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

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«До захисту допущено»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

спеціальності 121 Інженерія програмного забезпечення

на тему: «Програмна бібліотека для підтримки створення сервера авторизації на базі OAuth 2.0 для Node.js»

Виконав:
студент IV курсу, групи КП-62
Бай Ярослав Володимирович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н., доцент,
Заболотня Тетяна Миколаївна _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
Доцент кафедри ММСА ІПСА, к.т.н., доцент,
Дідковська Марина Віталіївна _____

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

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

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

ЗАТВЕРДЖУЮ
Науковий керівник кафедри
_____ Іван ДИЧКА
«___»_____2019 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Бай Ярославу Володимировичу

1. Тема проєкту «Програмна бібліотека для підтримки створення сервера авторизації на базі OAuth 2.0 для Node.js», керівник проєкту Заболотня Тетяна Миколаївна, к.т.н., доцент, затверджені наказом по університету від «25» травня 2020 р. № 1181-с.

2. Термін подання студентом проєкту «15» червня 2020 р.

3. Вихідні дані до проєкту: див. Технічне завдання.

4. Зміст пояснювальної записки:
   - огляд проблеми, яку повинна вирішити дана бібліотека, опис вимог до неї та аналіз існуючих рішень;
   - аналіз мов програмування, які можна застосувати для розроблення бібліотеки для оточення Node.js;
   - розроблення бібліотеки для підтримки створення сервера авторизації на базі протоколу OAuth 2.0;
   - опис використаних структур даних та допоміжних програмних засобів.

5. Перелік обов'язкового графічного матеріалу:
   - робота із демонстраційним додатком (креслення);
   - обробка запиту клієнта (креслення);

– діаграма авторизації за протоколом OAuth 2.0 (плакат);
– алгоритм оновлення маркерів доступу та оновлення доступу (плакат).

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент		

7. Дата видачі завдання «31» жовтня 2019 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	14.11.2019	
2.	Розроблення та узгодження технічного завдання	29.11.2019	
3.	Підготовка матеріалів першого розділу дипломного проєкту	30.12.2019	
4.	Розроблення архітектури бібліотеки для підтримки створення сервера авторизації	15.01.2020	
5.	Розроблення структури окремих частин бібліотеки та взаємодії між ними	03.02.2020	
6.	Підготовка матеріалів другого розділу дипломного проєкту	20.02.2020	
7.	Програмна реалізація бібліотеки для підтримки створення сервера авторизації	10.03.2020	
8.	Тестування розробленої програмної бібліотеки	17.03.2020	
9.	Підготовка матеріалів третього розділу дипломного проєкту	30.03.2020	
10.	Підготовка матеріалів четвертого розділу дипломного проєкту	11.04.2020	
11.	Підготовка графічної частини дипломного проєкту	21.04.2020	
12.	Оформлення документації дипломного проєкту	26.05.2020	

Студент Ярослав БАЙ

Керівник проєкту Тетяна ЗАБОЛОТНЯ

АНОТАЦІЯ

Даний дипломний проєкт присвячений розробленню програмної бібліотеки для створення сервера авторизації на базі протоколу OAuth 2.0 для програмної платформи Node.js.

У роботі виконано аналіз існуючих на даний момент програмних бібліотек для вирішення даної проблеми. Розроблена бібліотека повністю відповідає стандартам для фреймворку авторизації OAuth 2.0 та надає користувачам можливість у зручний спосіб створити сервер авторизації та самим обирати спосіб, у який бібліотека буде зберігати усю супутню інформацію (коди авторизації, маркери отримання та відновлення доступу). Окрім цього, дана бібліотека реалізована за стилем бібліотек програмного комплексу TypeStack, а отже є достатньо гнучкою та дозволяє користувачам використовувати їх власний контейнер впровадження залежностей. Окрім цього, є окрема версія для використання із фреймворком Nest.js.

У даному проєкті розроблено та досліджено: архітектуру двох версій бібліотек (загальної та розробленої спеціально для фреймворка Nest.js) та реалізовано алгоритми авторизації відповідно до типів гранту, описаних у специфікації фреймворку авторизації.

ABSTRACT

This project is dedicated to the development of a software library for creating an authorization server based on the OAuth 2.0 protocol for the software platform Node.js.

The analysis of the existing software libraries for solving this problem is performed in the work. The developed library fully complies with the standards for the OAuth 2.0 authorization framework and allows users to conveniently create an authorization server and choose the way in which the library will store all related information (authorization codes, access and refresh tokens). In addition, this library is implemented in the TypeStack style of libraries, and therefore is quite flexible and allows users to use their own dependency injection container. In addition, there is a separate version for use with the Nest.js framework.

In this project there were developed and researched: the architecture of two versions of libraries (common version and the version developed specifically for the Nest.js framework) and implemented authorization algorithms according to the grant types described in the authorization framework specification.

АННОТАЦИЯ

Данный дипломный проект посвящен разработке программной библиотеки для создания сервера авторизации на базе протокола OAuth 2.0 для программной платформы Node.js.

В работе выполнен анализ существующих на данный момент программных библиотек для решения данной проблемы. Разработанная библиотека полностью соответствует стандартам для фреймворка авторизации OAuth 2.0 и предоставляет пользователям возможность в удобной форме создать сервер авторизации и самим выбирать способ, которым библиотека будет хранить всю сопутствующую информацию (коды авторизации, маркеры получения и восстановления доступа). Кроме этого, данная библиотека реализована по стилю библиотек программного комплекса TypeStack, а следовательно является достаточно гибкой и позволяет пользователям использовать их собственный контейнер внедрения зависимостей. Кроме этого, есть отдельная версия для использования с фреймворком Nest.js.

В данном проекте разработаны и исследованы: архитектуру двух версий библиотек (общей и разработанной специально для фреймворка (Nest.js) и реализованы алгоритмы авторизации в соответствии с типами гранта, описанных в спецификации фреймворка авторизации.

ДП.045440-01-90 Програмна бібліотека для підтримки створення сервера авторизації на базі OAuth 2.0 для Node.js. Відомість проєкту

Позначення	Найменування	Кіл-ть	Примітка
	Документація проєкту		
ДП.045440-02-91	Програмна бібліотека для	5	
	підтримки створення		
	сервера авторизації на		
	базі OAuth 2.0 для		
	Node.js. Технічне		
	завдання		
ДП.045440-03-81	Програмна бібліотека для	62	
	підтримки створення		
	сервера авторизації на		
	базі OAuth 2.0 для		
	Node.js. Пояснювальна		
	записка		
ДП.045440-04-51	Програмна бібліотека для	4	
	підтримки створення		
	сервера авторизації на		
	базі OAuth 2.0 для		
	Node.js. Програма та		
	методика тестування		
ДП.045440-05-33	Програмна бібліотека для	12	
	підтримки створення		
	сервера авторизації на		
	базі OAuth 2.0 для		
	Node.js. Керівництво		
	програміста		

Позначення	Найменування	Кіл-ть	Примітка
ДП.045440-06-99	Програмна бібліотека для	1	
	підтримки створення		
	сервера авторизації на		
	базі OAuth 2.0 для		
	Node.js. Робота із		
	демонстраційним		
	додатком. Блок-схема		
	алгоритму		
ДП.045440-07-99	Програмна бібліотека для	1	
	підтримки створення		
	сервера авторизації на		
	базі OAuth 2.0 для		
	Node.js. Обробка запиту		
	клієнта. UML-діаграма		
	потоку даних		
ДП.045440-08-98	Програмна бібліотека для	1	
	підтримки створення		
	сервера авторизації на		
	базі OAuth 2.0 для		
	Node.js. Компакт-диск		

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2019 р.

ПРОГРАМНА БІБЛІОТЕКА ДЛЯ ПІДТРИМКИ СТВОРЕННЯ СЕРВЕРА АВТОРИЗАЦІЇ НА БАЗІ OAUTH 2.0 ДЛЯ NODE.JS

Технічне завдання

ДП.045440-02-91

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Тетяна ЗАБОЛОТНЯ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____Ярослав БАЙ

2019

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Програмна бібліотека для підтримки створення сервера авторизації на базі OAuth 2.0 для оточення Node.js.

Галузь застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБЛЕННЯ

Підставою для розроблення є завдання на дипломне проєктування, затверджене кафедрою програмного забезпечення комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» (КПІ ім. Ігоря Сікорського).

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробки призначена для використання при розробленні програмних додатків на мові програмування JavaScript та мовах, що можуть бути скомпільовані у JavaScript на програмній платформі Node.js. Дана бібліотека буде особливо корисною для додатків на мові програмування TypeScript, оскільки вона реалізована на даній мови та природньо підтримує її визначення типів. Окрім цього, бібліотека написана у такий спосіб, щоб її можна було зручно використовувати із власним контейнером впровадженням залежностей.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

Бібліотека повинна забезпечувати такі основні функції:

1) повноцінна підтримка стандартів фреймворку авторизації RFC 6749 та 6750;
2) незалежність відносно типу сховища;
3) масштабування виконавців завдань для збільшення відмовостійкості;
4) підтримка впровадження власних типів грантів;
5) повна підтримка типів мови програмування TypeScript;

6) незалежність від фреймворку для розробки серверної частини веб-додатків;

7) наявність окремої версії для фреймворку Nest.js.

Розробку виконати на TypeScript.

Додаткові вимоги:

1) незалежність від контейнеру впровадження залежностей;

2) має легко розгортатися на будь-якій операційній системі;

3) повинна мати інтерфейс керування у вигляді CLI – утиліти командного рядку;

5. ВИМОГИ ДО ПРОЄКТНОЇ ДОКУМЕНТАЦІЇ

У процесі виконання проєкту повинна бути розроблена наступна документація:

1) пояснювальна записка;

2) програма та методика тестування;

3) керівництво програміста;

4) креслення:

- «Робота із демонстраційним додатком. Блок-схема алгоритму»;
- «Обробка запиту клієнта. UML-діаграма потоку даних».

6. ЕТАПИ ПРОЄКТУВАННЯ

Вивчення літератури за тематикою роботи…….14.11.2019

Розроблення та узгодження технічного завдання……...29.11.2019

Розроблення архітектури бібліотеки……..……..15.01.2019

Розроблення структури окремих частин бібліотеки.…….…..03.02.2020

Програмна реалізація бібліотеки...……....10.03.2020

Тестування бібліотеки…….....17.03.2020

Підготовка матеріалів текстової частини проєкту……..….11.04.2020

Підготовка матеріалів графічної частини проєкту……...21.04.2020

Оформлення технічної документації проєкту……..…26.05.2020

7. ПОРЯДОК ТЕСТУВАННЯ РОЗРОБКИ

Тестування розробленого програмного продукту виконується відповідно до “Програми та методики тестування”.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

ПРОГРАМНА БІБЛІОТЕКА ДЛЯ ПІДТРИМКИ СТВОРЕННЯ СЕРВЕРА АВТОРИЗАЦІЇ НА БАЗІ OAUTH 2.0 ДЛЯ NODE.JS

Пояснювальна записка

ДП.045440-03-81

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Тетяна ЗАБОЛОТНЯ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Ярослав БАЙ

2020

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

ПБ – програмна бібліотека, набір об’єктів чи підпрограм для розробки програмного забезпечення.

Архітектура програмного забезпечення – сукупність високорівневих рішень щодо організації програмної системи.

Парадигма програмування – система правил для визначення стилю написання програм.

Скрипт – це програма або програмний файл (частіше невеликого розміру), який автоматизує певні дії.

Лінтер – програмне забезпечення, статичний аналізатор з можливість конфігурування для певної мови програмування.

Веб-браузер – програмне забезпечення, що надає можливість перегляду веб-сторінок.

JavaScript (JS) – динамічна, об'єктно-орієнтована прототипна мова програмування.

ECMAScript – стандарт мови програмування.

TypeScript (TS) – мова програмування, яка є строгим розширенням мови програмування JavaScript та компілюється у JavaScript.

CofeeScript – мова програмування, яка компілюється у JavaScript та додає до неї синтаксичний цукор як у мовах Python, Ruby та Haskell.

Kotlin – статично типізована мова програмування, що працює поверх JVM (Java Virtual Machine) та може бути скомпільована у JS.

Java – типізована об'єктно-орієнтована мова програмування, яка була створена задля зручного перенесення програм з однієї платформи до іншої.

Scala – статична мова, що поєднує в собі функціональну та об’єктно-орієнтовану парадигми, реалізована для платформ Java та JavaScript.

Gradle – система автоматизації збірки для платформи JVM, яка використовує задля конфігурації скрипти на мовах програмування Groovy та Kotlin.

Фронтенд – частина веб-додатку, яка описує інтерфейс та логіку взаємодії клієнта з ним.

Бекенд – частина веб-додатку, що описує логіку взаємодії клієнта з одним або декількома серверами.

Фреймворк – програмна платформа, що визначає структури програмної системи.

Node.js – програмне оточення для мови програмування JavaScript, що надає можливість взаємодіяти із засобами введення/виведення та підключати залежності, написані на різних мовах програмування.

Express.js – фреймворк для веб-додатків на базі Node.js, створений на основі Sinatra – веб-фреймворка для мови програмування Ruby.

NPM – node package manager, менеджер пакетів за замовченням для оточення Node.js.

DDD – domain driven design, предметно-орієнтоване проєктування. Набір принципів, спрямованих на оптимальне створення систем об’єктів.

ВСТУП

Разом із стрімким зростанням можливостей обчислювальних ресурсів, ми можемо спостерігати стрімке розширення можливостей програмних систем та їх вимог до цих самих ресурсів. Та зазвичай програми випереджають можливості обладнання, ~~та~~ стають все більш складними і їх важко підтримувати. Коли програмне забезпечення стає достатньо великим, гостро постає питання стосовно архітектури цього програмного забезпечення, адже саме архітектура визначає, наскільки складно або легко можна вносити зміни у вже написаний програмний код. На даний час достатньо популярною є об'єктно-орієнтовна парадигма програмування, яка тривалий час дозволяє створювати високо-масштабовані програмні системи за допомогою використання таких логічних одиниць, як класи та об'єкти.

Достатньо довгий час програмна мова JavaScript [1] використовувалася для написання невеличких підпрограм (скриптів) для створення анімованого контенту у веб-браузерах. Але зростання веб-індустрії призвело до підвищення вимог до інтерактивних сторінок і через деякий час звичайних маленьких скриптів було вже замало. Почали з'являтися програмні бібліотеки та високорівневі фреймворки, що абстрагували багато рутинних задач та надавали програмісту більше можливостей для створення та підтримки великої бази коду для сучасних веб-сторінок. У цей самий час мова програмування JavaScript завдяки своїй динамічній природі здобула любов не тільки у розробників інтерфейсу веб-сайтів, але й в інших сферах розроблення програмного забезпечення. Так ця програмна мова почала використовуватись ще й для написання серверної частини веб-додатків, створення повноцінних настільних та мобільних додатків та навіть для написання коду для мікроконтролерів. Дуже швидко з'явилася необхідність у використанні гнучких та надійних практик створення додатків. Саме тому з'явилася мова TypeScript [2].

Мова TypeScript є суворою надмножиною над мовою JavaScript та може компілюватися в неї. Головними перевагами цієї мови є можливість використання статичної строгої типізації та використання більш традиційної версії парадигми об’єктно-орієнтованого програмування, з використанням класів та інтерфейсів. Ця мова значно спростила написання як клієнтської частини веб-додатків, так і серверної частини. На разі дана мова програмування використовується як основа для створення багатьох TypeScript та JavaScript бібліотек завдяки тому, що її можна добре масштабувати.

Одним з фреймворків, що використовує мову TypeScript, є Nest.js [3] – фреймворк для написання бекенд-коду для оточення Node.js [4]. Не зважаючи на те, що ця платформа вибухово зростає у останній час, деякі з існуючих програмних рішень є досі недостатньо якісними або потребують оновлення. Однією з проблем є відсутність гнучкої бібліотеки для реалізації сервера авторизації на базі протоколу OAuth 2.0.

Виходячи з цього, метою даного дипломного проєкту є створення програмної бібліотеки для створення сервера авторизації на базі OAuth 2.0 для оточення Node.js та окремої версії бібліотеки, що спрощує інтеграцію цього рішення із популярним фреймворком Nest.js.

1. ОГЛЯД ІСНУЮЧИХ ПРОГРАМНИХ РІШЕНЬ

1.1. Огляд проблеми, яка вирішується ПБ

Протокол OAuth 2.0 – це розповсюджений протокол для авторизації, який використовується при розробленні додатків будь-якого типу: мобільних додатках, веб-додатках та додатках для настільного комп'ютера. Головними перевагами даного протоколу над іншими є його гнучка архітектура, яка окремо виділяє чотири ролі під час ідентифікації та аутентифікації: клієнт, сервер авторизації, власник ресурсу та сервер ресурсів. Даний підхід дозволяє використовувати даний протокол не тільки для складних сценарії авторизації (наприклад, за допомогою стороннього сервісу або соціальної мережі), але й для простих сценаріїв авторизації (для таких випадків ролі сервера авторизації поєднуються із роллю власника ресурсу).

Важливою особливістю даного протоколу є використання маркерів отримання та оновлення доступу, які є непрозорими для кінцевого користувача. Даний підхід підвищує надійність ідентифікації користувача, адже потребує від нього оновлення цих маркерів через певний проміжок часу.

Оскільки даний протокол авторизації є важливим та часто використовується при розробленні додатків, постає необхідність реалізації даного протоколу на серверній частині системи. А через те, що реалізація, описана у стандартах, залишається незмінною довгий час, задля збільшення швидкості розроблення логічним рішенням є створення бібліотеки. В даному конкретному випадку розглянемо бібліотеку для програмної платформи Node.js та сукупності технологій, які зазвичай використовують при розробленні для цієї платформи.

Під час розроблення додатків для оточення Node.js все частіше використовується мова програмування TypeScript. Мова TypeScript була створена як сувора надмножина над мовою JavaScript та може

компілюватися в неї. Головними перевагами цієї мови є можливість використання статичної строгої типізації та використання більш традиційної версії парадигми об’єктно-орієнтованого програмування, з використанням класів та інтерфейсів. Ця мова значно спростила написання як клієнтської частини веб-додатків, так і серверної частини. На разі дана мова програмування використовується як основа для створення багатьох TypeScript та JavaScript бібліотек завдяки тому, що її можна добре масштабувати.

Одним з фреймворків для написання бекенд-коду для оточення Node.js, що використовує мову TypeScript, є Nest.js. Не зважаючи на те, що ця платформа вибухово зростає у останній час, деякі з існуючих програмних рішень є досі недостатньо якісними або потребують оновлення. Однією з таких проблем є відсутність гнучкої бібліотеки для реалізації сервера авторизації на базі протоколу OAuth 2.0. Існуючі рішення не підтримують сучасні фреймворки для використання шаблону «впровадження залежностей», через погану підтримку мають достатньо велику кількість помилок та не надають гнучких типів для повноцінного використання мови TypeScript.

Саме тому метою даного дипломного проєкту є створення програмної бібліотеки для створення сервера авторизації на базі OAuth 2.0 для оточення Node.js та окремої версії бібліотеки, що спрощує інтеграцію цього рішення із популярним фреймворком Nest.js.

1.2. Опис вимог до розроблюваної ПБ

Сформовано ряд вимог, яким має відповідати програмна бібліотека для створення сервера авторизації на базі протоколу OAuth 2.0. По-перше, це відсутність прив’язки до певного фреймворку за допомогою використання власних високорівневих абстракцій. Таким чином цю бібліотеку можна буде використовувати з будь-яким фреймворком Node.js-додатку (такими як Express та Fastify). По-друге, бібліотека має бути

побудована таким чином, щоб її можна було зручно використовувати з контейнером впровадження залежностей (dependency injection container), як популярного фреймворку Nest.js, так і власних рішень (наприклад, InversifyJS [5] та TypeDI [6]). По-третє, архітектура має бути достатньо гнучкою для постійного розвитку бібліотеки – з використанням таких сучасних практик, як DDD (Domain Driven Design) та використанням слабко зв'язаних компонентів (loose coupled components). Окрім цього, бібліотека має повністю підтримувати загальноприйняті стандарти для створення сервера авторизації на базі цього протоколу: RFC 6749 [7] та RFC 6750 [8]. Більш специфічні вимоги стосуються самої реалізації: підтримка найбільш розповсюджених типів грантів (authorization_code, refresh_token, client_credentials та password) з можливістю створення власних типів (наприклад, для авторизації через соціальні мережі) та обробкою областей видимості. Важливо також впровадити можливість роботи із різними типами сховищ, з яких окремо потрібно виділити SQL-бази даних та NoSQL-бази даних.

Таким чином, можна узагальнити вищенаведені вимоги:

1. Відсутність прив'язки бібліотеки до конкретного фреймворка.
2. Підтримка бібліотеки зручної інтеграції з контейнерами впровадження залежностей.
3. Наявність гнучкої архітектури, що відповідає останнім тенденціям у цій області.
4. Повна підтримка стандартів RFC 6749 and RFC 6750.
5. Підтримка найбільш розповсюджених типів грантів з можливістю розширення власними типами.
6. Підтримка областей видимості грантів.
7. Підтримка декількох типів сховища.

1.3. Аналіз існуючих рішень

Ідея створення даної бібліотеки виникла після огляду існуючих рішень в якості основи для створення власного проєкту. Сучасний типовий Node.js-проєкт має відповідати певним вимогам, такими як: підтримка типізації (хоча б у формі декларацій типів розщирення *.d.ts*), наявність високорівневого фреймворку з використанням шаблону «впровадження залежностей» для підвищення гнучкості й масштабовності.

Отже, розглянемо існуючі рішення, що мають надавати зручний інтерфейс для подібного додатку.

