Київський національний торговельно-економічний університет

Кафедра комп'ютерних наук та інформаційних систем

ВИПУСКНИЙ КВАЛІФІКАЦІЙНИЙ ПРОЕКТ

на тему:

«Розробка API для авторизації користувачів для веб-ресурсів»

Студента 4 курсу, 9 групи,
спеціальності
122 «Комп'ютерні науки» Лукашевського
Євгенія Ігоровича

_____
підпис студента

Науковий керівник
Доктор технічних наук, професор Краскевич Валерій
Євгенович

_____
підпис керівника

Гарант освітньої програми
кандидат технічних наук, доцент Демідов Павло
Георгійович

_____
підпис керівника

Київ 2020

Київський національний торговельно-економічний університет

Факультет інформаційних технологій

Кафедра комп'ютерних наук та інформаційних систем

Спеціальність 122 «Комп'ютерні науки»

Затверджую

Зав. кафедри _____Пурський О.І.

«20» грудня 2019р.

Завдання
на випускний кваліфікаційний проект студенту

Лукашевському Євгенію Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема випускного кваліфікаційного проекту

«Розробка API авторизації користувачів для веб-ресурсів»

Затверджена наказом ректора від *«04» грудня 2019 р. № 4111*

2. Строк здачі студентом закінченої роботи *29 травня 2020 року*

3. Цільова установка та вихідні дані до роботи

Мета роботи: дослідження сучасних методів шифрування та захисту інформації, розробка єдиного мікросервісу(API) для авторизації користувачів.

Об'єкт дослідження: мікросервіси(API) та шифрування

Предмет дослідження: моделі, методи обробки та передачі даних між веб-ресурсами, методи шифрування та захисту даних.

4. Перелік графічного матеріалу_____

_____

_____

5. Консультанти по роботі із зазначенням розділів, за якими здійснюється консультування:

Розділ	Консультант (прізвище, ініціали)	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Краскевич В.Є.	15.12.2019 р.	15.12.2019 р.
2	Краскевич В.Є.	15.12.2019 р.	15.12.2019 р.
3	Краскевич В.Є.	15.12.2019 р.	15.12.2019 р.

6. Зміст випускного кваліфікаційного проекту (перелік питань за кожним розділом)

ВСТУП

РОЗДІЛ 1. Теоретичні аспекти при розробці мікросервісної архітектури

1.1. Теоретичний аналіз поняття “мікросервіс”.

1.2. Сучасні методи шифрування та їх варіації.

РОЗДІЛ 2. Огляд методик для розробки мікросервісів та шифрування.

2.1. Методологія REST API.

2.2. Принцип SOLID.

2.3. Cookie-файли.

2.4. Цифрові підписи.

2.5. Симетричне шифрування даних.

2.6. Асиметричне шифрування.

РОЗДІЛ 3. Створення захищеного API авторизації користувачів для веб-ресурсів

3.1. Розробка базової структури мікросервісу за принципами SOLID на Node.js

3.2. Імплементація роутів REST методології.

3.3. Настройка генерації та видачі токенів з цифровим підписом, згенерованим симетричним шифруванням.

ВИСНОВКИ

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

7. Календарний план виконання роботи

№ Пор.	Назва етапів випускної кваліфікаційної роботи	Строк виконання етапів роботи	
		За планом	фактично
1	2	3	4
1	*Вибір теми випускного кваліфікаційного проекту*	*01.10.2019*	*01.10.2019*
2	*Розробка та затвердження завдання на випускну кваліфікаційну роботу*	*15.12.2019*	*15.12.2019*
3	*Вступ*	*03.02.2020*	
4	*РОЗДІЛ 1. Теоретичні аспекти при розробці мікросервісної архітектури*	*28.02.2020*	
5	*РОЗДІЛ 2. Огляд методик для розробки мікросервісів та шифрування*	*06.04.2020*	
6	*РОЗДІЛ 3. Створення захищеного API для веб-ресурсів*	*12.05.2020*	
7	*Висновки*	*15.05.2020*	
8	*Здача випускної кваліфікаційної роботи на кафедру науковому керівнику*	*20.05.2020*	
9	*Попередній захист випускної кваліфікаційної роботи*	*03.06.2020*	
11	*Виправлення зауважень, зовнішнє рецензування випускної кваліфікаційної*	*09.06.2020*	

	роботи		
12	*Представлення готової зшитої випускної кваліфікаційної роботи на кафедру*	*12.06.2020*	
13	*Публічний захист випускної кваліфікаційної роботи*	*За розкладом роботи ЕК*	

8. Дата видачі завдання «15» грудня 2019 р.

9. Керівник випускної кваліфікаційної роботи (проекту)

Краскевич В.Є.

(прізвище, ініціали, підпис)

10. Гарант освітньої програми Демідов П.Г.

(прізвище, ініціали, підпис)

11. Завдання прийняв до виконання студент-дипломник

Лукашевський Є.І.

(прізвище, ініціали, підпис)

12. Відгук керівника випускного кваліфікаційного проекту

Керівник випускної кваліфікаційного проекту

_________________ ____________.

(підпис, дата)

13. Висновок про випускний кваліфікаційний проект

Випускний кваліфікаційний проект студента _____

(прізвище, ініціали)

може бути допущена до захисту в екзаменаційній комісії.

Гарант освітньої програми_____Демідов П.Г.____

(підпис, прізвище, ініціали)

Завідувач кафедри _____Пурський О.І.____

(підпис, прізвище, ініціали)

«_____»_____2020 р.

Анотація

У випускному кваліфікаційному проекті здійснено розробку API для авторизації користувачів для веб-ресурсів. За допомогою методу аналізу були досліджені сучасні технології, методи розробки API та шифрування за останні 5 років. Методом порівняльного аналізу було визначено оптимальні шляхи для реалізації проекту. Створена серверна частина API, імплементована робота бази даних, розроблені захищені маршрути для авторизованих користувачів, для входу і виходу з системи та реєстрації. Також, наглядним чином протестовано роботу API.

Ключові слова: API, мікросервіс, серверна частина, авторизація, шифрування.

Annotation

In the final qualification project, a development of the user authorization API was done. Using the analysis method, modern technologies and methods for API development and encryption for last 5 years were studied. Optimal ways for realization of project were studied by comparison analysis method. Server-side API, database structure, user login, logout, registration and protected routes were created and tested, as sequence – direct flow of user interactions were shown.

Keywords: API, microservice, server-side, authorization, encryption.

ВСТУП

Актуальність роботи полягає у створенні API для реєстрації користувачів, яке дасть дасть змогу створити надійну авторизацію для користувачів, та надійно захистить дані користувача на веб-ресурсі. Код якого можна використовувати повторно, і воно буде знаходитись у відкритому доступі для інших розробників що значно зменшить затрати часу на розробку серверної частини багатьох проектів. Актуальність даного додатку зумовлена нещодавніми хакерськими атаками у мережі інтернет.

Мета роботи полягає у створенні шаблону мікросервісу з вбудованою авторизацією користувачів і у дослідженні передових методів шифрування, та їх комбінування для досягнення найкращого результату з точки зору захисту та масштабування, що забезпечить надійність використання веб-ресурсу і захист від хакерських атак.

Предмет дослідження — принципи побудови сучасних API на мікросервісній архітектурі і методи шифрування, та захисту процесу аутентифікації та авторизації.

Методи дослідження ґрунтуються на вивченні предметної області шифрування методом порівняння і сучасних методів побудови API за допомогою системного аналізу.

Практична цінність полягає створенні додатку, що забезпечить надійну авторизацію та аутентифікацію користувачів у мережі інтернет, та який можна буде використовувати як основу для нових проектів, чи підключати до вже існуючих веб-ресурсів для захисту їх користувачів.

ЗМІСТ

РОЗДІЛ 1
Теоретичні аспекти при розробці мікросервісної архітектури

Мікросервіси - також відомі як мікросервісна архітектура - це архітектурний стиль, який структурує додаток як сукупність служб, які є:

- Високорентабельний і перевірений
- Нещільно з'єднані
- Незалежно розгортається
- Організовано навколо можливостей бізнесу
- Володіє невеликою командою

Мікросервісна архітектура дозволяє швидко часто та надійно розробляти великі, складні програми. Це також дозволяє організації розвивати свій набір технологій.

