

Міністерство освіти і науки України
Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки

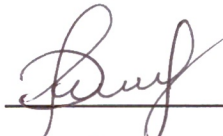
ПОЯСНЮВАЛЬНА ЗАПИСКА

до випускної кваліфікаційної роботи магістра
на тему: «Методи та технології обміну даними з
використанням безпечних API»

Виконав студент 2-го курсу групи КБМ-22-1
спеціальності 125 «Кібербезпека»

 А.Ю. Сірик

Керівник старший викладач кафедри комп'ютерної
інженерії та кібербезпеки

 Є. М. Трокоз

Рецензент доцент кафедри комп'ютерної інженерії та
кібербезпеки, к.пед.н., доцент

 О. С. Головня

Житомир – 2023

Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки
Спеціальність 125 «Кібербезпека»
Освітня програма «Кібербезпека»

ЗАТВЕРДЖУЮ

Завідувач

кафедри комп'ютерної
інженерії та кібербезпеки

 Андрій Єфіменко

13 жовтня 2023 р.

ЗАВДАННЯ

на випускн кваліфікаційну роботу магістра

Студент: Сірик Андрій Юрійович

Тема роботи: «Методи та технології обміну даними з використанням безпечних API»

затверджена Наказом університету від 13 жовтня 2023 р. № 572/с

Термін здачі студентом закінченої роботи _____

Вихідні дані роботи (зазначається предмет і об'єкт дослідження): Проєкт безпечного обміну документами за допомогою API;

Обладнання та програмне забезпечення:

1. Персональний комп'ютер Intel Core i5 8th;
2. ОС Windows 10;
3. Microsoft Visual Studio 2022;
4. Microsoft SQL Server Management Studio 18;
5. Microsoft Office Excel.

Консультанти з випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Граф М. С.	13.10.23 <i>М.С. Граф</i>	13.10.2023 <i>Сірик</i>

Студент *Сірик* А. Ю. Сірик
Керівник *Трокоз* Є. М. Трокоз

Календарний план

№ з/п	Назви етапів випускної кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка завдання	13.10.23-14.10.23	<i>She</i>
2	Аналіз завдання	14.10.23-16.10.23	<i>She</i>
3	Формування технічного завдання	17.10.23	<i>She</i>
4	Встановлення потрібних додатків	18.10.23-19.10.23	<i>She</i>
5	Розробка запитів для аутентифікації	20.10.23-31.10.23	<i>She</i>
6	Розробка запитів для отримання та відправлення документів	01.11.23-15.11.23	<i>She</i>
7	Розробка запитів та функцій для адміністрування	15.11.23-25.11.23	<i>She</i>
8	Розробка вивантаження на сервер	25.11.23-05.12.23	<i>She</i>
9	Тестування	05.12.23-10.12.23	<i>She</i>
10	Написання пояснювальної записки	10.12.23-20.12.23	<i>She</i>

Студент *А. Ю. Сірик* А. Ю. Сірик

Керівник *Є. М. Трокоз* Є. М. Трокоз

АНОТАЦІЯ

Пояснювальна записка до дипломної роботи «Методи та технології обміну даними з використанням безпечних API». Складається з вступу, трьох розділів, висновків, списку використаної літератури.

У дипломній роботі було досліджено особливості розробки та використання безпечних API.

У першому розділі було досліджено використання API як ключового інструменту у сучасному інформаційному суспільстві. Зазначено, що API, або інтерфейс програмування застосунків, відіграє важливу роль у веб-розробці та програмуванні, і його значення визначено як ключовий компонент сучасної технологічної екосистеми. В розділі детально розглянуто визначення API та вказано на його основні функції. API є інтерфейсом, який дозволяє різним програмам взаємодіяти одна з одною, обмінюючись даними.

У другому розділі було проаналізовано різноманітні аспекти технологій розробки безпечних API. Зокрема, досліджено протоколи та стандарти безпеки, які використовуються для забезпечення надійності та конфіденційності API. У цьому розділі висвітлені ключові аспекти технологій розробки безпечних API, що дозволяє розробникам та архітекторам створювати надійні та безпечні інтерфейси програмування додатків.

У третьому розділі було розроблено та впроваджено безпечний API для підсистеми електронного документообігу. Даний розділ присвячено практичним аспектам створення та інтеграції цього API з метою забезпечення найвищого рівня захисту інформації та конфіденційності у системі електронного документообігу.

Текстова частина викладена на 75 сторінках друкованого тексту. Список використаних джерел містить 34 найменувань і займає 3 сторінки. В роботі наведено 42 рисунки. Загальний обсяг роботи – 78 сторінок.

Ключові слова: API, HTTPS, JWT, SQL, SSL.

ANNOTATION

Explanatory Note for the Thesis "Methods and Technologies of Data Exchange Using Secure APIs".

This thesis explores the development and utilization of secure APIs. It consists of an introduction, three chapters, conclusions, and a list of references.

The first chapter investigates the use of APIs as a key tool in the contemporary information society. It emphasizes that Application Programming Interfaces (APIs) play a crucial role in web development and programming, defining them as a key component of the modern technological ecosystem. The chapter delves into the definition of APIs and outlines their main functions, serving as interfaces that enable different programs to interact with each other by exchanging data.

The second chapter analyzes various aspects of secure API development technologies. Specifically, it examines security protocols and standards employed to ensure the reliability and confidentiality of APIs. Key aspects of secure API development technologies are highlighted, empowering developers and architects to create reliable and secure application programming interfaces.

The third chapter focuses on the development and implementation of a secure API for the electronic document management subsystem. This section is dedicated to practical aspects of creating and integrating the API to ensure the highest level of information protection and confidentiality within the electronic document management system.

In conclusion, the thesis contributes to the understanding of secure API development and usage, emphasizing their importance in contemporary technological landscapes. The practical implementation of a secure API in the electronic document management subsystem illustrates the application of modern security standards and programming practices. The comprehensive investigation presented in this thesis aims to enhance the development and utilization of secure APIs in various technological contexts.

The text is set out on 75 pages of printed text. The list of used sources contains 34 names and occupies 3 pages. The paper presents 42 figures. The total volume of the work is 78 pages.

Keywords: API, HTTPS, JWT, SQL, SSL.

ЗМІСТ

ВСТУП.....	9
РОЗДІЛ 1 ТЕОРЕТИЧНИЙ АСПЕКТ ВИКОРИСТАННЯ API.....	11
1.1 Роль API у сучасному інформаційному суспільстві.....	11
1.1.1 Визначення API та його основні функції.....	13
1.1.2 Значення API у веб-розробці та програмуванні	16
1.2 Користь використання API	18
1.2.1 Забезпечення відкритості та доступності даних.....	20
1.2.2 Спрощення розробки програм та веб-додатків.....	22
1.3. Реальні застосування API у житті	25
1.3.1 Електронна комерція та універсальна інтеграція системи М.Е. Doc	25
1.3.2 Використання в розробці інтелектуальних інтерфейсів API GPT Chat	26
РОЗДІЛ 2 ТЕХНОЛОГІЇ РОЗРОБКИ БЕЗПЕЧНИХ API.....	29
2.1 Протоколи та стандарти безпеки для API.....	29
2.1.1 HTTPS як основний протокол безпеки.....	30
2.1.2 Механізми авторизації JSON Web Token, OAuth та інші	31
2.2 Архітектурні підходи до безпеки API.....	35
2.2.1 RESTful API та його безпекові принципи	37
2.2.2 GraphQL та безпека запитів у ньому	41
2.2.3 SOAP як альтернативний підхід та його безпекові аспекти	43
2.3 Інструменти для тестування та моніторингу безпеки API	48
2.3.1 Засоби перевірки вразливостей	48
2.3.2 Sql-інєкції в веб-застосунках.....	50
РОЗДІЛ 3 ПРАКТИЧНИЙ АСПЕКТ ЗАСТОСУВАННЯ БЕЗПЕЧНИХ API.....	53
3.1 Розробка та інтеграція безпечного API у підсистемі електронного документообігу	53
ВИСНОВКИ.....	71
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....	72
ДОДАТКИ.....	75

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface

PKI – Public Key Infrastructure

HTTPS – HyperText Transfer Protocol Secure

SFTP – SSH File Transfer Protocol

SQL – Structured Query Language

JSON – JavaScript Object Notation

XML – Extensible Markup Language

JWT – JSON Web Token

ASP – Active Server Pages

RSA – Криптографічний алгоритм

C# – Мова програмування

ВСТУП

Актуальність теми. Зростання кількості зловмисницьких атак та загрози конфіденційності даних підкреслюють важливість ефективних методів та технологій безпечного обміну інформацією через API. Дослідження цієї теми дозволить розкрити нові можливості та стратегії для забезпечення надійності та безпеки обміну даними в умовах постійно зростаючого цифрового виклику.

У сучасному інформаційному суспільстві, де обсяги даних стрімко зростають, важливою стає проблема безпечного обміну інформацією. Методи та технології обміну даними через безпечні API (інтерфейси програмування застосунків) набувають особливого значення у забезпеченні конфіденційності, цілісності та доступності інформації. Дипломна робота розглядає різноманітні аспекти цієї проблематики, включаючи аналіз сучасних методів шифрування, аутентифікації та контролю доступу в контексті API.

Актуальність обраної теми визначається стрімким розвитком інформаційних технологій та зростаючою кількістю та складністю обміну даними. У сучасному цифровому середовищі важливість безпечного обміну інформацією стає першочерговою завданням для багатьох галузей, включаючи бізнес, науку та громадські служби.

Високотехнологічні організації та проекти широко використовують API для взаємодії, що робить важливим розуміння та вдосконалення методів захисту цих інтерфейсів для запобігання можливим загрозам та вразливостям. Такий підхід вивчення безпечних API відображає сучасні вимоги до захисту інформації та забезпечує розробку вискоелективних стратегій безпеки в цифровому середовищі.

Метою кваліфікаційної роботи є систематизація та аналіз сучасних методів та технологій обміну даними через безпечні API з метою забезпечення конфіденційності, цілісності та доступності інформації.

Робота спрямована на вивчення різноманітних аспектів безпеки API, включаючи механізми шифрування, автентифікації та авторизації, а також аналіз протоколів і стандартів, таких як HTTPS, JWT, RESTful API. Поєднання академічного аналізу та

практичних досліджень дозволяє визначити оптимальні стратегії впровадження безпечних API в різноманітних сценаріях використання. Дипломна робота ставить за мету розкрити перспективи та виклики, пов'язані з сучасними методами обміну даними, та надати рекомендації для ефективного впровадження цих технологій в інформаційних системах.

Визначена мета обумовлює наступні **завдання**:

- 1) Систематизацію та аналіз сучасних методів та технологій;
- 2) Дослідження аспектів безпеки API;
- 3) Аналіз протоколів та стандартів;
- 4) Визначення оптимальних стратегій використання безпечних API в різних областях застосування;
- 5) Поєднання теоретичного аналізу та практичних досліджень.

Об'єктом дослідження є процес захищеного електронного документообігу за допомогою безпечного API.

Предметом дослідження є ефективність механізмів шифрування в обміні даними через безпечні API та методи автентифікації та авторизації в системах обміну даними

У процесі виконання кваліфікаційної роботи було досліджено методи та технології обміну даними та розроблено інтерфейс API-документообігу.

РОЗДІЛ 1 ТЕОРЕТИЧНИЙ АСПЕКТ ВИКОРИСТАННЯ API

1.1 Роль API у сучасному інформаційному суспільстві

В сучасному інформаційному суспільстві API (інтерфейси програмування додатків) відіграють важливу роль у забезпеченні ефективного обміну даними між різними системами та платформами. API є ключовим елементом в сучасній програмній інфраструктурі, що дозволяє різноманітним додаткам та сервісам взаємодіяти між собою. Розглядаючи різні сфери, такі як електронна комерція, медичні послуги, фінансові технології [15].

API визначаються як набір протоколів та інструментів, які дозволяють різним програмам обмінюватися даними та виконувати функції одна для одної. Це стає особливо актуальним в умовах загальної цифрової трансформації, коли компанії та організації шукають способи підвищення продуктивності та покращення зручності користування для своїх клієнтів [9].

Однією з ключових переваг використання безпечних API є створення єдиної точки доступу, що спрощує взаємодію між різними системами. Це сприяє швидкій інтеграції нових функцій, розширенню функціональності та покращенню взаємодії між додатками.

Важливим аспектом є також забезпечення безпеки даних під час обміну. Безпечні API використовують різноманітні механізми аутентифікації та шифрування для захисту конфіденційності та цілісності інформації. Це є критично в умовах постійного зростання кількості кіберзагроз, які можуть торкнутися різних аспектів сучасного інформаційного суспільства [22].

Збільшення використання API пояснюється можливістю технології "дозволяти розробникам програмного забезпечення працювати паралельно і незалежно". API стають все більш досконалими та розповсюдженими завдяки введенню їх у нову екосистему інструментів і постачальників послуг, що дозволяє інноваційному бізнесу отримати модель доходу. API є важливим веб-сервісом, оскільки контент генерується, маніпулюється та споживається користувачем у різний спосіб [26].

Медіа-групи використовують внутрішні технології API у своєму процесі "для реструктуризації та організації внутрішньої системи для підтримки інноваційних нових проєктів уніфікованим способом - зменшення витрат на обслуговування та підвищення гнучкості". Кінцеві користувачі новинних сервісів не помічають внутрішні API, які працюють для доставки контенту, адаптованого до їхніх потреб, але деякі з них використовують ці технології у режимі реального часу. На сьогодні API стали фундаментальною технологією для тих, хто прагне надати нові можливості для створення контенту, знаходження нових способів залучення клієнтів, отримання прибутку та укладання партнерських відносин [15].

Проте, коли новинні організації працюють зовнішніми API, вони вибирають різні стратегії, оскільки знаходяться в етапі експериментів, спробуючи об'єднати бізнес-моделі в цій області. Навіть якщо деякі з них конкурують за один і той же сегмент в аналоговому медіа-бізнесі, в цифровому середовищі ці медіа-компанії вибрали різні стратегії конкуренції у сфері цифрових даних для постачання аналітики [9].

API грають важливу роль у забезпеченні споживачам доступу до інформації та сервісів через мобільні додатки та інші пристрої. Це забезпечує зручність і доступність для кінцевих користувачів. Вони дозволяють створювати ефективні механізми автентифікації та авторизації, що є ключовими для забезпечення безпеки обміну даними. Крім того, вони дозволяють ефективно керувати доступом та правами. Також API дозволяють розробникам швидко використовувати функціональність інших систем без необхідності переписування коду. Це сприяє швидкому розгортанню нових функцій та продуктів, що є ключовим для інновацій [22].

Загалом, API визначають сучасну інформаційну архітектуру та грають ключову роль у забезпеченні ефективного функціонування та розвитку сучасного суспільства.

1.1.1 Визначення API та його основні функції

API (інтерфейс програмування додатків) представляє собою набір протоколів та інструментів, які дозволяють програмам взаємодіяти між собою. Це важливий компонент сучасних інформаційних систем, що забезпечує стандартизований інтерфейс для обміну даними та викликів функцій між різними програмами. Основні функції API включають [6]:

Стандартизація комунікації. Application programming interface надає стандартизований спосіб взаємодії між різними системами. API є невід'ємною частиною розробки програмного забезпечення, оскільки вона дозволяє створювати єдиний інтерфейс для взаємодії між різними додатками. Визначені стандарти дозволяють розробникам розуміти, яким чином взаємодіяти з певними сервісами чи додатками, що полегшує процес розробки та підтримки.

Стандартизація програмного інтерфейсу визначає загальні правила та формати, які додатки повинні дотримуватися при обміні даними. Це робить взаємодію більш передбачуваною та легшою для розробників, які можуть швидше розуміти та інтегрувати функціональність.

За допомогою стандартизованого API, різні додатки можуть використовувати різні мови програмування та операційні системи. Це робить API універсальним інструментом, який може об'єднувати різноманітні технології в єдиний інтерфейс.

Стандартизація дозволяє зменшити залежність між різними системами. Розробники можуть змінювати або оновлювати свої додатки, не змінюючи API, яке вони використовують. Це забезпечує стійкість та довговічність інтерфейсів.

Завдяки стандартизованому інтерфейсу, інтеграція нових додатків чи сервісів стає більш простою. Розробники можуть швидше розуміти документацію та взаємодіяти з іншими системами без додаткового навчання чи глибокого розуміння їхньої внутрішньої структури.

Стандартизовані API можуть легко розширюватися без необхідності внесення змін у внутрішню логіку самого додатка. Це робить можливим додавання нових функцій та можливостей без переробки всієї системи. Отже, комунікації

через API є визначальною чергою для побудови ефективних та стійких інформаційних систем, які можуть легко взаємодіяти між собою в сучасному інформаційному суспільстві.

Можливість використання функцій інших програм. API надає можливість викликати функції або отримувати дані від інших додатків чи сервісів. Це дозволяє розширювати функціональність свого додатка без необхідності повторного написання коду.

Інтерфейс програмування дозволяє використовувати функції, які вже реалізовані в інших програмах чи сервісах, без необхідності повторного їх написання. Це полегшує розробникам задачу, оскільки вони можуть ефективно використовувати готові рішення та швидше розробляти нові додатки. Розробники можуть комбінувати функціональність різних сервісів та програм для створення комплексних та повнофункціональних додатків. Це дозволяє швидше реагувати на змінні вимоги та побудовувати системи, які враховують широкий спектр функцій.

API сприяє розвитку екосистеми додатків, де різні розробники можуть створювати різні компоненти, які працюють разом. Це сприяє конкуренції, інноваціям та розширює функціональність загальної системи [26].

Розробники можуть концентруватися на специфічних аспектах своєї роботи, використовуючи готові API для інших завдань. Взаємодія та спільне використання функціональності через API створює динамічне та інноваційне середовище для розробників, що прискорює розробку нових технологій та покращення існуючих рішень [9].

Обмін даними. Однією з ключових функцій API є обмін даними між різними системами. Це може включати передачу структурованих даних, які використовуються для взаємодії між різними аплікаціями чи платформами.

Application programming interface дозволяє передавати дані в структурованому форматі, такому як JSON чи XML, що полегшує розпізнавання та опрацювання інформації. Це важливо для розробників, оскільки вони можуть легко інтегрувати дані з інших джерел та використовувати їх для власних потреб.

Сервіс API може включати механізми синхронізації для забезпечення узгодженості даних між різними системами. Це гарантує, що інформація залишається актуальною та точною, навіть коли вона змінюється в одному додатку чи сервісі.

Інтерфейс для взаємодії може забезпечити миттєвий обмін даними у реальному часі або асинхронний обмін, що дозволяє розробникам створювати більш гнучкі та відзивчиві додатки. Це особливо важливо в сучасних системах, де потрібна оперативна реакція на зміни у даних [6].

Автентифікація та безпека. Інтерфейс програмування надає механізми для перевірки автентичності користувача чи додатка, який робить запит до системи. Це може бути реалізовано через використання ключів API, токенів, аутентифікації на основі логіну та пароля чи інших методів, залежно від вимог системи.

API має вбудовані механізми захисту від різних видів атак, таких як атаки на отримання несанкціонованого доступу (наприклад, SQL-ін'єкції, атаки на переповнення буфера), для забезпечення стійкості до потенційних загроз.

Також сервіс API може включати систему управління правами доступу, що визначає, які користувачі чи додатки мають доступ до конкретних ресурсів. Це забезпечує прозорість та контроль над тим, хто може отримувати чи змінювати дані.

Модульність та розширюваність. Використання API дозволяє розробникам створювати додатки, які можуть бути легко розширені та модифіковані, використовуючи функціональність, яку надають інші програми чи сервіси.

API дозволяє розробникам використовувати лише ті частини функціоналу, які їм потрібні, що сприяє створенню компактних та ефективних додатків. Модульний підхід дозволяє розділити систему на невеликі компоненти, що полегшує розуміння та управління кодом.

Розробники можуть використовувати внутрішні модулі свого додатка чи використовувати зовнішні модулі через API. Це сприяє швидкому розширенню функціональності та використанню готових рішень без необхідності в повторному написанні коду [26].

Application programming interface дозволяє розширювати функціональність додатків без необхідності внесення змін у вихідний код. Розробники можуть додавати нові можливості, використовуючи існуючий інтерфейс та взаємодіючи з іншими додатками чи сервісами через API.

Варто також додати, що API може включати визначені стандарти для розширення, що дозволяє розробникам створювати розширення відповідно до зазначених правил та специфікацій. Це сприяє стандартизації та підтримці різноманітності додатків.

1.1.2 Значення API у веб-розробці та програмуванні

API, або інтерфейси прикладного програмування, слугують месенджером, передаючи запити та відповіді між різними програмними системами. Вони виконують невід'ємну роль у сфері веб-розробки для онлайн-бізнесу, забезпечуючи засоби спілкування та інтеграції між програмами [8].

Ця форма інтеграції набуває вирішального значення в сучасному цифровому ландшафті, де додатки регулярно вимагають взаємодії з іншими сервісами та базами даних для ефективного функціонування. API надає спільну мову для цієї взаємодії, спрощуючи процес інтеграції та дозволяючи розробникам створювати більш надійні та універсальні веб-додатки.

