да избират коренно различни технологии, библиотеки и инструменти за да изпълнят желаната задача. Качеството на малките софтуерни модули може да бъде контролирано по-лесно, когато те служат за една единствена цел. Също така се получава пълен контрол върху здравето на различните услуги, като могат да се намалят щетите от неработещ „микросервис“ само в рамките на неговата цел.

## Какво е Docker и каква цел има той в системите от микро услуги.

За да бъде възможно създаването на изолирани среди, изпълняващи микро услуги, задължително трябва да се използва софтуер, който изолира ресурсите, използвани в различните „микросервиси“. Най-популярното средство за постигане на тази цел са контейнерите на Docker [4, 5]. За разлика от хипервайзoрната виртуализация, където една или повече машини работят виртуално на физически хардуер през междинен слой, контейнерите се изпълняват на потребителско ниво върху ядрото на операционната система[1]. Контейнеризационната технология позволява множество изолирани потребителски пространства, да работят на един единствен хост.

Docker може да се изпълнява на всички 64 битови Линукс ядра и от скоро на windows- базирани системи. (Linux, OS X, windows 10…). Технологията е много бърза и изисква малко хардуерен ресурс, което означава, че може да бъде използвана на сървърни машини, лаптопи, телефони и дори микроконтролери.

Docker използва следните технологични концепции за да постигне целта на контейнеризация:

* Именовани пространства от ядрото на Линукс, които предоставят изолация за файлови системи, процеси и мрежи.
* Всеки контейнер си има изолация на файлова система .
* Всеки контейнер си има дърво на процесите, започващо от 1.
* Всеки контейнер си има виртуален мрежов интерфейс.
* Изолация на ресурси и групиране: ресурси като изчислителна мощност и памет се заделят индивидуално за всеки контейнер, използвайки така наречените cgroups, или Control Groups, което е функционалност на линукс ядрото. (за windows имат други наименования).

## Docker терминология

**Container(контейнер)-** Работеща инстанция, която изолира даден софтуер и съдържа само програмите, нужни за неговото изпълнение. Контейнерът може да предоставя различни портове и дисково пространство, със които да взаимодейства с външния свят( всичко извън изолираната среда). Контейнерите се създават от **шаблони**, които описват какво ще съдържа даден контейнер и какви дейности ще извършва. Контейнерите могат да се спират и пускат много бързо. [2]

**Image(шаблон)-** Шаблоните в Docker се използват за описание на една изолирана работна среда и действията, които се извършват в нея. Когато се създава шаблон, може да отнеме време за да се изтеглят нужните ресурси за работата на контейнера, но контейнерите могат да се изпълнят от шаблони много бързо. [2]

**Port-** Това са TCP/UDP портове, които могат да се предоставят на операционната система хост, или на други контейнери. Например, може да се конфигурира един контейнер да даде достъп на друг до определен негов порт.

**Volume(споделено дисково пространство)-** От последните версии на Docker, можем да създаваме споделени дискови пространства, които да се ползват от повече от един контейнер. Те са създадени за да пазят данни, които да не зависят от жизнения цикъл на контейнера. Подобно е на децентрализирана система за контрол на версии. [2]

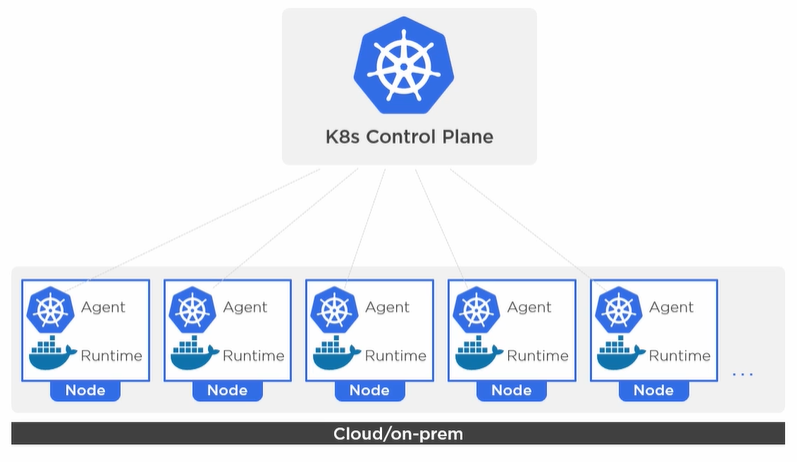
**Docker Hub-** Той представлява интернет регистър, който съдържа много готови шаблони, създадени от производителите на различни софтуери. Поради голямата популярност на Docker, там може да се намерят шаблони за почети всяка развойна среда и всеки език за програмиране. Също така там всеки може да си запази шаблон, който си е създал и да може да го достъпи от всяко устройство с достъп до интернет. [2]

**Network-** Docker предоставя възможност да се създават виртуални мрежи на една хост машина, до които имат достъп само дадени контейнери, на които са им зададени права.

## Kubernetes и как той взаимодейства с Doker

Kubernetes е безплатна система за създаване, скалиране, настройване и маршрутизиране на контейнери. Той е създаден от Гугъл и е разпространен по множество хостинг услуги Docker съдържа вграден Kubernetes софтуер, който може да бъде използван за оркестриране на Docker контейнери.

Когато се използват двете технологии за разработване и инсталиране на контейнери, Docker се използва за имплементиране на функционалността и описване на средата на всеки контейнер. Kubernetes менажира вече създадените изолирани среди. Той дава възможност да се контролира стабилността и скалирането на контейнерите, които се ползват за системата от микроуслуги. Пример за хостинг, който използва Kubernetes и Docker (Фиг.4):



фиг. 1 Хостинг, използващ Docker и Kubernetes

Kubernetes автоматично извършва разпределяне на ресурсите в рамките на един хостинг за да раздаде равномерно ресурсите по различните микроуслуги. Също така ако една микроуслуга спре да работи, автоматично я рестартира и прави всичко възможно за да я възстанови.

## Angular: Изграждане на уеб клиент за връзка с системата от микро услуги.

Angular е платформа за създаване на WEB приложения, използвайки typescript и HTML. [4].

## NodeJS: Перфектното средство за бърза и „лека“ микро услуга.

Node.js представлява среда за изпълнение на JavaScript. Той е създаден от създателите на JavaScript и позволява изпълнението на код, не само в браузъра, но и на Уиндоус и UNIX базирани операционни системи. Средата за изпълнение се нарича V8. Тя взима скриптовия код и го превръща в машинен код, който може да се изпълнява без да трябва да ползваме интерпретатор.

Входните и изходните операции при Node.js могат да бъдат изпълнени паралелно, като се използва модел, базиран на събития. Това означава, че когато се извършват операции като четене от диска, обработване на http заявки и др., не блокираме изпълнението на главната нишка. Това качество на Node.js го прави идеален за създаването на сървърни приложения.

## Dotnet Core: Стремежът на Microsoft да се наложат на „облачните“ пространства.

ASP.NET Core е новият framework на Майкрософт. Той представлява напълно пренаписан framework и не е продължение на стария .NET framework. Той често се използва за разработването на WEB приложения на езика C#.

Dotnet Core е подходящ за създаване на микросервиси, поради следните причини:

* Може да работи на Линукс и Уиндоус Базирани контейнери.
* Платформата използва малко хардуерни ресурси.
* Съдържа много малко начални модули и могат да се инсталират
* Майкрософт поддържа Docker шаблони за изпълнение на .NET Core софтуерен проект.

