МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІМ. Ю.М. ПОТЕБНІ ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Особливості побудови бібліотеки компонентів для створення інформаційних систем**

Виконав: студент 2 курсу, групи 8.1212-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
В.І. Кириченко
(ініціали та прізвище)
Керівник доцент, кандидат ф.-м. наук В.І. Попівщий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІМ. Ю.М. ПОТЕБНІ
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Т.В. Критська_

"_01" жовтня 2023_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Кириченко Владиславу Ігоровичу
(прізвище, ім'я, по батькові)

1.Тема роботи Особливості побудови бібліотеки компонентів для створення інформаційних систем

керівник роботи ___доцент, кандидат ф.-м. наук Попівщий Василь Іванович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від ___09.10.2023, № 1577-с

2. Строк подання студентом кваліфікаційної роботи 01.03.2024
3. Вихідні дані магістерської роботи
• комплект нормативних документів ;
• технічне завдання до роботи.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
• огляд та збір літератури стосовно теми кваліфікаційної роботи;
• огляд та аналіз існуючих рішень та аналогів;
• дослідження питання створення бібліотек компонентів та методів реалізації;
• створення програмного продукту та його опис;
• перелік вимог для роботи програми;
• дослідження поставленої проблеми та розробка висновків та пропозицій.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
12 слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата
		Завдання прийняв

7. Дата видачі завдання 01.10.2023

КАЛЕНДАРНИЙ ПЛАН

з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	02.09-10.09.23	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	11.09-12.09.23	виконано
3	Аналіз проблематики	13.09-14.09.23	виконано
4	Дослідження «цибулевої» архітектури в розробці веб-додатків	15.09-20.09.23	виконано
5	Дослідження архітектурних та технологічних особливостей бібліотеки компонентів	21.09-26.09.23	виконано
6	Узгодження подальших дій з науковим керівником	27.09-28.09.23	виконано
7	Створення інструментарію для розробки додатків в об’єктно – орієнтованому стилі	29.09-13.10.23	виконано
8	Розробка додатку в об’єктно-орієнтованому стилі за допомогою Node.js та фреймворку Express	14.10-16.10.23	виконано
9	Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження	17.10-19.10.23	виконано
10	Обробка отриманих результатів	20.10-09.11.23	виконано
11	Аналіз отриманих результатів	10.11-17.11.23	виконано
12	Оформлення звіту	17.11-20.02.24	виконано

Студент _____ Кириченко В.І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Попівщий В.І.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 111

Рисунків: 14

Таблиць: 0

Джерел: 26

Кириченко В.І. Особливості побудови бібліотеки компонентів для створення інформаційних систем: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник В. І. Попівщий. Запоріжжя : ЗНУ, 2023. 111 с.

Мета і завдання дослідження полягають у створенні ефективного інструментарію за допомогою якого буде можлива зручна, швидка та ефективна розробка додатків в об’єктно-орієнтованому стилі додатків за допомогою Node.js та фреймворку Express.

У процесі дослідження був проведений аналіз особливостей та існуючих рішень для розробки додатків в об’єктно-орієнтованому стилі за допомогою JavaScript, Typescript, Express, Node.js. Окрім того було проаналізовано особливості та існуючі рішення для розробки додатків в об’єктно-орієнтованому за допомогою інших об’єктно-орієнтованих мов програмування. Визначено, що для вирішення задачі побудови додатків за допомогою об’єктно-орієнтованого підходу існуючі рішення або критично застарілі, або занадто громіздкі та можуть бути оверхедом для виконання задач.

На основі проведеного дослідження була розроблена open source бібліотека для побудови додатків в об’єктно-орієнтованому стилі за допомогою Node.js та фреймворку Express.js.

Ключові слова: open source бібліотека, фреймворк, JavaScript, ECMAScript, Typescript, Express, Node.js, Рефлексія.

SUMMARY

Pages: 111

Figures: 14

Tables: 0

Sources: 26

Kyrychenko V.I. Features of Building a Component Library for Creating Information Systems: qualification work of the master of specialty 121 "Software Engineering"/ Science head V.I. Popivshchyi. Zaporizhzhia: ZNU, 2023. 111 p.

The aim and objectives of the research involve the development of an effective toolkit enabling convenient, fast, and efficient application development in an object-oriented style using Node.js and the Express framework.

During the research, an analysis of features and existing solutions for developing applications in an object-oriented style using JavaScript, Typescript, Express, Node.js was conducted. Additionally, the peculiarities and existing solutions for developing applications in an object-oriented style using other object-oriented programming languages were examined. It was determined that existing solutions for building applications using an object-oriented approach are either critically outdated or overly cumbersome and may introduce unnecessary overhead for task execution.

Based on the conducted research, an open-source library was developed for building applications in an object-oriented style using Node.js and the Express.js framework.

Keywords: open-source library, framework, JavaScript, ECMAScript, Typescript, Express, Node.js, Reflection.

ЗМІСТ

ВСТУП 9
РОЗДІЛ 1 АРХІТЕКТУРИ В РОЗРОБЦІ ВЕБ-ЗАСТОСУНКІВ 12
1.1 «Цибулева» архітектура в розробці веб-додатків 12
1.1.1 Порівняння "цибулевої" архітектури з традиційними моделями 12
1.1.2 Інверсія контролю (IoC) та її роль у "Цибулевій" архітектурі 17
1.2 Сучасні методи комунікації у веб – додатках 20
1.2.1 REST API: Кращі практики та розповсюджені помилки 21
1.2.2. WebSockets: Реальні приклади використання в додатках 23
1.3 Рефлексія та її вплив на веб-додатки 25
1.3.1 Використання рефлексії в C#: Переваги та ризики 26
1.3.2 Рефлексія у JavaScript/TypeScript: Інноваційні підходи 29
1.4 Мікросервісна архітектура у веб-додатках 32
1.4.1 Основи мікросервісної архітектури та її переваги 33
1.4.2 Виклики при впровадженні мікросервісів 35
1.5 Безпека у веб-додатках 37
1.5.1 Огляд загальних вразливостей веб-додатків 37
1.5.2 Стратегії захисту та криптографічні підходи 43
1.6 Висновки з розділу 1 51
РОЗДІЛ 2 АРХІТЕКТУРНІ ТА ТЕХНОЛОГІЧНІ ОСОБЛИВОСТІ БІБЛІОТЕКИ КОМПОНЕНТІВ 52
2.1 Структурні аспекти бібліотеки компонентів 52
2.1.1 Модульність та повторне використання компонентів 53
2.1.2 Принципи інкапсуляції та ізоляції в бібліотеці 55
2.2 Технологічні підходи в бібліотеці компонентів 57
2.2.1 Використання шаблонів проектування в компонентах 60
2.2.2 Застосування принципів SOLID та DRY у розробці 66
2.3 Фреймворки та інструменти для розробки бібліотеки 69
2.3.1 Аналіз популярних фреймворків для розробки веб-компонентів 69

2.3.2 Роль та вибір інструментів збірки та розгортання 73
2.4 Оптимізація та ефективність компонентів 77
2.4.1 Техніки оптимізації продуктивності компонентів 77
2.4.2 Метрики оцінки ефективності компонентів 79
2.5 Безпека та надійність у бібліотеці компонентів 81
2.5.1 Забезпечення безпеки компонентів 81
2.5.2 Стратегії тестування та валідації компонентів 83
2.6 Інтеграція бібліотеки компонентів в інформаційні системи 85
2.6.1 Підходи до інтеграції та взаємодії з іншими системами 86
2.6.2 Випадки використання та приклади реалізацій 87
2.7 Висновки з розділу 2 88
РОЗДІЛ 3 РОЗРОБКА ВИБІР СЕРЕДОВИЩА, ФРЕЙМВОРКІВ ТА СТОРОННІХ БІБЛІОТЕК ДЛЯ РЕАЛІЗАЦІЇ ПРОЕКТУ 90
3.1 Налаштування апаратного середовища 90
3.1.1 Visual Studio Code 90
3.1.2 WebStorm 91
3.1.3 TypeScript 93
3.2 Фреймворки 94
3.2.1 Node.js 94
3.2.2 Express.js 95
3.3 Сторонні бібліотеки 96
3.3.1 `body-parser` 96
3.3.2 CORS 97
3.3.3 `reflect-metadata` 98
3.4 Висновки з розділу 3 99
4.1 Декоратори класів та функцій 100
4.2 Імплементація DI контейнера для бібліотеки компонентів 103
4.3 Аналіз отриманих результатів 105
4.4 Висновки з розділу 4 106
ВИСНОВКИ 108

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 109

ВСТУП

Актуальність теми

Розробка серверної частини за допомогою мови програмування JavaScript та технології Node.js вже є загальноприйнятною та часто використовується у всьому світі. Найпопулярнішим фреймворком, який для цього використовується є Express. Як і сама мова програмування, Express дозволяє будувати додатки за допомогою функціонального підходу до програмування, що в деяких випадках є не зручним, а інколи навіть не є прийнятним. Тому постає необхідність в створення механізмів, за допомогою яких, досягається можливість розроблювати додатки за допомогою об’єктно-орієнтованого підходу.

Мета і завдання дослідження

Мета і завдання дослідження полягають у створенні ефективного інструментарію за допомогою якого буде можлива зручна, швидка та ефективна розробка додатків в об’єктно-орієнтованому стилі додатків за допомогою Node.js та фреймворку Express.

Завдання дослідження:

- проаналізувати можливості «цибулинної» архітектури в розробці веб-додатків;
- проаналізувати сучасні методи комунікації у веб-додатках;
- розглянути поняття Рефлексії та її вплив на веб-додатки;
- проаналізувати можливості мікросервісної архітектури у веб-додатках;
- розглянути питання безпеки у веб-додатках;
- розглянути архітектурні та технологічні особливості бібліотеки компонентів;
- побудувати інструментарію для розробки додатків в об’єктно-орієнтованому стилі;

- розробити open source бібліотеку для побудови додатків в об’єктно-орієнтованому стилі.

Об’єкт дослідження

Об’єктом дослідження є JavaScript, Typescript, Express, рефлексія.

Предмет дослідження

Предметом дослідження є побудова інструментарію для розробки додатків в об’єктно-орієнтованому стилі.

Методи дослідження

Для вирішення поставленої задачі використовуються наступні методи дослідження:

1. Аналіз особливостей та існуючих рішень для розробки додатків в об’єктно-орієнтованому стилі за допомогою JavaScript, Typescript, Express, Node.js.
2. Аналіз особливостей та існуючих рішень для розробки додатків в об’єктно-орієнтованому за допомогою інших об’єктно-орієнтованих мов програмування.
3. Аналіз побудованого додатку за допомогою розробленого інструментарію.

Наукова новизна одержаних результатів

Наукова новизна одержаних результатів дослідження полягає у тому, що для вирішення задачі побудови додатків за допомогою об’єктно-орієнтованого підходу існуючі рішення або критично застарілі, або занадто громіздкі та можуть бути оверхедом для виконання задач.

Практичне значення одержаних результатів

Практичне значення одержаних результатів дослідження полягає у тому, що була розроблена open source бібліотека для побудови додатків в об’єктно-орієнтованому стилі.

Апробація одержаних результатів

Результати дослідження були представлені на XVI науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2023» [1], а також на конференції здобувачів вищої освіти, аспірантів та молодих вчених Актуальні питання сталого науково-технічного та соціально-економічного розвитку регіонів України [2].

Глосарій

JavaScript – динамічна, об'єктно-орієнтована прототипна мова програмування. Реалізація стандарту ECMAScript. Найчастіше використовується для створення сценаріїв вебсторінок, що надає можливість на боці клієнта (пристрої кінцевого користувача) взаємодіяти з користувачем, керувати браузером, асинхронно обмінюватися даними з сервером, змінювати структуру та зовнішній вигляд вебсторінки.

ECMAScript – стандарт мови програмування, затверджений міжнародною організацією ECMA згідно зі специфікацією **ECMA-262**

Typescript – мова програмування, представлена Microsoft восени 2012; позиціонується як засіб розробки вебзастосунків, що розширює можливості JavaScript, а саме додає номінальну типізацію.

Express – фреймворк для розробки серверних частини веб застосунків для Node.js.

Node.js – платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript.

Рефлексія – це процес, під час якого комп’ютерна програма може відслідкувати та модифікувати власну структуру і поведінку під час виконання.

РОЗДІЛ 1 АРХІТЕКТУРИ В РОЗРОБЦІ ВЕБ-ЗАСТОСУНКІВ

1.1 «Цибулева» архітектура в розробці веб-додатків

В цей розділ присвячено опису базової інформації про «цибулеву» архітектуру, її порівнянню з деякими традиційними моделями. Окрім того буде розглянуто питання інверсії контролю (IoC) та її ролі в «цибулевій» архітектурі.

1.1.1 Порівняння "цибулевої" архітектури з традиційними моделями

"Цибулева" архітектура, також відома як "Цибулевидна архітектура", є архітектурним підходом, в якому програмне забезпечення розглядається як набір вкладених шарів або "оболонок", де кожен шар взаємодіє тільки з прилеглим шаром. Це сприяє високій модульності та легкості управління складністю системи. [3].

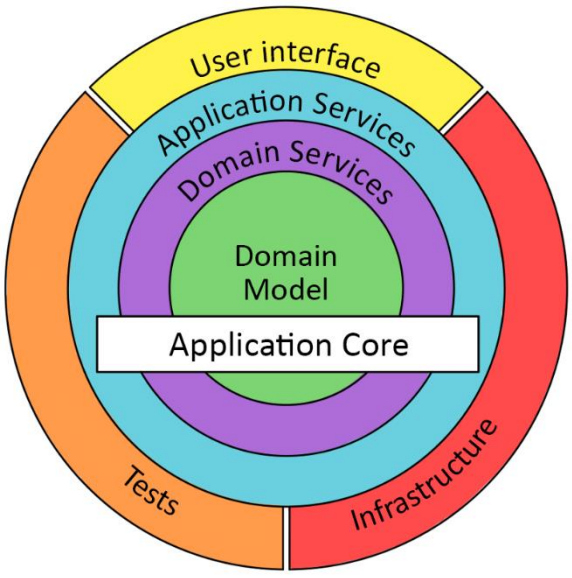

Рисунок 1 — *Концептуальна схема «цибулевої» архітектури*

На вигляд архітектура подібна до цибулі, де кожен шар представляє собою оболонку, інкапсульовану в іншому шарі. Кожен шар відповідає за конкретний аспект додатку, і вони розміщені в порядку з зовнішнього до внутрішнього, представляючи різні рівні абстракції.

Зовнішній (Outer) Шар: Найзовнішний шар містить вхідні точки та презентаційні елементи. Він є інтерфейсом для взаємодії з користувачем та іншими зовнішніми системами. Цей шар може включати веб-інтерфейс, API, консольні інтерфейси чи інші методи комунікації.

Застосування (Application) Шар: Другий шар містить бізнес-логіку та правила застосування. Він виконує обробку даних та реалізацію бізнес-правил. Цей шар може включати сервіси, які взаємодіють із зовнішніми джерелами, а також обробку даних і валідацію.

Інфраструктурний (Infrastructure) Шар: Третій шар містить всю інфраструктуру та технічні деталі, такі як бази даних, фреймворки, зовнішні служби та засоби зберігання. Цей шар ізолює бізнес-логіку від конкретних технічних деталей реалізації.

Доменний (Domain) Шар: Четвертий шар є ядром або доменом додатку. Він містить основні сутності, агрегати, репозиторії та сервіси, які визначають основну функціональність та правила бізнес-логіки.

Цибулева архітектура покликана забезпечити принцип інверсії залежностей (Dependency Inversion Principle), де високорівневі модулі не залежать від низькорівневих модулів, а обидва типи залежать від абстракцій. Це полегшує тестування і сприяє розділенню відповідальності.

Основні переваги цієї архітектури включають підвищену гнучкість, тестованість, розширюваність та утримання коду. Кожен шар може бути реалізований та тестований окремо, що спрощує розробку та підтримку додатку.

Цибулева архітектура і традиційні архітектури різняться у своєму підході до розподілення компонентів та залежностей в програмному забезпеченні. Можна порівняти Цибулева архітектура з деякими традиційними архітектурами, такими як Layered Architecture та MVC (Model-View-Controller):

Layered Architecture (Шарова архітектура):

Layered Architecture також розглядає програмне забезпечення як набір шарів [4].

Основні шари у Layered Architecture зазвичай включають:

Presentation Layer (Шар представлення), який відповідає за відображення інформації користувачу та обробку користувацького введення. Містить компоненти, такі як інтерфейс користувача, контролери (controllers) або презентери (presenters).

Business Logic Layer (Шар бізнес-логіки), що містить бізнес-логіку, яка визначає основні правила та операції системи. Цей шар відповідає за обробку бізнес-правил, валідацію та прийняття рішень.

Data Access Layer (Шар доступу до даних) має на меті забезпечити взаємодію зі збереженими даними, виконуючи операції збереження та витягування. Він містить компоненти для взаємодії з базами даних, файловою системою чи іншими механізмами збереження даних.

Infrastructure Layer (Шар інфраструктури) який містить загальні сервіси та утиліти, необхідні для функціонування додатку, такі як логування, аутентифікація, конфігурація і таке інше. Взаємодіє з зовнішніми системами та бібліотеками.

Рисунок 2 — *Концептуальна схема шарової архітектури*

Кожен шар може містити лише компоненти, які відносяться до його відповідальності. Залежно від конкретної реалізації, може бути різне число шарів, але загальна ідея полягає в тому, щоб створити модульну структуру, яка полегшує розширення, тестування та зміну компонентів.

У Цибулевої архітектури також є шари, але вони організовані так, щоб уникнути залежностей від зовнішніх шарів (наприклад, залежностей від бази даних або фреймворків).

MVC (Model-View-Controller):

MVC розділяє додаток на три основні компоненти: Модель (Model), Представлення (View) і Контролер (Controller).

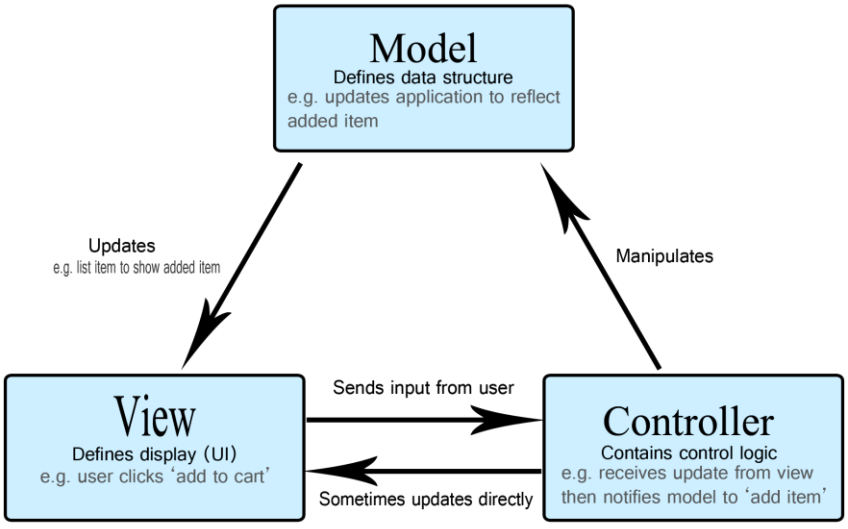

Рисунок 3 — *Концептуальна схема MVC архітектури*

MVC - це архітектурний шаблон, який використовується для розробки програмного забезпечення та додатків [5]. Його основна мета - розділити логічні компоненти програми для полегшення розробки, управління та розширення коду. Модель-Вид-Контролер включає три основні компоненти:

Модель (Model): Компонент відповідає за управління даними та бізнес-логікою програми. Модель представляє об'єкти даних, обробку даних та взаємодію з базою даних або іншими механізмами для зберігання даних. Вона не знає про інтерфейс користувача або як дані відображаються.

Вид (View): відповідає за відображення даних користувачеві та інтерфейсу користувача. Вид отримує дані від моделі та відображає їх користувачеві. Він також відповідає за обробку користувацьких вводу та відправлення їх до контролера.

Контролер (Controller): відповідає за обробку користувацьких вводу, управління потоком даних між моделлю та видом, а також за взаємодію з іншими частинами програми. Контролер приймає введення від користувача, обробляє його та ініціює необхідні зміни в моделі та вигляді.