1.3.1. Аналіз бібліотеки node-oauth2-server

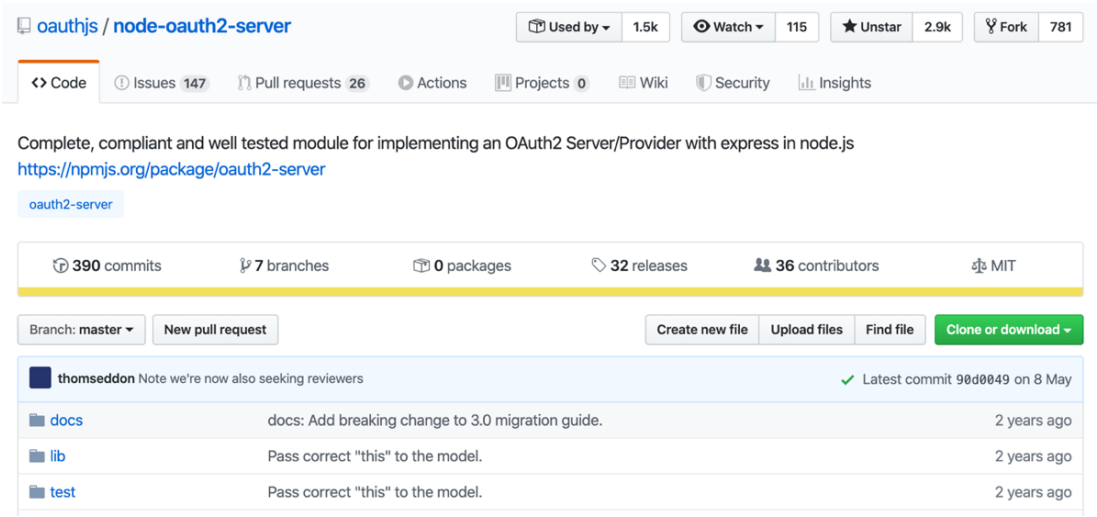


Рис. 1.1. GitHub-репозиторій бібліотеки node-oauth2-server

Дана програмна бібліотека надає повну функціональність для створення сервера авторизації на базі OAuth 2.0 та задовольняє протоколи, описані у стандартах RFC 6749 та RFC 6750. З особливостей варто відмітити підтримку багатьох типів сховищ (наприклад, PostgreSQL, MongoDB, Redis). З особливостей варто відмітити JavaScript в якості основної мови реалізації бібліотеки та відсутність активної підтримки (остання зміна була внесена 2 роки тому, хоча у бібліотеці наявні помилки – це можна побачити із кількості issues у репозиторії на рис. 1.1).

Недоліки даного рішення випливають із особливостей реалізації. По-перше, за основну мову тут узята застаріла на даний час мова JavaScript версії ECMAScript 2015. Через це конструкції async/await та генератори з останніх версії мови підтримуються лише за допомогою компілятора Babel. Це також впливає на зручність підтримки бібліотеки – стиль коду є неконсистентним та розбиття на логічні частини не відповідає сучасним практикам написання коду. Також це поскладнює внесення додаткових змін у код – через застарілі неочевидні конструкції мови JavaScript (наприклад, використання *var* для об'явлення змінної вже значно поскладнює розроблення). Через мову реалізації також виникла необхідність у створенні окремої бібліотеки з деклараціями типів до node-oauth2-server [9]. Типи, що надає дана бібліотека, є недостатньо гнучкими та змушують використовувати власні обгортки над ними.

1.3.2. Аналіз бібліотеки oauth2orize

Помітною бібліотекою для створення сервера авторизації є oauth2orize [10] – бібліотека від автора фреймворка авторизації *passport*. На відміну від розглянутої вище, ця бібліотека є не повністю самостійною, а потребує використання додаткових бібліотек для впровадження різного формату ідентифікації клієнта (таких як passport-http та passport-oauth2-client-password).

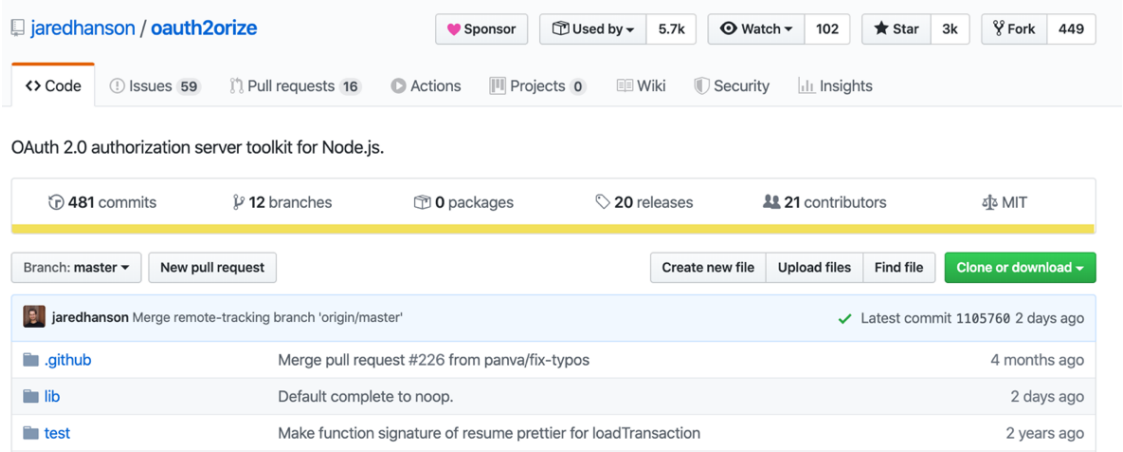


Рис. 1.2. GitHub-репозиторій бібліотеки oauth2orize

На рис. 1.2 можемо побачити що в плані підтримки ситуація збігається з node-oauth2-server. Стосовно реалізації варто відмітити, що бібліотека не є добре масштабованою та, згідно провадженим інтерфейсам, є достатньо специфічною. Власне код написаний мовою JavaScript та також потребує окремої бібліотеки для додавання типів.

1.4. Результати проведеного аналізу

Виходячи з отриманих даних після проведеного аналізу, можна стверджувати, що існуючі системи вирішують задану задачу не повністю та не дуже добре переживають перевірку часом. Обидві бібліотеки використовують застарілі підходи та майже не розвиваються. Окрім цього, вони не вирішують проблему максимально універсально (в першій використовуються типи даних, які потрібно обертати у власні інтерфейси напряму в контролерах, у іншій – специфічний набір інтерфейсів, який не підійде кожному).

Отже, сформовано порівняльну таблицю результатів аналізу існуючих програмних рішень.

Таблиця 1.1

Порівняння існуючих рішень

	Критерії				
Програмні рішення	Сучасна мова реалізації	Гнучкі типи	Актуальна підтримка	Підтримка власних грантів	Незалежність від фреймворку
node-oauth2-server	–	–	–	+	+/–
oauth2orize	+/–	–	–	+	+/–

Виходячи з вищенаведеного, розроблюване програмне рішення має вирішувати наступні проблеми:

1. Використання застарілої мови реалізації.

2. Відсутність масштабованої реалізації.
3. Залежність від фреймворку через недостатньо високорівневу реалізацію.

2. ОБҐРУНТУВАННЯ ВИБОРУ ЗАСОБІВ РЕАЛІЗАЦІЇ

Відповідно до поставленої задачі, а саме створення програмної бібліотеки для створення сервера авторизації на базі протоколу OAuth 2.0 для оточення Node.js, маємо необхідність вибору оптимальних засобів розроблення.

Вимоги до мови програмування:

- можливість компіляції у мову JavaScript або TypeScript – задля сумісності із оточенням Node.js та підтримкою існуючих бібліотек із регістру пакетів NPM;
- підтримка парадигми програмування, що дозволить писати гнучкий та надійний код (функціональної та/або об’єктно-орієнтованої).

2.1. Мова програмування TypeScript

TypeScript – мова програмування, яка позиціонується в якості засобу розробки веб-застосунків, розширює можливості JavaScript та є суворою надмножиною останньої. В силу повної зворотної сумісності адаптація коду існуючих застосунків на нову мову програмування може відбуватися поетапно, шляхом поступового визначення типів. Увесь існуючий код на JavaScript сумісний з TypeScript, тобто в програмах на TypeScript можна використовувати стандартні JavaScript-бібліотеки. Більш того, можна залишити існуючі JavaScript-проєкти в незмінному вигляді, а дані про типи розмістити у вигляді анотацій, що можна помістити в окремі файли, які не заважатимуть розробці і прямому використанню проєкту (наприклад, подібний підхід зручний при розробці JavaScript-бібліотек). Варто відзначити підтримку цієї мовою декілька парадигм програмування, зокрема:

- об’єктно-орієнтованої;
- функціональної;

- імперативної;
- аспектно-орієнтованої.

На даний час TypeScript є популярною мовою загального призначення, що використовується як для написання фронтенд-частини сайту, так і для бекенд-частини.

Виходячи з цього, маємо наступні переваги:

1. Розповсюджена мова програмування, що має потужну підтримку як спільноти розробників, так і великої компанії-розробника.
2. Підтримує статичну типізацію та водночас зберігає динамічну природу мови JavaScript.
3. Додає потужність об’єктно-орієнтованої парадигми у класичному розумінні до JS.
4. При написанні коду чисто на мові програмування TypeScript не потрібно окремо визначати файли з деклараціями типів.

Серед недоліків мови позначимо такі:

1. Через компіляцію даної мови у JS, типи після компіляції не зберігаються, тому усі обмеження, встановлені під час написання коду, не враховуються. Тобто, в програмі не виникне виняткової ситуації, якщо в поле типу «строка» буде передане значення типу «число» – це значення буде неявно приведене до типу «строка» завдяки слабкій типізації мови JavaScript.
2. Впровадження статичної типізації є дуже складною задачею для слабкотипізованої динамічної мови. Використання цього підходу інколи призводить до необхідності у компромісах. Так, JS підтримує гетерогенні масиви, а в TS необхідно створити масив типу any, а потім приводити кожне значення до потрібного типу.

2.2. Мова програмування CoffeeScript

CoffeeScript [11] – мова програмування, що компілюється у JS та доповнює її останню синтаксичним цукром по типу мов Ruby, Python та

Haskell. Метою цієї мови є покращення читабельності коду та зменшення його розміру – намагання зробити його більш лаконічним. В середньому код на мові JavaScript виходить у два рази довшим, ніж код на мові CoffeeScript для виконання однакових задач. Через близькість до мови програмування Ruby, CoffeeScript замінив JavaScript у веб-фреймворку Ruby on Rails. Тому ця мова є широко використовуваною, адже підтримується спільнотою Ruby розробників. Варто відзначити, що скомпільований з CoffeeScript JavaScript- код проходить перевірку лінтером JavaScript Lint. Окрім цього, дана мова підтримує такі самі парадигми програмування, як і мова TypeScript, але, на відміну від останньої, не впроваджує систему статичної типізації.

Виходячи з вищенаведених фактів, маємо наступні переваги:

1. Достатньо популярна мова програмування, яка має підтримку спільноти розробників.
2. Має лаконічний та більш змістовний синтаксис.
3. Підтримує об’єктно-орієнтовану (в дусі Ruby) та функціональну парадигму.
4. Достатньо висока продуктивність вихідного коду JS, адже компілятор використовує останні можливості мови.

Серед недоліків мови можна назвати наступні:

1. Відсутність статичної суворої типізації.
2. Додатковий час на вивчення радикально іншого синтаксису.

2.3. Мова програмування Scala

Scala [14] – мультипарадигмова мова програмування, що дозволяє повноцінно поєднувати підходи об’єктно-орієнтованої та функціональної парадигм програмування, може бути виконана на віртуальній машині Java або скомпільована у JavaScript. Головною відмінністю цієї мови від Java є можливість використання усіх можливостей функціональної парадигми, як у мовах Haskell та Erlang, поряд із більш лаконічним синтаксисом для

об’єктно-орієнтованої парадигми. Часто висловлюються думки, що Scala здатна замінити Java в довгостроковій перспективі.

Ця мова програмування визначається наступними перевагами:

- гнучка мова, яка фактично поєднує в собі функціональну та об’єктну-орієнтовану мови;
- дуже лаконічний синтаксис, який дозволяє значно зменшити кількість рядків коду;
- потужна система типів.

Недоліки даної мови наступні:

- гібридність мови та можливість написати один й той самий код великою кількістю способів призводить до значного ускладнення мови, через що вона потребує набагато більше часу на вивчення;
- майже повна відсутність прикладів для написання серверного коду для оточення Node.js.

2.4. Мова програмування Kotlin

Kotlin [13] – статично типізована мова програмування загального призначення, що може бути скомпільована у байткод JVM, JavaScript або у машиний код деяких платформ засобами LLVM. Автори ставили перед собою ціль створити лаконічнішу та типо-безпечнішу мову, ніж Java, і простішу, ніж Scala. Наслідками спрощення, порівняно з Scala стали також швидша компіляція та краща підтримка IDE. Ця мова повністю сумісна з мовою програмування Java та може використовуватися у одному проєкті разом із мовою Java завдяки системі збірки Gradle.

Для цієї мови варто відзначити такі переваги, як:

- більш простий синтаксис, аніж у мови Scala;
- потужна система типів;
- відмінна підтримка у IDE;
- наявність функціональних можливостей.

Окрім наведених переваг, зазначається і наступний недолік – майже повна відсутність прикладів коду та документації для роботи з оточенням Node.js.

Отже, після огляду усіх перелічених мов програмування, можемо стверджувати, що усі розглянуті мови задовольняють зазначеним вимогам. CoffeeScript дозволяє поєднувати підходи у розробці мов Python, Ruby та Haskell із філософією JavaScript та писати лаконічний код. TypeScript дозволяє поєднувати сувору статичну систему типізації поряд із динамікою JS. JVM-мови (Scala та Kotlin) мають потужну систему типів та синтаксичний цукор для продуктивного створення об'єктно-орієнтованих програм. Окрім цього, зазначені мови усі задовольняють головній вимозі – мають можливість компіляції у мову JavaScript. Узагальнене порівняння розглянутих мов відповідно до вимог представлено у таблиці:

Таблиця 2.1

Порівняння мов програмування

Мова програму-вання	Компіляція у JS	Гнучка та масштабована парадигма	Статична система типізації	Зручність використання
TypeScript	+	+	+	+
CoffeeScript	+	+	–	+/–
Scala	+	+	+	–
Kotlin	+	+	+	–

Після проведеного аналізу зроблено висновок, що тільки одна мова програмування задовольняє повністю усі необхідні вимоги – мова програмування TypeScript. Це поясняються тим, що її найпростіше використовувати для впровадження системи типізації у мову JS та є можливість впроваджувати її поступово, бік-о-бік із JS.

3. СТРУКТУРНО-АЛГОРИТМІЧНА ОРГАНІЗАЦІЯ БІБЛІОТЕКИ

3.1. Перелік вимог до програмних засобів

З огляду на поставлене завдання реалізації бібліотеки для створення сервера авторизації виникає необхідність у визначенні вимог до програмного забезпечення з їх наступним аналізом. У даному підрозділі описано проблеми, які вирішує проєкт, цілі та відповідні їм результати, які мають бути досягнуті після завершення процесу розробки.

3.1.1. Аналіз проблем, цілей та результатів роботи бібліотеки

Найбільш суттєвими проблемами у існуючих рішеннях є те, що вони тривалий час не підтримуються та використовують застарілі підходи мови JavaScript, що у сукупності призводить до наступних проблем:

1. Відсутність оновлень згідно нових вимог стандартів та вимог безпеки.
2. Поступове збільшення кількості помилок у існуючому коді, оскільки він не адаптований до нових підходів та способів використання.
3. Підвищена складність впровадження власних стратегій авторизації.

Окрім цього, відзначаються такі проблеми, як:

1. Відсутність декларацій типів для мови програмування TypeScript. Через це ускладнюється використання бібліотек, оскільки необхідно розбиратися у вихідному коді, аби зрозуміти, які аргументи використовує той чи інший метод.
2. Відсутність бібліотек для інтеграції з сучасними фреймворками (наприклад, Nest.js). Як наслідок, цієї проблеми, зростає час на інтеграцію бібліотек до новітніх фреймворків.

Отже, об'єднавши дані проблеми, ми отримуємо наступне дерево проблем (рис. 3.1).

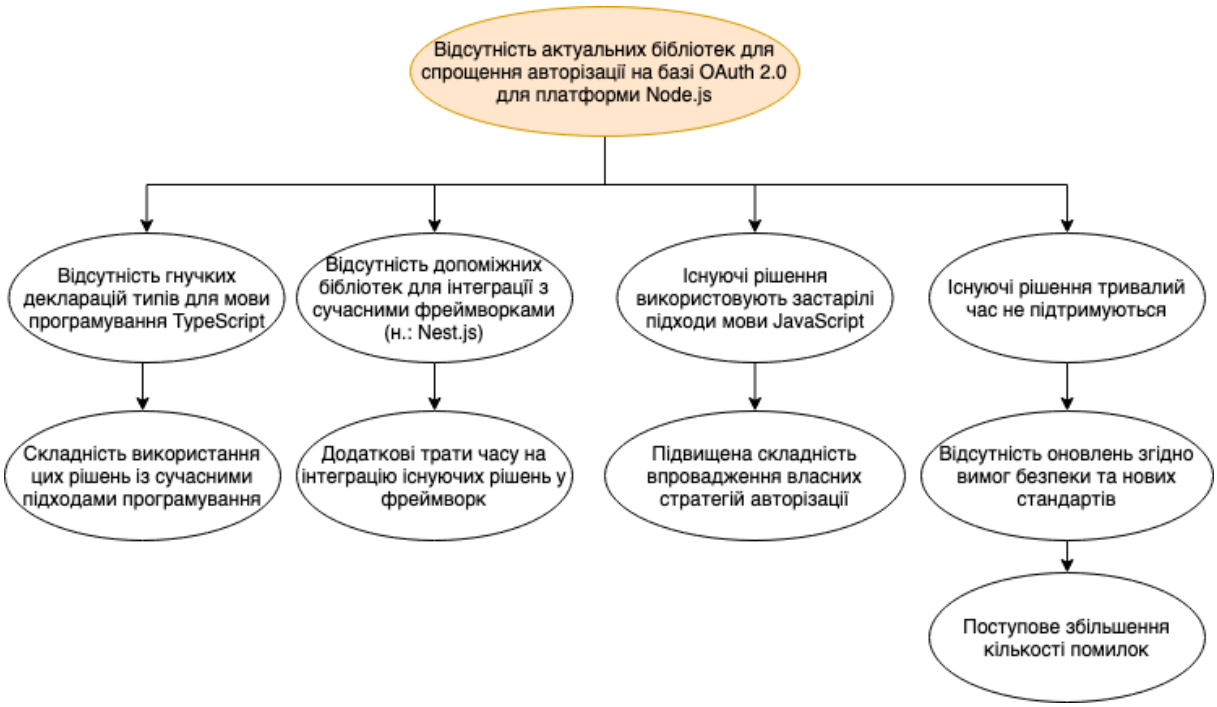


Рис. 3.1. Дерево проблем

З даних, що були отримані після дослідження, було сформовано цілі, що мають бути досягнуті у кінцевій версії розробки. Головною метою є створення універсальної бібліотеки, яку можна буде використовувати і з мовами програмування JavaScript та TypeScript. Ця бібліотека повинна забезпечувати надійність та простоту використання за рахунок мови програмування TypeScript в якості головної мови її коду та повинна бути написана таким чином, щоб забезпечити максимальну гнучкість при реалізації власної стратегії авторизації.

Іншою ж метою є створення допоміжної версії бібліотеки, що буде забезпечувати просту інтеграцію із фреймворком Nest.js, що допоможе зменшити часові витрати при адаптації існуючої бази коду на цьому фреймворку для використання із цією бібліотекою. Об’єднавши дані цілі, ми отримаємо дерево цілей (рис. 3.2).

Рис. 3.2. Дерево цілей

Взявши за основу дані цілі, утворимо дерево результатів (рис. 3.3), що відображає кінцеві цілі, які мають бути досягнені в процесі розроблення продукту.

Рис. 3.3. Дерево результатів

Отже, провівши дослідження проблематики та сформувавши цілі та очікувані результати продукту, сформовано ряд вимог, яким має відповідати розроблюване забезпечення.

3.1.2. Вимоги до програмної бібліотеки для створення сервера авторизації

Відповідно до загальновживаних стандартів поняття «вимоги» можна охарактеризувати як умови, за якими система або її компоненти відповідають стандартам, або можливості, необхідні користувачеві для досягнення поставлених цілей або вирішення проблем. Окрім цього, вимоги можуть визначитись як задокументоване представлення цих умов та можливостей.

Зазвичай, вимоги визначаються на стадії проєктування програмного забезпечення, але вони також можуть бути використані у процесі тестування ПЗ, оскільки даний процес перевірки якості програмного забезпечення ґрунтується на зазначених вимогах. Можна виділити наступні етапи розробки вимог:

1. З'ясування вимог – збір та уточнення потреб зацікавлених сторін проєкту.
2. Структуризація вимог – перестановка вимог відповідно до важливості, терміновості та зручності.
3. Документація вимог – занесення вимог до специфікації.
4. Перевірка вимог на правильність.

Розглянемо більш детально кожен з цих етапів.

Почнемо із етапу з’ясування вимог. Коли клієнт звертається до організації для отримання потрібного продукту, він висуває неточне уявлення про те, які функції очікуються від програмного забезпечення. Посилаючись на цю інформацію, аналітики роблять детальне дослідження про те, чи можливо розробити бажану функціональність системи. У цьому дослідженні аналізується, чи можна розробити програмний продукт з точки зору програмної реалізації, внеску проєкту в організацію, обмеження витрат та відповідно до цінностей та цілей організації. Це дослідження фокусується на таких аспектах проєкту та продукту, як зручність використання, можливість підтримки у майбутньому, швидкість та підтримка інтеграції.

У результаті цього етапу має бути утворений звіт про техніко-економічне обґрунтування, який повинен містити відповідні зауваження та рекомендації щодо того, чи слід здійснювати проєкт.

Якщо було вирішено здійснювати проєкт, то починається наступний етап збору вимог – збір вимог користувача. Для того, щоб зрозуміти, які функції найбільш важливі та які функції очікують звичайні користувачі від розроблюваного програмного забезпечення, аналітики та інженери збирають вимоги від користувачів. Ці вимоги вони отримують через комунікацію із клієнтом та майбутніми користувачами задля отримання від них ідей щодо продукту. Існує декілька способів виявлення вимог.

