Міністерство освіти і науки України

Запорізька державна інженерна академія
(повне найменування вищого навчального закладу)

Факультет енергетики, електроніки і інформаційних технологій
(назва факультету)

Кафедра програмного забезпечення автоматизованих систем
(повна назва кафедри)

Пояснювальна записка

до магістерської роботи

рівень вищої освіти другий (магістерський)
(другий (магістерський) рівень)

на тему **Дослідження та аналіз особливостей реалізації компонентного підходу в сучаснасних JS-фреймворках та бібліотеках**

Виконав: студент 2 курсу, групи ІПЗ-16-1мд
Утьонишев Владислав Андрійович
(ПІБ) (підпис)
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва)
Спеціалізація
(шифр і назва)
освітньо-професійна програма
Інженерія програмного забезпечення
(шифр і назва)

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

Запоріжжя – 2018 року

2

Запорізька державна інженерна академія
(повне найменування вищого навчального закладу)

Факультет енергетики, електроніки і інформаційних технологій
Кафедра програмного забезпечення автоматизованих систем
Рівень вищої освіти другий (магістерський)
(другий (магістерський) рівень)
Спеціальність 121 Інженерія програмного забезпечення
(шифр і назва)
Спеціалізація _____
(шифр і назва)
Освітньо-професійна програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ В.Г.Вербицький
"_04_" _вересня_____2017_року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Утьонишев Владислав Андрійович
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження та аналіз особливостей реалізації компонентного підходу в сучаснасних JS-фреймворках та бібліотеках
керівник роботи Попівщий Василь Іванович, к.ф. –м.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "_04_" _вересня_2017_року_№_352-01__

2. Строк подання студентом роботи 09.01.2018 р.
3. Вихідні дані до роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми дипломної роботи;
- дослідження предметної області;
- створення програмного продукту та його опис;
- перелік вимог в роботі з програмою та системою;
- реалізація та тестування створеного продукту.

3

5. Перелік графічного матеріалу 20 слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата завдання прийняв
Нормоконтроль	Полякова Н.П., доц.каф.ПЗАС	29.12.17

7. Дата видачі завдання 04.09.2017 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної Роботи	Строк виконання етапів роботи	Примітка
1.	Ознайомлення з предметною областю.	01.09-07.09.17	виконано
2.	Ознайомлення з технічним завданням.	08.09-10.09.17	виконано
3.	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником.	11.09-13.09.17	виконано
4.	Ознайомлення з існуючими методами та підходами, що займаються вирішенням подібних задач.	14.09-18.09.17	виконано
5.	Початкова реалізація програмного проекту.	19.09-27.09.17	виконано
6.	Попередній аналіз результатів та уточнення задачі у відповідності з дослідженням	28.09-10.10.17	
7.	Порівняння існуючих технологій.	11.10-30.10.17	виконано
8.	Доопрацювання проекту та виправлення існуючих помилок.	31.10-05.11.17	виконано
9.	Узгодження та звітування перед науковим керівником про отриманні результати.	06.11-16.11.17	виконано
10.	Розробка інтерфейсу для візуалізації отриманих результатів.	17.11-22.11.17	виконано
11.	Представлення отриманих результатів та узгодження подальшого плану робіт із науковим керівником.	23.11-30.11.17	виконано
12.	Розширення інтерфейсу та подальше тестування.	01.12-05.12.17	виконано
13.	Доопрацювання отриманих зауважень.	06.12-10.12.17	виконано
14.	Оформлення звіту та документації.	11.12-23.12.17	виконано
15.	Представлення розробленого програмного продукту науковому керівникові.	24.12-08.01.18	виконано

Студент _____ **В.А. Утьонишев**
(підпис) (прізвище та ініціали)

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

РЕФЕРАТ

Сторінок: 110

Рисунків: 13

Таблиць: 1

Джерел: 21

Мета роботи: Мета дослідження полягає у вивченні повторного використання компонентів у поточному застосунку або при розробці нових застосунків, що дозволяє уникати написання однакового коду та зберігати час.

Результати: Досліджено поняття компонентного підходу та історію його розвитку, оглянуті сучасні JavaScript фреймворки та бібліотеки, визначені їх переваги та недоліки, на основі дослідження для розробки тестового застосунку обрано фреймворк Angular 5.0 з найкращими характеристиками для впровадження компонентного підходу. Спроектовано та розроблено застосунок для корпоративного використання – Task Manager, який побудований на основі компонентного підходу та повторного використання компонентів.

Публікації: Публікації: В.А. Утьонишев, магістрант, В.І. Попівщий, доцент. Дослідження та аналіз особливостей реалізації компонентного підходу у сучасних JS фреймворках та бібліотеках. / В.А. Утьонишев, В.І. Попівщий. // Матеріали XXII науково-технічної конференції студентів, магістрантів, аспірантів і викладачів ЗДІА 2017. Енергетика, Електроніка та Інформаційні технології. Том III. – Запоріжжя: Видавництво ЗДІА, 2017. – с. 144-145.

КОМПОНЕНТНИЙ ПІДХІД, JAVASCRIPT ФРЕЙМВОРКИ ТА БІБЛІОТЕКИ, ВЕБ, ANGULAR

ABSTRACT

Pages: 110

Figures: 13

Tables: 1

References: 21

Thesis goal: The purpose of the study is to examine the reuse of the components in our application or in the development future applications, which avoids the ability to write the same code and save time of development.