Принцип розділення на компоненти дозволяє забезпечити модульність та підтримуваність коду. Якщо, наприклад, потрібно змінити спосіб відображення даних, вам може знадобитися лише змінити код в вигляді, не торкаючись моделі чи контролера.

У Цибулевої архітектури також є розділення на компоненти, але це робиться так, щоб бізнес-логіка була незалежною від конкретних деталей інтерфейсу або механізмів збереження даних.

Головна ідея Цибулевої архітектури це уникнути залежностей від конкретних імплементацій зовнішніх шарів і робити систему більш незалежною та придатною для тестування.

Основна відмінність Цибулевої архітектури від традиційних полягає в тому, що вона прагне створити ізольований ядро бізнес-логіки, навколо якого розташовані різні шари. Це полегшує тестування, розширення та зміну бізнес-логіки, не торкаючись зовнішніх компонентів.

1.1.2 Інверсія контролю (IoC) та її роль у "Цибулевій" архітектурі

Інверсія контролю (IoC) - це принцип проектування програмного забезпечення, який передбачає, що керування потоком виконання програми не здійснюється самою програмою, а відбувається через зовнішній фреймворк або контейнер. Цей принцип дозволяє створювати більш гнучкі та змінювані системи, оскільки вони стають менш залежними від конкретних деталей реалізації та можуть бути легше масштабованими та модифікованими.

У "цибулевій" архітектурі, IoC використовується для створення системи з різким розділенням внутрішніх та зовнішніх компонентів, що нагадує структуру цибулі. Основна ідея полягає в тому, щоб розділити програму на шари, де кожен шар відповідає за конкретний аспект функціональності [6]. Ці шари відокремлюють бізнес-логіку від інфраструктури та залежностей.

Розглянемо основні шари "цибулевої" архітектури та роль IoC в кожному з них:

Шар додатку (Application Layer), як ми казали вже, це самий внутрішній шар, де знаходиться бізнес-логіка.

IoC використовується для впровадження залежностей, необхідних для виконання бізнес-логіки. Наприклад, сервіси, які використовуються для обробки бізнес-логіки, можуть бути впроваджені через інтерфейси.

Лістинг 1 *Використання IoC на прикладі сервісу для керування користувачами додатку*.

```
public interface IUserService
{
    void AddUser(User user);
    User GetUserById(int userId);
}

public class UserService : IUserService
{
    private readonly IUserRepository _userRepository;

    public UserService(IUserRepository userRepository)
```

```
        {
            _userRepository = userRepository;
        }

        public void AddUser(User user)
        {
            // Використання репозиторію для додавання
користувача
            _userRepository.Add(user);
        }

        public User GetUserById(int userId)
        {
            // Використання репозиторію для отримання
користувача за ідентифікатором
            return _userRepository.GetById(userId);
        }
    }
```

Шар сервісів (Service Layer), який містить сервіси, які забезпечують конкретні функції для додатку.

IoC використовується для впровадження зовнішніх залежностей, таких як репозиторії баз даних або зовнішні сервіси.

Лістинг 2 *Приклад визначення інтерфейсу та його реалізація для репозиторію користувача.*

```
    public interface IUserRepository
    {
        void Add(User user);
        User GetById(int userId);
    }

    public class UserRepository : IUserRepository
    {
        // Реалізація методів репозиторію для взаємодії з
базою даних або іншими джерелами
        // (це може бути реалізація Entity Framework,
Dapper, тощо)
    }
```

Шар інфраструктури (Infrastructure Layer) забезпечує взаємодію програми з навколишнім середовищем (бази даних, зовнішні API, файлові системи тощо).

IoC використовується для впровадження інфраструктурних залежностей, таких як конкретні реалізації баз даних, HTTP клієнти тощо.

Лістинг 3 *Приклад реалізації інфраструктурного сервісу для взаємодії з базою даних.*

```
    public class DatabaseContext
    {
        // Контекст бази даних, наприклад, Entity Framework
DbContext
    }

    public class DatabaseUserRepository : IUserRepository
    {
        private readonly DatabaseContext _context;

        public DatabaseUserRepository(DatabaseContext
context)
        {
            _context = context;
        }

        public void Add(User user)
        {
            // Додавання користувача до бази даних
            _context.Users.Add(user);
            _context.SaveChanges();
        }

        public User GetById(int userId)
        {
            // Отримання користувача з бази даних за
ідентифікатором
            return _context.Users.Find(userId);
        }
    }
```

Лістинг 4 *Приклад Конфігурація IoC контейнера в ASP.NET Core (в класі Startup конфігуруємо контейнер IoC для впровадження залежностей.*

```
    public void ConfigureServices(IServiceCollection
services)
    {
        // Додавання служб для IoC контейнера
        services.AddTransient<IUserRepository,
DatabaseUserRepository>();
        services.AddTransient<IUserService, UserService>();

        // Додавання служби для контексту бази даних
        services.AddDbContext<DatabaseContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("Def
aultConnection")));
    }
```

Таким чином, IoC використовується для автоматичного впровадження залежностей при створенні об'єктів. Коли ми викликаємо методи сервісу користувачів, контейнер IoC автоматично постачає відповідні реалізації репозиторію та інших залежностей, дозволяючи нам зосередитися на бізнес-логіці додатку, забезпечуючи при цьому слабку залежність між компонентами системи.

Отже Застосування IoC у "цибулевій" архітектурі дозволяє забезпечити слабку залежність між різними частинами системи. Це робить систему більш гнучкою, дозволяє змінювати окремі компоненти без необхідності зміни всієї системи. Також це полегшує тестування та підтримку коду, оскільки розділені компоненти можуть бути замінені мокованими об'єктами для ефективного тестування окремих частин системи.

1.2 Сучасні методи комунікації у веб – додатках

Цей розділ присвячено розгляду деяких методів комунікації у веб-застосунках, як важливої складової їх розробки.

1.2.1 REST API: Кращі практики та розповсюджені помилки

REST (Representational State Transfer) є архітектурним стилем для розробки веб-сервісів, який базується на принципах, що покладені в роботу мережі. Основною ідеєю REST є використання стандартних протоколів, таких як HTTP, і ресурсів (зображених у вигляді URI), щоб забезпечити зручний та ефективний обмін даними між клієнтом і сервером [7].

Основні принципи REST:

1. Архітектурні обмеження REST:

Безстановність (Statelessness): Кожен запит від клієнта до сервера повинен містити всю необхідну інформацію для зрозуміння та обробки запиту. Сервер не повинен зберігати стан клієнта між запитами.

Кешування (Caching): Дозволяє клієнтам зберігати локальні копії ресурсів для покращення продуктивності та зменшення навантаження на сервер.

Ієрархічність інтерфейсу (Uniform Interface): Всі ресурси повинні бути доступні через стандартний інтерфейс, зокрема, використання URI для ідентифікації ресурсів та використання стандартних методів HTTP.

Однорівневість системи (Layered System): Система може бути розглядана як набір рівнів, при цьому кожен рівень може бачити тільки той рівень, з яким він безпосередньо взаємодіє.

2. URI та Ресурси:

У REST всі ресурси (наприклад, дані чи послуги) представлені у вигляді URI (Uniform Resource Identifier). Отже це чітка структура, бажано використовувати семантично багатий URI. Наприклад, /users для списку користувачів, а /users/{id} для конкретного користувача.

3. HTTP Методи та Представлення Ресурсів:

Використання HTTP методів: Для взаємодії з ресурсами використовуються стандартні HTTP методи такі як GET для отримання даних, POST для створення, PUT або PATCH для оновлення, DELETE для видалення.

Представлення Ресурсів: Ресурси можуть бути представлені у різних форматах, таких як JSON або XML. Клієнт та сервер повинні домовитися про формат, який вони обоє оброблять.

Кращі практики при роботі з REST API:

Чітка структура URI: URI повинні бути легко зрозумілі та поважати ієрархічну структуру. Наприклад, /users для отримання списку користувачів.

Використання HTTP методів згідно призначенню: Потрібно використовувати GET для отримання даних, POST для створення, PUT або PATCH для оновлення, та DELETE для видалення та ніяк інакше.

Версіонування API: Потрібно вказувати версію API в URI (наприклад, /v1/users). Це дозволяє вносити зміни до API, не ламаючи існуючих інтеграцій та забезпечує сумісність при зміні API.

Використання HTTP статус кодів: Необхідно повертати відповідний HTTP статус код для кожного запиту. Наприклад, 200 OK для успішного запиту або 404 Not Found, якщо ресурс не знайдено.

Аутентифікація та Авторизація: Потрібно забезпечте безпеку API за допомогою надійних методів аутентифікації та авторизації, таких як токени або ключі API.

Документація: Бажано надавати докладну документацію для API, включаючи приклади запитів та відповідей.

Моніторинг: Вкрай необхідно забезпечити систему моніторингу для виявлення помилок, високого навантаження та інших проблем.

Поширені помилки:

Неправильне використання HTTP методів: Звісно ж невірне використання методів може призвести до некоректної роботи або навіть до видалення даних.

Відсутність аутентифікації та авторизації: Зазвичай недостатній захист API призводить до небажаного доступу та витоку конфіденційної інформації.

Невірна обробка помилок: Необхідно повертати зрозумілі повідомлення про помилку та відповідний HTTP статус код для полегшення відладки.

Неправильне кешування: Неправильне кешування може призвести до некоректного оновлення даних на клієнтській стороні. Потрібно використовувати заголовки кешування для управління кешем і запобігання застарілості даних.

Неправильне використання SSL/TLS: Невірне налаштування безпеки може призвести до витоку конфіденційної інформації під час обміну даними між клієнтом та сервером.

Недостатнє тестування: Потрібно забезпечити широкомасштабне тестування API, включаючи тести для різних методів та сценаріїв.

1.2.2. WebSockets: Реальні приклади використання в додатках

WebSocket – це протокол, який був описаний у специфікації RFC 6455 [8], для забезпечення способу обміну даними між браузером і сервером через постійне з’єднання в режимі реального часу. За цим протоколом потрібні дані можна передавати в обох напрямках (от браузера к серверу та навпаки) у вигляді “пакетів”, без розриву з’єднання та додаткових HTTP-запитів.

Основна його перевага в тому, що він відмінно підходить для сервісів, що потребують безперервного обміну даними, наприклад онлайн-ігри, системи торгівлі в реальному часі і не тільки.

1. Основи:

Протокол: WebSockets - це протокол зв'язку, який надає повнодуплексне (full-duplex) з'єднання через один TCP-порт. Це дозволяє обмінюватися даними між клієнтом та сервером в обидві сторони одночасно.

2. Процес встановлення з'єднання:

Handshake: Розпочинається з відправлення HTTP-запиту на сервер для ініціювання WebSockets-з'єднання.

Апгрейд: Після успішного handshake, з'єднання підвищується до WebSockets, і подальший обмін даними відбувається за допомогою цього протоколу.

3. Архітектура:

Клієнт-серверна модель: WebSockets використовує клієнт-серверну архітектуру, де клієнт та сервер можуть ініціювати відправку даних одне одному.

4. API та Протокол:

API для роботи в браузері: В браузерах для роботи з WebSockets використовується JavaScript API, що дозволяє взаємодіяти з WebSocket-з'єднанням.

Протокол WebSocket: WebSocket має власний протокол рівня транспорту, що включає в себе заголовки для контролю фреймів та статус коди.

5. Використання:

Реальний час: WebSockets широко використовуються для реалізації режиму реального часу в веб-застосунках, таких як онлайн-чати, спільна робота над документами тощо.

Оптимізація трафіку: Для додатків, які потребують постійного оновлення даних, WebSockets може бути більш ефективним за традиційні HTTP-запити.

6. Плюси та Мінуси:

Плюси: Повнодуплексність, низька затримка, ефективність для передачі стрімів даних.

Мінуси: Потребує підтримки як на стороні клієнта, так і на стороні сервера, не підтримується у всіх мережевих середовищах.

Лістинг 5 *Приклад використання в JavaScript.*

```
    // Встановлення з'єднання
    const socket = new
WebSocket('ws://example.com/socket');

    // Обробка подій
    socket.addEventListener('open', (event) => {
```

```
    console.log('WebSocket відкрито');
});

socket.addEventListener('message', (event) => {
    console.log('Отримано повідомлення:', event.data);
});

// Відправка даних
socket.send('Hello, сервер!');
```

До реальних прикладів використання можна віднести наступне:

Онлайн-геймінг: В інтерактивних онлайн-іграх WebSockets використовуються для забезпечення миттєвого обміну даними між гравцями та сервером. Це включає в себе синхронізацію рухів, обмін повідомленнями та інші події у режимі реального часу. Гравці можуть спілкуватися, взаємодіяти та отримувати оновлення гри без необхідності постійних HTTP-запитів.

Онлайн-чат та коллаборативна спільна робота: Веб-чати та коллаборативні платформи для спільної роботи в реальному часі використовують WebSockets для швидкого обміну повідомленнями між користувачами. Коли один користувач відправляє повідомлення, інші користувачі отримують його миттєво без необхідності оновлення сторінки чи використання традиційних запитів.

Фінансові та торгові платформи: У фінансових додатках, таких як онлайн-торгові платформи, WebSockets використовуються для трансляції ринкових оновлень в реальному часі. Це дозволяє трейдерам отримувати актуальну інформацію про ціни, здійснювати угоди та отримувати оновлення про ринкові умови без затримок, що є критично важливим у фінансовому секторі.

1.3 Рефлексія та її вплив на веб-додатки

В цьому розділі будуть розглянуті питання рефлексії в різних мовах програмування, її переваги та ризики використання.

1.3.1 Використання рефлексії в C#: Переваги та ризики

Рефлексія C# (або «відображення» в термінах Microsoft) — це, в першу чергу, процес що дозволяє виявити типи (об’єктів типу Type) в той час, як виконується додаток. Будь який додаток C# складається з об’єктів, які реалізують певні класи та інтерфейси, а також з методів, властивостей об’єктів та інших елементів [9]. Рефлексія (відображення) це механізм, який надає можливість програмістам отримувати інформацію про об'єкти та типи в часі виконання. Це дозволяє вам в динаміці взаємодіяти з кодом, отримувати доступ до членів класів, створювати та викликати об'єкти, здійснювати маніпуляції з атрибутами і багато іншого.

Механізм рефлексії в C# базується на просторі імен System.Reflection. Цей простір імен містить класи та інші типи, які дозволяють вам отримувати інформацію про об'єкти та типи в часі виконання.

Лістинг 6 *Приклад отримання інформації про тип використовуючи об'єкт Type.*

```
using System;

class Program
{
    static void Main()
    {
        // Отримання типу для класу String
        Type stringType = typeof(string);

        // Виведення інформації про тип
        Console.WriteLine($"Повне ім'я типу:
{stringType.FullName}");
        Console.WriteLine($"Чи є це значимий тип?
{stringType.IsValueType}");
        Console.WriteLine($"Чи є це клас?
{stringType.IsClass}");

        // Отримання методів для типу
        MethodInfo[] methods = stringType.GetMethods();
        Console.WriteLine($"Кількість методів:
{methods.Length}");
    }
}
```

Лістинг 7 Приклад створення об’єкту та виклика методу за допомогою рефлексії.

```
using System;
using System.Reflection;

class Program
{
    static void Main()
    {
        // Створення об'єкта типу String
        Type stringType = typeof(string);
        object myString =
Activator.CreateInstance(stringType, new object[] { "Hello,
Reflection!" });

        // Виклик методу "Length"
        MethodInfo lengthMethod =
stringType.GetMethod("Length");
        int length = (int)lengthMethod.Invoke(myString,
null);

        Console.WriteLine($"Довжина рядка: {length}");
    }
}
```

Лістинг 8 *Приклад роботи з властивостями та полями.*

```
using System;
using System.Reflection;

class Person
{
    public string Name { get; set; }
    private int age;

    public Person(string name, int age)
    {
        Name = name;
        this.age = age;
    }
}

class Program
{
    static void Main()
```

```
        {
            Type personType = typeof(Person);

            // Отримання інформації про властивість
            PropertyInfo nameProperty =
personType.GetProperty("Name");
            Console.WriteLine($"Тип властивості Name:
{nameProperty.PropertyType}");

            // Отримання значення властивості
            Person person = new Person("John", 25);
            string nameValue =
(string)nameProperty.GetValue(person);

            Console.WriteLine($"Значення властивості Name:
{nameValue}");
        }
    }
```

Переваги використання рефлексії в C#:

Гнучкість та Динамічність: Рефлексія надає можливість динамічно отримувати інформацію про типи та об'єкти, що робить код більш гнучким та адаптивним.

Аналіз та Метапрограмування: Рефлексія використовується для аналізу програми та створення нового коду на основі цієї інформації. Це може бути корисно при створенні загальних фреймворків або інструментів метапрограмування.

Тестування та Налагодження: Рефлексія може бути корисною під час тестування, оскільки дозволяє отримувати доступ до приватних членів класів та викликати методи для тестування різних сценаріїв.

Автоматизація: Рефлексія може спрощувати автоматизацію рутинних завдань, таких як створення об'єктів, зчитування та запис атрибутів, обробка подій тощо.

Ризики використання рефлексії в C#:

Втрата продуктивності: Операції рефлексії зазвичай менш ефективні в порівнянні з прямим викликом методів чи доступом до властивостей. Використання рефлексії може призвести до падіння продуктивності програми.

Помилки на етапі виконання: Оскільки інформація отримується в часі виконання, можуть виникнути помилки, які було б важко виявити на етапі компіляції. Це може призвести до неперевірених типів та інших проблем в програмі під час виконання.

Неперевірені зміни: Зміни в структурі класів можуть призвести до виникнення помилок в програмі, яка використовує рефлексію, оскільки ця інформація може застаріти після змін.

Потенційні проблеми з безпекою: Використання рефлексії може порушити принципи ізоляції та безпеки, оскільки воно дозволяє отримувати доступ до приватних членів класу та обходити обмеження безпеки.

Підтримка та Зручність Обслуговування: Код, який використовує рефлексію, може бути менш зрозумілим для інших програмістів та важче обслуговуватися, оскільки не відображається структура програми на етапі компіляції.

Використання рефлексії в C# повинно бути обрано з урахуванням конкретних вимог і обмежень проекту, з усвідомленням можливих ризиків та зусиль, які можуть бути витрачені на управління цими ризиками.

1.3.2 Рефлексія у JavaScript/TypeScript: Інноваційні підходи

Рефлексія в JavaScript та TypeScript визначається здатністю до динамічного аналізу та взаємодії з об'єктами та їх структурами під час виконання програми. В цих мовах відсутня строга система типізації, тому рефлексія набуває особливого значення для роботи з об'єктами та їхніми характеристиками.

Використання Рефлексії в JavaScript:

Object Reflection: Використовуючи методи, доступні в Object, можна отримати інформацію про властивості об'єктів [10].

Лістинг 9 *Приклад використання методів, доступних в Object.*

```
const obj = { key: 'value' };
console.log(Object.keys(obj)); // Виведе: ['key']
```

Function Reflection: Функції також мають свої методи, що надають доступ до інформації про параметри та інші атрибути:

Лістинг 10 *Приклад використання методів функцій.*

```
function exampleFunction(param1, param2) {
    // ...
}

console.log(exampleFunction.length); // Кількість параметрів (виведе: 2)
```

Prototype Reflection: За допомогою Object.getPrototypeOf, можна отримати доступ до прототипу об'єкта.

Лістинг 11 *Приклад отримання доступу до прототипу об’єкта.*

```
const obj = {};
console.log(Object.getPrototypeOf(obj)); // Виведе: {}
```

Використання Рефлексії в TypeScript:

В TypeScript, як надмножина JavaScript, зберігає можливості рефлексії, але рефлексія дещо розширюється завдяки його системі статичної типізації та можливостям роботи з типами [11].

Type Information: TypeScript використовує статичну систему типів для надання інформації про типи під час компіляції.

Лістинг 12 *Приклад отримання інформації про типи під час компіляції.*

```
interface MyInterface {
    myProperty: number;
}
```

```
const obj: MyInterface = { myProperty: 42 };
```

Decorators: Декоратори в TypeScript дозволяють змінювати поведінку класів та їх елементів за часу компіляції.