Таблиця 3.1

Способи виявлення вимог

Назва способу виявлення вимоги	*Опис способу*
Інтерв'ю	Інтерв'ю є сильним середовищем для збирання вимог. Організація може проводити кілька типів інтерв'ю, таких як: • структуровані (закриті) інтерв'ю, де будь-яка інформація, яку потрібно зібрати, вирішується заздалегідь. Такі інтерв’ю чітко слідують заданому плану питанням обговорення; • неструктуровані (відкриті) інтерв'ю, де інформація для збору не визначається заздалегідь, а тому є гнучкою та менш упередженою; • усні співбесіди; • письмові співбесіди; • інтерв'ю один на один, яке проводиться тільки між двома особами; • групові інтерв'ю, які проводяться між групами учасників. Вони допомагають розкрити будь-яку відсутню вимогу, оскільки в них приймає участь велика кількість людей;

Продовження табл. 3.1

	• опитування, яке може проводитись у очній або заочній формі. Організація може проводити опитування серед зацікавлених сторін різного рівня, задля виявлення їх очікування від майбутньої системи.
Анкети	Документ із заздалегідь визначеним набором об’єктивних питань та варіантами відповіді, що передається всім зацікавленим сторонам. Недоліком цієї методики є те, що якщо варіант анкети не згадується в анкеті, питання може залишатися без нагляду.
Аналіз завдань	Команда інженерів та розробників може проаналізувати набір операцій, для якої потрібна нова система. Якщо у клієнта вже є певне програмне забезпечення для виконання певної операції, воно вивчається і збираються вимоги запропонованої системи.
Аналіз домену	Кожне програмне забезпечення належить до певного домену. Експерти в цій галузі можуть допомогти проаналізувати вимоги до розробки.
Мозковий штурм	Проводяться неофіційні дебати між різними зацікавленими сторонами, і всі їхні дані записуються для подальшого аналізу потреб.
Прототипування	Побудова інтерфейсу користувача без додавання детальної функціональності призначеного програмного продукту. Це допомагає краще розуміти вимоги. Якщо на стороні клієнта не встановлено програмне забезпечення для довідки розробника і клієнт не знає про власні вимоги, розробник створює прототип, заснований на початкових вимогах. Прототип демонструється клієнтові, для отримання зворотного зв'язку. Відгуки клієнтів служать вхідним фактором для збору вимог.

Продовження табл. 3.1

Спостереження	Команда експертів відвідує організацію або робоче місце клієнта. Вони спостерігають фактичну роботу поточних встановлених систем. Вони спостерігають за робочим процесом на стороні клієнта і тим, як вирішуються існуючі проблеми. Команда сама робить висновки, які допомагають формувати вимоги до програмного забезпечення.

Після того, як було зібрано усі вимоги, системним аналітиком утворюється документ зі специфікацією вимог до програмного забезпечення (Software Requirement Specification або SRS). Документ зі специфікацією вимог визначає, як програмне забезпечення буде взаємодіяти з апаратним забезпеченням, зовнішніми інтерфейсами, швидкість роботи, час відгуку системи, портативність, відмовостійкість та безпеку програмного забезпечення. Вимоги від клієнта надходять природною мовою, проте системний аналітик зобов'язаний документувати вимоги технічною мовою, щоб їх могла зрозуміти команда розробки програмного забезпечення. Після створення даного документу, розпочинається етап перевірки вимог на правильність.

Етапи перевірки вимог на правильність (валідації) вимог є важливим етапом при розробці вимог, оскільки він дозволяє відкинути незаконні або непрактичні рішення, які були запропоновані на попередніх етапах розробки. Таких непрактичних рішень варто уникати, оскільки вони можуть сильно збільшити вартість продукту. Для перевірки вимог на коректність можна використати наступний набір критеріїв:

- можливість практичної реалізації;
- відповідність функціональності та домену програмного забезпечення;
- наявність неточностей;
- повнота вимог;

- можливість демонстрації.

Загалом вимоги до програмного забезпечення слід класифікувати на дві категорії: функціональні та нефункціональні.

Функціональні вимоги – це вимоги, пов'язані з функціональним аспектом програмного забезпечення, вони визначають функціональність всередині програмної системи. Приклади:

- користувач повинен мати можливість надсилати будь-який звіт керівництву;
- користувачів можна розділити на групи, а групам можуть бути надані окремі права;
- програмне забезпечення розроблено, зберігаючи зворотню сумісність.

Нефункціональні вимоги – це вимоги, не пов'язані з функціональним аспектом програмного забезпечення, вони є неявними або очікуваними характеристиками програмного забезпечення. Нефункціональні вимоги включають такі категорії:

- безпека;
- ведення журналів;
- збереження інформації;
- конфігурація;
- ефективність;
- вартість;
- взаємодія із іншими програмами;
- відновлення роботи після збоїв;
- універсальний доступ.

Вимоги розподіляються за логікою на такі 4 категорії:

- обов’язкові – вимоги, без яких програмне них не можна вважати оперативним;

- можливі – вимоги, які відповідають за покращення функціональності програмного забезпечення;
- бажані – вимоги, з якими програмне забезпечення все ще може належним чином функціонувати;
- необов’язкові – вимоги, які не відповідають жодним цілям програмного забезпечення.

Під час розробки програмного забезпечення необхідно впровадити обов’язкові вимоги, можливі є предметом дискусій із зацікавленими сторонами та запереченням, тоді як бажані та необов’язкові можна зберігати для оновлення програмного забезпечення.

Отже, після процесу аналізу було визначено набір вимог, яким повинна відповідати програмна бібліотека. Дані вимоги було сформовані на основі комунікації з користувачами вже існуючих бібліотек, а отже, можна зробити висновок, що дані вимоги у найбільший ступінь відображають реальні задачі користувачів, вирішення яких вони очікують від бібліотек такого типу.

Вимоги до основної бібліотеки авторизації наведені у таблиці нижче.

Таблиця 3.2

Перелік вимог до основної версії бібліотеки

Код вимоги	Зміст вимоги	Опис вимоги
F1	Вимоги до оформлення	Методи бібліотеки повинні мати інтуїтивно зрозумілі для користувача бібліотеки назви.
F2	Вимоги до способу реалізації	Бібліотека повинна бути реалізована на мові програмування TypeScript, містити повне покриття типами задля найзручнішого її використання з існуючою базою коду.

Продовження табл. 3.2

F3	Вимоги до гнучкості	Бібліотека повинна бути незалежною від сторонніх фреймворків задля забезпечення максимальної гнучкості
F4	Вимоги до залежностей	Бібліотека повинна мати якнайменше залежностей від сторонніх бібліотек задля зменшення кількості потенційних вразливостей
F5	Вимоги до функціоналу	Бібліотека повинна містити увесь необхідну функціональність для реалізації стандартів фреймворку авторизації RFC 6749 та RFC 6750.
F6	Вимоги до можливостей інтеграції	Модулі основної бібліотеки повинні бути реалізовані у такий спосіб, щоб бібліотеку можна було легко під’єднати до власного контейнеру впровадження залежностей.
F7	Вимоги до кастомізації бібліотеки	Бібліотека має надавати можливості створення власних типів грантів, а не тільки стандартних.
F8	Вимоги до підтримки сховищ даних	Бібліотека повинна надавати зручний інтерфейс для підключення сховищ даних різного типу.
F9	Вимоги до портативності	Бібліотека повинна бути опублікована на реєстрі пакетів NPM для того, щоб її можна було легко встановити у будь-яку користувацьку програму.
F10	Вимоги до підтримки	Бібліотека повинна бути опублікована на сервісі для хостингу проєктів GitHub для того, щоб кожен бажаючий мав можливість розвивати її

Вимоги до супутньої бібліотеки авторизації для зручної інтеграції з фреймворком Nest.js наведені у таблиці нижче.

Таблиця 3.3

Перелік вимог до супутньої версії бібліотеки

Код вимоги	Зміст вимоги	Опис вимоги
F1	Вимоги до оформлення бібліотеки	Методи бібліотеки повинні мати інтуїтивно зрозумілі для користувача бібліотеки назви.
F2	Вимоги до способу реалізації бібліотеки	Бібліотека повинна бути реалізована на мові програмування TypeScript, містити повне покриття типами задля якнайшвидшого її використання з існуючою базою коду.
F3	Вимоги до залежностей бібліотеки	Бібліотека повинна мати якнайменше залежностей від сторонніх бібліотек задля зменшення кількості потенційних вразливостей.
F4	Вимоги до способу інтеграції	Модулі основної бібліотеки повинні бути реалізовані у такий спосіб, щоб можна бібліотеку було легко використовувати із контейнером для впровадження залежностей фреймворку Nest.js.
F5	Вимоги до функціоналу бібліотеки	Бібліотека повинна містити увесь необхідний функціональність для реалізації стандартів фреймворку авторизації RFC 6749 та RFC 6750.
F6	Вимоги до підтримки сховищ даних	Бібліотека повинна надавати зручний інтерфейс для підключення сховищ даних різного типу.
F7	Вимоги до кастомізації бібліотеки	Бібліотека має надавати можливості створення власних типів грантів, а не тільки стандартних.
F8	Вимоги до портативності	Бібліотека повинна бути опублікована на реєстрі пакетів NPM для того, щоб її можна було легко встановити у будь-яку користувацьку програму.

Продовження табл. 3.3

F9	Вимоги до підтримки бібліотеки	Бібліотека повинна бути опублікована на сервісі для хостингу проєктів GitHub для того, щоб кожен бажаючий мав можливість розвивати її

В результаті проведеного дослідження було встановлено вимоги до головних частин розроблюваного продукту (основної та супутньої бібліотеки) та створено реєстри з вимогами до кожної з них.

3.2. Опис розроблюваної системи

Після проведення аналізу поточних рішень та збору вимог до нового рішення було утворено структурну схему до розроблюваної програмної бібліотеки. Ця бібліотека складається з таким модулів:

- Parameters Validator;
- Grant Type Manager;
- Grant Type;
- Scope Manager;
- Token Storage.

Взаємодія між даними модулями описана у діаграмі нижче (рис. 3.4).

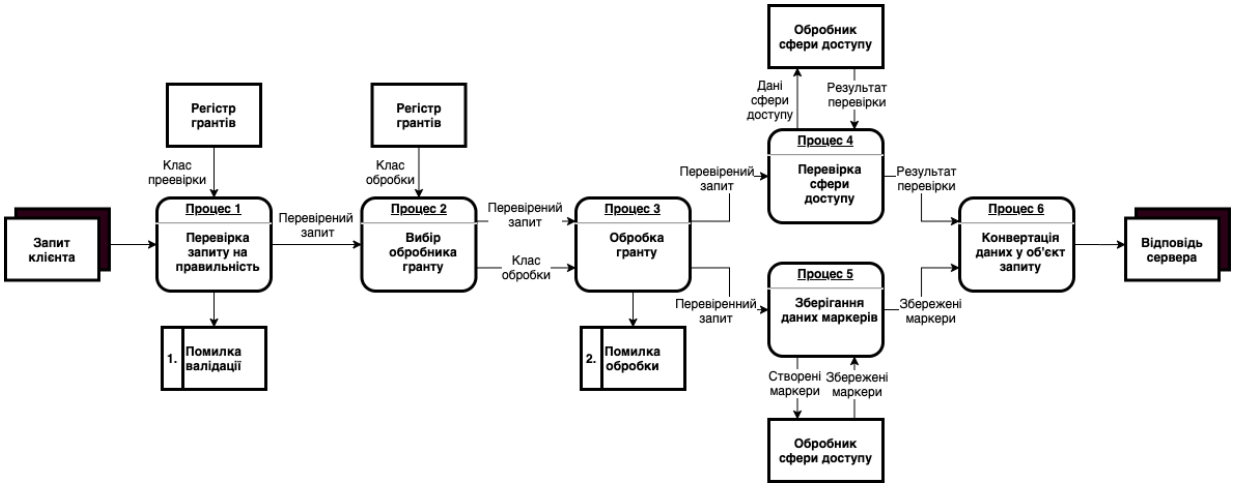


Рис. 3.4. Діаграма внутрішньої реалізації бібліотеки

3.2.1. Опис модулю перевірки запитів на коректність

Модуль, що відповідає за перевірку запитів користувача на правильність (валідацію) має назву Request Validator. Даний модуль приймає на вхід запит від користувача, після чого перевіряє отримані дані на коректність. Перевірка даних на коректність є гнучкою – модуль Request Validator автоматично обирає необхідний метод перевірки даних в залежності від типу гранту в отриманих даних. Якщо цей модуль не може знайти відповідності між наявними типами гранту та переданим типом у запиті, користувачу повертається помилка. Якщо ж на якомусь з етапів

валідації знаходиться невідповідність наявним правилам, то користувачу також повертається помилка. Коректність даних для кожного з типу грантів визначається згідно специфікації RFC 6750.

Даний модуль також має можливість перевірки користувацьких типів гранту. Це є можливим, оскільки всередині бібліотеки реєстрація грантів інкапсульована таким чином, що інші модулі не прив’язуються до деталей реєстрації, а тільки використовують цей модуль. Завдяки такому підходу, код даного модуля значно спрощується, навіть при наявності підтримки валідації користувацьких типів гранту. Для використання цієї можливості користувач має надати власну реалізацію метода валідації при реєстрації типу гранту. Варто відмітити, що код помилки буде однаковий для усіх типів гранту.

3.2.2. Опис менеджеру вибору типу гранту

Grant Type Manager – це модуль, який відповідає за вибір типу гранту для подальшої роботи бібліотеки. Цей модуль містить реалізацію логіки для зіставлення строки, яка ідентифікує певний тип гранту з класом усередині бібліотеки, який надає для нього відповідну логіку, та для реєстрації власних типів гранту. Саме в цьому модулі містяться необхідні базові абстракції, на основі яких будується специфікація типів грантів відповідно до стандарту про фреймворк авторизації OAuth 2.0 та для користувацьких типів грантів. Цей модуль вважає надані дані коректними, оскільки вони були валідовані на попередньому етапі, та не робить додаткових перевірок щодо їх коректності.

3.2.3. Опис модулю типу гранту

Центральним модулем у даній розробці є модуль типу гранту, який має назву Grant Type. Цей модуль об’єднує логічні блоки, які реалізують безпосередньо логіку аутентифікації та індентифікації користувача відповідно до специфікації обраного типу гранту. Саме в цьому модулі

містяться класи, які описують абстрактну логіку перевірки особистості користувача та класи, які реалізують цю логіку. Назви класів цього модуля відповідають типу гранта, логіку якого вони описують: AuthorizationCode, ClientCredentials, Implicit, Password та RefreshToken. Даний модуль представляє собою реалізацію найбільшої частини стандарту RFC 6750. Варто відмітити, що взаємодія, описана у цій частині бібліотеки, є фінальною частиною по відношенню до користувача, тобто виходом із цього модулю є коректна відповідь з усіма необхідними даними або помилка у випадку, якщо щось пішло не так. Даний модуль комунікує з двома іншими модулями: Scope Manager та Token Storage.

3.2.4. Опис модулю сфери обмеження дій користувача

Відповідно до стандарту, обмеження дій користувача реалізується через механізм надання йому певної сфери дій (scope). Перелік запитуваних сфер надається із запитом до сервера, а перевірка відбувається вже сумісно з логікою гранта за допомогою абстракцій модуля Scope Manager. Вимогою до цієї частини бібліотеки є перевірка наданих сфер дії відповідно до специфікації та надання повідомлення, що буде зрозумілим для користувача, у випадку, якщо були надані не валідні дані. Методи даного модуля викликаються всередині модуля Grant Type.

3.2.5. Опис модулю збереження токенів

Для встановлення особистості та реєстрації користувача у стандарті використовуються такі проміжні ідентифікатори:

- Access Token, або ідентифікатор доступу – короткочасний ідентифікатор, який є багаторазовим і використовується для ідентифікації користувача при взаємодії з системою;
- Refresh Token, або ідентифікатор відновлення доступу – довгостроковий ідентифікатор, який може бути використаний

тільки один раз для оновлення сесії та отримання нової пари ідентифікаторів.

Після закінчення строку дії ідентифікатору доступу користувачу необхідно зробити спеціальний запит на оновлення сесії доступу, використовуючи спеціальний ідентифікатор оновлення доступу, який був отриманий після початку сеансу роботи, або після попереднього оновлення доступу. Модуль Token Storage є достатньо маленьким, а його робота обмежується наданням інтерфейсу для зберігання цієї пари токенів. Даний інтерфейс має бути реалізований користувачем бібліотеки – такий підхід дозволяє розроблюваному рішенню бути незалежним від сховища даних, у якому користувач бажає зберігати інформацію.

3.3. Логічна структура розроблюваної системи

3.3.1. Ролі для фреймворку авторизації

OAuth визначає чотири ролі:

- власник ресурсу – це сутність, здатна надавати доступ до захищеного ресурсу. Коли власником ресурсу є людина, вона називається кінцевим користувачем;
- сервер ресурсів – це сервер, на якому розміщені захищені ресурси, здатний приймати та відповідати на запити для доступу до захищених ресурсів. Використовує ідентифікатори доступу (access tokens);
- клієнт – додаток, що робить захищені запити на ресурси від імені власника ресурсу та з його дозволом. Термін "клієнт" не передбачає конкретних характеристик реалізації (наприклад, незалежно від того, виконується програма на сервері, настільному ПК чи іншому пристрої);
- сервер авторизації – сервер, який після успішної видачі токена доступу клієнту, проводить аутентифікацію власника ресурсу для отримання дозволу.

Взаємодія між сервером авторизації та сервером ресурсів виходить за межі цієї специфікації. Сервер авторизації може бути тим же сервером, що і сервер ресурсів, або окремим об'єктом. Один сервер авторизації може видавати маркери доступу, прийняті користувачем декілька серверів ресурсів.

3.3.2. Логіка авторизації

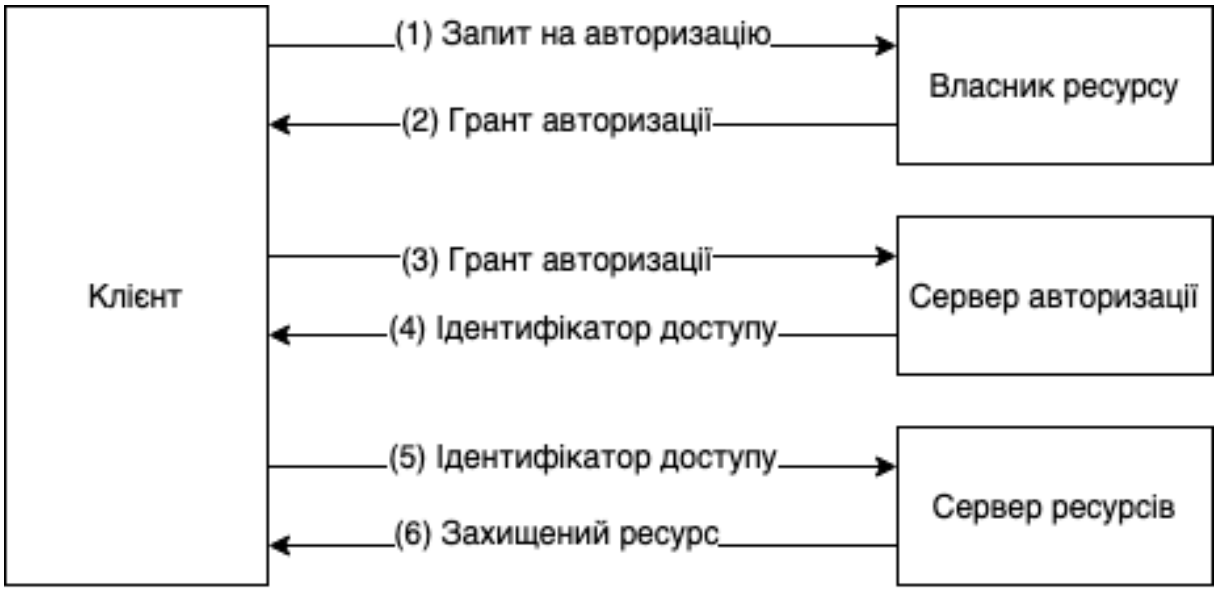


Рис. 3.5. Загальна логіка авторизації за протоколом OAuth 2.0

Логіка авторизації за допомогою протоколу OAuth 2.0, проілюстрована на рисунку, описує взаємодію чотирьох ролей і включає такі етапи:

1. Клієнт вимагає дозволу від власника ресурсу. Запит на авторизацію можна зробити безпосередньо власнику ресурсу, або через сервер авторизації в якості посередника.
2. Клієнт отримує грант авторизації, що представляє собою авторизацію до власника ресурсу, виражену за допомогою одного з чотирьох типів грантів зі специфікації або за допомогою власного типу гранту. Тип гранту авторизації залежить від методу, який використовується клієнтом для

запиту авторизації та типів, підтримуваних сервером авторизації.

3. Клієнт запитує ідентифікатор доступу шляхом аутентифікації на сервері авторизації та подання гранту авторизації.
4. Сервер авторизації аутентифікує клієнта та перевіряє грант авторизації, і якщо він дійсний, видає ідентифікатор доступу.
5. Клієнт надсилає запит до захищеного ресурсу із сервера ресурсів і аутентифікується за допомогою ідентифікатору доступу.
6. Сервер ресурсів перевіряє ідентифікатор доступу та, якщо він дійсний, виконує запит.

Кращим для клієнта способом отримання дозволу авторизації від власника ресурсу (зображеного на кроках 1 та 2) є використання сервера авторизації в якості посередника.

3.3.3. Типи грантів авторизації

Грант авторизації – це довірені дані, що представляють авторизацію від власника ресурсу на доступ до його захищених ресурсів, який використовується клієнтом для отримання ідентифікатору доступу. У специфікації RFC 6750 визначено чотири типи грантів – код авторизації, неявний тип гранту, облікові дані власника ресурсу та облікові дані клієнта – а також механізм розширення для визначення додаткових типів. Розглянемо детальніше кожен з основних типів грантів.

Код авторизації

Код авториації отримується з використанням сервера авторизації в якості посередника між клієнтом та власником ресурсу. Замість того, щоб запитувати авторизацію безпосередньо від власника ресурсу, клієнт спрямовує останнього на сервер авторизації, який, в свою чергу, повертає власника ресурсу назад до клієнта з кодом авторизації. Перш ніж повернути власника ресурсу назад до клієнта з кодом авторизації, сервер авторизації

аутентифікує його та отримує авторизацію. Оскільки власник ресурсу підтверджує аутентифікацію лише на сервері авторизації, облікові дані власника ресурсу ніколи не надаються клієнтові.

Код авторизації надає декілька важливих переваг безпеки, таких як можливість аутентифікації клієнта, а також передача ідентифікатора доступу безпосередньо клієнту, не передаючи його через агент власника ресурсу та виключаючи можливість розкривати цей код іншим, у т.ч. власнику ресурсу.

Неявний грант

Неявний грант – це спрощений алгоритм коду авторизації, оптимізований для клієнтів на базі веб-браузерів, що написані на скриптовій мові, такій як JavaScript. У алгоритмі неявного типу гранта замість авторизаційного коду клієнту безпосередньо видається ідентифікатор доступу (як результат авторизації власника ресурсу). Тип дозволу є неявним, оскільки проміжні облікові дані (наприклад, код авторизації) не видаються (і пізніше використовуються для отримання ідентифікатора доступу). Під час видачі ідентифікатора доступу у алгоритмі неявного типу гранта сервер авторизації не аутентифікує клієнта. В деяких випадках ідентифікація клієнта може бути підтверджена за допомогою URI перенаправлення, яке використовується для доставки ідентифікатора доступу клієнту. Ідентифікатор доступу може бути викритий власнику ресурсу або іншим додаткам, що мають доступ до агента користувача ресурсу. Неявні гранти покращують швидкість роботи деяких клієнтів (наприклад, веб-додатків), оскільки це зменшує кількість перенаправлень, необхідних для отримання ідентифікатора доступу. Однак варто пам'ятати, що ця зручність приносить із собою певні наслідки для безпеки, особливо коли підтримується авторизація за допомогою коду авторизації.