## PostgreSQL

PostgreSQL представлява система за управление на бази от данни (СУБД). Първоначално тя е създадена да работи на UNIX базирани платформи, но в днешно време се поддържа от всички популярни операционни системи. Лицензът за PostgreSQL е безплатен и може свободно да се ползва и дистрибутира под всякаква форма. Тази СУБД притежава много функционалности, които се съдържат и в други платени системи [11]. В нея могат да се добавят функции, създадени от потребителите, написани на езици като C, C++, Java и други

## MongoDB: Добро решение за архитектура от микроуслуги

В микросервисните архитектури, приложението се разделя на малки, самостоятелни услуги. Те обикновено се целят към специфична цел или функционалност, която пряко се влияе от бизнес зависимостите на проекта. Това помага на екипите от разработчици, да се фокусират върху конкретна цел. Всяка услуга се програмира, тества и инсталира на сървъра, без да зависи от останалите. Най-използваната практика за съхраняване на данни в системи от микросервиси е, всяка услуга да се свързва със собствена база от данни. Причините са:

* Ако има споделена база от данни между различни сървъри, в случай в който базата от данни се повреди, всички зависими услуги спират да функционират
* Всеки отделен екип, който работи по собствената си база от данни, може да прави промени по нея без да се притеснява, че ще пречи на останалите екипи,

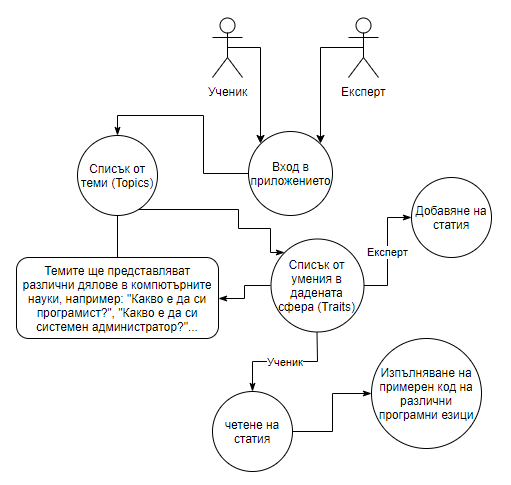
MongoDB e нерелационна база от данни, която много често се използва за разработването на микросервисни архитектури. Причините за това са нейния гъвкав модел от данни, повторяемост и скалируемост. Динамичната схема от данни, използвана в MongoDB, е идеална за покриване на изискванията на микроуслугите. Когато се добавя нова функционалност, не е нужно да се променят всички съществуващи записи, което би могло да отнеме много време при миграции в релационни бази от данни.

# **III Глава:** Проектиране на системата

Изграждане на идеалния план за успешна система от комуникиращи контейнери.

## Изграждане на софтуерно задание

Главната идея на програмата ще бъде да предоставя платформа за обучение. Системата трябва да изобрази ясно разделение между различните пътища, по които можем да се развиваме в софтуерното инженерство. За всеки един път трябва да можем да предоставим различни умения, които си струва да усвоим. И за финал, за всяко умение трябва да имаме различен вид за предоставяне на информация. Следната графика представлява базов план за потребителско взаимодействие(фиг. 4):



Фиг.4. Взаимодействие със системата

От тази графика става ясно, че приложението е създадено за хора с всякакъв опит. Те трябва да могат да влязат в системата и да разгледат различни отдели(Topics) в компютърните науки, например:

* Какво значи да сме софтуерни разработчици?
* Какво е да сме системни администратори?
* Пътят на хардуерния инженер?
* Каква е работата на софтуерният мениджър на проекти?

Това са само малка част от множеството пътища, по които един софтуерен инженер може да поеме. Когато изберем софтуерния отдел, от който се интересуваме, ще получим списък с различни качества, върху които можем да работим за да станем по добри в дадената сфера. Например ако искаме да станем софтуерен разработчик ще получим следните предложения:

* Алгоритми
* Дата структури
* Програмни езици
* Познания на различни инструменти
* Софтуерен дизайн (design patterns)
* Комуникация
* Работа в екип
* Бази данни
* Контрол на версии

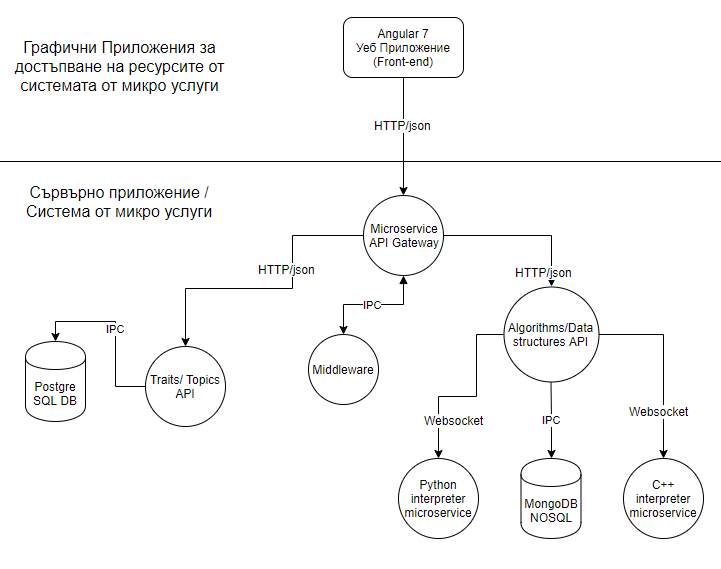
И много други…

Когато вече сме решили какво ни е интересно и искаме да развием ще получим списък с статии, които са по темата. Ако изберем алгоритми или дата структури например трябва да имаме различни статии, които съдържат информация за различните дата структури или популярни алгоритми за решаване на интересни проблеми. Последното изискване е да можем да изпълним кодът, който е предоставен заедно със статията. Той трябва да може да се изпълни на различни програмни езици и да можем да сравним бързодействието на различните изпълнения.

От друга страна, трябва да предоставим възможност на експерти, които са се доказали в дадена сфера да публикуват статии и работещ код, който да ги съпътства.

## Планиране на реализацията на проекта: Как проектирането на система от микроуслуги се различава от монолитната архитектура?

Най-важното нещо, което трябва да се има предвид, когато се създава система от микросервиси е целта да е дефинирана точно и да може да се разпределят различни длъжности на отделните микро-сървъри. Когато се придобие ясна представа как ще изглежда минималното бизнес изискване може да се начертае диаграма, която да представлява отделните части на приложението и връзките между тях. Например за създаване на приложение за обучение, което предоставя възможност за записване на статии и парченца код, които да могат да се изпълняват, може да се начертае следната диаграма (фиг.5):

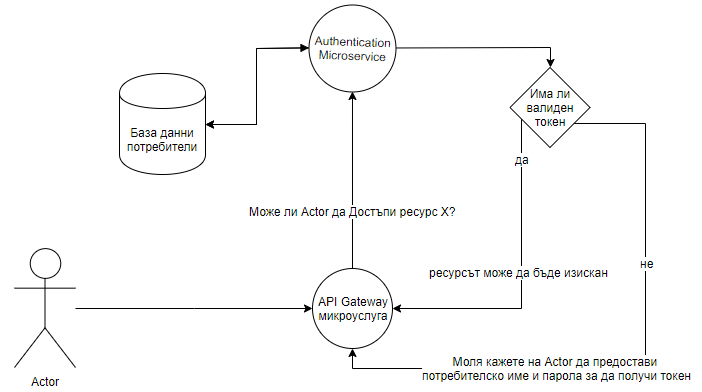


Фиг.5. Архитектура на приложението

където всяка фигура представлява отделен сървър, който се изпълнява на изолиран Docker контейнер. Тук можем да видим нашето сървърно приложение, което е независимо от графичните приложения, които ще работят с него. Той предоставя така нареченото RESTful API, което представлява програмен интерфейс, който използва HTTP/HTPPS заявки за да представлява дадени ресурси на неговите клиенти. Клиентите най-често получават желаните от тях данни в JSON формат, който прилича на Map<key,value>. Този начин на проектиране на WEB приложения означава, че можем да имаме графични среди, които да работят на множество платформи, като например в този случай ще използваме интернет страници, но в бъдеще може да добавим мобилно приложение, както и да предоставяме възможност на други софтуерни фирми да интегрират системата ни в техните графични приложения.