Лістинг 13 *Приклад зміни поведінки.*

```
function log(target: any, key: string) {
    console.log(`Method ${key} is called`);
}

class Example {
    @log
    myMethod() {
        // ...
    }
}
```

TypeScript Reflect API: TypeScript включає Reflect API, яке надає більше можливостей для взаємодії з об'єктами та їх метаданими.

Лістинг 14 *Приклад зміни поведінки.*

```
class Example {
    private _privateField: string = 'Hello';

    get privateField() {
        return this._privateField;
    }
}

const instance = new Example();
console.log(Reflect.get(instance, 'privateField')); // Виведе: 'Hello'
```

Розширені можливості TypeScript Reflect API:

- Reflect.get(target, propertyKey[, receiver]): Отримання значення властивості об'єкта.

- Reflect.set(target, propertyKey, value[, receiver]): Встановлення значення властивості об'єкта.
- Reflect.has(target, propertyKey): Перевірка існування властивості.
- Reflect.deleteProperty(target, propertyKey): Видалення властивості об'єкта.
- Інші методи для роботи з прототипами та іншими аспектами об'єктів.

Ці інструменти дозволяють розширені можливості взаємодії та аналізу об'єктів та їх метаданих в TypeScript порівняно з чистим JavaScript.

Інновації в TypeScript порівняно з C#:

Статична та Динамічна Типізація: TypeScript комбінує в собі статичну та динамічну типізацію, що робить його більш гнучким для розробників. C# використовує переважно статичну типізацію.

Декоратори: TypeScript використовує декоратори, що є засобом для зміни або розширення поведінки класів та їх членів за часу компіляції. Це є інновацією порівняно з C#.

Інтеграція з JavaScript-кодом: TypeScript дозволяє використовувати JavaScript-код безпосередньо, що робить його більш гнучким для інтеграції з існуючим кодом.

Типові Аліаси та Збіжні Типи: TypeScript має декілька концепцій, таких як типові аліаси та збіжні типи, які дозволяють створювати більш ефективні та читабельні типи даних.

1.4 Мікросервісна архітектура у веб-додатках

В цьому розділі буде розглянуто основи мікросервісної архітектури в розробці веб-додатків та розібрані основні виклики що до її впровадження.

1.4.1 Основи мікросервісної архітектури та її переваги

Мікросервісна архітектура - це підхід до розробки програмного забезпечення, в якому програма розбивається на невеликі і незалежні сервіси, які взаємодіють між собою через відкриті API. Кожен мікросервіс виконує конкретну функцію та може бути розгорнутий, масштабований і оновлений незалежно від інших сервісів [12].

Основні характеристики які є одночасно і перевагами мікросервісної архітектури:

Незалежність сервісів: Кожен мікросервіс може бути розроблений, розгорнутий та масштабований окремо. Це дозволяє ефективно керувати розвитком і підтримкою кожного сервісу.

Легкість вдосконалення та розширення: Додавання нового функціоналу або оновлення існуючого може бути здійснене швидше і без впливу на інші частини системи.

Масштабованість: Можливість масштабування окремих сервісів, що дозволяє ефективно реагувати на збільшення обсягу обробки даних або завдань.

Полегшення розвитку: Кожен мікросервіс може бути розроблений різними командами, навіть мовами програмування та технологіями.

Відкритість і взаємодія через API: Спілкування між сервісами відбувається через стандартизовані API, що дозволяє їм взаємодіяти незалежно від того, якою мовою програмування чи технологією вони використовують.

Покращена стійкість і відновлення: Якщо один мікросервіс виходить з ладу, інші можуть продовжувати роботу, забезпечуючи більшу стійкість системи в цілому.

Різні бази даних для кожного сервісу: Кожен мікросервіс може використовувати власну базу даних, що дозволяє оптимізувати структуру даних для конкретного сервісу.

Моніторинг і логування: Завдяки розділенню на сервіси, можливо вести моніторинг та логування для кожного з них окремо, що полегшує виявлення та вирішення проблем.

Мікросервісна архітектура дозволяє побудувати гнучкі та легкі для розвитку системи, але вона також вимагає добре налагодженої інфраструктури та ефективного управління конфігурацією для забезпечення успішного функціонування.

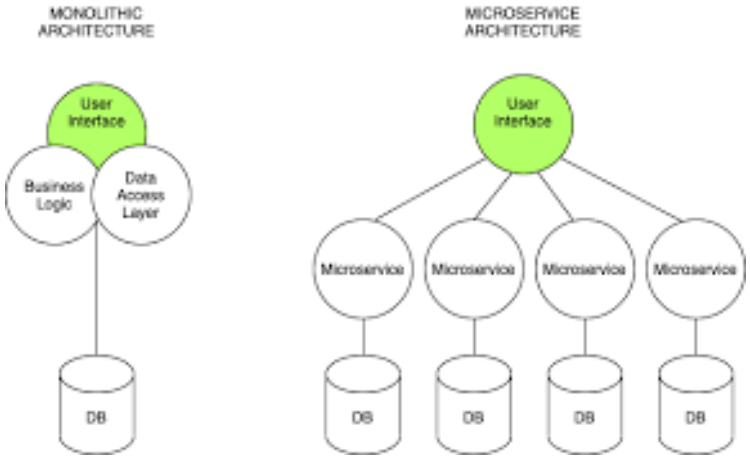

Рисунок 4 — *Візуальне порівняння монолітної та мікросервісної архітектур.*

Завдяки своїй специфіці мікросервісна архітектура має неабияку популярність:

Netflix використовує мікросервісну архітектуру для своєї великої платформи стрімінгового відео. Сервіси, такі як управління користувачами, пошук, рекомендації, обробка платежів та інші, реалізовані як окремі мікросервіси. Це дозволяє Netflix швидко реагувати на зміни, масштабувати окремі компоненти та запускати новий функціонал без впливу на всю систему.

Uber використовує мікросервіси для своєї платформи замовлення та управління таксі. Сервіси, що використовуються для керування користувачами, обробки платежів, геолокації та інші є окремими мікросервісами. Це дозволяє Uber швидко реагувати на зміни в попиті, вдосконалювати окремі частини системи та підтримувати велику кількість транзакцій.

Amazon використовує мікросервісну архітектуру для свого хмарного рішення Amazon Web Services (AWS). Кожна послуга AWS (наприклад, S3 для зберігання, EC2 для обчислень, Lambda для функцій) є окремим мікросервісом. Це дозволяє клієнтам використовувати тільки ті послуги, які їм потрібні, і масштабувати їхнє використання згідно з потребами.

Spotify використовує мікросервісну архітектуру для свого стрімінгового сервісу. Окремі сервіси відповідають за рекомендації, обробку платежів, управління користувачами та інші аспекти. Така архітектура допомагає Spotify ефективно керувати розвитком та масштабуванням своєї платформи.

1.4.2 Виклики при впровадженні мікросервісів

Хоча мікросервісна архітектура має багато переваг, вона також вносить свої виклики та проблеми, особливо при впровадженні та експлуатації. Ось деякі з них:

Складність управління конфігурацією: З великою кількістю мікросервісів стає складніше вести контроль за конфігурацією і версіями кожного сервісу.

Велика кількість взаємодій: Збільшена кількість взаємодій між сервісами може призводити до виникнення проблем в мережевій комунікації та затримок у відповідях.

Складність тестування та відлагодження: Тестування та відлагодження стають більш складними через розподіленість системи та потребу в ефективних інструментах для симуляції середовища.

Проблеми з транзакційною консистентністю: Забезпечення консистентності між різними мікросервісами може виявитися викликом, особливо у випадку розподілених транзакцій.

Справжній вимір динамічних залежностей: На початкових етапах розробки часто буває важко повністю розібратися в усіх динамічних залежностях між сервісами.

Велика кількість інфраструктурних вирішень: Кожен мікросервіс може вимагати власної інфраструктури, що може призводити до великої кількості інструментів та технологій.

Управління версіями та сумісністю: Синхронізація версій та управління сумісністю між різними версіями сервісів може бути викликом при швидкому внесенні змін.

Безпека та доступність: Забезпечення безпеки та високої доступності для всіх мікросервісів може вимагати значних зусиль та ресурсів.

Моніторинг та діагностика: Моніторинг та діагностика мікросервісів може бути складнішою задачею через їхню роздільність та розподіленість.

Культурні зміни: Впровадження мікросервісної архітектури може вимагати культурних змін у розробницькому колективі та організаційних процесах. Для реалізації програми, що працює на основі мікросервісної архітектури, знадобиться команда фахівців, які мають відповідні знання та навички. Як мінімум, до проекту потрібно буде додатково залучати DevOps-інженера.

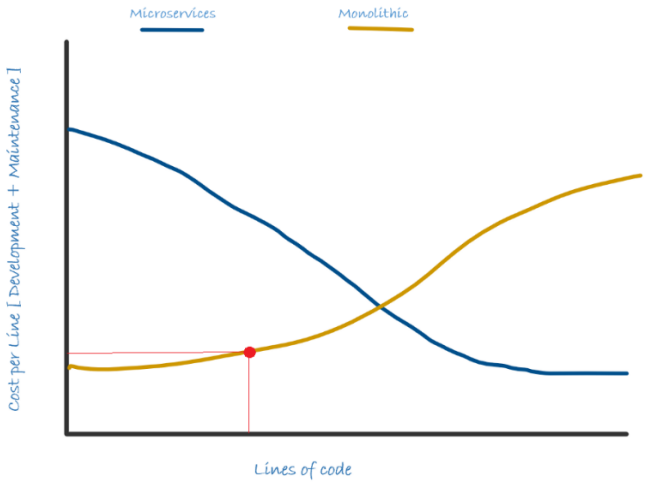

Рисунок 5 — *Візуальне порівняння витрат на розробку та супровід застосунків монолітної та мікросервісних архутектур*

Важливо зазначити, що багато з цих викликів можна подолати правильним плануванням, ефективним управлінням та використанням відповідних інструментів та технологій.

1.5 Безпека у веб-додатках

Цей розділ присвячено огляду загальних вразливостей притаманних веб-додаткам та методам і стратегіям захисту інформації включно з криптографічними підходами.

1.5.1 Огляд загальних вразливостей веб-додатків

Нажаль веб-додатки мають велику кількість вразливостей, але розглянуті будуть лише найбільш поширені з них [13].

Injection Attacks - це група атак, які використовують вразливості в обробці вхідних даних у веб-додатках для виконання небажаних дій або отримання несанкціонованого доступу. Однією з найпоширеніших форм атаки в рамках Injection Attacks є SQL Injection (SQLi) та Cross-Site Scripting (XSS).

SQL Injection (SQLi): Основний принцип полягає в тому, що в атаках SQL Injection, злоумисник вводить дані, які містять SQL-код, в ті місця, де програма взаємодіє з базою даних. Це може стати можливим через форми на веб-сайтах, URL-параметри, кукі-файли та інші механізми введення.

Cross-Site Scripting (XSS): Основний принцип полягає в тому, що в атаках XSS, зловмисник вбудовує скриптовий код у веб-сторінку, який потім виконується в браузері іншого користувача. Це може бути виконано через введені дані, які виводяться без відповідної фільтрації на веб-сторінках.

Брак Безпеки Аутентифікації та Авторизації веб-додатка може призвести до серйозних проблем з безпекою, оскільки ці механізми визначають, хто має доступ до ресурсів та які дії вони можуть виконувати. Вразливості в цих процесах можуть дозволити несанкціонованим користувачам отримати доступ до конфіденційної інформації або виконати небажані операції. Основні аспекти цих вразливостей:

Недоліки управління сесією: Якщо сесійні ідентифікатори недостатньо випадкові або легко передбачувані, зловмисник може легко підмінити чи викрасти сесійні дані.

Недоліки управління правами: Неправильна настройка прав доступу може дозволити користувачам виконувати дії, до яких вони не повинні мати доступ.

Несправжня аутентифікація: Якщо механізми аутентифікації не вірно перевіряють ідентичність користувача, зловмисники можуть використовувати методи атаки, такі як перехоплення сесій або фішинг.

Брутфорс атаки: Якщо існують слабкі обмеження на кількість спроб введення паролю, зловмисник може використовувати брутфорс або словникові атаки для вгадування паролів.

Недостатня перевірка ідентичності: В деяких випадках, системи можуть недостатньо перевіряти ідентичність користувача при використанні паролів, токенів чи інших механізмів аутентифікації.

Некоректна обробка помилок: Якщо система неправильно обробляє помилки аутентифікації чи авторизації, це може дати зловмиснику інформацію, яка допоможе вдосконалити атаку.

Cross-Site Request Forgery (CSRF або XSRF) - це тип атаки на безпеку веб-додатків, при якій зловмисник використовує авторизовану сесію користувача для виконання небажаної дії в системі без його згоди. Атака CSRF використовує довіру, яку веб-додаток має до користувача та його браузера.

Основні характеристики атаки CSRF:

Авторизована сесія (користувач повинен бути авторизований у веб-додатку та мати дію в системі, яку можна викликати з браузера), використання довіри браузера (зловмисник створює підступну сторінку або вбудовує зловмисний код на існуючу, яка завантажується в браузер користувача), виклик небажаної дії (зловмисник викликає дію (наприклад, зміна паролю, виконання фінансового переказу) в іншій вкладці браузера, не дававши користувачеві знати про це), використання автоматичного виконання запитів (зловмисник

використовує автоматичне виконання запитів, яке властиве браузерам (наприклад, використовуючи <img> або <script> теги)).

Основні кроки атаки CSRF:

Створення підступної сторінки: Зловмисник створює сторінку або вбудовує код на існуючу сторінку, яка має небезпечні запити до веб-додатка.

Отримання жертви: Зловмисник намагається привернути жертву на свою підступну сторінку, наприклад, надсилаючи посилання через електронну пошту, соціальні мережі або інші канали.

Виконання запиту в контексті жертви: Коли жертва відкриває підступну сторінку, браузер автоматично виконує небажані дії в контексті авторизованого користувача.

Виконання атаки: Атака може включати зміну налаштувань акаунту, виконання фінансових операцій, видалення даних тощо.

Security misconfigurations (неправильна конфігурація безпеки) веб-додатків вважається однією з найпоширеніших вразливостей, яка може призвести до серйозних проблем з безпекою. Ця вразливість виникає, коли система, сервер або додаток налаштовані неправильно, дозволяючи несанкціонований доступ, витік чутливої інформації та інші види атак.

Основні аспекти security misconfigurations:

Неправильна конфігурація сервера: може призвести до витоку конфіденційної інформації або дозволити несанкціонованому користувачеві отримати доступ до критичних ресурсів. Як наприклад неправильно налаштований веб-сервер, який виводить важливі файлові ресурси або включає дебаг-інформацію.

Некоректна конфігурація самого додатка: може викликати проблеми безпеки, такі як неправильне керування доступом, витік чутливої інформації та інші атаки. Прикладом може бути неправильне налаштування прав доступу до бази даних або неправильна обробка сесій. Некоректна конфігурація може

призвести до витоку чутливої інформації через неправильно встановлені правила безпеки (неправильні налаштування дозволів) або некоректно налаштовані файли конфігурації.

Неправильна обробка помилок може розкривати чутливу інформацію зловмисникам, що може бути використано для подальших атак. Як наприклад виведення повідомлень про помилки, що містять інформацію про внутрішню структуру програми або конфігурації.

Некоректне керування сесіями може призвести до зламу аутентифікації та несанкціонованого доступу. Прикладом можуть слугувати недостатньо випадкові або передбачувані сесійні ідентифікатори.

Безпека сторонніх бібліотек та компонентів в сенсі неконтрольованого використання застарілих чи вразливих версій бібліотек та компонентів може відкрити веб-додаток до відомих атак. Тобто використання застарілої версії фреймворку або бібліотеки з відомими вразливостями.

Sensitive Data Exposure (витік чутливої інформації) є серйозною вразливістю веб-додатків, коли конфіденційна інформація стає доступною для несанкціонованих осіб через різні канали. Ця вразливість може призвести до серйозних наслідків, таких як крадіжка особистої інформації, номерів кредитних карток, паролів або інших чутливих даних.

Основні аспекти Sensitive Data Exposure:

Некоректне зберігання паролів, тобто їх збереження у відкритому вигляді або недостатнє хешування/шифрування може призвести до витоку паролів користувачів.

Витік конфіденційних даних через неправильну конфігурацію, коли неправильна конфігурація може призвести до ненавмисного витоку конфіденційної інформації через відкриті або доступні ресурси. Наприклад неправильна конфігурація веб-сервера, що дозволяє доступ до конфіденційних файлів.

Відсутність або некоректне використання шифрування може призвести до витоку чутливої інформації під час передачі даних між клієнтом та сервером. Прикладом може бути відправка конфіденційних даних без використання протоколу HTTPS.

Незахищений сесійний механізм може призвести до витоку сесійних токенів та несанкціонованого доступу до облікових записів користувачів коли використовуються незахищені сесійні ідентифікатори або передача їх через незахищені канали.

Витік чутливої інформації через сторонні ресурси, коли запити до сторонніх ресурсів (наприклад, сторінки соціальних мереж) можуть призвести до витоку чутливої інформації, тобто коли вбудовані фрейми чи ресурси, які надаються іншими сервісами, ненавмисно передають конфіденційну інформацію.

XML External Entity (XXE) Injection - це вразливість веб-додатків, яка дозволяє зловмиснику вставляти зовнішні сутності XML у вхідні дані і отримувати чутливу інформацію або виконувати атаки на систему. Ця вразливість може виникнути, коли додаток не належним чином обробляє або парсить XML-дані зовнішніх джерел.

Основні концепції XXE Injection:

XML і Сутності. XML (Extensible Markup Language) є розширюваним мовою розмітки, яка використовується для представлення та обміну структурованої інформації. Використовує теги для визначення структури документа. Сутності XML визначають певні типи даних або текстові фрагменти, які можна включити у документ XML.

Зовнішні Сутності. У XXE Injection атаки, зловмисник може використовувати зовнішні сутності, які вказують на зовнішні ресурси, такі як файлові системи або віддалені сервери.

XXE Атака коли зловмисник вставляє в документ XML зовнішню сутність, яку спрямовує на зловживання. Наприклад, може бути введений експлойт, який витікає конфіденційну інформацію або виконує дії на сервері.

Remote Code Execution (RCE) тобто коли зловмисник може виконати віддалені команди або скрипти, що веде до виконання коду на сервері.

Security Headers Missing тобто відсутність заголовків безпеки у веб-додатку яка може створити серйозні потенційні вразливості та збільшити ризик для атак. Ці заголовки грають важливу роль у захисті від різних видів атак, і їх належне конфігурування сприяє покращенню безпеки веб-додатків. До найважливіших заголовків можна віднести ***Strict-Transport-Security (HSTS), Content Security Policy (CSP), X-Content-Type-Options, X-Frame-Options, X-XSS-Protection, Referrer-Policy, Feature-Policy.***

Відсутність або некоректне налаштування цих заголовків може підвищити ризик великої кількості атак, таких як XSS, Clickjacking, атаки зі зміненнями MIME-типів файлів та інші.

File Upload Vulnerabilities Уразливості, пов'язані з завантаженням файлів, можуть створити серйозний ризик для безпеки веб-додатків, оскільки зловмисники можуть використовувати ці механізми для виконання різних видів атак. Дозвіл на завантаження файлів може бути корисним функціоналом, але неправильна обробка може призвести до вразливостей. Основні аспекти вразливостей, пов'язаних з завантаженням файлів, включають:

Віруси та Шкідливі Файли. Зловмисники можуть завантажити віруси, троянські програми або інші шкідливі файли, які можуть шкодити серверу або іншим користувачам.

Викидання Файлів (File Traversal): зловмисники можуть намагатися використати спеціальні символи або точки для переходу до інших каталогів і отримання доступу до чутливих файлів.

Виконання Коду в Файлах Зображень: якщо система недостатньо перевіряє типи файлів, зловмисники можуть вставити код, наприклад, PHP, в файл зображення та викликати його виконання.

Денінг Атаки (Denial-of-Service): Завантаження великої кількості файлів або файлів великого розміру може призвести до перевантаження сервера та викликати атаки на відмову в обслуговуванні.

Витік Інформації: Неправильна обробка завантажених файлів може призвести до витоку конфіденційної інформації.