Мікросервіси слід розгядати як самодостатню частину функціоналу бізнесу з чіткими інтерфейсами, і може, завдяки власним внутрішнім компонентам, реалізувати шаровану архітектуру додатку. З точки зору стратегії, мікросервісна архітектура, по суті, відповідає філософії Unix "Роби одну справу і роби це добре". Мікросервісній архітектурі характерні такі властивості:

1. Піддається безперервному процесу розробки програмного забезпечення для деплойменту. Зміна невеликої частини додатку вимагає лише перебудови та перерозподілу лише однієї або невеликої кількості сервісів.
2. Дотримується таких принципів, як дрібні інтерфейси (для незалежно розгортуваних служб), бізнес-орієнтована розробка.

Переваги розкладання програми на різні менші сервіси:

- Модульність: Це полегшує програму зрозуміти, розвинути, протестувати, та стати більш стійкою до ерозії архітектури. Ця вигода часто аргументується порівняно зі складністю монолітних архітектур.
- Масштабованість: Оскільки мікросервіси реалізуються та розгортаються

незалежно один від одного, тобто вони працюють в незалежних процесах, їх можна контролювати і масштабувати незалежно.

- Інтеграція різнорідних та застарілих систем: мікросервіси розглядаються як життєздатний засіб модернізації існуючого монолітного програмного забезпечення. Процес модернізації програмного забезпечення застарілих додатків здійснюється за допомогою інкрементального підходу.
- Розподілений розвиток: він паралелізує розвиток, дозволяючи невеликим автономним командам самостійно розробляти, розгортати та масштабувати свої служби. Це також дозволяє архітектурі окремої послуги формуватися за допомогою постійного рефакторингу. Архітектури на основі мікросервісу сприяють постійній інтеграції, постійній доставці та розгортанню.

1.1. Теоретичний аналіз поняття “мікросервіс”.

Комп'ютерні мікросервіси можуть бути реалізовані на різних мовах програмування та можуть використовувати різні інфраструктури. Тому найважливішим вибором технології є спосіб спілкування мікросервісів один з одним (синхронна, асинхронна, інтеграція інтерфейсу) та протоколи, що використовуються для зв'язку (RESTful HTTP, обмін повідомленнями, GraphQL і т.д.). У традиційній системі більшість варіантів технологій, як мова програмування, впливають на всю систему. Тому підхід до вибору технологій зовсім інший.

Сервісна сітка

У сервісній сітці кожен службовий екземпляр поєднується з екземпляром зворотного проксі-сервера, який називається службовим проксі, проксі-провідником або коляскою. Службовий екземпляр та проксі-сервер спільного використання є контейнером, а контейнерами керує інструмент для оркестрування контейнерів, наприклад Kubernetes, Nomad, Docker Swarm або DC / OS. Проксі-сервери відповідають за зв'язок з іншими примірниками сервісу і можуть

підтримувати такі можливості, як пошук (екземпляр) служби, вирівнювання завантаження, аутентифікація та авторизація, захищена комунікація та інші.

Для службової сітки характерно, що службові екземпляри та їхні проксі-сервери складають площину даних, яка включає не тільки управління даними, але й запит на обробку та відповідь. Сервісна сітка також включає площину управління для управління взаємодією між службами, опосередковану їхніми проксі-серверами. Існує кілька варіантів архітектури службової сітки: Istio (спільний проект між Google, IBM та Lyft), Linkerd (проект CNCF під керівництвом Buoyant), Consul (продукт HashiCorp)) та багато інших у службовій сітці Landscape. Площина управління службовою сіткою Meshery, забезпечує життєвий цикл, конфігурацію та управління продуктивністю в межах розгортання службової сітки.

Порівняння платформ

Реалізувати мікросервісну архітектуру дуже складно. Netflix розробив мікросервісну стуктуру для підтримки своїх внутрішніх додатків, а потім відкрив багато частин цього фреймворку. Багато з цих інструментів були популяризовані у рамках Spring Framework - вони були повторно впроваджені як інструменти, на основі Spring в рамках проекту Spring Cloud. Одним із важливих аспектів екосистеми Spring Cloud є те, що всі вони базуються на Java-технологіях, тоді як Kubernetes є платформою для виконання поліглоту.

1.2. Поняття сучасних принципів програмування при розробці мікросевісів.

Розробляючи серверну програму, ви можете запустити її з модульної шестикутної або багатошарової архітектури, що складається з різних типів компонентів:

- Презентація - відповідає за обробку HTTP-запитів та відповідь за допомогою HTML або JSON / XML (для API веб-служб).
- Бізнес-логіка - бізнес-логіка програми.

- Доступ до бази даних - об'єкти доступу до даних, відповідальні за доступ до бази даних.
- Інтеграція додатків - інтеграція з іншими службами (наприклад, через обмін повідомленнями або REST API).

Незважаючи на логічно модульну архітектуру, додаток упаковується та розгортається як моноліт.

Переваги монолітної архітектури

- Простий у розвитку.
- Простий для тестування. Наприклад, ви можете впроваджувати тестування, щоб просто запустити додаток і протестувати інтерфейс користувача з Selenium.
- Простий у розгортанні. Вам просто потрібно скопіювати упаковану програму на сервер.
- Легко масштабувати горизонтально, виконавши кілька копій за балансиром навантаження.
- На ранніх стадіях проекту він працює добре, і в основному більшість великих і успішних проектів, які існують сьогодні, були започатковані як моноліт.

Недоліки монолітної архітектури

- Цей простий підхід має обмеження в розмірах і складності.
- Додаток занадто великий і складний, щоб повністю зрозуміти і вносити зміни швидко і правильно.
- Розмір програми може сповільнити час запуску.
- Ви повинні повторно розмістити всю програму під час кожного оновлення.
- Вплив змін зазвичай не дуже добре зрозумілий, що призводить до проведення широкого ручного тестування.
- Постійне розгортання важко.
- Монолітні програми також можуть бути важкими для масштабування, коли

різні модулі мають суперечливі вимоги до ресурсів.

- Ще одна проблема монолітних застосувань - надійність. Помилка в будь-якому модулі (наприклад, витік пам'яті) може потенційно знизити весь процес. Більше того, оскільки всі екземпляри програми однакові, ця помилка вплине на доступність всієї програми.
- Монолітні програми мають бар'єр для впровадження нових технологій. Оскільки зміни в рамках або мовах вплинуть на всю програму, це надзвичайно дорого і за часом, і за вартістю.

Недоліки архітектури мікросервісів

- Архітектура мікросервісів додає складності проекту лише тим, що додаток мікросервісів - це розподілена система. Потрібно вибрати та впровадити механізм міжпроцесорного зв’язку на основі повідомлень чи RPC та записати код для усунення часткових збоїв та врахування інших помилок розподілених обчислень.
- Мікросервіси мають архітектуру бази даних, що розділяється. Бізнес-транзакції, що оновлюють кілька суб’єктів господарювання у додатку на основі мікросервісів, потребують оновлення декількох баз даних, що належать різним службам. Використання розподілених транзакцій, як правило, не є варіантом, і вам доведеться використовувати підхід, орієнтований на послідовність, який є більш складним для розробників.
- Тестування мікросевісів також набагато складніше, ніж у випадку з монолітним веб-додатком. Для аналогічного тестування для служби вам потрібно буде запустити цю службу та будь-які служби, від яких вона залежить (або принаймні налаштувати заглушки для цих служб).
- Складніше впровадити зміни, що охоплюють кілька служб. У монолітній програмі ви можете просто змінити відповідні модулі, інтегрувати зміни та розгорнути їх за один раз. В архітектурі мікросервісів потрібно ретельно спланувати та скоординувати розроблення змін до кожної з служб.
- Розгортання програми на базі мікросервісів також є складнішою.

Монолітний додаток просто розгортається на наборі однакових серверів за балансиром навантаження. На відміну від цього, програма мікросервісу зазвичай складається з великої кількості послуг. Кожна служба матиме декілька екземплярів виконання. І кожен екземпляр потрібно налаштовувати, розгортати, масштабувати та контролювати. Крім того, вам також буде необхідно впровадити механізм виявлення служби. Мануальні підходи до операцій не можуть довести до цього рівня складності, а для успішного розгортання програми мікросервісів вимагає високого рівня автоматизації.

1.3. Сучасні методи шифрування та їх варіації.

Шифрування - оборотне перетворення інформації з метою приховування від неавторизованих осіб, з наданням, в цей же час, авторизованим користувачам доступу до неї. Головним чином, шифрування відповідає за дотримання конфіденційності інформації, що передається. Важливою особливістю будь-якого алгоритму шифрування є використання ключа, який стверджує вибір конкретного перетворення з сукупності можливих для даного алгоритму.

Користувачі є авторизованими, якщо вони мають певний аутентичний ключ. Вся складність і, власне, завдання шифрування полягає в тому, як саме реалізований цей процес.

В цілому, шифрування складається з двох складових - зашифровування і розшифрування.