Інша значуща функція API у сфері веб-розробки полягає в удосконаленні користувацького досвіду. Інтерфейси сервісів спрощують отримання даних із зовнішніх ресурсів, таких як платформи соціальних мереж чи сервіси прогнозування погоди, і інтегрують ці дані у веб-додаток у зручному та привабливому для користувачів форматі [21].

Ця можливість збирати дані з різних джерел та безперешкодно їх інтегрувати у веб-сайт чи додаток може суттєво підвищити загальний користувацький досвід, що робить API ключовим інструментом в арсеналі будь-якого веб-розробника.

Сервіс API також сприяють масштабованості веб-додатків. За допомогою API розробники можуть додавати чи змінювати функціонал, не порушуючи роботу існуючої системи, спрощуючи масштабування програми за потреби.

Крім того, сервіси для взаємодії програм дозволяють ефективно використовувати ресурси, оскільки вони дозволяють розробникам використовувати існуючі функціональні можливості, замість будівництва всього з нуля. Це не лише заощаджує час, а й сприяє зручності обслуговування та масштабованості додатку [21].

У сфері веб-розробки API відіграють важливу роль у автоматизації. Це дозволяє програмним системам керувати завданнями та процесами без участі людини, що підвищує ефективність та точність. Незалежно від того, чи це автоматизація введення даних, запуск подій на основі умов або полегшення пакетних операцій, API дозволяють автоматизувати різноманітні функції, підвищуючи продуктивність та знижуючи ймовірність помилок.

Безпека визначається як найважливіше завдання у сфері веб-розробки, і API відіграють ключову роль у гарантуванні цього аспекту. Програмні інтерфейси встановлюють рівень абстракції між додатком та основними системами, сприяючи захисту конфіденційних даних та запобігаючи несанкціонованому доступу. Крім того, API часто вбудовують протоколи та стандарти безпеки, що додатково підвищує безпеку веб-додатків [29].

У кінцевому підсумку, API допомагають стимулювати інновації у галузі веб-розробки. Розкриваючи функціональні можливості та дані, API вдихають розробникам створювати новаторські додатки та сервіси, які раніше були недосяжними. Це сприяє розвитку культури інновацій та творчості, що призводить до з'яви нових рішень та досягнень у галузі веб-розробки.

Використання API в сфері веб-розробки надає ряд значущих переваг, що спрощують процес створення додатків і підвищують їх функціональність. Розробники можуть здобути доступ до готових функцій та сервісів, які надаються зовнішніми платформами, що значно прискорює розробку, оскільки необхідно вдаватися в складний процес написання коду для вирішення завдань "з нуля" [8].

Використання інтерфесу програмування дозволяє розробникам заощадити час та ресурси, які інакше б витратилися на розробку та підтримку власних

інтеграцій для користувачів. Це призводить до зменшення витрат на розробку та в кінцевому підсумку приносить фінансові вигоди [29].

Сервіси API дозволяють створювати веб-додатки, які легко масштабувати. Завдяки використанню готової функціональності та послуг від третіх сторін, розробники можуть фокусуватися на створенні основних функцій, що гарантує, що їх додатки можуть безперешкодно розвиватися при збільшенні потреб користувачів [8].

Application programming interface можуть значно підвищити продуктивність веб-додатків, вивантажуючи ресурсомісткі завдання на зовнішні сервіси. Це сприяє зменшенню навантаження на сервер і поліпшенню взаємодії з користувачем.

Підсумовуючи, можна сказати, що API є важливим інструментом у веб-розробці та програмуванні, який дозволяє розробникам швидко і легко інтегрувати існуючі функціональні можливості у свої додатки. Крім того, вони сприяють автоматизації, підвищують безпеку та стимулюють інновації.

1.2 Користь використання API

Використання API (інтерфейсів прикладного програмування) у веб-розробці є вирішальним елементом, що впливає на ефективність, швидкість та гнучкість розробки.

Використання API дозволяє розробникам швидко і ефективно вбудовувати готову функціональність, надану зовнішніми сервісами, у свої власні додатки. Наприклад, використання готового API для оплати дозволяє уникнути витрат часу на написання складного коду для обробки транзакцій. Це прискорює процес розробки та дозволяє розробникам фокусуватися на унікальних аспектах їхнього проекту [1].

Використання API дозволяє автоматизувати багато рутинних завдань, таких як обробка даних, аналіз інформації, або взаємодія з іншими системами. Замість того, щоб витрачати час на монотонні операції, розробники можуть використовувати API для автоматизації цих завдань, підвищуючи ефективність роботи.

API надають можливість легко інтегрувати власний код чи додаток із зовнішніми сервісами та платформами. Це створює можливість швидко впроваджувати нові функції та покращення, використовуючи готові рішення замість розробки їх з нуля.

Використання API дозволяє розробникам вибирати найбільш підходящі технології для конкретних завдань. Вони можуть легко змінювати чи оновлювати компоненти системи, не руйнуючи всій код, завдяки гнучкій взаємодії за допомогою API [18].

Використання готових API дозволяє розробникам заощаджувати ресурси, які б інакше витрачалися на створення та підтримку власних інтеграцій для користувачів. Використання стійких та вже перевірених API може допомогти зменшити ризик фінансових втрат, пов'язаних з відмовами чи проблемами в інтеграціях. Замість того, щоб розробляти власні інтеграції, які можуть бути піддані помилкам, розробники можуть користуватися надійними зовнішніми сервісами.

Застосування готових API дозволяє командам розробників оптимізувати використання людських ресурсів. Замість того, щоб витрачати час і сили на розробку внутрішніх рішень, розробники можуть швидше впроваджувати функції, використовуючи зовнішні API, тим самим економлячи ресурси [19].

API роблять додатки більш масштабованими, оскільки розробники можуть легко додавати чи змінювати функціональність. Використання існуючих функцій та сервісів, наданих сторонніми API, дозволяє концентруватися на створенні ключових функцій, забезпечуючи, що програма легко масштабується за потреби.

API забезпечують можливість швидкої інтеграції нових сервісів для розширення функціональності. Розробники можуть додавати нові API для обробки конкретних завдань чи використовувати існуючі, не перероблюючи весь додаток. Це полегшує розширення системи та адаптацію до нових вимог [13].

Модульність API допомагає зменшити складність обслуговування та оновлення додатків. Оскільки кожен функціональний блок може бути реалізований

через API, зміни в одному модулі не вимагають переробки всього додатку. Це полегшує підтримку, розширення та вдосконалення окремих частин системи.

Також модульність програмного інтерфейсу забезпечує високу доступність та надійність системи. Кожен модуль може бути розгляданий як окремий елемент, що робить систему менш вразливою до відмов чи помилок в окремих компонентах. Це сприяє покращенню загальної надійності та продуктивності. Додавання нових модулів стає простішим завдяки готовим API, що вже існують у системі. Це дозволяє системі легко адаптуватися до зростаючих потреб користувачів та швидко впроваджувати інновації [34].

Усі ці аспекти детально висвітлюють позитивний вплив використання API на процес веб-розробки, включаючи прискорення розробки, економію ресурсів, можливість масштабування та забезпечення модульної архітектури додатків.

1.2.1 Забезпечення відкритості та доступності даних

В сучасному інформаційному суспільстві, де обмін даними відіграє ключову роль, забезпечення відкритості та доступності даних стає важливим завданням. Використання API в цьому контексті є ефективним засобом забезпечення надійного обміну інформацією між різними програмними системами. Розглянемо детальніше, як API сприяє відкритості та доступності даних.

API визначають стандартизовані формати даних для передачі інформації між різними системами. Такі формати, як JSON (JavaScript Object Notation) та XML (eXtensible Markup Language), використовуються для представлення структурованої інформації. Це забезпечує єдність у способі подання даних, незалежно від того, які мови програмування використовуються у взаємодії систем [24].

JSON став важливим стандартом для обміну даними через API. Цей легкий та зрозумілий формат полегшує розробникам роботу з даними. Його простота та зручність забезпечують зрозумілість та легкість розбору даних як на стороні сервера, так і на стороні клієнта [16].

XML, хоч і менш популярний у порівнянні з JSON, використовується в ряді систем для обміну структурованою інформацією. Він відрізняється високою розширюваністю, що робить його ефективним для великих проєктів, де важлива гнучкість у структурі даних.

Стандартизація формату даних забезпечує їх єдність та інтероперабельність між різними системами. Різні додатки та сервіси можуть легко обмінюватися інформацією, оскільки всі залучені сторони розуміють і дотримуються одного формату, визначеного через API [12].

Інтерфейси сервісів забезпечують широкий доступ до різноманітних даних та функціональностей, надаючи можливість різним користувачам отримувати інформацію відповідно до їх потреб. Від розробників та компаній до звичайних користувачів, API роблять інформацію доступною для широкого спектру аудиторій.

Програмні інтерфейси надають засіб для створення нових додатків та сервісів, що розширюють можливості користувачів. Розробники можуть використовувати доступні API для створення інноваційних застосунків, які полегшують життя, розважають або вирішують конкретні завдання. Забезпечення доступу до даних через API сприяє розвитку інформаційної культури серед користувачів. Вони можуть взаємодіяти з різними джерелами даних, розуміти їхню структуру та використовувати для вирішення конкретних завдань, що піднімає рівень компетентності користувачів у сфері обробки інформації [2].

Також забезпечення доступу до даних через API дозволяє створювати рішення, які відповідають конкретним потребам користувачів. Це може бути важливим для розробки персоналізованих додатків, які забезпечують ефективний та зручний досвід використання.

API включають механізми аутентифікації, що гарантують, що лише авторизовані користувачі мають доступ до визначених ресурсів чи функціональностей. Аутентифікація дозволяє перевірити ідентичність користувача та переконатися, що він має право на взаємодію з системою через API [27].

Засоби для обміну даними використовують різноманітні механізми авторизації для керування тим, наскільки широкий доступ має авторизований користувач. Токени, ключі та інші механізми дозволяють точно визначити, які операції чи ресурси може виконувати користувач під час взаємодії з API.

Аутентифікація через API важлива для забезпечення конфіденційності та безпеки даних. Захищений доступ зменшує ризик несанкціонованого доступу та неправомірного використання інформації. Механізми безпеки, такі як HTTPS, допомагають забезпечити шифрування даних під час передачі через мережу [12].

Авторизація через програмний інтерфейс дозволяє детально управляти дозволами користувачів. За допомогою механізмів авторизації можна точно налаштовувати, які операції чи ресурси доступні конкретним користувачам чи додаткам, що полегшує контроль над рівнем доступу.

Використання API в інформаційному суспільстві визначається не лише можливістю обміну даними, але і важливими аспектами безпеки та контролю над доступом. Аутентифікація гарантує перевірку ідентичності користувача, а механізми авторизації дозволяють точно визначити обсяг його доступу до ресурсів чи функціональностей. Ці механізми грають критичну роль у забезпеченні конфіденційності та безпеки даних, а також у підтримці дозволів для ефективного управління доступом.

1.2.2 Спрощення розробки програм та веб-додатків

Використання API в програмній розробці значно спрощує інтеграцію функціональності різних додатків та сервісів. Розробники можуть використовувати стандартні API для отримання доступу до визначених операцій чи функцій, що робить процес розробки більш ефективним та менш витратним.

API забезпечують стандартизовані інтерфейси, що дозволяють розробникам взаємодіяти з функціональністю інших систем чи сервісів. Це робить інтеграцію прозорою та спрощує взаємодію між різними компонентами програмного забезпечення [20].

Варто зауважити, що інтерфейси програмування надають спільну мову для взаємодії між різними додатками та сервісами. Це означає, що розробники можуть спілкуватися з іншими системами, використовуючи стандартні принципи та формати обміну даними, що робить інтеграцію більш зрозумілою та простою.

Розробники можуть використовувати функціональність, яка вже існує у вигляді сторонніх сервісів чи додатків, за допомогою сервісів API. Це дозволяє уникнути дублювання зусиль та використовувати готові рішення для полегшення розробки [25].

Інтерфейси для взаємодії програм дозволяють розробникам розширювати можливості своїх додатків за рахунок інтеграції з іншими сервісами. Наприклад, веб-сайт електронного магазину може легко інтегруватися з платіжним сервісом, щоб забезпечити зручну оплату для користувачів.

API дозволяють отримувати актуальні дані в режимі реального часу. Це важливо для додатків, які потребують постійного оновлення інформації, таких як новинні агрегатори або системи моніторингу. API дозволяють розробникам використовувати готові рішення та функціональність, що значно зменшує обсяг написаного коду. Це призводить до економії ресурсів та часу, оскільки розробники можуть уникнути повторного написання складного коду, використовуючи готові API [25].

Також API дозволяють ефективно обробляти та обмінюватися даними між різними додатками. Замість ручного написання коду для обробки та передачі даних, розробники можуть використовувати готові API, що спрощує взаємодію з іншими системами.

Використання інтерфейсів взаємодії програм дозволяє розробникам інтегрувати сторонні сервіси, замість того щоб самотійно розробляти та підтримувати власний функціонал. Це дозволяє зосередитися на основних аспектах додатка та уникнути дублювання зусиль.

Використання готових API дозволяє розробникам використовувати стандартні функції та методи, уникнувши потреби у великих обсягах власного

коду. Це сприяє зменшенню кількості написаного коду та забезпечує його більшу читабельність [20].

API дозволяють додаткам легко інтегруватися з різноманітними сторонніми сервісами. Наприклад, соціальні мережі, платіжні системи, картографічні сервіси та інші можуть бути доступні для додатків через відповідні API, розширюючи функціональність та корисність додатка.

Завдяки API, розробники можуть мати доступ до розширених функцій, які можуть бути недоступні в базовому функціоналі додатка. Це дозволяє підвищити рівень служб та функціональність додатка без необхідності повного переписування коду.

Сервіси для взаємодії програм надають доступ до спеціалізованих сервісів, що дозволяє додаткам використовувати вже існуючі рішення для вирішення конкретних завдань. Наприклад, сервіси перекладу, розпізнавання обличчя, аналізу тексту та інші можуть бути використані через API [25].

Завдяки розширенню можливостей додатків через API, користувачі отримують покращений та різноманітний досвід використання. Додатки можуть взаємодіяти з різними сервісами, що дозволяє створювати зручні та інноваційні рішення.

Розробники можуть створювати унікальні функціональні рішення, комбінуючи функціональність свого додатка з послугами сторонніх сервісів через API. Це відкриває двері для інновацій та творчих підходів в розробці програмного забезпечення.

Ключовими аспектами, які стають доступними завдяки використанню API у процесі розробки програм та веб-додатків є забезпечення сумісності та швидкість розробки. API сприяють уніфікації форматів даних та протоколів, що дозволяє створювати сумісні між собою додатки. Розробники можуть використовувати стандартні API для обміну даними без необхідності власноручно налаштовувати інтеграції, що полегшує роботу та зменшує ризик помилок.

Використання API дозволяє розробникам використовувати готові рішення та функціональність сторонніх сервісів. Це значно прискорює розробку, оскільки

розробники можуть скористатися готовими модулями, які вже пройшли тестування та використовуються в інших проектах.

API дозволяють автоматизувати рутинні та повторювані завдання у розробці. Наприклад, автоматичне розгортання середовища, тестування чи пакетна обробка даних можуть бути здійснені за допомогою API, звільняючи розробників від монотонної роботи та прискорюючи весь процес розробки.

За допомогою API розробники можуть використовувати готові бібліотеки та фреймворки, які прискорюють розробку шляхом використання готових модулів та робочих рішень. Це полегшує завдання та забезпечує високий рівень швидкості розробки [20].

Спільна сумісність та готові рішення через API дозволяють зберігати додаток актуальним та готовим до використання. При внесенні змін у сторонніх сервісах, розробники можуть швидко адаптувати свій додаток до нових умов без значного переписування коду.

Використання API в розробці програм та веб-додатків забезпечує високий рівень сумісності та швидкості у розробці. Уніфікація даних, готові рішення, автоматизація та гнучкість API дозволяють розробникам швидко створювати та підтримувати високоякісні програмні продукти. Готові бібліотеки та постійні оновлення через API забезпечують ефективну та актуальну розробку.

1.3. Реальні застосування API у житті

1.3.1 Електронна комерція та універсальна інтеграція системи М.Е. Дос

Програма М.Е. Дос визначається як найкращий універсальний помічник бухгалтера для обміну документами між будь-якою обліковою системою і програмою М.Е.Дос.. Ця програма виявляється особливо корисною для управління підприємством та бухгалтерії. Однак для забезпечення максимального комфорту для користувачів та впровадження додаткових функцій, розробники М.Е. Дос впроваджують концепцію універсальної інтеграції та надають доступ до API.

API в М.Е. Дос відкриває можливості для розробників і інтеграторів створювати зв'язки з програмою і розширювати її функціональність. За допомогою

API користувачі можуть отримувати та надсилати первинні документи (податкова накладна, видаткова накладна, рахунок, акт, ТТН) та регламентовану звітність (Декларація ПДВ, звіт ЄСВ, форми Балансу та інші).

Універсальна інтеграція та API програми М.Е. Дос грають ключову роль у покращенні бізнес-процесів. Для підприємств, які використовують М.Е. Дос, це відкриває нові можливості для автоматизації обліку, інтеграції з електронними системами тощо.

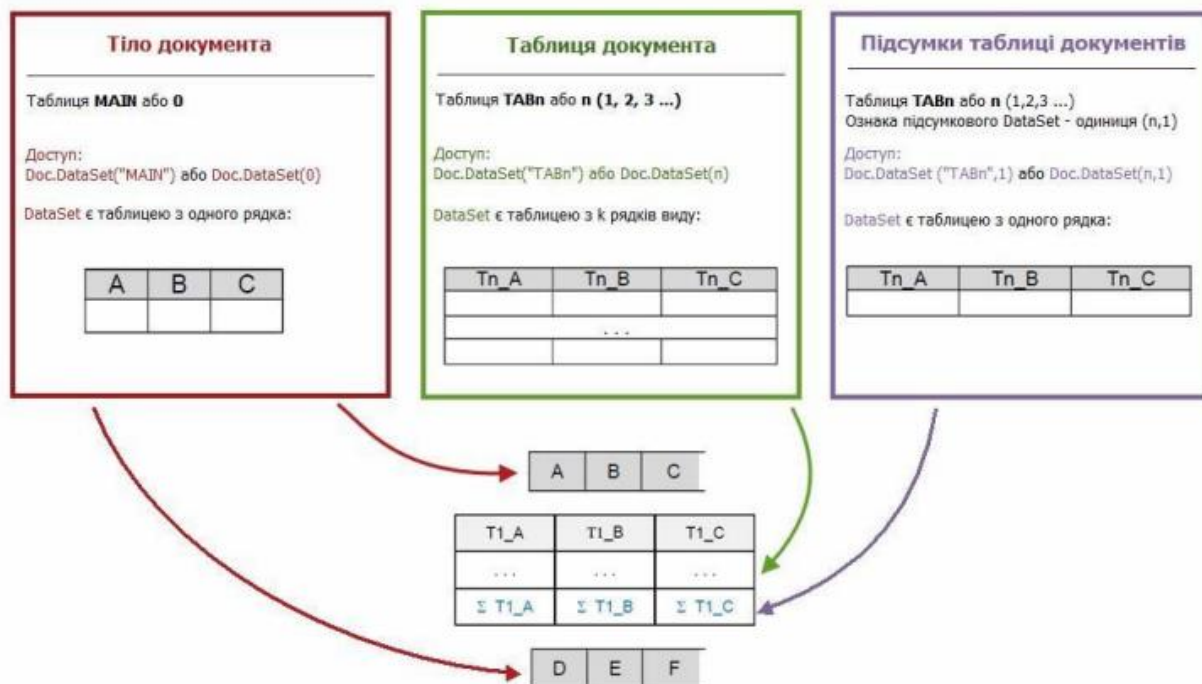


Рисунок 1.1 – Структура документа в API М.Е. Дос

Інтеграція працює на основі COM-об'єктів. Документ складається із: тіла документа (MAIN,0), підтаблиць (TABn,0) та підсумкових рядків для підтаблиць (TABn,1). Будь-які дані (тіло документа, підтаблиці та підсумки підтаблиць, вибірки, таблиці БД тощо) представлені у системі об'єктом MEDOC.IZDataSet. Весь набір операцій цього об'єкту може бути застосованим до будь-якої таблиці.

1.3.2 Використання в розробці інтелектуальних інтерфейсів API GPT Chat

API GPT Chat - це інтерфейс програмного забезпечення, який надає доступ до мовленнєво-генеративної моделі GPT для взаємодії з користувачем через текстовий ввід та вивід. Розроблений компанією OpenAI, GPT Chat API базується

на архітектурі GPT-3.5 та використовує передові техніки машинного навчання для створення природної та контекстно-залежної генерації тексту.

Основна мета API GPT Chat полягає у забезпеченні розширених можливостей взаємодії з мовленнєвими інтерфейсами шляхом використання передових алгоритмів обробки природної мови. Модель GPT використовує глибоке навчання для "передбачення" наступного слова або послідовності слів в контексті заданого тексту. Це дозволяє їй генерувати відповіді, які виглядають природним чином та враховують контекст обговорення.