Маскування Файлів: зловмисники можуть спробувати маскувати виконуваний код, використовуючи обмежені фільтри на приймання файлів.

Сповільнення Атак (Slowloris): Завантаження файлів може бути використано для реалізації атак Slowloris, при яких зловмисник намагається витягти максимум ресурсів сервера.

1.5.2 Стратегії захисту та криптографічні підходи

Захист веб-застосунків є критично важливим завданням у світі, де інтернет використовується для обміну чутливою інформацією та виконання різних операцій [14]. Сучасні стратегії захисту веб-застосунків та криптографічні підходи стали ускладненими та розвинутими, оскільки кіберзлочинці постійно намагаються вдосконалювати свої методи атак.

Firewall і WAF (Web Application Firewall) є ключовими складовими систем безпеки веб-застосунків.

Firewall:

Мережевий брандмауер: Контролює та фільтрує трафік на мережевому рівні, визначаючи, які пакети дозволено або блокується на основі заданих правил. Захищає від різних мережевих атак, таких як DDoS (розподілені атаки з обмеженим доступом) та проникнення в мережу.

Брандмауер застосунків: Аналізує пакети на рівні застосунків для забезпечення дозволу чи блокування трафіку в залежності від аплікаційних правил. Захищає від атак на рівні аплікацій, таких як SQL-ін'єкції та кросс-сайтовий скриптинг.

Web Application Firewall (WAF):

Рівень аплікації: Захищає веб-застосунки на рівні аплікації, аналізуючи HTTP-запити та відповіді. Застосовується для виявлення та блокування атак, таких як SQL-ін'єкції, кросс-сайтовий скриптинг, ін'єкції коду та інші вразливості застосунків.

Основні функції WAF: Фільтрація введених даних для виявлення та блокування потенційно шкідливих введених даних, які можуть викликати атаки на застосунки. Використання регулярних виразів для пошуку та блокування певних шаблонів атак. Відстеження та аналіз HTTP-запитів та відповідей для виявлення аномалій та атак.

Попередження і реагування: Логування та моніторинг, тобто запис подій для подальшого аналізу та моніторингу безпеки. Автоматичне блокування трафіку, що відповідає зарані визначеним атакам.

Налаштування правил: Кастомізація, тобто можливість налаштування правил WAF відповідно до конкретних потреб та вимог застосунку.

Важливі аспекти безпеки Firewall і WAF це постійне оновлення правил WAF для виявлення нових типів атак та вразливостей, інтеграція з SIEM системами для ефективного моніторингу та аналізу журналів безпеки, регулярне проведення тестів на проникнення для виявлення можливих слабкі місця в системі захисту.

Firewall і WAF в сукупності з іншими стратегіями безпеки є важливими елементами у захисті веб-застосунків від різноманітних кіберзагроз.

SSL (Secure Sockets Layer) та його сучасний варіант **TLS (Transport Layer Security)** є протоколами шифрування, призначеними для захисту конфіденційності та цілісності передачі даних через мережу Інтернет. SSL був введений компанією Netscape у 1995 році, але через численні вразливості та атаки був замінений TLS у 1999 році. TLS продовжив розвиток та поліпшення безпеки. Метою створення було забезпечення захищеної передачі даних через відкриті мережі, такими як Інтернет.

Робочий принцип: SSL/TLS використовує симетричне та асиметричне шифрування для забезпечення безпеки транзакцій. Симетричне шифрування використовується для шифрування самої інформації, тоді як асиметричне шифрування використовується для обміну ключами та перевірки цілісності.

Процес встановлення з'єднання SSL/TLS:

Привітання (Handshake): Клієнт надсилає серверу запит на підключення та вказує підтримувані криптографічні алгоритми. Сервер надсилає свій сертифікат клієнту для аутентифікації та вибору спільного ключа.

Обмін ключами (Key Exchange): Використовуючи асиметричне шифрування, клієнт і сервер обмінюються інформацією для встановлення спільного секретного ключа.

Шифрування симетричне: Після встановлення ключа використовується симетричне шифрування для шифрування та дешифрування даних, що передаються між клієнтом і сервером.

Завершення (Finish): Клієнт і сервер підтверджують, що шифрування встановлено і транзакції можуть початися.

Компоненти SSL/TLS є:

Сертифікат: Електронний документ, що підтверджує ідентичність власника та містить відкритий ключ для шифрування.

Протокол TLS: Визначає правила для ефективного та безпечного обміну даними.

Сесійні ключі: Ключі, що використовуються для симетричного шифрування в конкретній сесії.

Види сертифікатів SSL/TLS:

Одиночний домен: Покриває лише один домен або піддомен.

Сертифікат Wildcard: Покриває всі піддомени для певного домену.

Розширений валідований (EV) сертифікат: Найвищий рівень валідації, забезпечує користувачам браузера додаткову інформацію про компанію.

До потенційних загроз використання можна віднести вразливості сертифікатів, атаки на протокол, атаки типу "man-in-the-middle". Важливо використовувати найновіші версії TLS та оновлювати сертифікати для захисту від вразливостей.

Кріптографічні алгоритми: Існує постійний розвиток та апгрейд криптографічних алгоритмів для забезпечення стійкості до сучасних кіберзагроз, будуть оглядово розглянуті далі.

SSL/TLS шифрування є важливою складовою кібербезпеки, а його правильне використання дозволяє забезпечити конфіденційність та цілісність інформації, яка пересилається через мережу.

Multi-Factor Authentication (MFA), або міцна аутентифікація є методом забезпечення безпеки, що використовує не тільки щось, що користувач знає (наприклад, пароль) або щось, що користувач має (наприклад, фізичний ключ), але і щось, чим користувач є (наприклад, біометричні дані). Цей підхід зменшує ризик несанкціонованого доступу, оскільки навіть якщо один фактор порушено, інші залишаються для забезпечення безпеки.

Основні аспекти MFA:

Фактори аутентифікації:

Через щось, що ти знаєш (Knowledge): Паролі, PIN-коди.

Через щось, що ти маєш (Possession): Фізичні пристрої, такі як USB-ключі, смарт-карти або мобільні токени.

Через щось, чим ти є (Inherence): Біометричні дані, такі як відбиток пальця, розпізнавання обличчя або сканер очей.

Факторінг та Комбінування: Міцна аутентифікація може використовувати один або комбінацію факторів для забезпечення високого рівня безпеки. Наприклад, комбінування паролю та коду з SMS або використання біометричних даних разом із фізичним токеном.

Процес аутентифікації: Користувач вводить основний пароль або інші ідентифікуючі дані. Система вимагає додатковий фактор аутентифікації (наприклад, код з SMS, відбиток пальця або фізичний токен). Комбінація введених даних та додаткового фактору перевіряється для підтвердження ідентифікації.

Різні способи реалізації:

SMS-коди та Email-підтвердження: Висилаються одноразові коди для підтвердження через SMS або електронну пошту.

Токени: Фізичні пристрої або мобільні додатки, які генерують одноразові паролі.

Біометрія: Розпізнавання обличчя, відбиток пальця, сканер очей.

Апаратні ключі: USB-ключі або смарт-карти, які використовуються для аутентифікації.

Переваги Міцної аутентифікації полягають в підвищення безпеки, зменшуючи ризики в рази, оскільки потрібно подолати кілька різних бар'єрів для отримання доступу захист від втрати паролів, тому що навіть якщо пароль стає відомий, без іншого фактору важко отримати доступ, зменшення ризику фішингу, бо навіть якщо зловмисник отримає пароль, йому все одно доведеться подолати інший бар'єр для успішного вторгнення.

До викликів та обмежень можна віднести вартість та складність впровадження, бо реалізація та управління MFA може вимагати значних витрат і завдати додаткові труднощі користувачам, Можливість втрати додаткових факторів тобто загуба або пошкодження фізичних токенів може стати проблемою, та деякі користувачі можуть вважати MFA не зручною через додаткові кроки.

Міцна аутентифікація відіграє ключову роль в забезпеченні безпеки в онлайн-середовищі та дозволяє врегулювати ризики, пов'язані з втратою або компрометацією основних ідентифікаторів.

"Безпека сесій" — це область кібербезпеки, що вивчає методи захисту інформаційних сесій між клієнтом і сервером в ході взаємодії веб-застосунків чи інших систем. Сесії важливі для забезпечення продовження користувацької ідентифікації та стану взаємодії під час перегляду веб-сайтів чи використання інших веб-застосунків. Забезпечення безпеки сесій включає заходи для захисту від атак, таких як перехоплення сесій, фіктивний запит на входження в систему, та інші атаки, спрямовані на винищення конфіденційної інформації чи завдання шкоди.

Основні аспекти безпеки сесій:

Унікальні ідентифікатори сесій: Кожна сесія повинна мати унікальний ідентифікатор, який важко передбачити або вгадати. Це може бути досягнуто за допомогою випадкових чисел чи складних алгоритмів генерації.

Захищені механізми передачі: Всі дані, що передаються між клієнтом і сервером під час сесії, повинні бути шифрованими, щоб унеможливити їх перехоплення та читання зловмисниками.

Шифрування cookie-файлів: Інформація, збережена у cookie-файлах, повинна бути шифрованою, щоб запобігти несанкціонованому доступу.

Термін дії сесій: Важливо встановлювати адекватний термін дії сесій, щоб уникнути можливості довготривалого залишення сесії відкритою.

Відслідковування активності: Моніторинг та логування активності в сесії для виявлення надзвичайних подій чи спроб несанкціонованого доступу.

Захист від атак на фіктивне входження (Session Fixation): Заходи для уникнення атак, коли зловмисник намагається зафіксувати (фіксувати) сесійний ідентифікатор і використовувати його для власних цілей.

Автоматичний вихід після неактивності: Автоматичне завершення сесії після тривалого періоду неактивності для захисту від можливих атак.

Рекомендаціями для безпеки сесій можуть бути наступними - встановлення безпечного з'єднання через протокол HTTPS для шифрування трафіку між клієнтом і сервером, застосування механізмів шифрування для сесійних ідентифікаторів, щоб унеможливити їх перехоплення, періодичне оновлення сесійних ідентифікаторів для ускладнення атак, проведення аудиту безпеки сесій для виявлення можливих вразливостей та вдосконалення заходів захисту, розробка застосунків із врахуванням безпеки сесій на рівні програмного коду, повідомлення користувачів про важливість відключення сесії при завершенні роботи на публічних або спільних комп'ютерах.

Безпека сесій є ключовою складовою веб-застосунків та інших інтерактивних систем, і правильні заходи захисту є важливими для забезпечення безпеки веб-застосунків.

Обробка та **валідація** введених даних є критичним етапом в розробці програмного забезпечення для забезпечення коректності, безпеки та ефективності системи. Неправильна або некоректна обробка даних може призвести до

вразливостей, втрати інформації, атак з боку користувачів та інших проблем безпеки.

Обробка введених даних: Дані з різних джерел (користувачі, API, бази даних) збираються для подальшої обробки, перевіряються, чи введені дані відповідають очікуваному формату (наприклад, електронна адреса, номер телефону), подальша чистка та нормалізація, вилучення зайвих символів, пробілів, конвертація до стандартних форматів для уніфікації даних та захист від ін’єкцій, використовуючи параметризовані запити та механізми боротьби з ін’єкціями (SQL, XSS, etc.) для запобігання атакам.

Валідація введених даних: починається з перевірка на обов'язковість, визначення, чи всі обов'язкові поля заповнені, перевірка типів даних, валідація, чи тип введених даних (число, текст, дата) відповідає очікуваному, обмеження довжини текстових полів та перевірка розмірів файлів, щоб уникнути переповнення буфера, валідація за допомогою регулярних виразів, щоб переконатися, що дані відповідають певному формату (наприклад, електронна адреса, номер телефону), перевірка, чи введені дані (наприклад, ім'я користувача, електронна адреса) унікальні для системи.

Валідація на стороні клієнта та сервера: валідація на стороні клієнта для швидшого надання повідомлень про помилки та валідація на стороні сервера для запобігання обхіду та атак на безпеку, визначення, чи дані мають сенс та логічно взаємодіють (наприклад, дата народження не може бути в майбутньому), використання механізмів для запобігання впровадження в код атак на основі скриптів та міжсайтових атак на фальшивий запит, коректна обробка помилок валідації та надання інформативних повідомлень користувачам.

Важливими практиками є: Принцип найменшого доступу - надання лише необхідного рівня доступу до обробки та валідації даних, регулярне оновлення правил валідації - оновлення правил та механізмів валідації відповідно до змін в системі та змін в безпеці, Автоматизовані тести - використання автоматизованих тестів для валідації коректності та безпеки обробки даних.

Обробка та валідація даних є необхідними етапами в розробці безпечного та функціонального програмного забезпечення. Вони сприяють запобіганню численним видам атак та забезпечують коректну обробку введених даних.

Криптографія - це наука про забезпечення конфіденційності, цілісності та автентифікації інформації. Криптографічні підходи використовуються для захисту від несанкціонованого доступу, атак та недостойних методів забезпечення безпеки [15]. Можна виділити деякі основні криптографічні підходи:

Симетричне шифрування: Використовує один ключ для як шифрування, так і розшифрування. Алгоритми: DES, AES (Advanced Encryption Standard), 3DES, Blowfish. До переваг можна віднести швидкість шифрування та розшифрування, ефективність для великих обсягів даних. Недоліками можна вважати проблеми з безпекою ключа обмеженого обміну ключів, не забезпечує цілісність та автентифікацію.

Асиметричне шифрування: Використовує два ключі: публічний (для шифрування) та приватний (для розшифрування). Алгоритми: RSA, ECC (Elliptic Curve Cryptography). Перевагами асиметричного шифрування є безпека обміну ключами через публічний ключ, забезпечення цілісності та автентифікації. До недоліків можна віднести більшу повільність, порівняно з симетричним шифруванням та великі обчислювальні витрати.

Хеш-функції: Призначені для створення фіксованих розмірів "хеш-значень" з будь-яких даних. Алгоритми: SHA-256, SHA-3, MD5 (не рекомендується через вразливості). Використовується для зберігання паролів у безпечному вигляді та перевірка цілісності даних. Недоліком слід вважати вразливість до атак колізій (два різних набори даних мають той самий хеш).

Цифрові підписи: використовуються для підтвердження автентичності та цілісності документа чи повідомлення для чого застосовують асиметричне шифрування. Використовуються для електронних підписів документів та підтвердження походження даних. Недоліком є вимагання використання приватного ключа для підпису, що може бути проблематичним.

SSL/TLS протоколи: використовуються для захищеного обміну даними між клієнтом і сервером в Інтернеті. Докладно описані вище.

Квантова криптографія: Використовує властивості квантової механіки для забезпечення безпеки обміну ключами, застосовується для захисту від квантового обчислення. Потенційною перевагою може бути захист від атак, що базуються на алгоритмах квантового обчислення. Але має великі виклики технічного характеру та складність впровадження.

Криптографічні підходи представляють собою важливий інструмент для забезпечення безпеки в інформаційних технологіях. Вибір конкретного методу часто залежить від вимог конкретного використання та рівня безпеки, який потрібно забезпечити.

1.6 Висновки з розділу 1

В даному розділі були розглянуті наступні питання:

1. «Цибулевої архітектури», її порівняння з традиційними моделями та роль інверсії контролю в ній;
2. Зроблено огляд сучасних методів комунікації у веб-застосунках, зроблено акцент на REST API та WebSocets;
3. Розглянуто питання рефлексії та її вплив на веб-додатки;
4. Розглянути основи мікросервісної архітектури, її особливості та виклики в провадженні;
5. Зроблено огляд загальних вразливостей веб-додатків та розглянуто основні стратегії захисту включно з криптографічними підходами.

РОЗДІЛ 2 АРХІТЕКТУРНІ ТА ТЕХНОЛОГІЧНІ ОСОБЛИВОСТІ БІБЛІОТЕКИ КОМПОНЕНТІВ

2.1 Структурні аспекти бібліотеки компонентів

Під "бібліотекою елементів" розуміють бібліотеки програмного забезпечення, які містять готові елементи інтерфейсу користувача (UI) для використання у розробці програм [16]. Такі елементи можуть включати кнопки, тексти, таблиці, поля введення, вікна, та інші компоненти, які полегшують створення інтерфейсів.

Структурні аспекти бібліотек елементів можуть варіюватися залежно від конкретної бібліотеки та мови програмування, в якій вони реалізовані. Однак, основні елементи та концепції, які вони можуть включати, включають наступне:

Компоненти інтерфейсу:

Кнопки як елементи для виклику певної дії.

Текстові поля, що використовуються для введення тексту або відображення інформації.

Вікна та контейнери, що служать для розташування та організації інших елементів.

Таблиці та списки, що використовуються для представлення даних у вигляді таблиць або списків.

Події та обробники подій:

Обробники подій, це якийсь код, який викликається при виникненні певної події (наприклад, натискання кнопки).

Події, або наслідки, які виникають при певних діях користувача або системи.

Стилі та оформлення:

CSS (Cascading Style Sheets), який використовується для стилізації елементів інтерфейсу.

Теми та шаблони, що дозволяють швидко змінювати зовнішній вигляд за допомогою заздалегідь визначених стилів.

Мови програмування та платформи:

JavaFX, Java Swing (для Java) це набір бібліотек для створення графічних інтерфейсів в Java.

UIKit, AppKit (для Swift та Objective-C), який використовується для розробки програм під iOS та macOS.

Крос-платформеність:

Наприклад - Qt: Бібліотека, яка дозволяє розробляти крос-платформені програми.

Розташування та розміщення елементів:

Layout Managers, які визначають, як елементи розташовані та взаємодіють один з одним.

Документація та спільнота:

API-документація: Опис функцій та класів, що надає бібліотека.

Форуми та ресурси спільноти, які допомагають розробникам обмінюватися досвідом та отримувати підтримку.

Ці структурні аспекти використовуються зазвичай для створення зручних та естетичних інтерфейсів користувача у своїх програмах.

2.1.1 Модульність та повторне використання компонентів

Модульність та повторне використання компонентів є важливими концепціями в розробці програмного забезпечення, зокрема в контексті бібліотек компонентів [17]. Тому треба зупинитись на їх огляді трошки докладніше:

Модульність:

Модульність - це властивість системи розробки програмного забезпечення, яка дозволяє створювати програми з окремих, самостійних блоків (модулів). Модулі можуть бути розроблені, тестовані, та підтримувати незалежно один від одного.

Бібліотеки компонентів, такі як UI-бібліотеки, зазвичай будуються з великої кількості модульних компонентів. Кожен компонент може виконувати конкретну функцію та бути незалежним від інших.

До переваг можна віднести підтримка великих проектів, дозволяє легше управляти та підтримувати великі проекти шляхом визначення чітких меж між компонентами. Легкість в розробці та тестуванні, модульні компоненти можна розробляти та тестувати окремо, що полегшує виявлення та виправлення помилок.

Повторне використання компонентів:

Повторне використання - це принцип, за яким компоненти програмного забезпечення можуть бути використані у різних проектах або частинах одного проекту.

Бібліотеки компонентів надають готові до використання елементи, які можна легко інтегрувати в різні проекти. Наприклад, UI-компоненти бібліотеки можуть бути використані для швидкого створення інтерфейсу в різних додатках.

Перевагами повторного використання є зменшення затрат на розробку, тобто є можливість використання готових компонентів дозволяє заощаджувати час та зусилля при розробці нових проектів. Стандартизація та однорідність, використання одних і тих самих компонентів може забезпечити стандартизацію інтерфейсу та функціональності.

Взаємодія модульності та повторного використання:

Стандартизування, коли визначення чітких інтерфейсів для модулів та використання стандартів дозволяє підтримувати високий рівень модульності і забезпечує легке повторне використання.

Модулі і компоненти можуть використовуватись за межами одного проекту, тому документація модулів та компонентів грає важливу роль у сприянні їхньому коректному використанню в інших проектах.

Взаємодія модульності та повторного використання допомагає підтримувати гнучкі та ефективні проекти, зменшуючи зусилля, потрібні для розробки та підтримки програмного забезпечення.

2.1.2 Принципи інкапсуляції та ізоляції в бібліотеці

Інкапсуляція та ізоляція - це ключові принципи програмування, які допомагають покращити модульність, утриманість та безпеку програмного забезпечення, включаючи бібліотеки компонентів [17].

