НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ *Сергій СТІРЕНКО*

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення»

Виконав:
студент IV курсу, групи ІП-62
Линник Володимир Вікторович _____

Керівник:
Доцент, кандидат технічних наук
Порєв Віктор Миколайович _____

Рецензент:
Професор, доктор технічних наук
Сімоненко Валерій Павлович _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ *Сергій СТІРЕНКО*

«___»_____20__ р.

ЗАВДАННЯ

на дипломний проєкт студенту

Линнику Володимиру Вікторовичу

1. Тема проєкту _*«Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення»*_

керівник проєкту Порєв Віктор Миколайович, доцент, затверджені наказом по університету від «_*07*_» __*травня*___ 2020 р. №_1081-с_

2. Термін подання студентом проєкту _____

3. Вихідні дані до проєкту _*технічне завдання, теоретичні дані.*_

4. Зміст пояснювальної записки: _*аналіз та огляд існуючих систем, опис математичної задачі та алгоритму роботи, вибір технологій та розробка програмного забезпечення системи, демонстрація роботи та можливостей системи.*_

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Симоненко В.П.		

7. Дата видачі завдання _*01.09.2019 року*_

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	*Затвердження теми роботи*	*1.08.2019*	
2.	*Вивчення та аналіз завдання*	*15.09.2019 – 15.03.2020*	
3.	*Розробка архітектури та загальної структури систем*	*15.03 – 25.03.2020*	
4.	*Розробка структур окремих сервісів*	*25.03.2020 – 05.04.2020*	
5.	*Програмна реалізація системи*	*05.04.2020 – 15.04.2020*	
6.	*Оформлення пояснювальної записки*	*15.04.2020 – 20.05.2020*	
7.	*Передзахист*	*26.05.2020*	
8.	*Захист*	*25.06.2020*	

Студент Володимир ЛИННИК _____

Керівник Віктор ПОРЄВ _____

Анотація

В бакалаврській дипломній роботі реалізовані принципи побудови застосунків з мікросервісною архітектурою , призначені для моделювання, навчання, тестування та використання нейронних мереж прямого поширення.

Система підтримує роботу з табличним представленням вхідних даних для навчання та тестування нейромереж, має можливість обрахування вхідних параметрів та нормалізації даних, спираючись на сукупність вхідних даних, при чому користувач має можливість динамічної зміни обраних елементів вхідного шару нейромережі, вибору кількості внутрішніх шарів, функцію активації кожного шару, кількість епох навчання та динамічне визначення вихідного шару. До того ж, можливе створення декількох варіантів налаштування нейромережі для одних вхідних даних, що дозволяє користувачу провести навчання кожного варіанту, а потім отримати аналітичні дані стосовно якості роботи нейромереж. Система була розроблена з урахуванням принципів побудови мікросервісних застосунків , використовуючи Java та фреймворку Spring для сервісів, що забезпечують збереження, виведення та початкову обробку даних, а також використовуючи Python для сервісу навчання та тестування нейромереж.

Annotation

In this work for a Bachelor's Degree it was implemented the principles of building applications with microservice architecture, designed for modeling, training, testing and using feedforward neural networks.

The system supports tabular input data for training and testing neural networks, has the ability to calculate input parameters and normalize data based on a set of input data, while the user has an ability to dynamically change selected elements of the input layer of the neural network, set the number of internal layers, the number of learning epochs and dynamically define the source layer. In addition, it is possible to create several options for setting up a neural network for a single input data, which allows the user to train each option, and then obtain analytical data on the quality of neural networks. The system was designed based on the principles of building microservice applications, using Java and Spring framework for services that provide storage, output and initial data processing, as well as using Python for neural network training and testing service.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проєкт	2	
2	А4	ІЛАЦ.467100.001 ВП	Відомість проекту	1	
3	А4	ІЛАЦ.467100.002 ТЗ	Технічне завдання	4	
4	А4	ІЛАЦ.467100.003 ПЗ	Пояснювальна записка	75	
5	А3	ІЛАЦ.467100.004 Д1	Функціональна схема	1	
6	А3	ІЛАЦ.467100.005 Д2	Принципова схема	1	
7	А3	ІЛАЦ.467100.006 Д3	Структурна схема	1	
8	А4	ІЛАЦ.467100.007 Д4	Лістинг	19	

					ІЛАЦ.467100.001 ВП	
Зм.	*Арк.*	*№ документу*	*Підпис*	*Дата*		
Розробив		*Линник В.В.*			*Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення Відомість дипломного проекту*	Літ. / Аркуш 5 / Аркушів 1
Перевірив		*Порєв В.М.*				
						НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, гр. ІП-62
Н. Контр.		*Симоненко В.П.*				
Затверд.						

ТЕХНІЧНЕ ЗАВДАННЯ

до дипломної роботи
освітньо-кваліфікаційного рівня «бакалавр»

на тему: “ Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення ”

Київ – 2020 року

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку курсу «Веб-програмування». Область застосування: практичне використання людьми для використання нейронних мереж та вдосконалення навичок їх побудови.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп'ютерної інженерії», затверджене кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка серверної частини системи, побудованої з дотриманням принципів мікросервісної архітектури для моделювання нейромереж широкого призначення.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література з теорії і практики програмування, публікації та наукові статті в Інтернеті з даних питань.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розроблюваного продукту

- Можливість динамічної конфігурації апаратної частини системи.
- Незалежність – система повинна мати програмну автономність і не залежати від встановленого програмного забезпечення.
- Швидка обробка даних з мінімальними похибками

5.2. Вимоги до програмного забезпечення

- Операційна система MS Windows 98 та новіші версії, Linux-дистрибутиви та Mac OS.
- Доступ до мережі Інтернет.
- Пошуковий браузер.

5.3. Вимоги до апаратної частини

- Комп’ютер на базі процесора Intel або AMD мінімальної потужності.
- Оперативної пам’яті не менше 2 Гбайт.

6. ЕТАПИ РОЗРОБКИ

	Дата
Вивчення літератури	10.03.2020
Складання і узгодження технічного завдання	01.04.2020
Створення модулів системи, що розробляється	10.04.2020
Тестування окремих модулів системи	20.04.2020
Доопрацювання, налагодження і виправлення помилок	04.05.2020
Оформлення документації дипломної роботи	17.05.2020

ПОЯСНЮВАЛЬНА ЗАПИСКА
до дипломного проєкту
на тему: «Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення»

Київ – 2020 року

ЗМІСТ

					ІЛАЦ.467100.003 ПЗ		
Зм.	*Арк.*	*№ документу*	*Підпис*	*Дата*			
Розробив		*Линник В.В.*			*Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення Пояснювальна записка*	Літ.	Аркуш 1 / Аркушів 75
Перевірив		*Порєв В.М.*					
Н. Контр.		*Симоненко В.П.*				*НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, гр. ІП-62*	
Затверд.							

ПЕРЕЛІК СКОРОЧЕНЬ

OS	Open-source
MSA	MicroService Architecture (мікросервісна архітектура)
SOA	Service-Oriented Architecture (сервіс орієнтована архітектура)
OSS	Open source software (програмне забезпечення з відкритим кодом)
ESB	Enterprise service bus (інтеграційна або сервісна шина)
API	Application Programming Interface (набір інтерфейсів для взаємодії різнотипного програмного забезпечення)
DDD	Domain-Driven Design (набір принципів та схем, направлений на створення оптимальних систем об’єктів)
БД	База даних
СУБД	Система управління баз даних
ООП	Об’єктно-орієнтоване програмування
JVM	Java Virtual Machine (віртуальна машина Java)
XML	eXtensible Markup Language (розширювана мова розмітки)
REST	Representational State Transfer, архітектурний стиль для побудови API
SOLID	Абревіатура, п’ять базових принципів ООП та дизайну програмних застосунків
MVC	Model View Controller, схема розподілу даних в застосунках
CRUD	Акронім, що позначає чотири базові операції роботи з БД (create, read, update, delete)
POM	Project Object Model, фундаментальний елемент роботи в Maven, мова розмітки, що базується на XML

ВСТУП

Нейромережі – це однин із напрямків штучного інтелекту, ціль якого – моделювання аналітичних механізмів, що відбуваються в мозку людини. Основні задачі, що може вирішити типова нейромережа – це класифікація, передбачення та розпізнавання на основі отриманих даних. Нейромережі здатні самостійно навчатися та розвиватися на основі власного досвіду та скоєних помилок. В сучасному світі нейромережі набирають все більшої популярності, кожного дня нейромережі привертають увагу все більше розробників, що хочуть залучити нейромережі до їх власних проєктів. Саме на даному етапі в багатьох виникають труднощі, адже точність та коректність роботи мережі залежить від багатьох параметрів. Зокрема, неможливо створити одну універсальну «ідеальну» нейромережу для вирішення будь-якого завдання.

Об’єктом дипломної роботи є комплексний застосунок для побудови і аналізу нейромереж. Вибір об’єкту дослідження зумовлений відсутністю на момент написання дипломної роботи сервісу, який би дозволив не лише побачити результати роботи нейромереж, що були створені різноманітними розробниками, а й долучитися до їх створення власноруч без необхідності вивчення великої кількості бібліотек різних мов програмування. А також, мати змогу використовуючи різні параметри отримати наочне порівняння якості роботи нейромережі та обрати найбільш оптимальний варіант її реалізації.

Предметом дослідження є створення веб-застосунку для побудови та аналізу нейромереж загального призначення з графічного інтерфейсу користувача та з використанням переваг мікросервісної архітектури для аналізу та розподілу навантаженості серверів, а також з можливістю швидкого розширення їх потужності.

Метою роботи є:

1) Огляд доцільності використання мікросервісної архітектури для застосування у проєкті.
2) Спроба створення універсальної системи для побудуви та тестування різноманітних нейромереж.
3) Перегляд можливих реалізацій систем з мікросервісною архітектурою.

Були поставлені наступні завдання:

1) Перегляд нейромереж широкого призначення, розглянути відомі приклади застосування.
2) Аналіз стану розвитку сучасних веб-технологій для побудови систем з мікросервісною архітектурою.
3) Реалізація серверної частини сервісу та алгоритму створення, налагодження, навчання та тестування нейромереж.
4) Розробка графічного інтерфейсу користувача.
5) Налагодження цілісної системи як взаємодії та передачі даних між окремими її компонентами

РОЗДІЛ 1

АНАЛІЗ ПРОБЛЕМАТИКИ ДЛЯ ПОБУДОВИ НЕЙРОМЕРЕЖ ЗАГАЛЬНОГО ПРИЗНАЧЕННЯ

1.1 Теоретичні відомості

Структура нейромереж прийшла в світ програмування взявши свій аналог з біології. Тому, нейронна мережа [1] – це послідовність нейронів, що з'єднані між собою синапсами. Завдяки даної структури машина отримала здатність аналізувати і навіть запам'ятовувати інформацію, після чого її відтворювати. Іншими словами, нейромережа – це машинна інтерпретація мозку людини, в якому знаходиться мільйони нейронів, що передають інформацію у вигляді електричних імпульсів.

Нейромережі застосовують для вирішення складних задач, що потребують аналітичних обчислень, що подібні до тих, що виконує людський мозок. Найбільш поширеними серед них є:

1) Класифікація – розподілення даних по деяким характеристикам. Прикладом реалізації даного застосування є банківська система, що вирішує чи надавати клієнту кредит, спираючись на певні дані про самого клієнта.
2) Передбачення – можливість системи передбачити наступний крок. Для прикладу передбачення наступного росту чи падіння курсу валют, спираючись на останні дані стосовно економічного стану в країні.
3) Розпізнавання – виявлення сутності об'єкту, виходячи з певних його характеристик. На даний момент це є найбільш популярний спосіб використання нейромереж, його приклад застосування можливо знайти навіть в мобільному телефоні, що знаходить обличчя людини

при наведенні на неї, для пошуку по фото пошуковими системами також задіяні нейромережі для розпізнавання об’єктів що знаходяться на фото.

На даний момент існує досить велика кількість різних нейромереж, що мають різну структуру та цілі, тому доцільно буде провести їх класифікацію.

1.2. Класифікація нейромереж

В загальному, нейромережі класифікують за наступними критеріями:

1) За характером навчання.
2) За способом налаштування вагових коефіцієнтів.
3) За типом вхідної інформації.
4) За моделлю, що застосовується для нейромережі.

Розглянемо детальніше кожен з критеріїв.

1.2.1. Класифікація за характером навчання

За характером навчання нейромережі поділяють на два види: нейронні мережі, що використовують навчання з «вчителем», а також нейромережі, що використовують навчання без «вчителя».

Тобто, навчання з «вчителем» передбачає те, що при навчанні нейромережі для кожного вхідного вектору значень існує цільовий вектор, що являє собою потрібний вихід даної нейромережі. Разом дані вектори називають навчальною парою. Зазвичай, нейромережа навчається на певній кількості даних навчальних пар. Для цього на обробку нейромережі відправляються дані на обробку, а після обчислень відбувається порівняння отриманого результату з очікуваним, після чого відбувається зміна вагових коефіцієнтів для мінімізації похибки обчислень. Всю множину навчальних пар проходять послідовно, підлаштовуючи вагові коефіцієнти на кожній ітерації

до тих пір, поки похибка по всьому навчальному масиву не досягне прийнятного рівня.

Навчання нейромереж без вчителя є більш наближеною до навчання живого організму в реальному світі з точки зору біологічного походження штучних нейромереж. Даний тип нейромереж не потребує множини навчальних пар, а відповідно не потребує порівняння з зумовленими заранні ідеальними результатами. Навчальна множина складається лише з вхідного вектору значень та не зрівнюється результат. Основною метою є підбір вагових коефіцієнтів таким чином, щоб для схожих вибірок даних була можливість отримати схожий результат, тому основною ціллю даної нейромережі є виділення схожих характеристик вхідних даних та об'єднання їх в класи.

1.2.2 Класифікація за способом налаштування вагових коефіцієнтів

За класифікацією нейромереж по способу налаштування вагових коефіцієнтів нейромережі поділяються на мережі з фіксованими зв'язками та мережі з динамічними зв'язками.

Для мереж зі статичними зв'язками вагові коефіцієнти встановлюються відразу при ініціалізації нейромережі, та не змінюються впродовж всього життєвого циклу нейромережі. Саме тому для даного типу нейромережі не властивий процес навчання, оскільки головна ціль навчання нейромереж – зміна вагових коефіцієнтів. Даний тип нейромереж найкраще підходить вже для готової нейромережі, в якій вагові коефіцієнти приймають найбільш оптимальні значення.

Мережі з динамічними значеннями вагових коефіцієнтів в більшості випадків потребують процесу навчання для корегування початкових значень вагових коефіцієнтів.

1.2.3 Класифікація за типом вхідної інформації

За типом вхідних даних нейромережі поділяються на аналогові та двійкові. Для аналогових нейромереж вхідні дані подаються у вигляді дійсних чисел, в той же час, як для двійкових вхідні дані приставляються як вектор нулів та одиниць.

В сучасному світі більше застосовується саме аналогові нейромережі, оскільки при використанні двійкових нейромереж множина їх допустимих значень чітко обмежена, до того ж важко описати характеристики покладаючись лише на два допустимих значень, коли для аналогових нейромереж можливо досить легко описати ступінь впливу кожного елемента вхідного вектору. Проте, двійкові нейромережі потребують значно менших числових обрахунків, що потребує менше часу для навчання та роботи з нейромережею, в порівнянні з аналоговими.

1.2.4 Класифікація за моделлю, що застосовується в даній нейромережі

Нейромережі за даною класифікацію можна поділити на дві основні групи: мережі прямого поширення та рекурентні або із зворотним зв’язком мережі.

До мереж прямого поширення відносять нейромережі, зв’язки направлені строго від вхідних нейронів до вихідних. Прикладами даної нейромережі є одношаровий (Рис. 1.1.а) та багатошаровий перцептрон (Рис. 1.1.б), а також мережі радіальних базисних функцій (Рис. 1.1.б). Головною відмінністю багатошарового перцептрону та мережі радіальних базисних функцій є те, що мережі радіальних базисних функцій мають прихований шар із радіальних елементів, а вхідні та вихідні шари мають лінійну функцію

активації та вагові коефіцієнти рівні одиниці. Радіальні елементи – це елементи нейронної мережі, що використовують радіальні базисні функції в якості функцій активації. Дані нейромережі виглядають досить компактними та мають високу швидкість навчання.

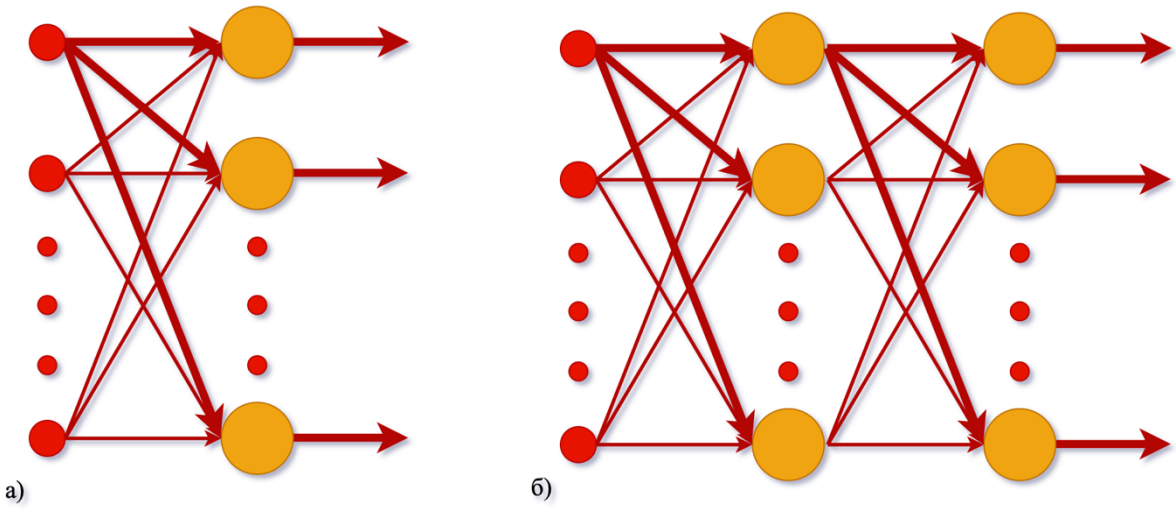


Рис. 1.1. Схематичне зображення нейромереж:
а – одношаровий перцептрон, б – багатошаровий перцептрон або мережа радіальних базисних функцій

Рекурентні нейромережі кардинально відрізняються за своєю структурою від нейромереж прямого поширення, оскільки в даному типі нейромереж з вихідних нейронів, або з нейронів прихованого шару сигнал передається назад на входи нейронів вхідного шару. У рекурентних нейронних мережах нейрони утворюють граф, що орієнтований у часі. Даний спосіб реалізації створює певний внутрішній стан мережі, що дозволяє нейромережі змінювати власну поведінку з часом, на відміну від мереж прямого поширення здатні користуючись внутрішньою пам'яттю виконувати обробку довільних послідовностей вхідних сигналів. Даний набір можливостей робить рекурентні мережі пристосованими до задач розпізнавання безперервного потоку рукописного тексту, а також розпізнавання голосу.

1.3 Огляд роботи нейромережі

Було отримано інформацію щодо загального визначення нейромереж, розглянуто класифікацію та їх основні функції, а також схематичне зображення. Тому розглянемо детальніше роботу нейромереж на рівні нейрону.

1.3.1 Нейрон

Нейрон – це найменша обчислювальна одиниця нейромережі, яка отримує інформацію, проводить над нею обчислення та передає її далі. Нейрони однієї мережі поділяються на три види: вхідний (Рис. 1.2(1)), прихований (Рис. 1.2(2)) та вихідний (Рис. 1.2(3)). В разі, коли нейромережа складається з великої кількості нейронів прийнято вводити термін шару. Рисунок 2.2 демонструє поділ мережі на шари, де шар – сукупність нейронів одного кольору.

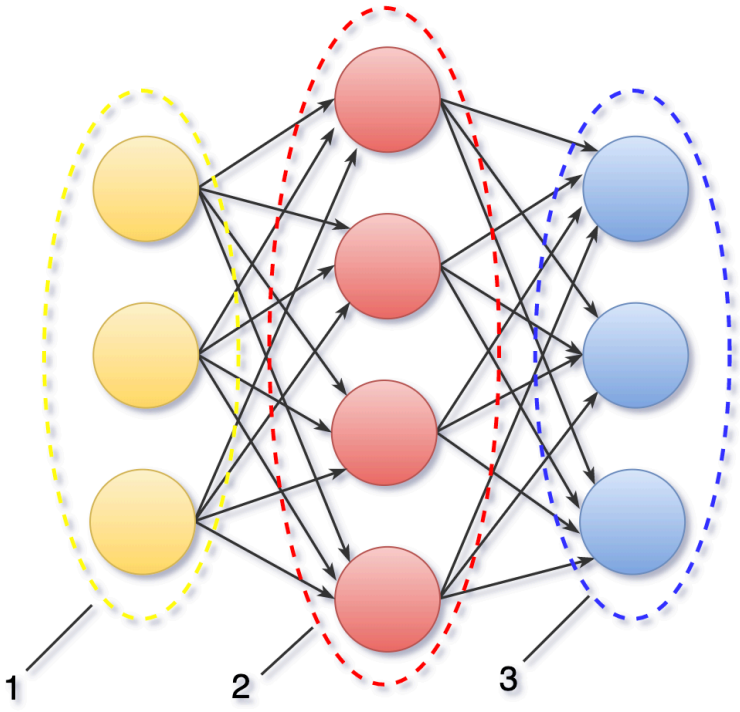


Рис. 1.2. Схематичне зображення шарів нейронної мережі:
1 – сукупність нейронів вхідного шару; 2 – сукупність нейронів прихованого шару; 3- сукупність нейронів вихідного шару

Відповідно, в базовій моделі нейромережі існує вхідний шар, який отримує початкові дані та передає до прихованого шару, яких зазвичай не більше трьох. Приховані шари виступають основними обробниками вхідної інформації, після обробки даних, прихований шар відправляє дані до вихідного шару та відповідно вихідний шар виводить результат. У кожного із нейронів існує два основних параметри: вхідні дані та вихідні дані. Для вхідних нейронів вхідні та вихідні дані є однаковими, а для інших вихідні дані отримуються виконанням операції сумування вхідних значень, а після чого нормалізується функцією активації. Зазвичай нейрони оперують числами в діапазоні від 0 до 1 або від -1 до 1.