Однією з ключових особливостей API GPT Chat є здатність розпізнавання та врахування широкого спектру запитань та вказівок, дозволяючи користувачам створювати різноманітні застосунки. Це може включати створення текстових чат-ботів, генерацію творчого контенту, вирішення завдань обробки мовлення та багато іншого.

Однією з важливих переваг використання GPT Chat API є можливість швидко та ефективно інтегрувати мовленнєві можливості в різноманітні програмні рішення без необхідності глибокого розуміння внутрішньої роботи моделі. Це робить його досить доступним для розробників та компаній, які хочуть покращити взаємодію своїх продуктів з користувачами.

Загалом, API GPT Chat відкриває нові перспективи для створення інтерактивних та інтелектуальних інтерфейсів, що базуються на природній мові, віддзеркалюючи поступовий розвиток технологій машинного навчання та обробки природної мови.

API GPT Chat використовує передові технології для безпечного обміну запитаними. Для забезпечення конфіденційності та інтегрованості з іншими системами використовується протокол HTTPS, що гарантує шифрування даних під час їхньої передачі між клієнтом та сервером.

Структура запитів в API GPT Chat зазвичай включає у себе JSON-об'єкти, які містять необхідні параметри. Важливим елементом є поле, яке містить текстовий ввід від користувача або інші вхідні дані, з якими модель має взаємодіяти. Ця структура дозволяє ефективно обмінюватися інформацією між клієнтом і сервером.

Щодо технології самого API, він базується на глибокому машинному навчанні, зокрема на архітектурі GPT. Модель GPT-3.5, яка використовується в API GPT Chat, є великим нейромережевим агентом, який був переднавчений на обширних обсягах текстових даних. Цей підхід дозволяє моделі ефективно розуміти контекст та генерувати природно мовлення відповідно до вхідних запитань або вказівок.

Усі ці технології спільно дозволяють API GPT Chat забезпечувати високий рівень безпеки та продуктивності у взаємодії з користувачами та іншими системами через текстовий інтерфейс.

Висновок до розділу 1

Отже, у першому розділі дипломної роботи, присвяченому теоретичному аспекту використання API, було ретельно розглянуто роль API у сучасному інформаційному суспільстві. Аналіз визначення API та його основних функцій дозволив глибше зрозуміти його значення у веб-розробці та програмуванні. Висвітлено користь використання API, фокусуючись на забезпеченні відкритості та доступності даних, а також на спрощенні розробки програм та веб-додатків.

РОЗДІЛ 2 ТЕХНОЛОГІЇ РОЗРОБКИ БЕЗПЕЧНИХ API

2.1 Протоколи та стандарти безпеки для API

Безпека API визначається як ключовий аспект у забезпеченні безпечного доступу до цінних ресурсів. Здійснення контролю доступу набуває критичного значення для гарантування того, що лише користувачам або системам, які мають відповідний дозвіл, надається доступ [23].

Контроль доступу. Ефективний контроль доступу особливо важливий при наданні доступу третім особам. Поза використанням стандартних методів автентифікації, таких як OAuth або багатофакторна автентифікація, важливо впроваджувати брандмауери та шлюзи API. Брандмауери використовуються для блокування несанкціонованого доступу, а шлюзи API діють як зворотний проксі, реалізуючи автентифікацію та авторизацію.

Класифікація даних. Організації повинні класифікувати дані відповідно до конфіденційності для застосування відповідних заходів безпеки. Наприклад, API, яке має доступ до конфіденційних даних клієнтів, повинно застосовувати заходи безпеки для обмеження доступу лише для авторизованих осіб.

Перевірка параметрів. Проведення перевірки параметрів гарантує правильний тип і відповідність вхідних параметрів критеріям безпеки. Цей процес допомагає уникнути обробки недійсних або зловмисних даних, забезпечуючи валідність та очікуваність вхідних даних [11].

Шифрування запитів та відповідей API. Шифрування є ефективним засобом захисту конфіденційності та цілісності даних, передаваних через API. Застосування HTTPS і SSL/TLS гарантує шифрування трафіку API, запобігаючи атакам типу "людина посередині".

Постійна безпека. Безперервна безпека вимагає систематичного моніторингу та вдосконалення заходів безпеки API протягом всього його життєвого циклу. Цей підхід дозволяє швидко виявляти та виправляти потенційні вразливості або ризики. Організації можуть досягати постійної безпеки за допомогою постійного тестування, моніторингу та навчання персоналу [28].

Завдяки цим новаторським підходам до безпеки API організації можуть ефективно захищати свої ресурси та забезпечувати безпечний обмін даними з користувачами та партнерами.

2.1.1 HTTPS як основний протокол безпеки

HTTPS (Hypertext Transfer Protocol Secure) є розширенням звичайного HTTP, яке забезпечує безпечний обмін даними між користувачем і веб-сайтом. Цей протокол став необхідним у сучасному цифровому світі, де збільшується кількість онлайн-транзакцій та обміну конфіденційною інформацією [16].

HTTPS використовує шифрування для захисту інформації від несанкціонованого доступу. Ключовим елементом є використання SSL/TLS (Secure Sockets Layer/Transport Layer Security) протоколу, який забезпечує конфіденційність та цілісність даних під час їх передачі між користувачем і сервером [30].

Шифрування даних у HTTPS використовується для того, щоб забезпечити приватність і конфіденційність інформації, яку передають користувачі через Інтернет. Основна мета полягає в тому, щоб уникнути можливості перехоплення та читання чутливих даних третіми сторонами, що може призвести до серйозних наслідків, таких як крадіжка особистої інформації або фінансових операцій.

SSL/TLS використовується для створення зашифрованого тунелю між клієнтом і сервером. Під час процесу "розгортання рукля" (handshake), який відбувається при початковому з'єднанні, сервер представляє свій цифровий сертифікат, підтверджуючи свою автентичність. Клієнт, у свою чергу, перевіряє цей сертифікат і встановлює зашифроване з'єднання [4].

За допомогою шифрування SSL/TLS, дані стають нерозбірливими для будь-якого, хто намагається перехопити їх по дорозі. Це забезпечує конфіденційність, оскільки навіть якщо зловмисник отримає доступ до перехопленого трафіку, він не зможе розшифрувати інформацію без відповідного ключа.

Крім того, SSL/TLS гарантує цілісність даних, тобто воно запобігає будь-яким змінам даних під час їх передачі. Зловмисники не можуть вносити зміни в

інформацію, яку користувач відправляє серверу, або навпаки. Це важливо для забезпечення достовірності даних та недопущення маніпуляцій [30].

У цілому, використання HTTPS і шифрування SSL/TLS є фундаментальною складовою безпеки в Інтернеті, що забезпечує високий рівень захисту для користувачів і їхніх конфіденційних даних.

HTTPS гарантує захищеність фінансових транзакцій під час онлайн-покупок. Під час проведення транзакції, такої як покупка товарів чи послуг, чутлива інформація, така як номери кредитних карт, персональні дані та інші фінансові деталі, залишаються конфіденційними завдяки шифруванню SSL/TLS [12].

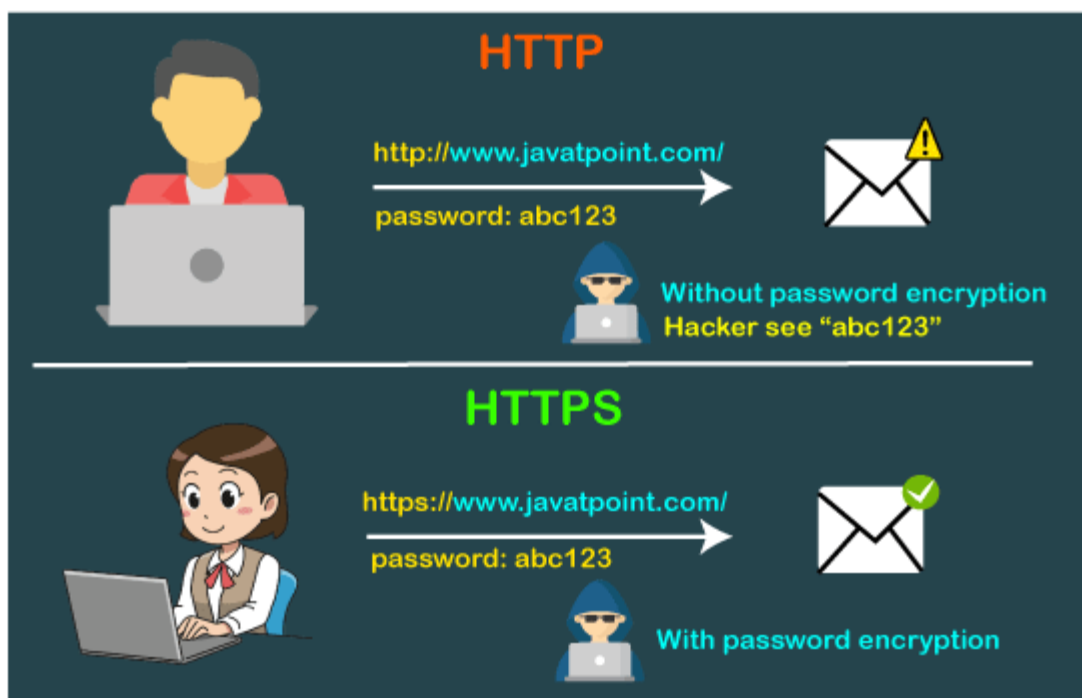


Рисунок 2.2 – Логіка роботи протоколу HTTPS

Для встановлення HTTPS необхідно отримати сертифікат безпеки від авторитетного постачальника. Цей сертифікат підтверджує, що веб-сайт є надійним, що забезпечує довіру користувачів та запобігає можливим атакам типу "man-in-the-middle".

2.1.2 Механізми авторизації JSON Web Token, OAuth та інші

Сучасний інтернет вимагає ефективних механізмів авторизації для забезпечення безпеки і конфіденційності обміну даними між користувачами та

сервісами. JSON Web Token (JWT), OAuth та інші механізми є ключовими елементами цього процесу, забезпечуючи безпеку та зручність взаємодії.

JSON Web Token (JWT) представляє собою стандарт веб-безпеки, який визначає компактний та самостійний спосіб передачі інформації між сторонами у вигляді об'єкта JSON. Використовуючи цей стандарт, можна створювати токени, які містять у собі дані, які можуть бути перевірені та декодовані. Це робить JWT надійним інструментом для автентифікації та авторизації в розподілених системах.

JWT базується на трьох частинах: заголовку (Header), вказівниках (Payload) та підпису (Signature). У заголовку міститься інформація про тип токена та алгоритм шифрування. Вказівники містять корисну інформацію, яку слід передавати, і підпис забезпечує перевірку цілісності та автентичності токена [16].

Під час генерації JWT, сервер створює заголовок та вказівники, які потім об'єднуються в один токен. Підпис обчислюється за допомогою секретного ключа сервера. При отриманні токена стороною, отримуюча сторона може перевірити його цілісність та автентичність, використовуючи публічний ключ сервера.

JWT використовується в різних контекстах, зокрема у веб-розробці та мікросервісних архітектурах. Одним із основних методів використання є автентифікація. Після того, як користувач успішно пройшов процедуру автентифікації, йому виділяється JWT, який він може включати в заголовок кожного запиту до сервера для авторизованого доступу.

JWT також використовується для обміну даними між різними компонентами системи. Наприклад, при створенні мікрослужб, де різні служби можуть бути розгорнуті незалежно, JWT може використовуватися для забезпечення безпеки та цілісності обмінюваних даних [12].

Основною метою JWT є забезпечення безпеки та достовірності обмінюваних даних в розподілених системах. Використання стандартизованого формату токена спрощує інтеграцію та забезпечує надійний механізм автентифікації та авторизації.

JWT також дозволяє зберігати корисну інформацію в самому токені, зменшуючи необхідність звертатися до центрального сервера для додаткових

даних. Це робить JWT ефективним інструментом для забезпечення безпеки та продуктивності в сучасних динамічних веб-системах.

JSON Web Token став важливим елементом веб-безпеки, забезпечуючи безпечний обмін даними в розподілених середовищах. Його логіка роботи, методи використання та мета відкривають нові можливості для забезпечення інтеграції та безпеки у різних додатках та сервісах.

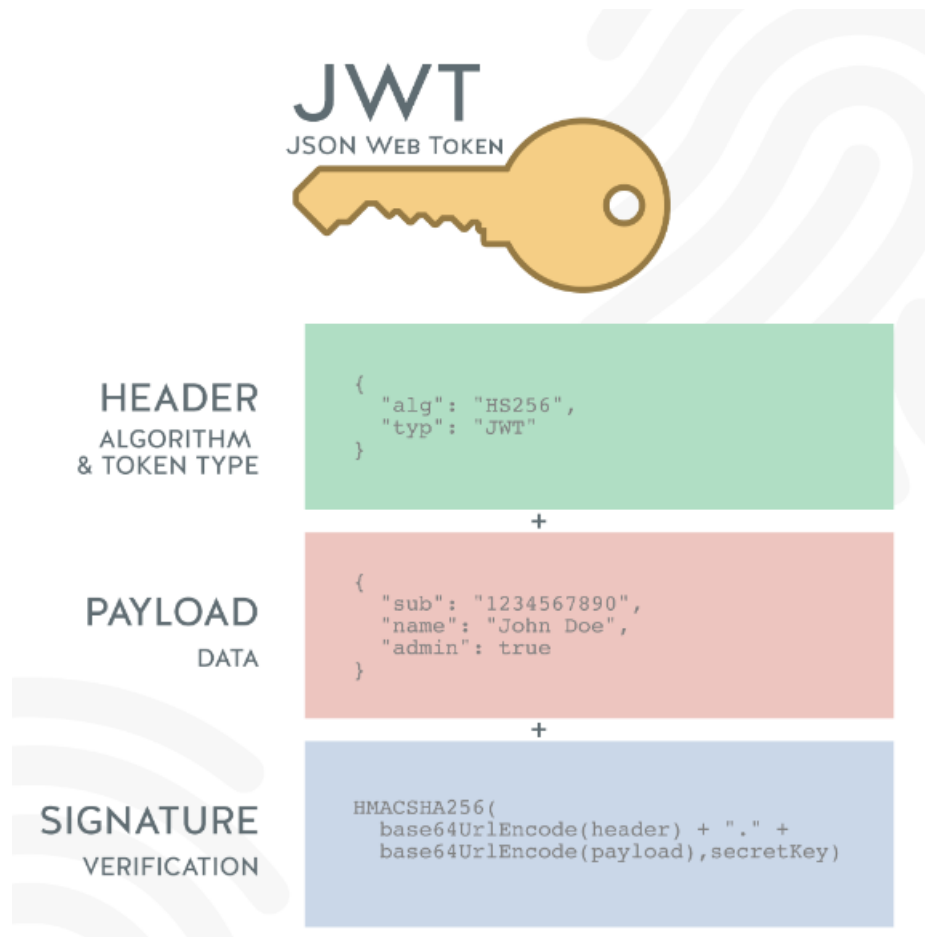


Рисунок 2.3 – Принцип роботи JWT

OAuth (від англійського "Open Authorization") є протоколом авторизації, розробленим з метою створення безпечних та стандартизованих засобів авторизації користувачів у веб-сервісах та додатках. Основною ідеєю OAuth є надання третім сторонам обмеженого доступу до ресурсів користувача без необхідності розкриття його облікових даних.

Протокол заснований на розділенні процесу автентифікації та авторизації, щоб забезпечити більшу безпеку та конфіденційність. У контексті OAuth,

автентифікація відповідає за перевірку ідентичності користувача, тоді як авторизація надає дозвіл на доступ до конкретних ресурсів чи функціональностей.

Логіка роботи OAuth полягає у взаємодії між трьома основними сутностями: власником ресурсу (користувачем), сервером авторизації та сервером ресурсів. Процес включає запит авторизації, отримання токenu доступу та використання цього токenu для отримання ресурсів [4].

Існують різні методи використання OAuth, включаючи автентифікацію за допомогою сторонніх служб, таких як Google або Facebook, а також власні реалізації протоколу для взаємодії між власними сервісами.

Головна мета технології полягає в тому, щоб спростити та забезпечити безпеку процесу авторизації в мережі. OAuth дозволяє користувачам контролювати доступ до своїх ресурсів, не розкриваючи свої облікові дані стороннім сервісам. Це робить технологію широко використовуваною в інтернет-сервісах та додатках для забезпечення безпеки та зручності для користувачів.

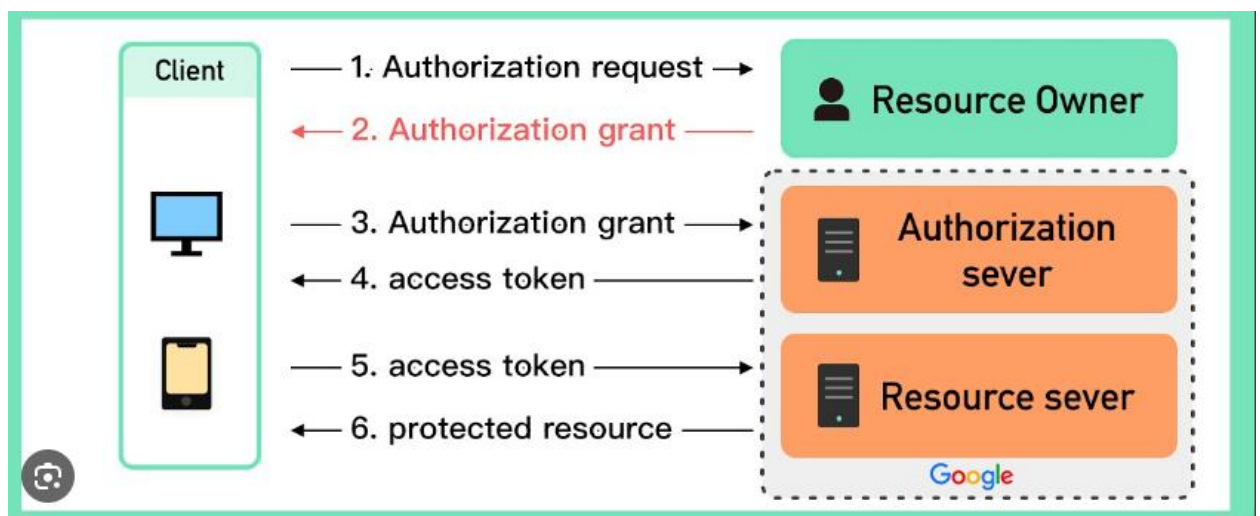


Рисунок 2.4 – Принцип роботи OAuth

OpenID Connect (OIDC) представляє собою розширення протоколу аутентифікації OAuth 2.0, яке надає прозорий та безпечний механізм ідентифікації користувачів в інтернет-сервісах. Ця технологія стала важливою складовою сучасної системи безпеки та ідентифікації в мережі.

Основна ідея OIDC полягає в тому, щоб дозволити клієнтській програмі отримати доступ до інформації про користувача від провайдера ідентичності за

допомогою токенів. Крім того, OIDC використовує JSON Web Tokens (JWT) для представлення інформації про аутентифікацію та авторизацію.

У процесі взаємодії між клієнтом, провайдером ідентичності та ресурсним сервером використовуються різні типи токенів, такі як ID токен, токен доступу та токен оновлення, які забезпечують безпеку та конфіденційність обміну інформацією [30].

Головною метою OIDC є створення відкритого та стандартизованого протоколу для реалізації аутентифікації та авторизації в розподілених системах. Це сприяє впровадженню єдиної системи ідентифікації в інтернеті, зменшуючи ризики безпеки та сприяючи прозорій обміну інформацією між різними онлайн-сервісами.

2.2 Архітектурні підходи до безпеки API

Із стрімким розвитком технологій інтернету та розширенням використання API, безпека цих інтерфейсів стала актуальною та критичною проблемою.

Необхідно ухилятися від використання підходу до документації API, як до традиційного процесу створення обширних текстових матеріалів чи візуальних діаграм як окремих артефактів. Письмові матеріали, слайди чи графічні схеми можуть бути важливими для відповідності чи входити в структуру аналізу дизайну.

Деякі організації намагаються перенести акцент з традиційних методів відповідності та каскадного підходу в справжній безпеці. Однак існує тиск на багато IT-організацій, оскільки групи розробників впроваджують гнучкі методології та DevOps-практики, які стають невідокремленою частиною пакетних угод із розробкою API. Традиційна документація може бути корисною для перевірки безпечного дизайну та моделювання загроз.

Проте, традиційні форми документації відомі своєю важкодоступністю та високою вартістю створення та підтримки. Ця обставина може призвести до відхилення у розвитку API, і документація API, ймовірно, не віддзеркалить реальний стан вашого виробничого розгортання, навіть за наявності додаткових зусиль для документування всіх внесених змін [9].