За допомогою шифрування забезпечуються три стану безпеки інформації:

1. Конфіденційність.

   Шифрування використовується для приховування інформації від неавторизованих користувачів при передачі або при зберіганні.

2. Цілісність.

   Шифрування використовується для запобігання зміни інформації при

передачі або зберіганні.

3. Ідентифікованість.

   Шифрування використовується для аутентифікації джерела інформації та запобігання відмови відправника інформації від того факту, що дані були відправлені саме їм.

Для того, щоб прочитати зашифровану інформацію, приймаючій стороні необхідні ключ і дешифратор (пристрій, що реалізує алгоритм розшифрування). Ідея шифрування полягає в тому, що зловмисник, перехопивши зашифровані дані і не маючи до них ключа, не може ні прочитати, ні змінити передану інформацію. Крім того, в сучасних криптосистемах (з відкритим ключем) для шифрування, розшифрування даних можуть використовуватися різні ключі. Однак, з розвитком криптоаналізу, з'явилися методики, що дозволяють дешифрувати закритий текст без ключа. Вони засновані на математичному аналізі переданих даних.

Шифрування застосовується для зберігання важливої інформації в ненадійних джерелах і передачі її по незахищеним каналам зв'язку. Така передача даних представляє із себе два взаємно зворотних процесу:

1. Перед відправленням даних по лінії зв'язку або перед приміщенням на зберігання вони піддаються зашифровування.
2. Для відновлення вихідних даних із зашифрованих до них застосовується процедура розшифрування.

   Шифрування спочатку використовувалося тільки для передачі конфіденційної інформації. Однак згодом шифрувати інформацію почали з метою її зберігання в ненадійних джерелах. Шифрування інформації з метою її зберігання застосовується і зараз, це дозволяє уникнути необхідності в фізично захищеному сховище.

Шифром називається пара алгоритмів, що реалізують кожне із зазначених перетворень. Ці алгоритми застосовуються до даних з використанням ключа. Ключі для шифрування і для розшифрування можуть відрізнятися, а можуть бути однаковими. Секретність другого (що розшифровується) з них робить дані

недоступними для несанкціонованого ознайомлення, а таємність першого (шифрувального) унеможливлює внесення неправдивих даних. У перших методах шифрування використовувалися однакові ключі, однак в 1976 році були відкриті алгоритми із застосуванням різних ключів. Збереження цих ключів в секретності і правильний їх поділ між адресатами є дуже важливим завданням з точки зору збереження конфіденційності інформації, що передається. Це завдання досліджується в теорії управління ключами (в деяких джерелах вона згадується як поділ секрету).

На даний момент існує величезна кількість методів шифрування. Головним чином ці методи діляться, в залежності від структури використовуваних ключів, на симетричні методи і асиметричні методи. Крім того, методи шифрування можуть мати різну криптостійкость і по-різному обробляти вхідні дані - блокові шифри і потокові шифри.

Методи шифрування:

- Симетричне шифрування використовує один і той же ключ і для зашифровування, і для розшифрування.
- Асиметричне шифрування використовує два різних ключі: один для зашифровування (який також називається відкритим), інший для розшифрування (називається закритим).

Ці методи вирішують певні завдання і мають як переваги, так і недоліки. Конкретний вибір застосовуваного методу залежить від цілей, з якими інформація піддається шифруванню.

Симетричне шифрування

У симетричних криптосистемах для шифрування і розшифрування використовується один і той же ключ. Алгоритм і ключ вибирається заздалегідь і відомий обом сторонам. Збереження ключа в секретності є важливим завданням для встановлення і підтримки захищеного каналу зв'язку. У зв'язку з цим, виникає проблема початкової передачі ключа (синхронізації ключів). Крім того існують методи кріптоатак, що дозволяють так чи інакше дешифрувати інформацію не

маючи ключа або ж за допомогою його перехоплення на етапі узгодження. В цілому ці моменти є проблемою криптостійкості конкретного алгоритму шифрування і є аргументом при виборі конкретного алгоритму.

Симетричні, а конкретніше, алфавітні алгоритми шифрування були одними з перших алгоритмів. Пізніше було винайдено асиметричне шифрування, в якому ключі у співрозмовників різні.

Схема реалізації

Розгляну на прикладі. Є два користувача - А і Б, вони хочуть обмінюватися конфіденційною інформацією.

1. Генерація ключа.

Користувач Б (або А) вибирає ключ шифрування *d* і алгоритми *E, D* (функції шифрування і розшифрування), потім посилає цю інформацію користувачу А (або Б).

2. Шифрування і передача повідомлення.

Користувач А шифрує повідомлення *m* з використанням отриманого ключа *d.*

E (m, d) = c

І передає користувачу Б, отриманий шифротекст *c*. Те ж саме робить користувач Б, якщо хоче відправити користувачеві А повідомлення.

3. Розшифрування повідомлення.

Користувач Б, за допомогою того ж ключа *d*, розшифровує шифротекст *c*.

D (c, d) = *m*

Недоліками симетричного шифрування є проблема передачі ключа співрозмовнику і неможливість встановити справжність або авторство тексту. Тому, наприклад, в основі технології цифрового підпису лежать асиметричні схеми.

Асиметричне шифрування (з відкритим ключем)

У системах з відкритим ключем використовуються два ключі - відкритий і закритий, пов'язані певним математичним чином один з одним. Відкритий ключ передається по відкритому (тобто незахищеному, доступному для спостереження)

каналу і використовується для шифрування повідомлення і для перевірки електронного цифрового підпису. Для розшифровки повідомлення і для генерації електронного цифрового підпису використовується таємний ключ.

Дана схема вирішує проблему симетричних схем, пов'язану з початкової передачею ключа іншій стороні. Якщо в симетричних схемах зловмисник перехопить ключ, то він зможе як слухати, так і вносити правки в передану інформацію. В асиметричних системах іншій стороні передається відкритий ключ, який дозволяє шифрувати, але не розшифровувати інформацію. Таким чином вирішується проблема симетричних систем, пов'язана з синхронізацією ключів.

Схема реалізації

Розгляну на прикладі. Є два користувача - А і Б, користувач А хоче передавати користувачеві Б конфіденційну інформацію.

1. Генерація пари ключів.

Користувач Б вибирає алгоритм *(E, D)* і пару ключів: відкритий і закритий ключі - *(e, d)* і посилає відкритий ключ *e* користувачу А по відкритому каналу.

2. Шифрування і передача повідомлення.

Користувач А шифрує інформацію з використанням відкритого ключа користувача Б *e*.

$E(m, e) = c$

І передає користувачу Б отриманий шифротекст *c*.

3. Розшифрування повідомлення.

Користувач Б, за допомогою закритого ключа *d*, розшифровує шифротекст *c*.

$D(c, d) = m$

Якщо необхідно налагодити канал зв'язку в обидві сторони, то перші дві операції необхідно виконати на обох сторонах, таким чином, кожен буде знати свої закритий, відкритий ключі та відкритий ключ співрозмовника. Закритий ключ кожної сторони не передається по незахищеному каналу, тим самим залишаючись в секретності.

РОЗДІЛ 2
ОГЛЯД МЕТОДИК ДЛЯ РОЗРОБКИ МІКРОСЕРВІСІВ ТА ШИФРУВАННЯ

2.1. Методологія REST API.

REST(англ. - REpresentational State Transfer) - це архітектурний стиль для розподілених гіпермедіа систем і вперше був представлений Роєм Філдінгом у 2000 році у своїй знаменитій дисертації.

Принципи REST:

1. Клієнт-сервер - відокремлюючи проблеми клієнтського інтерфейсу від проблем зберігання даних, ми покращуємо портативність призначеного для користувача інтерфейсу на кількох платформах та покращуємо масштабованість, спрощуючи серверні компоненти.
2. Без стану - кожен запит від клієнта до сервера повинен містити всю інформацію, необхідну для розуміння запиту, і не може скористатися будь-яким збереженим контекстом на сервері. Тому стан сесії повністю зберігається на клієнті.
3. Кешування - обмеження кешу вимагають, щоб дані у відповіді на запит неявно або явно позначали як кешовані або не кешовані. Якщо відповідь є кешованою, то кешу клієнта надається право повторно використовувати ці дані відповіді для наступних, рівнозначних запитів.
4. Уніфікований інтерфейс - за допомогою застосування принципу загальної інженерії програмного забезпечення до компонентного інтерфейсу спрощується загальна архітектура системи та покращується видимість взаємодій. Щоб отримати єдиний інтерфейс, для керування поведінкою компонентів потрібно кілька архітектурних обмежень. REST визначається чотирма обмеженнями інтерфейсу: ідентифікація ресурсів;
5. Багатошарова система - цей стиль системи дозволяє архітектурі складатися з

ієрархічних шарів, обмежуючи поведінку компонентів таким чином, що кожен компонент не може "бачити" за межами прямого шару, з яким він взаємодіє.