Ідентифікатори пароля власника ресурсу

Вхідні дані пароля власника ресурсу (тобто ім'я користувача та пароль) можуть використовуватися безпосередньо як дозвіл на отримання авторизації для отримання ідентифікатора доступу. Ці дані слід використовувати лише тоді, коли між власником ресурсу та клієнтом є високий ступінь довіри (наприклад, клієнт є частиною операційної системи пристрою або високопривілейованою програмою) та коли інші типи дозволів на надання авторизації недоступні. Незважаючи на те, що цей тип грантів вимагає прямого доступу клієнта до облікових даних власника ресурсу, облікові дані власника ресурсу використовуються для одного запиту та обмінюються на ідентифікатор доступу. Цей тип гранту може усунути потребу клієнта зберігати облікові дані власника ресурсу для подальшого використання шляхом обміну облікових даних на ідентифікатор відновлення доступу або ідентифікатора отримання доступу.

Клієнтські дані

Клієнтські дані (або інші форми аутентифікації клієнта) можуть використовуватися як дозвіл авторизації, коли область авторизації обмежена захищеними ресурсами, що знаходяться під контролем клієнта, або захищеними ресурсами, попередньо упорядкованими з сервером авторизації. Клієнтські дані зазвичай використовуються як дозвіл на авторизацію, коли клієнт діє від свого імені (клієнт також є власником ресурсу) або вимагає доступу до захищених ресурсів на основі авторизації, попередньо узгодженої з сервером авторизації.

3.3.4. Ідентифікатор доступу

Токени доступу – це облікові дані, які використовуються для доступу до захищених ресурсів. Ідентифікатор доступу – це рядок, що представляє авторизацію, видану клієнту. Рядок зазвичайне має особливого сенсу для клієнта. Токени представляють конкретні обсяги та тривалість доступу, що надаються власником ресурсу та застосовуються сервером ресурсів та

сервером авторизації. Токен може позначати ідентифікатор, який використовується для отримання інформації про авторизацію, або може самостійно містити інформацію авторизації, яку можна веріфікувати (тобто рядок, що складається з деяких даних і підпису). Для того, щоб клієнт міг використовувати ідентифікатор, можуть знадобитися додаткові облікові дані аутентифікації. Ідентифікатор доступу забезпечує рівень абстракції, замінюючи різні конструкції авторизації (наприклад, ім'я користувача та пароль) одним токеном, зрозумілим для серверу ресурсів. Ця абстракція дає змогу видавати ідентифікатору доступу більш обмежуючими, ніж грант авторизації, який використовується для їх отримання, а також усуває необхідність додаткової підтримки великого обсягу методів авторизації сервером ресурсів. Ідентифікатори доступу можуть мати різні формати, структури та способи використання (наприклад, криптографічні властивості), які залежать від вимог безпеки серверних ресурсів.

3.3.5. Ідентифікатор оновлення доступу

Токен для оновлення доступу – це облікові дані, які використовуються для отримання ідентифікаторів доступу. Токени оновлення видаються клієнту сервером авторизації та використовуються для отримання нового ідентифікатора доступу, коли поточний ідентифікатор доступу стає недійсним або закінчується, або для отримання додаткових токенів доступу з однаковим чи більш конкретним рівнем доступу (токени доступу можуть мати коротший термін служби та менше дозволів, ніж дозволено власником ресурсу). Видача токена оновлення є необов'язковою та відбувається на розсуд сервера авторизації. Якщо сервер авторизації видає токен оновлення, він включається при видачі ідентифікатора доступу (тобто, крок (4) на рисунку 1). Токен оновлення – це рядок, що представляє авторизацію, надану клієнту власником ресурсу. Рядок зазвичай не містить у собі додаткової інформації для клієнта. Токен позначає ідентифікатор, який використовується для отримання інформації про авторизацію. На відміну

від токенів доступу, токени оновлення призначені для використання лише з серверами авторизації і ніколи не надсилаються на сервери ресурсів.

Алгоритм роботи із токенами отримання та поновлення доступу представлений на рис. 3.6.

Рис. 3.6. Діаграма роботи маркерів отримання та оновлення доступу

На рис. 3.6 зображено такі етапи взаємодії клієнта та серверу авторизації й серверу ресурсів:

1. Клієнт запитує ідентифікатор доступу шляхом аутентифікації на сервері авторизації та подання дозволу авторизації.
2. Сервер авторизації аутентифікує клієнта і перевіряє надання дозволу авторизації, і якщо він дійсний, видає токен доступу та токен оновлення.
3. Клієнт робить захищений запит на ресурс на сервері ресурсів, надаючи токен доступу.
4. Сервер ресурсів перевіряє токен доступу та, якщо він дійсний, виконує запит.

5. Кроки 3 і 4 повторюються, поки термін токену доступу не закінчиться. Якщо клієнт знає, що токен доступу закінчився, він переходить до кроку 7; в іншому випадку він робить ще один запит на захищений ресурс.
6. Оскільки токен доступу недійсний, сервер ресурсів повертає недійсну помилку маркера.
7. Клієнт запитує новий токен доступу шляхом аутентифікації на сервері авторизації та подання токена оновлення. Вимоги до аутентифікації клієнта засновані на типі клієнта та на політиці сервера авторизації.
8. Сервер авторизації аутентифікує клієнта і перевіряє токен оновлення, і якщо він дійсний, видає новий токен доступу (і, необов'язково, новий токен оновлення).

Отже, було зроблено істотний аналіз розробки даного проєкту. Досліджені проблематика, цілі та результати, які потрібно досягти в ході розробки бібліотеки, та на основі цього сформовано ряд загальних вимог до бібліотеки, деякі з яких розглянуті у деталях. Також була описана логічна структура додатку на високому рівні абстракції та алгоритмічне підґрунтя кожного модулю. Визначено особливості роботи кожної частини програми в основних сценаріях разом із вхідними та вихідними даними.

4. ОПИС РЕАЛІЗАЦІЇ ПРОГРАМНИХ ЗАСОБІВ

4.1. Опис структур даних

Визначення та реалізація структур даних – центральні задачі, які необхідно виконати під час розробки програмного забезпечення. Правильно визначені структури даних – гарант легкості розширення та підтримки програмних рішень, адже вони грають ключову роль у забезпеченні коректної роботи кожного з розроблюваних програмних засобів.

В рамках даної роботи було розглянуто та порівняно мови програмування, необхідні для розроблення запропонованого програмного рішення. Результатом аналізу стало обрання мови програмування TypeScript, оскільки вона найкраще підходить для вирішення описаних проблем. Варто відмітити, що ця мова є строгою надмножиною над мовою програмування JavaScript, що означає повну зворотну сумісність з останньою (код на мові JavaScript є коректним кодом на мові TypeScript). І, не дивлячись на те, що механізм типізації, що надається обраною мовою, не є статичною типізацією у класичному розумінні (як, наприклад, у мові програмування Java), він дозволяє використовувати певні динамічні конструкції, які недоступні у компільованих мовах із класичною статичною типізацією. Обрана філософія даної мови має наступні переваги:

- можливість явного визначення типів із підтримкою парадигми узагальненого програмування;
- наявність засобів рефлексії, що додає гнучкості даній мові;
- підтримка класичних абстракцій: класів, інтерфейсів, перелічуваних типів даних тощо;
- підтримка підключення модулів;
- наявність додаткових конструкцій (наприклад, декораторів) та спеціального синтаксичного сахару для пришвидшення

розробки (наприклад, об’ява членів класу у аргументах конструктора).

На думку спільноти, наведені переваги даної мови позитивно впливають на використанні її у розробленні ПЗ. Так підтримка типізації дозволяє пришвидшити розробку, оскільки із нею відпадає необхідність у вивченні коду зовнішніх модулів – типи у методах та змінних фактично надають примітивну документацію до них та дозволяють уникнути певного кола помилок ще під час компіляції коду.

Створення самої бібліотеки потребує розуміння таких фундаментальних концепцій програмної платформи Node.js як цикл подій. Дана програма платформа використовує асинхронну, неблокуючу модель програмування. Асинхронна модель програмування передбачає те, що програма виконується не послідовно, строчка за строчкою, а відкладено. Наприклад, коли якщо в коді викликається дві асинхронні операції, то друга не буде очікувати завершення першої, а результат кожна з них отримує після завершення виконання. Саме цю модель реалізує цикл подій. Він оперує такими поняттями як черга подій, пул потоків, стек виконання та інші. Ця абстракція визначає реакцію на події та повернення результату після її завершення.

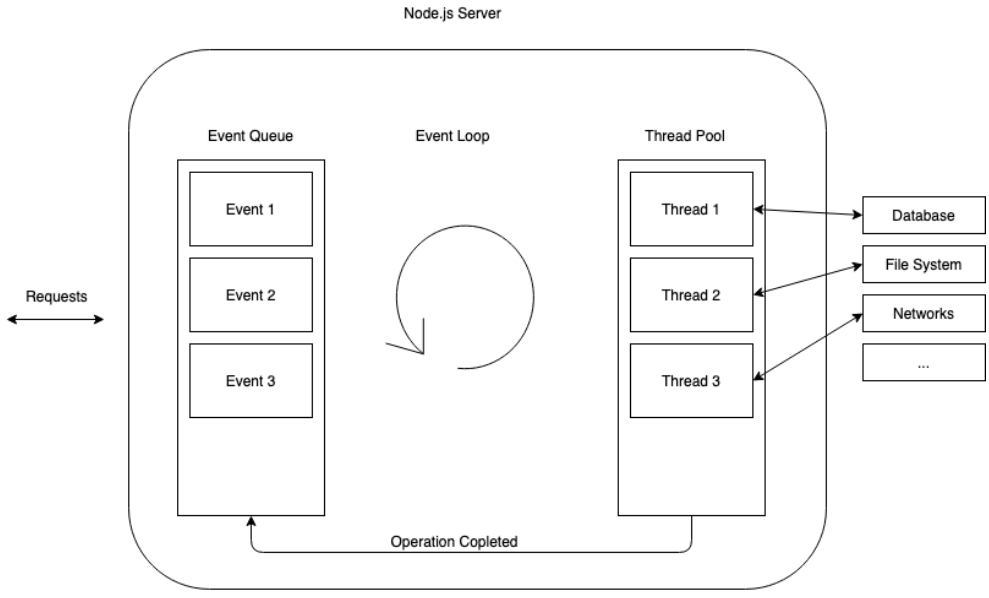


Рис. 4.1. Схема роботи циклу подій платформи Node.js

Для того, щоб коректно оброблювати результати у асинхронній мові програмування JavaScript кожна асинхронна функція приймає іншу функцію як аргумент (так званий callback). Велика кількість вкладених викликів асинхронних функцій призводить до так званого callback-hell, коли розробка сильно ускладнюється та читаність коду сильно страждає. Для запобігання таких ситуацій у останніх стандартах ECMAScript були уведені інші засоби для роботи з такими функціями – абстракція Promise, яка надає можливість створювати ланцюжки із асинхронних функцій. Й хоча цей підхід є досить потужним, його некоректне використання може призвести до аналогічної проблеми – promise hell. Для подолання цієї проблеми у стандарті ECMAScript були запропоновано використання спеціальних ключових слів – async та await, поєднання яких дозволяє писати асинхронний код, який виглядає як звичайний послідовний.

При розробці на даній програмній платформі варто пам’ятати про збереження сумісності із різними версіями, оскільки мова програмування JavaScript розвивається досить стрімко. Задля зручної міграції між стандартами даної мови використовується такий інструмент як Babel. Babel – це інструмент, який дозволяє транслювати один стандарт мови JavaScript у інший й широко використовується під час розробки клієнтської частини веб-сайтів, оскільки вендори веб-браузерів досить повільно додають функції із останніх стандартів.

Всі структури даних, використовувані у даній розробці, можна розділити на два великих класи:

- внутрішні структури даних – це структури даних, область видимості яких обмежуються самою бібліотекою. Ці структури даних використовуються під час комунікації внутрішніх модулів;
- користувацькі структури даних – це структури даних, які надають інтерфейс для взаємодії із розроблюваною бібліотекою. Це структури даних, які дозволяють користувачу зрозуміти, які методи варто використовувати для коректної взаємодії із

розробкою, які аргументи варто надавати на вхід та які результати роботи можна отримати на виході.

Варто відмітити, що дана бібліотека досить глибоко інтегрована із засобами мови TypeScript, тому тут не використовуються окремі файли для декларацій типів (файлів формату d.ts), які повсякчас використовуються для швидкого опису типів JavaScript-бібліотек. Такий підхід дозволяє спростити життя як розробнику, так і користувачу бібліотеки, адже розробник може розроблювати бібліотеку, використовуючи додаткові засоби цієї мови, а користувачу не має необхідності встановлювати додатковий пакет для типів.

При розробленні бібліотеки визначено, що залежні програмні пакети привносять додаткові проблеми та збільшують ризик появи вразливостей у безпеці. Саме тому найбільшу довіру у людей викликають пакети без залежностей. У даній бібліотеці є тільки одна залежність – це бібліотека для перевірки даних на правильність – class-validator. Дана бібліотека є частиною стеку програмних бібліотек typestack та базується на пакеті validator.js, але на відміну від нього надає зручний інтерфейс опису правил валідації за допомогою одного з найпотужніших механізмів мови програмування TypeScript – декораторів. Цей механізм дозволяє фактично додавати виконання функцій для цілих класів або їх полів, методів та аргументів останніх. Фактично для кожного випадку реалізована своя форма декораторів з власними можливостями обмеженнями. Фактично, декоратор – це фунція, яка приймає визначені типом декоратора аргументи та повертає функцію. Тут також потрібно відмітити гнучкість методів для перевірки цих класів-валідаторів, які дозволяють

Залежність між модулями розробки наведена у наступній UML-діаграмі (рис. 4.2).

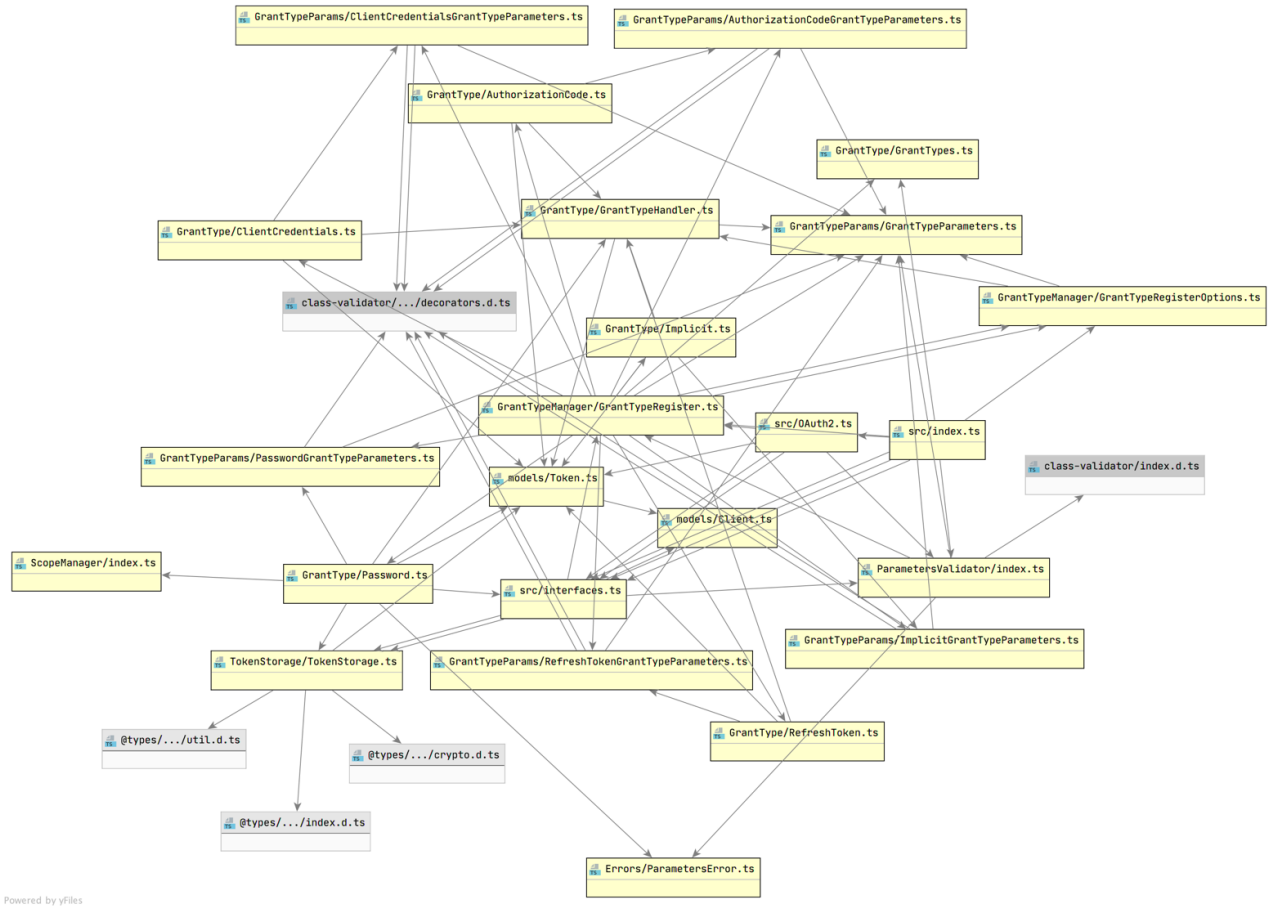


Рис. 4.2. Діаграма модулів розробленої бібліотеки

Наведемо конкретний список розроблених компонентів:

Пакет models містить опис найбільш використовуваних інтерфейсів даної бібліотеки:

- Client – інтерфейс, що містить опис даних клієнту (згідно стандарту). Тут містяться такі поля як:
  - id;
  - secret;
- Token – інтерфейс, що представляє собою зручну комбінацію даних для маркерів доступу та оновлення доступу, складається із наступних полів:
  - accessToken – об’єкт даних для маркеру доступу, що має поля value, expiresIn, expiresAt. Дані поля відповідно

містять інформацію про значення токену та час, через який він стане недоступним;

- refreshToken – опціональне поле даних для маркеру відновлення доступу. Це поле – об'єкт з полями values, expiresIn та expiresAt, які є повними відповідниками таких самих полів для маркеру доступу;
- scope – масив рядків, що позначає запитувані користувачем рівні доступу;
- client – масив об’єктів типу Client, який позначає набір клієнтів.

OAuth2 – основний клас бібліотеки. За допомогою об'єкта цього класу користувач може задавати початкові конфігурації бібліотеки та взаємодіяти із нею. Тут наявні такі поля:

- grantTypeRegister – поле типу GrantTypeRegister, яке представляє регістр типів гранту у бібліотеці;
- tokensStorages – поле типу TokenStorageInstances. Даний інтерфейс складається з трьох полів типу TokenStorage (обов’язкового accessToken та необов’язкових refreshToken та authCode), кожне з яких відповідає за процес збереження та отримання відповідних маркерів;
- parametersValidator – поле типу ParameteresValidator, яке відповідає за перевірку коректності уведених даних в залежності від присутнього типу гранту. Окрім цього, перевіряється й сам тип гранту;
- поля errorConfig та scopeManagementConfig, реалізація яких є необов’язковою. За допомогою реалізації даних полів користувач має можливість розширення функціональності бібліотеки.

Варто відзначити, що усі ці поля також містяться у інтерфейсі OAuth2InitConfig, який використовується у конструкторі даного класу для

підвищення читаності коду. Розглянемо більш детально кожний із описаних класів.

GrantTypeRegister – це клас, який відповідає за відповідність між типом гранту та його конфігурацією (параметрами та функцією, яка їх обробляє). Тут міститься одне з найважливіших полів – словник, який безпосередньо відповідає за цю відповідність та методи для взаємодії з ним.

Реалізація вже існуючих типів гранту та можливість впровадження своєї ґрунтуються на використанні абстрактного класу GrantTypeHandler з єдиним методом handle, що приймає поле GrantTypeParameters. Відповідно до типів гранту специфікації RFC 6749 даний клас реалізується у класах AuthorizationCode, ClientCredentials, Implicit, Password та RefreshToken. Логіка кожної з цих реалізації строго відповідає специфікації.

ParametersValidator – це клас, який приймає регістр типів грантів в якості параметра та використовує його для знаходження необхідної стратегії валідації для отриманого типу гранта. Стратегії валідації описуються у вигляді класів з полями, що мають декоратори із бібліотеки class-validator. За відсутності наданого типу гранта у регістрі користувачу повертається помилка з повідомленням про це.

TokenConfig – інтерфейс, який описує конфігурацію кожного з типів токену. Містить поля типу TokenConfiguration.

TokenConfiguration – інтерфейс, який містить тільки два поля для опису взаємодії з конкретним токеном: storage типу TokenStorage, що надає реалізацію для збереження та отримання маркеру та lifetime типу number, яке позначає строк життя маркеру у секундах.

Зберігання даних у цій бібліотеці підтримуються завдяки трьом класам – TokenStorage, UserStorage та ClientStorage, реалізація яких вже лежить на користувачеві. Дані класи фактично представляють собою частину концепції моделі сервера (ServerModel), яка представлена у бібліотеці node-oauth2-server. Використання даних абстракцій дозволяє бібліотеці бути незалежною від методу зберігання даних та таким чином

підтримувати їх усі. TokenStorage представляє собою абстрактний клас, який містить заздалегідь реалізований метод генерування токенів та абстрактні методи для збереження, отримання токенів та припинення їх строку дії (revoke).

Окрім цього, у коді містяться інтерфейси для зберігання даних клієнту та користувача – UserStorage та ClientStorage, кожний з яких має відповідні методи getUser та getClient.

Для спрощення прийняття структурної залежності між даними класами, наведемо UML-діаграми (рис. 4.3, 4.4).