Визначення схем API визначено для спрощення процесу розробки та управління API. Визначення схем може бути створено та задокументовано під час проектування API, і в подальшому використовуватися для тестування, інтеграції, публікації та операцій. Багато інструментів для проектування та розробки автоматизовано генерують визначення схем API під час інтеграції чи кодування API. Використання цих функцій може зменшити навантаження на документацію та уникнути проблем у подальшому.

Відкритий вихідний код, пакети програмного забезпечення, які включають веб-API, часто включають відповідні визначення схем API у сховищах коду (наприклад, git) або менеджерах пакетів (наприклад, npm), з яких їх можна отримати. Для REST API найпоширенішими машинними форматами є специфікації Swagger або OpenAPI (OAS). Залежно від конкретності дизайну API та інструментів розробки чи публікації, можуть використовуватися інші формати, такі як RAML або API Blueprint. Досліджуючи GraphQL API, слід розглядати використання визначень схем GraphQL.

Ефективна робота з визначеннями схем API є ключовим етапом у процесі розробки та управління. Обрані формати визначень схем визначаються специфікою API та інструментами, що використовуються. Автоматизовані засоби та використання відкритих стандартів сприяють забезпеченню консистентності та ефективності у роботі з API [31].

Недостатній контроль доступу до API означає відсутність належних заходів щодо обмеження або регулювання доступу до API, що може призвести до несанкціонованого доступу, неправильного використання або експлуатації конфіденційних даних або функціональних можливостей. Це потенційно може призвести до порушень безпеки, витоку даних та інших форм кібератак. В даному розділі розглядається тема "Архітектурні підходи до безпеки API", зокрема у контексті трьох популярних стандартів: RESTful API, GraphQL та SOAP.

2.2.1 RESTful API та його безпекові принципи

RESTful API використовує HTTP-методи для виконання операцій, і його безпека базується на принципах автентифікації та авторизації. Використання токенів доступу, обмеження прав доступу і валідація даних допомагають у запобіганні несанкціонованому доступу та атакам. Він підтримує автентифікацію SSL і HTTPS для досягнення безпечного зв'язку.

REST використовує стандарт JSON для використання корисних даних API, що спрощує передачу даних через браузер. REST не має стану – кожен запит HTTP містить всю необхідну інформацію, тобто ні клієнт, ні сервер не зобов'язані зберігати будь-які дані для задоволення запиту. На відміну від SOAP, який потребує синтаксичного аналізу та маршрутизації для кожного запиту для роботи в локальній веб-службі, REST використовує стандартні HTTP-запити та не вимагає перепакування даних.

Даний підхід використовує HTTP-методи, такі як GET, PUT, POST і DELETE, для реалізації операцій щодо ресурсів, що включають читання, оновлення, створення та видалення.

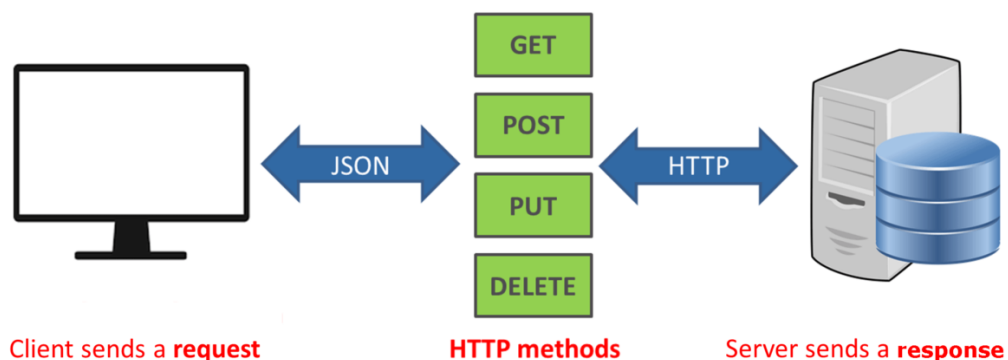


Рисунок 2.5 – Методи RESTful APIs

API для веб-сайту визначає код, що дозволяє взаємодіяти двом програмам. Воно встановлює правильний спосіб для розробника написати програму, яка запитує послуги від операційної системи чи іншої програми.

RESTful API, відомий також як веб-служба RESTful або REST API, ґрунтується на принципах передачі репрезентативного стану (REST). Цей архітектурний стиль та підхід широко використовуються у розробці веб-сервісів.

Технології REST найчастіше обираються через їхню низьку пропускну здатність, що робить їх ефективними для використання в Інтернеті.

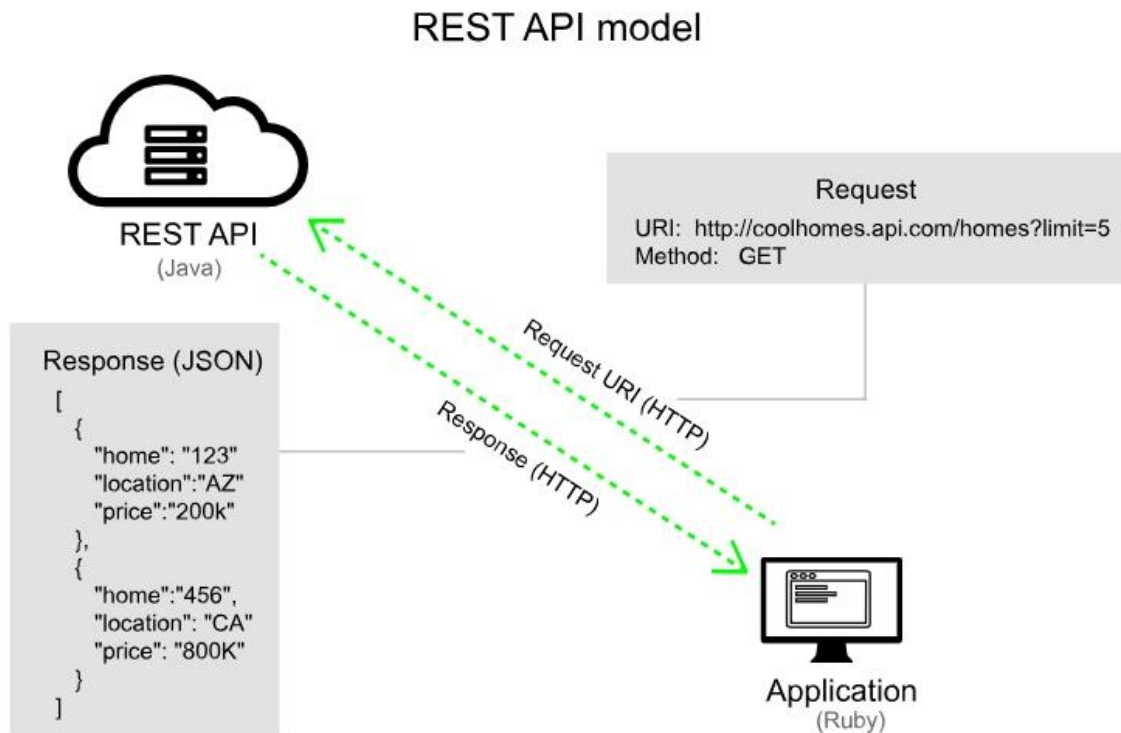


Рисунок 2.6 – Принцип роботи RESTful APIs

Методика та Особливості:

RESTful API розбиває транзакцію на серію невеликих модулів, кожен із яких відповідає конкретній частині транзакції. Це забезпечує велику гнучкість для розробників, хоча може виникнути складність при створенні API з нуля. Деякі компанії пропонують розробникам моделі для використання, серед яких популярними є Amazon S3, Cloud Data Management Interface (CDMI) і OpenStack Swift.

RESTful API використовує команди для отримання ресурсів, а стан ресурсу у будь-який момент часу називається представленням ресурсу. Використання методологій HTTP, таких як GET для отримання ресурсу, PUT для зміни чи оновлення ресурсу, POST для створення та DELETE для видалення, є типовим для REST API.

У мережевій архітектурі REST, компоненти розглядаються як ресурси, доступ до яких запитує користувач. Деталі реалізації ресурсів залишаються

невідомими, що сприяє їхній ізоляції. Дзвінки не мають стану, і нічого не може бути збережено службою RESTful між викликами.

Оскільки виклики до RESTful API не зберігають стану, цей архітектурний стиль стає вкрай корисним у хмарних програмах. Компоненти, які не зберігають стан, можуть легко перерозгортатися у випадку неполадок, а також масштабуватися відповідно до змін навантаження. Це пояснюється можливістю спрямовувати будь-який запит до будь-якого екземпляра компонента; нічого не потрібно зберігати для наступної транзакції. Такий підхід робить REST найбільш придатним для веб-сценаріїв. Модель RESTful також демонструє свою ефективність у хмарних службах, де контроль того, як URL-адреси декодуються, стає питанням строгої влади через API. Майбутнє хмарних обчислень та мікросервісів майже впевнено робить дизайн RESTful API стандартом.

Дизайн RESTful API визначив доктор Рой Філдінг у своїй докторській дисертації 2000 року. Щоб бути справжнім RESTful API, веб-служба повинна відповідати шести архітектурним обмеженням REST:

Використання Єдиного Інтерфейсу: Ресурси повинні бути однозначно ідентифіковані за допомогою єдиної URL-адреси, а маніпуляції ресурсом можуть здійснюватися лише за допомогою основних методів мережевого протоколу, таких як DELETE, PUT і GET з HTTP.

Клієнт-Серверна Архітектура: Повинно бути чітке розмежування між клієнтом і сервером, що дозволяє незалежний розвиток обох. Керування інтерфейсом користувача та збір запитів належать до домену клієнта, тоді як доступ до даних, керування навантаженням і забезпечення безпеки є доменом сервера.

Операції Без Стану: Операції клієнт-сервер не повинні зберігати стан, і будь-яке керування станом має відбуватися на рівні клієнта, а не сервера.

Кешування Ресурсів RESTful: Усі ресурси повинні дозволяти кешування, якщо це можливо, якщо не вказано, що кешування неможливе.

Пошарова Система: REST дозволяє побудову архітектури з декількох рівнів серверів.

Код на Вимогу: У більшості випадків сервери надсилають статичні представлення ресурсів у формі XML або JSON, проте, за необхідності, вони можуть відправити виконуваний код клієнту.

До появи REST розробники широко використовували SOAP для інтеграції API. Виклик API вимагав вручну написаного XML-документа з Remote Procedure Call (RPC) у тілі. Після цього розробники вказували кінцеву точку та відправляли свій SOAP-конверт до призначеної точки.

У 2000 році Рой Філдінг та група розробників розробили стандарт, спрямований на можливість спілкування будь-якого сервера з будь-яким іншим. Вони визначили REST та його архітектурні обмеження в докторській дисертації 2000 року, захищеній в Каліфорнійському університеті в Ірвайні. Ці універсальні правила значно полегшують розробникам інтеграцію програмного забезпечення.



Рисунок 2.7 – Засновник архітектурного стилю RESTful

Першою компанією, яка включила API в свій пакет "Інтернет як послуга", стала Salesforce у 2000 році. Проте, лише обмежена кількість розробників могла використовувати складний XML API. Пізніше eBay розробив REST API, що розширило його застосування до будь-якого сайту з доступом до API. Це спричинило зацікавленість інших компаній, і у 2002 році Amazon анонсувала свій API.

Flickr випустив свій RESTful API у серпні 2004 року, що дозволило блогерам легко вставляти зображення на свої сайти та соціальні мережі. У 2006 році Facebook

і Twitter представили свої API, відповідаючи на тиск розробників, які складали API "Франкенштейн" з різних сайтів. Запуск хмарних послуг Amazon Web Services (AWS) у 2006 році забезпечив розробникам миттєвий доступ до обчислювальних ресурсів через REST API AWS, і популярність публічних API стрімко зростає.

З того часу розробники широко використовують RESTful API для розширення можливостей своїх веб-сайтів і програм. Сьогодні REST API розглядають як "основу Інтернету", визначаючи новий рівень доступності та взаємодії в програмному забезпеченні.

2.2.2 GraphQL та безпека запитів у ньому

GraphQL є мовою запитів, але її також можна використовувати для маніпулювання даними. Facebook створив GraphQL для вирішення двох проблем. Перше полягало в зменшенні надмірних викликів веб-API. По-друге, потрібно мати справу з отриманням занадто великої або занадто малої кількості даних, що іноді властиво дизайну REST API. Переваги інтерфейсної продуктивності викликають більший інтерес до GraphQL, хоча REST API все ще, ймовірно, залишаться в кадрі.

У основі кожного GraphQL API знаходиться структура даних, відома як схема. Ця схема служить описом типів даних, з якими API може взаємодіяти, а також визначає, які типи даних можна отримати у відповідь на запит (ці типи детально визначаються в системах типів GraphQL). Таким чином, користувач API повинен мати знання про те, які об'єкти можна отримати, які поля вибрати, і які дані доступні у внутрішніх об'єктах та інше. Саме цю інформацію надає схема.

Важливо відзначити, що при використанні GraphQL API клієнту не потрібно знати деталі того, де та як обробляються дані, які він запитує. Клієнт просто надсилає свій запит з необхідним обсягом інформації, а сервер GraphQL повертає результат. Отже, можна уявити, що схема виступає як контракт між API і клієнтом. Перед тим, як клієнт виконує будь-який запит, цей запит вважається дійсним відповідно до схеми даного API.

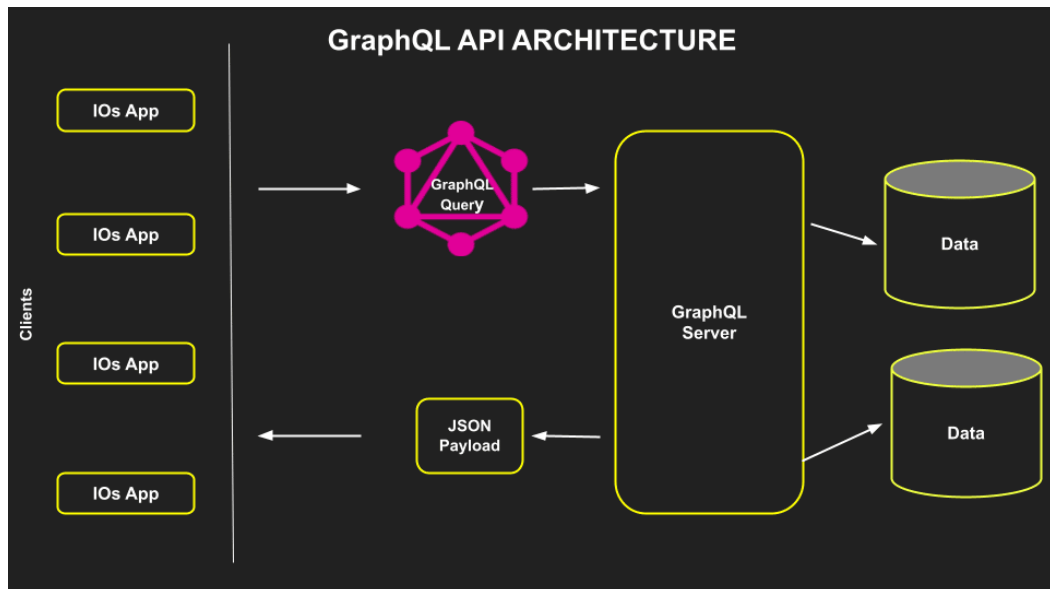


Рисунок 2.8 – Принцип роботи GraphQL

Робота з GraphQL на сьогоднішній день визначається високою підтримкою на різноманітних платформах, таких як веб, Android, iOS та інші. Процес взаємодії з GraphQL полягає в тому, що клієнт GraphQL формує запит для отримання або зміни даних, узгоджений із визначеною схемою, та надсилає його на GraphQL-сервер. GraphQL-сервер, у свою чергу, виступає в ролі HTTP-сервера, пов'язаного з визначеною схемою GraphQL. Це означає, що всі запити, отримані від клієнта, і повернуті відповіді, проходять через цю схему.

Слід відзначити, що сервер GraphQL не може автоматично визначити, як обробити отриманий запит, якщо йому не надали конкретних інструкцій. Для цього використовуються спеціальні функції, які служать розпізнавачами або резолверами (resolvers). Ці функції пов'язані із конкретними полями схеми і визначають, як отримати дані для відповідних запитуваних полів. Після виконання цих функцій, клієнту повертається відповідь, яка відображає структуру даних, запитаних з клієнта, зазвичай у форматі JSON.

Важливо підкреслити, що GraphQL дозволяє працювати з різноманітними джерелами даних, такими як бази даних (реляційні/NoSQL), результати веб-пошуку, Docker та інші.

Як було вказано раніше, GraphQL розроблявся як ефективна альтернатива архітектурі REST для розробки програмних інтерфейсів додатків.

Обидва ці підходи мають спільні риси:

1. Надають єдиний інтерфейс для взаємодії з віддаленими даними.
2. Зазвичай повертають відповідь у форматі JSON на запит.
3. Дозволяють відокремлювати запити на читання та запис.

Проте GraphQL володіє численними перевагами порівняно з REST:

1. Клієнтоорієнтований підхід, що дозволяє клієнту отримати лише ту інформацію, яка йому необхідна, ні більше, ні менше.
2. Вимагає лише одного запиту для виконання, відмінно від REST.
3. Сильна типізація мови GraphQL дозволяє передбачити правильність запиту до виконання програми.
4. Можливість комбінувати запити для отримання комплексної інформації.
5. Запити до GraphQL API завжди повертають очікуваний результат, який відповідає схемі даних GraphQL API.

Використання GraphQL дозволяє зменшити обсяг передаваних клієнту даних, оскільки клієнт запитує лише необхідну інформацію. У відмінну від REST, де сервер визначає форму та обсяг переданих даних, GraphQL дозволяє клієнту контролювати обсяг отриманої відповіді.

Практичне використання GraphQL сприяє збільшенню незалежності front-end та back-end-розробників. Після узгодження схеми, front-end-розробники можуть без додаткових запитань створювати запити та об'єднувати дані, оскільки back-end-розробники один раз описують схему даних.

2.2.3 SOAP як альтернативний підхід та його безпекові аспекти

SOAP (Simple Object Access Protocol) – це протокол обміну повідомленнями на основі XML для обміну інформацією між комп'ютерами. Вбудований у SOAP стандарт WS-Security використовує шифрування XML, підпис XML і маркери SAML для вирішення питань безпеки обміну транзакційними повідомленнями. SOAP також підтримує рекомендації OASIS і W3C.

Вбудовані стандарти SOAP і транспортування корисного навантаження у форматі конверта вимагають більших витрат порівняно з роботою з іншими реалізаціями API, такими як REST. Однак організації, які потребують більш комплексної безпеки та відповідності, можуть отримати вигоду від використання SOAP.

API SOAP все ще можна побачити в деяких внутрішніх API та бізнес-додатках. Однак реалізації API SOAP часто вважаються надто важкими, і вони були значною мірою витіснені дизайном REST API.

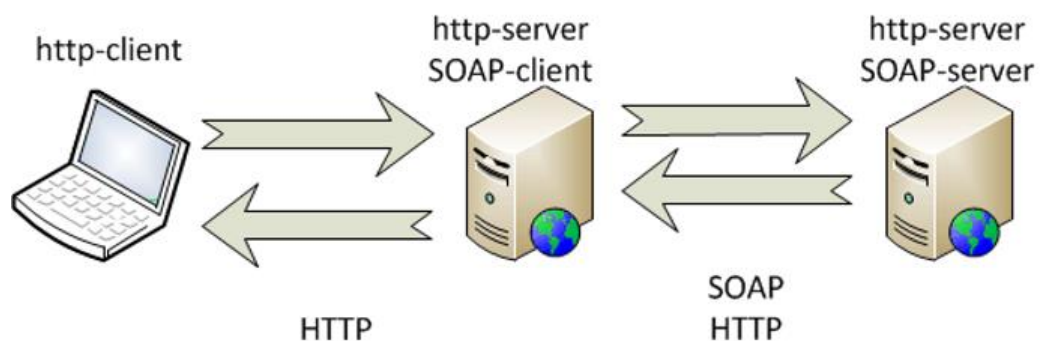


Рисунок 2.9 – Принцип роботи SOAP

SOAP представляє собою протокол, що визначає жорсткі правила комунікації, пов'язані з рядом стандартів, що регулюють різні аспекти обміну даними. Декілька з цих стандартів перераховані нижче.

1. Web Services Security (WS-Security): визначає заходи безпеки, такі як використання унікальних ідентифікаторів, відомих як токени.
2. Web Services Addressing (WS-Addressing): вимагає включення метаданих у вигляді маршрутної інформації, забезпечуючи контекст комунікації.
3. WS-ReliableMessaging: стандартизує обробку помилок у повідомленнях SOAP, забезпечуючи надійну доставку повідомлень.
4. Мова опису веб-сервісів (Web Services Description Language, WSDL): визначає область застосування та функції веб-сервісів SOAP, служачи мовою опису їхньої структури та функціоналу.