6. Код на вимогу (необов’язково) - REST дозволяє розширити функціональність клієнта, завантаживши та виконавши код у вигляді аплетів чи сценаріїв.

Ключовою абстракцією інформації в REST є ресурс. Будь-яка інформація, яку можна назвати, може бути ресурсом: документ чи зображення, тимчасова послуга, колекція інших ресурсів, невіртуальний об’єкт (наприклад, людина) тощо. REST використовує ідентифікатор ресурсу для ідентифікації конкретного ресурсу, що бере участь у взаємодії між компонентами.

Стан ресурсу в будь-яку конкретну часову позначку називається представленням ресурсу. Представлення складається з даних, метаданих, що описують посилання на дані та гіпермедіа, які можуть допомогти клієнтам перейти до наступного бажаного стану.

Формат даних представлення відомий як тип носія. Тип носія ідентифікує специфікацію, яка визначає, як представлення має оброблятися. Справді RESTful API схожий на гіпертекст. Кожна адресована одиниця інформації містить адресу або явно (наприклад, атрибути посилання та ідентифікатора), або неявно (наприклад, походить від визначення типу носія та структури представлення носія).

Гіпертекст (або гіпермедіа) означає одночасне представлення інформації та керування таким чином, що інформація стає доступною, завдяки якій користувач (або автомат) отримує вибір та вибирає дії. Пам'ятайте, що гіпертекст не повинен бути в форматі HTML (XML, або JSON) у браузері. Машини можуть переходити посилання, коли вони розуміють формат даних та типи зв’язків.

Кожен тип носія визначає модель обробки за замовчуванням. Наприклад, HTML визначає процес візуалізації гіпертексту та поведінку браузера навколо кожного елемента. Він не має відношення до ресурсних методів GET / PUT / POST /

DELETE, крім того, що деякі елементи медіа-типу визначають модель процесу, яка нагадує «елементи прив'язки з атрибутом href створювати гіпертекстовий зв'язок, який при виборі, викликає запит на пошук (GET) на URI, що відповідає атрибуту href, кодованому CDATA.

Ще одна важлива річ, пов'язана з REST - це ресурсні методи, які використовуються для виконання бажаного переходу. Велика кількість людей неправильно ставляться до ресурсних методів до методів HTTP GET / PUT / POST / DELETE.

Якщо ви вирішите, що HTTP POST буде використовуватися для оновлення ресурсу — всупереч тому, що більшість людей рекомендує HTTP PUT - це добре, і інтерфейс програми буде RESTful.

В ідеалі все, що потрібно для зміни стану ресурсу, має бути частиною відповіді API для цього ресурсу - включаючи методи та в якому стані вони залишать представлення.

Інша річ, яка допоможе вам під час створення API RESTful, полягає в тому, що результати API на основі запитів повинні бути представлені списком посилань із підсумковою інформацією, а не масивами оригінальних представлень ресурсів, оскільки запит не є заміною для ідентифікації ресурсів.

2.2. Принцип SOLID.

SOLID (скор. Від англ. Single responsibility, open-closed, Liskov substitution, interface segregation і dependency inversion) в програмуванні - мнемонічний акронім, введений Майклом Фезерсом (Michael Feathers) для перших п'яти принципів, названих Робертом Мартіном на початку 2000-х, які означали 5 основних принципів об'єктно-орієнтованого програмування і проектування.

При створенні програмних систем, використання принципів SOLID сприяє створенню такої системи, яку буде легко підтримувати і розширювати протягом довгого часу. Принципи SOLID - це методичні рекомендації, які також можуть

застосовуватися під час роботи над існуючим програмним забезпеченням, для його поліпшення.

Ініціал	Представлення	Назва, поняття
S	SRP	Принцип єдиної відповідальності Клас повинен бути відповідальний лише за щось одне. Якщо клас відповідає за вирішення декількох завдань, його підсистеми, що реалізують рішення цих задач, виявляються пов'язаними один з одним. Зміни в одній такій підсистемі ведуть до змін в іншій.
O	OCP	Принцип відкритості/закритості (*The Open Closed Principle*) «Програмні сутності повинні бути відкриті для розширення, але закриті для модифікації».
L	LSP	Принцип подстановки Барбары Лисков (*The Liskov Substitution Principle*) Мета цього принципу полягають в тому, щоб класи-спадкоємці могли б використовуватися замість батьківських класів, від яких вони утворені, не порушуючи роботу програми. Якщо виявляється, що в коді перевіряється тип класу, значить принцип підстановки порушується.
I	ISP	Принцип розділення інтерфейсу(The Interface

		Segregation Principle) Створюйте вузькоспеціалізовані інтерфейси, призначені для конкретного клієнта. Клієнти не повинні залежати від інтерфейсів, які вони не використовують.
D	DIP	Принцип інверсії залежностей (The Dependency Inversion Principle) Об'єктом залежності повинна бути абстракція, а не щось конкретне. 1. Модулі верхніх рівнів не повинні залежати від модулів нижніх рівнів. Обидва типи модулів повинні залежати від абстракцій. 2. Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.

Таб. 1.1

2.3. Cookie-файли

Cookie - невеликий фрагмент даних, відправлений веб-сервером і зберігається на комп'ютері користувача. Веб-клієнт (зазвичай веб-браузер) кожен раз при спробі відкрити сторінку відповідного сайту, пересилає цей фрагмент даних веб-сервера в складі HTTP-запиту. Застосовується для збереження даних на стороні користувача, на практиці зазвичай використовується для :

- аутентифікації користувача;
- зберігання персональних переваг та установки можуть бути;
- відстеження стану сеансу доступу користувача;

- відомості статистики про користувачів.

Підтримки браузерами cookie (отримання, збереження і подальша пересилання сервера збережених cookie) вимагають багато сайтів з обмеженнями доступу, більшість інтернет-магазинів. Налаштування оформлення та поведінки багатьох веб-сайтів по індивідуальних переваг користувача теж заснована на cookie.

Cookie легко перехопити і підмінити (наприклад, для отримання доступу до облікового запису), якщо користувач використовує нешифрований з'єднання з сервером. У групі ризику користувачі, що виходять в інтернет за допомогою публічних точок доступу Wi-Fi і не використовують при цьому таких механізмів, як SSL і TLS. Шифрування дозволяє також вирішити й інші проблеми, пов'язані з безпекою переданих даних.

Більшість сучасних браузерів дозволяє користувачам вибрати - приймати cookie чи ні, але їх відключення унеможливлює роботу з деякими сайтами. Крім того, за законами деяких країн (наприклад, за законом Євросоюзу від 2016 року, Загальний регламент щодо захисту даних) сайти повинні в обов'язковому порядку запитувати згоду перед установкою cookie.

Cookie використовуються веб-серверами для ідентифікації користувачів і зберігання даних про них.

Наприклад, якщо вхід на сайт здійснюється за допомогою cookie, то після введення користувачем своїх даних на сторінці входу, cookie дозволяють серверу запам'ятати, що користувач вже ідентифікований і йому дозволено доступ до відповідних послуг і операцій.

Багато сайтів також використовують cookie для збереження налаштувань користувача. Ці установки можуть використовуватися для персоналізації, яка включає в себе вибір оформлення та функціональності. Наприклад, Вікіпедія дозволяє авторизованим користувачам вибрати дизайн сайту. Пошукова система Google дозволяє користувачам (в тому числі і не зареєстрованим в ній) вибрати кількість результатів пошуку, що відображаються на одній сторінці.

Cookie також використовуються для відстеження дій користувачів на сайті. Як правило, це робиться з метою збору статистики, а рекламні компанії на основі такої статистики формують анонімні профілі користувачів для більш точного націлювання реклами.

З технічної точки зору, cookie є фрагментами даних, спочатку відправленними веб-сервером до браузеру. При кожному наступному відвідуванні сайту браузер пересилає їх назад сервера. Без cookie кожен перегляд веб-сторінки є ізольованим дією, не пов'язаним з переглядом інших сторінок того ж сайту, за допомогою ж cookie можна виявити зв'язок між переглядом різних сторінок. Крім відправки cookie веб-сервером, cookie можуть створюватися скриптами мовами на зразок JavaScript, якщо вони підтримуються і включені в браузері.

Специфікації вказують мінімальні обсяги, які повинні надаватися браузерами для зберігання cookie. Так, браузер повинен зберігати щонайменше 300 cookie по 4096 байт кожна, і щонайменше 20 cookie для одного сервера або домену.