## Използване на услуга вход към програмния интерфейс: API Gateway Pattern

Както се забелязва на предишната диаграма, имаме един микро сървър, който служи като врата на цялата ни мрежа от услуги. Той е отговорен за това да пренасочва заявките към всички останали сървъри в системата от микроуслуги. Също така, това е добро място където да имплементираме различни междинни софтуерни фрагменти, които да афектират заявката по определен начин. Например това е добро място за имплементация на микроуслуга за контрол на достъпа (потребители)(фиг.6.):



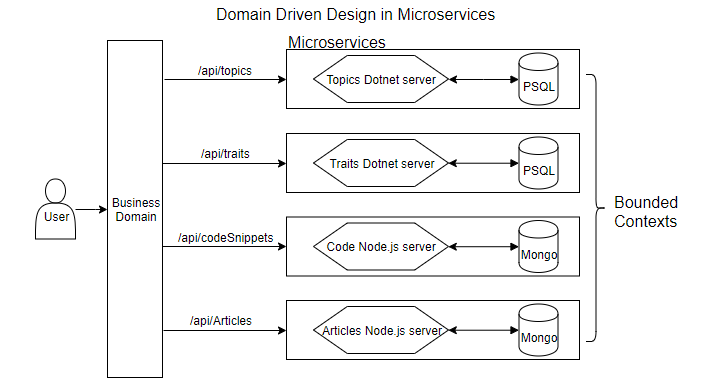
Фиг. 6. Пример за междинен сървърен метод

Това позволява на потребителите да получат въведат потребителското си име и парола, като в замяна получават код(или JSON WEB Token), който се съхранява в паметта на сървъра и стои запаметен за известно време. Той може да бъде ползван за получаване на достъп до различни ресурси, като трябва да го „покажем“ всеки път, когато достъпваме системата. Този код се асоциира със запис на даден потребител. Всеки потребител си има записани както потребителско име и парола, така и записи за контрол на достъп спрямо различните ресурси. Като например: един потребител, който се води експерт ще има права да променя, чете и качва нови статии, докато един обикновен потребител ще може само да чете този ресурс.

## Бази данни и модели на съхранение при системи от микросервиси

Съхраняването на данни при системите от микросервиси е много различно от това на монолитните архитектури. Прието е всяка услуга да притежава собствена база от данни, която е логически свързана с нея. Това прави възможно инсталирането на множество независими сървъри. Това означава, че всеки отбор, работещ по дадена услуга, също така ще ръководи нейната база от данни, без да се притеснява за останалите.[12]

В контекста на системата за обучение, също трябва да се спази парадигмата Domain Driven Design. Разделението на модули е съпътствано от бизнес модела, като всеки сървър си има отговаряща за него база от данни(Фиг.7).



Фиг.7. Бази от данни на приложението

Много важно правило е, че една услуга не трябва да може да достъпва базата от данни на друга. Това се извършва само чрез програмния интерфейс на всеки един контекст. Друго нещо, което е типично за базите от данни при микросервисите е, че за попълване на данни се спазва Code First подходът. Това означава, че първо се изграждат моделите в сървърното приложение и те са отговорни за структурата на базата от данни. Това е желателно да се прави най-вече, когато се планира разделянето на индивидуални сървъри в повече от един клъстер. Тогава може да има няколко еднакви услуги, които си имат собствена индивидуална база от данни.

# **IV Глава:** Софтуерна Реализация

Пътя по който проектирани микро услуги се превръщат в работещи клъстери от контейнери.

## Как се изолират отделните контейнери от „микросервиси“ използвайки Docker

Винаги когато започнем разработката на нов контейнер, на който ще работим сървър, трябва да опишем на Docker каква среда искаме да бъде създадена. Чистите Docker контейнери представляват Линукс или Уиндоус базирана изолирана среда. След това трябва да изпълним команди, които инсталират нужният за развойната ни среда софтуер. За да не се налага да правим това всеки път когато стартираме контейнер, създаваме конфигурационен файл, който да запомни всичките стъпки за настройване на изолирания контейнер. Най-често този файл се кръщава “Dockerfile”.

## Създаване на API Gateway микроуслуга и как тя е свързана с останалите компоненти на сървърната архитектура

### Създаване на “Dockerfile” и настройка на изолираната среда.

За реализирането на този компонент от използваната архитектура, се разчита на комбинация от Node.js, express и Typescript. За да бъде възможно да се ползват тези технологии в контейнер, трябва да се инсталират всички зависимости на него. Docker разполага с платформата си “Docker Hub”, която съдържа множество готови инструкции за изграждане на среда, които можем да ползваме като основа за нашата. В този случай се използва готов шаблон, предоставен от създателите на Node.js. Той се добавя по следния начин: „FROM node:8-slim“. Така вече имаме инсталиран node.js и всичко друго нужно за работата ни. Следващата стъпка е да се прехвърли директорията от машината на която се разработва проекта към контейнера, където ще изпълним сървърната си програма. Това става лесно със следните команди: „WORKDIR /server“ и „COPY . /server“. Така контейнерът разбира коя е работната му директория и се прехвърлят към нея текущата директория на конфигурационния файл. Крайната версия на този файл изглежда по следния начин:



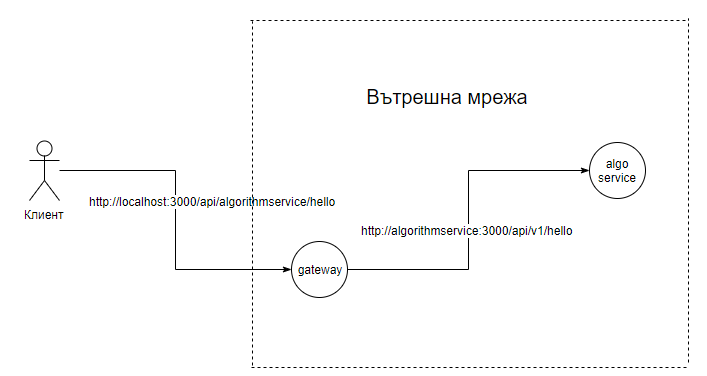
Тук се забелязва как след като се прехвърли работната директория стартират няколко скрипта, които стартират сървъра и правят така че порт 3000 да е видим за другите участници в мрежата. За да може да се създаде виртуална среда след тази конфигурация е нужно да се композира шаблон(Image) за стартиране на контейнер. Този може да се запази в Docker Hub профила на разработчика и може се достъпи от всяка машина, която има достъп до интернет.