При відправці запиту до SOAP API необхідно упакувати HTTP-запит в SOAP-обгортку. Ця структура даних модифікує базовий HTTP-контент згідно з вимогами до SOAP-запитів. Завдяки обгортці також можна відправляти запити до

веб-сервісів SOAP за допомогою інших транспортних протоколів, таких як TCP чи Протокол міжмережових управлінських повідомлень (Internet Control Message Protocol, ICMP). Важливо відзначити, що SOAP API та веб-сервіси SOAP завжди повертають у своїх відповідях XML-документи.

Повідомлення SOAP структуроване наступним чином:

1. Envelope: Кореневий елемент, що визначає повідомлення та простір імен, використаний у документі. Цей елемент визначає загальну структуру повідомлення.
2. Header: Містить атрибути повідомлення, такі як інформація про безпеку чи мережеву маршрутизацію. Важливий для передачі додаткових даних, що необхідні для коректної обробки повідомлення.
3. Body: Містить саме повідомлення, що обмінюють програми. Цей елемент включає в себе основний вміст, який передається між взаємодіючими системами.
4. Fault: Необов'язковий елемент, призначений для надання інформації про помилки, які виникли під час обробки повідомлень. Використовується для передачі деталей про помилки та їхнє ефективне вирішення.

Дана структура надає чіткий каркас для обміну даними між програмами, визначаючи елементи, які служать різним цілям у процесі комунікації за допомогою протоколу SOAP.

SOAP був спочатку представлений у 1998 році Дейвом Уайнером, Доном Боксом і Бобом Аткінсоном, які в той час працювали в UserLand Software. Цей протокол був концептуально розроблений для надання простого та стандартизованого засобу обміну даними та виклику методів у віддалених системах через Інтернет. Його виникнення визначало намір виправити недоліки раніше існуючих протоколів віддаленого виклику процедур (RPC), таких як CORBA і DCOM, які виявилися складними та непридатними для ефективного використання через Інтернет.

У 1999 році, разом із UserLand, до розробки SOAP приєдналися Microsoft і DevelopMentor, а протокол був переданий на стандартизацію в W3C (World Wide

Web Consortium). Протягом наступних років специфікація SOAP пройшла кілька ітерацій та ревізій, спрямованих на удосконалення протоколу та вирішення технічних та концептуальних питань.

Однією з ключових проблем, які виникли при розробці SOAP, було забезпечення безпеки та автентифікації в розподіленому гетерогенному середовищі. Це призвело до розробки специфікації WS-Security, яка визначила основу для забезпечення захисту повідомлень SOAP та гарантувала їхню цілісність та конфіденційність.

Іншим значущим етапом в історії SOAP стало появлення конкуруючого архітектурного стилю для веб-сервісів, відомого як Representational State Transfer (REST). REST підкреслював простоту, масштабованість та слабкий зв'язок. У той час як SOAP базувався на строгому форматі повідомлень та стандартизованих протоколах, REST забезпечував велику гнучкість та використовував існуючі веб-технології, такі як HTTP та URI.

Незважаючи на появу REST та інших альтернативних технологій веб-сервісів, SOAP широко використовувався в корпоративних додатках та веб-сервісах протягом початкових років 2000-х. Він підтримувався різними мовами програмування та платформами, а його стандартизований формат повідомлень та транспортні механізми робили його високосумісним.

Однак у останні роки SOAP став менш популярним, оскільки розробники віддають перевагу більш легким та гнучким технологіям веб-сервісів, таким як JSON та REST. Незважаючи на це, SOAP залишається важливим елементом історії веб-сервісів, а його вплив виявляється у використанні XML та інших стандартизованих технологій у сучасній веб-розробці.

В даному контексті пропонується детальне порівняння між двома ключовими протоколами обміну даними: SOAP і REST. Порівнюючи їхні основні характеристики, можна визначити відмінності і спільні риси, що допомагають визначити, який з цих підходів краще відповідає конкретним вимогам та завданням веб-розробки та інтеграції сервісів.

Порівняння SOAP і REST

SOAP	REST
Архітектура та дизайн	
Це протокол обміну повідомленнями, що використовує XML для форматування повідомлень і встановлює набір правил для кодування складних типів даних у форматі XML. SOAP покладається на використання WSDL (мова опису веб-служб) для опису інтерфейсу веб-служби та ґрунтується на моделі Remote Procedure Call (RPC).	Це архітектурний стиль для створення веб-сервісів, які використовують HTTP як базовий протокол. REST базується на принципах Representational State Transfer (RESTful) та використовує методи HTTP (GET, POST, PUT, DELETE) та URI для операцій над ресурсами.
Формат даних	
Використовує XML як формат даних для повідомлень, які можуть бути великими та детальними.	Підтримує різноманітні формати даних, такі як JSON, XML та інші, але JSON стає все більш популярним через свою легкість та зручність для читання.
Продуктивність	
Має тенденцію до більших розмірів повідомлень та більших накладних витрат через використання XML, що може призвести до зниження продуктивності.	Зазвичай має менші розміри повідомлень та менше накладних витрат, особливо при використанні JSON, що може призвести до кращої продуктивності.
Безпека	
Підтримує ряд стандартів безпеки, таких як WS-Security, забезпечуючи безпеку та шифрування на рівні повідомлень.	Покладається на базовий транспортний рівень (зазвичай HTTPS) та може підтримувати OAuth для авторизації.

Отже, SOAP і REST є двома відомими архітектурами веб-сервісів з різними перевагами і обмеженнями. SOAP найкраще підходить для програм, що потребують розширених функцій обміну повідомленнями та безпеки, тоді як REST підходить для програм, що вимагають високої продуктивності, масштабовані.

2.3 Інструменти для тестування та моніторингу безпеки API

Безпека API включає в себе заходи з запобігання вразливостям та неправомірним доступом, а також забезпечення конфіденційності та цілісності даних. Інструменти для тестування та моніторингу виграють критичну роль у визначенні та усуненні можливих проблем безпеки.

2.3.1 Засоби перевірки вразливостей

Інструменти для тестування безпеки API:

OWASP ZAP (Zed Attack Proxy). Інструмент з відкритим вихідним кодом, який використовується для знаходження та усунення вразливостей в веб-додатках, включаючи API. Забезпечує сканування на предмет вразливостей типу OWASP, таких як SQL-ін'єкції та кросс-сайт-скриптинг.

Postman. Набір інструментів для розробки та тестування API. Має вбудовані функції автоматизованого тестування безпеки, які допомагають виявляти потенційні уразливості [31].

Netsparker. Автоматизований сканер веб-додатків, який допомагає виявляти та аналізувати вразливості, що можуть вплинути на безпеку API.

Моніторинг безпеки API:

Datadog. Система моніторингу та аналітики, яка дозволяє в реальному часі спостерігати за активністю API та виявляти аномалії, які можуть свідчити про атаку [5].

AWS CloudWatch. Служба моніторингу та логування в хмарному середовищі AWS. Дозволяє встановлювати метрики та сповіщення щодо подій безпеки.

Рекомендації для тестування та моніторингу безпеки API:

Документація. Важливо використовувати та дотримуватися документації API для розуміння правильного використання та обмежень системи.

Логування запитів API є важливою складовою розробки програмного забезпечення, особливо при роботі з великими та складними системами. API, або інтерфейс програмування застосунків, забезпечує можливість взаємодії між різними програмами та сервісами, дозволяючи обмінюватися даними та

функціональністю. Логування запитів API відіграє ключову роль у відстеженні та аналізі взаємодії між різними складовими системи. Нижче розглянуті основні аспекти логування запитів API та його значення.

1. Значення логування запитів API

Логування запитів API полягає у фіксації та збереженні інформації про кожен запит до API, який робиться системою чи користувачем. Це включає в себе такі дані, як час виклику, параметри запиту, тіло відповіді та будь-які помилки, які можуть виникнути під час взаємодії.

2. Відстеження взаємодії

Дозволяє відстежувати взаємодію між різними компонентами системи та ідентифікувати проблеми або неполадки. Це особливо важливо у великих розподілених системах, де можуть виникати складні ситуації, пов'язані з мережевою взаємодією, обробкою даних та іншими чинниками.

3. Аналіз продуктивності

Дозволяє виконувати аналіз продуктивності, визначаючи час відповіді на запити та виявляючи ефективність роботи API. Це може бути корисно для вдосконалення швидкодії та оптимізації викликів API.

4. Безпека та відслідковування помилок

Допомагає виявляти можливі загрози безпеці та визначати помилки в роботі API. Запис даних про невдалі запити чи невірно сформовані запити може бути важливим елементом системи виявлення та відповіді на інциденти безпеки.

5. Забезпечення аудиту

Логування також є важливим для забезпечення аудиту та відповідності, особливо в області, де існують регуляторні вимоги щодо збереження та моніторингу даних. Запис інформації про всі запити та відповіді може бути корисним для подальшого аналізу та перевірки відповідності стандартам.

Логування запитів API є важливою практикою у розробці програмного забезпечення, дозволяючи відстежувати, аналізувати та забезпечувати безпеку взаємодії між різними системами та сервісами. Це стає ключовим елементом

управління та підтримки надійності та ефективності API в будь-якій сучасній розподіленій системі.

2.3.2 Sql-ін'єкції в веб-застосунках

Ін'єкційні атаки є важливим аспектом в області кібербезпеки, належать до широкого класу векторів атак та мають потенціал нанести значні збитки системам. Особливо важливим є розгляд атак із використанням SQL-ін'єкцій, які є однією з найстаріших та найнебезпечніших загроз для веб-додатків.

Під час ін'єкційної атаки зломисник впроваджує довільний код у програму. Цей код обробляється інтерпретатором як частина команди або запиту, що може змінити логіку програми. Ін'єкції можуть викликати серйозні наслідки, такі як крадіжка даних, втрата цілісності і втрата самого об'єкта даних, відмови в обслуговуванні та повна компрометація системи. Найчастіше вони виникають через недостатню перевірку введених користувачем даних [32].

Атаки SQL-ін'єкцій виявляють особливу небезпеку для веб-програм. Хоча вони тісно пов'язані з веб-додатками, будь-які програми, які взаємодіють із системами керування базами даних, можуть стати об'єктом таких атак. SQL-ін'єкція, яка вважається однією з найпоширеніших форм ін'єкцій, виникає тоді, коли зломисник використовує вразливості веб-додатку для впровадження SQL-запитів на основі даних, введених користувачем.

Незважаючи на те, що перші прояви SQL-ін'єкцій відзначені ще у 1998 році, цей тип атак актуальний й донині. Зломисники ефективно використовують вразливості в сучасних веб-додатках. Суть атаки полягає в техніці використання уразливостей у веб-додатку, який генерує SQL-запити на основі даних від користувача. Успішна атака може призвести до несанкціонованого доступу до конфіденційної інформації, зміни даних у базах даних, а також виконання привілейованих операцій, таких як зупинка системи керування базами даних і відновлення застарілого вмісту файлів у файловій системі цих систем [3].

Впровадження SQL-операторів (SQL injection) включає різноманітні атаки, спрямовані на отримання несанкціонованого доступу, редагування та видалення

інформації з систем управління базами даних. В деяких випадках це також може вести до отримання повного контролю над сервером СКБД та його операційною системою. Ін'єкції здійснюються через веб-сервери, які формують запити до серверів СКБД на основі введеної користувачем інформації. Якщо вхідні дані недостатньо фільтруються, зломисник може змінити запит до сервера СКБД, який відправляється додатком. Цей запит виконуватиметься з тими ж привілеями, що й веб-додаток.

У більшості випадків SQL-ін'єкція можлива лише тоді, коли сервер бази даних повертає помилки через некоректно сформований запит. Ці помилки можна вважати "помилками першого роду". Однак повідомлення про помилки бази даних не завжди є очевидними. Зломисники шукають помилки в усіх можливих місцях, для чого перевіряють фрази в повідомленнях про помилки, такі як "ODBC", "SQL Server", "Syntax" тощо.

Іноді розробники приховують повідомлення про помилки від сервера бази даних, генеруючи власні помилки програмним кодом - "помилки другого роду". Це свідчить про вищий рівень програмування. Деталі помилки можуть бути приховані в "прихованому введенні" (полях HTML-сторінки), коментарях та інших місцях. Іноді веб-додатки виводять повідомлення про помилку без вказівки на неї в тілі HTTP-відповіді, проте інформація може міститися в заголовках. Деякі веб-додатки мають вбудовані функції для налагодження та тестування, які іноді залишають увімкненими або не видаляють під час релізу [7].

При плануванні впровадження брандмауера як автономного інструменту для протидії стисненню, важливо ретельно розглянути існуючі механізми захисту. Це дозволить об'єктивно оцінити функціонал брандмауера баз даних, провести аналіз переваг і недоліків різноманітних варіантів використання. Різнорічні системи управління базами даних використовують різні клієнт-серверні протоколи та діалекти SQL, зробивши неможливим універсальне застосування одного брандмауера для різних баз даних. Серед існуючих брандмауерів баз даних можна виділити MySQL Enterprise Firewall, DataArmor, Oracle Firewall та інші.

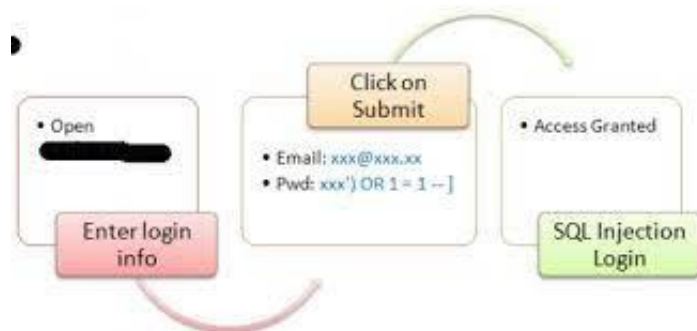


Рисунок 2.10 – Принцип роботи Sql-інекцій

Зазвичай веб-брандмауер (WAF) впроваджується у формі проксі-сервера перед веб-програмами, обмежуючи їхню здатність аналізувати весь трафік в мережах. Аналізуючи трафік до досягнення веб-програми, WAF може оцінювати запити перед їхнім пересиланням, надаючи йому перевагу перед системами запобігання вторгненням (IPS) [32]. Оскільки системи IPS призначені для обстеження усього мережевого трафіку, вони не здатні до такого детального аналізу рівня застосування, як WAF. Треба враховувати, що, незважаючи на ефективність WAF, вони не є абсолютно бездоганними, тому розглядаються методи обходу їхніх обмежень.

Висновки до розділу 2

Отже, у другому розділі "Технології розробки безпечних API" дипломної роботи було вивчено та систематизовано різноманітні аспекти, пов'язані з безпекою API. Проведений аналіз протоколів та стандартів безпеки, таких як HTTPS, а також механізмів авторизації, зокрема JSON Web Token та OAuth, дозволив визначити ключові компоненти ефективного захисту API в інформаційному просторі.

РОЗДІЛ 3 ПРАКТИЧНИЙ АСПЕКТ ЗАСТОСУВАННЯ БЕЗПЕЧНИХ API

3.1 Розробка та інтеграція безпечного API у підсистемі електронного документообігу

API Doc Flow представляє собою важливий інтерфейс для системи документообігу, призначений для ефективної взаємодії з додатком Doc Flow та іншими зовнішніми програмами. Цей інтерфейс надає широкі можливості для автоматизації обробки документів та обміну інформацією між різними організаціями.

API Doc Flow реалізований на платформі ASP.NET, що забезпечує високу стабільність та ефективність його функціонування. Інтеграція з ядром СУБД SQL Server дозволяє ефективно зберігати та управляти даними, пов'язаними з документами.

Однією з ключових особливостей API є система аутентифікації на основі JSON Web Tokens, що забезпечує високий рівень безпеки при обміні інформацією. Крім того, API реалізовано відповідно до принципів RESTful архітектури, що дозволяє здійснювати ефективний обмін запитами та отримання необхідної інформації.

Завдяки API Doc Flow користувачі можуть легко інтегрувати систему документообігу з власними програмами, розширюючи функціональність та оптимізуючи бізнес-процеси в сфері обробки документації.

Після ініціалізації програми автоматично відкривається вікно аутентифікації. Для введення у систему слід вказати код ЄДРПОУ вашої організації, ім'я користувача та його пароль. При відправці запиту програма надсилає додатковий параметр Token для додаткової безпеки, який генерується при реєстрації та оновлюється при кожному вході у програму.

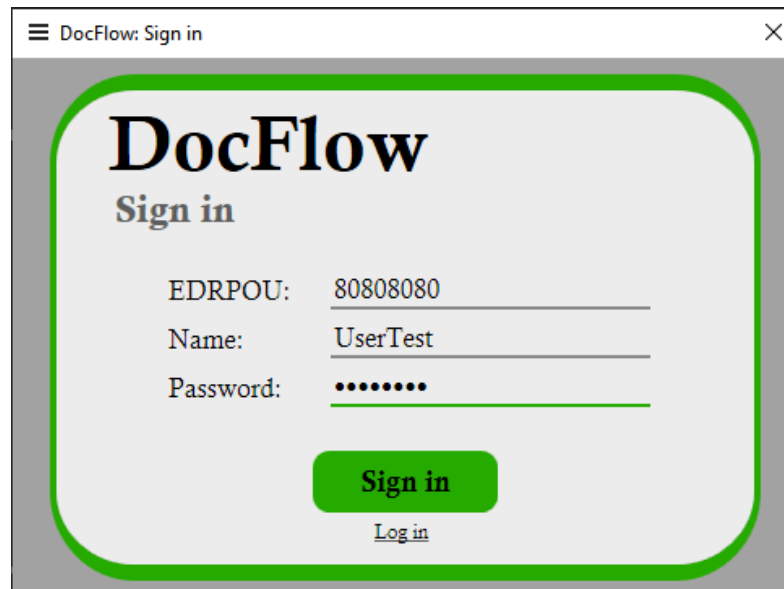


Рисунок 3.1 – Інтерфейс входу у застосунок

```
POST /api/LogIn/login
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Username": "UserTest",
  "Password": "653D716345D8915046B904B90F41F271",
  "EDRPOU": "80808080",
  "Token": "6noMW7fzqonB6tz"
}
```

Рисунок 3.2 – Запит Login від DocFlow до API

```
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Status": true,
  "Description_": "OrgTest",
  "Token": "eVwzDo5KpJlaEM1"
}
```

Рисунок 3.3 – Відповідь на запит Login від API до DocFlow

Для встановлення нової організації необхідно скористатися функціоналом, натиснувши на відповідну кнопку під назвою «Log in» та введенням необхідних даних щодо організації. Зазначте унікальне ім'я та код для вашої новоствореної організації. Зокрема, код повинен складатися з восьми цифр, і важливо забезпечити його унікальність в системі. Також врахуйте, що пароль для доступу повинен

містити не менше восьми символів. Цей процес дозволяє забезпечити безпеку та ідентифікацію організації під час реєстрації у системі.

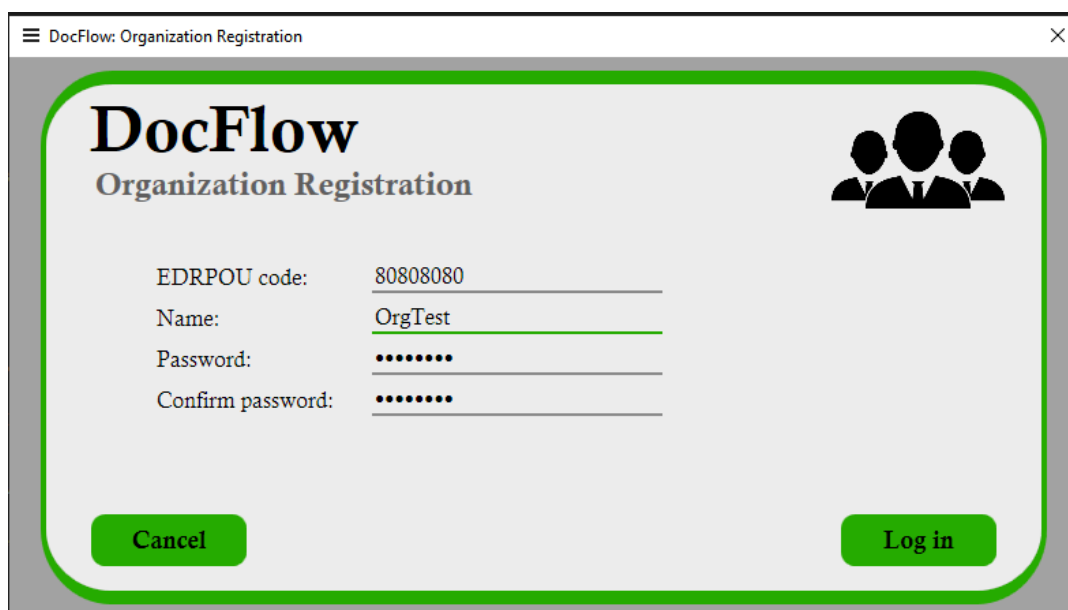
The image shows a web browser window titled "DocFlow: Organization Registration". The main content area has a light gray background with a green border. At the top left, the "DocFlow" logo is displayed in a large, bold, black serif font, with "Organization Registration" in a smaller, blue sans-serif font below it. To the right of the logo is a black icon of three stylized human figures. Below the header, there are four input fields: "EDRPOU code:" with the value "80808080", "Name:" with the value "OrgTest", "Password:" with masked characters ".....", and "Confirm password:" with masked characters ".....". At the bottom left is a green "Cancel" button, and at the bottom right is a green "Log in" button.