Популярні браузери мають відповідний максимум зберігаються cookie для кожного домена:

- Internet Explorer 6 - 20
- Internet Explorer 7 - 20
- Opera 9 - 30
- Firefox 2.0 - 50
- Google Chrome 58.0 - 176
- Safari 10.0 - 242

На практиці, деякі браузери можуть накладати більш жорсткі обмеження. Наприклад, Internet Explorer надає 4096 байт для всіх cookie в одному домені.

Імена cookie нечутливі до регістру відповідно до розділу 3.1 RFC 2965.

Cookie можуть встановлювати дату їх видалення, в цьому випадку вони будуть автоматично видалені браузером в зазначений термін. Якщо дата видалення не вказана, cookie видаляються одразу, як тільки користувач закриє

браузер. Таким чином, зазначення дати закінчення дозволяє зберегти cookie більш ніж на один сеанс і такі cookie називаються постійними. Наприклад, інтернет-магазин може використовувати постійні cookie для зберігання кодів предметів, які користувач помістив в кошик - і навіть якщо користувач закриє браузер, не зробивши покупки, при подальшому вході йому не доведеться формувати кошик заново.

Зберігання cookie також може обмежуватися в залежності від веб-сервера, домену або піддомену, де вони були створені.

2.4. Цифрові підписи.

JSON Web Token (JWT) - це відкритий стандарт (RFC 7519) для створення токенів доступу, заснований на форматі JSON. Як правило, використовується для передачі даних для аутентифікації в клієнт-серверних додатках. Токени створюються сервером, підписуються секретним ключем і передаються клієнту, який в подальшому використовує даний токен для підтвердження своєї особи.

Токен JWT складається з трьох частин: заголовка (header), корисного навантаження (payload) і підписи або даних шифрування. Перші два елементи - це JSON об'єкти певної структури. Третій елемент обчислюється на підставі перших і залежить від обраного алгоритму (у випадку використання не підписаного JWT може бути опущений). Токени можуть бути перекодовані в компактне представлення (JWS / JWE Compact Serialization): до заголовку і корисного навантаження застосовується алгоритм кодування Base64-URL, після чого додається підпис і всі три елементи розділяються крапками («.»).

Наприклад, для заголовка і корисного навантаження, які виглядають наступним чином:

```
{
  "alg": "RS512",
  "typ": "JWT"
```

}

{

"sub": "12345",

"name": "John Doe",

"admin": true

}

Отримаємо наступне(умовне) компактне представлення:

eyJhbGciOiJkUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMyM0NSIsIm5hbWUiOiJKb2huIEdvbGQiLCJhZG1pbiI6dHJ1ZX0K.LIHjWCBORSWMEibq-tnT8ue_desqZx1K0XxCOXZRrBI

Заголовок

У заголовку вказується необхідна інформація для опису самого токена. Обов'язковий ключ тут тільки один:

alg: алгоритм, який використовується для підпису / шифрування (в разі не підписаного JWT використовується значення «none»).

Необов'язкові ключі:

typ: тип токена (type). Використовується в разі, коли маркери змішуються з іншими об'єктами, що мають JOSE заголовки. Повинно мати значення «JWT».

cty: тип вмісту (content type). Якщо в токені крім зареєстрованих службових ключів є призначені для користувача, то даний ключ не повинен бути присутнім. В іншому випадку повинно мати значення «JWT»

Корисне навантаження

У даній секції вказується інформація користувача (наприклад, ім'я користувача і рівень його доступу), а також можуть бути використані деякі службові ключі. Всі вони є необов'язковими:

- iss: чутливий до регістру рядок або URI, який є унікальним ідентифікатором сторони, генеруючої токен (issuer).
- sub: чутливий до регістру рядок або URI, який є унікальним ідентифікатором сторони, про яку міститься інформація в даному токені

(subject). Значення з цим ключем повинні бути унікальні в контексті сторони, що генерує JWT.

- aud: масив чутливих до регістру рядків або URI, який є списком одержувачів даного токена. Коли приймаюча сторона отримує JWT з даними ключем, вона повинна перевірити наявність себе в іменах одержувачів, інакше - проігнорувати токен (audience).
- exp: час в форматі Unix Time, що визначає момент, коли токен стане не дійсним (expiration).
- nbf: на противагу ключу exp, цей час в форматі Unix Time, що визначає момент, коли токен стане дійсним (not before).
- jti: рядок, що визначає унікальний ідентифікатор даного токена (JWT ID).
- iat: час в форматі Unix Time, що визначає момент, коли токен був створений. iat і nbf можуть не збігатися, наприклад, якщо токен був створений раніше, ніж час, коли він повинен стати дійсним.

Access і refresh токени

Access-токен - це токен, який надає доступ його власнику до захищених ресурсів сервера. Зазвичай він має короткий термін життя і може нести в собі додаткову інформацію, таку як IP-адреса сторони, запитуючої даний токен.
Refresh-токен - це токен, що дозволяє клієнтам запитувати нові access-токени після закінчення їх часу життя. Дані маркери зазвичай видаються на тривалий термін.

Схема роботи

Як правило, при використанні JSON-токенів в клієнт-серверних додатках реалізована наступна схема:
Клієнт проходить аутентифікацію в додатку (наприклад, з використанням логіна і пароля).
У разі успішної аутентифікації сервер відправляє клієнту access- і refresh-токени.
При подальшому зверненні до сервера клієнт використовує access-токен. Сервер перевіряє токен на валідність і надає клієнту доступ до ресурсів

У разі, якщо access-токен стає дійсним, клієнт відправляє refresh-токен, у відповідь на який сервер надає два оновлених токена.

У разі, якщо refresh-токен стає дійсним, клієнт знову повинен пройти процес аутентифікації.

JWT має ряд переваг над cookie:

- При використанні cookie, сервер повинен зберігати інформацію про видання в сесіях, в той час як використання JWT не вимагає зберігання додаткових даних про видані токенах: все, що повинен зробити сервер - це перевірити підпис.
- Сервер може не займатися створенням токенів, а надати це для зовнішніх сервісів.
- У JSON-токенах можна зберігати додаткову корисну інформацію про користувачів. Як наслідок - більш висока продуктивність. У разі з cookie, іноді необхідно здійснювати запити для отримання додаткової інформації. При використанні JWT ця інформація може бути передана в самому токені.
- JWT уможливлює надання одночасного доступу до різних доменів і сервісів.

2.5. Симетричне шифрування даних.

За допомогою симетричних алгоритмів шифрування дані перетворюються у форму, яку не може зрозуміти тому, хто не має секретного ключа, щоб розшифрувати його. Після того, як призначений одержувач, який володіє ключем, має повідомлення, алгоритм повертає свою дію так, що повідомлення повертається у початковий та зрозумілий вигляд. Секретний ключ, в якому і відправник, і одержувач можуть бути впевними може паролем / кодом, або це може бути випадковий рядок літер або цифр, що були створені захищеним генератором випадкових чисел (ГВЧ). Для шифрування банківського класу симетричні ключі повинні бути створені за допомогою ГВЧ, який сертифікований відповідно до галузевих стандартів, таких як FIPS 140-2.

Існує два типи алгоритмів симетричного шифрування:

1. Блокування алгоритмів. Встановлені довжини бітів шифруються в блоках електронних даних із застосуванням конкретного секретного ключа. Коли дані шифруються, система зберігає дані у своїй пам'яті під час очікування повних блоків.

2. Алгоритми потоку. Дані шифруються під час потоку, а не зберігаються в пам'яті системи.

Деякі приклади алгоритмів симетричного шифрування включають:

- AES (розширений стандарт шифрування)
- DES (стандарт шифрування даних)
- IDEA (Міжнародний алгоритм шифрування даних)
- Blowfish (Заміна викиду для DES або IDEA)
- RC4 (Rivest Cipher 4)
- RC5 (Rivest Cipher 5)
- RC6 (Rivest Cipher 6)
- AES, DES, IDEA, Blowfish, RC5 і RC6 - це блокові шифри. RC4 - це потоковий шифр.
- DES

У «сучасних» обчисленнях DES був першим стандартизованим шифром для захисту електронних комунікацій і застосовується у різних варіаціях (наприклад, 2-ключевий або 3-клавішний 3DES). Оригінальний DES більше не використовується, оскільки вважається занадто слабким, через потужність обробки сучасних комп'ютерів. Навіть 3DES не рекомендується NIST та PCI DSS 3.2, як і всі 64-бітні шифри. Однак 3DES досі широко використовується в чіпових картах EMV.