Коректність роботи нейромережі напряму залежить від утворення синапсів (Рис. 1.3). Синапс – це зв’язок між двома нейронами. У синапсів є один параметр – ваговий коефіцієнт (W), завдяки даного параметра вхідні дані змінюються при передачі даних від одного нейрону до іншого. Допустимо, існує три нейрони, що передають дані іншому нейрону, тому, відповідно до цього маємо три синапси, а відповідно і три вагові коефіцієнти. То для нейрона, у якого ваговий коефіцієнт найбільший, дані будуть домінуючими у наступному нейроні.

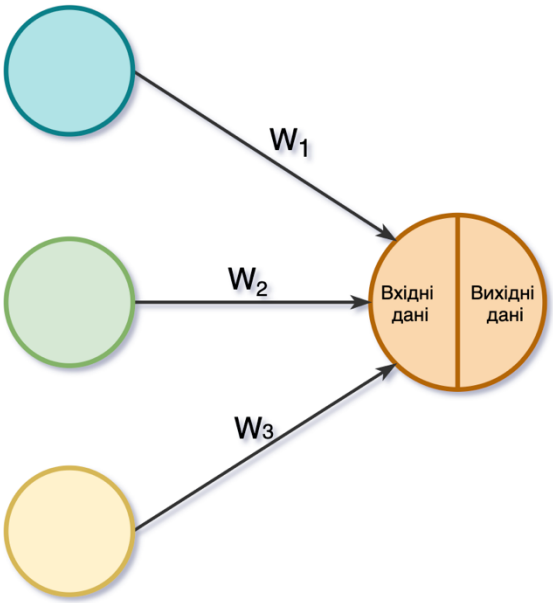


Рис. 1.3. Схематичне зображення синапсів нейромережі

Вагові коефіцієнти мають дуже важливу роль для отримання найбільш точних результатів, вхідна інформація оброблюється та перетворюється в результат. Важливим є початкова ініціалізація вагових коефіцієнтів випадковими числами, оскільки під час навчання, якщо вагові коефіцієнти будуть однаковими, то після визначення похибки після першої епохи всі коефіцієнти будуть змінені пропорційно, тобто залишаться однаковими, що негативно вплине на процес навчання нейромережі.

1.3.2 Принцип роботи нейромережі

Для аналізу кроків для моделювання та обчислення роботи нейронів змоделюємо частину нейромережі (Рис. 1.4). Якщо відомо вихідні дані нейронів I_1 та I_2 та відповідні вагові коефіцієнти W_1 та W_2 синапсів сполучення з нейроном H_1, тому розрахуємо вхідний та вихідний сигнал для нейрону H_1:

$$H_{1\text{вхід}} = (I_1 * W_1) + (I_2 * W_2), \quad (1.1)$$

де $\boldsymbol{H_{1}}_{\text{вхід}}$ – значення вхідного сигналу для нейрону H_1;

$\mathbf{I_1}$ – значення вхідного сигналу для нейрону I_1;

$\mathbf{W_1}$ – ваговий коефіцієнт синапсу між нейронами I_1 та H_1;

$\mathbf{I_2}$ - значення вхідного сигналу для нейрону I_2;

$\mathbf{W_2}$ - ваговий коефіцієнт синапсу між нейронами I_2 та H_1.

$$H_{1\text{вихід}} = f_{activation}(H_{1\text{вхід}}), \tag{1.2}$$

де $\boldsymbol{H_{1}}_{\text{вихід}}$ – значення вихідного сигналу для нейрону H_1;

$\mathbf{H_{1вхід}}$ – значення вхідного сигналу для нейрону H_1;

$\mathbf{f_{activation}}$ – функція активації нейрону H_1.

Для обрахунку значення вхідного сигналу необхідно скористатись формулою (1.1). Для обрахунку значення вихідного сигналу нейрону H_1 необхідно скористатися формулою (1.2) . З формули (1.1) видно, що вхідні дані – це сума всіх вихідних даних що мають синапси з входом нейрону, вхідний сигнал якого обраховується (в даному випадку нейрони I є вхідними нейронами). Дані обчислення проводяться для кожного нейрону мережі по шарам та в напрямку синапсів, до того часу, доки не буде вирахувано вихідні сигнали кожного нейрону вихідного шару мережі.

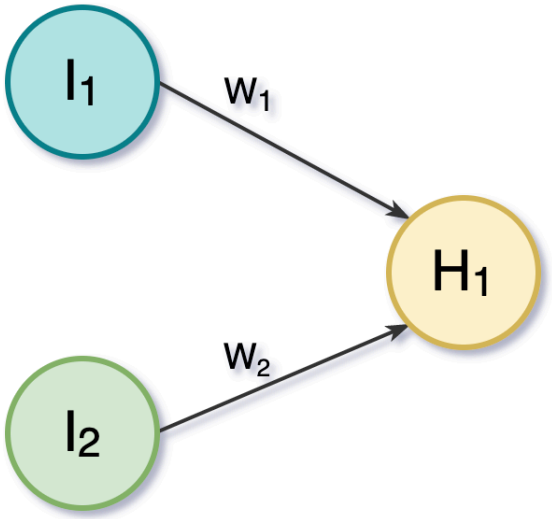


Рис. 1.4. Схематичне зображення елементів моделі для проведення розрахунків

Висновки до розділу 1

В даному розділі було розглянуто основну класифікацію нейронних мереж та проведено загальний опис кожного типу. Відповідно до цього, існує п’ять основних критеріїв, за якими класифікують нейромережі:

1) За характером навчання.
2) За способом налаштування вагових коефіцієнтів.
3) За типом вхідної інформації.
4) За моделлю, що застосовується для нейромережі.

Спираючись на отримані дані, було обрано основний тип нейронних мереж, створення яких буде реалізовано в даному дипломному проєкті, а саме: за характером навчання розроблювана система дозволить моделювати нейронні мережі для навчання з «учителем», за методом встановлення вагових коефіцієнтів система матиме лише динамічні синаптичні зв’язки, оскільки статичні зв’язки суперечать основній меті засобів моделювання – створення моделі «з нуля» та проводити навчання для отримання найбільш оптимальних показників точності мережі. За типом вхідної інформації в системі буде передбачено використання як аналогових вхідних даних, так і двійкових, що додає нейромережі більшої гнучкості та значно розширює її область застосування. За моделлю, що застосовується в нейромережі було обрано нейромережі прямого поширення, адже даний тип значно спрощує моделювання нейромережі та її наочне зображення.

Було розглянуто загальну структуру елементарного нейрону, сформовано його загальне визначення та принципи роботи нейромережі на прикладі одного нейрону, було дано визначення основним термінам, що використовуються при моделюванні нейромереж, зокрема: нейрон, вагові коефіцієнти, синапс та шари нейронної мережі.

Розглянута інформація дозволила більш детально зрозуміти структуру нейромереж та сформувала уявлення щодо їх загального представлення.

РОЗДІЛ 2

РОЗРОБКА СИСТЕМИ З АРХІТЕКТУРОЮ MSA

2.1 Вступ

«Мікросервіси» [2] – це один із нових термінів, що спричинив галас серед розробників програмного забезпечення. Конкретно даний термін описує стиль розробки програмного забезпечення, який набирає все більшої популярності. За останні декілька років з'являється велика кількість проєктів, що використовують даний стиль та варто зазначити, що результати до цих пір є досить позитивними. Навіть для багатьох розробників саме даний стиль стає основним в розробці програмного забезпечення. На жаль, існує не так багато інформації яка детально описує мікросервіси та те, як їх застосовувати.

Архітектурний стиль мікросервісів [2] – це підхід, при якому єдиний застосунок будується як набір невеликих сервісів, кожен з яких працює по власному алгоритму із власним процесом та здійснює обмін з іншими сервісами використовуючи різні взаємодії, зокрема HTTP. Дані сервіси побудовані, спираючись саме на бізнес-логіку та розгортаються незалежно один від одного з використанням повністю автоматизованого середовища. При цьому ключовим є мінімальний вплив на сервіси збоку централізованого управління, або ж його повна відсутність. Самі сервіси можуть бути написані на різних мовах програмування та використовувати різні засоби для збереження інформації.

2.2 Порівняння мікросервісної архітектури з «монолітом»

«Enterprise» [3] застосунки часто складаються з трьох основних частин:

1) Інтерфейс користувача, що включає в себе, як правило, HTML сторінки та javascript.

2) База даних, як правило реляційна з великою кількістю таблиць.

3) Сервер

Серверна частина оброблює HTTP запити, отримує та оновлює дані в базі даних, наповнює змістом HTML сторінки, які потім відправляються до браузеру користувача. При цьому, будь-яка зміна в системі призводить до перезбірки та розгортанню нової версії серверної частини застосунку.

Монолітний сервер – досить очевидний спосіб побудови подібних систем. Вся логіка по обробці запитів знаходиться в єдиному процесі, при цьому можливо використовувати можливості обраної мови програмування для поділу застосунку на класи та функції. Розробник може запускати та проводити тестування застосунку на власній локальній машині. Монолітний застосунок може бути масштабований горизонтально, шляхом запуску декількох фізичних серверів при наявності балансувальника навантаженості.

Монолітні додатки можуть бути успішними, але все більше розробників розчаровуються в них, особливо враховуючи те, що все більше додатків розгортуються в «хмарі». Будь-які зміни, навіть зовсім незначні потребують того, щоб увесь моноліт було перезібрано та знову розгорнуто. З плином часу розробникам стає все важче підтримувати хорошу модульну структуру застосунку, зміна логіки роботи одного модуля має тенденцію впливати на код інших моделей. Саме тому масштабувати доводиться весь додаток повністю, навіть коли зміни потрібні у роботі лише одного модулю.

Всі ці незручності призвели до архітектурного стилю мікросервісів: побудові застосунків у вигляді набору сервісів. Кожен сервіс має можливість бути незалежно розгорнутим та масштабованим. Кожен сервіс також отримує чіткий фізичний кордон, що дозволяє сервісам бути написаним на різних мовах програмування. Монолітні застосунки вкладають всю свою

функціональність в один процес (Рис. 2.1.а). Натомість мікросервісна архітектура вбачає вкладення кожного елементу функціональності в окремий сервіс (Рис. 2.1.б).

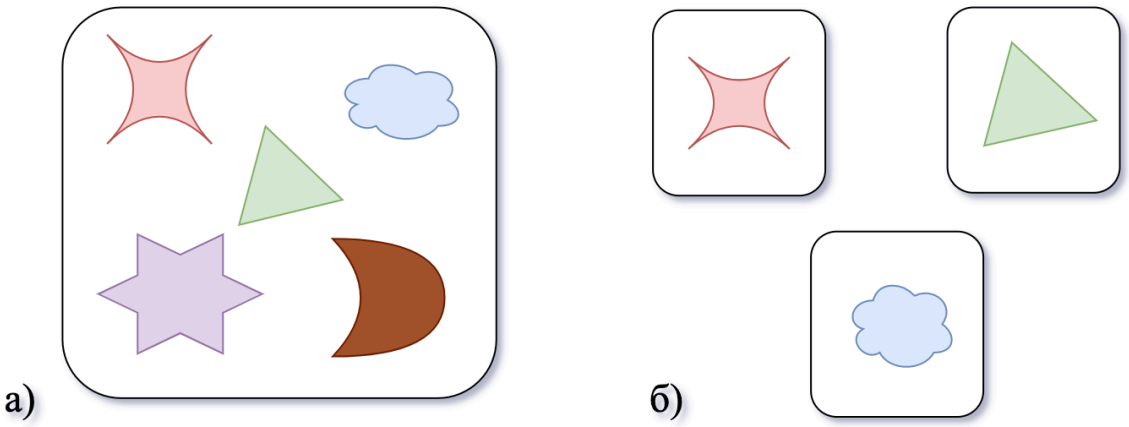


Рис. 2.1. Порівняння монолітної та мікросервісної архітектури додатків: а) монолітна архітектура; б) мікросервісна архітектура

Що стосується масштабування систем, то для системи з монолітною архітектурою масштабування може відбуватися тільки за рахунок розгортання копії системи на множині інших серверів (Рис. 2.2.а). Для систем з мікросервісною архітектурою масштабування відбувається в абсолютно інший спосіб. В даному випадку масштабування системи реалізується залежно від потреб. Виходячи з аналізу навантаженості або ж від його прогнозування на кожен з сервісів можна розгортати лише ті сервіси, які потрібні (Рис. 2.2.б). Такий варіант реалізації дозволяє уникнути надлишкової навантаженості фізичного серверу шляхом виконання лише того функціоналу, який потрібен для реалізації бізнес-логіки проєкту.

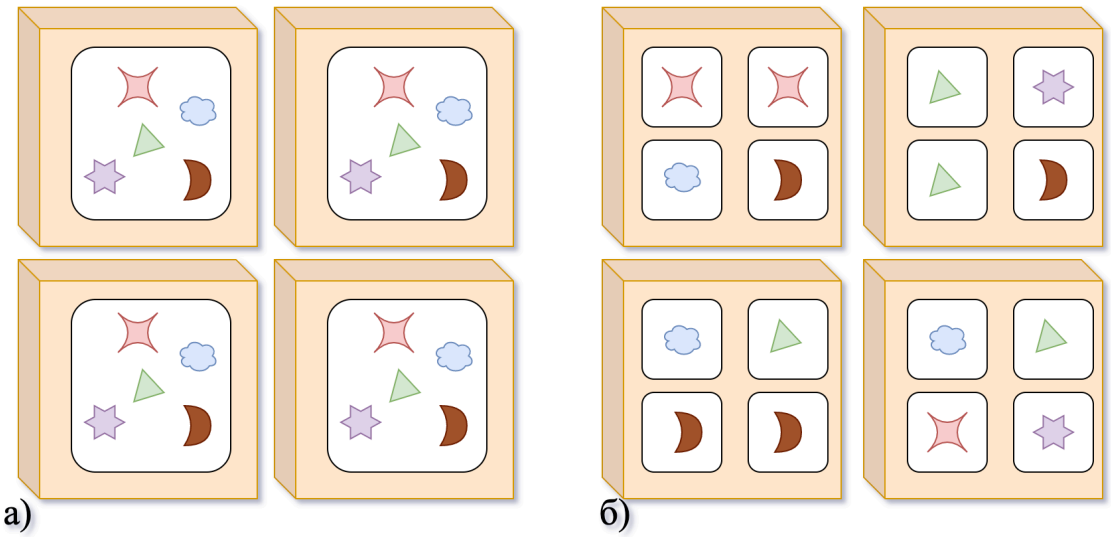


Рис. 2.2. Масштабування монолітної та мікросервісної архітектури додатків: а) монолітна архітектура; б) мікросервісна архітектура

Звісно, стиль мікросервісної архітектури не можна назвати інновацією. Оскільки подібні рішення були задіяні ще при створенні Unix-систем.

2.3 Властивості архітектури мікросервісів

На жаль, не існує формального визначення для опису стилю програмування що базується на створенні систем з мікросервісною архітектурою, тим не менше можливо описати загальні характеристики, що могли б описати застосунок, що був написаний з використанням даного стилю.

Незважаючи на те, який саме продукт намагається створити інженер, це може бути веб-застосунок чи додаток для мобільного телефону, в нього завжди з'являється бажання будувати системи шляхом поєднання різноманітних компонентів між собою, в більшості так само, як і в реальному житті. Тому, за останні декілька десятків років можна спостерігати швидкий ріст кількості різних бібліотек, що використовуються в більшості мов програмування.

Ключовою складовою в проектуванні мікросервісів є розбиття проєкту на окремі сервіси, оскільки саме від цього напряму залежить доцільність використання мікросервісної архітектури в цілому. Говорячи про компоненти мікросервісної архітектури вперше, можна отримати цілком логічне запитання: «Що таке компонент?». Беручи дане визначення з архітектури мікросервісів отримаємо, що компонент – це незалежна одиниця комплексу програмного забезпечення, що може бути незалежно замінена чи оновлена.

Звісно, архітектура мікросервісів може використовувати ті ж модулі та класи, що і монолітна архітектура, але ключовим є розбиття додатку на окремі сервіси. При проектуванні монолітного застосунку компоненти додатку, які були спроектовані підключаються до цього застосунку та виконуються в тому ж процесі, коли в той же час сервіси – це компоненти, що виконуються в окремому процесі та можуть взаємодіяти один з одним. Саме мікросервіси здатні вирішити одну з найбільш вагомих проблем монолітної архітектури, а саме перезбірки всього проєкту при наявності будь-якої зміни. В для мікросервісів достатньо лише перезібрати той сервіс, в якому відбулися зміни. Безумовно, можуть існувати зміни інтерфейсів сервісів, що понесе за собою внесення змін в комунікації між серверами, але основною ціллю даної архітектури є мінімізація необхідності даної взаємодії шляхом становлення найбільш коректних меж між сервісами, а також можливості створення механізму еволюції контрактів, що в свою чергу допоможе передбачити можливі зміни чи вдосконалення системи, що може вберегти від внесення змін в інтерфейсах, що мають взаємодіяти зі зміненим сервісом.

Іншою вагомою особливістю сервісів як компонентів додатку є наявність явного інтерфейсу між компонентами. Більшість мов програмування не мають хорошого механізму підтримки так званого «Published Interface» [2], тому лише дисципліна та домовленість між собою розробників додатку можуть гарантувати дотримання умов інкапсуляції компонентів. При

використанні мікросервісів можна уникнути даної проблеми шляхом використання явного механізму виконання віддалених викликів.

Але варто відмітити і вагомий мінус у використанні мікросервісів – це те, що швидкість передачі даних при здійсненні віддалених викликів значно нижча ніж виклики в рамках єдиного процесу і тому важливо щоб дані, що передаються були менш деталізованими, що може призвести до незручності у використанні.

На перший погляд можна подумати, що кількість сервісів додатку мають прямопропорційну залежність, а то й рівну кількість з загальним числом процесів цього додатку. Але це не так, оскільки сервіс може складатися з декількох процесів, які завжди будуть проектуватися та розгортатися незалежно один від одного (Рис. 2.3). Наприклад, процес компонента, тобто самого сервісу та процес бази даних, яку використовує даний сервіс.

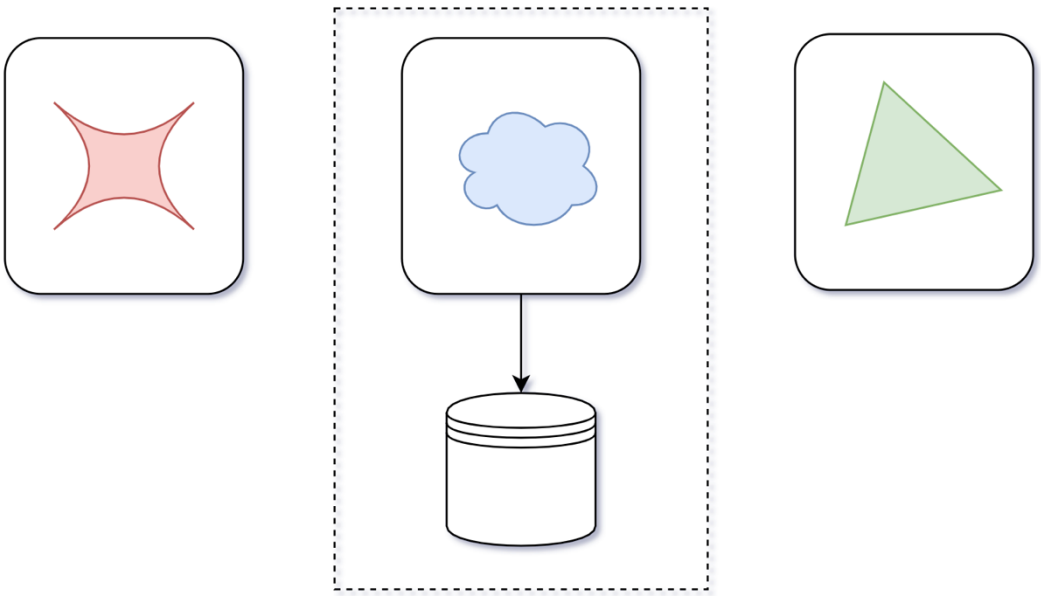


Рис. 2.3. Огляд сервісу в розрізі процесів, з яких він складається

2.4 Мікросервіси з точки зору організації командної роботи

Мікросервісна архітектура не обійшла стороною і зміни в організації роботи команди, що залучена до розробки певного проєкту. В загальному,

беручи будь-який трудомісткий та «змістовний» проєкт, над яким працює велика кількість людей, можна поділити фахівців на три основні категорії: UI група, власне команда розробників та команда архітекторів баз даних. Відповідно до цього для моделі проєкту з монолітною архітектурою (Рис. 2.4) «розбиття» команд утворюється таким чином що над однією структурною одиницею проєкту відразу працює декілька людей. При даній структурі навіть невеликі зміни тягнуть за собою велику втрату часу через необхідність крос-командної взаємодії для узгодження змін. Звідси, як наслідок команди розміщують довільну логіку на тих структурних одиницях, до яких мають доступ.