Інкапсуляція:

Основний принцип інкапсуляції полягає в обгортанні даних та функцій, які працюють з цими даними, у єдиний компонент, який називається класом або модулем.

Принципи реалізації в бібліотеках компонентів:

Бібліотеки надають класи та інтерфейси, які визначають структури даних та функції для роботи з ними.

Деякі частини бібліотеки можуть бути приховані від користувача за допомогою модифікаторів доступу, таких як private чи protected в об'єктно-орієнтованому програмуванні.

Геттери та сеттери: Забезпечення контролю над доступом до даних шляхом використання методів для отримання (геттери) та встановлення (сеттери) значень.

Рисунок 4 — *Геттери та Сеттери в інкапсуляції*

До переваг можна віднести зменшення залежностей між різними частинами системи, захист даних від прямого доступу та зміни з боку інших компонентів, забезпечення чіткої інтерфейсної частини, яка залишається стабільною при зміні внутрішньої реалізації.

Ізоляція:

Основний принцип ізоляція полягає в обмеженні доступу одного компонента до внутрішніх деталей іншого. Це допомагає забезпечити незалежність компонентів та уникнути небажаних взаємодій.

Принципи реалізації в бібліотеках компонентів:

Ізоляція функціональності: Кожен компонент повинен виконувати конкретну функцію і бути відокремленим від інших компонентів.

Інтерфейси: Визначення чітких інтерфейсів для взаємодії між компонентами.

Модульність: Розділення функціональності на модулі чи пакети, щоб забезпечити легку заміну та оновлення.

Обмеження доступу: Використання механізмів обмеження доступу для забезпечення, щоб тільки необхідна інформація була доступна іншим компонентам.

Перевагами ізоляції є можливість забезпечення стабільності та надійності системи, зменшення можливості виникнення помилок через взаємодію непов'язаних компонентів, можливість заміни або оновлення компонентів без впливу на інші частини системи.

Таким чином, інкапсуляція та ізоляція допомагають створювати гнучкі, масштабовані та підтримувані системи, особливо коли мова йде про розробку бібліотек компонентів.

2.2 Технологічні підходи в бібліотеці компонентів

Створення бібліотек компонентів є важливою частиною розробки програмного забезпечення, оскільки вона спрощує процес створення програм та допомагає у відновленні коду. Існує кілька технологічних підходів до створення бібліотек компонентів [19]:

Atomic Design:

Цей підхід базується на ідеї поділу інтерфейсу на найменші можливі частини, які називають атомами. Атоми об'єднуються в молекули, які в свою чергу створюють організми. Це дозволяє створювати компоненти на різних рівнях абстракції.

Рисунок 5 — *Atomic Design*

Component-Based Development (CBD):

Цей підхід передбачає створення програмного забезпечення шляхом з'єднання вже наявних компонентів, які можуть бути використані в різних контекстах. Це сприяє використанню готових рішень і рефакторингу коду.

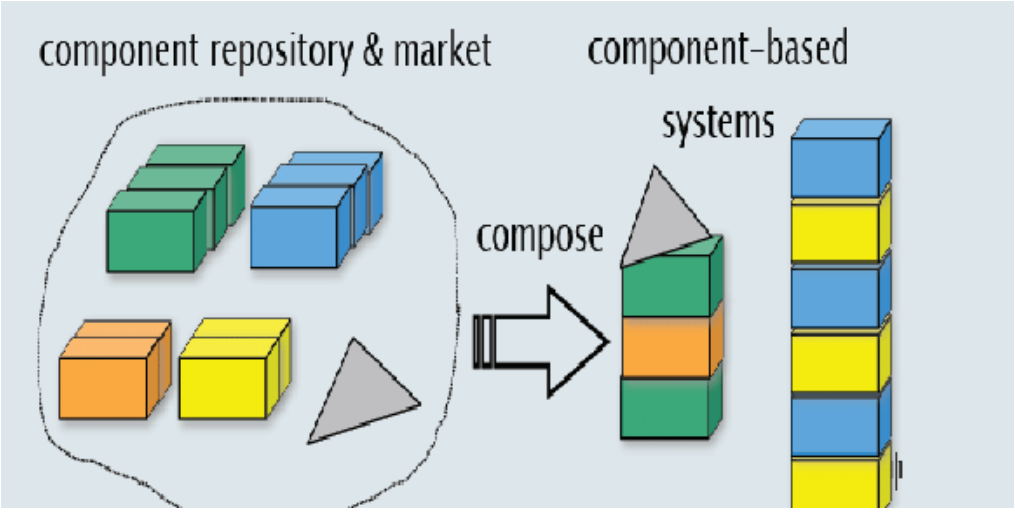

Рисунок 7 — *Концепція CBD*

Web Components:

Цей підхід є набором технологій веб-стандартів, що дозволяє створювати власні, повторно використовувані елементи (компоненти) і вбудовувати їх в інші веб-додатки. Зокрема, вони включають у себе Shadow DOM, HTML Templates, Custom Elements та HTML Imports.

UI Frameworks та бібліотеки:

Використання популярних фреймворків і бібліотек, таких як React, Angular, Vue.js. Ці інструменти дозволяють створювати компоненти, які можна використовувати в різних частинах програми.

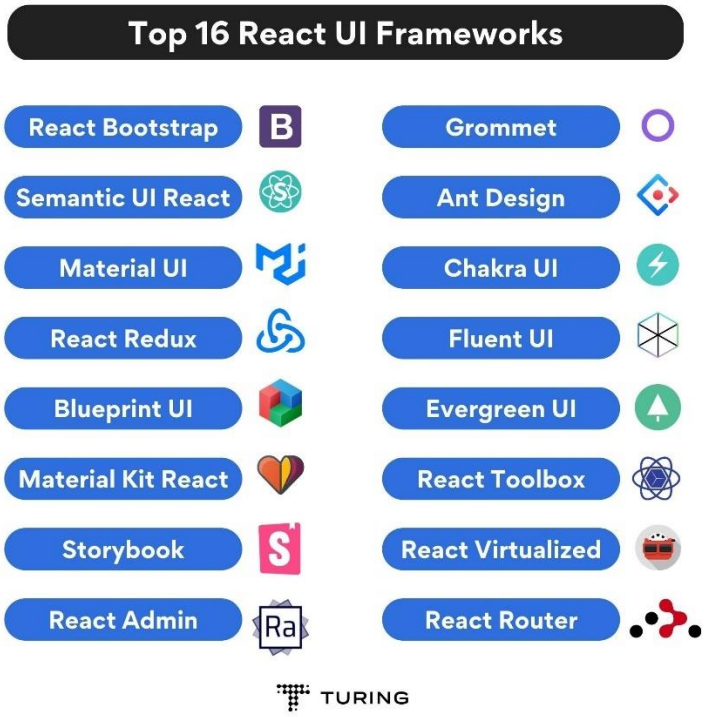

Рисунок 8 — *UI Frameworks*

Storybook:

Storybook - це інструмент для розробки UI компонентів в ізольованому середовищі. Він дозволяє вам створювати, тестувати та документувати компоненти незалежно від основного додатку.

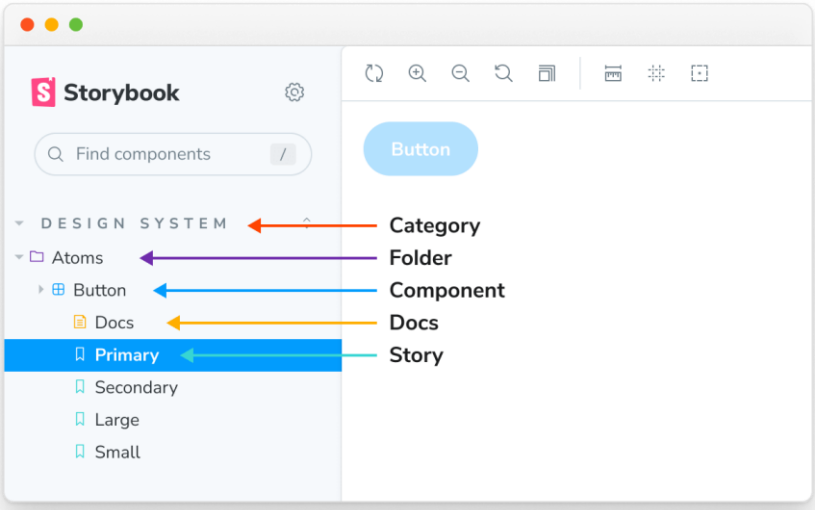

Рисунок 9 — *Іменування компонентів та ієрархія*

CSS-in-JS:

Використання бібліотек, таких як Styled Components (React), що дозволяють включати CSS безпосередньо в компоненти. Це спрощує управління стилями та зберігання їх разом із компонентами.

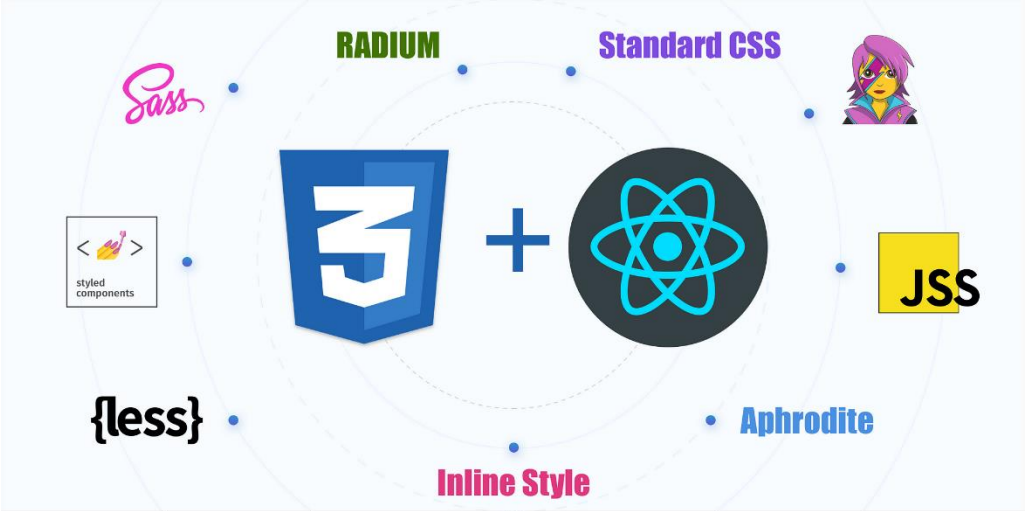

Рисунок 10 — *9 способів впровадження CSS у React JS, один з яких SC*

Ці підходи можна комбінувати в залежності від потреб проекту та вибору розробника. Створення бібліотек компонентів допомагає забезпечити консистентність та повторне використання коду, що є важливими принципами в розробці програмного забезпечення.

2.2.1 Використання шаблонів проектування в компонентах

Шаблони проектування є загальними рекомендаціями або визначеннями структур та взаємодій, які можна використовувати для розв'язання типових проблем в програмуванні. Вони можуть бути дуже корисними при створенні компонентів, оскільки допомагають впорядковувати код, полегшують його розуміння та підтримку. Як приклад декілька шаблонів проектування, які можна використовувати в компонентах:

Шаблон Синглтон (Singleton) - це один з структурних шаблонів проектування, який забезпечує, що клас має лише один екземпляр і надає глобальну

точку доступу до цього екземпляра. Це корисний шаблон, коли існує потреба в одному об'єкті, який керує доступом до ресурсів, таких як база даних, файлова система або будь-яка інша глобальна інстанція.

Основні компоненти Синглтона це Приватний конструктор (Private Constructor), який заважає створенню екземплярів класу ззовні, приватне статичне поле (Private Static Field), яке буде містити єдиний екземпляр класу та Публічний метод доступу (Public Access Method), який надає доступ до єдиного екземпляра класу. Якщо екземпляр ще не створений, то цей метод створює його.

Механізм роботи Синглтона полягає в перевірці наявності екземпляру, коли при кожному виклику методу getInstance, перевіряється, чи вже створений екземпляр класу; створення екземпляру при необхідності, тобто коли екземпляр ще не існує, викликається приватний конструктор для створення нового екземпляра класу; поверненні єдиного екземпляру, коли екземпляр вже існує або був щойно створений, повертається цей єдиний екземпляр.

Шаблон Синглтон використовується там, де необхідно гарантувати, що клас має лише один екземпляр, і цей екземпляр легко доступний для всіх інших частин програми. Такі ситуації можуть включати керування доступом до ресурсів, таких як база даних, логгери, конфігураційні файли тощо.

Фабричний метод (Factory Method), який дозволяє класам делегувати вибір типу об'єкта класум, які вони створюють. Використовується для створення об'єктів на основі певного інтерфейсу без визначення конкретного класу.

Цей метод є одним із п'яти базових шаблонів проектування, що належать до категорії шаблонів "Створення". Шаблон надає інтерфейс для створення об'єктів в суперкласі, але залишає вибір конкретного класу об'єкта на пізніший етап його використання. Тобто, він дозволяє підкласам змінювати тип створеного об'єкта.

Основні учасники шаблону це Creator (Творець), тобто клас або інтерфейс, що оголошує фабричний метод, який повинен бути реалізований підкласами. Він визначає загальний інтерфейс для створення об'єктів, але сам не конкретизує конкретний тип об'єкта; ConcreteCreator (Конкретний творець)це вже підклас, який реалізує фабричний метод і повертає конкретний тип об'єкта; далі йде Product (Продукт) що є класом або інтерфейсом, що визначає тип створюваного об'єкта; і нарешті ConcreteProduct (Конкретний продукт) або клас, який реалізує інтерфейс Product і представляє конкретний тип об'єкта, який створюється фабричним методом.

Основні етапи використання шаблону Фабричний метод відповідають поведінці основних учасників: створення інтерфейсу Product для визначення інтерфейсу або абстрактного класу, який представляє продукт, який буде створювати фабричний метод; створення ConcreteProduct або реалізація класів, які відповідають інтерфейсу Product (кожен клас представляє конкретний тип продукту); далі створення Creator для визначення інтерфейсу або абстрактного класу, який має фабричний метод для створення об'єктів типу Product; і на сам кінець створення ConcreteCreator, реалізація підкласу, який наслідує від Creator і реалізує фабричний метод. В цьому методі визначається конкретний тип об'єкта, який буде створений.

Клієнтський код викликає фабричний метод для отримання об'єкта типу Product, не вказуючи конкретний клас продукту. Таким чином, конкретний тип об'єкта вибирається на етапі виконання.

Декоратор (Decorator) є структурним шаблоном проектування, який дозволяє динамічно додавати нові функціональності або змінювати поведінку об'єктів, не змінюючи їх основного коду. Цей шаблон базується на принципі відкритості/закритості, оскільки він надає можливість розширювати функціональність без внесення змін у сам клас об'єкта.

Структура Декоратора виглядає наступним чином: компонент (Component) як базовий або основний клас, який визначає інтерфейс для об'єктів, які можуть бути декоровані; конкретний компонент (Concrete

Component), це клас, який реалізує інтерфейс компонента та представляє основний об'єкт, до якого можна додавати функціональність; декоратор (Decorator) абстрактний клас або інтерфейс, який також реалізує інтерфейс компонента, але також має посилання на об'єкт класу компонента та конкретний декоратор (Concrete Decorator) тобто клас, який розширює функціональність базового об'єкта (Concrete Component) та імплементує декоратор, додаючи свою власну функціональність.

Стратегія (Strategy) відноситься до родини шаблонів "Поведінковий патерн" та дозволяє визначати сімейство алгоритмів, інкапсулювати кожен з них та зробити їх взаємозамінними. Використовується для визначення, інкапсуляції та обміну алгоритмами в об'єкті. Основна ідея полягає в тому, щоб визначити ряд стратегій (алгоритмів), упакувати їх та зробити заміну однієї стратегії іншою прозорою.

Ключові учасники шаблону це: контекст (Context), або об'єкт, який має посилання на стратегію та може використовувати її. Контекст може бути налаштований для використання конкретної стратегії; стратегія (Strategy) тобто інтерфейс або абстрактний клас, який визначає метод, що представляє алгоритм (кожна конкретна стратегія реалізує цей інтерфейс/клас) та конкретні стратегії (Concrete Strategies) або класи, які реалізують конкретні варіації алгоритму, оголошеного у стратегії.

Спостерігач (Observer) це поведінковий шаблон проектування, який визначає залежність один до багатьох між об'єктами так, що коли один об'єкт змінює свій стан, всі йому залежні отримують сповіщення і оновлюються автоматично. Цей шаблон дозволяє створювати слабоспряжені системи, де зміна в одному об'єкті автоматично призводить до змін в інших об'єктах.

Структура Спостерігача наступна: суб'єкт (Subject) що визначає інтерфейс для додавання, видалення та сповіщення спостерігачів (утримує список спостерігачів); конкретний суб'єкт (ConcreteSubject) який реалізує інтерфейс суб'єкта та зберігає свій стан (відправляє сповіщення своїм спостерігачам про

будь-які зміни); спостерігач (Observer), який визначає інтерфейс для оновлення, який використовує суб'єкт для повідомлення про зміни; та конкретний спостерігач (ConcreteObserver) що зберігає посилання на конкретний суб'єкт (реалізує інтерфейс оновлення для збереження відповідності зі суб'єктом).

Принцип роботи Спостерігача виглядає наступним чином - спостерігачі реєструються у суб'єкта для отримання сповіщень про зміни, коли стан суб'єкта змінюється, він повідомляє всіх своїх зареєстрованих спостерігачів. Тоді кожен спостерігач автоматично оновлює свій стан відповідно до змін в суб'єкті.

Компоновщик (Composite) є структурним шаблоном проектування, який дозволяє клієнтам обробляти окремі об'єкти та їх складові однаковим чином. Цей шаблон дозволяє створювати ієрархії об'єктів, де як окремі об'єкти, так і їх складові можуть бути оброблені єдинообразно.

Основні елементи шаблону Компоновщик це компонент, який визначає інтерфейс для всіх конкретних класів, які складають ієрархію. Забезпечує уніфікований інтерфейс для всіх об'єктів (як окремих, так і складових); лист, або конкретна реалізація компонента, який не має дочірніх елементів. Виконує конкретні дії, які визначені у батьківському інтерфейсі; та контейнер, як конкретна реалізація компонента, яка має дочірні елементи. Він координує дії своїх дочірніх об'єктів та передає їм запити через єдинообразний інтерфейс.

Цей шаблон має деякі особливості, такі як уніфікований інтерфейс (всі об'єкти у ієрархії мають спільний інтерфейс, що дозволяє їм бути обробленими однаково), рекурсивна структура (Ієрархія може бути складною та рекурсивною, що дозволяє вкладати контейнери в інші контейнери, створюючи деревоподібну структуру), додавання та видалення елементів (компоновщик дозволяє динамічно додавати та видаляти елементи в ієрархії без зміни інтерфейсу клієнта) та спільне використання коду (код, що опрацьовує об'єкти, може бути спільним для всіх елементів ієрархії, що спрощує код та підтримку).

Шаблон компоновщика використовується в ситуаціях, де є необхідність у вираженні складних структур, які складаються з ієрархії об'єктів, та в тих

випадках, коли клієнту потрібно обробляти окремі об'єкти та їх групи однаковим чином.

Шаблонний метод (Template Method) - це паттерн проектування, який визначає загальну структуру алгоритму у базовому класі та визначає деякі кроки алгоритму в похідних класах. Він входить до категорії паттернів "Поведінка". Головна ідея полягає в тому, щоб визначити скелет алгоритму, але залишити деякі кроки реалізації для підкласів.

Ключові елементи шаблону це: абстрактний клас, тобто базовий клас, який містить скелет алгоритму. Він може містити як абстрактні, так і конкретні методи. Абстрактні методи визначаються як частини алгоритму, які повинні бути реалізовані в похідних класах; конкретні класи, або похідні класи що визначають конкретні реалізації абстрактних методів, які є частиною алгоритму. Ці класи можуть також перевизначати абстрактні методи, якщо потрібно.

Принцип роботи шаблону полягає в тому, що базовий клас визначає загальну структуру алгоритму та включає в себе кроки, які мають бути виконані в певному порядку. Потім базовий клас визначає абстрактні методи, які будуть виконуватися у підкласах, але конкретна реалізація цих методів залишається покладеною на похідні класи. Далі **п**охідні класи реалізують абстрактні методи, визначені у базовому класі, надаючи конкретну реалізацію кожного з кроків алгоритму. І, нарешті, базовий клас викликає абстрактні методи у визначених місцях алгоритму, запускаючи тим самим виконання конкретної логіки у підкласах.