AES.Найчастіше використовуваний симетричний алгоритм. Розширений стандарт шифрування (AES), який спочатку був відомий під назвою Rijndael. Це

стандарт, встановлений Національним інститутом стандартів і технологій США в 2001 році для шифрування електронних даних, оголошених у FIPS FIPS США PUB 197. Цей стандарт замінює DES, який застосовувався з 1977 року. За NIST шифр AES має: розмір блоку - 128 біт, але може мати три різні довжини ключів: AES-128, AES-192 та AES-256.

Хоча симетричне шифрування є більш старим методом шифрування, воно швидше та ефективніше, ніж асиметричне шифрування, через проблеми з продуктивністю з розміром даних та великим використанням ЦП. Завдяки кращій продуктивності та більш швидкій швидкості симетричного шифрування (порівняно з асиметричною) симетрична криптографія зазвичай використовується для масового шифрування / шифрування великих обсягів даних, наприклад. для шифрування бази даних. У випадку з базою даних секретний ключ може бути доступний лише самій базі даних для шифрування чи розшифрування.

Приклади використання симетричної криптографії:

- Платіжні додатки, такі як карткові трансакції, де PII потрібно захистити, щоб запобігти крадіжці особи або шахрайським зборам
- Перевірки на підтвердження того, що відправник повідомлення є тим, ким він претендує
- Генерація або хешування випадкових чисел
- Ключове управління для симетричного шифрування - те, що нам потрібно врахувати

Симетричне шифрування має свої недоліки. Найслабшою мірою його аспектів управління ключовими словами, включаючи:

- Ключове виснаження

  Симетричне шифрування страждає від передачі інформації, коли кожне використання ключа видає певну інформацію, яка може потенційно використати зловмисник для реконструкції ключа. Захист проти такої поведінки включає використання ієрархії ключів, щоб переконатися, що користувачі або ключі шифрування ключів не використовуються надмірно, і

відповідно, повернення ключів, які шифрують обсяги даних. Щоб відслідковувати обидва ці рішення, необхідні грамотні стратегії управління ключами, на випадок, коли замінили ключ шифрування і неможливо відновити, у такому разі дані потенційно втрачаються.

- Дані про віднесення

  На відміну від асиметричних (відкритих ключів) сертифікатів, симетричні ключі не мають вбудованих метаданих для запису такої інформації, як дата закінчення терміну дії або списку контролю доступу, щоб вказати на використання ключа, який може бути застосований для шифрування, але не для розшифрувати, наприклад.

Останній пункт деякою мірою вирішений такими стандартами, як ANSI X9-31, де ключ може бути пов'язаний з інформацією, яка визначає його використання. Але для повного контролю над тим, для чого ключ може бути використаний, і коли його можна використовувати, потрібна система управління ключами.

Керування ключами у великих масштабах

Якщо в схемі задіяно лише кілька ключів (від десятків до низьких сотень), накладні витрати керуються скромно і їх можна обробляти за допомогою ручної діяльності людини. Однак, з великим обсягом, відстеження закінчення терміну дії та впорядкування обертання ключів швидко стає недоцільним.

2.6. Ассиметричне шифрування

Асиметрична криптографія, також відома як криптографія з відкритим ключем, - це процес, який використовує пару пов'язаних ключів - один відкритий ключ та один приватний ключ - для шифрування та розшифрування повідомлення, та захисту його від несанкціонованого доступу чи використання. Публічний ключ - це криптографічний ключ, який може використовуватись будь-якою особою для шифрування повідомлення, щоб його можна було розшифрувати лише призначений одержувач за допомогою свого приватного ключа. Приватний ключ -

також відомий як секретний ключ - надається лише ініціатору ключа.

Багато протоколів покладаються на асиметричну криптографію, включаючи протоколи безпеки транспортного рівня (TLS) та протоколи захищеного сокета (SSL), що робить можливим HTTPS. Процес шифрування також використовується в програмних програмах - таких як браузери, які потребують встановлення безпечного з'єднання через незахищену мережу, як Інтернет, або потребують перевірки цифрового підпису.

Асиметричне шифрування зазвичай використовується для аутентифікації даних за допомогою цифрових підписів. Цифровий підпис - це математична методика, яка використовується для перевірки справжності та цілісності повідомлення, програмного забезпечення або цифрового документа.

На основі асиметричної криптографії цифрові підписи можуть гарантувати докази походження, ідентичності та статусу електронного документа, транзакції чи повідомлення, а також підтверджувати інформовану згоду підписанта.

Асиметрична криптографія також може бути застосована до систем, в яких багатьом користувачам може знадобитися шифрувати та розшифровувати повідомлення, включаючи:

- Зашифрований електронний лист - відкритий ключ може використовуватися для шифрування повідомлення, а приватний ключ - для його розшифровки.
- Криптографічні протоколи SSL / TSL - встановлення зашифрованих зв’язків між веб-сайтами та браузерами також використовує асиметричне шифрування.
- Bitcoin та інші криптовалюти покладаються на асиметричну криптографію, оскільки користувачі мають відкриті ключі, які кожен може бачити, і приватні ключі, які зберігаються в таємниці. Bitcoin використовує криптографічний алгоритм, щоб гарантувати що витрачати кошти можуть лише законні власники.

РОЗДІЛ 3
СТВОРЕННЯ ЗАХИЩЕНОГО API ДЛЯ АВТОРИЗАЦІЇ КОРИСТУВАЧІВ ДЛЯ ВЕБ-РЕСУРСІВ

3.1. Розробка базової структури мікросервісу за принципами SOLID на Node.js

Базова структура мікросервісу передбачає підняття окремого серверу та встановлення проксі. Структура тек серверу виглядає наступним чином:

myServer/bin/www – файл з глобальними настройками серверу(порт, створення серверу)

myServer/node_modules – тека, де знаходяться усі використовувані бібліотеки, пакети

myServer/controllers – опціональна тека, куди можна винести функції роутів

myServer/routes – тека, де знаходяться роути(функції що відповідають за відповідний контроллер для урл шляху)

myServer/utils – утілітарна тека, де знаходяться функції обробники, які мають багаторазове використання

myServer/app.js – файл, що відповідає за настройку серверу

myServer/package.json – файл, що відповідає за контроль версій використовуваних модулів(обновлюється автоматично)

Така структура проекту є канонічною для SOLID архітектури, адже всі компоненти додатку максимально дрібно рознесені, та можуть буду повторно використані і додання нового функціоналу не змінить минулої структури.

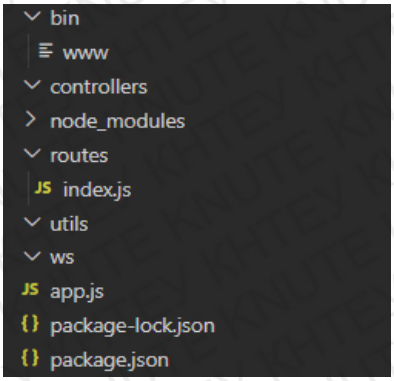

Рисунок 3.1.1. Структура проекту

3.2. Імплементація роутів REST методології.

Створюється файл index.js у теці routes, де прописую тестовий роут з відповіддю “It`s alive!”, та статусом 200. Тестування роуту за допомогою Postman.

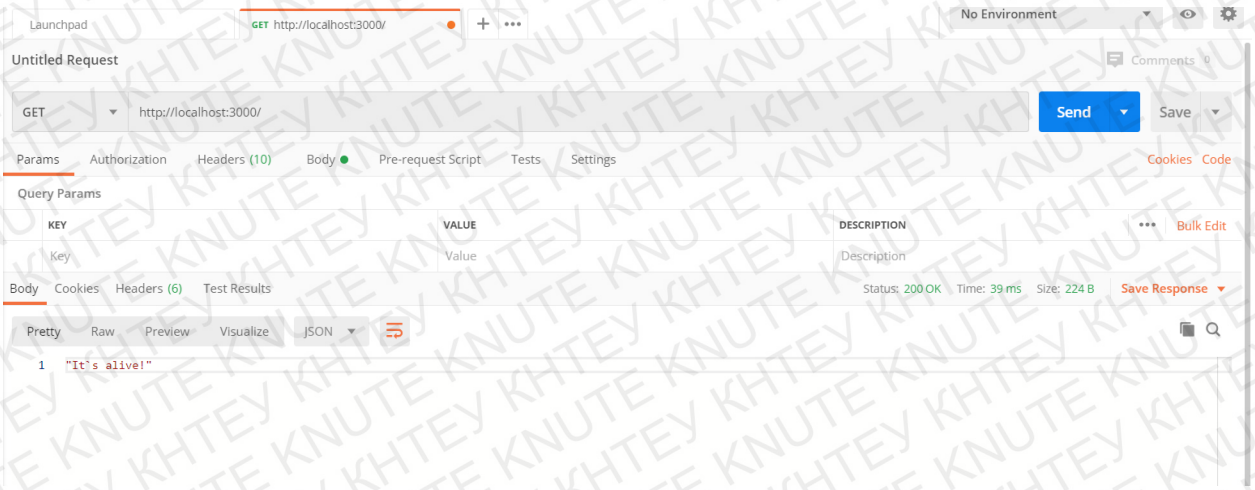

Рисунок 3.2

Роут відпрацював успішно, адже видно статус 200, і відповідь від серверу “It`s alive!”. Таким чином дотримується методологія REST відправляючи лише стан та статус відповіді, а не веб-сторінку.

