Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему: «**Розробка системи вступу в НаУКМА**»

Виконав: студент 4-го року навчання,

Спеціальності

121 «Інженерія Програмного Забезпечення»

Студента Синицина Владислава

Керівник Глибовець А.М.

магістр комп’ютерних наук, асистент

«19» травня 2022 р.

Київ – 2022

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“10” жовтня 2020 року

ЗАВДАННЯ
ДЛЯ КУРСОВОЇ РОБОТИ СТУДЕНТУ

Синицину Власдиславу

1. Тема роботи **«Розробка системи вступу в НаУКМА»**, керівник роботи Глибовець Андрій Миколайович, декан факультету інформатики

2. Строк подання студентом роботи 20 травня 2022

3. План роботи

    Анотація

    Вступ

    Розділ 1. Дослідження та аналіз предметної області

        Архітектура застосунків в програмній інженерії

        Критерії якісної архітектури

        Монолітна архітектура

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата с науко
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	10 жовтня 2020	
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	10 жовтеня 2020 – 2 листопада 2020	
3.	Складання плану каліф. роботи та узгодження з науковим керівником	2 листопада 2020	
4.	Написання розділів роботи	2 листопада 2020 – 01 березня 2021	
5.	Проміжний контроль виконання роботи	01 лютого 2021	
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	11 січня 2021 – 29 березня 2021	
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2021	
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2021	
	Розділ 3 (проектно-рекомендаційна частина)	29 березня 2021	
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	01 квітня 2021 – 06 травня 2021	
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	17 травня 2021	
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК	

Графік узгоджено 10 жовтня 2021 р.

Науковий керівник Глибовець Андрій Миколайович
Виконавець курсової роботи Синицин Владислав

ЗМІСТ

Анотація

У даній роботі розглядаються особливості побудови архітектури програмних застосунків. Критерії, які допомагають визначити, наскільки якісною є архітектура певної програмної системи. Розглядаються найбільш часто використовувані підходи до побудови архітектури застосунку, а саме монолітний та мікросервісний підходи. Надаються порівняння даних підходів, їх характеристики, переваги та недоліки на основі вимог до конкретної програмної системи. Розглядаються принципи побудови мікросервісного застосунку на прикладі системи вступу в НаУКМА.

Вступ

Метою цієї роботи є дослідити різні методи побудови архітектури програмного забезпечення; критерії якості розробленої архітектури на основі вимог до системи; розглянути різні підходити до створення дизайну архітектури програмного забезпечення; проаналізувати монолітну та мікросервісну архітектури; порівняти дані підходи; обґрунтувати вибір мікросервісної архітектури для розробки системи вступу в НаУКМА; описати принципи побудови даного мікросервісного застосунку.

Розділ 1. Дослідження та аналіз предметної області

1.1. Архітектура застосунків в програмній інженерії

Визначення поняття «архітектура програмного забезпечення» не має одного чіткого значення, та може відрізнятися залежно від автора, типу роботи, часу її написання, тенденцій в розробці програмного забезпечення, області, щодо якої визначається дане поняття та багатьох інших чинників.

Тим не менше, варто розглянути спільні риси, які стосуються архітектури програмного забезпечення в широкому розумінні цього поняття, та які можуть найчастіше зустрічатися при спробі його описати.

Перш за все, архітектура застосунку – це фундаментальні домовленості та рішення.

Якщо порівнювати з будівництвом та архітектурою, звідки, власне, до нас даний термін і прийшов, то архітектура програмного рішення – це як проект будинку. Він креслиться та узгоджується до того, як почалося будівництво. На плані передбачено, де будуть проведені комунікації, яка плаща приміщень та їх взаємне розташування, де та як будуть розташовані сходи та ліфти. Також при розробці плану заздалегідь розраховуються необхідна товщина стін та перекриттів, міцність приміщень, скільки людей можуть одночасно в них знаходитися тощо.

Все це можна застосувати і до архітектури застосунку. Ще до початку розробки треба визначитися з тим, як різні частини програмного рішення будуть спілкуватися між собою; якими будуть ці частини, які інтерфейси вони будуть використовувати; яке навантаження повинен витримувати застосунок, та які рішення мають бути застосовані для цього тощо.

Друге твердження, яке часто зустрічається в різних джерелах, – архітектура застосунку – це щось важливе.