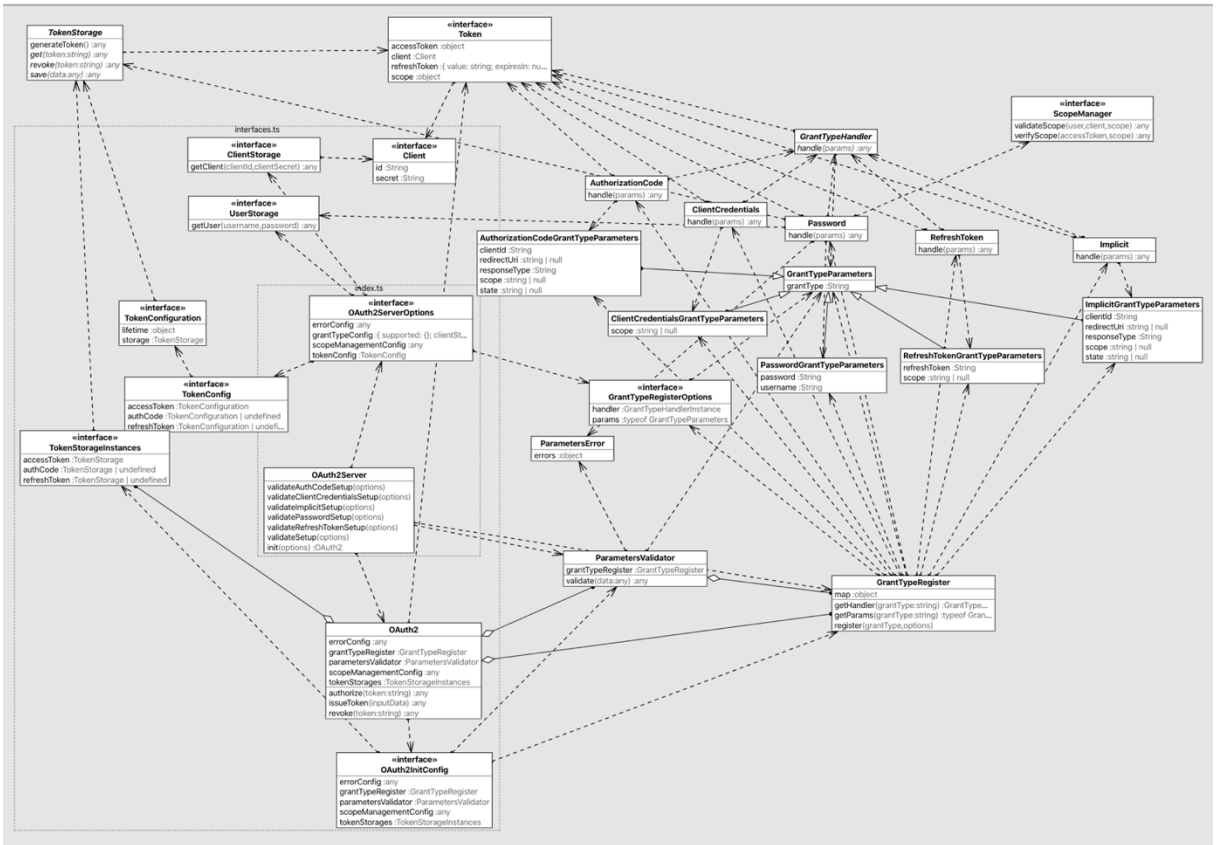


Рис. 4.3. Діаграма класів розробленої бібліотеки

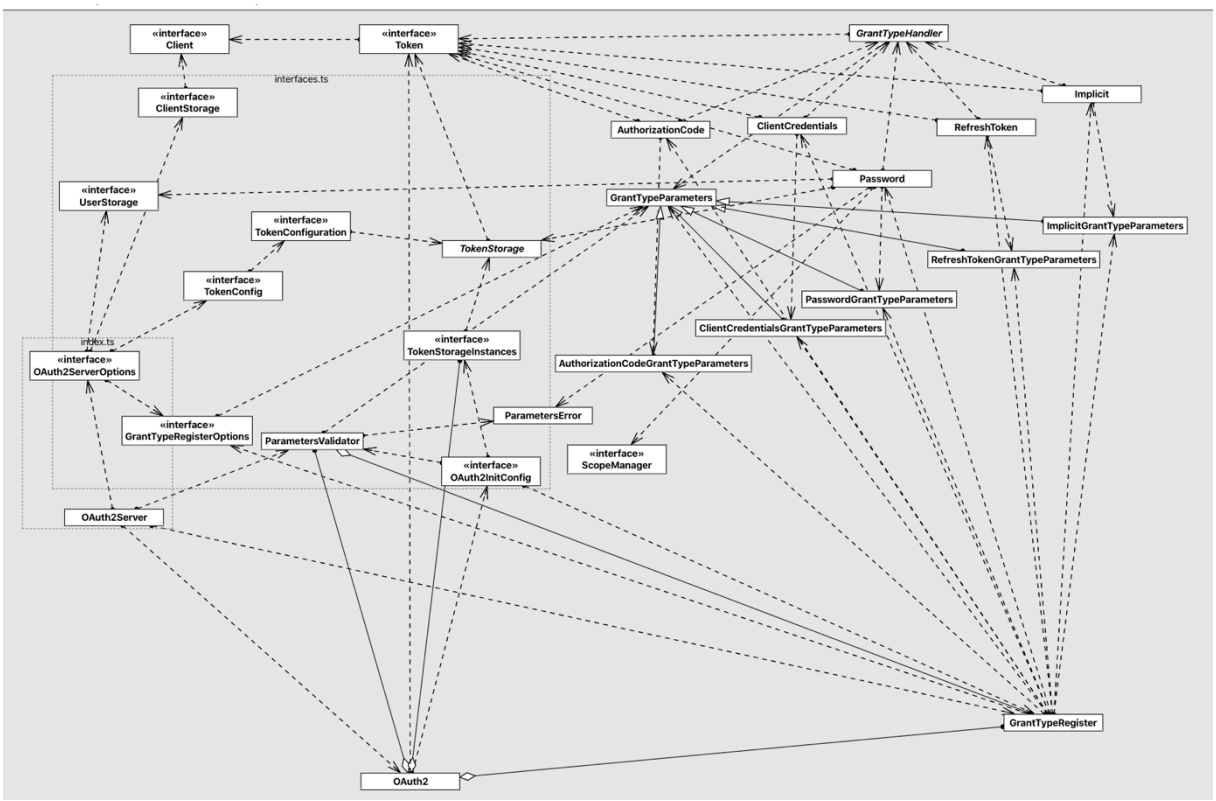


Рис. 4.4. Спрощена діаграма класів розробленої бібліотеки

4.2. Опис допоміжних програмних засобів

Під час створення програмної бібліотеки були використані допоміжні програмні засоби, які забезпечують найбільшу швидкість розроблення і тестування даної бібліотеки. Цей набір складався з наступних інструментів:

- NVM (Node Version Manager) – це менеджер версій для Node.js, призначений для встановлення для окремого користувача та викликається для кожного терміналу окремо. NVM працює на будь-якому терміналі, що сумісний з POSIX (sh, dash, ksh, zsh, bash), зокрема на таких платформах: UNIX, macOS та Windows WSL. Даний інструмент дозволяє зручно та швидко перемикатися між різними версіями оточення Node.js, що є важливою функцією для бібліотеки, яка підтримує декілька таких версій;
- tslint – це розширюваний інструмент для статичного аналізу програмного коду, який перевіряє код TypeScript на наявність помилок читабельності, підтримуваності та функціональності.

Він широко підтримується в сучасних текстових редакторах, і його можна налаштувати за допомогою власних правил та конфігурацій;

- GitHub – найбільший сервіс для хостингу програмних проєктів та їх розробки. Даний сервіс дозволяє у зручний спосіб проводити спільну розробку та робити внесок у існуючі проєкти;
- Travis CI – розподілений веб-сервіс для збірки та тестування програмного забезпечення. Даний сервіс дозволяє тестувати роботу бібліотеки для декількох версій оточення Node.js після кожної зміни коду;
- NPM – node package manager, менеджер пакетів, який входить до складу оточення Node.js. За допомогою програми, яка входить до складу node, було встановлено додаткові пакети для коректної роботи бібліотеки. Окрім цього, дана програма використовується для встановлення пакетів під час збірки на веб-сервісі Travis CI.

4.3. Приклади використання бібліотеки

Для того, щоб продемонструвати можливості бібліотеки, було розроблено демонстраційні версії додатків, кожен з яких утилізує окрему частину бібліотеки. Тестування даних додатків відбувалось за допомогою запитів до цих додатків за допомогою HTTP-клієнта. У цьому розділі наведено лістинги з програмним кодом додатків, які були використані під час тестування функціоналу бібліотеки та приклади запитів та відповідей на них зі сторони сервера.

Для проведення демонстрації роботи програми було створено додаток, у якому підтримуються наступні функції: реєстрація користувача за допомогою логіна, пароля та імені, реєстрація за допомогою соціальних мереж (в даному конкретному випадку – Google та Facebook), вхід у систему за допомогою соціальних мереж або логіну та пароля, вихід із системи. Для демонстрації роботи додатку наведено скріншоти із HTTP-клієнту Insomnia.

Розроблений додаток використовує базу даних PostgreSQL для збереження даних. Тестування відбувалось локально за допомогою створення бази даних безпосередньо на локальній машині.

Програмний код, який відповідає за функції авторизації та аутентифікації в даному додатку, складається із одного контролера AuthController на базі фреймворку Nest.js:

Код класу AuthController

```
import { Controller, Delete, Inject, Post, Req, Res, UseGuards } from
'@nestjs/common';
// … more imports
@Controller()
export class AuthController {
    constructor(
     @Inject(OAuth2ServerType)
     private readonly oauth: OAuth2Server,
     @InjectRepository(AccessTokenEntity)
     private readonly accessTokenRepository: Repository<AccessTokenEntity>,
     @InjectRepository(RefreshTokenEntity)
     private readonly refreshTokenRepository:
Repository<RefreshTokenEntity>,
    ) { }
// Тут буде міститься реалізація необхідних методів
}
```

Даний контролер у конструкторі отримує значення для класів, що реалізують репозиторії для таблиць із маркерами доступу та оновлення доступу. Окрім цього, у конструкторі тут мається поле oauth, яке відповідає об'єкту класу бібліотеки авторизації. Для коректної роботи цього коду необхідно реалізувати відповідний клас модулю:

Код класу AppModule

```
import { Module } from '@nestjs/common';
// more imports

@Module({
```

```
    imports: [
        TransactionalTypeOrmModule.forFeature([
            AccessTokenEntity,
            RefreshTokenEntity,
        ]),
    ],
    providers: [ {
        provide: OAuthServerModelType,
        useClass: OAuthServerModel,
    }, {
        provide: OAuth2ServerType,
        useFactory: (serverModel: OAuthServerModel) =>
            new OAuth2Server({
                model: serverModel,
                accessTokenLifetime: config.auth.accessTokenLifetime,
                refreshTokenLifetime: config.auth.refreshTokenLifetime,
                extendedGrantTypes: {
                    'social-network': SocialNetworkGrantType,
                },
            }),
        inject: [ OAuthServerModelType ],
    } ]
    controllers: [ AuthController ],
})
export class AppModule { }
```

Тоді метод входу до систему буде мати наступний вигляд:

Код методу login

```
@Post('login')
@ApiBody({ description: 'Login input', type: LoginInput })
@ApiOkResponse({ description: 'User has been login successfully', type:
AuthTokenResponse })
public async login(@Req() request: Request, @Res() response: Response):
Promise<void> {
    await validateOrThrow(LoginInput, request);
    const token = await this.oauth.token(request, response);
    const response = AuthTokenResponse.fromObject({
        accessToken: token.accessToken,
        refreshToken: token.refreshToken as string,
    });
```

```
    sendResponse(result, response);
}
```

Відповідний до нього метод, який дозволяє користувачеві вийти із системи за допомогою видалення токенів отримання та оновлення доступу, матиме такий вигляд:

Код методу logout

```
@Delete('logout')
@UseGuards(AuthGuard)
@ApiOkResponse({ description: SUCCESS })
public async logout(@CurrentUser({ required: true }) authUser: AuthUser):
Promise<void> {
    const accessTokenToUpdate = await this.accessTokenRepository.findOne({
where: { accessToken }, cache: true }) || null;
    if (!accessTokenToUpdate) {
        throw new UnauthorizedException('Invalid token');
    }
    const refreshTokenToUpdate = await
this.refreshTokenRepository.findOne({ where: { refreshToken }, cache: true
}) || null;
    if (!refreshTokenToUpdate) {
        throw new UnauthorizedException('Invalid token');
    }
    const updatedAccessToken = this.disableToken(accessTokenToUpdate);
    const updatedRefreshToken = this.disableToken(refreshTokenToUpdate);
    await Promise.all([
        this.accessTokenRepository.save(updatedAccessToken),
        this.refreshTokenRepository.save(updatedRefreshToken),
        ]);
    }
```

Тут метод disableToken — це простий метод, який оновлює маркери:

Код методу disableToken

```
private disableToken<T>(token: T): T {
    return {
        ...token,
        isActive: false,
    };
}
```

(POST) /api/registration – кінцева точка, яка реєструє користувача, а потім виконує вхід у систему. Повертає два маркери – маркер доступу та маркер оновлення доступу. Конкретно в даному випадку для входу в систему використовується тип гранту password.

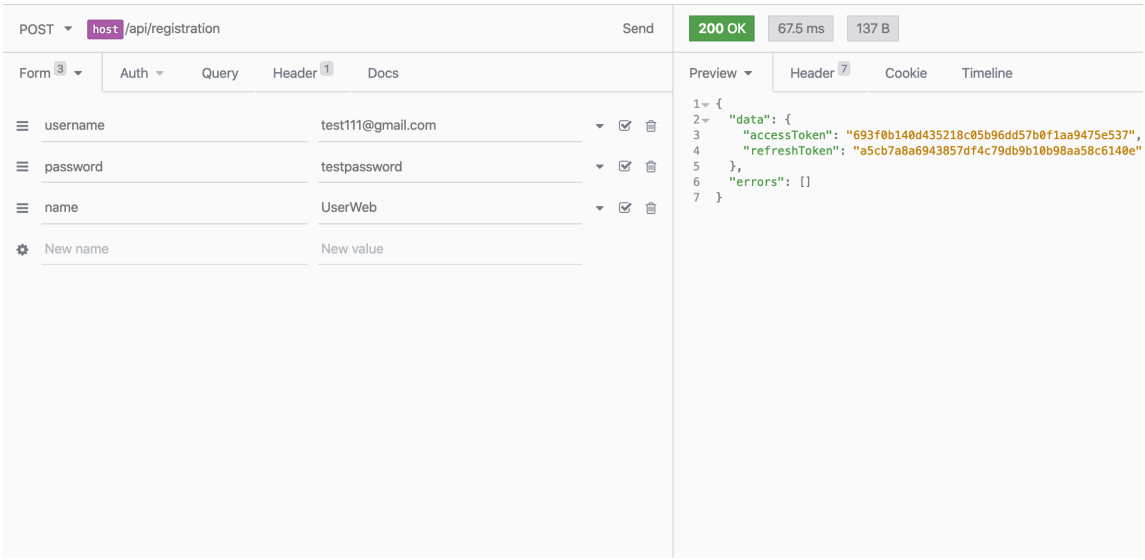


Рис. 4.5. Приклад запиту та відповіді під час реєстрації

(POST) /api/login – кінцева точка, яка відповідає за вхід користувача у систему. Параметри запиту відповідають специфікації RFC 6749.

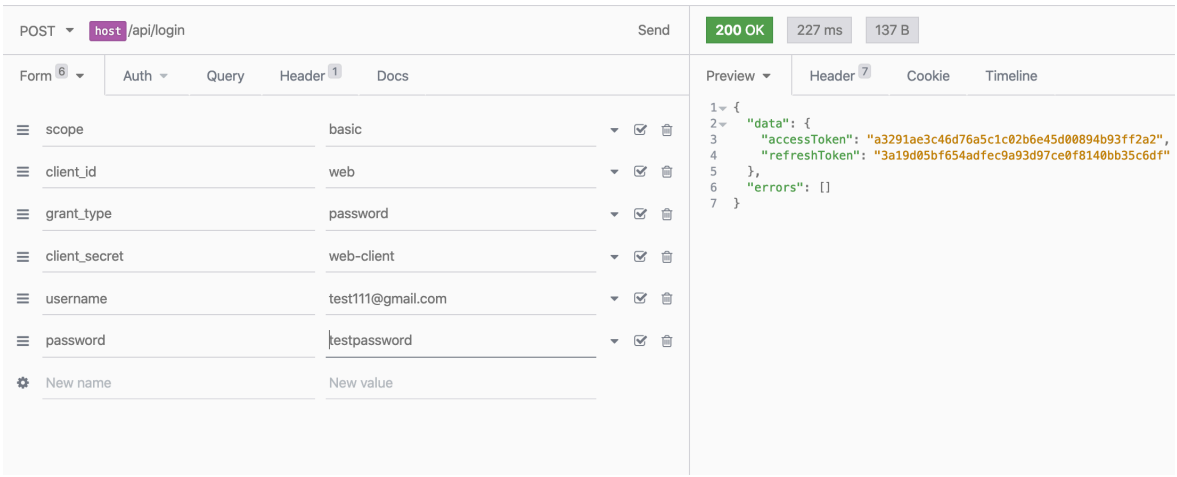


Рис. 4.6. Приклад запиту та відповіді під час входу до системи

(POST) /api/social-login – кінцева точка, яка відповідає за реєєстрацію або вхід користувача за допомогою соціальної мережі. Якщо вхід було виконано успішно, то користувачу повертається пара ідентифікаторів, якщо ж ні – повертається поимлка з відповідним повідомленням. Даний ендпоінт має таку саму реалізацію, як і звичайний метод входу.

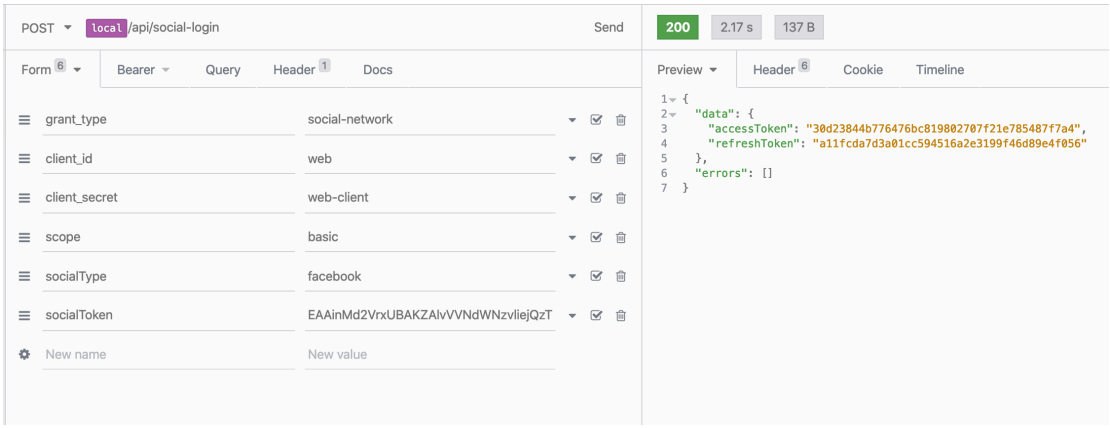


Рис. 4.7. Приклад відправлення запиту на вхід через Facebook

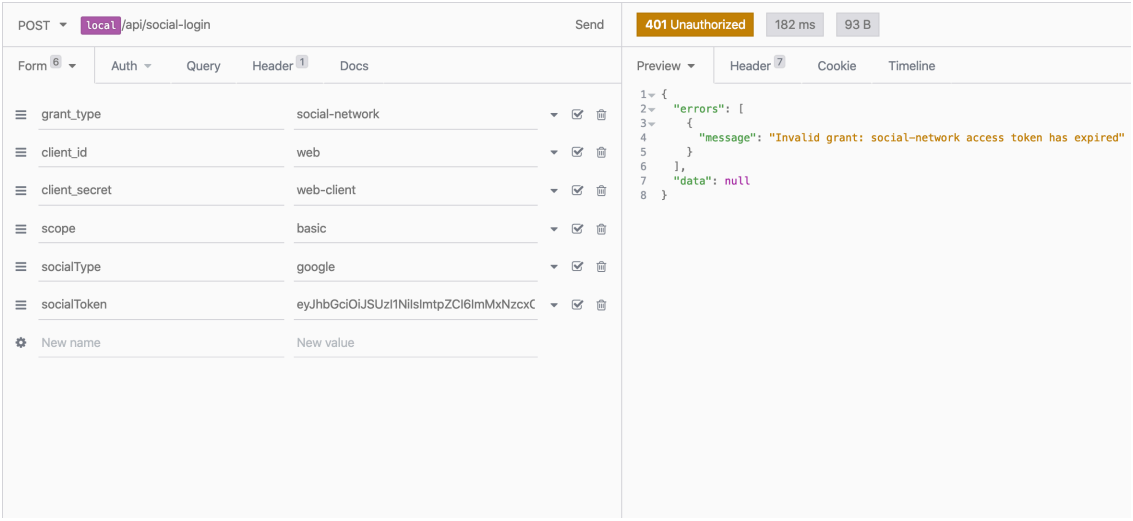


Рис. 4.8. Приклад помилки під час входу через Google

(POST) /api/refresh-token – кінцева точка, який відповідає за оновлення ідентифікатору доступу.

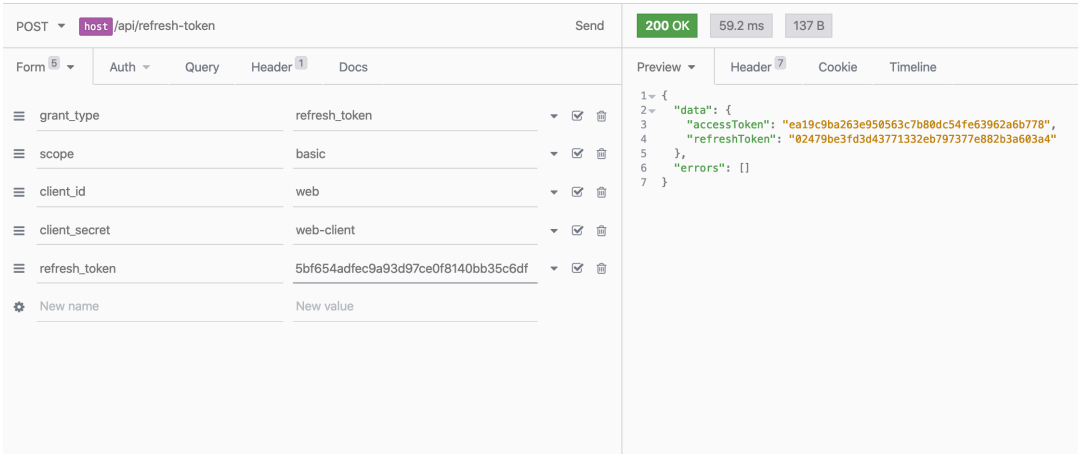


Рис. 4.9. Приклад запиту на маркер оновлення доступу

(DELETE) /api/logout – кінцева точка, який дозволяє користувачу вийти із системи. В даному випадку зміна token відповідає маркеру доступа, який користувач отримав на етапі входу в систему.