3.3 Настройка генерації та видачі токенів з цифровим підписом, згенерованим симетричним шифруванням з веб-сокет та REST роутів

Для щифрування біло выбрано алгоритм RSA-512. Алгоритм є

асиметричним, що забезпечує більшу надійність у порівнянні з симетричними.

Для генерації токенів буде використан модуль з npm – jsonwebtoken та jws. Генеруємо ключі за допомогою веб-сайту http://travistidwell.com/blog/2013/09/06/an-online-rsa-public-and-private-key-generator/ . Після генерації необхідно створити два файли public.key та private.key, та занести їх у корінь проекту.

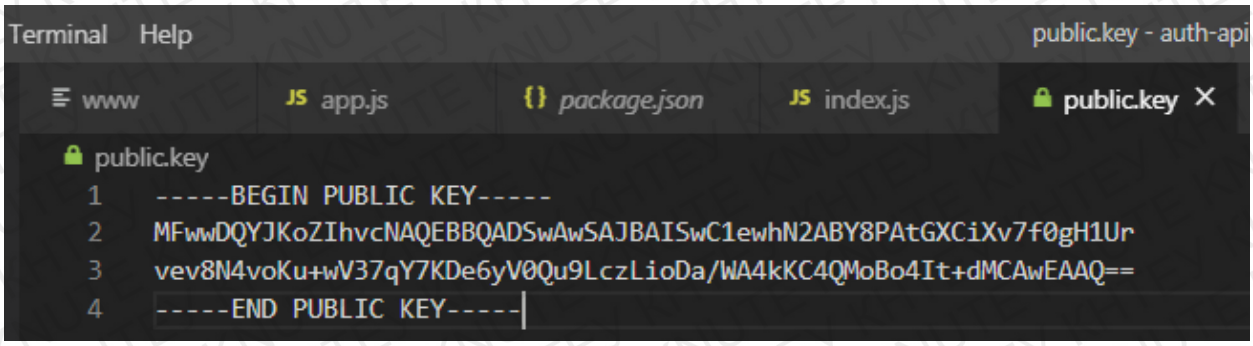

Рисунок 3.3.1 Публічний ключ

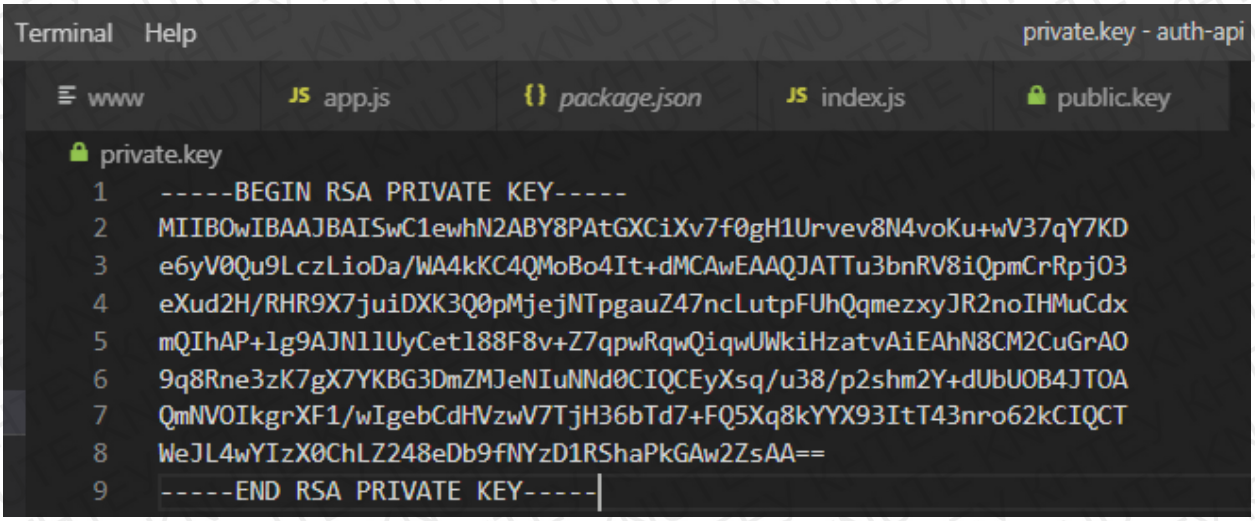

Рисунок 3.3.2 Приватний ключ

Далі створюється роут для реєстрації користувачів, видається токен записаний у кукі файли. Після чого для перевірки створюється захищений роут для перевірки роботи токену

```
14
15  // видача токену
16  router.post('/register', (req, res) => {
17    const token = jws.sign({
18      header: {alg: 'RS256'},
19      payload: 'sectret data',
20      privateKey: privateKey
21    });
22
23    if(!token) {
24      res.status(400).json('error while creating a token')
25    }
26
27    res.cookie('jwt', token, { maxAge: 1000 * 3600 })
28    res.status(200).json({token})
29  })
30
31  // захищений роут
32  router.get('/protectedRoute', async (req, res) => {
33    const token = req.cookies['jwt'];
34    jwt.verify(token, publicKey,  {algorithms: ['RS256']}, (err, decoded) => {
35      if(!decoded || err) {
36        res.status(403).json('unathorized')
37      }
38      res.status(200).json(decoded)
39    });
40
41  })
```

Рисунок 3.3.3 Роут реєстрації і захищений роут

Для перевірки роботи використовують Postman, і якщо в цьому випадку сервер видасть відповідь зі статусом 403, та текстом unathorized, а при правильному незміненому токені — 200 і декодований токен, то цей код працює вірно.

Рисунок 3.3.4 Токен у cookie

Рисунок 3.3.5 Відповідь сервера на реєстрацію

Як видно, видача токену працює коректно, тепер необхідно перевірити правильність роботи захищеного роуту.

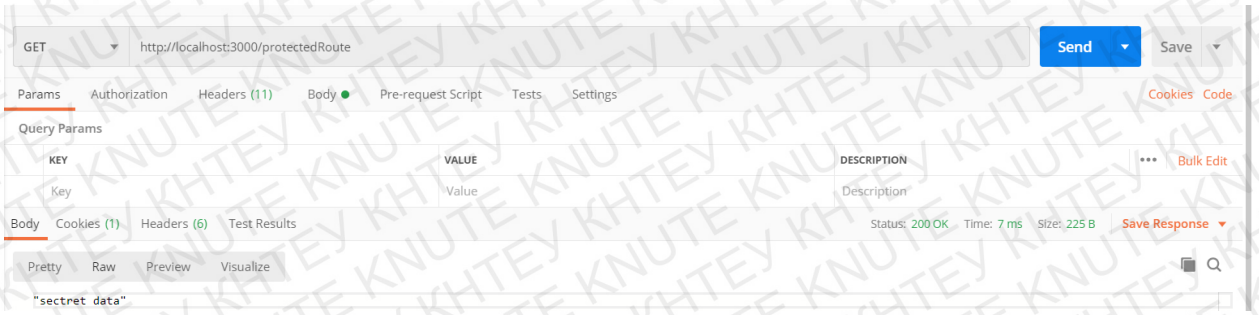

Рисунок 3.3.6 Відповідь сервера на захищений роут

Впевнелись у тому, що якщо токен є і він у незміненому стані, то отриманий статус — 200, і у відповідь ми отримали декодовані дані.