І знову складно не погодитися. Архітектура визначає фундаментальні концепції та обмеження, які має програмний продукт. Так само, як складно переоцінити важливість фундаменту для будинку, так і складно переоцінити значення архітектури в застосунках. Велику та надійну програмну систему не можливо створити без чогось, що буде об’єднувати різні її частини, буде описувати принципи їх взаємодії; диктувати, як саме впроваджувати те чи інше рішення. Всі ці завдання покладено саме на архітектуру застосунку, тому складно перебільшити її важливість.

З перших двох складових опису архітектури досить логічно випливає ще одне твердження. А саме: архітектуру застосунку складно змінювати.

Знову повернемося до порівняння з будинком. Якщо він вже майже або, тим більше, цілком зведений, то перенести комунікації, змінити матеріал, з якого виготовлені стіні, або додати ще сорок поверхів – задача, яку, загалом, вирішити можливо, але скільки будуть коштувати такі зміни, та чи буде після них взагалі функціонувати будівля – серйозне питання, на яке складно дати просту відповідь. Як і на питання доцільності таких змін. В більшості випадків, легше, дешевше та швидше буде побудувати нову будівлю з нуля, враховуючи всі нові вимоги, чим перероблювати стару.

Те ж справедливо і для архітектури програмної системи. Зміна її на пізньому етапі розробки може бути настільки складною, довгою та дорогою по бюджету і людських ресурсах, що доцільність таких змін викликає серйозне запитання. І як і з будинком, часто розробити систему з нуля буде швидше та дешевше, ніж зміна вже існуючої.

Останню за номером, але не за значенням, вимогу до архітектури програмних систем, яка присутня у більшісті джерел, можна описати так: архітектура має бути використана повторно.

Таке твердження не викликає запитань, оскільки проектування чогось в будь-якій сфері є дуже ресурсозатратною роботою. Як по грошам, так і по часу. Звідси виникає цілком логічне бажання розробити щось один раз, і потім використовувати його повторно якомога більшу кількість разів.

До прикладу, забудова за радянських часів здійснювалася однаковими типовими будинками, що дозволяло в короткі терміни зводити велику кількість останніх, при цьому дешево, оскільки всі процеси були стандартизовані, та не витрачаючи кожного разу час на проектування.

Те ж можна стверджувати і стосовно архітектури програмних систем. Хорошу архітектуру можна використати повторно, іноді трохи адаптувавши відповідно до задачі. Звичайно, це не є вирішальним фактором при проектуванні, але вигідно відрізняє продуману якісну архітектуру від поганої.

Такими є основні аспекти стосовно поняття архітектури в програмних системах. Крім них, є ще багато неоднозначних концепцій та принципів, які стверджують, як саме вона повинна створюватися, та які є способи нею керувати.

Важлива думка, навколо якої побудовані викладені концепції та судження, та яка, можна так сказати, є джерелом визначення архітектури – «Архітектурні рішення складно змінювати». Або, якщо підійти до цього питання з іншого боку, – «Рішення, які складно змінити, є архітектурними».

Ось яке визначення поняттю, яке ми розглядаємо, дає Roy Thomas Fielding: Архітектура програмного забезпечення — це абстракція елементів під час виконання програмної системи під час деякої фази її роботи. Система може складатися з багатьох рівнів абстракції та багатьох етапів роботи, кожна з яких має свою власну архітектуру програмного забезпечення. [1]

Архітектура програмного забезпечення визначається конфігурацією архітектурних елементів — компонентів, з'єднувачів і даних — обмежених у своїх відносинах, щоб досягти бажаного набору архітектурних властивостей. [1]

Враховуючи наведені визначення, можна зробити висновок, що архітектура програмного забезпечення системи відображає організацію або структуру системи та визначає те, як система поводиться. Система – це набір компонентів, що виконують певну функцію або набір функцій. Іншими словами, архітектура програмного забезпечення – це основа, на якій будується саме програмне забезпечення.

Ряд архітектурних рішень та компромісів, застосованих до цих рішень, впливають на якість, продуктивність, відмовостійкість та загальний успіх програмної системи. Нездатність врахувати загальні проблеми певного підходу до побудови системи та довгострокові наслідки ненадійно спроектованої архітектури, можуть позначитися на надійності застосунку.