Рисунок 3.4 – Інтерфейс для реєстрації організації

Після введення реєстраційних даних про організацію науково обґрунтовано рекомендується провести створення першого користувача у системі. У рамках цього процесу важливо забезпечити високий рівень кібербезпеки, адже створення користувача є ключовим етапом формування доступу до інформаційних ресурсів.

Згідно наукових принципів безпеки, при створенні облікового запису в системі визначено обов'язкові параметри паролю. З метою забезпечення відповідного рівня захисту, рекомендується встановлювати пароль, що складається не менше ніж із восьми символів. Це сприяє утворенню надійного бар'єру для неповноважених осіб і забезпечує високий ступінь конфіденційності та інтегритету інформаційних ресурсів, пов'язаних з організацією.

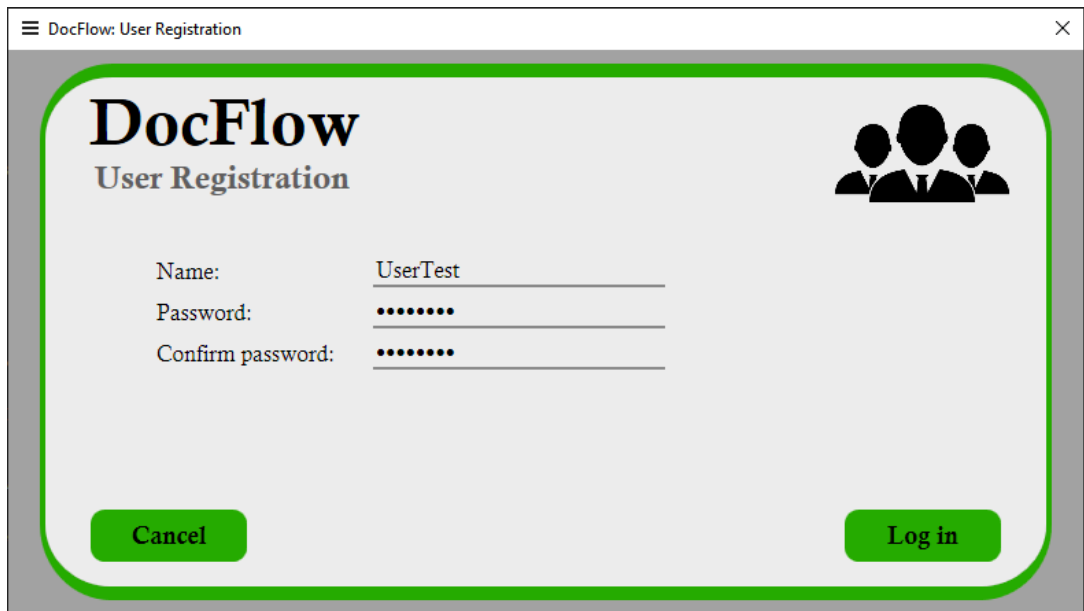


Рисунок 3.5 – Інтерфейс для реєстрації користувача

У процесі взаємодії з інтерфейсом системи, при натисканні кнопки «Log in», відбувається автоматизована перевірка введених користувачем полів. У випадку успішної реєстрації важливо врахувати, що система автоматично застосовує обов'язкове збереження приватного ключа користувача. Важливо відзначити, що в даному контексті відсутня можливість відмовитися від збереження ключа, оскільки подальше відновлення зазначеного ключа є технічно неможливим.

Цей захід забезпечення є необхідним для гарантування безпеки та невідмовності від ключового інформаційного елементу системи. Враховуючи той факт, що можливість відновлення приватного ключа недоступна, його збереження є критичною складовою процесу реєстрації. Такий підхід дозволяє забезпечити високий рівень безпеки та виключити потенційні ризики втрати доступу до системних ресурсів в майбутньому.

У рамках виконання програмного компоненту розроблено функціонал для проведення реєстрації користувачів у системі. Основна мета даного процесу полягає в наданні зареєстрованим користувачам доступу до функціоналу системи через API. Програма автоматично генерує запит типу "Registration" з метою перевірки та реєстрації користувача в базі даних.

У випадку успішної реєстрації, сервер повертає у відповіді приватний ключ, який є важливим елементом для забезпечення безпеки взаємодії із системою.

Публічний ключ, отриманий під час реєстрації, система зберігає у внутрішній базі даних, використовуючи його для подальшої ідентифікації користувача.

Окрім того, після успішної обробки запиту реєстрації, система надсилає токен, який дозволяє зареєстрованому користувачеві отримати доступ до API системи. Цей токен виступає важливим засобом автентифікації та авторизації, забезпечуючи зручний та безпечний спосіб взаємодії із функціоналом системи.

```
POST /api/Registration/registration
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Username": "UserTest",
  "UserPassword": "653D716345D8915046B904B90F41F271",
  "EDRPOU": "80808080",
  "OrgName": "OrgTest",
  "OrgPassword": "653D716345D8915046B904B90F41F271"
}
```

Рисунок 3.6 – Запит Registration від DocFlow до API

Після успішного виконання процедури реєстрації або авторизації користувача розгортається головне вікно програмного застосування. На цьому етапі відкриваються функціональні можливості щодо передачі та отримання документів. У відповідності з введеними обліковими даними користувача, відбувається активація функцій, що надають змогу здійснювати обмін інформацією у формі документів між власним обліковим записом та іншими користувачами системи.

```
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Status": true,
  "Description": "Account created successfully!",
  "Token": "1VzgEKrJDnfwwR8",
  "PrivateKey": "<RSAKeyValue><Modulus>0R00frf35Q3ifiq9eDJSAI9hEewpvQCZLe0ssAg7p1zvWcSjaJ6Uu
NOoSHR/BLIs4VK0gw8HwT/0HCzhYJDudWQCkSwk06B6V1438DvzvcajBuWlJqk5Wshoa37HwK90xpA3mH7XVioqvPAKDC591N
oI2jLYmrjbzM35NDZPTk=</Modulus><Exponent>AQAB</Exponent><P>9QLEaBtoEVJb46qR/HkxkfmsfXqmwmai4j9cJ
PHjuNFKAQ2WzNSICK6n8R18TiDqPC11ndkd6LkXbG1s2Wjyw=</P><Q>2nswNM0PsLyaQ0P5cZVyXcYcHAArnnLdP5/YABHd
L1wt41nk7YADFrya1uiZgi2TTSccPz1a3W2D0aRQXh4qiw=</Q><DP>vDFtb9KdQeTWc1TP4LEhe1fREFKKG4I0w4wrXOLiaK
NvSy9joc4bpBwEEIaICwcUaBBLL7dZJGke1TuBcGX7xmw=</DP><DQ>Z7K4TYNvCRHbnSUXhpkKrpurTR
+8ExQX74XD60r2n4Ay+cwRxfTydmrMWQx7PjYwYGVGn95Q1hget
+zP00VfUFQ=</DQ><InverseQ>eIotERzkTcI3s1UVvz9M7JQpdn2QS7W9BtrhWqQ
+1g65gg5zmZDwJwm56nEoArE69QIdVQ4ViTu0+8FDK1ftw=</InverseQ><D>YaeaYp/qXwGIAiJV0F/D/tL3Y5Zmu8UexM
sidqexrG0Dv2nNYxvEmzkG06hkdb38pdKpSFWhSohKpb+Up8J5e
+Uh7MxBanC0rHhtGnXfPS7AZI1jagJYDjKK0F/InsyENLpfCyE2ecq5gNjj5Un3lqz9bNXz2X1btr0FZann7V0=</D></RSAK
eyValue>"
}
```

Рисунок 3.7 – Відповідь на запит Registration від API до DocFlow

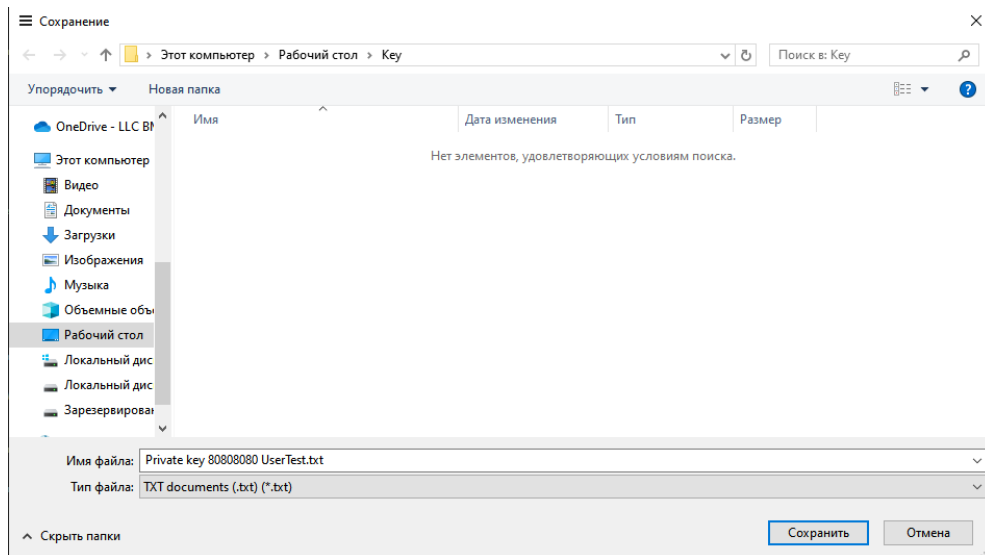


Рисунок 3.8 – Збереження приватного ключа після реєстрації



Рисунок 3.9 – Збережений приватний ключ користувача для обміну документами

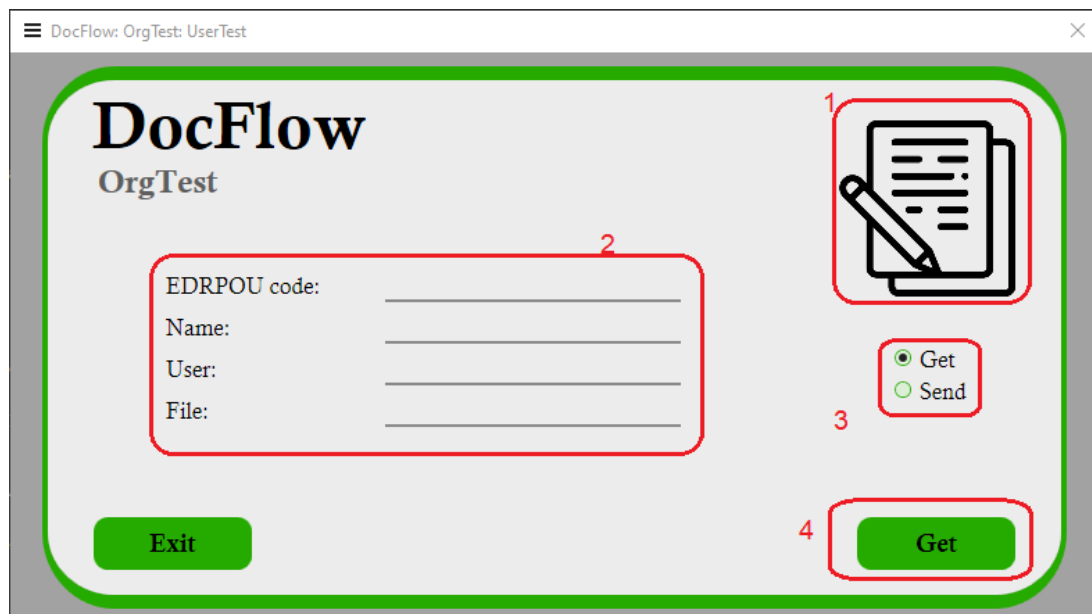


Рисунок 3.10 – Головний інтерфейс застосунку. 1 – Налаштування, 2 – реквізити для отримання або відправки документів, 3 – перемикач відправити/одержати, 4 – кнопка відправити/одержати документ

Під час активації зображення в документі спостерігається відкриття вікна параметрів. Перша сторінка цього вікна представляє собою електронну бібліотеку, яка утримує історію відправлення або отримання документів. Крім того, надається можливість завантажити порожній шаблон документу. Ця функціональність вирізняється значущістю, оскільки відправлення документу можливе лише за наявності відповідного шаблону.

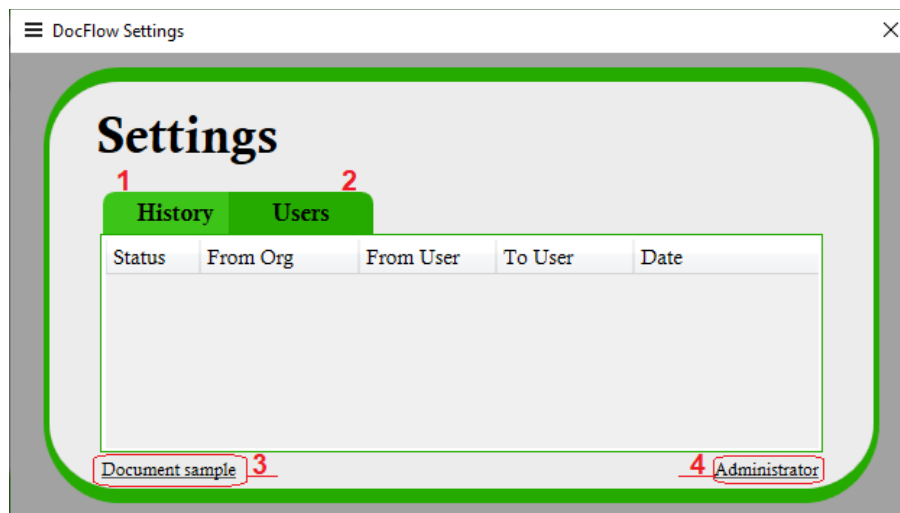


Рисунок 3.11 – Інтерфейс налаштувань. 1 – Сторінка історії обміну, 2 – список користувачів, 3 – кнопка для завантаження прикладу шаблону документа, 4 – Вхід у режим адміністратора

Під час ініціювання інтерфейсу налаштувань на сервері відбувається передача запиту типу POST із назвою FillHistory. Цей запит містить відомості, пов'язані з користувачем, організацією та токеном. В у відповідь на цей запит сервер передає масив, що містить структури, пов'язані із історією обміну документами.

```
POST /api/FillHistory/fillhistory
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "ToOrg": "80808080",
  "ToUser": "UserTest",
  "FromOrg": "11111111",
  "FromUser": "UserTest",
  "Token": "eVwzDo5KpJlaEM1"
}
```

Рисунок 3.12 –11 Запит FillHistory від DocFlow до API

```

Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Status": true,
  "History": [
    {
      "Number": 1,
      "FromOrg": "11111111",
      "FromUser": "UserTest",
      "ToOrg": "80808080",
      "ToUser": "UserTest",
      "Status": "Received",
      "Date": "02.12.2023 16:51:38"
    }
  ]
}

```

Рисунок 3.13 – Відповідь на запит FillHistory від API до DocFlow

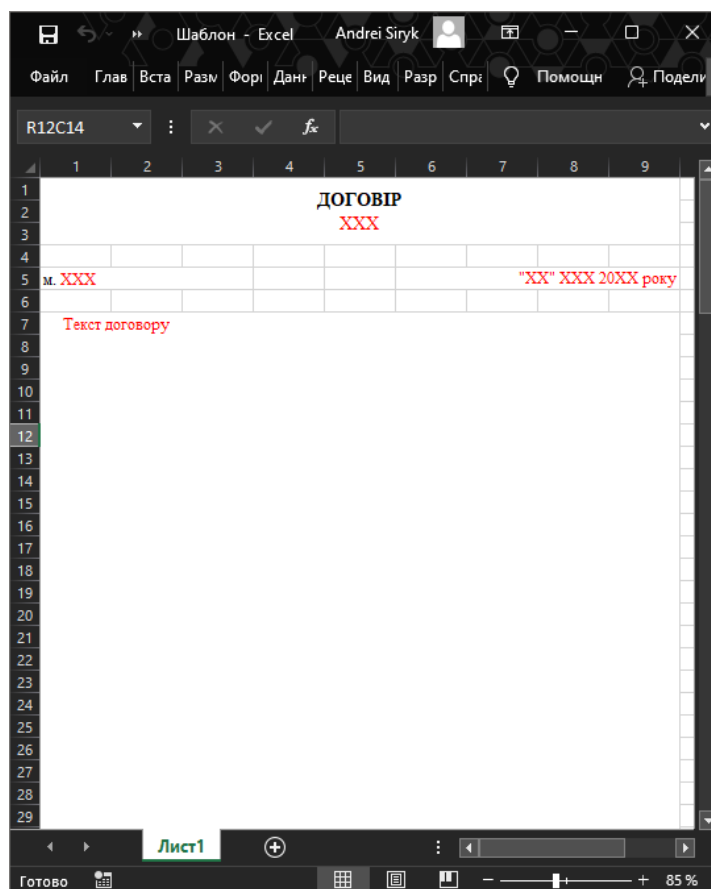


Рисунок 3.14 – Порожній документ

При натисканні на кнопку "Administrator" ініціюється відкриття вікна для введення облікового запису у режимі адміністратора, у якому необхідно вказати пароль, що належить організації. Адміністратор має повноваження виконувати операції з управління користувачами, зокрема, можливість видаляти та додавати нові облікові записи, змінювати паролі та здійснювати перегляд електронної бібліотеки для всіх користувачів системи.

У зворотньому випадку, звичайному користувачеві доступні обмежені функції, що обмежуються можливістю зміни власного пароля та перегляду історії відправки та отримання власних документів. Адміністраторські привілеї є винятковими і недоступними для звичайного користувача, що підкреслює важливість обмеженого доступу для забезпечення безпеки та конфіденційності в інформаційній системі.

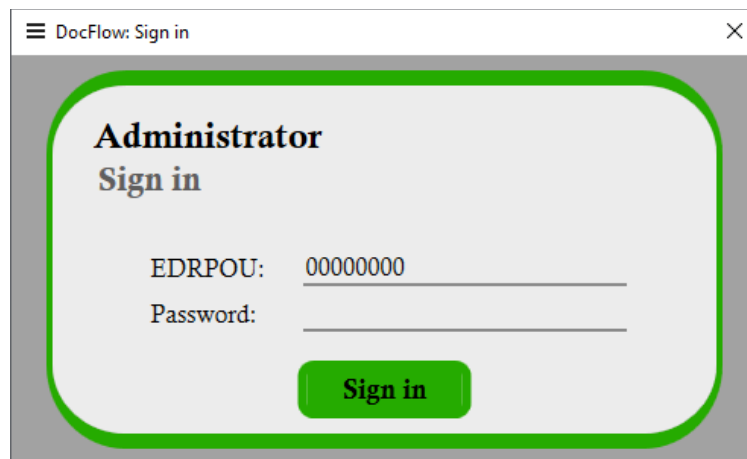


Рисунок 3.15 –12 Доступ до режиму адміністратора

У режимі адміністратора з'являються кнопки додавання, видалення користувачів, а також перезаповнюються табличні частини.

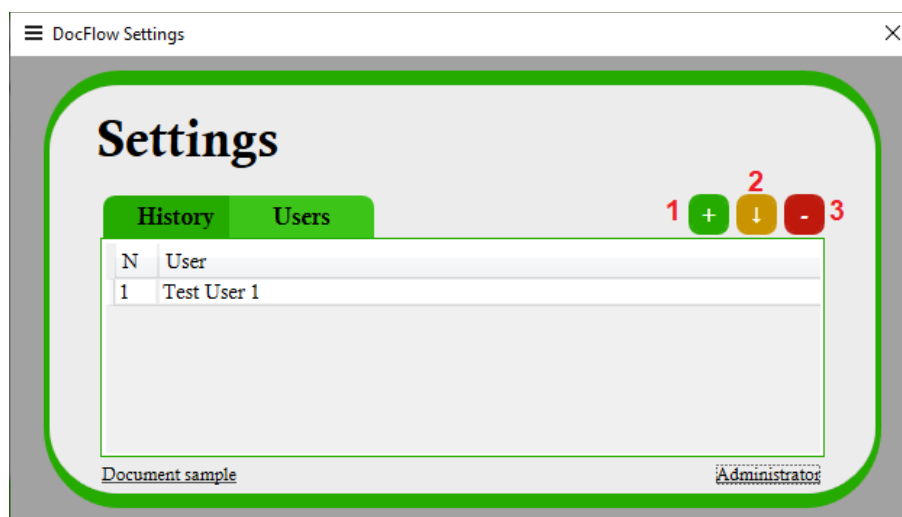


Рисунок 3.16 – Сторінка «Користувачі». 1 – Створення нового користувача, 2 – зміна пароля, 3 – видалення користувача

Під час взаємодії з користувачами системи виникає необхідність передавати запити для редагування їхніх облікових записів. Для цього використовується спеціальний запит, названий " POST EditUser". У цьому запиті міститься поле "Operation", яке вказує на тип виконуваної операції і може приймати значення

"Add" (додавання), "Del" (видалення) або "Edit" (редагування). При отриманні такого запиту сервер системи виконує відповідну операцію з редагування облікового запису користувача.