### Конфигуриране на микроуслугата.

Тъй като целта на този контейнер е да препраща заявки към другите микроуслуги в сървърната ни мрежа, той използва конфигурационен файл, в който посочваме всички ресурси към които можем да достигнем през него. Той е от тип JSON. Например един запис в този конфигурационен файл би изглеждал по следния начин:



Това казва на сървъра да пренасочи всяка заявка с URL път “/api/algorithmservice” до услугата за достъп то алгоритми. Заявката пристига до този сървър и той я препраща към  
http://algorithmservice:3000/api/v1. Например(фиг. 7.):



Фиг.7. Пример за препращане на заявка

В “middleware” колекцията на конфигурацията могат да се изредят сървърни методи, които да се изпълнят преди да се продължи напред със изпълняването на заявката. Те могат да афектират текущата заявка, или да я откажат. Например един междинен сървърен метод би изглеждал по следния начин:



### Създаване на скрипт за стартиране на сървъра

Когато се изпълни командата “npm start” се стартира скрипт, който настройва express сървъра и стартира пренасочващата функционалност на сървъра:



Тук се използва класът – APIGateway. Неговата цел е да имплементира функционалността на сървърното приложение. Конструкторът му изглежда по следния начин:



В него се настройва сървърното приложение да използва нужните библиотеки. Също така се добавя път, през който да може да се провери дали сървъра работи. Най-важният метод, който се извиква е bootstrapService, който се грижи за разчитането на конфигурационния файл и за настройване на пренасочващите заявки:



Тази функция обхожда всички конфигурирани микроуслуги и прави възможен достъпа до тях.

## Създаване на микросервис за съхраняване на списъци: Topics API

Този контейнер представлява среда за разработка на dotnet core. Неговата цел е да съхранява данни в релационна база от данни. Сървърът предоставя програмен интерфейс, от който може да се извлече информация в JSON формат. Заявки към него трябва да може да прави само API Gateway микросервиса. Приложението използва парадигмата модел, изглед, контролер, като вместо да връща изглед под формата на HTML документ, той предоставя структурирани данни.

### Връзка с базата данни

Това сървърно приложение използва PostgreSQL за база от данни и за да осъществи връзка с нея използва Entity Framework. Изградени са класове на езикът C#, които съответстват и служат за връзка с таблици в базата от данни. Тях наричаме модели. Пример за модел е следния клас:



Всяко поле съответства на колона в базата от данни.

### Контролери и обработване на заявки

Обработването на клиентски заявки се извършва от контролерите. Те съдържат функциите, които се изпълняват при получаването на заявки и извършват операции върху базите от данни. Например, за ресурс Topics(Теми) отговаря следният контролер:



Той позволява достъпването на всички теми чрез „GET /api/topics” и само една конкретна тема чрез „GET /api/topics/{id}”.

## Как се съхраняват статиите и парчетата код, които се демонстрират на потребителите (algorithms service)?

Mикрсервисът, отговорен за съхраняването на статиите и парчетата код, които ще се предоставят на страниците за обучение използва node.js и express. Данните се съхраняват в нерелационна база данни: MongoDB. Заедно двата ресурса формират една обучителна статия.

## Услуга за интерпретиране на примерен код

Системата за обучение предлага функционалност, която позволява динамично изпълняване на примерни фрагменти от програмен код. Също така потребителите ще могат да променят тези фрагменти и да наблюдават как се променя изпълнението на програмата. Тези примери трябва да могат да се изпълняват на повече от един програмен език и да могат да се сравняват времената за изпълнение.

Най-голямото предизвикателство при осъществяването на подобна услуга е защитаването на сървъра от злонамерен програмен код. За това, тези скриптове трябва да бъдат подложени на определени забрани:

* Не трябва да позволяваме достъп до файловата система на сървъра
* Не трябва да могат да се отварят портове за достъп.
* Не може да бъдат използвани голямо количество ресурси от машината хост.
* Трябва да се изключва ако не е завършил изпълнението си до определено време.

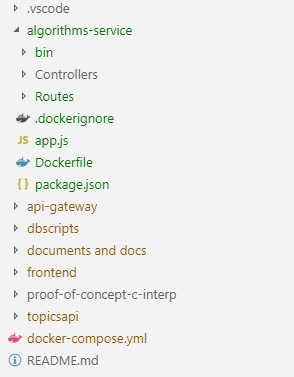
Въпреки че Docker не гарантира защитата от „атаки“ срещу ядрото на операционната система, той гарантира, че не могат да се използват повече ресурси от колкото е нужно. От сървъра зависи да се ограничи достъпа на клиента до други ресурси, които могат да позволят злонамерено поведение.

Микросервиса, който ще изпълнява задачата на интерпретатор е написан на JavaScript, използвайки express.js за сървърното приложение. Следният програмен фрагмент е пример за изпълняване на външен JavaScript код във Docker контейнера:



## Как е съединена системата от микроуслуги?

Архитектурата от микроуслуги зависи от много независими контейнери, които работят заедно за да предоставят множество сървърни услуги. За да се групира стартирането на всички микросервиси е използван инструментът docker-compose. За да може да се използва трябва да се създаде конфигурационен файл, в който се описват зависимостите между различните контейнери, директориите, индивидуалните конфигурации на микросервисите и др.. Главната директория изглежда по следния начин(Фиг.8.):



Фиг.8. Директория на приложението

Всеки микросервис се намира в собствена папка, заедно със своя Dockerfile. Конфигурационният файл “docker-compose.yml” е отговорен за групирането на всички отделни услуги. Съдържанието на файла е следното:

version: '3'

services:

web:

ports:

- "8080:80"

gateway:

image: gateway

build:

context: ./api-gateway

dockerfile: Dockerfile

command: npm run debug

volumes:

- ./api-gateway/dist:/server/dist

ports:

- "3000:3000"

- "5858:5858"

networks:

- microservices-api

algorithmservice:

build: ./algorithms-service/

ports:

- "3000"

networks:

- microservices-api

topicsapi:

image: topicsapi:latest

depends\_on:

- "postgresdb"

build:

context: ./topicsapi

dockerfile: Dockerfile

ports:

- "8000:80"

environment:

DB\_CONNECTION\_STRING: "host=postgresdb;port=5432;database=educationaldb;username=eduuser;password=asdqwe"

networks:

- microservices-api

postgresdb:

image: postgres:latest

ports:

- "5432"

restart: always

volumes:

- db\_volume:/var/lib/postgresql/data

- ./dbscripts/seed.sql:/docker-entrypoint-initdb.d/seed.sql

environment:

POSTGRES\_USER: "eduuser"

POSTGRES\_PASSWORD: "asdqwe"

POSTGRES\_DB: "educationaldb"

networks:

- microservices-api

mongodb:

image: mongo

networks:

- microservices-api

adminer:

image: adminer

restart: always

ports:

- 8080:8080

networks:

- microservices-api

volumes:

db\_volume:

networks:

microservices-api:

driver: bridge

В него се намират няколко главни конфигурации. “**version**” уточнява коя версия на docker-compose се използва. „**volumes**” е поле, в което можем да декларираме изолирани дискови пространства които могат да бъдат ползвани от различните контейнери. “**networks**” съдържа всички мрежи, които ще бъдат използвани от микроусугите. В полето „**services**” са групирани всички самостоятелни изолирани среди за разработка. Там може да се посочат индивидуалните им „Dockerfile“ конфигурации и да се допълни тяхната функционалност.

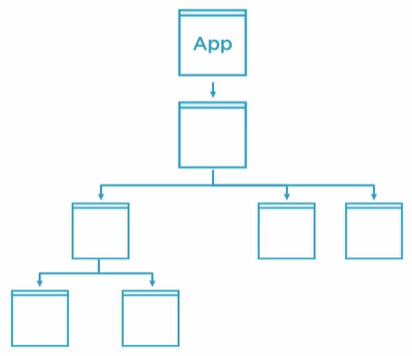
За да се стартират всички контейнери, описани в създадения конфигурационен файл, се използва командата „docker-compose up”. Тя е отговорна за това да инициализира контейнер от всички описани микросервиси. За да се изключат стартираните изолирани среди се използва „docker-compose down”.

### Всеки отделен „service” има определена роля.

**Web** услугата има за цел да стартира контейнер, в който ще бъде изпълнен nginx сървър, който ще бъде отговорен за доставянето на WEB страниците на потребителя.