Існує достатньо велика кількість архітектурних схем та принципів високого рівня, які здебільшого використовуються в сучасних програмних системах. Їх часто називають архітектурними стилями. Архітектура програмної системи рідко обмежується одним архітектурним стилем. Натомість, комбінування

різних стилів часто дозволяє створити більш досконалу систему, яку буде легко підтримувати та розвивати.

Тим не менш, зараз існує два кардинально різні підходи до побудови архітектури програмних систем: монолітний та мікросервісний. У кожного з них є свої переваги та недоліки. Проте, перш, ніж описувати кожний тип окремо, я б хотів зупинитися на загальних принципах та критеріях якісно спроектованої архітектури.

1.2. Критерії якісної архітектури

Як ми вже з’ясували до цього, в загальному випадку, немає однозначного визначення терміну «архітектура програмного забезпечення». Проте, є певні архітектурні властивості, які є бажаними, та які відрізняють продуману якісну архітектуру від невдалої.

Так, продумана архітектура, це, в першу чергу, та, яка робить процес розробки програмного забезпечення вигідним. Тобто дозволяє більш просто та ефективно як власне розробляти програмне забезпечення, так і в подальшому його супроводжувати та підтримувати. Програмній системі, з якісно продуманою архітектурою, простіше еволюціонувати, підлаштовуючись під вимоги ринку та замовників; її простіше змінювати та розширювати. Також такий підхід дозволяє спростити тестування та його підтримку, а також відлагоджування самого застосунку. В кінці кінців, продумана архітектура сприяє простішому та швидшому розумінню розроблювальної системи, що є дуже важливим, коли люди, які над нею працюють, можуть періодично змінюватися.

Тобто, враховуючи такі судження, модна цілком успішно визначити обмежений список критеріїв, які є універсальними, та показують, наскільки якісною та продуманою є архітектура.

1.2.1. Ефективність програмного забезпечення

Головною вимогою до будь-якого програмного забезпечення є те, що воно повинно виконувати задачі та функції, заради яких було створено. При чому, воно повинно виконувати ці функції ефективно, та за різних умов. Тобто програмне забезпечення повинно бути безпечним, відмовостійким (надійним), повинно виконувати свої задачі з високою продуктивністю, повинно вміти витримувати за працювати за збільшених навантажень.

1.2.2. Гнучкість програмного забезпечення

Як вже було сказано раніше, будь-яке програмне забезпечення з часом потребує змін та доповнень. Це викликано тим, що вимоги до системи змінюються та доповнюються з часом. Гнучкість системи дозволяє мінімізувати кількість помилок та багів, які можуть виникнути під час цього процесу, а, отже, зробити дане програмне забезпечення набагато більш конкурентоспроможним, оскільки швидше та з меншими трудовитратами дозволить внести необхідні зміни.

Щоб досягти такого ефекту, починаючи з етапу проектування потрібно думати над тим, яка архітектура виходить, та оцінювати її з точки зору того, що, можливо, доведеться вносити правки та перероблювати певні компоненти. Якщо дизайн певного компонента системи виявиться невдалим, або перестане відповідати вимогам, які ставляться перед програмним забезпеченням, має бути можливість такий компонент змінити або переробити. При цьому, така зміна не повинна впливати на роботу інших компонентів.

При розробці архітектурних рішень, треба закладати можливість, що вони були помилковими. Тому наслідки таких рішень маються бути певною мірою обмежені.

Мета якісною архітектури – дозволити відстрочувати та відкладати рішення. [2]

1.2.3. Розширюваність програмного забезпечення

Розширюваність системи. Можливість додавати в систему нові сутності та функції, не порушуючи її основної структури. На початковому етапі в систему має сенс закладати лише основний та найнеобхідніший функціонал. Але при цьому архітектура повинна дозволяти легко нарощувати додатковий при необхідності. Причому так, щоб внесення найімовірніших змін вимагало найменших зусиль.

Даний принцип називається «YAGNI» (англ. You Aren't Going to Need It — «Вам це не знадобиться») — процес і принцип проєктування, при якому основною метою та цінністю є відмова від додавання функціональності, в якій немає безпосередньої потреби. Цей принцип варто застосувати ще на етапі формування юзкейсів та UX-тестування на прототипах, оскільки це дозволяє визначити потрібні для користувачів елементи програми та усунути непотрібні функції з вимог до початку їхньої розробки. [5]

Іншими словами: Має бути можливість розширити/змінити поведінку системи без зміни/переписування вже існуючих елементів системи.