Після виконання операції сервер надсилає відповідь, яка містить інформацію про статус виконання запиту та опис результату. Це може включати в себе код статусу (наприклад, успішно чи з помилкою) і текстовий опис стану або результату виконання операції.

```
POST /api/EditUsers/editusers
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Operation": "Del",
  "Username": "UserTest",
  "Password": "",
  "EDRPOU": "80808080",
  "Token": "6noMW7fzqonB6tz"
}
```

Рисунок 3.17 – Запит EditUser від DocFlow до API

```
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Status": true,
  "Description_": "OK"
}
```

Рисунок 3.18 – Відповідь на запит EdidUser від API до DocFlow

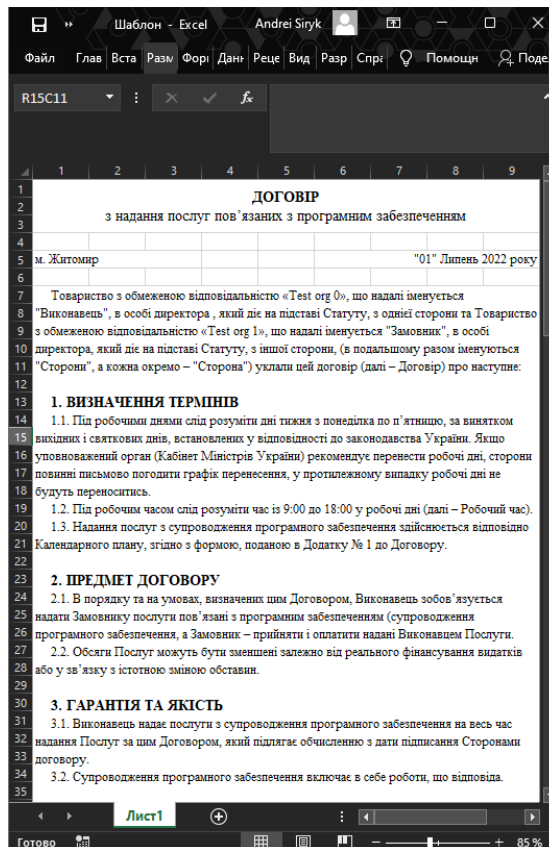


Рисунок 3.19 – Приклад документу для відправлення

Після завершення підготовки документу необхідно внести реквізити для подальшої експедиції відповідного документального матеріалу. Цей процес передбачає заповнення необхідних полів для належного визначення інформації, що міститься у документі, та підготовки його до відправлення.

У контексті визначення параметрів для експедиції слід враховувати обов'язкові реквізити, серед яких можуть бути вказані дата відправлення, отримувач, тема та інші важливі дані. Після уважного введення цих параметрів користувач має можливість вибрати відповідний файл для прикріплення до документу перед надсиланням.

Надалі, для забезпечення швидкості та точності введення інформації, в системі передбачено автоматичне заповнення інших реквізитів, таких як код організації, ім'я відправника та список користувачів. Це дозволяє значно полегшити процес введення даних, сприяючи ефективній та стандартизованій експедиції документів.

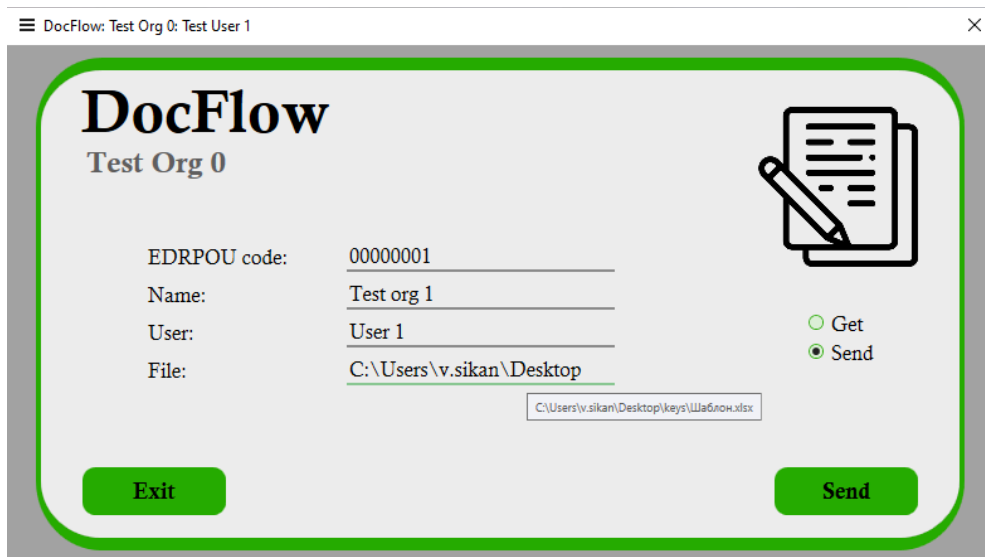


Рисунок 3.20 – Інтерфейс для відправки документа

Під час надсилання документа застосунок використовує запит «POST GetOpenKey» з метою отримання відкритого ключа організації-одержувача для подальшого шифрування документа. На запит сервер відповідає, повертаючи значення ключа у форматі XML разом із статусом запиту.

```
POST /api/GetOpenKey/getopenkey
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "EDRPOU": "80808080",
  "Name": "UserTest",
  "Token": "eVwzDo5KpJ1aEM1"
}
```

Рисунок 3.21 – Запит GetOpenKey від DocFlow до API

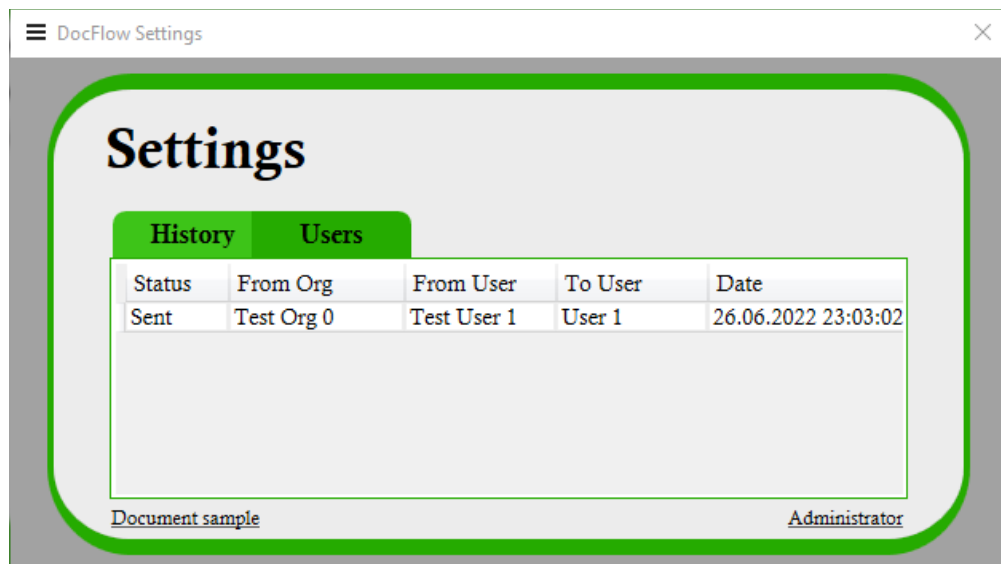
```
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Status": true,
  "OpenKey": "<RSAKeyValue><Modulus>0Ro0frf35Q3ifiq9eDJsAI9hEewpvQCZLe0ssAg7p1zvWcSjaJ6UuN0oSHR/BLIs4VK0gW8HwTj/0HCzhYJDudWQckSwk06B6V1438DvzvcAjBuWlJqk5Wshoa37HwK90xpA3mH7XVIoqvPAKDc591NoI2jLYmRjbzM35NDZPTk=</Modulus><Exponent>AQAB</Exponent></RSAKeyValue>",
}
```

Рисунок 3.22 – Відповідь на запит GetOpenKey від API до DocFlow

Після ініціювання процесу відправлення, система перевіряє коректність введених даних. У разі успішного завершення цього етапу відображається повідомлення про вдале відправлення.

На наступному етапі документ розбивається на окремі текстові фрагменти, які потім об'єднуються та конвертуються у байтову систему. Після цього масив байтів блоково шифрується за допомогою відкритого ключа підрядника. Розмір кожного блоку складає 64 байти.

У веденні історії фіксується поява документа зі статусом «Відправлено». Після отримання документа його статус автоматично змінюється.



Status	From Org	From User	To User	Date
Sent	Test Org 0	Test User 1	User 1	26.06.2022 23:03:02

Рисунок 3.23 – Графічне представлення відправленого документа та його історії обміну

Табличний розділ ілюструє хронологію подій, пов'язаних із етапами відправлення документів, а також подальшу динаміку подій у контексті історії обміну документами. Поля From/To служать для визначення напрямку документообігу від відправника до одержувача. Кожен вузол на діаграмі репрезентує ключові точки обміну інформацією, а метадані кожного етапу надають деталізовану інформацію для подальшого аналізу та вдосконалення процесу обміну документами.

Початковим етапом процедури перевірки завантаження файлу є встановлення з'єднання із SFTP-сервером за допомогою WinSCP. Це забезпечує безпечний обмін даними між локальним комп'ютером та віддаленим сервером. Наступним етапом є перевірка статусу відправленого файлу, що гарантує, що документ успішно пройшов вказаний ланцюжок подій.

Такий науковий підхід до опису процесів забезпечує точну та структуровану інформацію для подальших досліджень і вдосконалення механізмів обміну документами в організації.

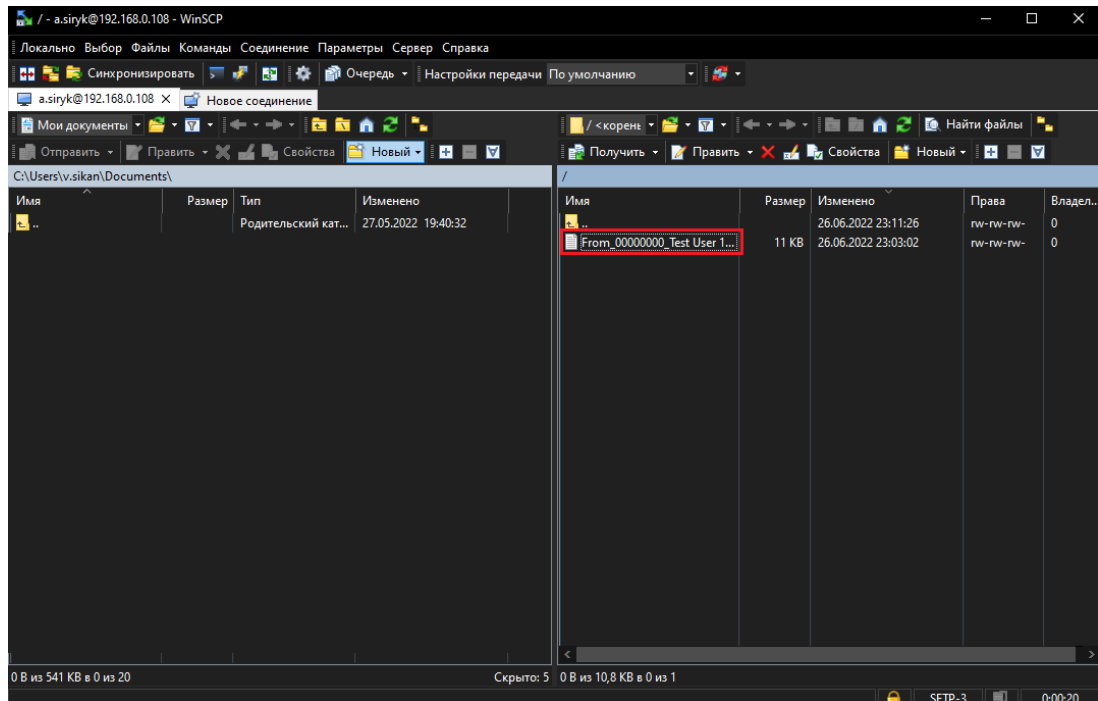


Рисунок 3.24 – Приклад відображення документу на сервері

Аналізуючи вміст, можна констатувати, що зазначений текст ефективно зашифрований.



Рисунок 3.25 – Приклад документу на сервері після шифрування

Увімкнемо програму в контексті контрагента для взаємодії та отримання документації. При отриманні документів відкривається можливість зазначити конкретну організацію та користувача, від якого ми маємо отримати документи, або не вказувати жодних параметрів і отримати всі наявні документи. При

натисканні на кнопку "Отримати" важливо обрати відповідний приватний ключ; у випадку вибору неправильного ключа виникне помилка, і буде надано відповідне пояснення щодо виниклої проблеми.

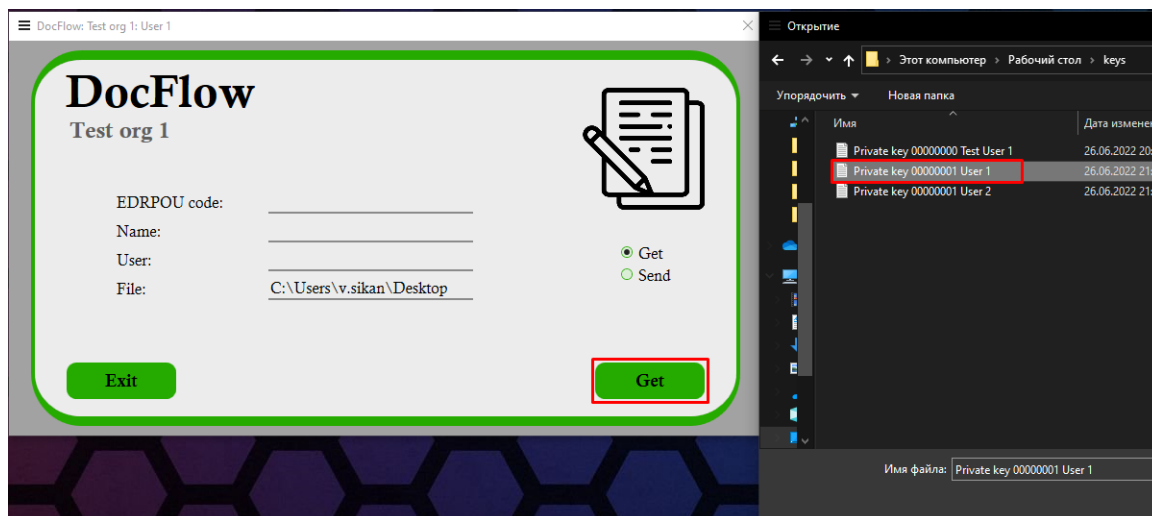


Рисунок 3.26 – Інтерфейс завантаження документу та вибір приватного ключа

Зазначений застосунок ініціює взаємодію з сервером шляхом відправки запиту «POST GetDoc». Цей запит містить необхідні параметри, які визначають організацію отримувача та організацію відправника, спрямованих на отримання конкретного документу.

На сервері, що приймає запит, відбувається обробка інформації, переданої у запиті. Зокрема, визначаються реквізити організації отримувача та організації відправника, які використовуються для ідентифікації та визначення контексту отримання документу.

У відповідь на коректний запит сервер генерує масив документів, який містить необхідні дані про кожен документ, що відповідає вказаним параметрам. Кожен елемент масиву документів містить реквізити організації та шлях до документу на сервері, який використовує протокол передачі файлів SFTP.

Таким чином, взаємодія між застосунком та сервером відбувається за стандартами безпеки та забезпечує точність та конфіденційність обміну інформацією між сторонами. Цей процес визначається чіткими принципами обміну даними, що сприяє ефективності та надійності отримання документу за допомогою вказаного протоколу.

```

POST /api/GetDoc/getdoc
Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "ToOrg": "80808080",
  "ToUser": "UserTest",
  "FromOrg": "11111111",
  "FromUser": "UserTest",
  "Token": "eVwzDo5KpJlaEM1"
}

```

Рисунок 3.27 – Запит GetDoc від DocFlow до API

```

Accept: application/json
Content-Type: application/json; charset=utf-8
{
  "Status": true,
  "History": [
    {
      "FromOrg": "11111111",
      "FromUser": "UserTest",
      "Path": "\\SFTP_Root\\From_11111111_UserTest_80808080_UserTest_02.12.2023 16.49.11"
    }
  ]
}

```

Рисунок 3.28 – Відповідь на запит GetDoc від API до DocFlow

У випадку успішного отримання документу, він буде збережений у визначеній текі у форматі .pdf.

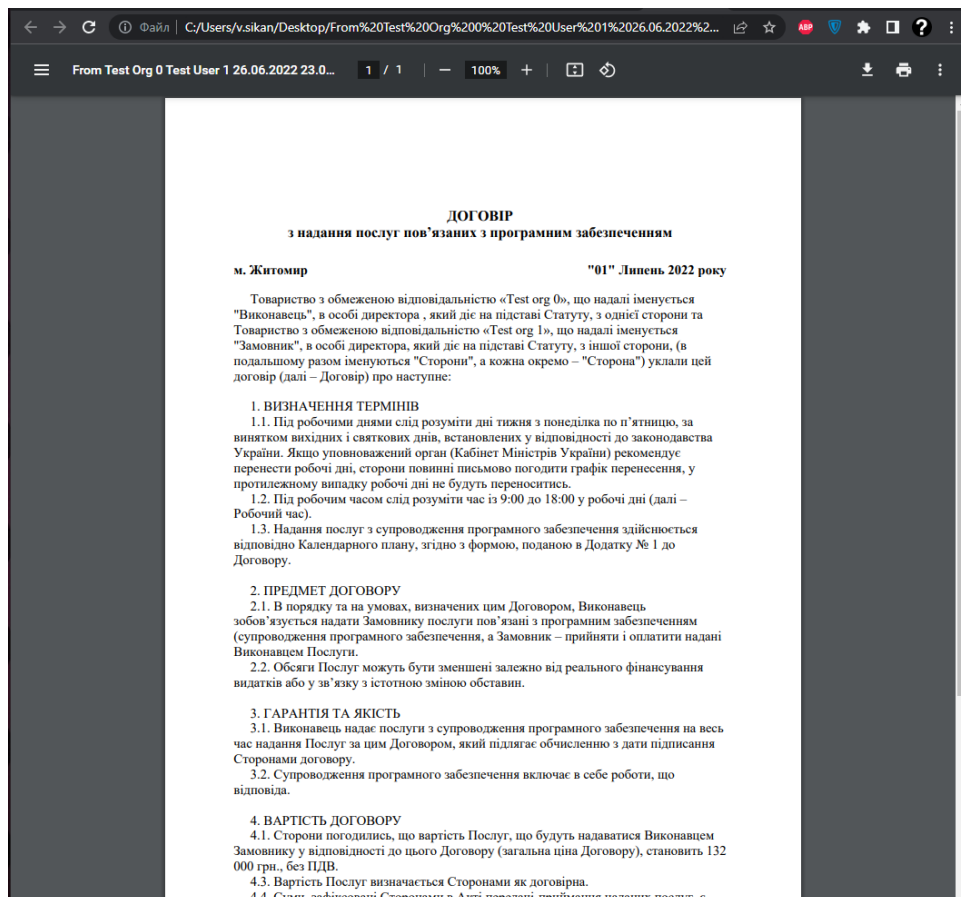


Рисунок 3.29 – Приклад одержаного документу у форматі .pdf

Локальный диск (D:) > Диплом > DocFlow > Logs			
Имя	Дата изменения	Тип	Размер
LogFile_2023-12-20_20_36_42_api_EditUsers_editusers.txt	20.12.2023 20:22	Текстовый докум...	1 КБ
LogFile_2023-12-20_20_35_53_api_FillHistory_fillhistory.txt	20.12.2023 20:22	Текстовый докум...	1 КБ
LogFile_2023-12-20_20_31_23_api_GetDoc_getdoc.txt	20.12.2023 20:22	Текстовый докум...	1 КБ
LogFile_2023-12-20_20_22_08_api_GetOpenkey_getopenkey.txt	20.12.2023 20:22	Текстовый докум...	1 КБ
LogFile_2023-12-20_20_20_35_api_Login_login.txt	20.12.2023 20:22	Текстовый докум...	1 КБ
LogFile_2023-12-20_20_19_28_api_Registration_registration.txt	20.12.2023 20:19	Текстовый докум...	2 КБ

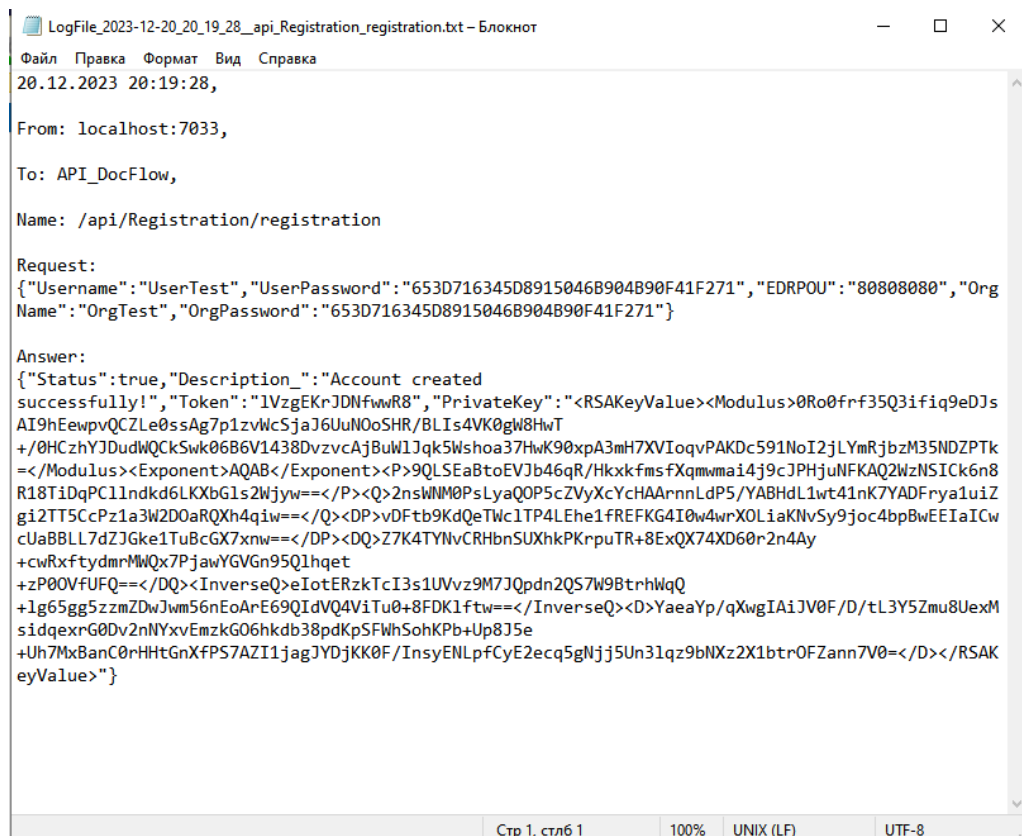
Рисунок 3.30 – Список логів запитів збережених у каталозі серверу

Список логів запитів, які зберігаються у каталозі серверу, представлений нижче:

1. Журнал вхідних запитів: Цей лог вміщує інформацію про всі вхідні запити, які отримані системою з різних джерел.
2. Журнал вихідних запитів: В цьому лозі фіксується інформація про всі вихідні запити, які система відправляє на зовнішні ресурси чи служби.
3. Журнал помилок запитів: Тут вказана детальна інформація про будь-які помилки, які виникають під час обробки запитів.
4. Журнал безпеки запитів: У цьому лозі фіксуються дані про всі запити, пов'язані з безпекою системи, такі як спроби несанкціонованого доступу чи атаки.
5. Журнал транзакцій: Цей лог містить інформацію про всі транзакції, які виконуються системою в рамках запитів користувачів.