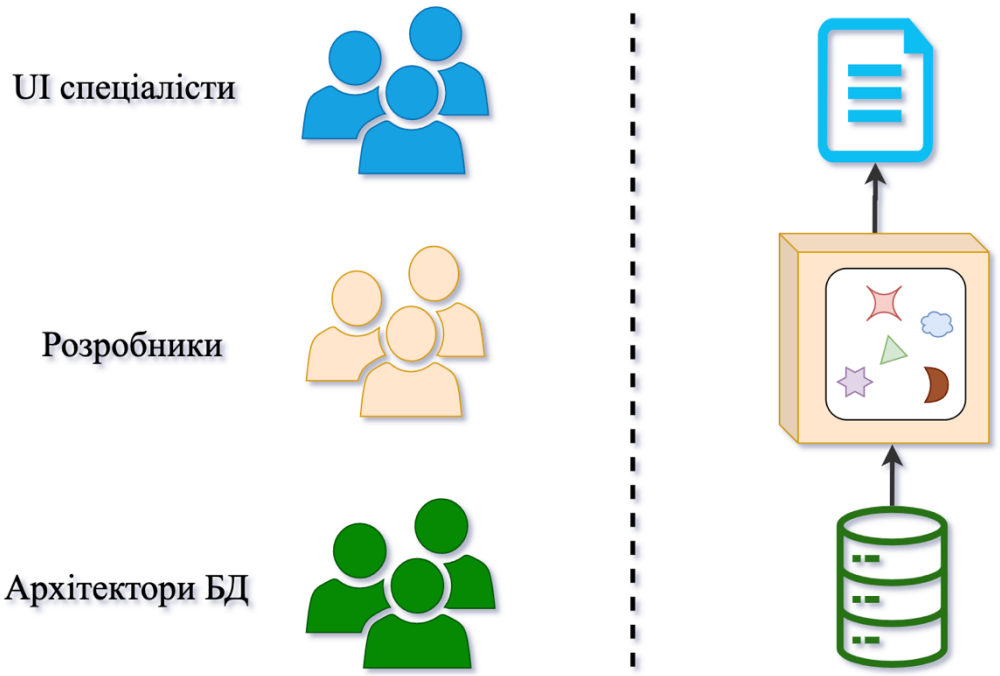


Рис. 2.4. Модель організації командної роботи над проєктом з монолітною архітектурою

В результаті можна отримати порушення задекларованих норм використання інтерфейсів, порушення зв’язків та інших наслідків що можуть призвести до того, що згодом проєкт буде важко підтримувати та вдосконалювати.

Натомість, мікросервісний підхід до утворення команди базується на тому, що формування команд відповідає бізнес-логікі проєкту. Оскільки кожен сервіс є окремим повноцінним додатком, то він включає в себе інтерфейс користувача, серверну частину застосунку, взаємодію з базою даних та інші зовнішні взаємодії. Даний набір функціоналу що закладено в один компонент мікросервісної архітектури призводить до формування так званих «крос-функціональних команд» (Рис. 2.5), до яких входять розробники з усіх категорій, зазначених вище: UI група, власне команда розробників та команда архітекторів баз даних.

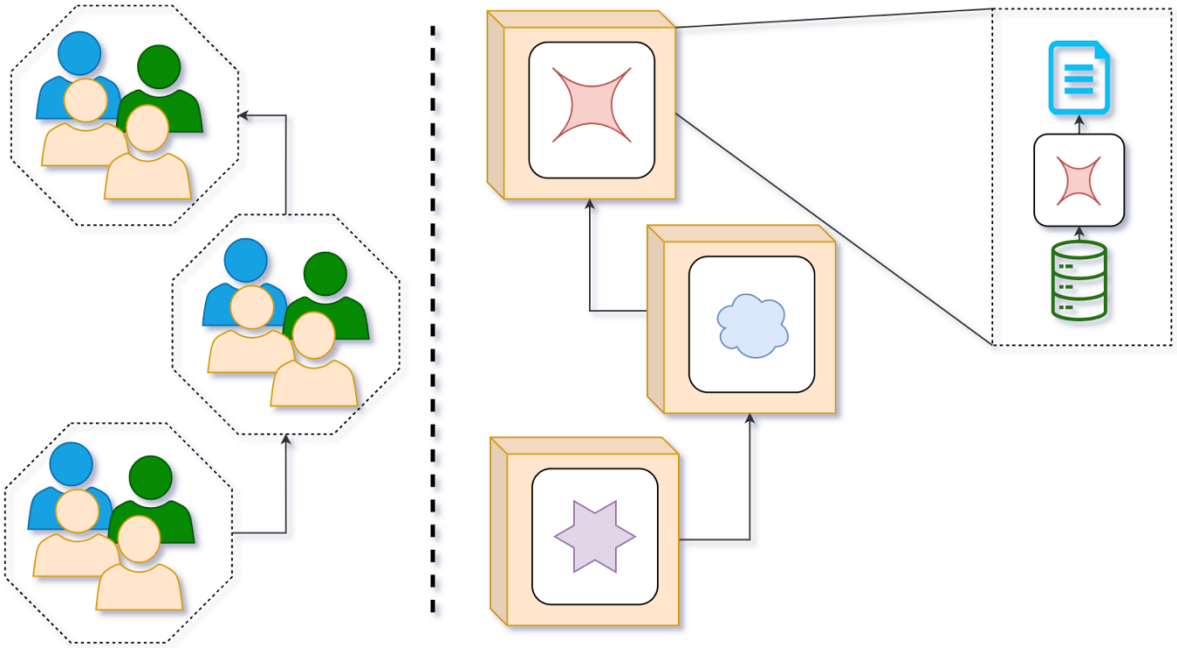


Рис. 2.5. Модель організації командної роботи над проєктом з мікросервісною архітектурою

Таким чином зміна основного функціоналу серверу не призводить до утворення крос-командної взаємодії, а навпаки, будь-які зміни будуть внесені в межах однієї команди. Лише при необхідності внесення змін в структуру інтерфейсів взаємодії між компонентами виникає зв'язок між командами та включаючи лише ті команди, які використовують інтерфейси даного сервісу.

Наслідком цього є більш чітке розмежування обов’язків кожного учасника процесу розробки.

2.5 Децентралізоване управління системою

Використання монолітної архітектури призводить до централізованого управління процесом, що з одного боку може дати перелік переваг, але з іншого боку і ряд недоліків. Одним із найбільш вагомих наслідків централізованого управління є тенденція до стандартизації використовуваних платформ. Досвід роботи з монолітними застосунками показує, що такий підхід до створення значною мірою обмежує вибір. Оскільки кожна проблема є цвяхом, але не кожне платформне рішення є для неї молотом. Тобто, часто виникають задачі, що мали б простіше та доцільніше вирішення саме в іншій платформі.

Саме за допомогою мікросервісної архітектури з’являється можливість використання правильного інструменту для поставленої задачі навіть в рамках одного проєкту. В монолітному застосунку, в деяких випадках, є можливість використання різних мов програмування, але це не є стандартною практикою.

Наприклад, для створення логіки створення та відображення певних даних, які не потребують великих обчислень та обробки даних складними алгоритмами можна використовувати мови програмування що мають найбільш широкі можливості саме серед веб-сервісів, для зручності та швидкості написання великої частини функціоналу. А модуль, що відповідає за обробку інформації та обрахунки певних даних краще використовувати мови програмування, що мають широкий функціонал для проведення такого роду операцій та можуть обробити дані значно швидше.

Звісно, наявність можливості використання як різних мов програмування, так і мікросервісів не може значити, що їх необхідно

використовувати, але система, розроблена таким чином надає можливість вибору.

Децентралізоване управління даними може бути представлене в різному вигляді. В абстрактному сенсі це може означати концептуально для різних систем «модель світу» може відрізнятися. Ця поширена проблема має аналогію в реальному житті. Для прикладу: для звичайного користувача термін «комп'ютер» може означати пристрій, де він має можливість встановити потрібні застосунки, а в той же час для інженера комп'ютерних систем це є набір обладнання з певними характеристиками. При цьому навіть частини атрибутів «комп'ютеру» відсутні в контексті звичайного користувача, більш того, однакові реквізити можуть приймати різне значення. Дана проблема існує не лише в розрізі різних застосунків, а й навіть в рамках однієї системи, особливо тоді, коли даний застосунок розділено на окремі компоненти. Дану проблему можна вирішити так званим поняттям обмеженого контексту (Bounded Context) із DDD. В даному випадку DDD пропонує поділити складну предметну область на контексти, які будуть записані до контексту «Maps», що в свою чергу слугує поєднанням всіх контекстів одного визначення для того щоб показати як вони повинні взаємодіяти один з одним та які саме дані повинні бути спільними. Даний процес є корисним також і для монолітних програм, але для мікросервісної архітектури дане рішення має прямий зв'язок та допомагає оголосити та підтримувати чіткі межі контекстів.

Крім децентралізації прийняття рішень, мікросервіси також виконують децентралізацію методів збереження даних. Для монолітних програм часто використовують одну БД для збереження даних. Натомість керуючись архітектурою мікросервісів розробники дають мікросервісам можливість виконувати управління власною базою даних, це стосується як можливості створення окремих екземплярів загальної СУБД , так і використовувати нестандартні види БД. Даний підхід носить назву «Polyglot Persistence» (Рис. 2.6) – концепція використання різних технологій збереження даних для задоволення різноманітних потреб в збереженні даних в рамках одного програмного комплексу.

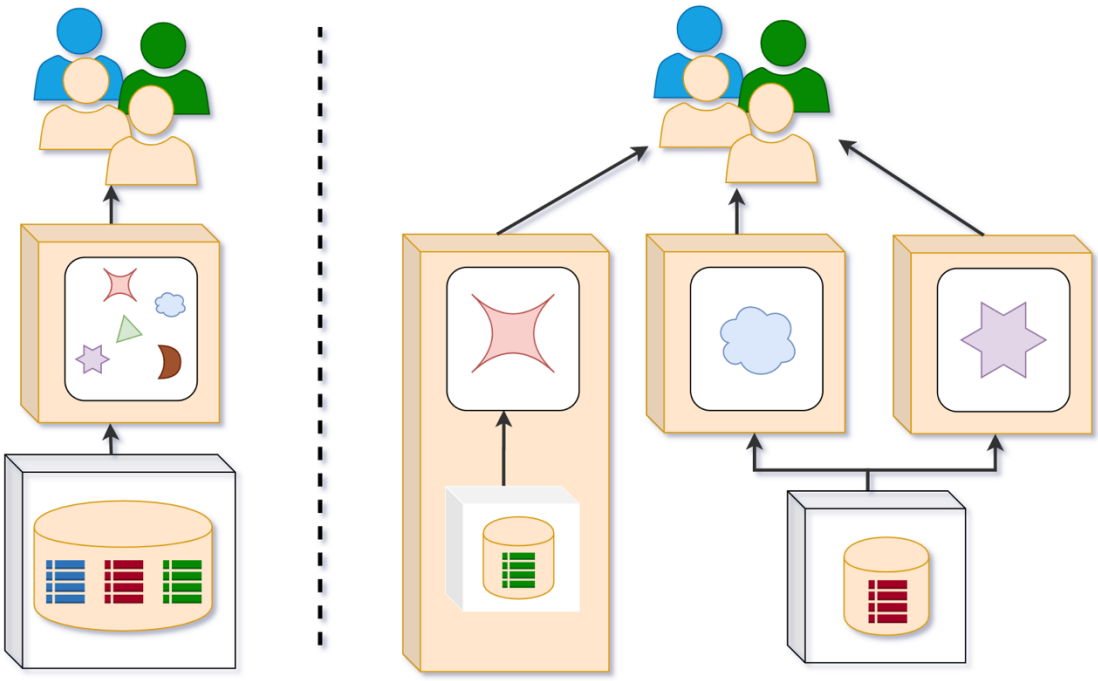


Рис. 2.6. Порівняння архітектури збереження даних для монолітної та мікросервісної архітектури

Децентралізація відповідальності за збереження даних серед мікросервісів впливає на зміну даних. Звичайний підхід до внесення змін в дані базується на використанні транзакцій для гарантування коректності при зміні даних, що знаходяться на декількох ресурсах, саме тому такий підхід використовується саме в монолітних програмах оскільки всі записи до БД

здійснюються в одному процесі. Також, подібне використання транзакцій гарантує коректність збереження даних, але призводить до суттєвої залежності в часі, яка в свою чергу призводить до проблем при роботі з багатьма сервісами. Саме тому архітектура MSA придає особливе значення використання з'єднання між сервісами без використання транзакцій, при чому коректність записів є лише підсумковою, тобто проблеми, що виникають можуть вирішуватись операціями компенсації.

2.6 Порівняння MSA та SOA

Сервіс-орієнтована архітектура або SOA [3] – це підхід до розробки програмного забезпечення, що базується на використанні розподілених або слабко зв'язаних, замінних компонентів.

На перший погляд мікросервісна архітектура MSA та SOA не мають видимої різниці, тому розглянемо їх детальніше. Сам термін SOA має досить багато значень та охоплює досить широкий спектр можливих варіантів реалізації. Головна ідея даної архітектури базується на тому, що застосунок складається з набору сервісів, які взаємодіють один з одним за допомогою стандартизованих протоколів та інтерфейсів. Але більшість прихильників сервіс-орієнтованої архітектури зазвичай зв'язують її з ESB, що часто використовують для інтеграцій монолітних програм.

ESB - це програмне забезпечення, що дозволяє зв'язувати та об'єднувати між собою велику кількість різних додатків, що були розроблені за допомогою різних платформ та мов програмування, основна її функція – забезпечення саме централізованого обміну інформації, що є цілковитою противагою ідеї мікросервісів.

Також, історія свідчить, що існувало досить багато невдалих реалізацій архітектури SOA, зокрема тенденція тяжкі реалізації за ESB та багатьма

невдалими ініціативами, що тривали роками та не принесли компаніям, що їх використовували жодної користі.

Безумовно, велика кількість практик, що застосовується в мікросервісній архітектурі була взята саме від SOA. Наприклад: шаблон «Tolerant Reader», що в першу чергу встановлює певні правила до побудови API викликів, що базується на законі Джона Постела, головною метою якого є використання лише тих полів запиту, які потрібні для його обробки. Іншим, досить важливим прикладом є використання простих протоколів з'єднань, які виникли в знак протесту складним централізованим стандартам.

Все ж, наявність описаних проблем призвела до того, що прихильники архітектури MSA відмовились від терміну «SOA», задля уникнення неприємних асоціацій, хоч в той же час інші вважають мікросервіси однією з форм SOA, або ж правильною реалізацією архітектури SOA. Тим не менше, при наявності конкретизації у використанні мікросервісів, в порівнянні з широким спектром значень терміну «SOA» досить корисно мати окремий термін.

2.7 Інструменти взаємодії мікросервісної архітектури

Для вдалого функціонування системи, побудованої на мікросервісній архітектурі існує ряд інструментів, що забезпечує взаємодію мікросервісів, їх запуск та підтримку, що значно спрощує їх розробку.

2.7.1 API Gateway

Користувач будь-якого веб-застосунку не повинен знати про внутрішню архітектуру сервісної частини програмного комплексу, оскільки звернення окремо но кожного елементу мікросервісної архітектури може як створювати незручності користувачу, так і ставити під загрозу безпеку даних сервісів. Для

рішення даної проблеми було створено шаблон API Gateway [3] (Рис. 2.7), який буде існувати як єдиний компонент, що буде слугувати точкою входу для всього програмного комплексу.

Даний сервіс може виступати не лише як єдина точка входу, а також як проміжковий набір фільтрів для задоволення бізнес логіки застосунку, а також для виконання автентифікації користувача, та за відповідність правам доступу до ресурсу відповідно до ролей користувача. Також даний сервіс відповідає за переправлення запитів користувача до потрібного сервісу.

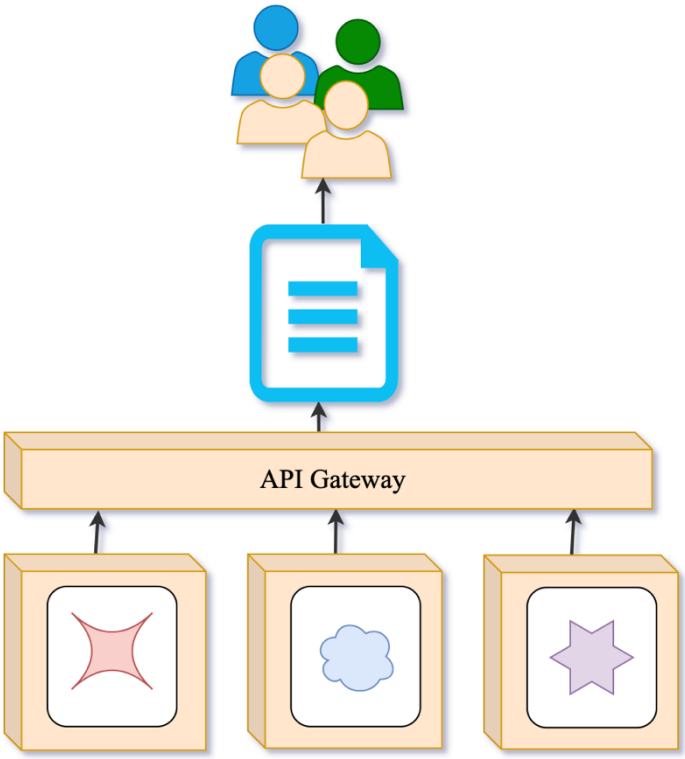


Рис. 2.7. Представлення сервісу API Gateway з огляду на взаємодію клієнтів з системою

Наявність даного сервісу спрощує розробку з точки зору безпеки, оскільки описувати параметрами безпеки потрібно було б на кожному з вузлів застосунку, натомість застосування модулю Gateway дає можливість встановлювати параметри безпеки для всієї системи в цілому.

Даний шлюз є певною аналогією структурного патерну ООП фасад, ціллю якого є створення певного інтерфейсу, дозволяючи сховати складність

системи шляхом зведення всіх можливих зовнішніх викликів до одного об’єкту який делегує їх до відповідних об’єктів.

Такий шаблон має безліч переваг, зокрема зручність використання єдиної адреси для доступу до цілої системи, ніж використання десятків адрес для доступу до кожного з ресурсів. Також можливе встановлення централізованого обмеження на кількість сумнівних запитів, що має велике значення для забезпечення безпеки системи, з’являється можливість поєднання відповідей від декількох сервісів та передавати клієнту комплексну відповідь та з’являється можливість їх кешування. Так і в цілому система стає більш гнучкою, завдяки тому, що клієнту не доведеться змінювати параметри підключення незалежно від наявних змін в екосистемі всього застосунку.

Але даний шаблон має не тільки корисні функції, оскільки може вносити певні обмеження до майбутнього розширення системи. Оскільки це окремий сервер, то він потребує розгортання та запуску на ряду з іншими, а також даний сервіс потребує того, щоб кінцеві точки всіх сервісів були додані до параметрів даного сервісу. Попри всі недоліки даний сервіс є дуже важливим та цілком себе виправдовує.

2.7.2 Discovery server

В мікросервісній архітектурі можлива велика кількість різноманітних серверів, що можуть знаходитись на різних портах та навіть на іншій IP-адресі, тим більше може існувати декілька екземплярів одного класу, тому статичний запис параметрів підключення до кожного з серверів може значно обмежити можливості та створити незручності в розробці програмного комплексу. Дана проблема нівелює навіть однією з головних цілей мікросервісної архітектури – динамічне масштабування системи шляхом створення нових екземплярів сервісів.

Саме тому для рішення даної проблеми було розроблено додатковий сервіс що відповідає за реєстрацію окремих компонентів - Discovery server (Рис. 2.8). В даному сервісі кожен клієнт (для даного випадку кожен сервіс програмного комплексу) при запуску надсилає повідомлення про своє існування, таким чином, реєструючись в Discovery server, таким чином даний сервер виступає своєрідним реєстром всіх сервісів застосунку, а також зберігає основні дані про них, такі як: назва сервісу, IP-адреса, порт для підключення, після чого слідкує за доступністю даного сервісу, тобто якщо, за довгий період часу сервер не передає повідомлення про те, що він ще працює, автоматично видаляє його зі списку доступних для підключення серверів.