Це означає, що програмне забезпечення слід проектувати так, щоб зміна його поведінки та додавання нової функціональності досягалося б за рахунок написання нового коду (розширення), і при цьому не доводилося б змінювати вже існуючий код. У такому разі поява нових вимог не спричинить

модифікацію вже існуючої логіки. Натомість вони можуть бути реалізовані насамперед за рахунок розширення програмної системи. Саме цей принцип є основою «плагінної архітектури» (Plugin Architecture).

1.2.4. Процес розробки повинен масштабуватися

Архітектура програмного застосунку повинна бути розроблена таким чином, щоб дозволяти паралельну одночасну роботу над різними частинами системи. Таке розпаралелення розробки потрібно для того, щоб можна було скоротити час розробки за рахунок залучення до неї нових людей. І всі ці люди могли працювати над програмним забезпеченням одночасно, не заважаючи одне одному.

1.2.5. Код повинен підлягати тестуванню

Тестування коду допомагає зменшити кількість помилок та збільшити надійність програмного застосунку. Можна сказати, чим легше тестувати код, тим більш надійним він, найімовірніше, буде.

Крім того, вимога до коду, щоб він краще підлягає тестуванню, може автоматично вести до поліпшення архітектурного дизайну системи. Навіть більше, така вимога дозволяє оцінити загальну якість коду.

Використовуючи такий підхід до проектування, навіть не написавши жодного тесту, можна з доволі солідною вірогідністю сказати, наскільки якісний дизайн був використаний для створення того чи іншого класу. Це зумовлено тим, що для тестування потрібно виокремити певний модуль від його залежностей, і якщо це зробити складно або неможливо, то такий дизайн не є оптимальним.

На основі даного принципу виникла ціла технологія розробки програмного забезпечення – керована тестами розробка або розробка через тестування (TDD – test driven development).

Ця методологія розробки передбачає короткі ітерації, які складаються з написання тестів, а вже потім основної логіки програмного продукту. Мета такого підходу – розробити код, який пройде тести, що пришвидшує процес внесення змін.

1.2.6. Можливість повторного використання.

Архітектура програмної системи має бути спроектована таким чином, щоб її саму або її частини можна було повторно використати під час розробки іншої системи.

1.2.7. Код повинен бути структурований

Добре структурований, та зрозумілий код. Над програмою, як правило, працює багато людей — одні йдуть, приходять нові. Після написання супроводжувати програму теж, як правило, доводиться людям, які не брали участь у її розробці. Тому хороша архітектура повинна давати можливість легко і швидко розібратися в системі новим людям. Проект має бути добре структурований, не містити дублювання, мати добре оформлений код та бажано документацію. І наскільки можна у системі краще застосовувати стандартні, загальноприйняті рішення звичні програмістів. Чим екзотичніша система, тим складніше її зрозуміти іншим (Принцип найменшого подиву - Principle of least astonishment).

1.3. Монолітна архітектура

Монолітна архітектура - це традиційна модель програмного забезпечення, складається з єдиного модуля, що працює автономно і не залежить від інших додатків. Монолітом часто називають щось велике і неповоротке, і ці два слова добре описують монолітний підхід до проектування архітектури програмного забезпечення.

Монолітна архітектура — це окрема велика обчислювальна мережа з єдиною базою коду, в якій об'єднані всі бізнес-завдання. Щоб внести зміни в таку програму, необхідно оновити весь стек технологій, які використовуються системою, через кодову базу, а також створити та розгорнути оновлену версію інтерфейсу, що знаходиться на стороні служби. Це обмежує роботу з оновленнями, та потребує багато часу на їх впровадження.

Моноліти зручно використовувати на початкових етапах розробки програмних проектів, щоб полегшити їх розгортання, та не витрачати надто багато зусиль при керуванні кодом. Це дозволяє одразу випускати все, що є в монолітному додатку.

1.3.1. Переваги монолітної архітектури

Організації можуть отримати вигоду як з монолітної архітектури, так і з мікросервісної, в залежності від різних факторів. При використанні монолітної архітектури зручно створювати програми на основі однієї кодової бази, тому її основна перевага полягає у швидкості розробки.

До переваг монолітної архітектури можна віднести такі особливості:

- Простота розгортання. Зазвичай, весь монолітний застосунок знаходиться в одному каталозі, або й взагалі міститься в одному виконуваному файлі, що значно спрощує розгортання.