Шаблонний метод є корисним там, де існує спільна структура алгоритму, але різні частини цього алгоритму можуть бути реалізовані різними способами.

Ці шаблони проектування можна використовувати як основу для створення компонентів з більшою структурованістю та підтримкою змін. Вони дозволяють розбити систему на менші, легко зрозумілі частини та дозволяють більш гнучку розробку та зміну.

2.2.2 Застосування принципів SOLID та DRY у розробці

Принципи SOLID та DRY є ключовими концепціями в об'єктно-орієнтованому програмуванні, і вони можуть бути успішно використані при розробці бібліотек компонентів для забезпечення ефективності, розширюваності та легкості обслуговування [20]. Дуже коротко опишемо, як ці принципи можуть бути застосовані:

SOLID - це акронім, який представляє собою п'ять основних принципів об'єктно-орієнтованого програмування та дизайну програмного забезпечення. Ці принципи були сформульовані Робертом С. Мартіном та іншими експертами, і вони спрямовані на створення гнучких, підтримуваних та легко розширюваних систем.

Рисунок 11 — *Принцип SOLID*

Назви та короткий опис кожного принципу:

- Принцип одного обов'язку (Single Responsibility Principle - SRP):

  Кожен компонент бібліотеки повинен мати лише один конкретний обов'язок. Наприклад, якщо це компонент для роботи з базою даних, він повинен концентруватися лише на цьому завданні.

- Принцип відкритості/закритості (Open/Closed Principle - OCP):

Компоненти повинні бути відкритими для розширення та закритими для модифікації. Нові функціональності повинні додаватися без внесення змін до існуючого коду.

- Принцип підстановки Барбари Лісков (Liskov Substitution Principle - LSP):

  Класи та інтерфейси, які визначають компоненти, повинні бути взаємозамінними. Користувачі бібліотеки повинні мати можливість використовувати різні реалізації компонентів без втрати функціональності.

- Принцип інтерфейсів (Interface Segregation Principle - ISP):

  Клієнти не повинні залежати від інтерфейсів, які вони не використовують. Це означає, що клієнти повинні залежати від тих інтерфейсів, які вони дійсно використовують. Це дозволяє створювати компоненти, які використовують лише необхідний функціонал.

- Принцип інверсії залежностей (Dependency Inversion Principle - DIP):

  Залежності між компонентами бібліотеки повинні використовувати абстракції та інтерфейси. Високорівневі модулі повинні залежати від абстракцій, а не від деталей низькорівневих модулів.Використання цих принципів допомагає створювати гнучкі, розширювані та підтримувані системи, а також полегшує роботу з кодом у великих проектах.

DRY - це принцип розробки програмного забезпечення, направлений на зниження повторення інформації різного роду, особливо в системах з множинністю слоїв абстрагування.

Рисунок 11 — *Принцип DRY*

Його вимоги та рекомендації:

- Уникання дублювання коду:

  Потрібно уникати копіювання та вставляння коду в різних частинах бібліотеки. Необхідно використовувати функції, класи або інші конструкції для організації та узагальнення коду.

- Централізація логіки:

  Логіка та функціональність, які можуть бути загальними для декількох компонентів, повинні бути централізовані. Це дозволяє змінювати та оновлювати логіку в єдиному місці.

- Використання абстракцій:

  Розробник має створювати абстракції, які дозволяють узагальнювати та використовувати код в різних контекстах. Це може бути реалізовано через використання інтерфейсів, абстрактних класів тощо.

- Генеріка та параметризація.

  Необхідно використовувати генеріки та можливості параметризації для створення універсальних компонентів, які можуть працювати з різними типами даних чи функціональністю.

- Тестування та валідація.

  Потрібно створюйти тести та валідації для уникнення невірної або непередбачуваної роботи коду. Це допомагає підтримувати якість коду та уникнути дублювання помилок.

Застосування цих принципів у розробці бібліотек компонентів допомагає створювати гнучкі, надійні та легко обслуговувані рішення, які можуть бути використані в різних проектах.

2.3 Фреймворки та інструменти для розробки бібліотеки

Станом на кінець 2023 в порівнянні з кінцем 2022 року, відповідно до деяких оглядів, можна зробити висновок, що популярність фреймворків та інструментів змінюється відповідно до тенденцій ринку та вибору розробників.

2.3.1 Аналіз популярних фреймворків для розробки веб-компонентів

Короткий аналіз трьох найбільш популярних фреймворків станом на кінець 2023.

React - це відкритий JavaScript фреймворк для розробки інтерфейсів користувача, розроблений Facebook. Він зосереджується на створенні ефективних і декларативних користувацьких інтерфейсів [22].

React використовує JSX (JavaScript XML), розширення JavaScript, яке надає синтаксис схожий на XML/HTML для опису структури UI-компонентів. JSX полегшує організацію та розуміння коду, а також дозволяє використовувати властивості HTML в JavaScript.

React розділяє інтерфейс користувача на невеликі, компоненти які можна перевикористовувати. Кожен компонент може мати власний стан та методи життєвого циклу, що полегшує розробку та управління додатками. При цьому він React використовує віртуальний DOM для оптимізації процесу оновлення інтерфейсу. Замість безпосереднього оновлення реального DOM, React створює віртуальне представлення DOM і порівнює його з попереднім станом для визначення необхідних змін.

Дані у React рухаються в одному напрямку, від батьківського компонента до дочірніх. Це спрощує відстеження стану додатка та впровадження визначеної логіки.

React має велику та активну спільноту розробників. Це призводить до великої кількості сторонніх бібліотек, компонентів та рішень, що спрощують розробку. З введенням Hooks у React 16.8, з'явився зручний спосіб використання стану та інших можливостей в функціональних компонентах, що дозволяє уникнути класового підходу, якщо це необхідно.

React є популярним вибором для розробки веб-додатків, особливо там, де важлива ефективність та декларативний підхід до побудови інтерфейсів. До недоліків можна віднести великий обсяг вивчення, та можливу складність ініціалізація проекту.

Vue.js - це прогресивний JavaScript-фреймворк для побудови користувацьких інтерфейсів [23]. Він часто порівнюється з React і Angular, але Vue визначається своєю простотою, гнучкістю та легкістю вивчення.

Основні характеристики Vue.js:

- Декларативний підхід до UI:

  Vue дозволяє розробнику описувати інтерфейс користувача в декларативному стилі, що полегшує розуміння та управління кодом.

- Компонентна архітектура:

  Vue побудований на основі компонентної архітектури, що дозволяє розділити інтерфейс на невеликі та повторно використовувані компоненти.

- Простота використання:

  Вивчення Vue.js досить просте, що робить його доступним для новачків. Розробник може поступово впроваджувати Vue в існуючий проект або використовувати його для створення нових додатків.

- Реактивність:

Vue використовує систему реактивності, яка автоматично оновлює відображення даних при їх зміні. Це дозволяє легко синхронізувати дані та інтерфейс.

- Добра документація та спільнота:

  Vue.js має докладну та зрозумілу документацію, а також активну спільноту розробників, яка готова допомогти та ділитися досвідом.

- Екосистема:

  Хоча екосистема Vue менша за React або Angular, вона все ж багатоцільова. Є багато плагінів та розширень для покращення функціональності та розширення можливостей Vue.js.

Загалом, Vue.js є відмінним вибором для розробки веб-додатків, особливо якщо вам важлива простота та ефективність в роботі з користувацьким інтерфейсом. До умовних недоліків можна віднести меншу спільноту, порівняно з React та обмеженішу екосистему.

Angular - це повноцінний фреймворк для створення веб-додатків. Основні риси Angular включають в себе використання TypeScript, розширену підтримку для реактивного програмування, повноцінну систему модулів, об'єктно-орієнтований підхід до розробки, інструменти для тестування та багато іншого [24].

Основні поняття Angular:

- Компоненти:

  Angular програми складаються з компонентів, які є основною будівельною одиницею. Компоненти містять логіку та представлення для частини користувацького інтерфейсу.

- Модулі:

  Angular додатки будуються з модулів, які об'єднують функціонально спрямовані компоненти та сервіси. Кожен додаток має принаймні один головний модуль, який містить кореневий компонент.

- Сервіси:

  Сервіси в Angular використовуються для спільного використання даних або функціональності між компонентами. Вони можуть бути використані для здійснення HTTP-запитів, обробки логіки бізнес-рівня та ін.

- Інжектори та Залежності:

  Angular використовує систему інжекторів для управління залежностями між компонентами та сервісами. Це дозволяє ефективно використовувати інші компоненти або сервіси, не створюючи їх вручну.

- Директиви:

  Angular надає директиви для розширення HTML та додавання нового функціоналу. Наприклад, ngFor використовується для повторення елементів у шаблоні, а ngIf - для управління видимістю елементів.

- Роутинг:

  Angular має вбудовану систему роутингу, що дозволяє створювати односторінкові додатки зі змінною вмістом без перезавантаження сторінки.

- Форми:

  Angular надає потужні засоби для створення та обробки форм, включаючи валідацію та взаємодію з даними користувача.

Таким чином Angular добре підходить для великих і складних додатків, де потрібна велика структура та підтримка об'єктно-орієнтованого програмування. Використання TypeScript дозволяє розробникам впроваджувати строгу типізацію та інші особливості сучасних версій ECMAScript. До недоліків можна віднести його складну структуру та великий обсяг коду, що вимагає глибокого розуміння TypeScript.

2.3.2 Роль та вибір інструментів збірки та розгортання

Велику роль в процесі розробки програмного забезпечення відіграють такі інструменти як збірки та розгортання (CI/CD). Ці інструменти допомагають автоматизувати і упорядковувати процеси збирання, тестування, розгортання та моніторингу програмного продукту. Основні етапи включають:

Процес *збірки (Build)* в розробці програмного забезпечення — це етап, на якому вихідний код перетворюється в виконуваний продукт чи бібліотеку. Це може включати компіляцію, пакування, оптимізацію та інші операції, щоб забезпечити коректну та ефективну роботу програми під час її виконання.

Ключові аспекти процесу збірки:

- Конфігурація збірки (Build Configuration):

  Конфігурація визначає, які етапи та операції повинні бути виконані під час збірки. Вона може бути визначена у конфігураційних файлах, які вказують параметри, такі як шляхи до вихідного коду, завдання збірки, опції компіляції тощо.

- Залежності (Dependencies):

  Збірка може включати у себе інші бібліотеки чи компоненти, які програма використовує в процесі роботи. Інструменти збірки вирішують залежності та включають необхідні ресурси у вихідний продукт.

- Компіляція (Compilation):

  У випадку компільованих мов програмування, таких як Java, C++ або C#, етап компіляції перетворює вихідний код у виконуваний код або проміжний бінарний код.

- Пакування (Packaging):

  На цьому етапі зібраний код та всі необхідні ресурси упаковуються в архів або інший формат, який може бути легко розповсюджений чи використовуваний.

- Тестування (Testing):

В деяких випадках, етап тестування може бути включений в процес збірки для автоматизованої перевірки якості та функціональності програми.

- Оптимізація (Optimization):

  Операції оптимізації можуть бути застосовані для покращення продуктивності, зменшення розміру виконуваного файлу чи інші оптимізаційні заходи.

- Документація (Documentation):

  Деякі інструменти збірки можуть генерувати документацію з вихідного коду, щоб забезпечити зручність розробників та користувачів.

Процес збірки може бути автоматизований за допомогою різних інструментів, таких як Maven, Gradle, Apache Ant для Java-проектів, або npm, webpack для проектів на JavaScript. Ці інструменти дозволяють легко керувати процесом збірки та створювати стандартизовані та автоматизовані рішення для розробників.

Тестування (Test): Тестування включає в себе процес перевірки функціональності, ефективності, інтерфейсу та інших аспектів програмного продукту без автоматизованих тестів. Ручне тестування є необхідною складовою частиною тестового процесу, особливо на етапах, коли автоматизовані тести не покривають всі можливі сценарії або коли потрібно провести тестування на користувачах.

Основні аспекти ручного тестування включають функціональне тестування (перевірка того, чи виконуються функціональні вимоги продукту), інтерфейсне тестування (оцінка зручності використання та дизайну інтерфейсу користувача), відповідь на помилки (Error Handling) (перевірка, як продукт веде себе в умовах неправильного введення або інших помилок)., спроби вторгнення (Security Testing) (оцінка безпеки продукту, виявлення потенційних вразливостей), тестування продуктивності (Performance Testing) (оцінка ефектив-

ності та швидкодії продукту в різних умовах навантаження) та тестування сумісності (Compatibility Testing) (визначення того, як продукт взаємодіє з різними платформами, браузерами, операційними системами тощо).

Ручне тестування використовується в комбінації з автоматизованим тестуванням для забезпечення широкого охоплення тестування та виявлення різноманітних проблем у програмному продукті.

Розгортання (Deploy): це процес впровадження програмного забезпечення або його частини в живе середовище, готове для використання. Цей процес може включати в себе різні етапи, від підготовки середовища до відповідного налаштування та запуску додатку. Основні аспекти розгортання включають:

Підготовка середовища:

Створення інфраструктури: Забезпечення наявності необхідних серверів, баз даних, мережевих налаштувань та інших компонентів.

Конфігурація середовища: Налаштування параметрів середовища, таких як змінні оточення, конфігураційні файли, налаштування баз даних та інше.

Збірка (Build) та Пакування (Package):

Створення виконуваного коду: Здійснюється компіляція програми та інші необхідні операції для отримання виконуваного коду.

Пакування або створення образу: Упакування виконуваного коду разом з усіма його залежностями та ресурсами у вигляді архіву чи образу контейнера (якщо використовується Docker).

Розгортання коду:

Фаза запуску: Розгортання нового коду або оновлення існуючого коду на цільовому середовищі.

Автоматизація: Використання інструментів автоматизації, таких як Ansible, Chef, або Puppet, для впровадження конфігурацій та коду на віддалених серверах.

Тестування після розгортання:

Перевірка цілісності: Виконання тестів, щоб переконатися, що новий код працює правильно та не впливає на роботу системи.

Моніторинг: Включення інструментів моніторингу для слідкування за продуктивністю та іншими метриками після розгортання.

Верифікація та валідація:

Підтвердження правильності: Перевірка, що нова версія програми працює як очікувалося та що вона відповідає всім вимогам.

Запуск тестових випадків: Виконання тестів для підтвердження, що система працює стабільно після внесення змін.

Відкат (Rollback):

Можливість відміни: Підготовка процедур для можливості відкату (відновлення до попередньої версії) у випадку помилки або непередбачених проблем.

Моніторинг після розгортання:

Слідкування за використанням ресурсів: Збір та аналіз метрик продуктивності, навантаження на сервер та інших параметрів для виявлення можливих проблем.

Виявлення аномалій: Встановлення інструментів для автоматичного виявлення аномалій та відправка сповіщень у випадку проблем.

Важливо, щоб процес розгортання був якомога більш автоматизованим та надійним, щоб забезпечити ефективність та безпеку релізів. Різні організації можуть використовувати різні інструменти для досягнення цієї мети, враховуючи особливості своїх проектів та вимог.

Моніторинг (Continuous Monitoring): Це спостереження за роботою програмного продукту після розгортання для виявлення можливих проблем.

Вибір інструментів збірки та розгортання визначається наступними факторами: спрощення процесів (інструменти повинні допомагати спростити та автоматизувати процеси розробки та розгортання) , сумісність з технологіями (важливо вибрати інструменти, які підтримують технології, що використову-

ються в проекті), гнучкість та розширюваність (інструменти повинні бути гнучкими та легко розширюваними для відповіді на змінні потреби проекту), спільнота та підтримка (інструменти, які користуються підтримкою та активною спільнотою, забезпечують доступ до допомоги та оновлень), інтеграція з іншими інструментами (важливо, щоб інструменти були сумісні з іншими елементами стеку розробки, такими як системи контролю версій, тестування, моніторингу тощо).

Усі ці аспекти взаємодіють для забезпечення швидкого та надійного циклу розробки та постійного вдосконалення програмного продукту.

2.4 Оптимізація та ефективність компонентів

Оптимізація та вимірювання ефективності компонентів в бібліотеці компонентів є важливими завданнями при розробці програмного забезпечення [25]. Оптимізація спрямована на поліпшення швидкодії та ресурсоємності компонентів, тоді як метрики дозволяють об'єктивно вимірювати цю ефективність.

2.4.1 Техніки оптимізації продуктивності компонентів

Оптимізація продуктивності компонентів бібліотеки включає в себе ряд технік та стратегій, спрямованих на поліпшення швидкодії, ефективності використання ресурсів та загальної продуктивності. Ось докладний огляд деяких ключових технік оптимізації:

- Профілювання коду:

  Визначення того, як саме компонент використовує ресурси (час, пам'ять, процесор). Використання інструментів для вимірювання часу виконання, аналізу стеку викликів, використання пам'яті та інших характеристик.

- Використання ефективних алгоритмів:

Вибір оптимальних алгоритмів для виконання конкретних завдань. Уникання непотрібних операцій або визначення шляхів для їх оптимізації.

- Лініаризація пам'яті:

Зменшення фрагментації пам'яті шляхом використання більших блоків пам'яті та мінімізація частоти виділення/звільнення пам'яті.

- Кешування даних:

Використання кешів для тимчасового зберігання результатів обчислень або доступу до даних. Використання локальних кешів для зберігання часто використовуваних даних.

- Асинхронність та паралелізм:

Використання асинхронних операцій для уникнення блокування інтерфейсу користувача або інших чутливих застосунків.
Використання паралельних обчислень для використання багатоядерних систем та підвищення продуктивності.

- Ліниве завантаження та оптимізація ресурсів:

Відкладання завантаження ресурсів (зображень, даних) до того моменту, коли вони фактично потрібні. Використання адаптивного завантаження ресурсів залежно від характеристик пристрою та мережі.

- Мінімізація обсягу даних:

Стискання або мінімізація обсягу передаваних та оброблюваних даних. Використання легковажних форматів обміну даними (наприклад, JSON замість XML).

- Оптимізація анімацій та інтерфейсу:

Використання апаратного прискорення для плавних анімацій. Мінімізація кількості перерисовувань та використання оптимізованих графічних елементів.

- Керування пам'яттю:

Ефективне управління пам'яттю, уникання витоків пам'яті та зайвого використання.

- Інструменти та бібліотеки для оптимізації:

  Використання спеціалізованих інструментів та бібліотек для оптимізації коду та виявлення проблем продуктивності (наприклад, профілювальних інструментів, JIT компіляторів, оптимізаторів коду).

Завжди треба пам’ятати, що конкретні техніки оптимізації можуть варіюватися в залежності від платформи, мови програмування та характеристик конкретного застосунку. Оптимізація завжди має бути обґрунтованою та спрямованою на конкретні потреби проекту.

2.4.2 Метрики оцінки ефективності компонентів

Оцінка ефективності компонентів у бібліотеці компонентів вимагає використання різноманітних метрик для об'єктивного вимірювання їх продуктивності, надійності та інших характеристик. Нижче подано докладний огляд ключових метрик оцінки ефективності компонентів:

- Час виконання.

  Час виконання функцій або операцій метрика, що вимірює час, який компонент витрачає на виконання конкретних завдань. Метрика може включати загальний час виконання та середній час на операцію.

- Використання ресурсів:

  Використання пам'яті це міра обсягу пам'яті, який використовує компонент. Може включати обсяг оперативної пам'яті та може допомагати виявляти можливі витоки пам'яті.

  Використання процесора параметр, що вимірює частоту та тривалість використання ресурсів процесора компонентом.

- Стійкість до помилок:

  Кількість виявлених помилок це кількість помилок, виявлених під час тестування або експлуатації компонента.

Час відновлення після помилок це час, який потрібен для компонента для відновлення до нормального стану після виникнення помилки.

- Швидкодія інтерфейсу користувача:

Час відгуку інтерфейсу це час, який компонент витрачає на відгук на взаємодію користувача. Кількість оновлень інтерфейсу на секунду: Для графічних компонентів - кількість кадрів на секунду.