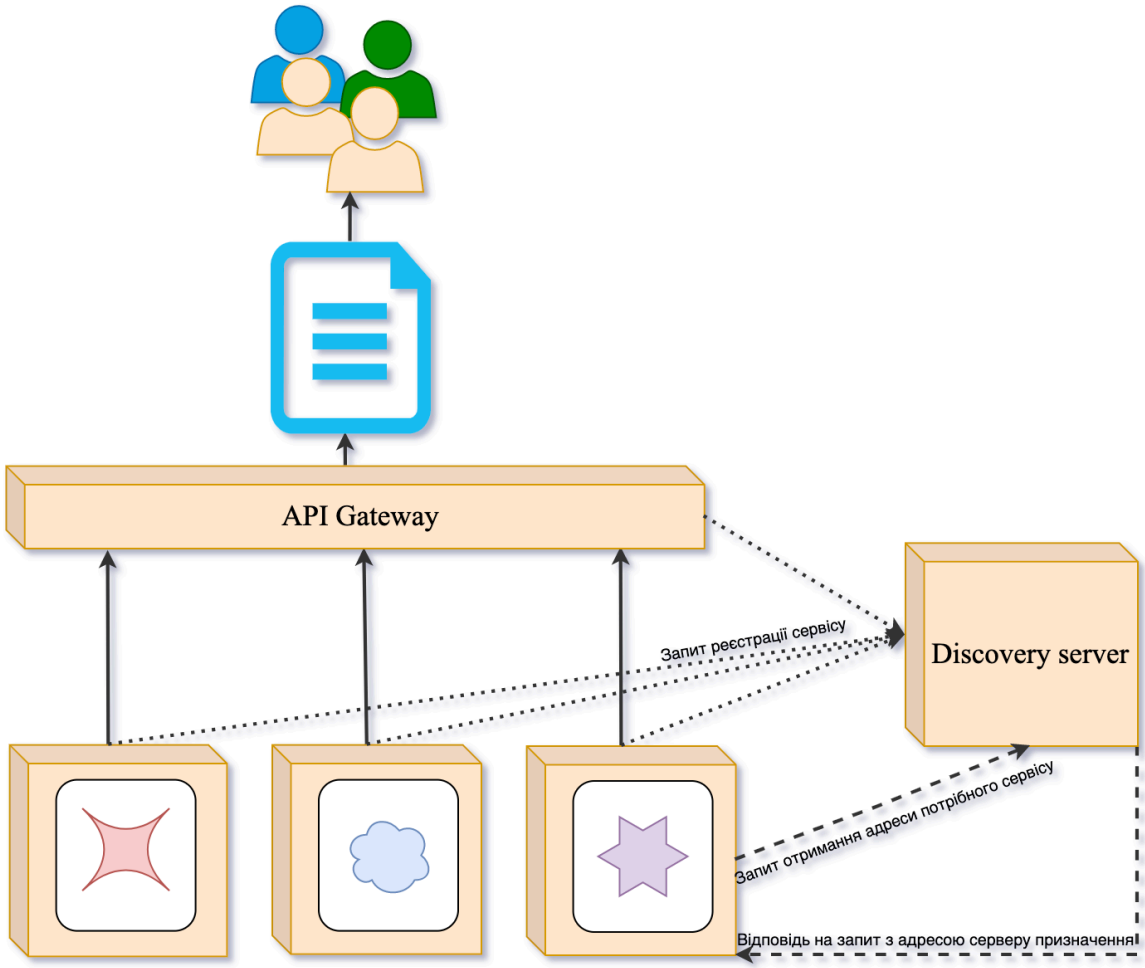


Рис. 2.8. Взаємодія серверів-клієнтів з Discovery server

При наявності даного серверу, будь-який сервер, якому потрібно отримати певні дані з іншого сервісу, знаючи назву серверу призначення може

запросити дані в Discovery server, та отримати завжди актуальну адресу та порт для підключення, навіть якщо він був неодноразово змінений за час, коли програмний комплекс був запущений. Даний сервер має бути постійно запущеним, інакше інші сервери не можуть обмінюватись даними за його відсутності.

Варто додати, що для деяких реалізацій мікросервісів також існує можливість підключення балансувальника навантаження, що створений для того, щоб при наявності декількох екземплярів одного сервісу оцінювати завантаженість кожного з них та передавати параметри підключення того, що найменш навантажений.

2.7.3 Circuit Breaker

З опису архітектури MSA зрозуміло, що в такому варіанті реалізацій сервіси досить часто можуть взаємодіяти між собою, іноді навіть може утворюватися ланцюжок з викликів (Рис. 2.9).

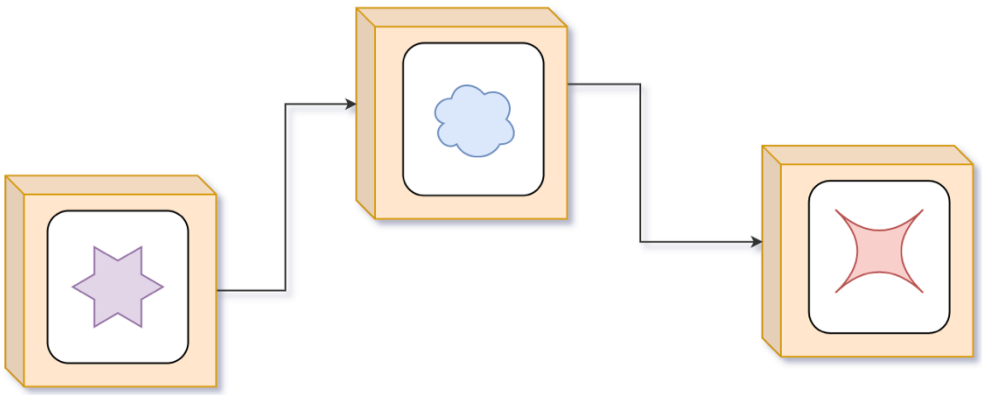


Рис. 2.9. Демонстрація ланцюга викликів
при взаємодії мікросервісів

Попри використання асинхронних викликів, що не призводять до зупинки загального виконання процесу сервісу можуть значною мірою

навантажувати сервер. Та можлива ситуація, коли на одному з етапів даного ланцюжку стається помилка (Рис. 2.10), тоді дана помилка передається назад по ланцюжку до клієнта. Таким чином, системою було затрачено досить велика кількість ресурсів, але результату, що очікував клієнт так і не було отримано. Попри те, клієнт може спробувати відправити даний виклик знову та знову, при цьому система буде оброблювати інформацію і кінцевому рахунку отримає ту ж помилку.

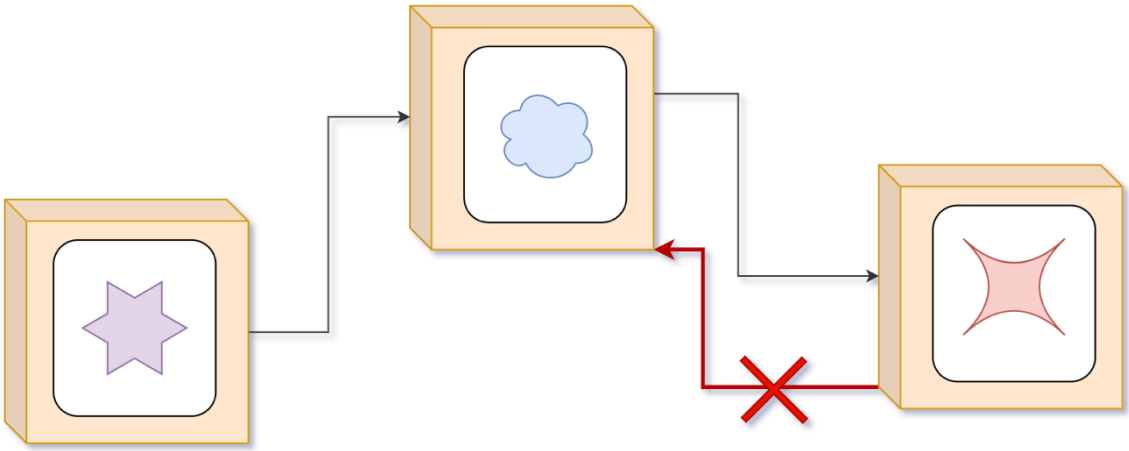


Рис. 2.10. Помилка при виконанні ланцюга викликів

Дана помилка може бути викликана будь-яким чинником, для прикладу, збій серверу, на який потрібен час для того, щоб сервер відновив свою роботу, але такі виклики можуть не дати йому зробити цього.

Для вирішення даної проблеми було розроблено патерн Circuit Breaker. Реалізацією даного патерну є сервіс, що слугує проксі-сервером між сервісом, що запитує інформацію та віддаленим сервером, аналізуючи останні помилки, що виникали в даному сполученні для визначення того, чи даний виклик може бути успішним, якщо ні – то даний сервер відразу поверне помилку клієнту, що звільнить систему від надлишкових навантажень та прискорять швидкість реагування серверу для клієнта.

За патерном Circuit Breaker даний проксі-сервер має три стани: «Closed», «Open», «Half-Open». Розглянемо детальніше дані стани. Коли

сервер знаходиться в стані «Closed», він дозволяє відсилати запити в будь-якому напрямку. Проксі-сервер має лічильник помилок, тож коли кількість помилок перевищує певне значення, то сервер перемикається в режим «Open». В даному режимі сервер дає можливість ресурсу, до якого адресовано віддалений виклик відновитися, встановивши таймер, під час якого до серверу обмежені запити. Після закінчення даного таймеру сервер переходить в режим «Half-Open» для того, щоб можна було перевірити стан серверу. Після наступного запиту до даного ресурсу відбувається перевірка відповіді. Якщо віддалений сервер повернув помилку, то проксі-сервер знову переходить в режим «Open» та запускає новий таймер, інакше переходить в режим «Closed» та дозволяє будь-які запити до даного віддаленого серверу.

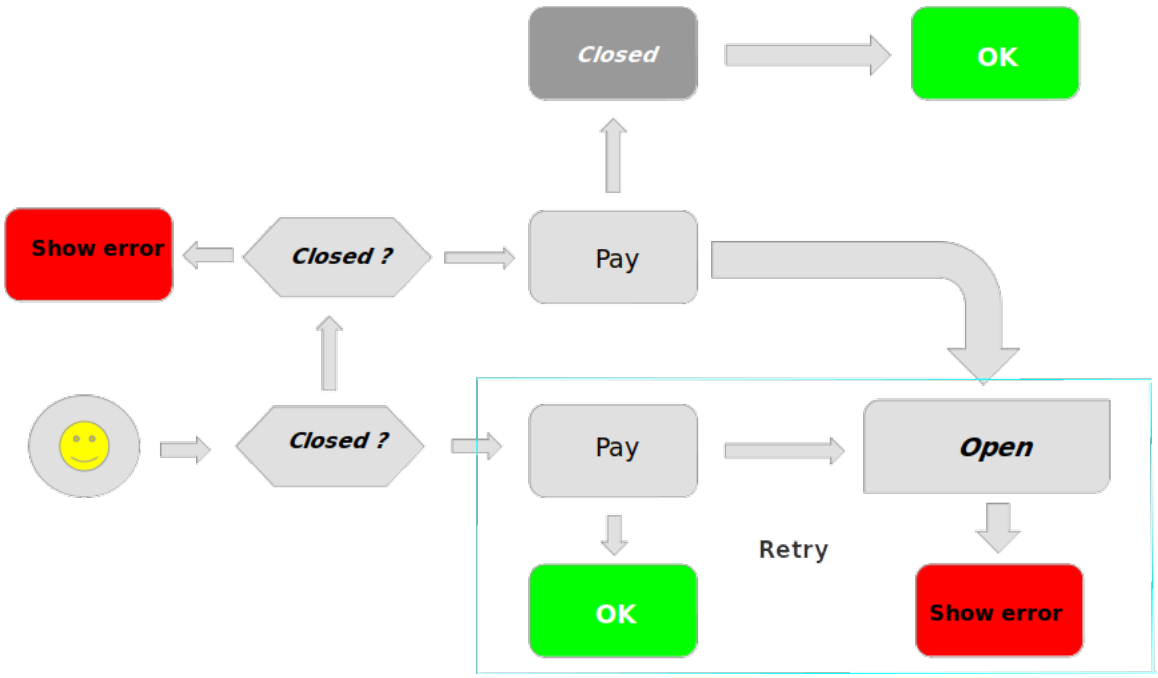


Рис. 2.11. Схематичне зображення зміни станів проксі серверу на основі патерну Circuit Breaker

Реалізація даного патерну грає важливу роль при побудові систем з мікросервісною архітектурою, оскільки додає стабільності роботи програми в цілому, особливо тоді, коли система відновлюється після падіння та мінімізує вплив даного падіння на продуктивність системи в цілому. Крім цього,

можливо відслідковувати падіння серверів, тобто події переходу статусів проксі-серверу для їх моніторингу та передачі сповіщень про наявність помилок сервісів.

2.8 Приклади реалізації

Не залежно від області, в якій працюють розробники, завжди виникає бажання стандартизації деяких рішень. Тож і команди розробників додатків з мікросервісною архітектурою не є виключенням. Замість того, щоб використовувати набір стандартів, що були написані кимось раніше, віддають перевагу побудові інструментів чи навіть їх комплекси для того, щоб інші розробники в майбутньому змогли скористатись ними для вирішення схожих задач.

На даний момент досить хорошою практикою серед великих компаній, що займаються розробкою власного програмного забезпечення існує практика взяття вдалих рішень реалізації з програмного модулю однієї команди та поширення цього серед різними командами, при цьому використовуючи модель внутрішнього OS. Тобто кожна команда розробників даної компанії може скористатися даним рішенням при наявності подібної проблеми. Зараз, коли git став стандартною системою контролю версій, то OS-практики стають все більш популярними, та це стосується не тільки внутрішніх проєктів компанії.

Одним із найбільш яскравих прикладів є компанія Netflix, структура якої повністю наслідує ідеї MSA. Дана компанія стала так званим «піонером» в розробці програмного забезпечення для хмарних обчислень та розробила власний набір бібліотек під назвою Netflix OSS, який дає змогу будь-якому розробнику творити власну систему, що буде базуватися на MSA моделі, при чому залишаючи вибір підходу для рішення задачі розробнику.

Розглянемо детальніше способи реалізації мікросервісної архітектури для різних мов програмування.

2.8.1 Засоби реалізації MSA для Java

Java – це суворо типізована, об’єктно орієнтована мова програмування, що є однією з найпопулярніших мов, якою користується велика кількість розробників по всьому світу. Програмний код, що було написано на даній мові програмування компілюється в байт-код, що може виконуватись на будь-якому пристрої, де встановлено віртуальну машину JVM, що інтерпретує байт-код в команди процесора. Тобто в даному випадку, розроблене програмне забезпечення не залежить від апаратного чи програмного забезпечення на якому запускається.

Що стосується мікросервісів, то саме для Java створена неймовірна кількість бібліотек, що можуть бути задіяні для створення та підтримки роботи мікросервісів, саме тому велика кількість міжнародних компаній розроблюють власне програмне забезпечення з мікросервісною архітектурою за допомогою засобів Java.

2.8.1.1 Spring framework

Spring framework [4]– один із найпотужніших фреймворків, написаний для Java та слугує для зручного створення веб-застосунків будь-якої складності. Даний фреймворк складається з великої кількості бібліотек та складається з модулів, основні з яких: Spring Boot та Spring Cloud. Дані модулі можуть добре працювати разом.

Spring Boot – це модуль, що відповідає за створення застосунків з використанням всіх можливостей бібліотек Spring, головна ціль якого – спрощення створення застосунків на основі Spring. Він дозволяє найбільш

простим способом створити веб-додаток, при цьому потребуючи від розробника мінімум часу та написаного програмного коду.

Серед ключових можливостей Spring Boot відповідає за управління залежностями, автоматично конфігурує сервер та має вбудовані контейнери сервлетів. Конфігурація серверу, його налаштування та перелік залежностей вказується в файлі конфігурації в форматі XML. Це створено для того, щоб пришвидшити процес управління залежностями, тому Spring Boot неявно запаковує всі сторонні залежності та надає їх розробнику у вигляді starter-пакетів, при цьому розробнику достатньо встановити лише декілька залежностей та отримати доступ до потрібного функціоналу.

При підключенні модулю Spring Boot розробник отримує можливість підключити та використовувати будь-який з модулів сімейства Spring, зокрема Spring Cloud. Spring Cloud – модуль, що розроблений для створення розподілених веб-систем та має велику кількість інструментів для розробки. Даний модуль включає в себе набір бібліотек Netflix OSS, що відповідають за створення компонентів мікросервісної архітектури, їх конфігурації та підтримки.

Підключення основних елементів бібліотек відбувається за допомогою анотацій, наприклад анотація @SpringBootApplication слугує для позначення застосунку, що буде побудований на основі модулю Spring Boot та вказується для головного класу програми.

Spring надає зручний інтерфейс для з’єднання на до великого переліку баз даних шляхом підключення драйверу необхідної БД в залежності додатку, в файл конфігурації XML.

2.8.1.2 Spark framework

Apache Spark – це фреймворк для виконання розподіленої пакетної та потокової обробки неструктурованих та слабко структурованих даних, що допомагає в розробці розподілених систем.

Керування в даному фреймворку побудовано на основі лямбда-виразів, тому в даному випадку можливе використання лише версій Java 8 та вище. Його основною особливістю є мала кількість потрібного коду для створення повноцінного REST API.

Spark framework слугує для побудови невеликих та зручних веб-застосунків. Зокрема його використовують в інтернеті речей для розробки окремих модулів. Також Spark можна використати для машинного навчання, але він має часову затримку при обробці даних, а саме цей параметр є критичним для застосування даних серверів для обробки запитів від клієнтської частини додатку.

2.8.2 Засоби реалізації MSA для Python

Python – високорівнева мова програмування загального призначення, що орієнтована на підвищення продуктивності розробника та на покращену читаності коду, само тому синтаксис даної мови досить мінімалістичний, але в той же час стандартна бібліотека Python включає великий об’єм корисних функцій.

Дизайн мови Python побудований навколо об'єктно-орієнтованої моделі програмування. Реалізація ООП в Python є елегантною, потужною і добре продуманою, але разом з тим досить специфічною в порівнянні з іншими об'єктно-орієнтованими мовами.

Дана мова програмування стає все більш популярною, її використовують для проведення складних обчислень, машинного навчання та

проведення обчислень з використанням нейромереж. А також, для побудови веб застосунків.

2.8.2.1 Flask

Flask [5] – фреймворк для створення веб-застосунків на мові програмування Python, відноситься до категорії так званих мікрофреймворків – мінімалістичних каркасів для створення веб-застосунків, заздалегідь розуміючи, що вони будуть представляти лише базові можливості реалізації.

Спираючись на характеристику даного фреймворку можна сказати, що він не достатньо зручний для побудови з його допомогою повної моделі мікросервісної архітектури, але можливе застосування як додаткового сервісу для виконання певних операцій.

2.8.2.2 Tornado

Tornado – розширюваний та неблокуючий фреймворк, написаний на Python, має підтримку асинхронної обробки викликів, був створений для забезпечення високої продуктивності систем та славиться як один із веб-серверів, що здатний витримувати навантаження в понад 10 000 з'єднань.

Дані досягнення працюють для так званих «легких» запитах, що не потребують великих обчислень та довготривалої роботи з даними, зокрема з базами даних. Тому при виконанні складних обчислень переваги даного фреймворку не працюють.

Дану бібліотеку доцільно використовувати коли важлива підтримка великої кількості одночасних звернень до системи для підтримки довготривалого зв'язку з клієнтом, при цьому без виконання складних операцій з даними.

2.8.3 Засоби реалізації MSA для C++

C++ – відома та широко використовувана мова програмування загального призначення, що здебільшого використовується для написання високопродуктивних програмних продуктів. Дана мова програмування є компільованою та статично типізованою, де є можливість прямої роботи з пам'яттю, саме тому важливе значення має архітектура машини, для якої написано застосунок. На відміну від Java один і той же програмний код не може бути використаним для різних операційних систем та для різної архітектури комп'ютерного обладнання.

Областю застосування C++ є створення операційних систем, драйверів до пристроїв, різноманітних прикладних застосунків, високопродуктивних серверів, а також ігор. Дані застосування можливі завдяки наявності в C++ властивостей, притаманних як мовам високого рівня, так і низького, включаючи можливості підтримки багатопоточності.

2.8.3.1 C++ MicroServices

C++ Micro Services – колекція компонентів для побудови модульного та динамічного сервіс-орієнтованого застосунку. Головною функцією якої є створення програмних рішень повторного використання, з забезпеченням слабкої зв'язності між окремими компонентами, що створені на основі архітектури SOA, а також створення чистого API, побудованого на інтерфейсах сервісів та створення масштабованих та реконфігурованих систем.

Даний набір бібліотек має хороші функціональні можливості, проте велика частина запланованого функціоналу ще знаходиться в розробці, тому для забезпечення повного набору функціональних можливостей розробнику доведеться написати велику кількість коду, на відміну від фреймворків інших

мов програмування, зокрема Java, де потрібно мінімальна кількість написання коду для налаштування базової конфігурації серверів.

2.8.3.2 Pistache framework

Pistache – це C++ REST фреймворк, що слугує для створення комплексних веб-серверів та повноцінних REST API.

Перевагою даного фреймворку є зручність створення та конфігурування серверів. Його основними функціями є створення багатопотокогово HTTP серверу для створення API, створення асинхронного HTTP клієнту для виконання запитів до API інших сервісів, а також створення HTTP «роутеру» для обробки запитів у C++ функції.

Даний фреймворк найкраще підходить для швидкого створення високопродуктивних REST API з можливістю проведення складних обчислень, проте даний фреймворк не містить достатньо функцій для створення повноцінної екосистеми мікросервісів, що може утруднити їх написання для розробників.

Висновки до розділу 2

Дані, що були отримані в даному розділі зіграли важливу роль у виборі платформи для побудови застосунку, та окреслили основні правила для його побудови. Звісно, кінцевий вигляд мікросервісної архітектури залежить лише від розробника та від конкретних цілей, що він намагається досягти, проте було встановлено основні елементи, що допомагають слідувати архітектурному стилю MSA та засоби для досягнення легкого масштабування та високої гнучкості та стійкості до змін системи в цілому, що може значно спростити процес розробки.

Було розглянуто основні архітектурні рішення для побудови додатку на основі MSA, проведено порівняння з монолітною архітектурою та описано істотні переваги мікросервісної архітектури, особливо для довготривалих розробок, що постійно вдосконалюються з часом.