- Розробка. Використання однієї кодової бази під час розробки також спрощуй цей процес.
- Продуктивність. Шлях виконання атомарної бізнес-операції в монолітному застосунку, з використанням однієї кодової бази, може бути значно коротшим, ніж аналогічний запит в системі розподілених мікросервісів.
- Спрощене тестування. Монолітна програма є єдиним централізованим модулем, тому наскрізне тестування можна проводити швидше, ніж при використанні розподіленої системи з різними модулями.
- Зручне відлагоджування. Весь код знаходиться в одному місці, завдяки чому стає легше виконувати запити та знаходити проблеми.

1.3.2. Недоліки монолітної архітектури

Як і у випадку з Netflix, монолітні програми працюють досить ефективно доти, доки вони не стають занадто великими і не викликають проблем із масштабуванням. Щоб внести невелику зміну в одну функцію, необхідно виконати компіляцію та тестування всієї платформи, що суперечить agile-підходу, якому віддають перевагу сучасні розробники.

До недоліків монолітної архітектури можна віднести такі особливості:

- Зниження швидкості розробки на довгій дистанції. Великий монолітний комплекс програмних систем ускладнює та уповільнює розробку.
- Масштабованість. Неможливо масштабувати окремі компоненти в середині моноліту.
- Надійність. Помилка в одному модулі може призвести до відмови в роботі іншого модуля.

- Складність упровадження нових технологій. Будь-які зміни в інфраструктурі програмного забезпечення або в мові розробки впливають на систему цілком, що часто призводить до збільшення вартості розробки та підвищення тимчасових витрат.
- Недостатня гнучкість. Можливості монолітних програм обмежені використовуваними технологіями.
- Розгортання. При внесенні невеликої зміни буде потрібно повторне розгортання всього монолітного програмного забезпечення.

1.4. Мікросервісна архітектура

Мікросервісна архітектура (або просто «мікросервіси») є методом організації архітектури програмного забезпечення, який полягає в паралельному розгортанні кількох служб, що працюють паралельно. Ці служби мають власну, виокремлену бізнес-логіку та базу даних з конкретною метою.

Оновлення, тестування, розгортання та масштабування виконуються всередині кожної служби. Мікросервіси розбивають великі завдання, притаманні кожному конкретному бізнесу, на кілька незалежних кодових баз.

Мікросервіси не знижують складність, але вони роблять будь-яку складність видимою і більш керованою, поділяючи великі завдання на дрібніші процеси, які функціонують незалежно один від одного і роблять внесок у загальний результат.

Використання мікросервісів часто тісно пов'язане з DevOps, оскільки вони лежать в основі методології безперервного постачання, яка дозволяє командам швидко адаптуватися до вимог користувачів, які постійно змінюються та оновлюються.

1.4.1. Переваги мікросервісної архітектури

Мікросервіси не є чарівною паличкою, але вони вирішують низку проблем, з якими стикаються компанії під час розростання, в процесі розвитку програмного забезпечення. Оскільки архітектура мікросервісів складається з модулів, що незалежно працюють, кожний сервіс можна розробляти, оновлювати, розгортати і масштабувати окремо від інших. Оновлення можна виконувати частіше, що збільшує надійність, час безперебійної роботи та продуктивність програмного забезпечення.

До переваг програмного забезпечення, яке використовує мікросервісний підхід, можна віднести прискорене розгортання, можливість аварійного відновлення, зниження витрат та підвищення продуктивності. Завдяки цьому можна швидше досягати поставленої перед системою мети.

Якщо говорити про команди, які займаються розробкою програмного забезпечення, то мікросервісний підхід спрощує для них оновлення коду та прискорює цикли релізу, завдяки безперервній інтеграції та безперервному розгортуванню (CI/CD). Команди можуть поекспериментувати з кодом і, якщо щось піде не так, повернутись до попередньої версії.

Отже, мікросервісний підхід в розробці програмного забезпечення надає такі переваги:

- Гнучкість. Мікросервіси дозволяють впровадити гнучкі методи розробки серед невеликих команд, які регулярно займаються розгортуванням.
- Гнучке масштабування. Коли мікросервіс досягає граничного навантаження, можна швидко виконати розгортання нових екземплярів цього сервісу в супутньому кластері, та знизити навантаження. Це дозволяє працювати без збереження стану, а клієнти розподілені на різних екземплярах застосунку. З таким підходом можлива підтримка екземплярів сервісу значно більшого розміру.
- Безперервне розгортання. Мікросервісний підхід дозволяє пришвидшити цикли релізу. Якщо, для прикладу, з монолітним підходом до проектування, оновлення виходило випускати раз на тиждень, то з мікросервісним підходом частоту випуску оновлень можна збільшити до, приблизно, двох-трьох раз на день.