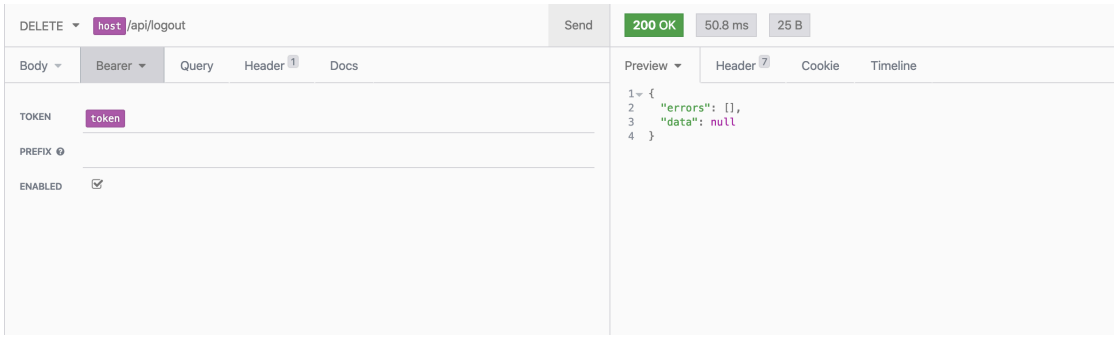


Рис. 4.10. Приклад запиту на вихід із системи

Отже, в даному розділі виконано аналіз структур даних, які реалізовано для даної бібліотеки. Наведено приклади типових сценаріїв використання даної бібліотеки у коді та виконано перевірку її працездатності за допомогою HTTP-клієнта.

ВИСНОВКИ

Даний дипломний проєкт присвячений вирішенню задачі побудови бібліотеки для підтримки створення сервера авторизації на базі протоколу OAuth 2.0 для оточення Node.js. Це стало можливим завдяки виконанню низки дій, а саме було проведено детальний аналіз проблеми поряд з аналізом існуючих на ринку рішень, на основі цих даних побудовано шлях для досягнення цілей, після завершення роботи було проведена оцінка результатів. Для покращення якості даної розробки були враховані потреби спільноти розробників даного оточення та останніх тенденцій у цьому напрямку. На основі цих даних було зроблено вибір мови програмування для бібліотеки, її ключових функцій та стилю написання. Ця інформація була оброблена та оформлена у вигляді вимог до розроблюваного рішення. Відповідний процес мав місце і для супутньої версії бібліотеки. Необхідність у даній версії виникла через неможливість створення універсального рішення для фреймворку Nest.js. В останніх розділах проєкту поряд із особливостями розроблених структур даних наведено опис реалізації головних алгоритмів даного рішення.

Напрямок розроблення програмних бібліотек для створення сервера авторизації є перспективним для подальшого дослідження, оскільки навіть за наявності незмінної специфікації, у середовищі розробки з'являються більш витончені та продумані підходи до написання програмного коду, яким повинна відповідати сучасна бібліотека. Заплановано зменшення кількості залежностей до нуля, з можливою розробкою власних рішень для реалізації необхідного функціоналу. Для основної версії бібліотеки рекомендовано покращення архітектури відповідно до сценаріїв її використання.

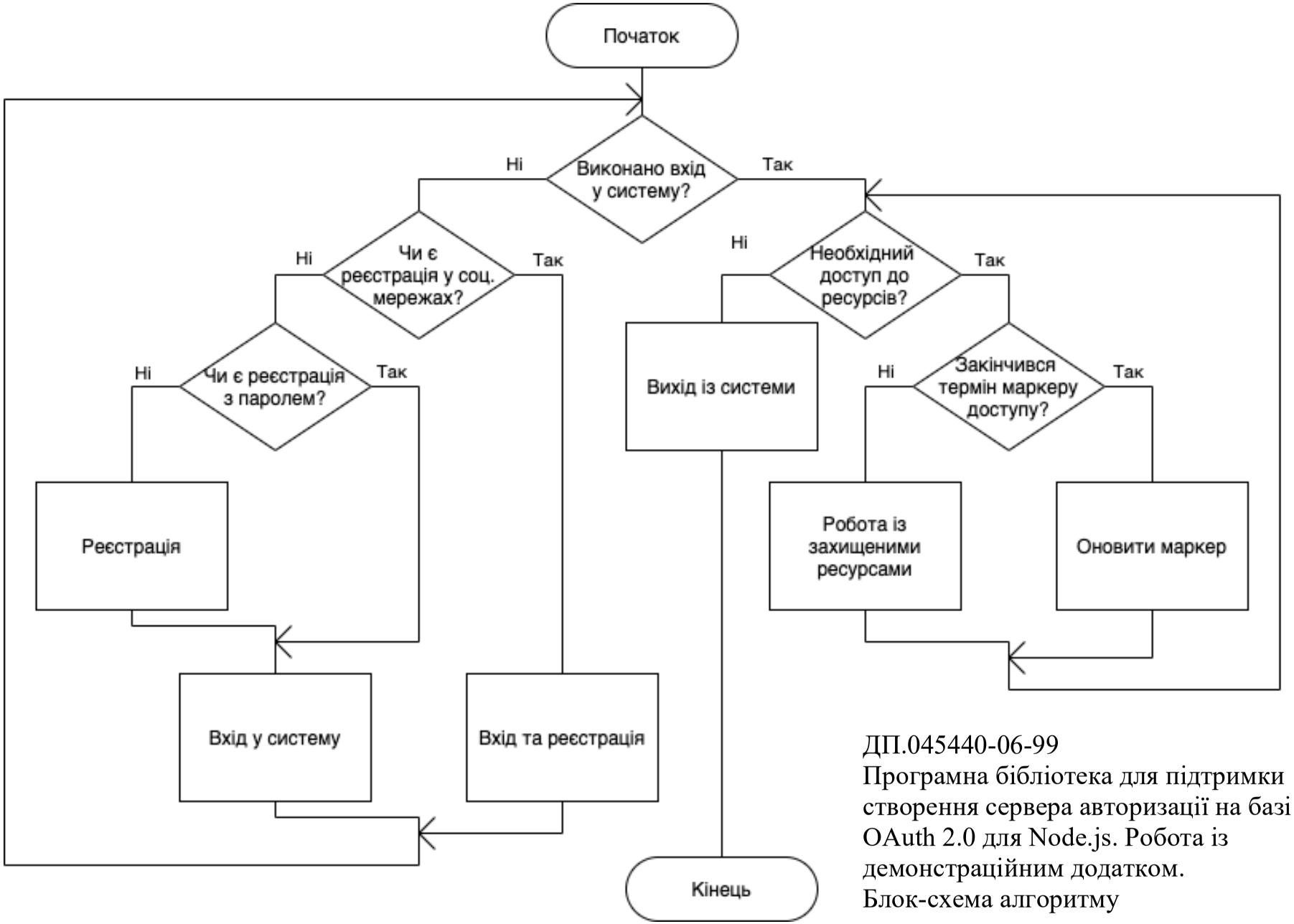
СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. JavaScript [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/JavaScript. — (20.04.2020) — Назва з екрана.
2. TypeScript [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/TypeScript. — (20.04.2020) — Назва з екрана.
3. Nest.js Documentation [Електронний ресурс]. — Режим доступу : https://docs.nestjs.com. — (04.05.2020) — Назва з екрана.
4. Node.js Документація [Електронний ресурс]. — Режим доступу : https://nodejs.org/uk/docs. — (04.05.2020) — Назва з екрана.
5. InversifyJS [Електронний ресурс]. —Режим доступу : https://github.com/inversify/InversifyJS. — (10.05.2020) — Назва з екрана.
6. TypeDI [Електронний ресурс]. —Режим доступу : https://github.com/typestack/typedi. — (10.05.2020) — Назва з екрана.
7. The OAuth 2.0 Authorization Framework [Електронний ресурс]. — Режим доступу : https://tools.ietf.org/html/rfc6749. — (06.05.2020) — Назва з екрана.
8. The OAuth 2.0 Authorization Framework: Bearer Token Usage [Електронний ресурс]. — Режим доступу : https://tools.ietf.org/html/rfc6750. — (06.05.2020) — Назва з екрана.
9. node-oauth2-server [Електронний ресурс]. — Режим доступу : https://github.com/oauthjs/node-oauth2-server. — (27.04.2020) — Назва з екрана.
10. oauth2orize [Електронний ресурс]. — Режим доступу : https://github.com/jaredhanson/oauth2orize. — (27.04.2020) — Назва з екрана.

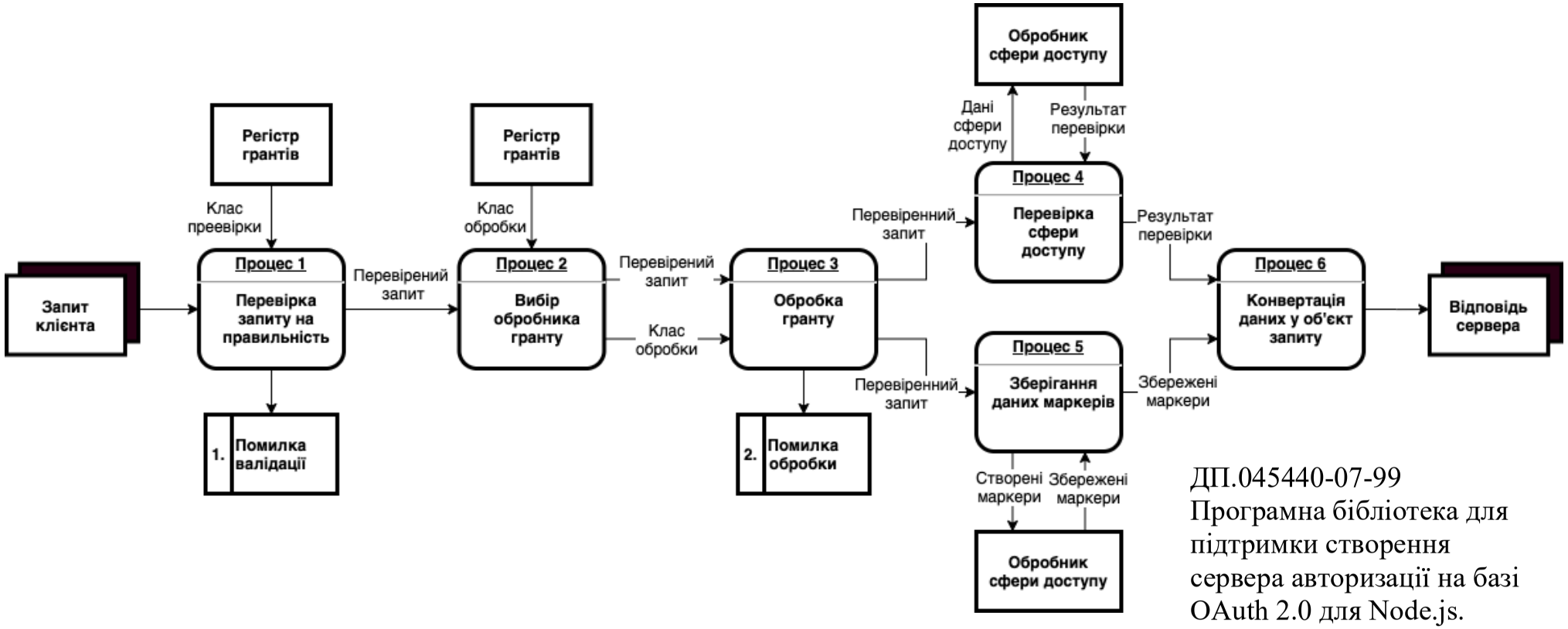
11. CoffeeScript [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/CoffeeScript. — (20.04.2020) — Назва з екрана.
12. Scala [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/Scala. — (20.04.2020) — Назва з екрана.
13. Kotlin [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/Kotlin. — (20.04.2020) — Назва з екрана.
14. Dependency Injection in JavaScript 101 [Електронний ресурс]. — Режим доступу : https://dev.to/azure/dependency-injection-in-javascript-101-2b1e. — (10.05.2020) — Назва з екрана.

ДОДАТКИ

Додаток 1
Копії графічних матеріалів

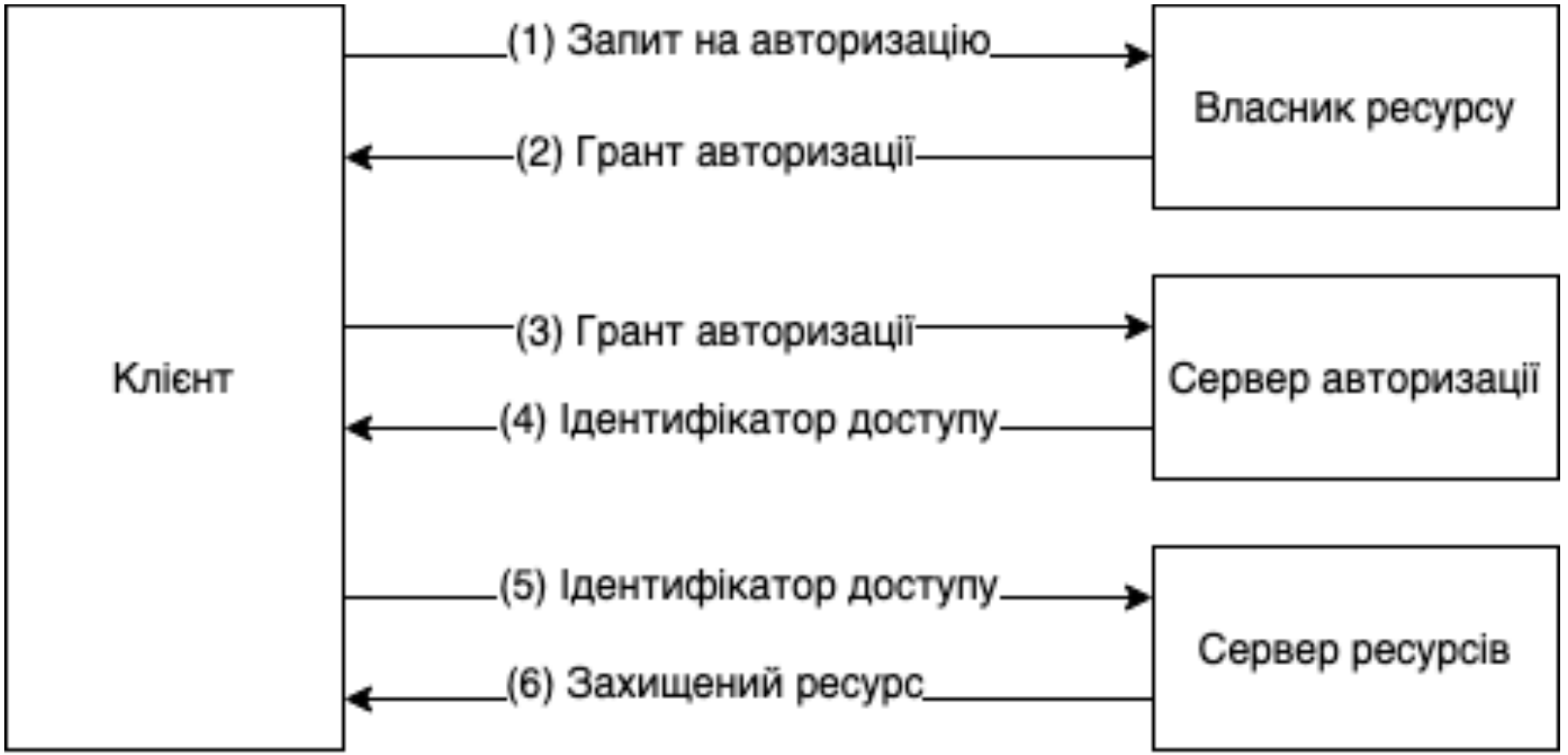


ДП.045440-06-99
Програмна бібліотека для підтримки створення сервера авторизації на базі OAuth 2.0 для Node.js. Робота із демонстраційним додатком.
Блок-схема алгоритму



ДП.045440-07-99
Програмна бібліотека для підтримки створення сервера авторизації на базі OAuth 2.0 для Node.js. Обробка запиту клієнта. UML-діаграма потоку даних

Діаграма авторизації за протоколом OAuth 2.0



Бай Я.В., група КП-62

Алгоритм оновлення маркерів доступу та оновлення доступу

Бай Я.В., група КП-62

Додаток 2
Лістинг головного модулю, модулю ініціалізації, модулю регістру грантів, класів валідації та обробки

index.ts

```
import { GrantTypes } from './GrantType/GrantTypes';
import { GrantTypeRegister } from './GrantTypeManager/GrantTypeRegister';
import { GrantTypeConfig, TokenConfig } from './interfaces';
import { OAuth2 } from './OAuth2';
import { ParametersValidator } from './ParametersValidator';
import { ScopeManager } from './ScopeManager';

export class OAuth2Server<U> {
    public init(options: OAuth2ServerOptions<U>) {
        this.validateSetup(options);
        const grantTypeRegister = new
GrantTypeRegister(options.grantTypeConfig.supported);
        const customGrantTypes = options.grantTypeConfig.custom;
        if (customGrantTypes) {
            for (const [grantType, config] of
Object.entries(customGrantTypes)) {
                grantTypeRegister.register(grantType, config);
            }
        }

        const parametersValidator = new
ParametersValidator(grantTypeRegister);
        return new OAuth2({
            grantTypeRegister,
            tokenStorages: {
                accessToken: options.tokenConfig.accessToken.storage,
                refreshToken: options.tokenConfig.refreshToken?.storage,
                authCode: options.tokenConfig.authCode?.storage,
            },
            tokenConfig: options.tokenConfig,
            userStorage: options.grantTypeConfig.userStorage,
            clientStorage: options.grantTypeConfig.clientStorage,
            parametersValidator,
            errorConfig: options.errorConfig,
            scopeManagementConfig: options.scopeManagementConfig,
            scopeManager: options.scopeManager,
        });
    }

    private validateSetup(options: OAuth2ServerOptions<U>) {
        const {supported} = options.grantTypeConfig;

        if (supported.includes(GrantTypes.AUTHORIZATION_CODE)) {
            this.validateAuthCodeSetup(options);
        }
        if (supported.includes(GrantTypes.IMPLICIT)) {
            this.validateImplicitSetup(options);
        }
        if (supported.includes(GrantTypes.PASSWORD)) {
            this.validatePasswordSetup(options);
        }
        if (supported.includes(GrantTypes.CLIENT_CREDENTIALS)) {
            this.validateClientCredentialsSetup(options);
        }
        if (supported.includes(GrantTypes.REFRESH_TOKEN)) {
            this.validateRefreshTokenSetup(options);
        }
    }

    private validateAuthCodeSetup(options: OAuth2ServerOptions<U>) {
        if (!options.tokenConfig.authCode) {
            throw new Error('Auth code storage required');
        }
```

```
    }
    private validateImplicitSetup(options: OAuth2ServerOptions<U>) {
        if (!options.tokenConfig.authCode) {
            throw new Error('Auth code storage required');
        }
    }
    private validatePasswordSetup(options: OAuth2ServerOptions<U>) {
        if (!options.grantTypeConfig.userStorage) {
            throw new Error('User storage required');
        }
    }
    private validateClientCredentialsSetup(options: OAuth2ServerOptions<U>)
{
        if (!options.grantTypeConfig.userStorage) {
            throw new Error('User storage required');
        }
        if (!options.grantTypeConfig.clientStorage) {
            throw new Error('Client storage required');
        }
    }
    private validateRefreshTokenSetup(options: OAuth2ServerOptions<U>) {
        if (!options.tokenConfig.refreshToken) {
            throw new Error('Refresh token storage required');
        }
    }
}

export interface OAuth2ServerOptions<U> {
    tokenConfig: TokenConfig;
    errorConfig?: any;
    scopeManagementConfig: {
        addAcceptedScopesHeader: boolean;
        addAuthorizedScopesHeader: boolean;
    };
    scopeManager: ScopeManager;
    bearerTokenConfig: {
        allowBearerTokensInQueryString: boolean;
    };
    grantTypeConfig: GrantTypeConfig<U>;
}
```

OAuth2.ts

```
import * as auth from 'basic-auth';
import { GrantTypeRegister } from './GrantTypeManager/GrantTypeRegister';
import { ClientStorage, OAuth2InitConfig, TokenConfig,
TokenStorageInstances, UserStorage } from './interfaces';
import { Token } from './models/Token';
import { ParametersValidator } from './ParametersValidator';
import { Request } from './Request';
import { ScopeManager } from './ScopeManager';

export class OAuth2<U> {

    private grantTypeRegister: GrantTypeRegister<U>;

    private tokenStorages: TokenStorageInstances;

    private parametersValidator: ParametersValidator;

    private readonly clientStorage: ClientStorage | null;

    private readonly userStorage: UserStorage<U> | null;

    private errorConfig: any;

    private readonly tokenConfig: TokenConfig;

    private scopeManagementConfig: any | null;

    private readonly scopeManager: ScopeManager;

    constructor(config: OAuth2InitConfig<U>) {
        this.grantTypeRegister = config.grantTypeRegister;
        this.tokenStorages = config.tokenStorages;
        this.parametersValidator = config.parametersValidator;
        this.errorConfig = config.errorConfig;
        this.scopeManagementConfig = config.scopeManagementConfig;
        this.userStorage = config.userStorage || null;
        this.clientStorage = config.clientStorage || null;
        this.scopeManager = config.scopeManager;
        this.tokenConfig = config.tokenConfig;
    }

    public async issueToken(request: Request): Promise<Token> {
        const inputData = request.body;
        const grantTypeParameters = await
this.parametersValidator.validate(inputData);

        const grantTypeConstructor =
this.grantTypeRegister.getHandler(inputData.grant_type);

        const clientCredentials = this.getClientCredentials(request);
        grantTypeParameters.clientId = clientCredentials.clientId;
        grantTypeParameters.clientSecret = clientCredentials.clientSecret;

        const grantType = new grantTypeConstructor(
            this.scopeManager,
            this.userStorage,
            this.clientStorage,
            this.tokenConfig,
        );

        return await grantType.handle(grantTypeParameters);
    }
```

```
    public async revoke(request: Request): Promise<any> {
        const clientCredentials = this.getClientCredentials(request);
        if (!this.clientStorage) {
            throw new Error('Client storage should be implemented in order
to revoke token');
        }
        const client = await this.clientStorage.getClient(
            clientCredentials.clientId,
            clientCredentials.clientSecret,
        );
        if (!client) {
            throw new Error('Invalid client: client is invalid');
        }
        if (!client.grants) {
            throw new Error('Server error: missing client `grants`');
        }
        const token = request.;

        return this.tokenStorages.accessToken.revoke(token);
    }

    public async authorize(token: string): Promise<Token> {
        const foundToken = await this.tokenStorages.accessToken.get(token);

        if (!foundToken) {
            throw new Error('Token not found');
        }

        if (foundToken.accessToken.expiresAt >= new Date()) {
            throw new Error('Token expired');
        }

        return foundToken;

    }

    private getClientCredentials(request: Request): { clientId: string;
clientSecret: string; } {
        const credentials = auth(request as any);
        if (credentials) {
            return {
                clientId : credentials.name,
                clientSecret : credentials.pass,
            };
        } else if (request.body.client_id && request.body.client_secret) {
            return {
                clientId : request.body.client_id,
                clientSecret : request.body.client_secret,
            };
        }
        throw new Error('Invalid client: cannot retrieve client
credentials');
    }

}
```