Також розглянуто основні засоби для забезпечення взаємодії системи з її користувачем, основні засоби для пошуку та виявлення окремих мікросервісів для спрощення взаємодії між ними, а також засоби для контролю навантаженості та для забезпечення вімовостійкості цілого застосунку, а саме:

1) Сервіс API Gateway, що слугує єдиною точкою входу для користувачів системи.
2) Discovery server, що слугує для виявлення та аналізу станів всіх мікросервісів системи, а також спрощує знаходження даних сервісів для взаємодії з ними, що може працювати разом з балансувальником навантаженості для найбільш оптимального сервісу для зв’язку з ним.
3) Патерн Circuit Breaker, що виступає проміжним проксі-сервером для можливості виявлення помилок та миттєву відповідь користувачу без необхідності навантаженості інших серверів та для уникнення повторних каскадних збоїв системи, даючи при цьому можливість серверу відновитись після падіння.

Тому, при наявності даних знань про побудову внутрішньої структури програми, а також про інструменти та фреймворки реалізації мікросервісів на різних мовах програмування розробник може з легкістю створити власну систему з мікросервісною архітектурою на базовому рівні.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ПРОГРАМНОГО КОМПЛЕКСУ

В сучасному світі існує досить багато інструментів для побудови систем, що базуються на мікросервісній архітектурі, тому на початку роботи розробника постає досить важливе питання: «Який саме інструмент обрати?». Адже вибір конкретного програмного рішення може значно полегшити розробку програмного комплексу, але може і значно ускладнити розробку, не маючи потрібних інструментів для обміну та обробки даних.

3.1 Постановка задачі

В даному дипломному проєкті буде реалізовано програмний комплекс для моделювання, навчання та тестування нейронних мереж прямого поширення, оскільки даний вид нейронних мереж є найпростішим в реалізації та дозволяє вирішити широкий спектр задач, а також є простим для створення користувачем. На даному етапі було вирішено створити веб-застосунок. Оскільки до веб-застосунку може мати доступ безліч користувачів, то повинен існувати спосіб ідентифікації користувача для відображення його власних проєктів, тобто власне змодельованих нейронних мереж. Виходячи з того, що користувач може вносити в систему персональні дані, а також власні методи побудови нейромереж, то необхідною є система захисту, що допоможе при наявності імені користувача та секретного слова не лише ідентифікувати користувача, а ще й захистити дані від зловмисників.

Враховуючи те, що в нейронні мережі можуть мати розгалужений вигляд, потрібно реалізувати спосіб збереження даних, що зможе забезпечити достатню гнучкість та зручність при записі та зчитуванні даних.

Варто зазначити, що найбільшу обчислювальну складність мають операції тренування та тестування нейромереж, тому варто забезпечити модульність та гарну масштабованість програми для того, щоб виконання складних арифметичних обчислень не призвело до затримки відповідей користувачам системи.

Не менш важливим є використання найбільш оптимальних інструментів для створення модулів для обробки викликів користувача, моделювання нейромереж та обміну даних, а також і для самого модулю обчислення нейромереж.

Виходячи з поставленого завдання найкращим способом реалізації є використання мікросервісної архітектури, оскільки це дає змогу розділити завантаженість системи та проводити розрахунки розподіллено, незалежно один від одного, тобто в даному випадку при високій навантаженості сервісу по проведенню тренування нейронної мережі інші сервіси можуть бути розвантаженими та мати високу швидкість обробки даних та швидкість реагування на запити користувача.

Оскільки в програмному комплексі використовуються нейромережі, тобто набір окремих самостійних компонентів, то надзвичайно важливим є захищене з’єднання між цими компонентами, адже при наявності у зловмисника даних стосовно IP-адреси серверу та порту, на якому він працює можливо отримати всі необхідні дані за відсутності авторизації між серверами.

Наявність декількох сервісів може створити певні незручності клієнту, якому потрібно отримати дані з декількох серверів, оскільки для цього потрібно володіти всіма даними про підключення кожного сервісу, а також їх власні інтерфейси. Для вирішення даної проблеми необхідно створити сервер, що буде слугувати єдиною точкою входу до програмного комплексу, який має змогу переправляти запити клієнта до відповідного серверу, що в свою чергу

додає серверам більшої захищеності, оскільки користувач в даному випадку не буде мати уявлення про внутрішню структуру серверу.

При виконанні всіх вищевказаних умов буде отримано систему, що буде відповідати сучасним стандартам побудови застосунків, буде мати високий рівень захисту від зловмисників та буде легко масштабованою та матиме змогу витримувати високе навантаження.

3.2 Вибір набору інструментів для реалізації проєкту

Виходячи з вищевказаних вимог до проєкту найбільш доцільним є використання Java фреймворку Spring [4] для розробки цілої екосистеми застосунку, оскільки окремі модулі даного фреймворку надають зручні та сучасні інструменти для створення веб застосунків, управління базами даних а також інструменти для реалізації взаємодії у хмарному середовищі.

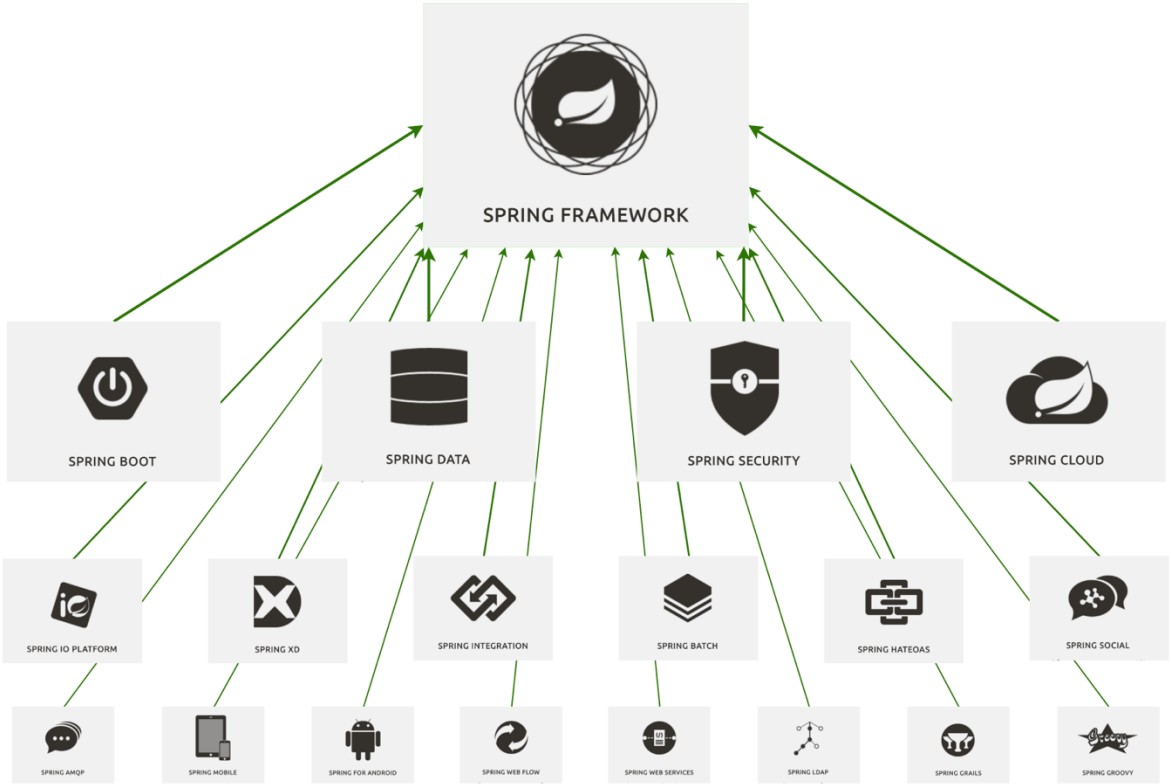


Рис. 3.1. Схематичне зображення інфраструктури фреймворку Spring

Частини даного фреймворку класифікуються згідно області їх застосування, відповідно:

- Spring Data це набір інструментів для взаємодії з БД;
- Spring Boot виступає набором інструментів для розробки веб-застосунків;
- Spring Cloud слугує для реалізації взаємодії між окремими компонентами системи;

Використання даних інструментів дозволяє розробнику значно легше досягти розробки програми, слідуючи принципам SOLID та MVC моделі (Рис. 3.2.). Вищевказані принципи дозволяють розробити архітектуру застосунку беручи до уваги можливий розвиток застосунку в майбутньому для забезпечення зрозумілої структури гарної масштабованості.

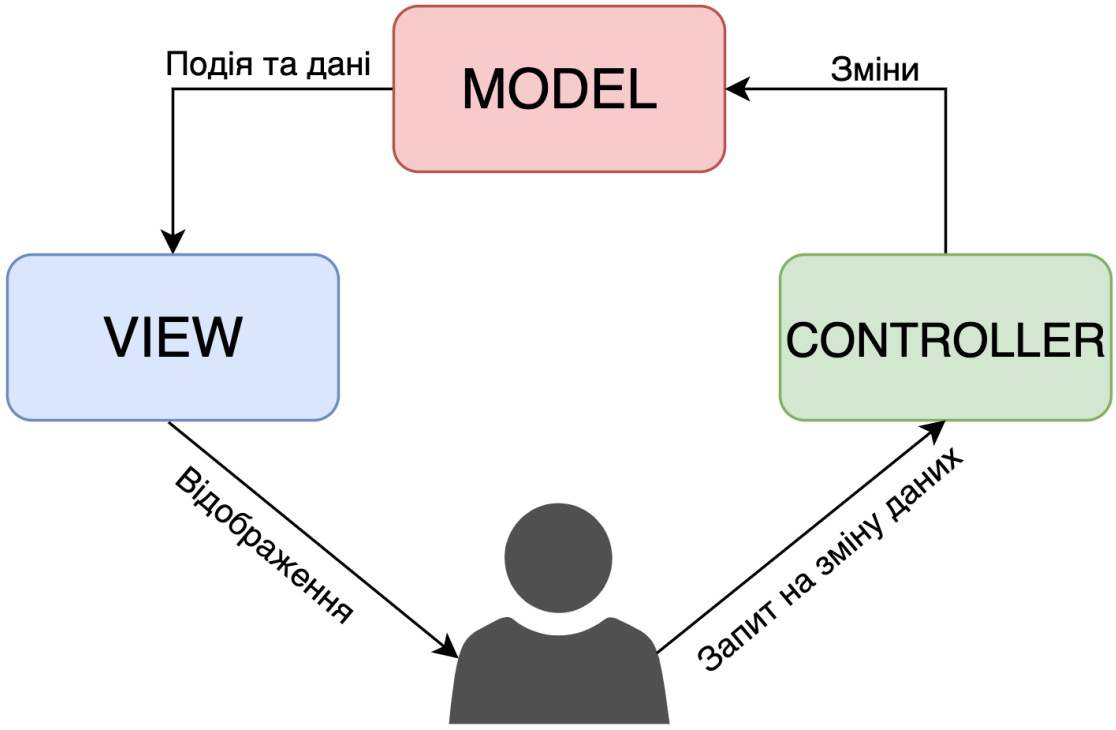


Рис. 3.2. Схематичне зображення MVC моделі

Загальна конфігурація застосунку у фреймворку Spring відбувається за допомогою анотацій, що позначаються знаком «@». За допомогою даних анотацій система визначає окремі модулі застосунку та автоматично зв'язує їх

потрібним об’єктом. Для прикладу, анотація «@Autowired» забезпечують слабку зв’язність елементів програми шляхом використання так званої «ін’єкції залежності», тобто в потрібний момент програма автоматично надає екземпляр потрібного класу для роботи іншого. При цьому клас, якому потрібна дана залежність не може створювати новий екземпляр, що дозволяє створити ієрархічну структуру даних, зв’язки якої мають єдиний напрямок, що є ключовим в ідеології SOLID.

Аналогічним чином фреймворк Spring забезпечує реалізацію архітектурного патерну Model — View — Controller за рахунок слабко зв’язаних програмних компонентів, при цьому розділяючи між собою логіку вводу та збереження даних, бізнес-логіку, а також логіку представлення об’єкту, забезпечуючи при цьому вільний зв’язок компонентів.

Для збереження об’єктів та забезпечення зв’язку з БД фреймворк Spring має зручний механізм для роботи з сутностями бази даних, організації їх в репозиторії, збереження даних, їх зміна або видалення – Spring Data. Основна сутність в Spring Data – це репозиторій. Репозиторій має декілька інтерфейсів, найбільш поширеним є CrudRepository, що виконує основні CRUD. Запити сутності з БД можуть будувати прямо з імені методу, для цього використовується префікси find…By, read…By, query…By, count…By, и get…By, або ж бути написаними звичайним запитом за допомогою анотації @Query.

Для розробки веб-застосунку потрібен набір інструментів, що зможе надати гнучку структуру для реалізації маршрутів доступу до застосунку та окремих його модулів. Дані інструменти надає Spring Boot, який дозволяє легко створювати повноцінні застосунки, при цьому від розробника потрібно лише налаштування потрібної конфігурації запуску додатку. Spring Boot має вбудований сервер Tomcat або Jetty, що також спрощує розробку та запуск додатків. Також Spring Boot має так звані starter-пакети, що представляє собою

набір дескрипторів залежностей, які можна підключити до застосунку. Це дозволить отримати універсальне рішення для застосунків розроблених за допомогою фреймворку Spring, звільняючи розробника від довгого пошуку прикладів коду та завантаження з них потрібних дескрипторів. Для прикладу, при підключенні залежності «spring-boot-starter-web» до застосунку буде підключено всі необхідні бібліотеки, що необхідні для розробки Spring MVC застосунків, таких як spring-webmvc, jackson-json, validation-api, а також Tomcat. Дані інструменти слугують для спрощення обробки та формування відповідей на запити користувача, проводити початкову валідацію вхідних параметрів, а також побудови MVC моделі об’єктів.

Для розробки цілої системи веб-застосунків, що виходять з архітектури MSA необхідно також набір інструментів для налаштування зв’язків між застосунками. Для цього у фреймворку Spring існує набір інструментів під назвою Spring Cloud, що здобув свою популярність за свій досить короткий час існування. Spring Cloud – це модуль Spring, в якому міститься велика кількість інструментів для розробки програмних комплексів мікросервісної архітектури, включаючи вже частини готових модулів, які при наявності певного опису конфігурації застосунку. Даний модуль є одним із найбільш нових у всій інфраструктурі Spring, та за чотири роки існування отримав велику кількість прихильників та на даний час є одним із найбільш популярним інструментом для реалізації хмарних технологій.

Одним із найбільших потужних підмодулів Spring Cloud є імплементація та вдосконалення частини розробок Spring Cloud, що також має велику кількість додаткових інструментів для управління мікросервісами - Spring Cloud Netflix. Даний проєкт є напрацюванням всесвітньовідомої компанії Netflix, що здобув назву Netflix OSS, він включає в себе як набір готових рішень для управління мікросервісами, так і додаткові інструменти, такі як балансувальник навантаження Ribbon та інструмент обміну даними між

сервісами Feign. Одним із ключових реалізацій є їх спільна взаємодія. Дані інструменти дають можливість розробнику самому обирати які інструменти використовувати та широкий функціонал налаштувань, до того ж більшість інструментів можуть працювати при мінімальній кількості рядків коду розробника. Враховуючи всі плюси розробленого продукту розробники фреймворку Spring вирішили додати Spring Cloud Netflix [7] до стандартного модулю Spring Cloud, після чого кожен розробник може вільно користуватися надбанням досить успішної компанії сучасності.

Для налаштування безпеки системи фреймворк Spring має модуль spring-boot-starter-security, що виступає проміжною ланкою при обробці запитів користувача та має широкий спектр налаштувань безпеки що представлений як набір фільтрів, що дозволяє не лише ідентифікувати користувача і регулювати доступність ресурсу, а й проводити попередню обробку даних. Для забезпечення безпеки серверу при обміні даними між сервісами, а також для можливості майбутнього розширення функціоналу застосунку і створення API для зовнішнього користування існує модуль spring-cloud-starter-oauth2, що надає додатку можливість авторизації сучасним протоколом OAuth2, що на даний момент є найбільш розповсюдженим засобом захисту при побудові REST API.

Для забезпечення розробнику зручного інструменту управління залежностями проєкту слугує фреймворк Maven. Apache Maven – фреймворк, що слугує для забезпечення автоматизованої збірки проєктів на основі опису їх структури в файлах на мові POM. Фреймворк забезпечує декларативну збірку проекту, при чому в файлі конфігурації вказується специфікація проєкту, а не окремі команди, виконання яких забезпечує створення проєкту.

Інформація щодо збірки проєкту на основі Maven зберігається в конфігураційному файлі під назвою «pom.xml». Даний фреймворк слідує принципу погодження конфігурації, що полягає в тому, що розглядані аспекти

потребують конфігурації тоді і тільки тоді, коли даний аспект не задовольняє деякій специфікації. Як наслідок, це дозволяє зменшити кількість необхідної конфігурації, при чому не втрачаючи гнучкість програми. Одним із наслідків цього є відсутність необхідності явного вказання шляху до конкретного файлу, що значено спрощує вміст файлу «pom.xml».

Maven застосовує принцип архетипів – інструментів шаблону, кожен з яких визначений патерном чи моделлю, по аналогії з якими відбувається створення похідних об'єктів. Стандартна структура каталогів (Рис. 3.3) – один із найяскравіших прикладів реалізації архетипів Maven.

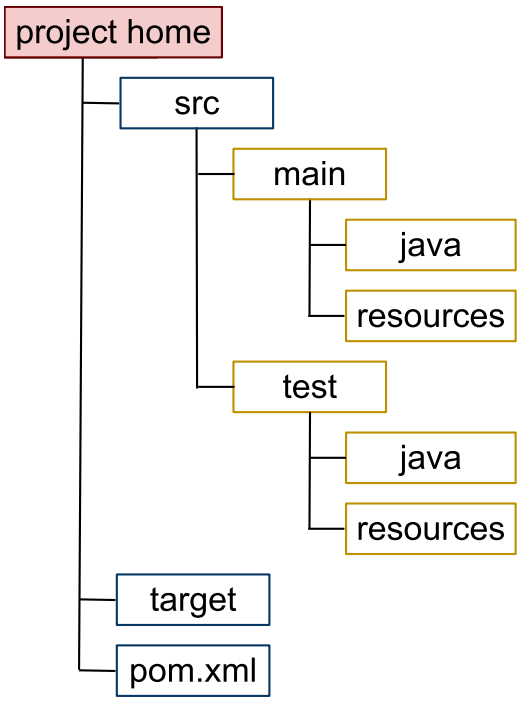


Рис. 3.3. Структура каталогу проєкту Maven на Java

Файл конфігурації застосунку знаходиться в кореневій папці проєкту, папка «src» слугує для збереження вихідних даних, а також коду програми та файли тестування, папка «target» зберігає всі створювані в процесі роботи Maven файли, а також скомпільовані класи застосунку.

Попри те, що фреймворк Spring надає велику кількість функціоналу для побудови веб-сервісів, в ньому відсутні засоби для проведення великої

кількості обчислень, тому використання даного фреймворку може призвести до написання великої кількості коду програми та не оптимально затрачених часу та зусиль, до того ж легко допустити помилки в розрахуванні нейромереж, які можуть залишись непоміченими, а також виводити некоректні дані щодо точності роботи нейронної мережі. Для виконання великої кількості обчислень, існують мови програмування, що мають велику кількість різноманітних бібліотек для обробки даних та проведення різноманітних операцій над ними.

Зокрема, мова програмування Python має досить велику кількість бібліотек по роботі з даними. Python [7] – високорівнева мова програмування загального призначення, що орієнтовано на підвищення продуктивності розробника та доброго читання програмного коду застосунків за рахунок значного спрощення синтаксису. Проте написання веб-застосунку на даній мові програмування дещо поступається можливостям фреймворку Spring, тому велику кількість інструментів та можливостей застосунку потрібно створювати самому розробнику.

Виходячи з цього сервіс, написаний на Python, добре підходить для виконання тренування нейромереж та проведення тестування. В Python існує досить потужна бібліотека для обробки нейромереж під назвою Keras. Keras [9] – нейромережева бібліотека з відкритим програмним кодом, що представляє собою імплементацію фреймворків Deeplearning4j, TensorFlow та Theano. Головна ціль даної бібліотеки – оперативна робота з мережами глибокого навчання, при цьому спроектована таким чином, щоб бути компактною, модульною та розширюваною.

Для мови програмування Python існує бібліотека для роботи з Eureka – сервісу, що входить до складу Netflix OSS в фреймворку Spring, тому це значно спростить налаштування взаємодії сервісів, оскільки в даному випадку досить легко динамічно визначити адресу потрібного серверу та відправити запит.

Також, для створення застосунків розробнику надзвичайно важлива добре читання коду та стисліть його написання. Для вирішення даної проблеми існує велика кількість додаткових інструментів для різних мов програмування. Одним із прикладів є мова програмування Kotlin [6]. Kotlin – статично-типізована, об'єктно-орієнтована мова програмування, що працює поверх JVM. Основною метою розробників даної мови програмування є створення більш лаконічною та з контролем безпеки типів з можливістю захисту застосунку від NullPointerException, а також більш простої мови програмування в порівнянні зі Scala.

Kotlin є повністю сумісним з Java, що дозволяє java-розробникам поступово перейти до його використання. Іншими словами, при розробці застосунків на Kotlin можливо використовувати всі переваги Java та Kotlin одночасно, адже на ряду з java-класами можуть існувати класи Kotlin, а головним є те, що в даній мові програмування є доступними всі бібліотеки Java.