- Легкість обслуговування та тестування. Команди можуть експериментувати з новими функціями та повертатися до попередньої версії, якщо щось не працює. Це спрощує оновлення коду та прискорює випуск нових функцій на ринок. Крім того, в окремих службах легко знаходити та виправляти помилки та баги.
- Незалежне розгортання. Мікросервіси є окремими модулями, тому їх застосування дозволяє легко та швидко виконувати незалежне розгортання окремих функцій.
- Гнучкість технологій. При використанні мікросервісного підходу до проектування програмних рішень, команди можуть вибирати інструменти з урахуванням компетенції членів команди.
- Висока надійність. Розгортання змін для одного конкретного сервісу не впливає та працездатність всієї системи. Тому можливі перебої в роботі цього конкретного сервісу не вплинуть на всю систему мікросервісів.
- Задоволені команди. Команди, що працюють з мікросервісами, можуть бути більше задоволені своєю роботу, завдяки автономності і можливості самостійно створювати і розгортати застосунки, не чекаючи схвалення pull-запиту протягом декількох тижнів.

1.4.2. Недоліки мікросервісної архітектури

Перехід від невеликої кількості монолітних кодових баз до множини розподілених систем і сервісів, які складатимуть основу програмних систем, може викликати непередбачені складнощі.

Мікросервіси можуть зробити процес розробки складнішим і призвести до його швидкого та некерованого зростання. Іноді буває складно визначити, як різні компоненти пов'язані між собою, хто займається розробкою конкретного

програмного компоненту, або як уникнути втручання в роботу залежних компонентів.

Часто, мікросервісний підхід дозволяє створити спільні функціональні можливості, які будуть фундаментом для майбутніх програмних систем та продуктів. Проте, якщо розробляти планується тільки один програмний застосунок, то мікросервіси можуть не знадобитися.

До недоліків мікросервісів можна віднести такі особливості:

- Розростання процесу розробки. Мікросервісний підхід в проектуванні ускладнює роботу по налагодженню взаємодії між сервісами, порівняно з монолітною архітектурою, оскільки у різних місцях виникає дедалі більше сервісів, створених кількома командами. Якщо розростання не контролюється належним чином, воно призводить до уповільнення розробки та зниження операційної ефективності.
- Експонентне зростання витрат на інфраструктуру. Кожен новий мікросервіс може мати індивідуальний стек технологій, які потребують розгортання інфраструктури мікросервісу відповідним чином, що відображається на вартість використаних ресурсів для тестування, розгортання, інфраструктури хостингу, інструментів моніторингу тощо.
- Додаткові організаційні витрати. Командам потрібний додатковий рівень комунікації та співпраці, щоб координувати роботу над оновленнями та інтерфейсами взаємодії мікросервісів.
- Проблеми при налагодженню. Кожен мікросервіс записує логи окремо від інших, що ускладнює налагодження. Крім того, додаткові труднощі можуть виникати у тому випадку, коли один бізнес-процес потребує задіяння кількох сервісів.

- Відсутність стандартизації. При переході до мікросервісної архітектури застосунку від монолітною, може виникнути ситуація, коли розширюється список мов, які використовуються для реалізації конкретних сервісів; змінюються стандарти логування між різними сервісами та засобів їх моніторингу.
- Відсутність прозорості в питаннях відповідальності за конкретні сервіси. У міру появи нових сервісів, збільшується і кількість команд, що працюють над ними. Згодом стає складно визначити, які сервіси команда може використовувати і до кого слід звертатись за підтримкою.