Додатково до цього, нижче подано опис "Список логів запитів збережених у каталозі користувача":

Цей перелік включає журнали, що містять детальну інформацію про різні типи запитів, які система отримує та відправляє. Кожен лог спрямований на відстеження конкретного аспекту обробки запитів, надаючи користувачеві можливість ефективно аналізувати та вирішувати питання, пов'язані з взаємодією системи з оточуючим середовищем.



```
LogFile_2023-12-20_20_19_28_api_registration_registration.txt - Блокнот
Файл  Правка  Формат  Вид  Справка
20.12.2023  20:19:28,

From: localhost:7033,

To: API_DocFlow,

Name: /api/Registration/registration

Request:
{"Username": "UserTest", "UserPassword": "653D716345D8915046B904B90F41F271", "EDRPOU": "80808080", "Org
Name": "OrgTest", "OrgPassword": "653D716345D8915046B904B90F41F271"}

Answer:
{"Status": true, "Description_": "Account created
successfully!", "Token": "1VzgEKrJDNfwwR8", "PrivateKey": "<RSAKeyValue><Modulus>0Ro0frf35Q3ifiq9eDJs
AI9hEewpvQCZLe0ssAg7p1zvWcSjaJ6UuN0oSHR/BLIs4VK0gW8HwT
+/0HCzhYJDudWQCkSwk06B6V1438DvzvcAjBuWlJqk5Wshoa37HwK90xpA3mH7XVIoqvPAKDC591NoI2jLYmRjbzM35NDZPTk
=</Modulus><Exponent>AQAB</Exponent><P>9QLSEaBtoEVJb46qR/HkxkfmsfXqmwmai4j9cJPHjuNFKAQ2WzNSICk6n8
R18TiDqPC11ndkd6LkXbG1s2Wjyw=</P><Q>2nsWNM0PsLyaQ0P5cZVyXcYcHAArnnLdP5/YABHdL1wt41nK7YADFrya1uiZ
gi2TT5CcPz1a3W2D0aRQXh4qiW=</Q><DP>vDFtb9KdQeTWclTP4LEhe1fREFKG4I0w4wrXOLiaKNvSy9joc4bpBwEEIaICw
cUaBBL7dZJGke1TuBcGX7xnnw=</DP><DQ>Z7K4TYNvCRHbnSUXhkPKrpuTR+8ExQX74XD60r2n4Ay
+cwRxftydmrMwQx7PjauYGVGn95Q1hget
+zP00VfUFQ=</DQ><InverseQ>eIotERzkTcI3s1UVvz9M7JQpdn2QS7W9BtrhWqQ
+lg65gg5zzmZDwJwm56nEoArE69QIdVQ4ViTu0+8FDK1ftw=</InverseQ><D>YaeaYp/qXwgIAiJV0F/D/tL3Y5Zmu8UexM
sidqexrG0Dv2nNYxvEmzkG06hkd38pdKp5FWHsohKpb+Up8J5e
+Uh7MxBanC0rHntGnXfPS7AZI1jagJYDjKK0F/InsyENLpfCyE2ecq5gNjj5Un31qz9bNXz2X1btr0FZann7V0=</D></RSAK
eyValue>"}

Стр 1, столб 1    100%    UNIX (LF)    UTF-8
```

Рисунок 3.31 – Приклад файлу логу

Висновок до розділу 3

У розділі 3 увагу було зосереджено на конкретному використанні безпечних API у підсистемі електронного документообігу. Аналізуючи цей практичний аспект, було досліджено вплив безпечних API на функціонування та безпеку конкретної підсистеми.

ВИСНОВКИ

У дипломній роботі було досліджено особливості розробки та використання безпечних API в сучасному інформаційному суспільстві. Пояснювальна записка складається з анотацій, вступу, трьох розділів, висновків та списку використаної літератури.

У першому розділі визначено ключову роль API в сучасній технологічній екосистемі та його значення для веб-розробки та програмування. API, як інтерфейс програмування застосунків, розглянуто як засіб стандартизації взаємодії програм, забезпечення доступу до функціоналу системи та спрощення розробки програм.

Другий розділ присвячено аналізу технологій розробки безпечних API, зокрема протоколів та стандартів безпеки. Розглянуті ключові аспекти, спрямовані на забезпечення надійності та конфіденційності API, що дозволяють розробникам створювати безпечні інтерфейси програмування додатків.

У третьому розділі впроваджено безпечний API для підсистеми електронного документообігу. Розглянуті практичні аспекти розробки та інтеграції API з урахуванням сучасних стандартів безпеки та програмувальних практик. Розділ присвячено процесу розробки безпечного API з метою забезпечення високого рівня захисту інформації та конфіденційності у системі електронного документообігу.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Adya, A., Howell, J., Theimer, M., Bolosky, W. J., & Douceur, J. R. Cooperative Task Management without Manual Stack Management. ACM Transactions on Computer Systems (TOCS), 20(3), 2002. – 289-327 p.
2. Benkler, Y. The Wealth of Networks: How Social Production Transforms Markets and Freedom. Yale University Press. 2006.
3. Best Practices to prevent SQL-injections [Електронний ресурс]. - Режим доступу URL: <https://tableplus.io/blog/2018/08/best-practices-to-prevent-sqlinjection-attacks.html>
4. Cantor S., Kemp J., Philpott R. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard. March 2005.
5. Chin T.H., T. Ching-Kit. Systems of monitoring and auditing in information technologies. 2010
6. Chou, T. Programming Interviews Exposed: Coding Your Way Through the Interview. 2019 – 189-204 p.
7. Cloud Web Application Firewall [Електронний ресурс]. - Режим доступу URL: <https://www.cloudflare.com/waf/>
8. DeSantis, N., & Girdley, J. Practical API Architecture and Development with Azure and AWS. Apress. 2019.
9. Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures. 2000 – 76-89 p.
10. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley. 2004.
11. Gourley D., Totty B. HTTP: The Definitive Guide. O'Reilly Media, 2002.
12. Hardt D. The OAuth 2.0 Authorization Framework. Internet Engineering Task Force (IETF). October 2012.
13. Hunt, A., & Thomas, D. The Pragmatic Programmer: Your Journey to Mastery. Addison-Wesley. 1999.
14. Bloch, J. How to Design a Good API and Why It Matters." Google. 2006.

15. Jacobson, I., Christerson, M., Jonsson, P., & Övergaard, G. Object-Oriented Software Engineering: A Use Case Driven Approach. 1992.
16. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). Internet Engineering Task Force (IETF). December 2015.
17. Kavis, M. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS). 2014.
18. Martin, R. C. Agile Software Development: Principles, Patterns, and Practices. Pearson Education. 2003.
19. McConnell, S. Code Complete: A Practical Handbook of Software Construction. Microsoft Press. 2004.
20. Oram, A., & Wilson, G. Beautiful Code: Leading Programmers Explain How They Think. O'Reilly Media. 2007.
21. O'Reilly, T. Web 2.0: Principles and Best Practices. 2007 – 45-62 p.
22. O'Reilly, T. What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. 2007.
23. Rescorla E. SSL and TLS: Designing and Building Secure Systems. Addison-Wesley. 2000.
24. Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Engineering Task Force (IETF). 2018.
25. Resig, J., & Bibeault, B. Secrets of the JavaScript Ninja. Manning Publications. 2011.
26. Richardson, L., & Amundsen, M. RESTful Web APIs: Services for a Changing World. 2013 – 45-62 p.
27. Riehle, D. The Economic Motivation of Open Source Software: Stakeholder Perspectives. IEEE Computer Society. 2007.
28. Ristic I. Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications. O'Reilly Media, 2014.
29. Ruby, S., Thomas, D., & Hansson, D. H. Agile Web Development with Rails. O'Reilly Media. 2004.

30. Sakimura N., Bradley J., Jones M. OpenID Connect Core 1.0. OpenID Foundation. November 2014.
31. SOAP Version 1.2 Part 0: Primer (Second Edition). URL: <https://www.w3.org/TR/soap12-part0/>
32. SQL Injections Top Attack Statistics [Электронный ресурс]. - Режим доступа URL: <https://www.darkreading.com/risk/sql-injections-top-attack-statistics/d/did/1132988>
33. Stephen Norris, Ben Shaw. Monitoring and management of events in information systems. 2013.
34. Trowbridge, D. APIs: A Strategy Guide." O'Reilly Media. 2014.

ДОДАТКИ

Літинг коду проекту

```

using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using System.Data.SqlClient;
using System.Data;
using Newtonsoft.Json;

namespace API_DocFlow.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class Registration : ControllerBase
    {
        [HttpPost("registration")]

        public IActionResult registration([FromBody] RegistrationModel model)
        {
            AnswerRegistration answerRegistration = new AnswerRegistration();
            answerRegistration = IsValidModel(model);

            Logs logs = new Logs();
            string Request_ = JsonConvert.SerializeObject(model);
            string Answer = JsonConvert.SerializeObject(answerRegistration);
            string From = Request.Host.Value == null ? "" : Request.Host.Value;
            string To = "API_DocFlow";
            string Name = Request.Path.Value == null ? "" : Request.Path.Value;
            logs.WriteLogs(Request_, Answer, From, To, Name);

            if (answerRegistration.Status)
            {
                // Вхід успішний
                return Ok(new { answerRegistration.Status, answerRegistration.Description_, answerRegistration.Token, answerRegistration.PrivateKey });
            }
            else
            {
                // Вхід не удался
                return Unauthorized(new { answerRegistration.Status, answerRegistration.Description_, answerRegistration.Token, answerRegistration.PrivateKey });
            }
        }

        private AnswerRegistration IsValidModel(RegistrationModel model)
        {
            DataBase dataBase = new DataBase();

            AnswerRegistration answerRegistration = new AnswerRegistration();

            SqlDataAdapter adapter = new SqlDataAdapter();
            DataTable Table = new DataTable();
            int checkOrganization = CheckOrganization(model.EDRPOU);
            int checkUser = CheckUser(model.Username, model.EDRPOU);

            if (checkOrganization == 2 || checkUser == 2)
            {
                answerRegistration.Status = false;
                answerRegistration.Description_ = "No database connection!";
                answerRegistration.Token = "";
                answerRegistration.PrivateKey = "";
                return answerRegistration;
            }

            if (checkOrganization == 1)
            {
                answerRegistration.Status = false;
                answerRegistration.Description_ = "Error! Organization with this EDRPOU already exists!";
                answerRegistration.Token = "";
                answerRegistration.PrivateKey = "";
                return answerRegistration;
            }

            if (checkUser == 1)
            {
                answerRegistration.Status = false;
                answerRegistration.Description_ = "Error! User already exists!";
                answerRegistration.Token = "";
                answerRegistration.PrivateKey = "";
                return answerRegistration;
            }

            Token TokenClass = new Token();
            answerRegistration.Token = TokenClass.GenerateRandomString(15);

            string[] Keys = new string[2];
            CryptoClass CryptoClass = new CryptoClass();
            Keys = CryptoClass.GenerateRSAKeys();
            string OpenKey = Keys[0];
            answerRegistration.PrivateKey = Keys[1];

            string CommandWriteOrganization = $"insert into Organization(Name_, EDRPOU, Password_) values('{model.OrgName}', '{model.EDRPOU}', '{model.OrgPassword}')";
            string CommandWriteUser = $"insert into Users(Name_, EDRPOU, Password_, OpenKey, Token) values('{model.Username}', '{model.EDRPOU}', '{model.UserPassword}', '{OpenKey}', '{answerRegistration.Token}')";

            SqlCommand commandOrg = new SqlCommand(CommandWriteOrganization, dataBase.GetConnection());
            SqlCommand commandUser = new SqlCommand(CommandWriteUser, dataBase.GetConnection());
            dataBase.openConnection();
            if (commandOrg.ExecuteNonQuery() == 1 && commandUser.ExecuteNonQuery() == 1)
            {
                answerRegistration.Status = true;
                answerRegistration.Description_ = "Account created successfully!";
            }
            else
            {
                answerRegistration.Status = false;
                answerRegistration.Description_ = "Error! Account not created!";
                answerRegistration.Token = "";
                answerRegistration.PrivateKey = "";
            }

            return answerRegistration;
        }
    }

    public class RegistrationModel
    {
        public string Username { get; set; }
        public string UserPassword { get; set; }
        public string EDRPOU { get; set; }
        public string OrgName { get; set; }
        public string OrgPassword { get; set; }
    }

    public RegistrationModel(string username, string userPassword, string edrpou, string orgName, string orgPassword)

```

```

    {
        // Проверка на null и установка значений свойств
        Username = username ?? throw new ArgumentNullException(nameof(username));
        UserPassword = userPassword ?? throw new ArgumentNullException(nameof(userPassword));
        EDRPOU = edrpou ?? throw new ArgumentNullException(nameof(edrpou));
        OrgName = orgName ?? throw new ArgumentNullException(nameof(orgName));
        OrgPassword = orgPassword ?? throw new ArgumentNullException(nameof(orgPassword));
    }
}

public class AnswerRegistration
{
    public bool Status = false;
    public string Description_ = "";
    public string Token = "";
    public string PrivateKey = "";
}

public int CheckOrganization(string EDRPOU)
{
    DataBase dataBase = new DataBase();
    SqlDataAdapter adapter = new SqlDataAdapter();
    DataTable Table = new DataTable();
    string Request = $"select id_org, Name_, EDRPOU, Password_ from Organization where EDRPOU = '{EDRPOU}'";

    SqlCommand command = new SqlCommand(Request, dataBase.GetConnection());
    adapter.SelectCommand = command;

    try
    {
        adapter.Fill(Table);
    }
    catch
    {
        return 2; //Error database
    }
    if (Table.Rows.Count > 0)
    {
        return 1; //Error org
    }
    else
    {
        return 0;
    }
}

private int CheckUser(string Name_, string EDRPOU_)
{
    DataBase dataBase = new DataBase();
    SqlDataAdapter adapter = new SqlDataAdapter();
    DataTable Table = new DataTable();
    string Request = $"select id_user, Name_, EDRPOU, Password_, OpenKey from Users where EDRPOU = '{EDRPOU_}' and Name_ = '{Name_}'";

    SqlCommand command = new SqlCommand(Request, dataBase.GetConnection());
    adapter.SelectCommand = command;

    try
    {
        adapter.Fill(Table);
    }
    catch
    {
        return 2; //Error database
    }
    if (Table.Rows.Count > 0)
    {
        return 1; //ErrorUser
    }
    else
    {
        return 0;
    }
}

}
}

using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using System.Data.SqlClient;
using System.Data;
using Newtonsoft.Json;

namespace API_DocFlow.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class Login : ControllerBase
    {
        [HttpPost("login")]

        public IActionResult Login([FromBody] LoginModel model)
        {
            AnswerLogin AnswerLogin = new AnswerLogin();
            AnswerLogin = IsValidUser(model.Username, model.Password, model.EDRPOU, model.Token);

            Logs logs = new Logs();
            string Request_ = JsonConvert.SerializeObject(model);
            string Answer = JsonConvert.SerializeObject(AnswerLogin);
            string From = Request.Host.Value == null ? "" : Request.Host.Value;
            string To = "API_DocFlow";
            string Name = Request.Path.Value == null ? "" : Request.Path.Value;
            logs.WriteLogs(Request_, Answer, From, To, Name);

            if (AnswerLogin.Status)
            {
                // Вход успешен
                return Ok(new { AnswerLogin.Status, AnswerLogin.Description_, AnswerLogin.Token });
            }
            else
            {
                // Вход не удался
                return Unauthorized(new { AnswerLogin.Status, AnswerLogin.Description_, AnswerLogin.Token });
            }
        }

        private AnswerLogin IsValidUser(string username = "", string password = "", string EDRPOU = "", string Token = "")
        {
            DataBase dataBase = new DataBase();

            AnswerLogin AnswerLogin = new AnswerLogin();

            SqlDataAdapter adapter = new SqlDataAdapter();

```

```

        DataTable Table = new DataTable();
        string Request = $"select Users.id_user, Users.Name_ , Users.EDRPOU, Users.Password_ , Organization.Name_ as OrgName from Users Innerjoin Organization on Users.EDRPOU = Organization.EDRPOU where Users.EDRPOU = '{EDRPOU}' and Users.Name_ = '{username}' and Users.Password_ = '{password}' and Users.Token = '{Token}'";

        SqlCommand command = new SqlCommand(Request, dataBase.GetConnection());
        adapter.SelectCommand = command;

        try
        {
            adapter.Fill(Table);
        }
        catch
        {
            AnswerLogin.Status = false;
            AnswerLogin.Description_ = "No database connection!";
            AnswerLogin.Token = "";
        }
        if (Table.Rows.Count > 0)
        {
            Token TokenClass = new Token();
            AnswerLogin.Status = true;
            AnswerLogin.Description_ = Table.Rows[0].ItemArray[4].ToString();
            AnswerLogin.Token = TokenClass.GenerateRandomString(15);

            string CommandChangeToken = $"UPDATE Users SET Token = '{AnswerLogin.Token}' WHERE Name_ = '{username}' and EDRPOU = '{EDRPOU}' and Password_ = '{password}'";
            SqlCommand CommandChange = new SqlCommand(CommandChangeToken, dataBase.GetConnection());
            dataBase.openConnection();
            if (CommandChange.ExecuteNonQuery() != 1)
            {
                AnswerLogin.Status = false;
                AnswerLogin.Description_ = "No database connection!";
            }
        }
        else
        {
            AnswerLogin.Status = false;
            AnswerLogin.Description_ = "The user does not exist!";
            AnswerLogin.Token = "";
        }

        return AnswerLogin;
    }
}

public class LoginModel
{
    public string Username { get; set; }
    public string Password { get; set; }
    public string EDRPOU { get; set; }
    public string Token { get; set; }

    public LoginModel(string username, string password, string edrpou, string token)
    {
        // Проверка на null и установка значений свойств
        Username = username ?? throw new ArgumentNullException(nameof(username));
        Password = password ?? throw new ArgumentNullException(nameof(password));
        EDRPOU = edrpou ?? throw new ArgumentNullException(nameof(edrpou));
        Token = token ?? throw new ArgumentNullException(nameof(token));
    }
}

public class AnswerLogin
{
    public bool Status = false;
    public string Description_ = "";
    public string Token = "";
}
}

```