## Потребителски интерфейс и връзка с архитектурата от микроуслуги

За създаването на WEB базиран потребителски интерфейс е използван Angular 7 framework. Когато се използва тази платформа винаги има компонент, който служи за основа на всички останали (Root component). Когато браузерът направи заявка към графичната система, компонентът маршрутизатор на Angular проверява дали има компонент, който отговаря на заявения път и предоставя съответния шаблонен компонент. След това се зарежда статичната страница и след време, когато се изпълнят заявките към сървъра се добавят и извлечените ресурси. Ако на една страница има много информация, тя най-често се визуализира в различни компоненти. По този начин се получава дървовидна структура от вложени графични елементи, които могат да се ползват на различни страници от WEB приложението (Фиг. 2).



фиг. 2 Вложени компоненти при Angular

За връзка с сървърното приложение е използван похват наречен Observable Pattern. Има TypeScript клас, който се използва за създаване на Http/Https заявки. Той се нарича Service. Например за създаване на заявки към Traits микроуслугата се използва следния клас:

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable, of } from 'rxjs';

import { Trait } from './Models/Trait';

import { tap, catchError } from 'rxjs/operators';

@Injectable({

providedIn: 'root'

})

export class TraitsService {

private uri: string = 'http://localhost:8000'

constructor(private http: HttpClient) {

}

getTrait(id: number): Observable<Trait[]> {

return this.http.get<Trait[]>(`${this.uri}/api/traits/byid/${id}`)

.pipe(

tap(\_ => console.log('fetched traits')),

catchError(this.handleError<Trait[]>('getHeroes', []))

);

}

private handleError<T> (operation = 'operation', result?: T) {

return (error: any): Observable<T> => {

// TODO: send the error to remote logging infrastructure

console.error(error); // log to console instead

// TODO: better job of transforming error for user consumption

console.log(`${operation} failed: ${error.message}`);

// Let the app keep running by returning an empty result.

return of(result as T);

};

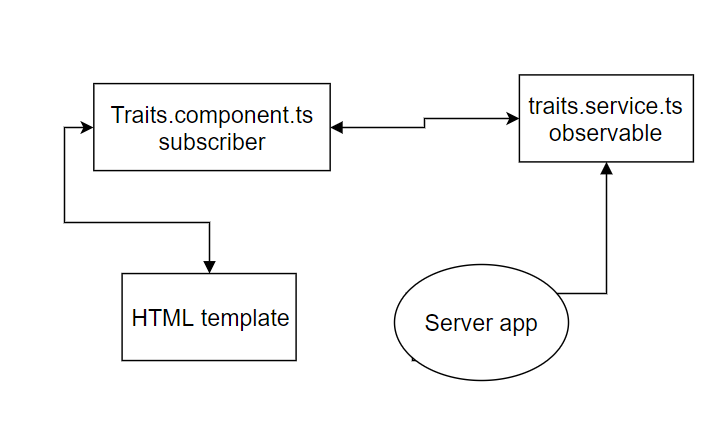
}

}

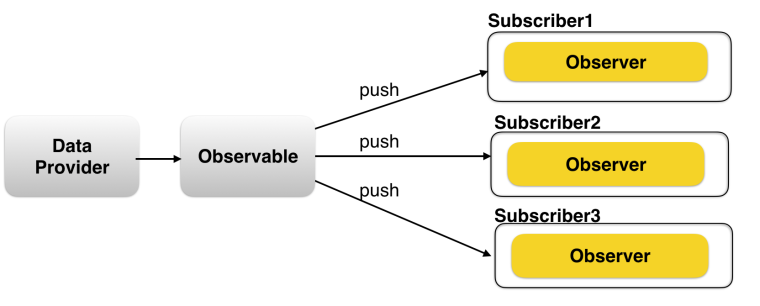
Неговите методи връщат обект от тип Observable. За да се предоставят данните до компонента се създава „абонамент“(Subscription):

this.traitsService.getTrait(id).subscribe(traits => (this.traits = traits));

по този начин член променливата traits, която е свързана с html шаблона ще се обнови когато заявката е приключила.(Фиг.3)



Така променливата, която е свързана с графичната визуализация не блокира възпроизвеждането на останалите елементи. Тя се абонира за данни, които ще получи след като класът Observer получи данните от сървъра. Към един Observer могат да се прикачат повече от един Subscriber(фиг).



# V Глава: Тестване на системата от микро услуги.

## Използване на системата за обучение

## Как пишем тестове за малки сървъри

# VI Глава: Заключение и бъдещо развитие

## Защо е лесно и безопасно да разширяваме системата ни от микроуслуги?

## Използвана литература

1. Building Microservices by Sam Newman Publisher: O'Reilly Media, Inc. Release Date: February 2015
2. Microservices: Flexible Software Architecture 1st Edition Addison-Wesley Professional; October 3, 2016
3. Microservice Architecture (https://microservices.io/)
4. Angular: One framework, Mobile & Desktop (<https://angular.io/>)
5. Learn Docker .NET Core, Java, Node.JS, PHP or Python
6. The Docker Book, James Turnbull, March 11, 2017
7. Hands-On RESTful Web Services with TypeScript 3, Design and develop scalable and RESTful APIs with TypeScript and Node.js
8. Introduction to MongoDB, Express, Angular and Node.js A practical guide to the tried-and-true production-ready MEAN stack, with additional tips, tricks and best practices, Paul Oluyege, Feb 11, 2019
9. MongoDB: The Definitive Guide by Kristina Chodorow
10. Define and run multi-container applications with Docker (<https://docs.docker.com/compose/>)
11. What is PostgreSQL - <http://www.postgresqltutorial.com/what-is-postgresql/>
12. Breaking the Monolithic Database in Your Microservices Architecture (https://dzone.com/articles/breaking-the-monolithic-database-in-your-microserv)

# Приложение – програмен код

1. **Docker-compose.yml:**

version: '3'

services:

#web:

#ports:

#- "8080:80"

gateway:

image: gateway

build:

context: ./api-gateway

dockerfile: Dockerfile

command: npm run debug

volumes:

- ./api-gateway/dist:/server/dist

ports:

- "3000:3000"

- "5858:5858"

networks:

- microservices-api

algorithmservice:

build: ./algorithms-service/

ports:

- "3000"

networks:

- microservices-api

topicsapi:

image: topicsapi:latest

depends\_on:

- "postgresdb"

build:

context: ./topicsapi

dockerfile: Dockerfile

ports:

- "8000:80"

environment:

DB\_CONNECTION\_STRING: "host=postgresdb;port=5432;database=educationaldb;username=eduuser;password=asdqwe"

networks:

- microservices-api

postgresdb:

image: postgres:latest

ports:

- "5432"

restart: always

volumes:

- db\_volume:/var/lib/postgresql/data

- ./dbscripts/seed.sql:/docker-entrypoint-initdb.d/seed.sql

environment:

POSTGRES\_USER: "eduuser"

POSTGRES\_PASSWORD: "asdqwe"

POSTGRES\_DB: "educationaldb"

networks:

- microservices-api

mongodb:

image: mongo

networks:

- microservices-api

adminer:

image: adminer

restart: always

ports:

- 8080:8080

networks:

- microservices-api

ccompilerservice:

build: ./ccompilerservice/

ports:

- "3002:3000"

networks:

- microservices-api

volumes:

db\_volume:

networks:

microservices-api:

driver: bridge

1. **serverSetup(api-gateway):**

#!/usr/bin/env node

// Module dependencies.

import { APIGateway } from '../index'

const apig = new APIGateway();

const app = apig.getApp();

const debug = require('debug')('api-gateway:server');

const http = require('http');