Розділ 2. Проектування та розробка системи

2.1. Опис складових системи та використаних технологій для розробки

2.1.1. Spring Data

Місія Spring Data полягає в тому, щоб забезпечити знайому та послідовну модель програмування на основі Spring для доступу до даних, зберігаючи при цьому особливі риси основного сховища даних. [4]

Spring Data JPA, частина великого сімейства Spring Data, дозволяє легко реалізувати репозиторії на основі JPA. Цей модуль додає розширену підтримку рівня доступу до даних, який спирається на JPA (Java Persistent API). Це спрощує створення програм, які спираються на фреймворк Spring та використовують технології доступу до даних. [3]

2.1.2. Flayway

При розробці програмних застосунків майже завжди виникає необхідність контролювати версії бази даних. Навіть одне поле, яке розробники додали до певного об’єкту бази даних, може зламати всю систему. Так стається, тому що інші частини системи можуть все ще користуватися старими версіями бази даних, або не знати, як користуватися новими. Також проблема версіонування бази даних виникає, коли для продакшену використовується одна версія, а для розробки інша, до прикладу, новіша або змінена. Така розбіжність у версіях бази даних є нормою для систем, що розробляються.

Для вирішення проблеми версіонування даних в базі даних існують різні інструменти, які дозволяють створювати міграції та керувати ними. Одним з найпоширеніших інструментом серед них є Flyway.

Flyway дозволяє оновлювати базу даних від однієї версії до іншої за допомогою міграцій. Ми можемо писати міграції або на SQL, створюючи міграції напряму до бази даних, або на Java, з підтримкою розширених перетворень бази даних.

Міграції можуть бути версійними або повторюваними. Перший тип має унікальну версію і застосовується рівно один раз. Натомість повторювані міграції версій не мають. Вони (повторно) застосовуються щоразу, коли їхня контрольна сума змінюється.

У межах одного запуску міграції бази даних, повторювані міграції завжди застосовуються в останню чергу, після того, як були виконані нерозглянуті версійні міграції. Повторювані міграції застосовуються в порядку їх опису. Для однієї міграції всі перетворення виконуються в рамках однієї транзакції бази даних.

Приклад міграції наведений в додатку Г.

2.2. Опис мікросервісів

Система вступу в НаУКМА – це мікросервісна розподілена система, яка складається з багатьох компонентів, кожен з яких відповідає за певний бізнес процес.

Зараз система складається з таких сервісів:

- account-server
- user-info
- course-work
- document-generator
- document-management
- notifications
- storage-management
- ui-constructor
- ui-server

Давайте детальніше зупинимося на кожному з них.

2.2.1. Account-server

Під час розробки мікросервісної архітектури можуть виникати певні проблеми з безпекою всієї системи. Наприклад, до таких проблем відноситься керування ролями користувачів на різних серверах.

Імплементація такої логіки, як авторизація та автентифікація користувачів на сервері є достатньо схожою для кожного сервера, та рідко відрізняється. Можна навіть сказати, що логіка видачі прав користувачеві на сервері є однаковою, та не змінюється від серверу до серверу.

Це дозволяє винести імплементація такої логіки, в окремий мікросервіс, який буде обслуговувати запити на авторизацію та автентифікацію користувачів з усіх інших серверів у системі.

Проте, такий підхід створює певні додаткові проблеми. Вони проявляються в тому, що сервер, до якого звертаються із запитом про ролі користувача, також повинен знати, хто саме до нього звертається, та чи має він на це право.

Отже, account-server працює таким чином. У кожного нового сервіса в системі є свій власний логін та пароль, з яким він звертається до сервісу керування акаунтами. Якщо такий сервіс зареєстрований в системі, він отримує свій згенерований JWT-токен, який може використовувати для подальших запитів до account-server.

Такий підхід дозволяє винести питання керування ролями користувачів в одне окреме місце, що спрощує підтримку всієї системи, та пришвидшує її налаштування.

Отже, коли користувач звертається до певного сервісу із системи, він авторизується та/або автентифікується з допомогою account-server, останній, в свою чергу, співставляє імейл користувача з його ролями на сервісах та повертаю цю інформацію.

Також слід зазначити, що для отримання імейлу користувача, використовується сервіс Office365. Це домомагає впевнитися, що імейл користувача справжній, та такий користувач існує.

2.2.2. User-info

Даний сервіс створювався за подібною логікою, але для виокремлення та зберігання інформації про користувачів системи.

До такої інформації відносяться особисті дані людини, її посада в університеті (якщо вона працівник), дані, які стосуються студента (якщо людина студент): роки навчання, факультет, курс, спеціальність тощо.

Також хочу описати певні деталі роботи даного сервісу.

2.2.2.1. Опис роботи сервісу

В мікросервісі використовується система фільтрації та сортування даних.