3.3 Реалізація поставленого завдання

Проєкт (Рис. 3.4) побудований, використовуючи фреймворк Maven з використанням файлу конфігурації «Супер POM», що слугує батьківською конфігурацією для кожного з сервісів. В даному файлі визначені всі сервіси комплексу застосунків, як модулі одного застосунку.

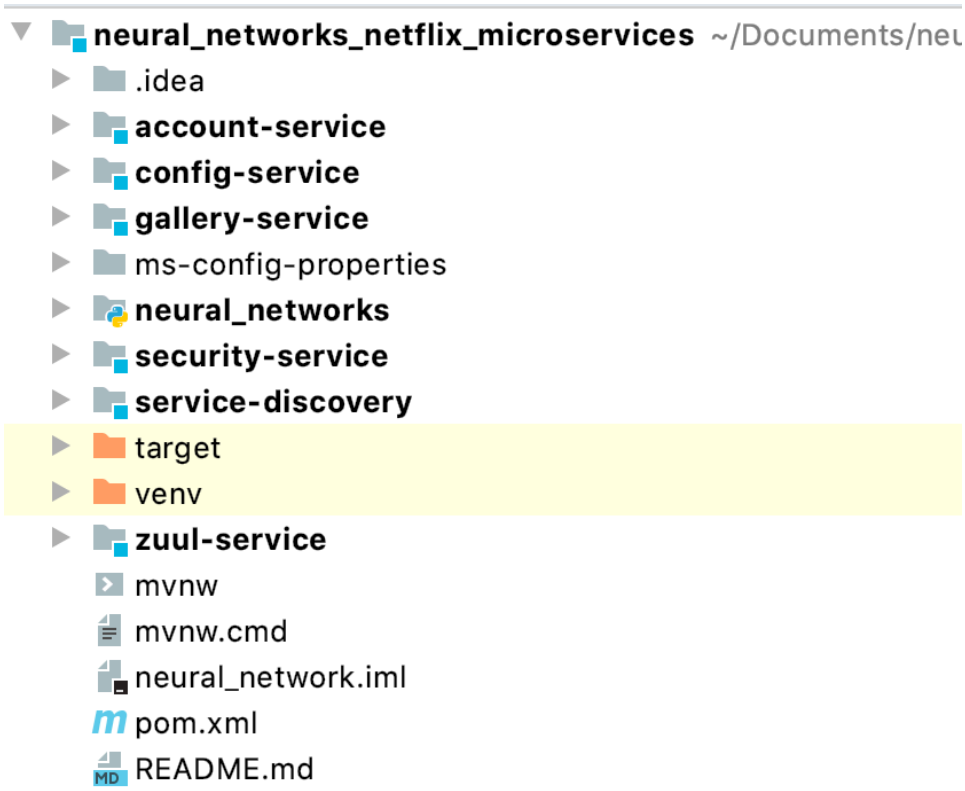


Рис. 3.4. Структура проєкту

Виходячи з постановки завдання та обраних програмних інструментів для його реалізації було розроблено 7 мікросервісів, що забезпечують роботу серверної частини застосунку, серед них:

1) Discovery service.
2) Configuration service.
3) Gateway service.
4) Authentication service.
5) Account service.
6) Gallery service.
7) Computation service.

Розглянемо детальніше внутрішню структуру кожного з наведених сервісів.

3.3.1 Discovery service

Даний сервіс є реалізацією однойменного шаблону проектування в мікросервісній архітектурі та слугує для виявлення сервісів в системі, забезпечуючи при цьому стійкий зв’язок між сервісами одного застосунку.

Основною задачею сервісу є виявлення сервісів, підтримання зв’язку з ними, збереження даних щодо IP-адрес та порту, на якому знаходиться той чи інший сервіс та його статус в системі, а також слугує для передачі даних сервісів системи по запиту.

Сервіс було реалізовано мовою програмування Kotlin, з застосуванням Maven, фреймворку Spring та модулю Spring Cloud Netflix, а саме Netflix Eureka, що слугує реалізацією патерну service discovery. Структура (Рис. 3.5) даного сервісу наслідує стандартну структуру каталогів Maven, тож містить основні його складові:

- Файл конфігурації pom.xml;
- Тека src, в якій зберігається код та параметри даного сервісу;
- Тека target, що містить файли залежностей Maven, а також скомпільовані класи сервісу;

Даний сервіс є клієнтом серверу конфігурації та при своєму запуску отримує необхідну конфігурацію.

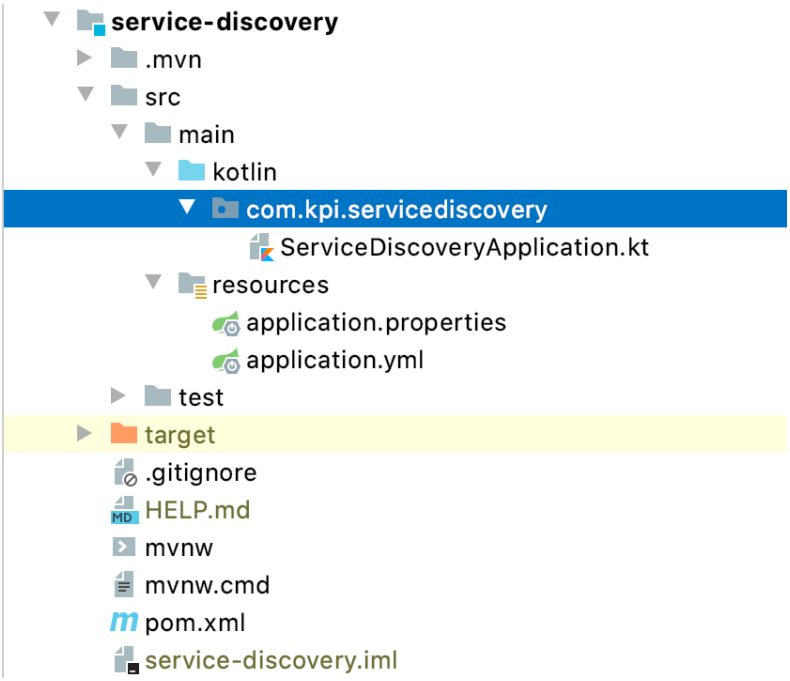


Рис. 3.5. Структура Discovery service

3.3.2 Configuration service

Основною ціллю даного сервісу (Рис. 3.6) є забезпечення всіх сервісів файлами конфігурації, що включають в себе рекомендований порт для запуску, конфігурації для отримання доступу до інших ресурсів, шляхи перевірки авторизації користувачів.

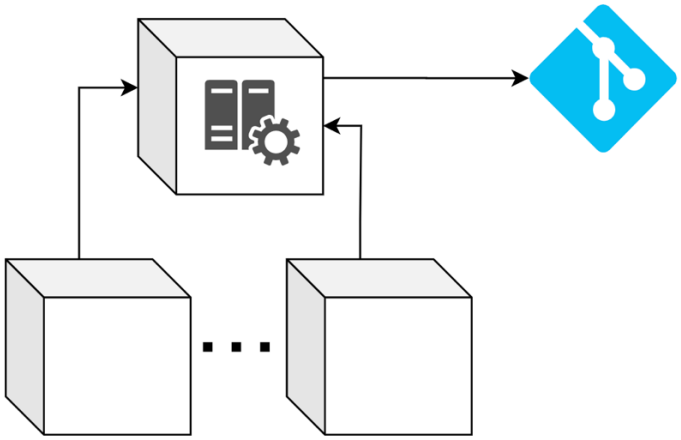


Рис. 3.6. Принцип реалізації постачання конфігурацією

При запиті одного із серверів проєкту власної конфігурації сервер отримує дані від попередньо зазначеній репозиторії в системі контролю версій, після чого відправляє отримані дані безпосередньо до потрібного сервісу. Даний процес дає можливість динамічної зміни конфігурації без виконання перезавантаження окремих сервісів.

Сервіс було реалізовано мовою програмування Kotlin, з застосуванням Maven, фреймворку Spring та модулю Spring Cloud Netflix, а саме Netflix Config Server. Структура (Рис. 3.7) даного сервісу наслідує стандартну структуру каталогів Maven, тож містить основні його складові:

- Файл конфігурації pom.xml;
- Тека src, в якій зберігається код та параметри даного сервісу;
- Тека target, що містить файли залежностей Maven, а також скомпільовані класи сервісу;

Даний сервіс є клієнтом Discovery Service та при своєму запуску реєструється в ньому.

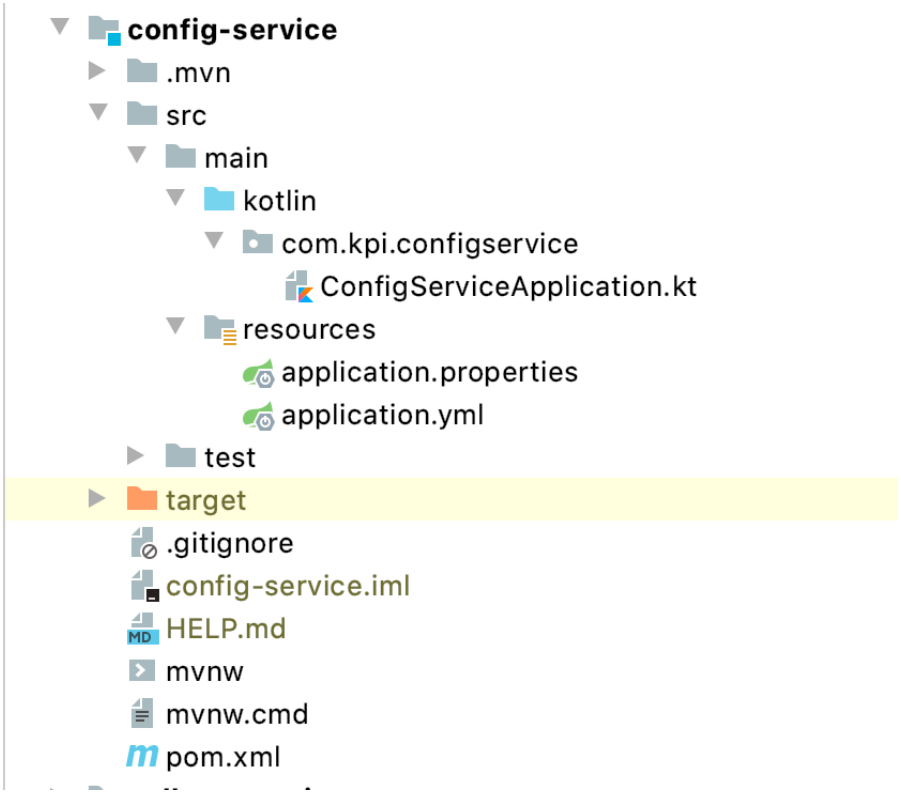


Рис. 3.7. Структура Configuration service.

3.3.3 Gateway service

Gateway service слугує єдиною точкою для цілого програмного комплексу та виконує переправлення клієнтських викликів до відповідного сервісу.

Сервіс було реалізовано мовою програмування Kotlin, з застосуванням Maven, фреймворку Spring та модулю Spring Cloud Netflix, а саме Netflix Zuul, що в свою чергу є реалізацією структурного патерну gateway, що за своєю структурою схожий з патерном Фасад в ООП, дозволяючи приховати внутрішню структуру проєкту. Структура (Рис. 3.8) даного сервісу наслідує стандартну структуру каталогів Maven, тож містить основні його складові:

- Файл конфігурації pom.xml;
- Тека src, в якій зберігається код та параметри даного сервісу;
- Тека target, що містить файли залежностей Maven, а також скомпільовані класи сервісу;

Даний сервіс є клієнтом Discovery Service та при своєму запуску реєструється в ньому, а також отримує конфігурацію з Configuration service.

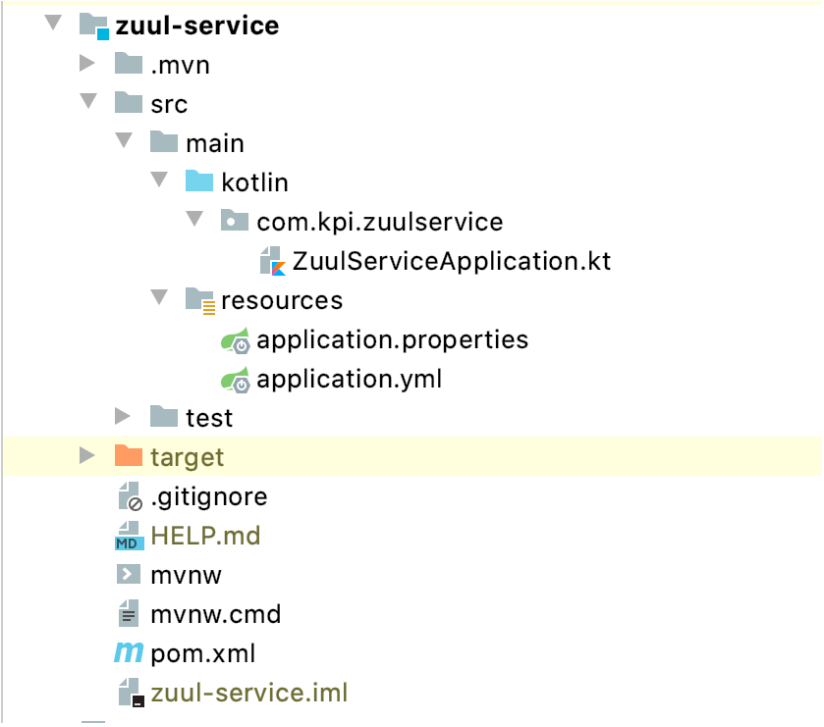


Рис. 3.7. Структура Gateway service

В своєму файлі конфігурації даний сервіс отримує список сервісів, на які в процесі роботи системи буде переправлено виклики користувачів. При цьому даний сервіс заздалегідь не має даних щодо підключення до сервісів, маючи при цьому лише назву сервісу та початковий маршрут. Спираючись на ці дані в момент отримання запиту від клієнта сервіс надсилає запит до Discovery service, після чого, спираючись на отримані дані, переправляє запит потрібному сервісу.

3.3.4 Authentication service

Authentication service слугує додатковим сервісом системи, що відповідає за авторизацію користувача та його автентифікацію. При запиті на захищений ресурс будь-якого сервісу, сервіс призначення надсилає для перевірки токен автентифікації даному сервісу, після чого отримує об’єкт

Principal, з якого може отримати необхідні дані про користувача, зокрема його ім’я та представлення в системі. Використання даного сервісу надає системі централізовану систему захисту з єдиною точкою управління доступом.

Існують реалізації, що дозволяють проводити валідацію токену користувача безпосередньо при запиті до Gateway service, а кінцевий сервіс призначення отримує вже готовий об’єкт Principal. В такій системі Authentication service слугує лише для авторизації користувача та генерації токену підключення. Проте, схожа система реалізації є надзвичайно вразливою, оскільки при наявності IP-адреси та порту підключення до деякого сервісу зловмисник, відправивши завідома фіктивні дані може отримати доступ до ресурсу.

Сервіс було реалізовано мовою програмування Kotlin, з застосуванням Maven, фреймворку Spring та інструментів spring-cloud-starter-oauth2 та spring-boot-starter-security, що відповідають за створення системи як для авторизації користувачів, так і для перевірки підключеного сервісу, що він є складовою комплексу та має дозволи на отримання даних. Структура (Рис. 3.9) даного сервісу наслідує стандартну структуру каталогів Maven, тож містить основні його складові:

- Файл конфігурації pom.xml;
- Тека src, в якій зберігається код та параметри даного сервісу;
- Тека target, що містить файли залежностей Maven, а також скомпільовані класи сервісу;

Даний сервіс є клієнтом Discovery Service та при своєму запуску реєструється в ньому, а також отримує конфігурацію з Configuration service.

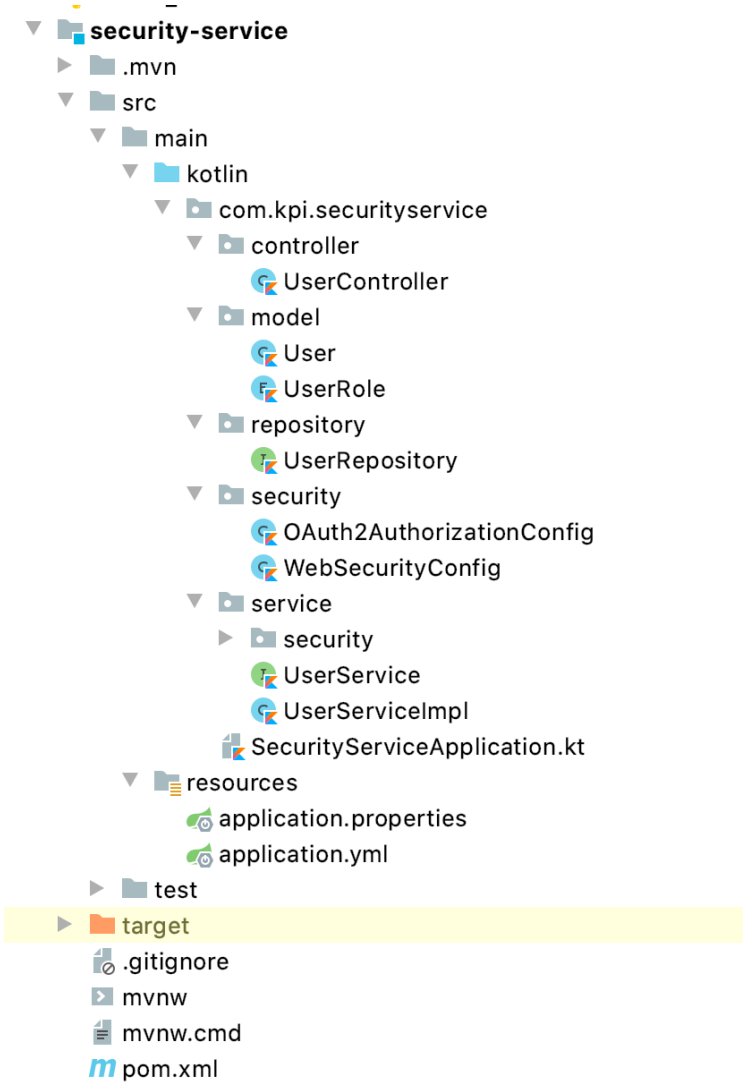


Рис. 3.9. Структура Authentication service

Authentication service має підключення до бази даних MongoDB та зберігає дані користувача, такі як ім'я представлення в системі, пароль та права доступу, що було надано користувачу.

3.3.5 Account service

Account service – сервіс призначений для управління та збереження додаткової інформації користувача та персональні дані, формує наповненням даних для особистої сторінки користувача графічного інтерфейсу.

Сервіс було реалізовано мовою програмування Kotlin, з застосуванням Maven, фреймворку Spring та інструментів spring-cloud-starter-oauth2 та spring-

boot-starter-security, що відповідають за створення системи для валідації даних користувачів і для реєстрації нових, а також виконує перевірку підключеного сервісу, що він є складовою комплексу та має дозволи на отримання даних. Структура (Рис. 3.10) даного сервісу наслідує стандартну структуру каталогів Maven, тож містить основні його складові:

- Файл конфігурації pom.xml;
- Тека src, в якій зберігається код та параметри даного сервісу;
- Тека target, що містить файли залежностей Maven, а також скомпільовані класи сервісу;

Даний сервіс є клієнтом Discovery Service та при своєму запуску реєструється в ньому, а також отримує конфігурацію з Configuration service.

Сервіс має підключення до бази даних MongoDB та зберігає особисті дані користувача, а також при реєстрації відправляє дані користувача до Authentication service за допомогу інструменту Spring Cloud Netflix Feign, що маючи лише назву сервісу призначення встановлює зв’язок на основі отриманих даних від Discovery service. Account service має в свої структурі аналогічний клас User, як в Authentication service, що позбавлений основного функціоналу та слугує контейнером для передачі даних.

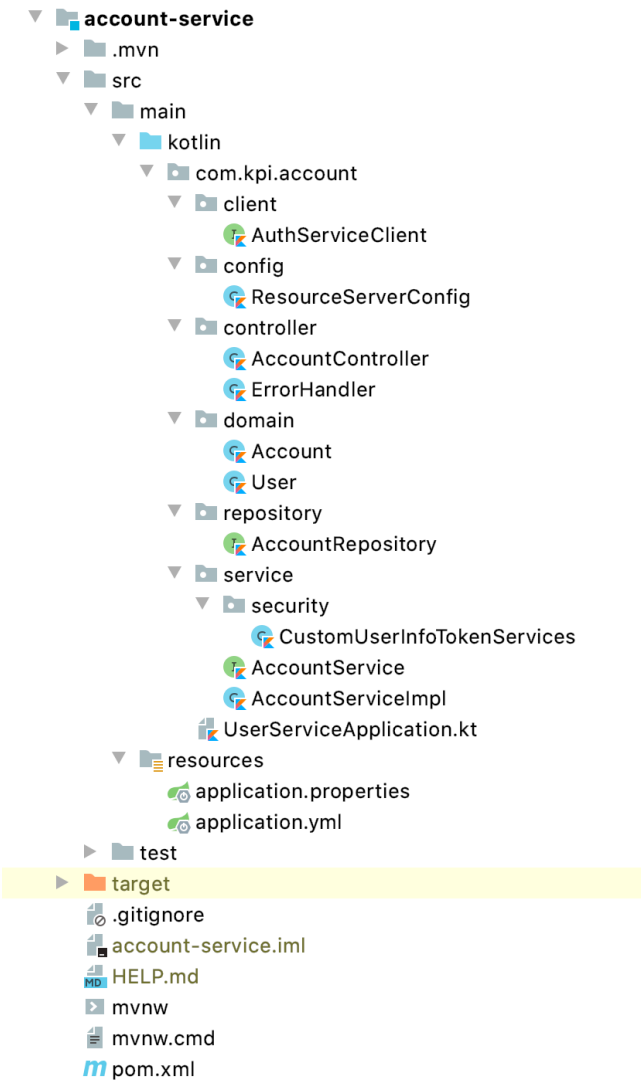


Рис. 3.10 Структура Account service

3.3.6 Gallery service

Gallery service призначений для створення, редагування, видалення нейронних мереж, для нормалізації представлення даних, а також для збереження та отримання статистики щодо коректності роботи застосунку.