// Използваме порт от глобална променлива за дефиниране на среда ако е наличен

// Порт по подразбиране 3000

const port = normalizePort(process.env.PORT || '3000');

app.set('port', port);

// създаваме инстанция на сървъра

const server = http.createServer(app);

// Стартираме сървъра на избрания порт

server.listen(port);

server.on('error', onError);

server.on('listening', onListening);

function normalizePort(val) {

const port = parseInt(val, 10);

if (isNaN(port)) {

return val;

}

if (port >= 0) {

return port;

}

return false;

}

function onError(error) {

if (error.syscall !== 'listen') {

throw error;

}

const bind = typeof port === 'string'

? 'Pipe ' + port

: 'Port ' + port;

switch (error.code) {

case 'EACCES':

console.error(bind + ' requires elevated privileges');

process.exit(1);

break;

case 'EADDRINUSE':

console.error(bind + ' is already in use');

process.exit(1);

break;

default:

throw error;

}

}

function onListening() {

const addr = server.address();

const bind = typeof addr === 'string'

? 'pipe ' + addr

: 'port ' + addr.port;

debug('Listening on ' + bind);

}

1. **Index.ts(api-gateway):**

import express from 'express';

import bodyParser = require('body-parser');

import logger = require('morgan');

import cookieParser = require('cookie-parser');

import url = require('url');

import services from '../services.json';

import { proxy } from './lib/proxy';

import { restreamer } from './lib/restreamer';

export class APIGateway {

private readonly expressInstance: any = express();

constructor() {

// Настройваме библиотеки, които обработват заявката.

this.setUpApp(this.expressInstance);

// Създаваме път, който достъпваме за да проверим дали сървъра работи.

this.setWelcomeRoute(this.expressInstance)

// имплементираме функционалността, където казваме при коя заявка към коя услуга да изисква отговор

this.bootstrapServices(this.expressInstance);

}

/\*\*

\* Set up the app

\*/

setUpApp(app) {

app.use(logger('dev'));

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.use(cookieParser());

}

/\*\*

\* A welcomming message that tells the user the api gateway is running

\* @param app express instance

\*/

setWelcomeRoute(app) {

app.get('/', function (req, res) {

res.json({

message: "The API Gateway is up!"

});

});

}

bootstrapServices(app){

// Обхождаме всички конфигурации в services.json конфигурационен файл

services.forEach(service => {

const {name, host, port} = service

const rootPath = service.rootPath || "";

const protocol = service.protocol || "http";

console.log(`Adding service: ${protocol}://${host}:${port}/${rootPath}`);

// Тук достъпваме написани от нас библиотеки, които искаме да

// повлиаят на заявката по някакъв начин (това е добро място за имплементиране на автентикация)

let middleware = [];

if (service.middleware) {

middleware = service.middleware.map(text => require(`./middleware/${text}`));

}

// restream-ваме заявката за да може да бъде препратена.

middleware.push(restreamer());

// настройваме сървърът да препраща заявките към зададената конфигурация

app.use(`/api/${name}\*`, middleware, (req, res, next) => {

const newPath = url.parse(req.originalUrl).pathname.replace(`/api/${name}`, rootPath);

console.log(`Forwarding request to: ${newPath}`);

proxy.web(req, res, { target: `${protocol}://${host}:${port}/${newPath}` }, next);

});

});

}

public getApp = () => this.expressInstance

};

1. **Dockerfile(api-gateway):**

FROM node:8-slim

WORKDIR /server

COPY . /server

RUN npm install

EXPOSE 3000

CMD [ "npm", "start" ]

1. **Index.js(compiler-service):**

const promisify = require('util').promisify;

const { exec } = require('child\_process');

const testFolder = '.';

const fs = require('fs');

const express = require('express')

const app = express()

const port = 3000

const options = {

timeout: 100,

stdio: 'inherit',

shell: true

}

const vm = require('vm');

const writeFile = promisify(fs.writeFile);

app.use(express.json())

app.get('/', (req, res) => res.send('Hello World!'))

app.get('/readdir', (req, res) => {

fs.readdir(testFolder, (err, files) => {

console.log("readdir invoked by client");

let response = ''

files.forEach(file => {

response += `${file} \n`;

});

res.send(response)

});

})

app.post('/executec', async (req, res) => {

try {

console.log(req.body);

let sourceCode = req.body.code;

// write to a new(or existing) file named main.c

const status = await writeFile('main.c', sourceCode);

exec('gcc -o myapp main.c', (error, stdout, stderr) => {

console.log('Compilation stdout: ', stdout);

console.log('Compilation stderr: ', stderr);

exec('./myapp', options, (error, stdout, stderr) => {

if (error) {

console.log(`exec error: ${error}`);

return;

}

console.log(`stdout ${stdout}`);

console.log(`stdout ${stderr}`);

res.json({

stdout: stdout,

stderr: stderr

});

})

});

} catch (err) {

res.send('something went wrong with compilation: ' + err)

}

})

app.post('/executejs', async (req, res) => {

try {

console.log(req.body);

let sourceCode = req.body.code;

const script = new vm.Script(sourceCode, {

filename: 'my-index-stacktrace.js', // filename for stack traces

lineOffset: 1, // line number offset to be used for stack traces

columnOffset: 1, // column number offset to be used for stack traces

displayErrors: true,

timeout: 1000 // ms

});

let sandBox = {

require: require,

console: {

log: function () {

for (var i=0; i < arguments.length; i++) {

if (arguments[i]) {

this.stdout += arguments[i];

}

}

this.stdout += "\n";

}

},

stdout: '',

stderr: ''

}

script.runInNewContext(sandBox);

res.json({

stdout: sandBox.stdout,

stderr: sandBox.stderr

});

} catch (err) {

res.send('something went wrong with compilation: ' + err)

}

})

app.listen(port, () => console.log(`c compiler/interpreter app listening on port ${port}!`))

1. **seed.sql(dbscripts)**

\connect educationaldb

CREATE TABLE traits

(

id serial PRIMARY KEY,

title VARCHAR (50) NOT NULL,

description VARCHAR (100) NOT NULL,

imageurl VARCHAR (100),

nosqlarticlepage varchar (50) NOT NULL,

topic\_id integer REFERENCES topics(id),

);

CREATE TABLE topics

(

id serial PRIMARY KEY,

title VARCHAR (50) NOT NULL,

description VARCHAR (100) NOT NULL,

imageurl VARCHAR (100)

);

ALTER TABLE "topics" OWNER TO eduuser;

ALTER TABLE "traits" OWNER TO eduuser;

Insert into traits(title,description) values( 'Title 1','Description 1');

Insert into traits(title,description) values( 'Title 2','Description 2');

Insert into traits(title,description) values( 'Title 3','Description 3');

Insert into traits(title,description) values( 'Title 4','Description 4');

Insert into topics(title,description, topic\_id)

values( 'Title 1','Description 1', 1);

Insert into topics(title,description, topic\_id)

values( 'Title 2','Description 2', 1);

Insert into topics(title,description, topic\_id)

values( 'Title 3','Description 3', 2);

Insert into topics(title,description, topic\_id)

values( 'Title 4','Description 4', 2);

**7. app.module.ts (frontend service)**

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { MarkdownModule } from 'ngx-markdown';

import { AppRoutingModule } from './app-routing.module';

import { MonacoEditorModule, NgxMonacoEditorConfig } from 'ngx-monaco-editor';

import { AppComponent } from './app.component';

import { LoginComponent } from './login/login.component';

import { HomeComponent } from './home/home.component';

import { NavbarComponent } from './navbar/navbar.component';

import { ListPageComponent } from './list-page/list-page.component';

import { CodeEditorComponent } from './code-editor/code-editor.component';

import { TopicService } from './topic.service';

import { TopicsComponent } from './topics/topics.component';

import { HttpClientModule, HttpClient } from '@angular/common/http';

import { TutorialPageComponent } from './tutorial-page/tutorial-page.component';

import { TraitsPageComponent } from './traits-page/traits-page.component';

const monacoConfig: NgxMonacoEditorConfig = {

baseUrl: '/assets', // configure base path for monaco editor

defaultOptions: { scrollBeyondLastLine: false }, // pass default options to be used

onMonacoLoad: () => { console.log((<any>window).monaco); } // here monaco object will be available as window.monaco use this function to extend monaco editor functionalities.