Всі запити на перегляд очікують на вхід список полів, які будуть представлені у відповіді. Конкретний список полів можна дізнатися в документації для конкретної сутності.

Також всі запити на отримання списку `getListOf<Entity>`, крім списку полів, очікують на вхід **json** стрічку з обмеженнями (фільтрами). Та повертають `PageListMapResponce`. Спільні для всіх сутностей фільтри можна переглянути в додатку А.

Всі запити до сервіса повертають код 200, та:

- результат, якщо запит виконано успішно;
- ResponseError (Додаток Б) в полі error, якщо на одному з етапів обробки запиту була згенерована помилка.

Наприклад, щоб додати інформацію про студента на сервіс, потрібно відправити PUT запит на `/api/student`, заповнивши відповідні поля, та мати відповідний дозвіл або роль. Серед таких полів є імейл, ім’я та прізвище, курс, факультет тощо.

Приклад такого запиту дивіться в додатку В.

2.2.3. Course-work

Даний сервіс створено для виокремлення та зберігання даних про курсові роботи.

Дані, які зберігаються на сервісі включають в себе назву курсової, тему, викладача, який запропонував дану тему, навчальні роки, опис теми, дату створення, кафедру тощо.

Висновки

Було досліджено різні методи побудови архітектури програмного забезпечення; критерії якості розробленої архітектури на основі вимог до системи; розглянуто різні підходити до створення дизайну архітектури програмного забезпечення; проаналізувано монолітну та мікросервісну архітектури; порівняні дані підходи.

Був обґрунтуваний вибір мікросервісної архітектури для розробки системи вступу в НаУКМА та описані принципи побудови даного мікросервісного застосунку.ь

Список використаних джерел

1. Fielding R. Architectural Styles and the Design of Network-based Software Architectures : дис. докт. філос. наук / Fielding Roy Thomas, 2000.
2. Martin R. Clean Architecture A Craftsman's Guide To Software Structure And Design [Електронний ресурс] / Robert C. Martin. – 2017. – Режим доступу до ресурсу: https://archive.org/details/CleanArchitecture/mode/2up.\
3. Бібліотека Spring Data; документація [Електронний ресурс] – Режим доступу до ресурсу: https://spring.io/projects/spring-data-jpa
4. Spring Data documentation [Електронний ресурс] – Режим доступу до ресурсу: https://spring.io/projects/spring-data.
5. YAGNI principe [Електронний ресурс] – Режим доступу до ресурсу: https://web.archive.org/web/20150112003354/http://www.xprogramming.com/Practices/PracNotNeed.html
6.

Додатки

Додаток А

```
{
  "limit": 10,
  "offset": 0,
  "order_by": "id",
  "order_direction": "ASC"
}
```

limit - кількість відповідей на одній сторінці
offset - офсет, від якого рахується ліміт
order_by - за яким полем сортувати (за замовчуванням - *id*) **order_direction**:

- ACS
- DESC

Додаток Б

```
class ResponseError {
    private int code;
    private String message;
    private List<String> errors;
}
```

code - HTTP status code:

HTTP status	Error thrown by server
400	`WrongRestrictionExcept`
403	`ForbiddenException`
404	`NoSuchEntityException`
409	`ValidationException`
500	`ServiceErrorException`

message - опис помилки, який може бути локалізований використовуючи код повідомлення наявний в `messages_<lang>.properties` . *Проте зараз, схоже, деякі класи повертають тільки своє ім'я, як* `ValidationException`*, або системну помилку, як* `NoSuchEntityException`

errors - список детальних описів помилок, які виникли під час обробки запиту. Опис, як і повідомлення, локалізується, використовуючи коди помилок (коди помилок описані в документацій на відповідний ендпоінт).

Додаток В

Roles:

- `Admin`
- `Manager`
- `Student`

Permission:

- `CREATE_ANY_STUDENT_INFO`
- `CREATE_STUDENT_INFO` && User email == creating user email

Request example:

```
{
  "email": "example@example.com",
  "firstName": "John",
  "lastName": "Smith",
  "course": 1,
  "facultyId": 1
}
```

Додаток Г

Приклад міграції схеми бази даних:

V1__user_info.sql

```
CREATE TABLE user_info (
  id SERIAL PRIMARY KEY,
  email VARCHAR(320) NOT NULL UNIQUE,
  first_name VARCHAR(250) NOT NULL,
  last_name VARCHAR(250) NOT NULL
);
```