Сервіс було реалізовано мовою програмування Kotlin, з застосуванням Maven, фреймворку Spring та інструментів spring-cloud-starter-oauth2 та spring-boot-starter-security, що відповідають за створення системи для валідації даних нейромереж і для реєстрації нових, а також виконує перевірку підключеного сервісу, що він є складовою комплексу та має дозволи на отримання даних.

Структура (Рис. 3.11) даного сервісу наслідує стандартну структуру каталогів Maven, тож містить основні його складові:

- Файл конфігурації pom.xml;
- Тека src, в якій зберігається код та параметри даного сервісу;
- Тека target, що містить файли залежностей Maven, а також скомпільовані класи сервісу;

Даний сервіс є клієнтом Discovery Service та при своєму запуску реєструється в ньому, а також отримує конфігурацію з Configuration service.

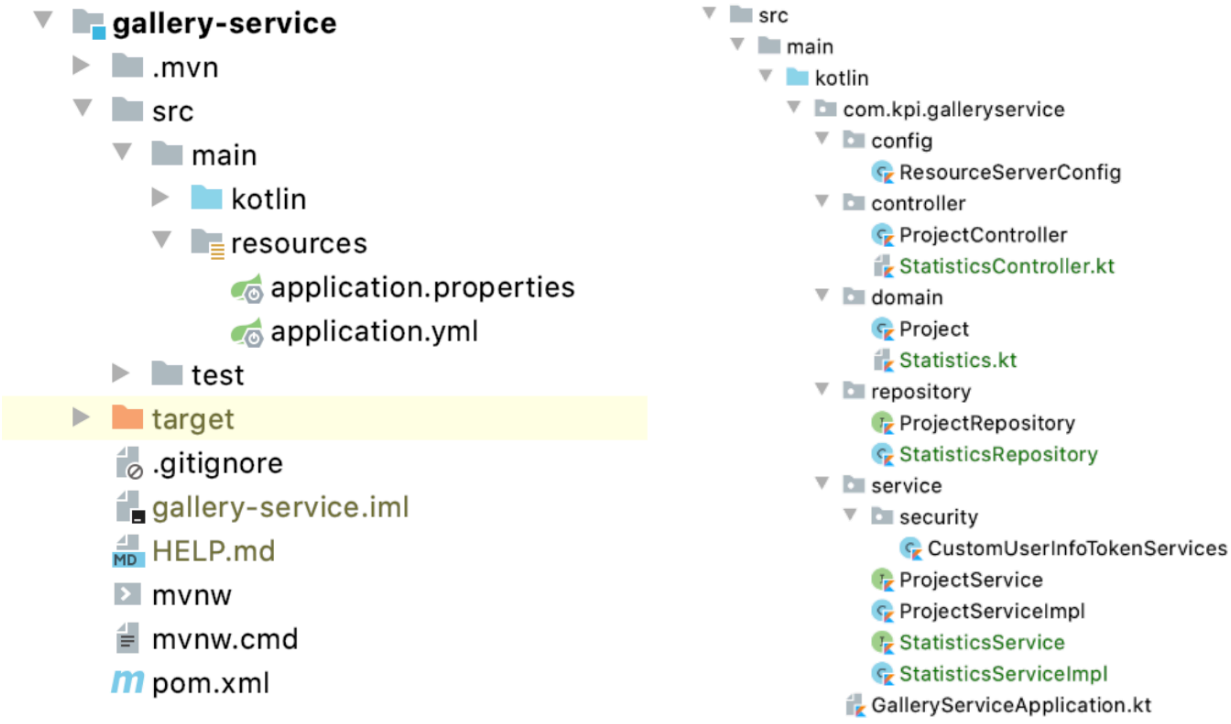


Рис. 3.11. Структура Gallery service

3.3.7 Computation service

Головною ціллю Computation service є проведення тренування та тестування вже створених моделей нейромереж, що пройшли валідацію та нормалізацію даних, а також формування та збереження статистичних даних щодо їх виконання.

Сервіс було реалізовано мовою програмування Python, з застосуванням фреймворку Flask та інструментів eureka_client та tensorflow.keras, що відповідають за створення системи для реалізації тренування та тестування нейромереж і для формування статистики. Структура (Рис. 3.12) даного сервісу наслідує стандартну структуру проєктів Python, тож містить основні його складові:

- Головний файл програми app.py;
- Тека modules, в якій зберігається код та параметри для калькуляції нейромереж;
- Тека venv, що містить файли залежностей проєкту, а також підключені елементи середовища;

Даний сервіс є клієнтом Discovery Service та при своєму запуску реєструється в ньому.

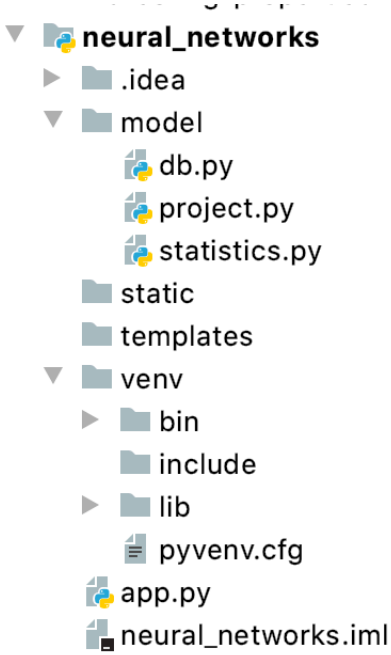


Рис. 3.12. Структура Computation service

Виконання обчислень над моделями нейронних мереж відбувається за участю бібліотеки tensorflow.keras, що надає широкий вибір функціоналу для обчислень різних за своєю структурою нейромереж.

Висновки до розділу 3

В даному розділі було описано внутрішню структуру кожного модулю проєкту, так і всього проєкту в цілому. Було описано основні модулі, фреймворки та бібліотеки, що були застосовані для розробки проєкту та перелічено основні їх властивості та функціональні можливості.

Під час розробки дипломного проєкту було використано 2 мови програмування: Kotlin та Python, використано 4 фреймворки:

- Spring для побудови серверів та налаштування взаємодії між ними для модулів, написаних на Kotlin;
- Maven для спрощення збірки серверів шляхом декларативного вказання залежностей на Kotlin;
- Keras для виконання дій над моделями нейронних мереж для модуля на Python;
- Flask для створення REST API для забезпечення обміну між сервісами на Python.

При успішному запуску програмного комплексу можна перевірити статуси всі сервісів в інтерфейсі Discovery service (Рис. 3.13).

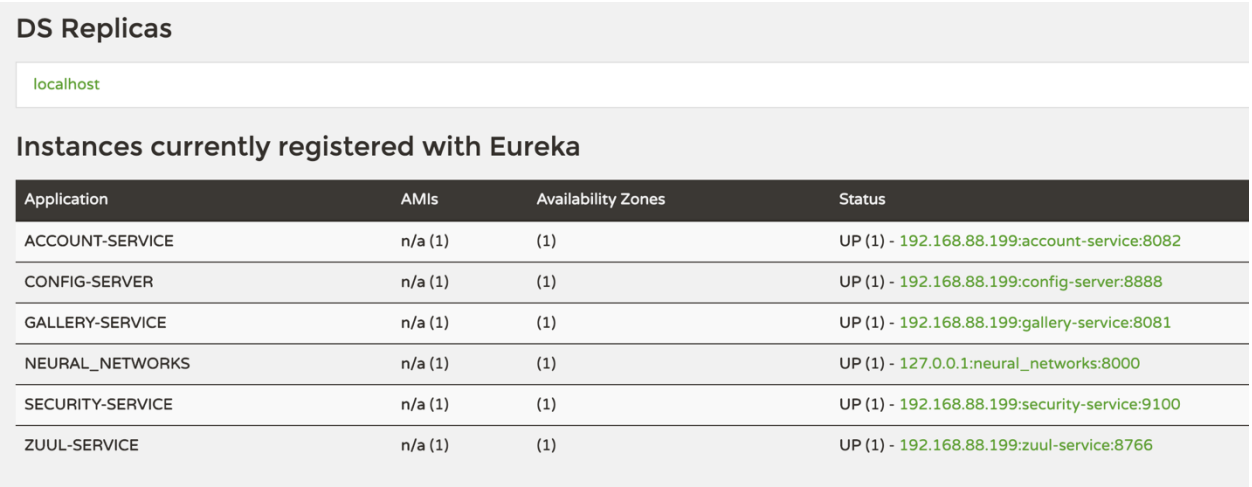


Рис. 3.13. Статуси сервісів в інтерфейсі Discovery service

ВИСНОВКИ

Метою цієї дипломної роботи є розробка системи для моделювання нейронних мереж прямого поширення загального призначення. Після аналізу можливих варіантів реалізації проєкту було обрано веб-застосунок, що має мікросервісну архітектуру з використанням найбільш оптимальних інструментів для реалізації поставленого завдання.

У першому розділі було проаналізовано класифікацію нейронних мереж за різними ознаками та було визначено загальну модель нейромереж, що буде доступна для моделювання в проєкті. Проведено огляд принципів роботи нейромереж, розглянуто основні їх складові та правила взаємодії, описано основні формули для проведення тренування нейромереж та для їх тестування, продемонстровано роботу елементарного нейрону.

У другому розділі було проаналізовано доцільність використання архітектури MSA для даного проєкту, а також проведена порівняльна характеристика мікросервісної та монолітної архітектури. Розглянуто основні принципи побудови застосунків з мікросервісною архітектурою, розглянуто допоміжні засоби взаємодії окремих модулів системи та наявні інструменти різних мов програмування для побудови даних застосунків.

У третьому розділі дипломного проєкту було поставлено завдання щодо побудови застосунку, ключові складові взаємодії сервісів, а також описані основні вимоги до забезпечення захищеності кожного модулю окремо, так і загалом всього комплексу. Було обрано основні технічні засоби, такі як мови програмування, фреймворки, системи управління базами даних, а також окремі інструменти для забезпечення коректності та ефективності роботи системи. Також було розглянуто результуючу структуру програмного комплексу, описано основні етапи його розробки та описано основні функції, властивості та інструменти, що застосовані для кожного з них.

Розроблена система займає свою нішу на ринку застосунків, оскільки немає аналогів даному застосунку, які б мали змогу моделювання, тренування, тестування нейронних мереж, а також проводити порівняльну характеристику точності розрахунків та швидкості навчання в залежності від обраної конфігурації.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

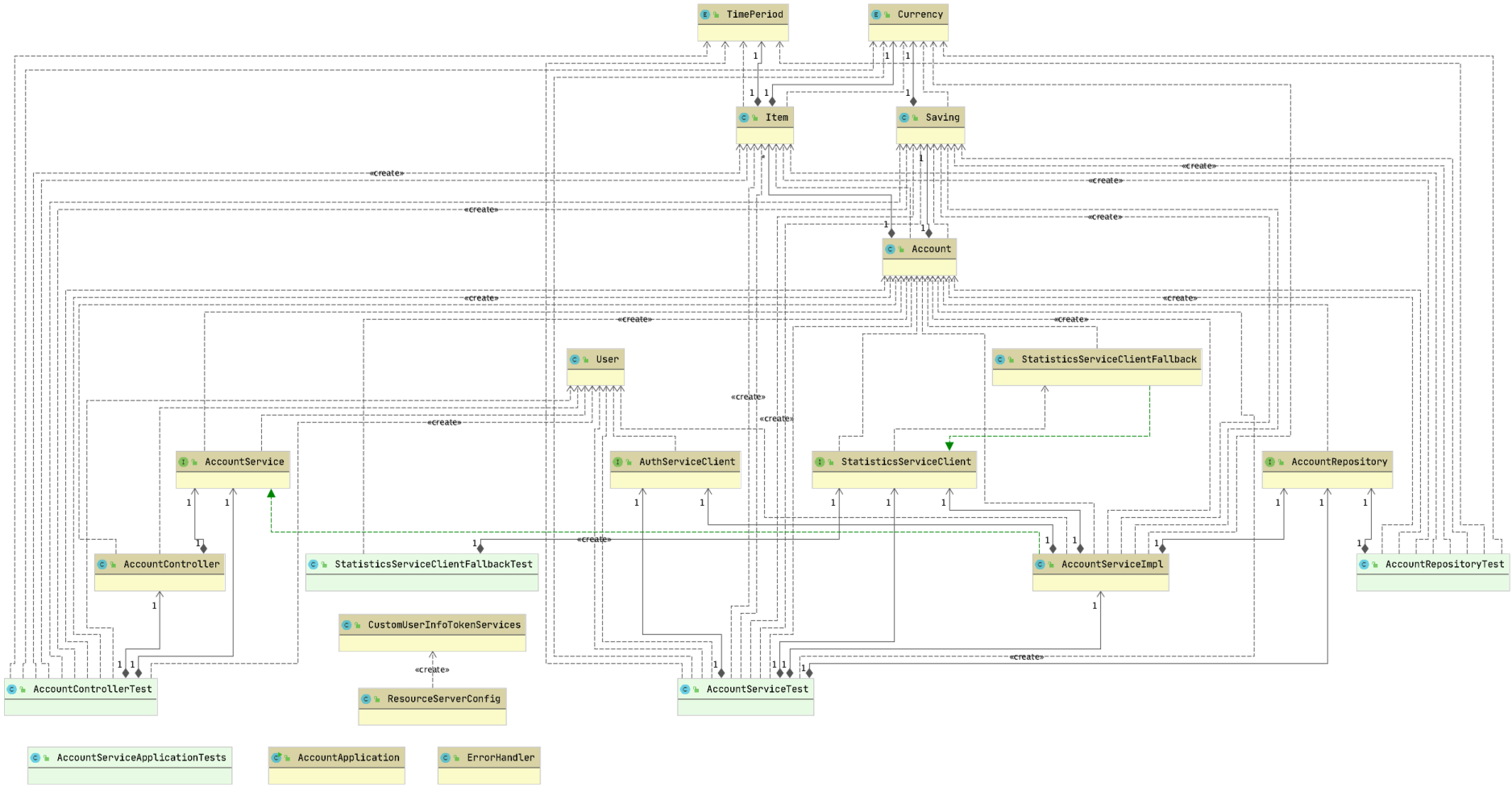
1. Нейронные сети [Електронний ресурс]: https://neuralnet.info/ (дата звернення: 03.04.2020)
2. Microservices: a definition of this new architectural term [Електронний ресурс]: https://martinfowler.com/articles/microservices.html (дата звернення: 14.02.2020)
3. Sam Newman, Building Microservices: Designing Fine-Grained Systems 1st Edition: O'Reilly Media; February 2015, ISBN: 9781491950333
4. Офіційна веб-сторінка фреймворку Spring [Електронний ресурс]: https://spring.io/projects/ (дата звернення: 17.03.2020)
5. Документация на Flask (русский перевод), Выпуск 0.10.1 февр. 24, 2019 [Електронний ресурс]: https://buildmedia.readthedocs.org/media/pdf/flask-russian-docs/latest/flask-russian-docs.pdf (дата звернення: 12.04.2020)
6. Tutorial: Reactive Spring Boot Part 1 – A Kotlin REST Service [Електронний ресурс]: https://blog.jetbrains.com/idea/2019/10/tutorial-reactive-spring-boot-a-kotlin-rest-service/?gclid=CjwKCAjwt-L2BRA_EiwAacX32ZMWtpsV7k26zKuLIZdzAvZErc8v-e4SdLsZ2Z6T9R2NStG5lAIDdBoCMMwQAvD_BwE (дата звернення: 17.03.2020)
7. Netflix OSS, Spring Cloud, or Kubernetes? How About All of Them! [Електронний ресурс]: https://blog.christianposta.com/microservices/netflix-oss-or-kubernetes-how-about-both/ (дата звернення: 19.03.2020)
8. James Loy, Neural Network Projects with Python: Packt Publishing; February 2019, ISBN: 9781789133318
9. Офіційна веб-сторінка фреймворку Keras [Електронний ресурс]: https://keras.io/ (дата звернення: 13.04.2020)

ДОДАТОК 1
Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення

Функціональна схема
ІЛАЦ.467100.004 Д1

Аркушів 1

Київ – 2020 року



					ІЛАЦ.467100.004 Д1
Зм.	*Арк.*	*№ документу*	*Підпис*	*Дата*	
Розробив		*Линник В.В.*			*Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення* *Функціональна схема*
Перевірив		*Порєв В.М.*			
Н. Контр.		*Симоненко В.П.*			
Затверд.					

Літ.	Аркуш	Аркушів
	1	1

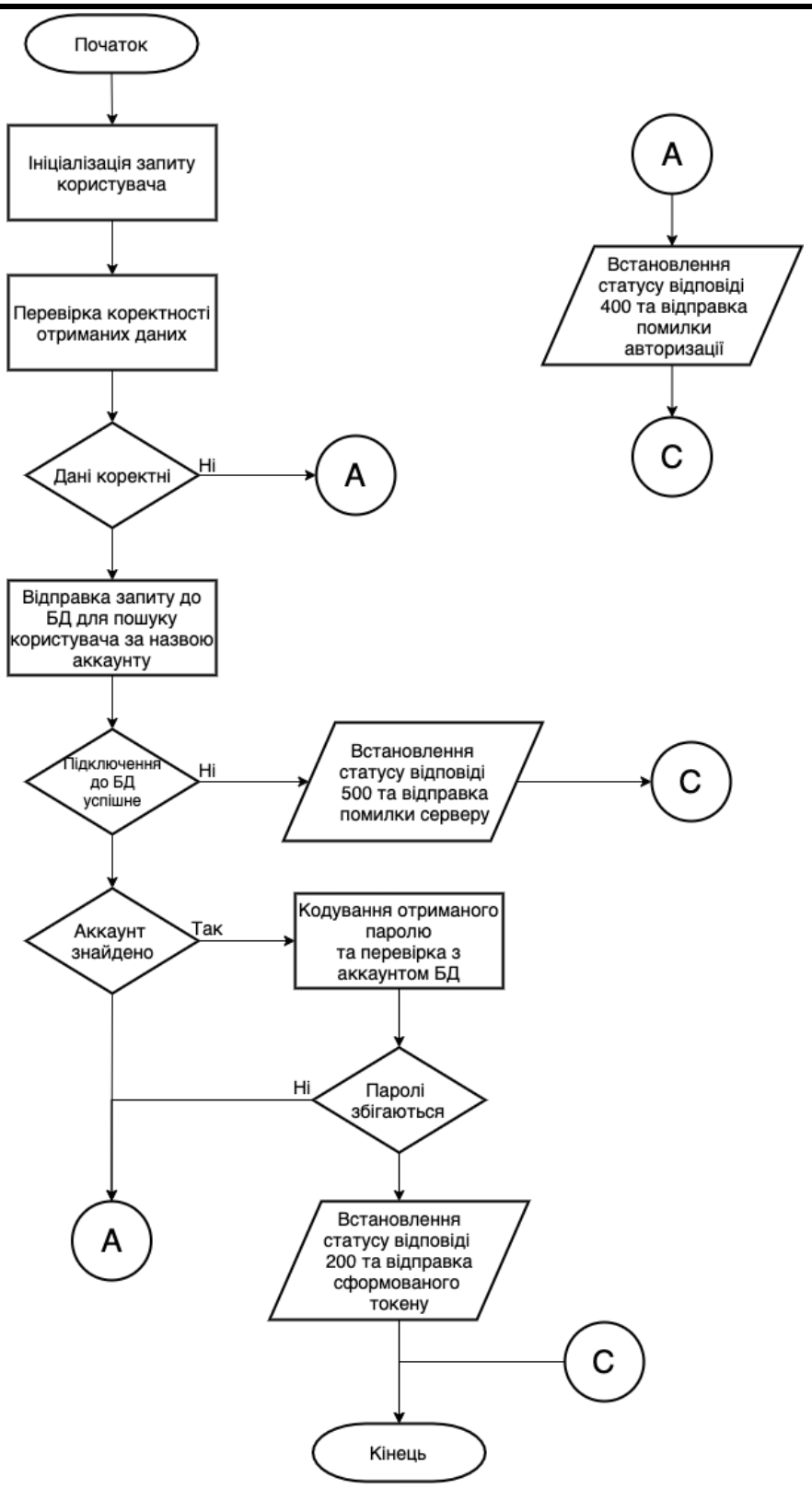
НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, гр. ІП-62

ДОДАТОК 2
Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення

Принципова схема
ІЛАЦ.467100.005 Д2

Аркушів 1

Київ – 2020 року



					ІЛАЦ.467100.005 Д2		
Зм.	*Арк.*	*№ документу*	*Підпис*	*Дата*			
Розробив		*Линник В.В.*			*Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення Принципова схема*	Літ.	Аркуш 1 / Аркушів 1
Перевірив		*Порєв В.М.*					
						НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, гр. ІП-62	
Н. Контр.		*Симоненко В.П.*					
Затверд.							