- Обсяг коду та розмір файлів:

Кількість рядків коду метрика, що вимірює обсяг самого коду компонента. Розмір файлів це обсяг файлів, які входять до складу компонента, може бути важливим для завантаження та розпакування.

- Покриття тестами:

Процент покриття тестами метрика, що визначає, яка частина коду компонента охоплена тестами. Високий рівень покриття сприяє надійності.

- Масштабованість:

Час виконання при великому обсязі даних: Метрика визначає, наскільки ефективно компонент працює при збільшенні обсягів вхідних даних або користувачів.

- Вартість обслуговування:

Час на розробку, тестування та виправлення помилок: Оцінює кількість ресурсів, витрачених на розробку та утримання компонента.

- Використання залежностей:

Кількість та тип залежностей це метрика, що визначає, наскільки компонент залежить від інших бібліотек чи компонентів.

- Взаємодія з іншими компонентами:

Сумісність це метрика яка визначає, наскільки легко компонент може взаємодіяти з іншими компонентами в рамках системи.

Важливо визначити конкретні метрики відповідно до вимог проекту та його контексту. Крім того, важливо враховувати, що однією метрикою може

бути недостатньо для повного оцінювання ефективності компонента, і часто використовують комплексні підходи для отримання більш повної карти.

2.5 Безпека та надійність у бібліотеці компонентів

Питання безпеки та надійності у бібліотеці компонентів є критичними аспектами при розробці програмного забезпечення.

2.5.1 Забезпечення безпеки компонентів

Можна виділити декілька ключових аспектів реалізації безпеки компонентів в бібліотеці компонентів:

- Перевірка вразливостей (Security Vulnerability Assessment):

  Перед включенням будь-якого компонента до бібліотеки важливо провести аналіз безпеки для виявлення можливих вразливостей. Це може включати огляд відкритих джерел коду, аудит безпеки, тестування на проникнення та інші заходи. Прикладом може слугувати Використання інструментів статичного аналізу коду, таких як ESLint чи SonarQube, для виявлення потенційних вразливостей в коді компонентів.

- Оновлення та Патчі (Updates and Patches):

  Забезпечення своєчасних оновлень та патчів для компонентів важливо для усунення виявлених вразливостей та забезпечення безпеки. Для цього можна використання сервісів, які автоматично перевіряють вразливості залежностей, наприклад, npm audit для проектів на Node.js. Потрібна регулярна перевірка відомих джерел безпеки та автоматизоване оновлення компонентів через інструменти керування залежностями, такі як npm або Maven.

- Автентифікація та Авторизація (Authentication and Authorization):

Забезпечення правильної автентифікації та авторизації при використанні компонентів для запобігання несанкціонованому доступу до системи. Як варіант можливе використання HTTPS для зашифрування комунікації між компонентами бібліотеки та захисту від SQL-ін'єкцій через використання параметризованих запитів.

- Криптографічна Безпека:

Використання сильних алгоритмів шифрування та правильне управління ключами для захисту конфіденційності даних.

- Моніторинг та Журналювання (Monitoring and Logging):

Реалізація систем моніторингу та журналювання для вчасного виявлення та реагування на потенційні проблеми безпеки. Можливе використання інструментів моніторингу, таких як Prometheus, та ведення журналів аудиту заходів безпеки з допомогою системи журналювання, такої як ELK (Elasticsearch, Logstash, Kibana).

- Тестування Безпеки (Security Testing):

Проведення регулярних тестів на безпеку, таких як тестування на проникнення, для перевірки стійкості системи до можливих атак, виявлення можливих слабких місць безпеки, таких як вразливості вводу через форми або невірне керування сесією.

- Надійність та Відновлення (Reliability and Recovery):

Розробка механізмів відновлення та резервування для забезпечення надійності системи в умовах непередбачуваних ситуацій.

- Забезпечення Конфіденційності та Інтеграція (Confidentiality and Integration):

Захист конфіденційності даних та забезпечення сумісності та інтеграції компонентів у загальну систему.

- Керування Доступом (Access Control):

Ефективне управління правами доступу до функцій та ресурсів бібліотеки для забезпечення принципу найменших привілеїв.

- Документація та Спільнота (Documentation and Community):

Наявність чіткої документації та активної спільноти розробників може покращити якість безпеки та надійності, дозволяючи швидше виявляти та вирішувати проблеми. Для цього потрібно розробляти чітку документацію безпеки, яка включає поради щодо безпеки, інструкції щодо безпеки та регулярні звіти щодо стану безпеки для внутрішнього та зовнішнього використання.

Загальна стратегія безпеки та надійності повинна бути вбудована у весь життєвий цикл розробки програмного забезпечення, від вибору компонентів до етапу експлуатації.

2.5.2 Стратегії тестування та валідації компонентів

Стратегія тестування та валідації компонентів в бібліотеках компонентів включає в себе кілька етапів, щоб забезпечити високу якість та надійність коду [26]. Нижче наводиться загальний опис такої стратегії:

- Розробка юніт-тестів:

Для цього потрібно спроектувати тестові випадки для кожного компонента бібліотеки. Можна використовувати автоматизовані тести для перевірки основних функціональних можливостей кожного компонента. Потрібно включити різні сценарії введення та обробки помилок.

- Інтеграційне тестування:

Необхідно визначити, як компоненти взаємодіють між собою та з іншими частинами системи. Треба створити тести для перевірки взаємодії та сумісності компонентів. Також врахувати різні конфігурації та вхідні дані для виявлення потенційних проблем.

- Функціональне та негативне тестування:

Для цього потрібно використати тести для перевірки коректності роботи кожного функціонального елементу. Провести тести на обробку некоректних даних, помилок та винятків.

- Тестування великого навантаження та продуктивності:

Для цього мають бути створені тести для перевірки великої кількості одночасних запитань та обробки даних. Треба визначити максимальні можливості бібліотеки та переконатися, що вона ефективно впорається з великою кількістю запитань.

- Тестування безпеки:

На цьому етапі необхідно проводити аудит безпеки для знаходження потенційних вразливостей. Необхідно перевіряйте обробку вхідних даних для попередження атак, таких як SQL-ін'єкції, введення з боку, тощо.

- Валідація документації:

Будь який розробник має переконатися, що документація для кожного компонента є актуальною та вичерпною. Для цього потрібно використовувати тести для перевірки прикладів використання, наведених у документації.

- Автоматизація тестів:

Має великий сенс використання інструментів автоматизації тестування для виконання тестів під час кожного внесення змін у код. Дуже важливо забезпечити швидкість та повторюваність тестування.

- Тестування на різних платформах та середовищах:

І на сам кінець потрібно використовувати тести для перевірки роботи бібліотеки на різних операційних системах, браузерах та інших середовищах.

Ці етапи допомагають створити вичерпну стратегію тестування та валідації компонентів в бібліотеках для забезпечення високого рівень якості та надійності коду.

2.6 Інтеграція бібліотеки компонентів в інформаційні системи

Інтеграція бібліотеки компонентів в інформаційні системи є ключовим етапом розробки та управління програмними проектами. Цей процес включає в себе об'єднання різних компонентів (частин програмного забезпечення) в єдиний функціональний блок для забезпечення взаємодії та спільної роботи між ними.

Можна сформулювати деякі ключові аспекти інтеграції бібліотеки компонентів:

- Визначення компонентів:

  Інтеграція розпочинається з визначення компонентів, які мають бути об'єднані в системі. Компоненти можуть бути розроблені власноруч, використовувати сторонні бібліотеки чи фреймворки.

- Стандарти і протоколи:

  Важливо встановити стандарти та протоколи для взаємодії між компонентами. Це включає в себе вибір форматів даних, способів забезпечення безпеки та механізми обміну інформацією.

- Модульність:

  Забезпечення модульності важливо для зручності управління та розробки. Кожен компонент повинен бути самостійним і можливим для заміни чи оновлення без впливу на інші компоненти.

- Тестування:

  Налаштування ефективного процесу тестування для переконання в тому, що інтеграція не порушує функціональність компонентів. Випробовування різних сценаріїв та взаємодії для виявлення можливих проблем.

- Керування конфігурацією:

  Необхідність забезпечити ефективне керування конфігурацією для відстеження змін у коді та конфігураціях компонентів.

- Моніторинг та логування:

Впровадження системи моніторингу та логування дозволяє відстежувати роботу системи та виявляти можливі проблеми.

- Сумісність версій:

Важливо враховувати сумісність версій компонентів та забезпечити можливість їх оновлення без перебоїв в роботі системи.

Якщо дотримуватись наслідування цим аспектам, то інтеграція бібліотеки компонентів дозволить створювати потужні та гнучкі інформаційні системи, які можуть ефективно взаємодіяти та вирішувати різноманітні завдання.

2.6.1 Підходи до інтеграції та взаємодії з іншими системами

Існує кілька підходів до інтеграції та взаємодії з іншими системами, і вибір конкретного методу може залежати від конкретних вимог проекту, технічних характеристик систем та інших факторів. Маємо дуже стисло розглянути декілька основних підходів:

Point-to-Point (P2P) Integration: Цей підхід включає в себе пряме підключення між двома системами. Це може бути ефективно для невеликої кількості взаємодій, але стає складним для управління великою кількістю точкових з'єднань.

Hub-and-Spoke (uintegration hub) Integration: Використовує центральний хаб, який обробляє всі з'єднання між системами (спільний пункт обміну даними). Кожна система спілкується тільки з хабом, що забезпечує спрощення комунікацій та керування.

Message-Oriented Middleware (MOM): Використовує асинхронний обмін повідомленнями між системами через спеціальний посередник (брокера повідомлень). Забезпечує роботу з різними технологіями та мовами програмування.

Service-Oriented Architecture (SOA): Розбиває функціональність на невеликі, незалежні служби, які можуть бути викликані через мережу. Використовує стандартизовані протоколи (наприклад, SOAP або REST) для обміну даними між службами.

Application Programming Interface (API) Integration: Використовує інтерфейси програмування застосунків для забезпечення доступу до функціональності системи через стандартизовані запити та відповіді.

Event-Driven Architecture (EDA): Заснована на обміні подіями між системами. Компоненти реагують на події та сповіщають інші системи про зміни.

Data Synchronization: Синхронізація даних між різними системами. Використовується для забезпечення консистентності даних між джерелами.

Кожен з цих підходів має свої переваги та недоліки, і вибір конкретного підходу повинен враховувати потреби та характеристики конкретного проекту. Комбінування різних підходів часто використовується для досягнення оптимальної інтеграції між системами.

Нажаль біль детальний огляд цих підходів займе дуже багато часу та виходить за рамки цієї Роботи.

2.6.2 Випадки використання та приклади реалізацій

Наприкінці другого розділу розглянемо деякі випадки використання та приклади реалізації:

Графічні бібліотеки в веб-додатках:

Приклад: Інтеграція бібліотеки D3.js для візуалізації даних в реальному часі у веб-додатку для аналізу фінансових ринків.

Бібліотеки для обробки мови в прикладах чат-ботів:

Приклад: Використання бібліотеки Natural Language Toolkit (NLTK) для обробки тексту та розпізнавання інтенцій в чат-боті для клієнтського обслуговування.

Інтеграція платіжних систем у веб-магазинах:

Приклад: Використання бібліотеки Stripe або PayPal SDK для забезпечення безпечних та зручних оплат товарів у веб-магазині.

Інтеграція соціальних мереж в додатках:

Приклад: Додавання можливості авторизації через соціальні мережі (наприклад, Facebook або Google) за допомогою бібліотеки OAuth.

Геопросторова інтеграція в мобільних додатках:

Приклад: Використання Google Maps API для інтеграції геолокації та навігації в мобільному додатку для доставки їжі.

Інтеграція аналітичних бібліотек у системах управління підприємством (ERP):

Приклад: Використання бібліотеки Apache Flink для реального аналізу даних в ERP-системі для покращення стратегічного прийняття рішень.

Інтеграція медичних інформаційних систем:

Приклад: Використання бібліотеки HL7 (Health Level Seven) для стандартизованого обміну медичною інформацією між різними системами у сфері охорони здоров'я.

Ці приклади вказують на широкий спектр можливостей для інтеграції бібліотек компонентів у різні типи інформаційних систем, що сприяє розвитку функціональності та забезпечує взаємодію між різними частинами програмного забезпечення.

2.7 Висновки з розділу 2

В даному розділі були розглянуті наступні питання:

1. Структурних аспектів бібліотеки компонентів. Модульність, повторне використання компонентів, принципи інкапсуляції та ізоляції в бібліотеці;
2. Зроблено огляд технологічних підходів в бібліотеці компонентів, використання шаблонів проектування в компонентах, використання принципів SOLAD та DRY у розробці;

3. Розглянуто питання фреймворків та інструментів для розробки бібліотек, проаналізовані популярні фреймворкі, досліджена роль та вибір інструментів для розгортання;
4. Розглянути техніки оптимізації продуктивності компонентів та метрики оцінки ефективності компонентів;
5. Досліджено питання безпеки та надійності біліотеки компонентів, методи забезпечення безпеки та стратегії тестування та валідації компонентів;
6. Розглянуте питання інтеграції бібліотеки компонентів в інформаційні системи, досліджені різні підходи до цього питання, та взаємодії з іншими системами.

РОЗДІЛ 3 РОЗРОБКА ВИБІР СЕРЕДОВИЩА, ФРЕЙМВОРКІВ ТА СТОРОННІХ БІБЛІОТЕК ДЛЯ РЕАЛІЗАЦІЇ ПРОЕКТУ

3.1 Налаштування апаратного середовища

Для вирішення поставленої задачі було обрано ноутбук Lenovo з наступними основними характеристиками:

Назва ОС Майкрософт Windows 11 Home

Версія 10.0.22621 Збірка 22621

Тип x64-based PC

Процесор 13th Gen Intel(R) Core(TM) i7-13700K 3.40 GHz

Відеокарта NVIDIA GeForce GTX 1080

Установлена фізична пам’ять (ОЗП) 32,0 ГБ

3.1.1 Visual Studio Code

В процесі розробки бібліотеки компонентів було використано за необхідністю два редактору коду. Перший з них це Visual Studio Code - легкий, потужний та безкоштовний редактор коду, розроблений компанією Microsoft.

Рисунок 12 — *Логотип VS Code*

Ось деякі ключові риси цього інструменту:

Відкритий код: VS Code є відкритим програмним забезпеченням, що означає, що його можна вільно використовувати, модифікувати та розповсюджувати в межах ліцензії MIT.

Легкість використання: Інтерфейс VS Code дуже простий та інтуїтивний. Він надає базовий набір функцій, які легко вивчати, але в той же час має широкий спектр можливостей для просунутих користувачів.

Мови програмування: VS Code підтримує багато мов програмування, включаючи, але не обмежуючись, JavaScript, TypeScript, Python, C#, Java, HTML, CSS та інші. Ви можете встановлювати розширення для підтримки конкретних мов та інструментів розробки.

Розширення: Велика перевага VS Code - це екосистема розширень. Ви можете легко встановлювати розширення для розширення функціональності редактора, додаючи нові можливості, синтаксичне підсвічування, підтримку фреймворків та інше.

Інтеграція з Git: Вбудована підтримка Git дозволяє легко вести контроль версій вашого коду, переглядати зміни та виконувати коміти прямо з редактора.

Налагодження: VS Code підтримує налагодження для різних мов програмування, забезпечуючи можливість встановлення точок зупинки, аналізу стеку викликів та інші інструменти для полегшення налагодження коду.

3.1.2 WebStorm

Основний масив розробки виконувався в WebStorm. Це дуже зручний інструмент, яки являє собою інтегроване середовище розробки (IDE) для веброзробки, створене компанією JetBrains. Воно спеціалізується на розробці вебпрограм з використанням таких мов, як JavaScript, TypeScript, HTML, CSS, і Node.js.

Рисунок 13 — *Логотип WebStorm*

Ось деякі ключові особливості WebStorm:

Підтримка мов програмування: WebStorm надає підтримку для різних мов, таких як JavaScript, TypeScript, HTML, CSS, LESS, SASS, і багатьох інших. Також підтримується робота з фреймворками, такими як Angular, React, і Vue.js.

Автоматичне завершення коду: Інтелектуальне автодоповнення коду полегшує написання коду, допомагаючи визначати функції, методи та властивості об'єктів.

Вбудована підтримка систем контролю версій: WebStorm добре інтегрований з популярними системами контролю версій, такими як Git, Mercurial, і SVN.

Інструменти рефакторингу: Забезпечує ряд інструментів для полегшення рефакторингу коду, що дозволяє покращити його структуру та ефективність.

Вбудовані інструменти для тестування: WebStorm підтримує вбудовані інструменти для запуску та відлагодження тестів, зокрема для фреймворків, таких як Jest, Mocha, і Karma.

Live Editing та перегляд змін: Дозволяє відслідковувати зміни в коді в реальному часі без перезавантаження сторінки.

Інтеграція з іншими інструментами JetBrains: WebStorm може інтегруватися з іншими продуктами JetBrains, такими як IntelliJ IDEA, PyCharm, і т.д., що спрощує роботу в різних мовах програмування.

Отже ці два редактори коду повністю задовольнили потреби при розробці бібліотеки компонентів мовою Typescript, на якій зупинимось далі.

3.1.3 TypeScript

TypeScript - це мова програмування, яка є розширенням JavaScript, призначена для роботи з великими проектами, де важлива стабільність і підтримка коду з більшою кількістю розробників. При розробці бібліотеки компонентів, TypeScript значно полегшує роботу, забезпечуючи зручність в організації коду, його розуміння та підтримку великих проектів.

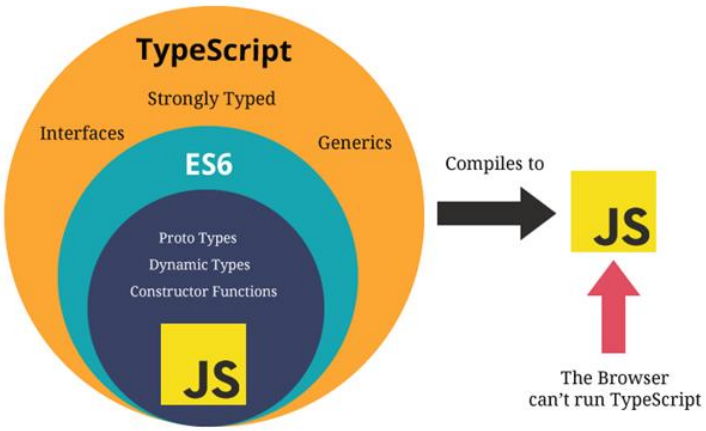

Рисунок 14 — *Чому TypeScript*

Основні переваги TypeScript у розробці бібліотеки компонентів:

Статична типізація: TypeScript дозволяє визначати типи даних для змінних, параметрів функцій, об'єктів та інших елементів коду. Це допомагає виявляти помилки на етапі розробки, забезпечуючи більшу стабільність і підтримку коду.

Підтримка сучасних стандартів ECMAScript: TypeScript підтримує останні версії стандарту ECMAScript, що дозволяє використовувати нові функції та покращення JavaScript, підтримуючи при цьому сумісність з попередніми версіями.

Робота з інтерфейсами та класами: TypeScript сприяє використанню інтерфейсів та класів для визначення структури та поведінки компонентів, що полегшує організацію коду та розширення функціональності.

Автоматична генерація документації: TypeScript може генерувати автоматичну документацію з коментарів у коді, що допомагає іншим розробникам легше розуміти та використовувати вашу бібліотеку.

Широка підтримка від спільноти: TypeScript має активну спільноту розробників, яка надає багато корисних інструментів, бібліотек та рекомендацій, спрощуючи розробку та підтримку проекту.

3.2 Фреймворки

При реалізації проекту було обрано два фреймворки, а саме Node.js та Express.js. Тому наведемо короткий опис цих інструментів.

3.2.1 Node.js

Node.js - це виконавче середовище для JavaScript, яке дозволяє виконувати код на сервері. Однією з його ключових особливостей є можливість асинхронного програмування, що дозволяє ефективно обробляти багатозадачні операції без блокування виконання інших завдань.

Основні переваги Node.js в розробці бібліотек компонентів:

JavaScript на обох сторонах:

Node.js використовує мову JavaScript, що дає можливість розробникам використовувати одну мову програмування (JavaScript) як на клієнтській, так

і на серверній сторонах. Це спрощує взаємодію між клієнтом і сервером, а також полегшує розробку.