GrantTypeRegister.ts

```
import { AuthorizationCode } from '../GrantType/AuthorizationCode';
import { ClientCredentials } from '../GrantType/ClientCredentials';
import { GrantTypes } from '../GrantType/GrantTypes';
import { Implicit } from '../GrantType/Implicit';
import { Password } from '../GrantType/Password';
import { RefreshToken } from '../GrantType/RefreshToken';
import { AuthorizationCodeGrantTypeParameters } from
'../GrantTypeParams/AuthorizationCodeGrantTypeParameters';
import { ClientCredentialsGrantTypeParameters } from
'../GrantTypeParams/ClientCredentialsGrantTypeParameters';
import { GrantTypeParameters } from
'../GrantTypeParams/GrantTypeParameters';
import { ImplicitGrantTypeParameters } from
'../GrantTypeParams/ImplicitGrantTypeParameters';
import { PasswordGrantTypeParameters } from
'../GrantTypeParams/PasswordGrantTypeParameters';
import { RefreshTokenGrantTypeParameters } from
'../GrantTypeParams/RefreshTokenGrantTypeParameters';
import { GrantTypeHandlerInstance, GrantTypeRegisterOptions } from
'./GrantTypeRegisterOptions';

export class GrantTypeRegister<U> {
    private readonly map: { [k: string]: GrantTypeRegisterOptions<U> } = {
        [GrantTypes.PASSWORD]: {
            params: PasswordGrantTypeParameters,
            handler: Password,
        },
        [GrantTypes.IMPLICIT]: {
            params: ImplicitGrantTypeParameters,
            handler: Implicit,
        },
        [GrantTypes.REFRESH_TOKEN]: {
            params: RefreshTokenGrantTypeParameters,
            handler: RefreshToken,
        },
        [GrantTypes.CLIENT_CREDENTIALS]: {
            params: ClientCredentialsGrantTypeParameters,
            handler: ClientCredentials,
        },
        [GrantTypes.AUTHORIZATION_CODE]: {
            params: AuthorizationCodeGrantTypeParameters,
            handler: AuthorizationCode,
        },
    };

    constructor(supportedGrantTypes: string[]) {
        this.map = supportedGrantTypes.reduce((a, grantType) => ({ ...a,
[grantType]: this.map[grantType] }), { });
    }

    public getParams(grantType: string): typeof GrantTypeParameters {
        return this.map[grantType].params;
    }

    public getHandler(grantType: string): GrantTypeHandlerInstance<U> {
        return this.map[grantType].handler;
    }

    public register(grantType: string, options:
GrantTypeRegisterOptions<U>) {
        this.map[grantType] = options;
    }
}
```

PasswordGrantTypeParameters.ts

```
import { IsString } from 'class-validator';
import { GrantTypeParameters } from './GrantTypeParameters';

export class PasswordGrantTypeParameters extends GrantTypeParameters {

    @IsString()
    username: string;

    @IsString()
    password: string;

    @IsString()
    clientId: string;

    @IsString()
    scope: string | null;

    constructor(data: { username: string, password: string, clientId:
string, scope?: string | null, grant_type: string }) {
        super(data);
        this.password = data.password;
        this.username = data.username;
        this.clientId = data.clientId;
        this.scope = data.scope || null;
    }

}
```

RefreshTokenGrantTypeParameters.ts

```
import { IsOptional, IsString } from 'class-validator';
import { GrantTypeParameters } from './GrantTypeParameters';

export class RefreshTokenGrantTypeParameters extends GrantTypeParameters {

    @IsString()
    refreshToken: string;

    @IsOptional()
    @IsString()
    scope: string | null;

    constructor(data: {
        grant_type: string,
        refresh_token: string,
        scope?: string
    }) {
        super(data);
        this.refreshToken = data.refresh_token;
        this.scope = data.scope || null;
    }

}
```

ImplicitGrantTypeParameters.ts

```
import { IsOptional, IsString } from 'class-validator';
import { GrantTypeParameters } from './GrantTypeParameters';

export class ImplicitGrantTypeParameters extends GrantTypeParameters {

    @IsString()
    responseType: string;

    @IsString()
    clientId: string;

    @IsOptional()
    @IsString()
    redirectUri: string | null;

    @IsOptional()
    @IsString()
    scope: string | null;

    @IsOptional()
    @IsString()
    state: string | null;

    constructor(data: {
        grant_type: string,
        response_type: string,
        client_id: string,
        redirect_uri?: string,
        scope?: string,
        state?: string
    }) {
        super(data);
        this.responseType = data.response_type;
        this.clientId = data.client_id;
        this.redirectUri = data.redirect_uri || null;
        this.scope = data.scope || null;
        this.state = data.state || null;
    }

}
```

ClientCredentialsGrantTypeParameters.ts

```
import { IsOptional, IsString } from 'class-validator';
import { GrantTypeParameters } from './GrantTypeParameters';

export class ClientCredentialsGrantTypeParameters extends
GrantTypeParameters {

    @IsOptional()
    @IsString()
    scope: string | null;

    constructor(data: {
        grant_type: string,
        scope?: string
    }) {
        super(data);
        this.scope = data.scope || null;
    }

}
```

AuthorizationCodeGrantTypeParameters.ts

```
import { IsOptional, IsString } from 'class-validator';
import { GrantTypeParameters } from './GrantTypeParameters';

export class AuthorizationCodeGrantTypeParameters extends
GrantTypeParameters {

    @IsString()
    responseType: string;

    @IsString()
    clientId: string;

    @IsOptional()
    @IsString()
    redirectUri: string | null;

    @IsOptional()
    @IsString()
    scope: string | null;

    @IsOptional()
    @IsString()
    state: string | null;

    constructor(data: {
        grant_type: string,
        response_type: string,
        client_id: string,
        redirect_uri?: string,
        scope?: string,
        state?: string
    }) {
        super(data);
        this.responseType = data.response_type;
        this.clientId = data.client_id;
        this.redirectUri = data.redirect_uri || null;
        this.scope = data.scope || null;
        this.state = data.state || null;
    }

}
```

Password.ts

```
import { ParametersError } from '../Errors/ParametersError';
import { PasswordGrantTypeParameters } from
'../GrantTypeParams/PasswordGrantTypeParameters';
import { Token, TokenInterface } from '../models/Token';
import { GrantTypeHandler } from './GrantTypeHandler';

export class Password<U> extends GrantTypeHandler<U> {

    public async handle(params: PasswordGrantTypeParameters):
Promise<Token> {
        const { username, password } = params;
        if (!this.userStorage) {
            throw new Error('You should provide User Storage for Password
Grant Type');
        }
        if (!this.clientStorage) {
            throw new Error('You should provide Client Storage for Password
Grant Type');
        }
        const user = await this.userStorage.getUser(username, password);
        const client = await this.clientStorage.getClient(params.clientId);
        const scope = await this.scopeManager.validateScope(username,
client.id, params.scope!);
        if (!user) throw new ParametersError([ 'User not found' ]);
        const accessToken = await this.saveAccessToken();
        const refreshToken = await this.saveRefreshToken();
        return {
            accessToken,
            refreshToken,
            scope,
            client: { id: client.id, secret: client.secret },
        };
    }

    private async saveRefreshToken(): Promise<TokenInterface | undefined> {
        if (!this.tokenConfig.refreshToken) {
            return undefined;
        }
        const refreshTokenValue = await
this.tokenConfig.refreshToken.storage?.generateToken();
        const refreshToken = {
            value: refreshTokenValue!,
            expiresIn: this.tokenConfig.refreshToken.lifetime,
            expiresAt: new Date(Date.now() +
this.tokenConfig.refreshToken.lifetime * 1000),
        };
        await this.tokenConfig.refreshToken.storage.save(refreshToken);
        return refreshToken;
    }

    private async saveAccessToken(): Promise<TokenInterface> {
        const accessTokenValue = await
this.tokenConfig.accessToken.storage.generateToken();
        const accessToken = {
            value: accessTokenValue,
            expiresIn: this.tokenConfig.accessToken.lifetime,
            expiresAt: new Date(Date.now() +
this.tokenConfig.accessToken.lifetime * 1000),
        };
        await this.tokenConfig.accessToken.storage.save(accessToken);
        return accessToken;
    }
}
```

Додаток 3
Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

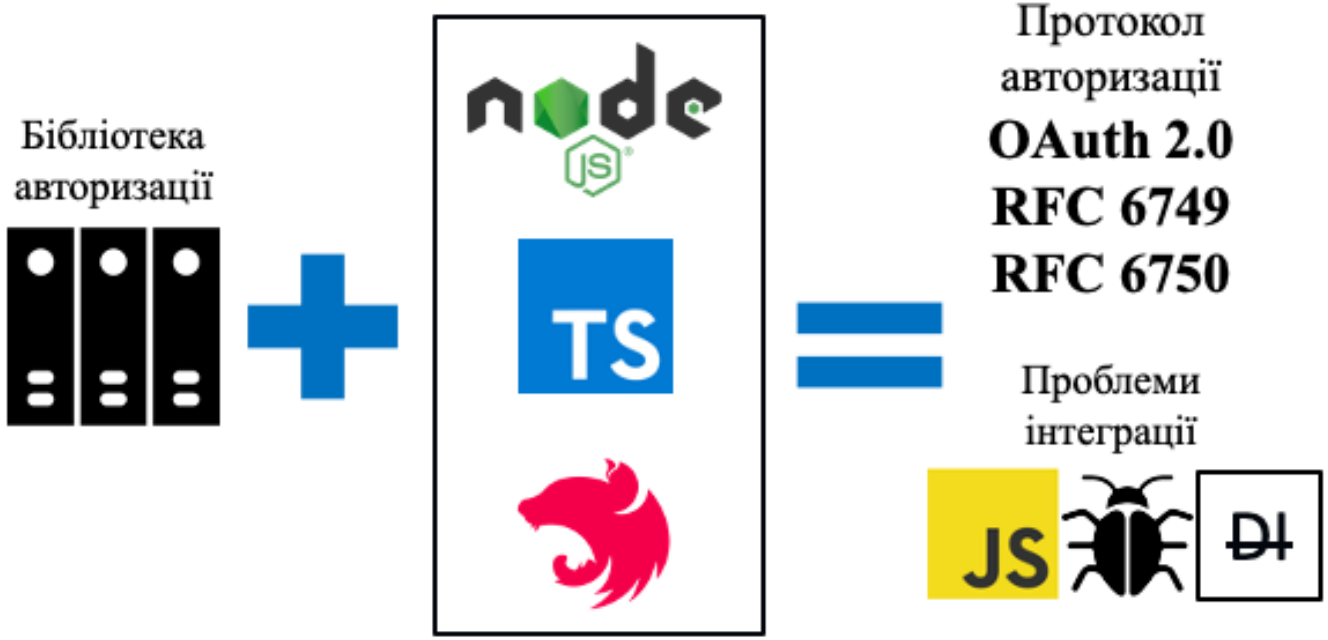
Програмна бібліотека для підтримки створення сервера авторизації на базі OAuth 2.0 для Node.js

Виконав: студент 4 курсу, групи КП-62 Бай Я.В.

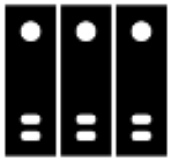
Керівник: доц.каф.ПЗКС, доц., к.т.н. Заболотня Т.М.

Київ – 2020

Актуальність

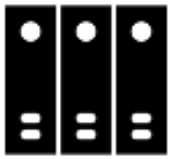


Існуючі рішення



node-oauth2-server

- ✓ Не залежить від типу сховища
- ✓ Не залежить від фреймворка
- ✓ Дозволяє розширювати типи грантів
- ✗ Використовує застарілий JavaScript
- ✗ Довгий час не підтримується



oauth2orize

- ✓ Не залежить від типу сховища
- ✓ Дозволяє розширювати типи грантів
- ✗ Використовує застарілий JavaScript
- ✗ Довгий час не підтримується
- ✗ Використовує специфічні інтерфейси
- ✗ Прив’язана до Passport framework

Постановка задачі

Мета: створити програмну бібліотеку для підтримки створення сервера авторизації на базі OAuth 2.0

Завдання:

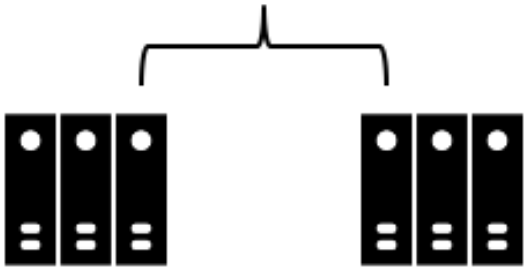
1. Забезпечити підтримку контейнерів впровадження залежностей (враховуючи контейнер Nest.js)
2. Забезпечити легкість інтегрування із будь-яким фреймворком за рахунок гнучких типів даних
3. Використовувати більш сучасну архітектуру та мову програмування для легкості підтримки у майбутньому

Розроблені програмні засоби

Бібліотека для авторизації

Демо-додаток для основної версії

Демо-додаток для Nest.js-версії



Основна версія

Nest.js-версія

Мета програмних засобів: забезпечення підтримки створення сервера авторизації на базі OAuth 2.0

Алгоритм авторизації

Алгоритм оновлення маркерів

Функції бібліотеки

1. Підтримка основних типів грантів:

Алгоритм для кожного описаний у окремому класі та обирається динамічно GrantTypeManager'ом

2. Можливість розширення типів грантів:

Обробник типу гранту визначається «на льоту». Реєстрація нового – за допомогою методів у класі GranTypeRegister

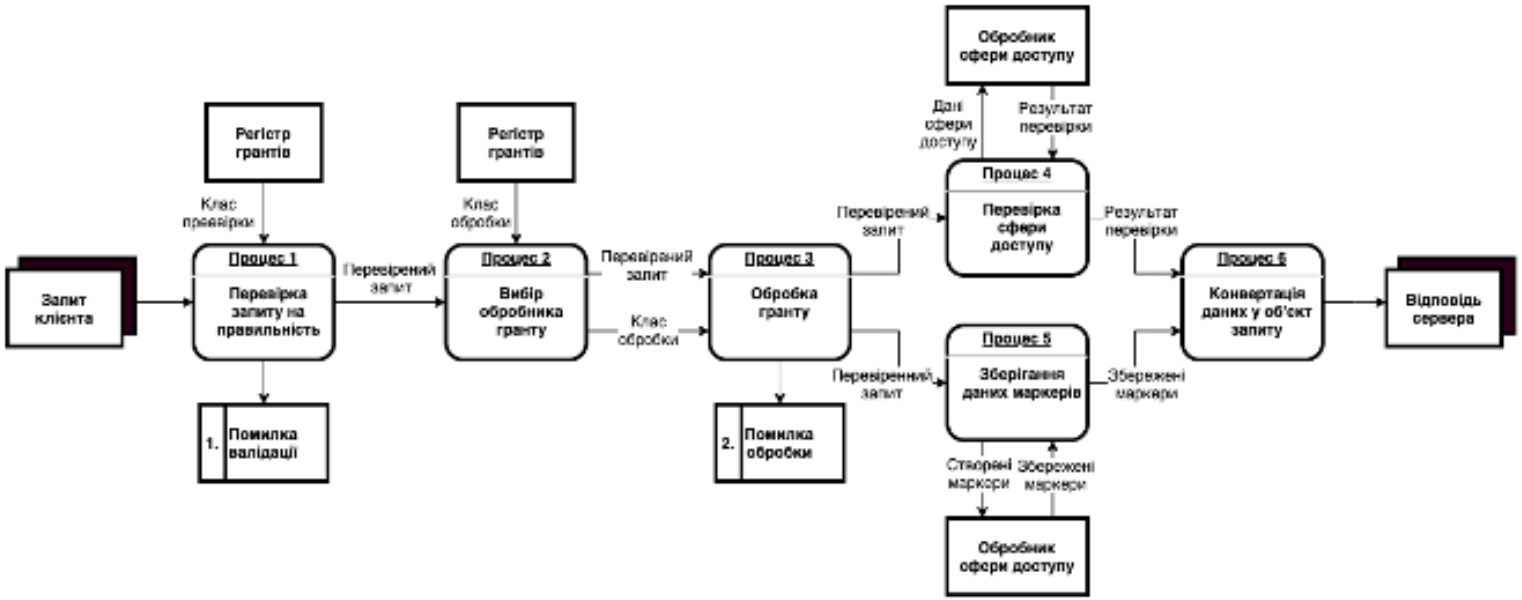
3. Незалежність від типу сховища:

Усі необхідні методи для зберігання даних абстрактні та реалізуються програмістом

4. Підтримка контейнерів впровадження залежностей

Реалізовано за прикладом бібліотек із групи TypeStack

Алгоритм роботи бібліотеки



* UML-діаграма потоку даних

Можливості демо-додатку

1. Реєєстрація за допомогою паролю:

Перевірка роботи бібліотеки із станадртним типом гранту

2. Реєстрація за допомогою соціальних мереж:

Перевірка роботи бібліотеки із власними типами гранту

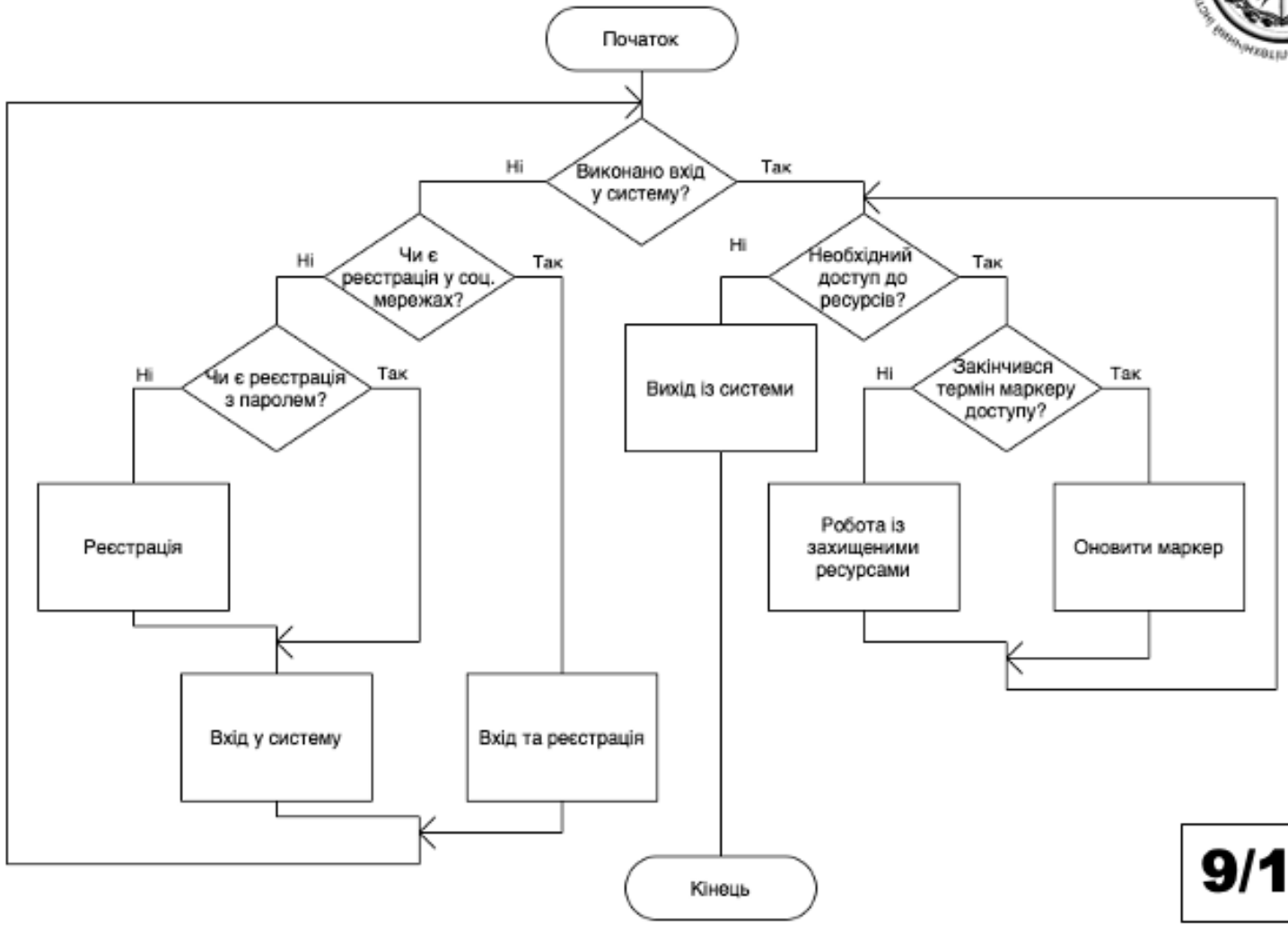
3. Вхід у систему

4. Оновлення маркерів доступу

5. Вихід із системи:

Перевірка функції видалення маркерів доступу

Алгоритм роботи демо-додатку

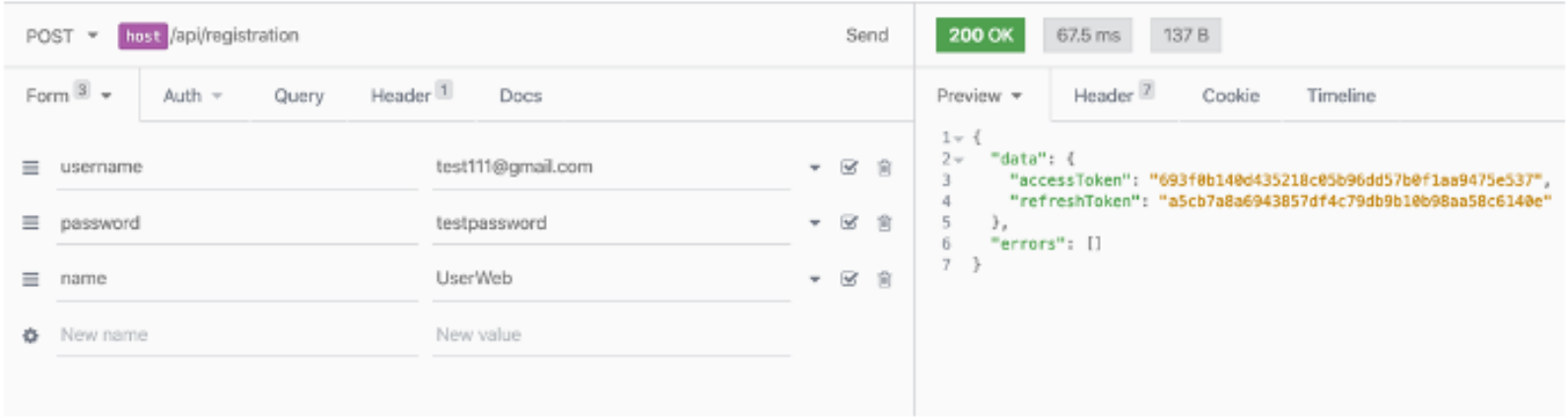


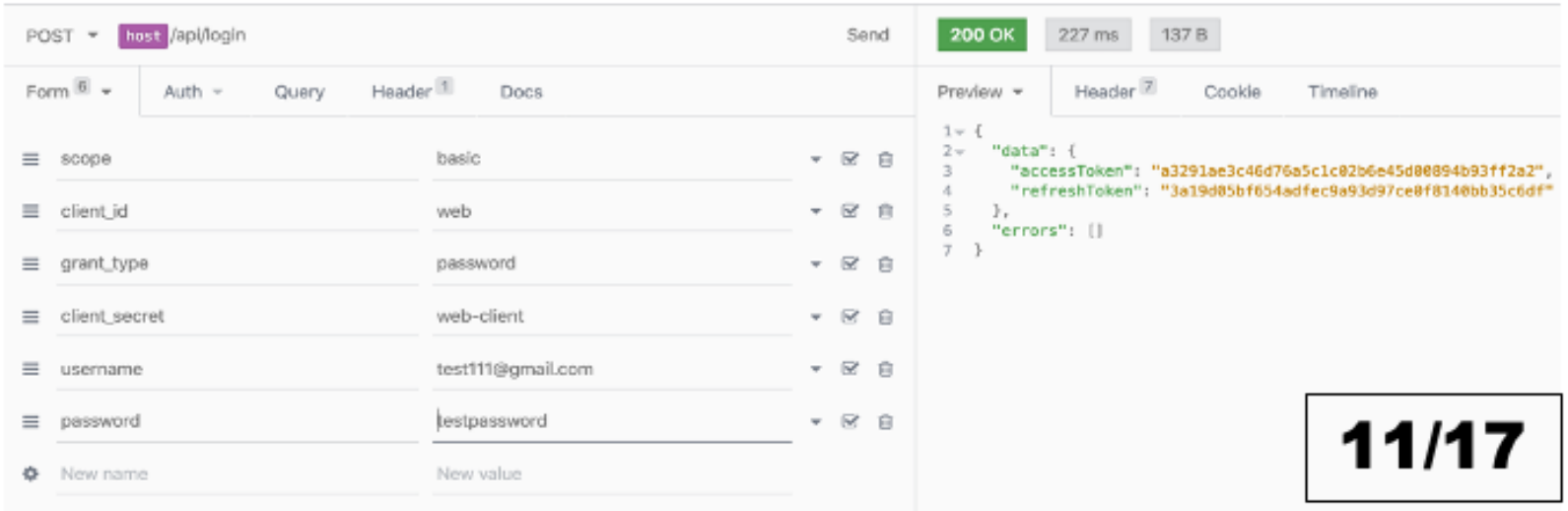
Приклади використання бібліотеки

Приклад кінцевої точки на вхід у систему

```
  @Post('login')
  @ApiBody({ description: 'Login input', type: LoginInput })
  @ApiOkResponse({ description: 'User has been login successfully', type:
AuthTokenResponse })
  public async login(@Req() request: Request, @Res() response: Response):
Promise<void> {
      await validateOrThrow(LoginInput, request);
      const token = await this.oauth.token(request, response);
      const response = AuthTokenResponse.fromObject({
          accessToken: token.accessToken,
          refreshToken: token.refreshToken as string,
      });
      sendResponse(result, response);
  }
```

Приклади використання бібліотеки





Використаний стек

- Мова програмування TypeScript
- Система контролю версій Git
- Пакетний менеджер NPM
- Менеджер версій Node.js NVM
- Статичний аналізатор коду TSLint
- Фреймворк Nest.js
- Пакет class-validator

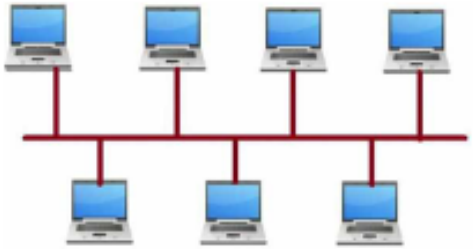
Апробація

МІЖНАРОДНИЙ МОЛОДІЖНИЙ ФОРУМ
«РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ У XXI СТОЛІТТІ»
КОНФЕРЕНЦІЯ
«ІНФОРМАЦІЙНІ ІНТЕЛЕКТУАЛЬНІ СИСТЕМИ»

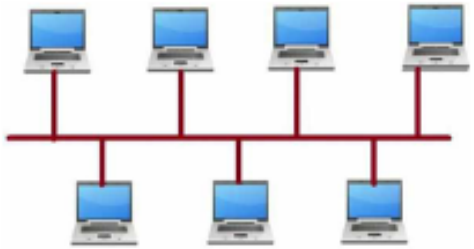
«Програмна інженерія. Інформаційні технології в освіті»

Результати опубліковані в вигляді тез доповіді.

Підготовлено пакет документів на отримання авторського свідоцтва.



NURE
Харківський національний університет радіоелектроніки



13/17

Висновки

Розглянуто проблему, поставлено задачу, досягнуто ціль побудови бібліотеки. Для цього було проаналізовано існуючі рішення, побудовано шлях досягнення цілей, проведено оцінку результатів.

Розроблені програмні бібліотеки забезпечують:

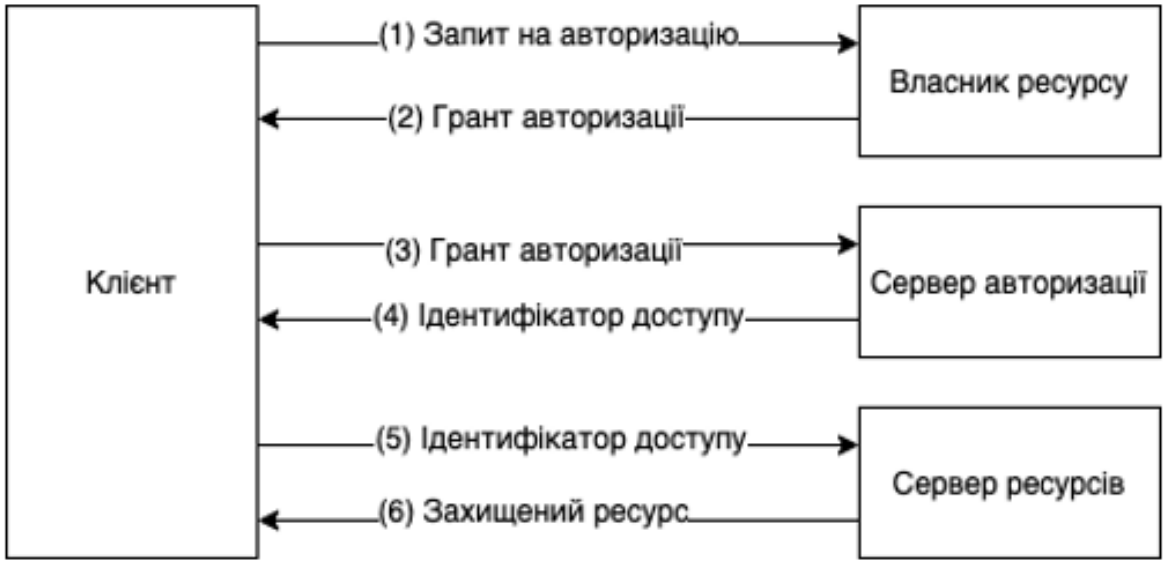
1. Підтримку контейнерів впровадження залежностей.
2. Використання із будь-яким фреймворком.
3. Легкість підтримки завдяки реалізації на сучасній мові.

Напрямки подальшої роботи:

1. Покращення архітектури згідно з потребами користувачів.
2. Зменшення кількості залежностей.

Дякую за увагу!

Алгоритм авторизації



Алгоритм оновлення маркерів

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2019 р.

ПРОГРАМНА БІБЛІОТЕКА ДЛЯ ПІДТРИМКИ СТВОРЕННЯ СЕРВЕРА АВТОРИЗАЦІЇ НА БАЗІ OAUTH 2.0 ДЛЯ NODE.JS

Програма та методика тестування

ДП.045440-04-51

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Тетяна ЗАБОЛОТНЯ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Ярослав БАЙ

2019

ЗМІСТ

1. ОБ’ЄКТ ВИПРОБУВАНЬ

Бібліотека для підтримки створення сервера авторизації на базі протоколу OAuth 2.0.

2. МЕТА ТЕСТУВАННЯ

У процесі тестування має бути перевірено наступне:

1) працездатність та коректна робота стандартних типів грантів згідно стандартів RFC 6749;
2) працездатність та коректна робота власних типів грантів;
3) коректна робота запису у користувацьке сховище даних;
4) забезпечення коректного оброблення виняткових ситуацій із належними кодами помилок;
5) відповідність розробки вимогам Технічного завдання.

3. МЕТОДИ ТЕСТУВАННЯ

Тестування відбувається за допомогою методу Grey Box Testing. Grey Box Testing – це методика тестування програмного продукту з частковим знанням внутрішньої роботи програми, метою такого тестування є пошук дефектів через помилки в логіці коду або неправильного використання програми.

Для тестування даної бібліотеки використовуються такі методи:

1) функціональне тестування на рівні Smoke Testing;
2) тестування продуктивності програмного забезпечення методами Failover and Recovery Testing (тестування на відмову та відновлення).
3) покриття коду.

4. ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Працездатність черги завдань перевіряється шляхом:

1) динамічного ручного тестування на відповідність функціональним вимогам та стандартам фреймворку авторизації;
2) статичного тестування коду;
3) тестування бібліотеки в різних операційних системах;
4) тестування бібліотеки для різних фреймворків платформи Nest.js;
5) тестування стабільності роботи при граничних умовах.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

ПРОГРАМНА БІБЛІОТЕКА ДЛЯ ПІДТРИМКИ СТВОРЕННЯ СЕРВЕРА АВТОРИЗАЦІЇ НА БАЗІ OAUTH 2.0 ДЛЯ NODE.JS

Керівництво програміста

ДП.045440-05-33

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Тетяна ЗАБОЛОТНЯ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Ярослав БАЙ

2020

ЗМІСТ

1. Призначення і умови застосування програми

1.1. Призначення програми

Дана бібліотека реалізує можливість створення сервера авторизації на базі протоколу OAuth 2.0 для оточення Node.js. Даний протокол авторизації є розповсюдженим та використовується як під час авторизації за допомогою сторонніх сервісів, так і під час авторизації на власному сервері.

Цей протокол авторизації відзначається серед інших тим, що в ньому визначаються 4 окремі ролі, які використовуються під час авторизації – клієнт, власник ресурсу, сервер авторизації та сервер ресурсів. Наявність даних ролей забезпечує високий рівень гнучкості даного протоколу, що дозволяє його використовувати, наприклад, під час авторизації за допомогою соціальних мереж.

Бібліотека розроблена в рамках даного дипломного проєкту відповідає основним вимогам, що постають перед бібліотекою для створення сервера авторизації. Основна відмінність даної бібліотеки від аналогів полягає в тому, що вона реалізована у такий спосіб, щоб її можна було максимально зручно використовувати із додатками, що реалізують архітектурні принципи SOLID.

Головні функції даної бібліотеки включають:

- реалізація повністю задовольняє стандартам фреймворку авторизації RFC 6749 та RFC 6750;
- підтримка будь-яких типів сховищ даних;
- незалежність від фреймворку;
- підтримка власних контейнерів впровадження залежностей.

1.2. Функції, що виконує програма

Основою функцією програми є авторизація та аутентифікація користувача у системі. Даний процес повністю налаштовується відповідно до потреб конкретного додатку.

Важливо відмітити, що дана програма не тільки підтримує стандартні типи грантів авторизації (authorization code,implicit, client credentials, resource owner password credentials), а й також надає можливість для створення власних типів грантів.

Отже, основними функціями даної бібліотеки є:

- підтримка типів грантів, описаних у стандарті RFC 6749;
- можливість додавати власні типи грантів;
- інкапсуляція логіки створення та обміну маркерів доступу;
- інкапсуляція логіки створення та обміну маркерів оновлення доступу;
- інкапсуляція логіки обробки даних клієнта;
- інкапсуляція логіки обробки даних сфери доступу.

1.3. Умови необхідні для виконання програми

Для роботи даної бібліотеки потрібні деякі сторонні залежності:

- @nestjs/core – один з центральних пакетів фреймворку Nest.js, який відповідальний за низькорівневі функції;
- @nestjs/common – пакет фреймворку Nest.js, у якому реалізовані функції, які є спільними для усіх модулів даного рішення;
- rxjs – бібліотека, що використовується для підтримки парадигми реактивного програмування та дане рішення використовується задля спрощення обробки асинхронного коду;
- reflect-metadata – програмний пакет, який надає можливість описувати метадані для полів класу у зручний та гнучкий спосіб й дане рішення базується на синтаксисі декораторів;

- class-validator – програмна бібліотека, яка дозволяє описувати логіку перевірки полів за допомогою декораторів.

Код бібліотеки був перевірений за допомогою лінтеру tslint та протестований за допомогою інструменту вибору версій Node NVM. Для них також потрібні деякі залежності:

- NVM – менеджер версій для програмної платформи Node.js;
- tslint – інструмент для перевірки коду на відповідність заздалегідь описаним у зручному форматі правилам;
- tslint-eslint-rules – розширення для tslint, що дозволяє використовувати додаткові правила із статичного аналізатору ESLint;
- typescript – програмний пакет, який забезпечує підтримку мови програмування TypeScript для додатків на базі Node.js.

Окрім цього, як і для будь-яких додатків на базі Node.js, для коректної роботи необхідно встановити саму програмну платформу Node.js. Варто відмітити, що в залежності від потреб конкретного додатку, може з’явитись необхідність у встановленні додаткових програмних пакетів для забезпечення підтримки сховища даних. Прикладом таких сховищ можуть бути СКБД PostgreSQL та розподілене сховище класу NoSQL для даних в оперативній пам'яті Redis.

2. Характеристики програми

2.1. Опис основних характеристик програми

2.1.1. Режим роботи програми

Дана бібліотека складається з двох частин: основної версії бібліотеки та версії для фреймворку Nest.js.

Головна логіка реалізації бібліотеки міститься у її основній частині. Версія для фреймворку Nest.js представляє собою обгортку над основною частиною, що написана із використанням самого фреймворку Nest.js. Таке рішення є необхідним для підтримки сумісності із контейнером впровадження залежностей даної платформи.

Основна версія програми має реалізацію програмного коду, що дозволяє використовувати користувацький контейнер впровадження залежностей. Даний код написаний згідно із тенденціями у бібліотеках програмного комплексу TypeStack.

Основна бібліотека складається з декількох модулів, які взаємодіють між собою:

- Request Validator – відповідає за валідацію даних;
- Grant Type Manager – відповідає за вибір типу гранту;
- Grant Type – відповідає за реалізацію конкретного типу гранту;
- Scope Manager – відповідає за перевірку наявних доступів у користувача;
- Token Storage – відповідає за зберігання маркерів доступу та маркерів оновлення доступу.

2.1.2. Контроль правильності виконання програми

Працездатність розробленої бібліотеки можна перевірити наступним чином:

- створити об’єкт конфігурації для даної бібліотеки, в якому реалізувати наступні поля:
  - tokenStorage – клас, що містить методи для зберігання маркерів;
  - userStorage – клас, відповідний за зберігання даних користувача;
  - clientStorage – клас, що відповідний за зберігання даних клієнта;
  - scopeManager – клас, що реалізує методи перевірки та підтвердження сфери можливих дій користувача;
- створити об’єкт класу OAuth2 та надати необхідну конфігурацію в якості параметра;
- реалізувати контрольні точки програми з використанням методів бібліотеки;
- за допомогою HTTP-запитів до додатку перевіряти дію контрольних точок програми.

Дана бібліотека не має вбудованого засобу логування, але користувач має можливість це реалізувати у власних методах класів, що викроистовуються для конфігурації бібліотеки.

2.2. Опис основних особливостей програми

Дана бібліотека надає користувачеві можливість налаштування за допомогою реалізації окремих класів, що потім передаються в якості параметра при створенні об’єкту класу OAuth2. Об’єкт конфігурації має наступні поля:

- tokenStorage – клас, що містить методи для зберігання маркерів;

- userStorage – клас, відповідний за зберігання даних користувача;
- clientStorage – клас, що відповідний за зберігання даних клієнта;
- scopeManager – клас, що реалізує методи перевірки та підтвердження сфери можливих дій користувача.

3. Звернення до програми

3.1. Запуск програми

Програма може бути запущена за допомогою команд програми npm, що використовує реалізацію команд із package.json. Для того, щоб запустити додаток, треба спочатку скомпілювати код на мові програмування TypeScript у мову JavaScript.

Команда запуску компіляції коду

```
npm run build
```

Команда запуску додатка

```
npm run start
```

Окрім цього, для покращення досвіду використання бібліотеки можна використовувати команду, яка автоматично буде заново компілювати код та перезапускати додаток при кожній зміні у сирцевому коді .

Команда запуску автоматичного моніторингу змін у коді:

```
npm run nodemon
```

Також форматування бібліотеки перевірено за допомогою статичного аналізатору tslint. Запустити цей інструмент можна за допомогою власне

команди tslint або за допомогою команди npm, реалізація якої описана у package.json:

Команди запуску tslint:

```
tslint -c tslint.json -p tsconfig.json
npm run tslint
```

3.2. Виконання програми

3.2.1. Робота модулю, що відповідає за перевірку запиту

Модуль Request Valdiator потрібен для перевірки даних запиту на коректність відповідно до типу гранту, який передається клієнтом. Даний модуль фактично містить тільки один метод, який і виконує перевірку даних. В цьому методі на основі типу гранту обирається відповідний клас валідації. Дані класи містять поля із декораторами class-validator, що дозволяє легко та гнучко писати нові правила для власних типів гранту.

3.2.2. Робота модулю вибору обробника типу гранту

Модуль Grant Type Manager відповідає за вибір необхідного для отриманого типу гранту обробника. Тут мається центральний клас даної бібліотеки – Grant Type Register. Даний клас ставить у відповідність строкове представлення типу гранту його класу валідації та класу обробника. Даний клас містить наступні методи:

- getParams – метод, який повертає клас із валідацією параметрів типу гранту;
- getHandler – метод, який повертає клас-спадкоємець типу GrantTypeHandler, який визначає обробку типу гранту;
- register – метод, який дозволяє додавати нові типи грантів у реєстр, приймає тип гранту у строковому представленні та клас валідації й відповідний обробник..

3.2.3. Робота модулю-обробника типу гранту

Модуль GrantType складається з набору класів, які сумісно забезпечують обробку як типів грантів, що описані у стандарті, так і власних типів грантів. Тут міститься абстрактний клас GrantTypeHandler, на основі якого базуються усі інші класи. Тут також містяться окремі класи для кожного із стандартних типів гранту – implicit, client credentials, resource owner credentials та authorization code. Реалізація цих класів повністю відповідає стандарту RFC 6749, стосовно алгоритмічного підґрунтя та обробки помилок.

4. Вхідні та вихідні дані

4.1. Організація вхідної інформації, що використовується програмою

Бібліотека приймає на вхід конфігурацію, що містить класи, які реалізуються користувачем. Тут також маються поля конфігурації. Прикладом такого поля є tokenConfig, в якому міститься інформація про те, скільки часу потрібно зберігати маркери доступу та маркери оновлення доступу. Тут також важливо відмітити окремий для кожного типу маркера екземпляр класу TokenStorage, що дозволяє користувачу визначити методи окремо для токену доступу та окремо для токену оновлення доступу.

4.2. Організація вихідної інформації, що використовується програмою

Частини даної бібліотеки передають, приймають, виконують та повертають результат завдання. Результат роботи самої бібліотеки можна побачити у об’єкті інтерфейсу Response.

5. Повідомлення

Якщо під час роботи бібліотеки були отримані певні невідповідності, то вона поверне помилку згідно із специфікацією. Прикладом таких помилок можуть бути:

- *Невірний запит* – у запиті відсутній необхідний параметр, міститься недійсне значення параметра, певний параметр включається більше, ніж один раз, або присутні інші помилки у запиті;
- *Неавторизований клієнт* – клієнт не має права запитувати авторизацію код за допомогою цього методу;
- *У доступі відмовлено* – власник ресурсу або сервер авторизації заборонили цей запит;
- *Непідтримуваний тип відповіді* – сервер авторизації не підтримує отримання коду авторизації за допомогою цього методу;
- *Невірна запрошена сфера дій* – запрошений діапазон недійсний, невідомий або неправильно сформований;
- *Помилка серверу* – сервер авторизації зіткнувся з несподіваною умовою, яка завадила йому виконати запит (цей код помилки потрібен, оскільки помилку сервера з кодом 500 не можна повернути клієнтові через переспрямування HTTP);
- *Тимчасово недоступний* – наразі сервер авторизації не може обробити запит через тимчасове перевантаження чи технічне обслуговування сервера (цей код помилки потрібен, оскільки HTTP-помилку «Сервіс Недоступний» неможливо повернути до клієнта через переспрямування HTTP).