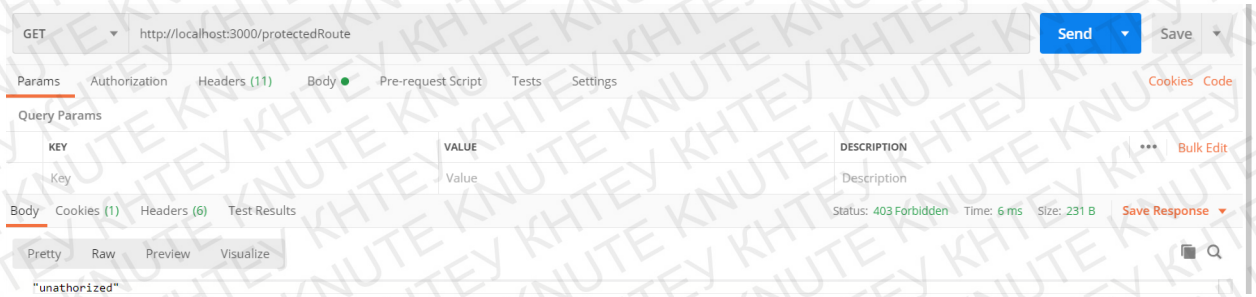

Рисунок 3.3.7 Відповідь сервера на захищений роут при неправильному або відсутньому токені

При зміні або відсутності токену, сервер видає статус 403, та у відповідь відсилає текст “unathorized”. Це означає, что шифрування працює необхідним чином. Тепер

необхідно підключити форму реєстрації. Так як це мікросервіс серверної частини, то візуалізації тут бути не може, адже цей мікросервіс відповідає виключно за API запити. Тому тепер у токен будуть зашифровані дані форми. Для цього необхідно записувати дані форми записаны у базі данних у тіло токену.

База даних буде використана MongoDB. Це нереляційна база даних, а це означає, що вона у краще піддається масштабуванню і змінам.

```
mongoose.connect(`mongodb://localhost:27017/knute`, settings).then(() => {
  console.log('connected');
}).catch((err) => {
  console.log(err);
});
```

Рисунок 3.3.8 Підключення до бази

Рисунок 3.3.9 Модель користувача

У моделі користувача використовується симетричне шифрування і у базі даних зберігається в зашифрованому вигляді, так як зашифрований пароль у відкритому доступі не знаходиться.

```
// видача токену
router.post('/register', async (req, res) => {
  try {
    const { name, lastName, password, username } = req.body;

    const user = await UserModel.create({name, lastName, password, username})

    if(!user) {
      throw new Error('no user')
    }

    const token = jws.sign({
      header: {alg: 'RS256'},
      payload: {name, lastName, username},
      privateKey: privateKey
    });

    if(!token) {
      throw new Error('error while creating a token')
    }

    res.cookie('jwt', token, { maxAge: 1000 * 3600 })
    res.status(200).json({token})
  } catch(err) {
    res.status(400).json(err)
  }
})
```

Рисунок 3.3.10 Роут реєстрації користувача

При реєстрації користувача унікальним полем ідентифікації для бази даних було взято поле username. У токен додатково передано ім’я і прізвище, для зручності доступу.

Далі необхідно зробити роут логіну, який би перевіряв наявність користувача у базі даних і порівнював зашифровані паролі(введений користувачем при логіну, і той, що знаходиться у базі даних).

```
router.post('/login', async (req, res) => {
  try {
    const {password, username } = req.body;
    const user = await UserModel.findOne({username})
    if(!user) {throw new Error('wrong pw')}
    user.comparePassword(password, (err, match) => {
      if(err || !match) {
        res.status(400).json('wrong pw')
      } else {
        const token = jws.sign({
          header: {alg: 'RS256'},
          payload: {name: user.name, lastName: user.lastName, username},
          privateKey: privateKey
        });
        if(!token) {res.status(400).json('wrong pw') }
        res.cookie('jwt', token, { maxAge: 1000 * 3600 })
        res.status(200).json({token})
      }
    })
    if(!user) {throw new Error('no user')}
  } catch(err) {res.status(400).json(err) }
})
```

Рисунок 3.3.11 Роут логіну

Далі останній крок — логаут. Для цього створюється роут, де cookie видаляється.

```
router.get('/logout', async (req, res) => {
  res.clearCookie('jwt');
  res.status(200).json('logout completed');
})
```

Рисунок 3.3.12 Роут логауту

ВИСНОВКИ

У випускному кваліфікаційному проекті представлені знання, набуті у сферах захисту API, і практичні приклади реалізації цих знань.

Розглянувши різні методи авторизації і аутентифікації було вирішено будувати проект, у якому авторизація і буде на токенах, де підпис генерується асиметричним шифруванням, та токени передаються у cookie-файлах відправлених з серверу. Такий вибір зумовлений наступними факторами:

- По-перше, було вирішено використовувати не лишу cookie або токени, а їх разом. Cookie надають зручний спосіб передачі і отримання даних сервером, та дають додатковий шар захисту для токена, адже він як правило зберігається у localStorage браузеру, який є чутливим до атак. Передання токену у cookie дає додатковий шар захисту cookie — якщо данні у cookie будут змінені, то цифровий підпис токену буде не валідним.
- По-друге, для шифрування цифрового підпису було вирішено надати перевагу симетричному шифруванню, так як воно є більш надійним, в плату за більш повільну генерацію підписів. З таким об’ємом даних це ніяк не скажеться і навіть якщо максимальний розмір корисного навантаження токену буде досягнуто.

Пароль користувача зберігається у симетрично зашифрованому вигляді, це ніяк не скажеться на безпеці, адже зашифрований пароль ніде не передається у відкритому доступі.

API побудовано на SOLID принципах для зручного розширення, яке не понесе багато змін для імплементації у проект. Частково такий шаблон неможливо побудувати дотримуючись усіх принципів, через свою специфіку, адже кожен новий проект буде мати різні поля в базі даних, а тому модель у будь-якому випадку доведеться правити, але сам проект на основі цього шаблону

побудований з дотриманням усіх принципів SOLID. Для передачі використовується методологія REST, адже використання аналогів (GraphQL), наклало би на проект зобов’язання використовувати лише їх через специфіку побудови таких додатків: до REST можна дописати GraphQL, а навпаки — ні. Відкритий код розміщено на https://github.com/lukashevkyi/knute_VCR

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Э. Мэйволд. Безпека мереж — 2016.
2. Шнайер Б. Applied Cryptography. Protocols, Algorithms and Source Code in C 2018(Reprint).
3. Жельников В. Криптогрія з папіруса до 1996. 2014
4. К. Шеннон. Теорія звязку в секретных системах 2019(Переклад Литвиненко О.О.).
5. Мао В. Сучасна криптографія: Теорія и практика 2015.
6. А. П. Алферов, А. Ю. Зубов, О. С. Кузьмин, О. В. Черемушкин. Основи Криптографії.. — 2017.
7. Носов В. А. Короткий історичний переказ розвитку криптографії — 2014.
8. JWT[Електронний ресурс] - https://jwt.io
9. https://tools.ietf.org [Електронний ресурс] - IETF офіційний веб-сайт
10. https://web.archive.org/web/20171201173336/https://auth0.com/e-books/jwt-handbook - JWT Handbook, автор - Sebastián Peyrott
11. Картіт, Заїд (Січень 2016). "Applying Encryption Algorithms for Data Security in Cloud Storage, Kartit, et al"
12. Дельфс, Ханс & Кнельб, Хельмут (2015). "Symmetric-key encryption". Introduction to cryptography: principles and applications. Springer.
13. Мюллен, Гарі & Муммерт, Карл (2014). Finite fields and applications. American Mathematical Society
14. "Demystifying symmetric and asymmetric methods of encryption". Журнал - Cheap SSL Shop. 2017-09-28.
15. Пельц & Паар (2015). Understanding Cryptography.
16. Роедер, Том. "Symmetric-Key Cryptography" [Електронний ресурс] - www.cs.cornell.edu. Retrieved 2017-02-05.
17. Саломаа А. Криптографія з відкритим ключем. — М.: Мир, 2015(Переклад Волков А.І.)
18. А. Д. Менезес, П. К. ван Ооршох, С. А. Ванстон. Handbook of Applied

Cryptography. — 2018(Reprint).
19. Express [Електронний ресурс] - https://www.npmjs.com/package/express
20. JWT [Електронний ресурс] - https://www.npmjs.com/package/jwt
21. Mongoose [Електронний ресурс] - https://www.npmjs.com/package/mongoose
22. JWS [Електронний ресурс] - https://www.npmjs.com/package/jws
23. Node.js official documentation [Електронний ресурс] - https://nodejs.org/uk/docs/
24. MongoDB [Електронний ресурс] - https://docs.mongodb.com/manual/
25. PostMan [Електронний ресурс] - https://www.postman.com/product/api-client/
26. Bcrypt [Електронний ресурс] - https://www.npmjs.com/package/bcrypt
27. REST API [Електронний ресурс] - https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
28. REST [Електронний ресурс] - https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
29. Robo3T [Електронний ресурс] - https://robomongo.org/
30. Git [Електронний ресурс] - https://git-scm.com/docs