Ефективність та висока швидкість:

Двигун V8, що використовується у Node.js, забезпечує високу швидкість виконання коду. Це дозволяє розробникам створювати високопродуктивні застосунки та бібліотеки.

Модульність:

Node.js працює з модульною системою, що дозволяє розробникам легко використовувати сторонні бібліотеки та модулі для полегшення розробки.

Асинхронність:

Підтримка асинхронного програмування дозволяє ефективно виконувати операції вводу-виводу та мережеві запити, не блокуючи інші операції.

Підтримка пакетного менеджера npm:

Node.js використовує npm (Node Package Manager) для управління залежностями та пакетами, що робить легким розгортання та управління бібліотеками.

Спільнота та екосистема:

Node.js має велику та активну спільноту розробників, а також широкий вибір сторонніх бібліотек і модулів, що полегшує розробку.

Ці переваги роблять Node.js популярним вибором для розробки бібліотек компонентів, особливо в контексті серверної сторони веб-застосунків.

3.2.2 Express.js

Express.js це фреймворк для розробки веб-додатків на мові JavaScript, побудований на основі Node.js. Більшою частиною використовувався саме він. Ось деякі з його основних переваг у розробці бібліотеки компонентів:

Легкість використання: Express.js робить розробку веб-додатків швидкою та простою. Він надає простий інтерфейс для створення серверів і обробки HTTP-запитів, що полегшує розробку бібліотек компонентів.

Розширюваність: Express.js дозволяє легко розширювати можливості за допомогою сторонніх модулів і middleware. Це робить його ефективним інструментом для побудови різноманітних компонентів та модулів.

Маршрутизація: Express надає потужні інструменти для визначення маршрутів, що дозволяє організовувати логіку додатка та взаємодіяти з різними запитами.

Middleware: Система middleware дозволяє вам легко обробляти різні етапи обробки запиту. Це може бути корисно при розробці бібліотек, які потребують специфічної обробки для різних етапів запиту.

Активна спільнота: Express.js має велику та активну спільноту розробників, що дозволяє отримати підтримку, знайти рішення для проблем і використовувати сторонні пакети.

Отже ці два фреймворки повністю задовільнили потреби в своїй частині при розробці бібліотеки компонентів.

3.3 Сторонні бібліотеки

При реалізації поставленого завдання розробки бібліотеки компонентів використовувались різні сторонні бібліотеки, таки як `body-parser`, CORS, `reflect-metadata`, eslint, nodemon та ts-node. Але з точки зору важливості варто зосередитись на перших трьох.

3.3.1 `body-parser`

Ця бібліотека безпосередньо не пов'язана з розробкою бібліотеки компонентів, але вона використовувалась для обробки даних, які надходять в запитах HTTP.

body-parser призначений для обробки тіла запиту (body) від клієнта в середовищі Node.js. Основна його задача - розпарсити тіло запиту, яке може бути у формі JSON, URL-encoded або іншого типу, залежно від вказівки клієнта.

Основні функції body-parser включають:

json() - розпарсює JSON-форматовані дані.

«const bodyParser = require('body-parser'); app.use(bodyParser.json())»;

urlencoded() - розпарсює дані, що передаються у формі URL-encoded.

«app.use(bodyParser.urlencoded({ extended: true }));»

raw() - розпарсює тіло у вигляді буфера.

«app.use(bodyParser.raw());»

text(), який розпарсює текстові дані.

«app.use(bodyParser.text());»

Після встановлення і налаштування body-parser, вже можна звертатися до об'єкта req.body для отримання розпарсених даних у обробнику маршруту.

3.3.2 CORS

З бібліотекою "CORS" (Cross-Origin Resource Sharing) пов'язані заборони на виконання запитів з різних джерел на стороні клієнта. Це застосовується веб-браузерами для безпеки, щоб уникнути можливих атак, таких як атаки типу Cross-Site Request Forgery (CSRF) та інші.

Оскільки в проекті розробляється бібліотеку компонентів, яка взаємодіє з різними джерелами даних або серверами, може виникнути проблема, якщо веб-додаток запускається на одному домені, а дані або ресурси намагаються завантажитися з іншого домену. Це порушує політику same-origin, і браузер може блокувати такі запити.

Щоб вирішити цю проблему, було використовано бібліотеку CORS на стороні сервера. CORS дозволяє серверу вказати, з яких джерел дозволяється запитувати ресурси. Це робиться за допомогою HTTP-заголовків, таких як Access-Control-Allow-Origin.

При розробці бібліотеки компонентів, довелося враховувати можливість взаємодії з CORS. Було перевірено чи був сервер належним чином налаштований для обробки запитів від інших джерел та чи встановлюються необхідні HTTP-заголовки.

3.3.3 `reflect-metadata`

Бібліотека reflect-metadata - це бібліотека для роботи з метаданими у JavaScript та TypeScript. Метадані - це інформація про код, яка не входить до виконавчого коду, але може бути корисною для різних інструментів та бібліотек.

Основна задача reflect-metadata полягала в надані способу читання та запису метаданих для об'єктів у TypeScript. Це особливо корисно в контексті розробки фреймворків та бібліотек, які працюють з метаданими для створення динамічних рішень.

Основні функції reflect-metadata:

Reflect.metadata: Дозволяє додавати метадані до об'єктів.

Reflect.getMetadata: Дозволяє отримувати метадані з об'єктів.

Reflect.defineMetadata: Дозволяє визначати нові метадані або змінювати існуючі.

Ця бібліотека широко використовується в розробці фреймворків та бібліотек, які працюють із залежностями та компонентами. Наприклад, в Angular, reflect-metadata використовується для збереження метаданих компонентів та їх залежностей. Це дозволяє фреймворку динамічно будувати та інжектувати залежності під час виконання.

В розробці бібліотеки компонентів, використання reflect-metadata було корисним для динамічного аналізу та використання метаданих, пов'язаних з компонентами. Метадані використовувались для визначення типів властивостей компонентів, їх залежностей та деяких конфігураційних параметрів.

В ході розробки також використовувались такі бібліотеки як eslint, nodemon, ts-node опис яких ми наводити не будемо.

3.4 Висновки з розділу 3

В цьому розділі були розглянуті основні етапи підготовки до реалізації поставленої задачі та зроблені наступні висновки:

1. Задачу розробки бібліотеки компонентів можливо реалізувати сучасними програмними та технічними засобами;

2. Для реалізації проекту було використано програмне середовище Visual Studio Code, WebStorm, мова програмування TypeScript;

3. Вибрані наступні фреймворки та бібліотеки для роботи: Фреймворкі: Node.js, Express.js. Бібліотеки: `body-parser`, CORS, `reflect-metadata`, eslint, nodemon та ts-node.

РОЗДІЛ 4 АНАЛІЗ СТРУКТУРИ РОЗРОБЛЕНОЇ БІБЛІОТЕКИ КОМПОНЕНТІВ

4.1 Декоратори класів та функцій

В процесі розробки бібліотеки були створені такі типи декораторів:

- Декоратори класів, які маркують декоровані класи ключем контролера (лістинг 15).
- Декоратори методів, які маркують декоровані методи в класі контролері ключем, який відповідає за тип REST метода (лістинг 16).
- Декоратори параметрів, які маркують параметри методів, які помічені як REST методи, задля інжекції частин користувацького запиту в якості аргументу методу (лістинг 17).
- Декоратори класів, які маркують декоровані класи ключем middleware, що дозволяє аналізувати запит і приймати рішення до того, як буде виконаний метод контролеру (лістинг 18).

Лістинг 15 *Код функції-декоратора контролера*

```
    export function Controller(basePath = "/"):
ClassDecorator {
      return function <TFunction extends Function>(target:
TFunction) {
        Reflect.defineMetadata(controllerMetadataKey,
basePath, target);
        Reflect.defineMetadata(transientMetadataKey, "",
target);
        return target;
      };
    }
```

В наведеному вище лістингу оголошена функція-декоратор, яка додає до декорованого класу метадані, в яких міститься базовий шлях контролера за ключем контролера.

Лістинг 16 *Код функції-декоратора для REST методу*

```
export function Post(path = ""): MethodDecorator {
  return function (
    target: object,
    propertyKey: string | symbol,
    descriptor: PropertyDescriptor,
  ): void {
    registerRestAction(
      target,
      propertyKey,
      descriptor,
      path,
      postMetadataKey,
      true,
    );
  };
}
```

В наведеному вище лістингу оголошена функція-декоратор для «POST» методу. Функції-декоратори для інших методів аналогічні.

Лістинг 17 *Код функції, яка помічає аргумент як такий, що має прийти з параметра URL з відповідною назвою*

```
export function Param(name: string): ParameterDecorator
{
  return function (
    target: object,
    propertyKey: string | symbol | undefined,
    parameterIndex: number,
  ) {
    if (!propertyKey) {
      return;
    }
    const existingParams: ParamMetadata[] =
      Reflect.getOwnMetadata(paramMetadataKey, target,
propertyKey) || [];

    existingParams.push({ position: parameterIndex,
name });
    Reflect.defineMetadata(
      paramMetadataKey,
```

```
            existingParams,
            target,
            propertyKey,
          );
        };
      }
```

Лістинг 18 *Код middleware декораторів*

```
      export function Middleware(): ClassDecorator {
        return function <TFunction extends Function>(target:
TFunction) {
          Reflect.defineMetadata(transientMetadataKey, "",
target);
          return target;
        };
      }

      export function UseBefore(...middlewares:
ConstructorType[]): any {
        return
registerMetadataAndReturnDecorator(middlewares,
useBeforeMetadataKey);
      }

      export function UseAfter(...middlewares:
ConstructorType[]): any {
        return
registerMetadataAndReturnDecorator(middlewares,
useAfterMetadataKey);
      }

      function registerMetadataAndReturnDecorator(
        middlewares: ConstructorType[],
        metadataKey: symbol,
      ) {
        return (
          target: unknown,
          propertyKey?: string | symbol,
          descriptor?: TypedPropertyDescriptor<unknown>,
        ) => {
          if (propertyKey && descriptor) {
            return
registerMethodMetadataAndReturnDecorator(middlewares,
metadataKey)(
              target as object,
              propertyKey,
```

```
            descriptor,
          );
        } else {
          return registerClassMetadataAndReturnDecorator(
            middlewares,
            metadataKey,
          )(target as UnknownFunction);
        }
      };
    }

    function registerMethodMetadataAndReturnDecorator(
      middlewares: ConstructorType[],
      middlewareKey: symbol,
    ): MethodDecorator {
      return function (
        _: object,
        __: string | symbol,
        descriptor: PropertyDescriptor,
      ) {
        if (middlewares.length === 0) {
          return;
        }
        Reflect.defineMetadata(middlewareKey, middlewares,
descriptor.value);
      };
    }

    function registerClassMetadataAndReturnDecorator(
      middlewares: ConstructorType[],
      middlewareKey: symbol,
    ): ClassDecorator {
      return function (target: object) {
        if (middlewares.length === 0) {
          return;
        }
        Reflect.defineMetadata(middlewareKey, middlewares,
target);
      };
    }
```

4.2 Імплементація DI контейнера для бібліотеки компонентів

При створенні подібних бібліотек компонентів складно обійтися без використання DI контейнера для контрольованого і уніфікованого створення об’єктів.

Для бібліотеки був розроблений власний DI контейнер код якого наведений у лістингу 19.

Лістинг 19 *Імплементація DI контейнера*

```
    export class DiContainer {
      private readonly singletons: Record<string, unknown>
= {};

      public get<T>(type: ConstructorType<T>,
creationContext: symbol[] = []): T {
        const keys: symbol[] =
Reflect.getMetadataKeys(type);

        const key = keys.find(
          (k) => k === singletonMetadataKey || k ===
transientMetadataKey,
        );

        if (!key) {
          throw new DependencyInjectionError(
            `${type.name} is not decorated with proper
decorator and cannot be created.`,
          );
        }

        if (
          creationContext.includes(singletonMetadataKey) &&
          key !== singletonMetadataKey
        ) {
          throw new DependencyInjectionError(
            `It is impossible to create ${type.name}
because it is not singleton but creation context contains
singleton`,
          );
        }

        if (!!this.singletons[type.name]) {
          return this.singletons[type.name] as T;
        }

        const ctor = type.prototype["constructor"] as
UnknownFunction;

        const paramTypes =
Reflect.getMetadata("design:paramtypes", ctor);
```

```
        if (!paramTypes?.length) {
          creationContext.push(key);
          const object = new type() as T;

          if (key === singletonMetadataKey) {
            this.singletons[type.name] = object;
          }

          return object;
        }

        const params = [];

        for (const paramType of paramTypes) {
          creationContext.push(key);
          const obj = this.get(paramType,
[...creationContext]);
          params.push(obj);
        }

        const object = new type(...params) as T;

        if (key === singletonMetadataKey) {
          this.singletons[type.name] = object;
        }

        return object;
      }
    }
```

4.3 Аналіз отриманих результатів

Доцільність використання бібліотеки була підтверджена практичним шляхом.

На лістингу 20 показане можливе практичне використання розробленої бібліотеки компонентів

Лістинг 20 *Приклад використання розробленої бібліотеки*

```
@UseBefore(ClassLevelMiddleware)
@Controller("/users")
export class UserController {
  @UseBefore(MethodLevelMiddleware)
  @Get("/:id")
  getUser(@Param("id") id: number): UserModel {
    console.log(`Getting user by ${id} id`);
    return {
      id,
      name: "Test Name",
      age: -1,
    };
  }

  @UseAfter(MethodLevelMiddleware)
  @Post()
  createUser(@Body dto: CreateUserDto, @Header sus:
any) {
    console.log("Received dto: ", dto);
    return -1;
  }
}

function main() {
  const config: AppConfig = {
    cors: true,
    globalPrefix: "api",
    controllers: [UserController],
  };

  const appContainer = new AppContainer();
  appContainer.build(config);

  appContainer.listen(8000, () => {
    console.log(`[server]: Server is running at
http://localhost:${8000}`);
  });
}

main();
```

4.4 Висновки з розділу 4

В цьому розділі були розглянуті основні етапи підготовки до реалізації поставленої задачі та зроблені наступні висновки:

1. Створена бібліотека компонентів для розробки додатків в об’єктно-орієнтованому стилі;

2. Розроблена імплементація DI контейнера для створеної бібліотеки;

3. Показана потенційна можливість та практична цінність використання бібліотеки при створенні сучасних додатків.

ВИСНОВКИ

1. В кваліфікаційній роботі розглянута проблематика побудови бібліотеки компонентів для розробки веб-застосунків.
2. Були розглянуті типи архітектур в розробці веб-застосунків. Була показана доцільність використання інверсії контролю.
3. Були розглянуті сучасні методи комунікації у веб додатках, та розібрані приклади коли треба застосовувати той чи інакший тип комунікації.
4. Була досліджена рефлексія та причина та доцільність її використання при розробленні веб-застосунків.
5. Була досліджена мікросервісна архітектура та були розглянуті її переваги та недоліки.
6. Були досліджені структурні аспекти бібліотеки компонентів та використання шаблонів проектування в компонентах.
7. Була розроблена бібліотека компонентів для побудови веб-застосунків і доведена доцільність її використання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Кириченко В.І., магістрант, Попівщий В. І., доцент — науковий керівник. Розробка об’єкто – орієнтованої бібліотеки для створення веб-додатків на основі JavaScript, TypeScript, Node.js та Express. ЗБІРНИК наукових праць студентів, аспірантів, докторантів і молодих вчених «МОЛОДА НАУКА-2023», Запоріжжя : Запорізький національний університет, 2023. С. 91-92.

2. Кириченко В.І., магістрант, Попівщий В. І., доцент — науковий керівник. Розробка об’єкто – орієнтованої бібліотеки для створення веб-додатків на основі JavaScript, TypeScript, Node.js та Express. III Всеукраїнської науково-практичної конференції за участю молодих науковців «АКТУАЛЬНІ ПИТАННЯ СТАЛОГО НАУКОВО-ТЕХНІЧНОГО ТА СОЦІАЛЬНО-ЕКОНОМІЧНОГО РОЗВИТКУ РЕГІОНІВ УКРАЇНИ». Запоріжжя : Запорізький національний університет, 2023. С. 151-152.

3. The Onion Architecture: part 1 // Jeffrey Palermo (.com). URL: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/ (дата звернення: 30.10.2023).

4. S. Somasegar, Scott Guthrie, David Hill. Microsoft Application Architecture Guide 2nd Edition, Patterns & Practices, Microsoft Press, pp. 53–144, 2009.

5. S. Somasegar, Scott Guthrie, David Hill. Microsoft Application Architecture Guide 2nd Edition, Patterns & Practices, Microsoft Press, pp. 21–43, 2009.

6. Steven van Deursen, Mark Seemann. Dependency Injection Principles, Practices, and Patterns, Manning, pp. 552, 2019.

7. Richardson, Leonard; Amundsen, Mike; Ruby, Sam. RESTful Web APIs p. 570, 2013

8. The WebSocket Protocol [Електронний ресурс]. – Режим доступу: https://datatracker.ietf.org/doc/html/rfc6455 Дата звернення: 28.10.2023

9. Jon Skeet. C# in Depth 4th Edition, Manning, pp. 528, 2019.

10. The Modern JavaScript Tutorial [Електронний ресурс]. – Режим доступу: https://javascript.info/ – Дата звернення: 29.10.2023.

11. TypeScript Documentation [Електронний ресурс]. – Режим доступу: https://www.typescriptlang.org/docs/ – Дата звернення: 29.10.2023.

12. Sam Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, O'Reilly Media Inc., pp. 263, 2020.

13. Dafydd Stuttard, Marcus Pinto, The Web Application Hacker's Handbook, Wiley Publishing Inc., pp. 729, 2008.

14. Bryan Sullivan, Vincent Liu, Web Application Security, A Beginner's Guide, McGraw Hill Professional, pp. 384, 2011

15. CRYPTO101 [Електронний ресурс]. – Режим доступу: https://www.crypto101.io/ – Дата звернення: 01.11.2023.

16. Jenifer TidwellCharles BrewerAynne Valencia-Brooks, Designing Interfaces: Patterns for Effective Interaction Design 3rd Edition, O'Reilly, pp. 500, 2021.

17. MartinFowler [Електронний ресурс]. – Режим доступу: https://refactoring.com/– Дата звернення: 27.10.2023.

18. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, pp. 464, 2009.

19. Martin Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems 1st Edition, O'Reilly, pp. 590, 2017.

20. Erich GammaRichard HelmRalph JohnsonJohn Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, pp. 396, 1995.

21. Andrew Hunt, David Thomas, The Pragmatic Programmer: Your Journey To Mastery, 20th Anniversary Edition (2nd Edition), Print2print, pp. 352, 2020.

22. Addy Osmani, Learning JavaScript Design Patterns: A JavaScript and React Developer's Guide 2nd Edition, O'Reilly, pp. 296, 2023.

23. Daniel Andres and Pelaez Lopez, Full-Stack Web Development with Jakarta EE and Vue.js. 1st Edition, pp. 592, 2021

24. Adam Freeman, Pro Angular 9: Build Powerful and Dynamic Web Apps 4th Edition, Apress, pp. 812, 2020.

25. Steve Mc Connell, Code Complete: A Practical Handbook of Software Construction, Second Edition, Microsoft Press, pp. 916, 2004.

26. Roy Osherove, The Art of Unit Testing: with examples in C#, Manning Publications Co., pp. 296, 2013.

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Кириченко Владислав Ігорович, студент _2_ курсу форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти se22m-07@stu.zsea.edu.ua

- підтверджую, що написана мною кваліфікаційна робота на тему **«Особливості побудови бібліотеки компонентів для створення інформаційних систем»** відповідає нормам академічної доброчесності та не містить порушень, що визначені у ст. 42 Закону України «Про освіту», зі змістом яких ознайомлений;

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь який спосіб, у тому числі за допомогою інтернет-систем, а також на архівування моєї роботи в базі даних цієї системи.

Дата 20.02.2024 Підпис _____ Кириченко Владислав Ігорович
(студент)

Дата 20.02.2024 Підпис _____ Попівщий Василь Іванович
(науковий керівник)