};

@NgModule({

declarations: [

AppComponent,

LoginComponent,

HomeComponent,

NavbarComponent,

ListPageComponent,

CodeEditorComponent,

TopicsComponent,

TutorialPageComponent,

TraitsPageComponent

],

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule,

FormsModule,

MonacoEditorModule.forRoot(monacoConfig),

MarkdownModule.forRoot()

],

providers: [ TopicService ],

bootstrap: [AppComponent]

})

export class AppModule { }

1. **app.component.html(frontend service)**

<!--The content below is only a placeholder and can be replaced.-->

<app-navbar></app-navbar>

<div class="router-wrapper">

<router-outlet></router-outlet>

</div>

App-routing.module.ts (frontend service)

import { NgModule } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

import { LoginComponent } from './login/login.component';

import { TopicsComponent } from './topics/topics.component';

import { HomeComponent } from './home/home.component';

import { ListPageComponent } from './list-page/list-page.component';

import { CodeEditorComponent } from './code-editor/code-editor.component';

import { TutorialPageComponent } from './tutorial-page/tutorial-page.component';

import { TraitsPageComponent } from "./traits-page/traits-page.component";

import { from } from 'rxjs';

const routes: Routes = [

{

path: '',component: HomeComponent,

//canActivate: [AuthGuard]

},

{ path: 'login',component: LoginComponent },

{ path: 'topics',component: TopicsComponent },

{ path: 'list', component: ListPageComponent },

{ path: 'codetest', component: CodeEditorComponent },

{ path: 'tutorial', component: TutorialPageComponent },

{ path: 'trait/:id', component: TraitsPageComponent },

// препратка към главен компонент

{ path: '\*\*', redirectTo: '' }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

1. **tutorial-page.component.html: (frontend service):**

<div class="container">

<div class="row">

<div class="col-sm">

<markdown [data]="markdown"></markdown>

</div>

<div class="col-sm">

<app-code-editor></app-code-editor>

</div>

</div>

</div>

1. **tutorial-page.component.ts**

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-tutorial-page',

templateUrl: './tutorial-page.component.html',

styleUrls: ['./tutorial-page.component.scss']

})

export class TutorialPageComponent implements OnInit {

markdown: string = `

## Example Header

Example content

- some things

`

constructor() { }

ngOnInit() {

}

}

1. **Traits-page.component.html**

<div style="text-align:center">

<h1>

{{ title }}

</h1>

</div>

<div \*ngFor="let trait of traits" class="list-group">

<a href="#" class="list-group-item list-group-item-action flex-column align-items-star">

<div class="d-flex w-100 justify-content-between">

<h5 class="mb-1">{{trait.title}}</h5>

<small>3 days ago</small>

</div>

<p class="mb-1">Donec id elit non mi porta gravida at eget metus. Maecenas sed diam eget risus varius

blandit.</p>

<small>Donec id elit non mi porta.</small>

</a>

</div>

1. **Traits-page.component.ts**

import { Component, OnInit, Input } from '@angular/core';

import { ActivatedRoute, Params } from '@angular/router';

import { TraitsService } from '../traits.service';

import { Trait } from '../Models/Trait';

@Component({

selector: 'app-traits-page',

templateUrl: './traits-page.component.html',

styleUrls: ['./traits-page.component.scss']

})

export class TraitsPageComponent implements OnInit {

@Input() traits: Trait[] = [];

navigated = false; // true if navigated here

constructor(

private traitsService: TraitsService,

private route: ActivatedRoute

) { }

ngOnInit() {

this.route.params.forEach((params: Params) => {

if (params['id'] !== undefined) {

const id = +params['id'];

this.navigated = true;

this.traitsService.getTrait(id).subscribe(traits => (this.traits = traits));

} else {

this.navigated = false;

this.traits = [];

}

});

}

}

1. **Topics.component.html**

<div style="text-align:center">

<h1>

{{ title }}

</h1>

</div>

<div \*ngFor="let topic of topics" class="list-group">

<a href="trait/{{topic.id}}" class="list-group-item list-group-item-action flex-column align-items-star">

<div class="d-flex w-100 justify-content-between">

<h5 class="mb-1">{{topic.title}}</h5>

<small>3 days ago</small>

</div>

<p class="mb-1">Donec id elit non mi porta gravida at eget metus. Maecenas sed diam eget risus varius

blandit.</p>

<small>Donec id elit non mi porta.</small>

</a>

</div>

1. **Topics.component.ts**

import { Component, OnInit } from '@angular/core';

import { TopicService } from '../topic.service';

import { Topic } from '../Models/Topic';

@Component({

selector: 'app-topics',

templateUrl: './topics.component.html',

styleUrls: ['./topics.component.scss']

})

export class TopicsComponent implements OnInit {

title: string = "What Are You interested In?";

topics: Topic[];

constructor(private topicService: TopicService) { }

ngOnInit() {

this.getTopics();

}

getTopics(): void {

this.topicService.getTopics()

.subscribe(topics => {

console.log("ASDASDASDASDADS",topics);

this.topics = topics

});

}

}

1. **Navbar.component.html**

<!-- Navigation -->

<nav class="navbar navbar-expand-lg navbar-light bg-light shadow fixed-top">

<div class="container">

<a class="navbar-brand" href="#">Development Path</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarResponsive" aria-controls="navbarResponsive" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarResponsive">

<ul class="navbar-nav ml-auto">

<li class="nav-item active">

<a class="nav-link" href="#">Home

<span class="sr-only">(current)</span>

</a>

</li>

<li class="nav-item">

<a class="nav-link" href="topics">Our Incredible World</a>

</li>

<li class="nav-item">

<a class="nav-link" href="tutorial">Services</a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">Contact a tutor</a>

</li>

</ul>

</div>

</div>

</nav>

1. **Модели(Topic, Trait):**

export interface Topic {

name: string;

description: string;

}

export class Trait {

name: string;

description: string;

noSqlArticlePage: number | null;

imageUrl: string | null;

topicId: number | null;

}

1. **Code-editor.component.html**

<ngx-monaco-editor class="my-code-editor" [options]="editorOptions" [(ngModel)]="code"></ngx-monaco-editor>

1. **Code-editor.component.ts**

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-code-editor',

templateUrl: './code-editor.component.html',

styleUrls: ['./code-editor.component.scss']

})

export class CodeEditorComponent implements OnInit {

editorOptions = {theme: 'vs-light', language: 'javascript'};

code: string= 'function x() {\nconsole.log("Hello world!");\n}';

constructor() { }

ngOnInit() {

}

onInit(editor: { getPosition: () => void; }) {

let line = editor.getPosition();

console.log(line);

}

}

1. **Traits.service.ts**

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable, of } from 'rxjs';

import { Trait } from './Models/Trait';

import { tap, catchError } from 'rxjs/operators';

@Injectable({

providedIn: 'root'

})

export class TraitsService {

private uri: string = 'http://localhost:8000'

constructor(private http: HttpClient) {

}

getTrait(id: number): Observable<Trait[]> {

return this.http.get<Trait[]>(`${this.uri}/api/traits/byid/${id}`)