ДОДАТОК 3

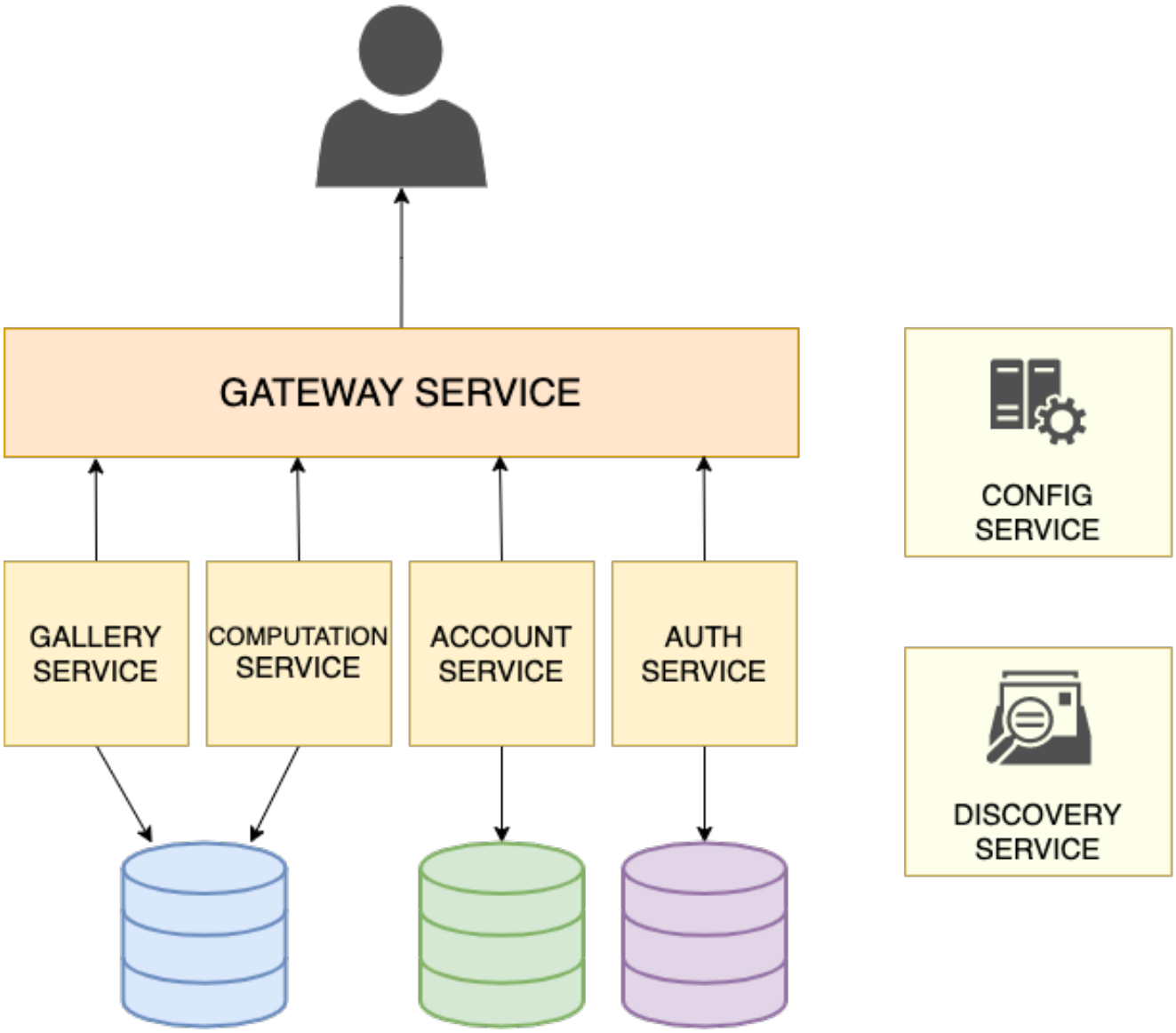
Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення

Структурна схема

ІЛАЦ.467100.006 Д3

Аркушів 1

Київ – 2020 року



					ІЛАЦ.467100.006 Д3		
Зм.	*Арк.*	*№ документу*	*Підпис*	*Дата*			
Розробив		*Линник В.В.*			*Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення Структурна схема*	Літ.	Аркуш 1 / Аркушів 1
Перевірив		*Порєв В.М.*					
Н. Контр.		*Симоненко В.П.*				*НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, гр. ІП-62*	
Затверд.							

ДОДАТОК 4

Веб сервіс із мікросервісною архітектурою для моделювання нейромереж широкого призначення

Лістинг частин програми

ІЛАЦ.467100.007 Д4

Аркушів 1

Київ – 2020 року

Файли конфігурації сервісів

account-service

```
spring:
  data:
    mongodb:
      uri: mongodb+srv://<user>:<password>@cluster0-
edkji.mongodb.net/account?replicaSet=Cluster0-shard-
0&readPreference=primary&connectTimeoutMS=10000&authSource=admin&aut
hMechanism=SCRAM-SHA-1

eureka:
  instance:
    prefer-ip-address: true
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

security:
  oauth2:
    client:
      clientId: "account-service"
      clientSecret: {$PASSWORD}
      accessTokenUri: http://localhost:9100/uaa/oauth/token
      grant-type: client_credentia
```

```
    scope: server

resource:
    user-info-uri: http://localhost:9100/uaa/user/current

server:
  servlet:

context-path: /account
  port: 8082
```

gallery-service

```
spring:
  data:
    mongodb:
      uri: mongodb+srv://<user>:<password>@cluster0-
edkji.mongodb.net/gallery?retryWrites=true&w=majority

eureka:
  instance:
    prefer-ip-address: true
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

security:
  oauth2:
    client:
      clientId: "gallery-service"
      clientSecret: {$PASSWORD}
```

```
      accessTokenUri: http://localhost:9100/uaa/oauth/token
      grant-type: client_credentials
      scope: server
    resource:
      user-info-uri: http://localhost:9100/uaa/user/current
server:
  servlet:
    context-path: /gallery
  port: 8081
```

zuul-service

```
zuul:
  routes:

    security-service:
      stripPrefix: false
      path: /uaa/**
      service-id: security-service
      sensitiveHeaders:

    gallery-service:
      path: /gallery/**
      service-id: gallery-service
      stripPrefix: false
      sensitiveHeaders:

    account-service:
      path: /account/**
      service-id: account-service
      stripPrefix: false
```

```
      sensitiveHeaders:

  ignoredServices: '*'
  host:
    connect-timeout-millis: 20000
    socket-timeout-millis: 20000

eureka:
  instance:
    prefer-ip-address: true
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

security:
  oauth2:
    resource:
      user-info-uri: http://auth-service:5000/uaa/users/current

server:
  port: 8766
```

Security service

SecurityServiceApplication

```
package com.kpi.securityservice

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.netflix.eureka.EnableEurekaClient
import
```

```
org.springframework.security.config.annotation.method.configuration.EnableGlob
alMethodSecurity
import
org.springframework.security.oauth2.config.annotation.web.configuration.EnableR
esourceServer

@SpringBootApplication
@EnableEurekaClient
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
class SecurityServiceApplication

fun main(args: Array<String>) {
  runApplication<SecurityServiceApplication>(*args)
}
```

MongoUserDetailsService

```
package com.kpi.securityservice.service.security

import com.kpi.securityservice.repository.UserRepository
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.security.core.userdetails.UserDetails
import org.springframework.security.core.userdetails.UserDetailsService
import
org.springframework.security.core.userdetails.UsernameNotFoundException
import org.springframework.stereotype.Service


@Service
class MongoUserDetailsService: UserDetailsService {

   @Autowired
   private lateinit var repository: UserRepository

   override fun loadUserByUsername(username: String): UserDetails {
      return repository.findById(username).orElseThrow {
```

```
UsernameNotFoundException(username) }
    }
}
```

WebSecurityConfig

```
package com.kpi.securityservice.security

import com.kpi.securityservice.service.security.MongoUserDetailsService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.security.authentication.AuthenticationManager
import
org.springframework.security.config.annotation.authentication.builders.Authentica
tionManagerBuilder
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import
org.springframework.security.config.annotation.web.configuration.WebSecurityCo
nfigurerAdapter
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder


@Configuration
class WebSecurityConfig : WebSecurityConfigurerAdapter() {
    @Autowired
    private lateinit var userDetailsService: MongoUserDetailsService

    @Throws(Exception::class)
    override fun configure(http: HttpSecurity) {
        http
                .authorizeRequests().anyRequest().authenticated()
                .and()
                .csrf().disable()
    }

    @Throws(Exception::class)
    override fun configure(auth: AuthenticationManagerBuilder) {
        auth.userDetailsService(userDetailsService)
                .passwordEncoder(BCryptPasswordEncoder())
    }
```

```
    @Bean
    @Throws(Exception::class)
    override fun authenticationManagerBean(): AuthenticationManager {
        return super.authenticationManagerBean()
    }
}
```

OAuth2AuthorizationConfig

```
package com.kpi.securityservice.security

import com.kpi.securityservice.service.security.MongoUserDetailsService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.beans.factory.annotation.Qualifier
import org.springframework.context.annotation.Configuration
import org.springframework.core.env.Environment
import org.springframework.security.authentication.AuthenticationManager
import org.springframework.security.crypto.password.NoOpPasswordEncoder
import
org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsS
erviceConfigurer
import
org.springframework.security.oauth2.config.annotation.web.configuration.Authori
zationServerConfigurerAdapter
import
org.springframework.security.oauth2.config.annotation.web.configuration.Enable
AuthorizationServer
import
org.springframework.security.oauth2.config.annotation.web.configurers.Authorizat
ionServerEndpointsConfigurer
import
org.springframework.security.oauth2.config.annotation.web.configurers.Authorizat
ionServerSecurityConfigurer
import org.springframework.security.oauth2.provider.token.TokenStore
import
org.springframework.security.oauth2.provider.token.store.InMemoryTokenStore
```

```
@Configuration
@EnableAuthorizationServer
class OAuth2AuthorizationConfig : AuthorizationServerConfigurerAdapter() {
    private val tokenStore: TokenStore = InMemoryTokenStore()
    private val NOOP_PASSWORD_ENCODE = "{noop}"

    @Autowired
    @Qualifier("authenticationManagerBean")
    private lateinit var authenticationManager: AuthenticationManager

    @Autowired
    private lateinit var userDetailsService: MongoUserDetailsService

    @Autowired
    private lateinit var env: Environment

    @Throws(Exception::class)
    override fun configure(clients: ClientDetailsServiceConfigurer) {

        // TODO persist clients details

        // @formatter:off
        clients.inMemory()
                .withClient("browser")
                .authorizedGrantTypes("refresh_token", "password")
                .scopes("ui")
                .and()
                .withClient("account-service")
                .secret(env.getProperty("ACCOUNT_SERVICE_PASSWORD"))
                .authorizedGrantTypes("client_credentials", "refresh_token")
                .scopes("server")
                .and()
                .withClient("gallery-service")
                .secret(env.getProperty("ACCOUNT_SERVICE_PASSWORD"))
                .authorizedGrantTypes("client_credentials", "refresh_token")
```

```
            .scopes("server")
        // @formatter:on
    }

    @Throws(Exception::class)
    override fun configure(endpoints: AuthorizationServerEndpointsConfigurer) {
        endpoints
                .tokenStore(tokenStore)
                .authenticationManager(authenticationManager)
                .userDetailsService(userDetailsService)
    }

    @Throws(Exception::class)
    override fun configure(oauthServer: AuthorizationServerSecurityConfigurer) {
        oauthServer
                .tokenKeyAccess("permitAll()")
                .checkTokenAccess("isAuthenticated()")
                .passwordEncoder(NoOpPasswordEncoder.getInstance())
    }
}
```

UserRepository

```
package com.kpi.securityservice.repository

import com.kpi.securityservice.model.User
import org.springframework.data.repository.CrudRepository
import org.springframework.stereotype.Repository

@Repository
interface UserRepository: CrudRepository<User, String> {
}
```

User

```
package com.kpi.securityservice.model

import org.springframework.data.annotation.Id
import org.springframework.data.mongodb.core.mapping.Document
import org.springframework.security.core.GrantedAuthority
import org.springframework.security.core.userdetails.UserDetails

@Document(collection = "users")
class User(@Id private var username: String, private var password: String):
UserDetails{


    override fun getAuthorities(): MutableCollection<out GrantedAuthority>? {
        return null
    }

    override fun isEnabled(): Boolean {
        return true
    }

    override fun getUsername(): String {
        return this.username
    }

    fun setUsername(username: String) {
        this.username = username
    }

    fun setPassword(password: String) {
        this.password = password
    }

    override fun isCredentialsNonExpired(): Boolean {
        return true
    }
```

```
    override fun getPassword(): String {
        return password
    }

    override fun isAccountNonExpired(): Boolean {
        return true
    }

    override fun isAccountNonLocked(): Boolean {
        return true
    }

}
```

UserController

```
package com.kpi.securityservice.controller

import com.kpi.securityservice.model.User
import com.kpi.securityservice.service.UserService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.security.access.prepost.PreAuthorize
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
import org.springframework.web.bind.annotation.RestController
import java.security.Principal
import javax.validation.Valid


@RestController
@RequestMapping("/user")
class UserController {

    @Autowired
    private lateinit var userService: UserService
```

```
    @RequestMapping(value = ["/current"], method = [RequestMethod.GET])
    fun getUser(principal: Principal): Principal {
        return principal
    }

    @PreAuthorize("#oauth2.hasScope('server')")
    @RequestMapping(method = [RequestMethod.POST])
    fun createUser(@RequestBody user: @Valid User) {
        userService.create(user)
    }
}
```

Account service

AccountApplication

```
package com.kpi.account

import org.springframework.boot.autoconfigure.SpringBootApplication
import
org.springframework.boot.autoconfigure.security.oauth2.client.EnableOAuth2Sso
import org.springframework.boot.runApplication
import org.springframework.cloud.netflix.eureka.EnableEurekaClient
import org.springframework.cloud.openfeign.EnableFeignClients
import
org.springframework.security.config.annotation.method.configuration.EnableGlob
alMethodSecurity
import
org.springframework.security.oauth2.config.annotation.web.configuration.Enable
OAuth2Client

@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
@EnableOAuth2Client
@EnableGlobalMethodSecurity(prePostEnabled = true)
class AccountApplication()

fun main(args: Array<String>) {
```

```
    runApplication<AccountApplication>(*args)
}
```

AccountService

```
package com.kpi.account.service

import com.kpi.account.domain.Account
import com.kpi.account.domain.User

interface AccountService {

    fun findByName(accountName: String): Account

    fun create(user: User): Account

    fun saveChanges(name: String, update: Account)
}
```

AccountServiceImpl

```
package com.kpi.account.service

import com.kpi.account.client.AuthServiceClient
import com.kpi.account.domain.Account
import com.kpi.account.domain.User
import com.kpi.account.repository.AccountRepository
import org.slf4j.Logger
import org.slf4j.LoggerFactory
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service
import org.springframework.util.Assert
import java.math.BigDecimal
import java.util.*


@Service
```

```
class AccountServiceImpl : AccountService {
    private val log: Logger = LoggerFactory.getLogger(javaClass)

    @Autowired
    private lateinit var authClient: AuthServiceClient

    @Autowired
    private lateinit var repository: AccountRepository


    override fun findByName(accountName: String): Account {
        Assert.hasLength(accountName)
        return repository.findByName(accountName)
    }

    override fun create(user: User): Account {
        val existing: Account = repository.findByName(user.username)
        Assert.isNull(existing, "account already exists: " + user.username)
        authClient.createUser(user)
        val account = Account(user.username, Date())
        repository.save(account)
        log.info("new account has been created: " + account.name)
        return account
    }

    override fun saveChanges(name: String, update: Account) {
        val account: Account = repository.findByName(name)
        Assert.notNull(account, "can't find account with name $name")
        account.note = update.note
        account.lastSeen = Date()
        repository.save(account)
        log.debug("account {} changes has been saved", name)
    }
}
```

CustomUserInfoTokenServices

```
package com.kpi.account.service.security

import org.apache.commons.logging.Log
import org.apache.commons.logging.LogFactory
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.autoconfigure.security.oauth2.resource.AuthoritiesExtra
ctor
import
org.springframework.boot.autoconfigure.security.oauth2.resource.FixedAuthorities
Extractor
import
org.springframework.security.authentication.UsernamePasswordAuthenticationTo
ken
import org.springframework.security.core.AuthenticationException
import org.springframework.security.core.GrantedAuthority
import org.springframework.security.oauth2.client.OAuth2RestOperations
import org.springframework.security.oauth2.client.OAuth2RestTemplate
import
org.springframework.security.oauth2.client.resource.BaseOAuth2ProtectedResour
ceDetails
import org.springframework.security.oauth2.common.DefaultOAuth2AccessToken
import org.springframework.security.oauth2.common.OAuth2AccessToken
import
org.springframework.security.oauth2.common.exceptions.InvalidTokenException
import org.springframework.security.oauth2.provider.OAuth2Authentication
import org.springframework.security.oauth2.provider.OAuth2Request
import
org.springframework.security.oauth2.provider.token.ResourceServerTokenService
s
import java.util.*
import java.util.Collections.singletonMap
import kotlin.collections.LinkedHashSet
```

```
class CustomUserInfoTokenServices(private val userInfoEndpointUrl: String,
private val clientId: String) : ResourceServerTokenServices {
    protected val logger: Log = LogFactory.getLog(javaClass)
    @Autowired
    private lateinit var restTemplate: OAuth2RestOperations
    private var tokenType: String = DefaultOAuth2AccessToken.BEARER_TYPE
    private var authoritiesExtractor: AuthoritiesExtractor =
FixedAuthoritiesExtractor()

    fun setTokenType(tokenType: String) {
        this.tokenType = tokenType
    }

    fun setRestTemplate(restTemplate: OAuth2RestOperations) {
        this.restTemplate = restTemplate
    }

    fun setAuthoritiesExtractor(authoritiesExtractor: AuthoritiesExtractor) {
        this.authoritiesExtractor = authoritiesExtractor
    }

    @Throws(AuthenticationException::class, InvalidTokenException::class)
    override fun loadAuthentication(accessToken: String): OAuth2Authentication {
        val map = getMap(userInfoEndpointUrl, accessToken)
        if (map != null) {
            if (map.containsKey("error")) {
                logger.debug("userinfo returned error: " + map["error"])
                throw InvalidTokenException(accessToken)
            }
        }
        return extractAuthentication(map)
    }

    private fun extractAuthentication(map: MutableMap<*, *>?):
OAuth2Authentication {
        val principal = getPrincipal(map)
```

```
    val request: OAuth2Request = getRequest(map)
    val authorities: List<GrantedAuthority> = authoritiesExtractor
            .extractAuthorities(map as MutableMap<String, Any>)
    val token = UsernamePasswordAuthenticationToken(
            principal, "N/A", authorities)
    token.details = map
    return OAuth2Authentication(request, token)
}


private fun getPrincipal(map: MutableMap<*, *>?): Any? {
    for (key in PRINCIPAL_KEYS) {
        if (map != null) {
            if (map.containsKey(key)) {
                return map[key]
            }
        }
    }
    return "unknown"
}

@Suppress("UNCHECKED_CAST")
private fun getRequest(map: MutableMap<*, *>?): OAuth2Request {
    val request = map?.get("oauth2Request") as Map<String, Any>
    val clientId = request["clientId"] as String
    val scope: Set<String> =
            if (request.containsKey("scope")) {
        LinkedHashSet(request["scope"] as Collection<String>)
    }
    else {
        LinkedHashSet(emptySet<String>())
    }


    return OAuth2Request(null, clientId, null, true, HashSet(scope),
            null, null, null, null)
```

```
    }

    override fun readAccessToken(accessToken: String): OAuth2AccessToken {
        throw UnsupportedOperationException("Not supported: read access token")
    }

    @Suppress("UNCHECKED_CAST")
    private fun getMap(path: String, accessToken: String): MutableMap<*, *>? {
        logger.debug("Getting user info from: $path")
        try {
            var restTemplate: OAuth2RestOperations? = this.restTemplate
            if (restTemplate == null) {
                val resource = BaseOAuth2ProtectedResourceDetails()
                resource.clientId = this.clientId
                restTemplate = OAuth2RestTemplate(resource)
            }
            val existingToken: OAuth2AccessToken? =
restTemplate.oAuth2ClientContext
                    .accessToken
            if (existingToken == null || accessToken != existingToken.value) {
                val token = DefaultOAuth2AccessToken(
                        accessToken)
                token.tokenType = this.tokenType
                restTemplate.oAuth2ClientContext.accessToken = token
            }
            return restTemplate.getForEntity(path, MutableMap::class.java).body
        } catch (ex: Exception) {
            logger.info("Could not fetch user details: " + ex.javaClass + ", "
                    + ex.message)
            return singletonMap<String, Any>("error",
            "Could not fetch user details")
        }
    }

    companion object {
        private val PRINCIPAL_KEYS = arrayOf("user", "username",
```

```
            "userid", "user_id", "login", "id", "name")
    }
}
```

AuthServiceClient

```
package com.kpi.account.client

import com.kpi.account.domain.User
import org.springframework.cloud.openfeign.FeignClient
import org.springframework.http.MediaType
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod


@FeignClient(name = "security-service")
interface AuthServiceClient {
    @RequestMapping(method = [RequestMethod.POST], value = ["/uaa/user"],
consumes = [MediaType.APPLICATION_JSON_UTF8_VALUE])
    fun createUser(user: User)
}
```