.pipe(

tap(\_ => console.log('fetched traits')),

catchError(this.handleError<Trait[]>('getHeroes', []))

);

}

private handleError<T> (operation = 'operation', result?: T) {

return (error: any): Observable<T> => {

// TODO: send the error to remote logging infrastructure

console.error(error); // log to console instead

// TODO: better job of transforming error for user consumption

console.log(`${operation} failed: ${error.message}`);

// Let the app keep running by returning an empty result.

return of(result as T);

};

}

}

**20. Topic.service.ts**

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable, of } from 'rxjs';

import { catchError, map, tap } from 'rxjs/operators';

import { Topic

} from './Models/Topic';

@Injectable({

providedIn: 'root'

})

export class TopicService {

constructor(private http: HttpClient) { }

uri: string = 'http://localhost:8000'

getTopics(): Observable<Topic[]> {

return this.http.get<Topic[]>(`${this.uri}/api/topics`)

.pipe(

tap(\_ => console.log('fetched topics')),

catchError(this.handleError<Topic[]>('getHeroes', []))

);

}

getTopicById(id: string) {

return this.http.get(`${this.uri}/api/topics/${id}`);

}

addTopic(title, responsible, description, severity) {

const topic = {

title: title,

description: description

};

return this.http.post(`${this.uri}/Topic/add`, topic);

}

/\*\*

\* Handle Http operation that failed.

\* Let the app continue.

\* @param operation - name of the operation that failed

\* @param result - optional value to return as the observable result

\*/

private handleError<T> (operation = 'operation', result?: T) {

return (error: any): Observable<T> => {

// TODO: send the error to remote logging infrastructure

console.error(error); // log to console instead

// TODO: better job of transforming error for user consumption

console.log(`${operation} failed: ${error.message}`);

// Let the app keep running by returning an empty result.

return of(result as T);

};

}

}

1. **Startup.cs (Topics API)**

using System;

using System.Collections.Generic;

using System.Linq;

using System.IO;

using System.Threading.Tasks;

using System.Reflection;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.HttpsPolicy;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Logging;

using Microsoft.Extensions.Options;

using Microsoft.EntityFrameworkCore;

using topicsapi.Models;

using Swashbuckle.AspNetCore.Swagger;

namespace topicsapi

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

readonly string MyAllowSpecificOrigins = "\_myAllowSpecificOrigins";

// This method gets called by the runtime. Use this method to add services to the container.

public void ConfigureServices(IServiceCollection services)

{

var connectionString = Environment.GetEnvironmentVariable("DB\_CONNECTION\_STRING");

services.AddDbContext<ApiDbContext>(options =>

options.UseNpgsql(

connectionString

)

);

services.AddCors(options =>

{

options.AddPolicy(MyAllowSpecificOrigins,

builder =>

{

builder.WithOrigins(

"http://localhost:4200",

"http://localhost:3000"

);

});

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

// Register the Swagger generator, defining 1 or more Swagger documents

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });

});

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseHsts();

}

// Enable middleware to serve generated Swagger as a JSON endpoint.

app.UseSwagger();

app.UseCors(MyAllowSpecificOrigins);

// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),

// specifying the Swagger JSON endpoint.

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");

});

app.UseHttpsRedirection();

app.UseMvc();

}

}

}

1. **Dockerfile(Topics API)**

FROM microsoft/dotnet:2.1-sdk AS build

WORKDIR /app

COPY \*.csproj ./

RUN dotnet restore topicsapi.csproj

COPY . ./

RUN dotnet publish topicsapi.csproj -c Release -o out

FROM microsoft/dotnet:2.1-aspnetcore-runtime AS runtime

WORKDIR /app

COPY --from=build /app/out .

ENTRYPOINT ["dotnet", "topicsapi.dll"]`

1. **Topic.cs**

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace topicsapi.Models

{

public class Topic

{

public int Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public string ImageUrl { get; set;}

}

}

1. **Trait.cs**

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace topicsapi.Models

{

public class Trait

{

public int Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

// link to the nosql database record of the asset

public string NoSQLArticlePage { get; set; }

public string ImageUrl { get; set; }

[ForeignKey("topicid")]

public int TopicId { get; set; }

}

}

**25. ApiDBContext.cs**

using topicsapi.Maps;

using Microsoft.EntityFrameworkCore;

using topicsapi.Models;

using System;

namespace topicsapi.Models{

public class ApiDbContext : DbContext

{

public ApiDbContext(DbContextOptions<ApiDbContext> options)

: base(options)

{

}

public DbSet<Topic> Topics { get; set; }

public DbSet<Trait> Traits { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

new TopicMap(modelBuilder.Entity<Topic>());

new TraitMap(modelBuilder.Entity<Trait>());

}

}

}

1. **TopicMap.cs**

using topicsapi.Models;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace topicsapi.Maps{

#pragma warning disable CS1591

public class TopicMap

{

public TopicMap(EntityTypeBuilder<Topic> entityBuilder)

{

entityBuilder.HasKey(x => x.Id);

entityBuilder.ToTable("topics");

entityBuilder.Property(x => x.Id).HasColumnName("id");

entityBuilder.Property(x => x.Title).HasColumnName("title");

entityBuilder.Property(x => x.Description).HasColumnName("description");

entityBuilder.Property(x => x.ImageUrl).HasColumnName("imageurl");

}

}

#pragma warning restore CS1591

}

1. **TraitMap.cs**

using topicsapi.Models;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace topicsapi.Maps{

#pragma warning disable CS1591

public class TraitMap

{

public TraitMap(EntityTypeBuilder<Trait> entityBuilder)

{

entityBuilder.HasKey(x => x.Id);

entityBuilder.ToTable("traits");

entityBuilder.Property(x => x.Id).HasColumnName("id");

entityBuilder.Property(x => x.Title).HasColumnName("title");

entityBuilder.Property(x => x.Description).HasColumnName("description");

entityBuilder.Property(x => x.ImageUrl).HasColumnName("imageurl");

entityBuilder.Property(x => x.NoSQLArticlePage).HasColumnName("nosqlarticlepage");

entityBuilder.Property(x => x.TopicId).HasColumnName("topic\_id");

}

}

#pragma warning restore CS1591

}

1. **TraitsController.cs**

using topicsapi.Models;

using Microsoft.AspNetCore.Mvc;

using System.Text.Encodings.Web;

using System.Linq;

namespace topicsapi.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class TraitsController : ControllerBase

{

private readonly ApiDbContext \_context;

public TraitsController(ApiDbContext context)

{

\_context = context;

}

[HttpGet]

public object Get()

{

return \_context.Traits.Where(b => b.Title.Contains("Title")).Select((c) => new

{

Id = c.Id,

Title = c.Title,

Description = c.Description,

NoSqlArticlePage = c.NoSQLArticlePage,

ImageUrl = c.ImageUrl,

TopicId = c.TopicId

}).ToList();

}

[HttpGet("{title}")]

public object GetByTitle(string title)

{

return \_context.Traits.Where(b => b.Title == title).Select((c) => new

{

Id = c.Id,

Title = c.Title,

Description = c.Description,

NoSqlArticlePage = c.NoSQLArticlePage,

ImageUrl = c.ImageUrl,

TopicId = c.TopicId

}).ToList();

}

[HttpGet("byid/{id}")]

public object GetById(int id)

{

return \_context.Traits.Where(b => b.TopicId == id).Select((c) => new

{

Id = c.Id,

Title = c.Title,

Description = c.Description,

NoSqlArticlePage = c.NoSQLArticlePage,

ImageUrl = c.ImageUrl,

TopicId = c.TopicId

}).ToList();

}

}

}