Results: Was researched component approach and its development in history and in the web, examined Javascript framework and libraries, their advantages and disadvantages were determined, and on the basis of this was chosen Angular 5.0 which is best suited for the development of the test application and will show using component approach. The application for the corporate Pick-up Task Manager, built on the basis of the component approach, and their re-use, has been designed and developed.

Publications: Публікації: В.А. Утьонишев, магістрант, В.І. Попівщий, доцент. Дослідження та аналіз особливостей реалізації компонентного підходу у сучасних JS фреймворках та бібліотеках. / В.А. Утьонишев, В.І. Попівщий. // Матеріали XXII науково-технічної конференції студентів, магістрантів, аспірантів і викладачів ЗДІА 2017. Енергетика, Електроніка та Інформаційні технології. Том III. – Запоріжжя: Видавництво ЗДІА, 2017. – с. 144-145.

COMPONENT APPROACH , JAVASCRIPT FRAMEWORK AND LIBRARY, WEB, ANGULAR

РЕФЕРАТ

Страниц: 110

Рисунков: 13

Таблиц: 1

Источников: 21

Цель работы: Цель исследования заключается в изучении повторного использовании компонентов в текущем приложении или при разработке будущих приложений, что позволяет избегать написания одинакового кода и уменьшить время на разработку.

Результаты: Исследовано понятие компонентного подхода и история его развития, осмотрены современные JavaScript фреймворки и библиотеки, определены их преимущества и недостатки, и на основе этого был выбран Angular 5.0, который как наилучше подходит для разработки тестового приложения и покажет использование компонентного подхода. Спроектировано и разработано приложение для корпоративного использования – Task Manager, который построен на основе компонентного подхода, и повторном использовании компонентов.

Публикации: Публікації: В.А. Утьонишев, магістрант, В.І. Попівщий, доцент. Дослідження та аналіз особливостей реалізації компонентного підходу у сучасних JS фреймворках та бібліотеках. / В.А. Утьонишев, В.І. Попівщий. // Матеріали XXII науково-технічної конференції студентів, магістрантів, аспірантів і викладачів ЗДІА 2017. Енергетика, Електроніка та Інформаційні технології. Том III. – Запоріжжя: Видавництво ЗДІА, 2017. – с. 144-145.

КОМПОНЕНТНЫЙ ПОДХОД, JAVASCRIPT ФРЕЙМВОРКИ И БИБЛИОТЕКИ, ВЕБ, ANGULAR

ЗМІСТ

ВСТУП

Актуальність теми

Технічний прогрес не стоїть на місці. Зараз семимильними кроками розвивається всесвітня мережа Internet. З кожним роком кількість веб-сайтів збільшується, та вони стають все масштабнішими, тому над кожним із них працюють команди та навіть декілька команд при розробці, такі веб-сайти підтримуються понад декілька років. Тому компонентно-орієнтований підхід, який спирається на незалежні модулі вихідного коду, призначеного для повторного використання, для таких задач якнайкраще підходить. Такий підхід збільшить надійність застосунку, дозволе краще читати код інших розробників та уникнути ситуацій, коли створюється однаковий функціонал у декількох шматках коду, що дозволить зберегти багато часу при розробці та підтримці сайтів.

Мета і завдання дослідження

Мета дослідження полягає у виявленні найбільш перспективних підходів в розробці веб компонентів, створення та дослідження компонентів повторного використання на прикладі тестового веб-додатку.

Об’єкт дослідження

Об’єктом дослідження в даній роботі є JS-фреймворки та JS-бібліотеки, в яких втілений компонентний підхід.

Предмет дослідження

Способи реалізації компонентів в різних фреймворках.

Методи дослідження

Для розв’язання поставлених завдань використовуються такі методи дослідження:

- аналіз літератури та документації компонентного підходу;
- знаходження найбільш спірних та складних моментів у реалізації таких систем;
- огляд та порівняння сучасних JavaScript-фреймворків та бібліотек
- методи оцінки готового програмного продукту.

Наукова новизна одержаних результатів

Проблема з використанням компонентного підходу у веб-застосунках встає дуже давно, насамперед у веб-додатках, над якими працює декілька людей і треба розпаралелювати завдання. Тож створено тестовий застосунок, який використовує новітній JS-фреймворк, що дозволяє використовувати якнайкраще компонентний підхід. Крім цього були створені компоненти, які можуть використовуватись у інших веб-додатках, що полегшує розробку.

Практичне значення одержаних результатів

Практичним результатом є створення веб додатку Task Manager. Даний веб додаток розроблений для корпоративного застосування, для реєстрації користувачів, які мають можливість створювати завдання з drag-and-drop інтерфейсом, та повністю редагованою таблицею для них. Для введення даних та інших операцій запропоновані відповідні компоненти.

Глосарій

Application Programming Interface (API) — це зазвичай (але не обов'язково) метод абстракції між низькорівневим та високорівневим програмним забезпеченням.

Back-end — програмно-адміністративна частина застосунку.

Front-end — це інтерфейс для взаємодії між користувачем і back-end. Front-end та back-end можуть бути розподілені між однією або кількома системами.

JavaScript (JS) — динамічна, об'єктно-орієнтована мова програмування. Реалізація стандарту ECMAScript.

Алгоритм (*Algorithmi*) — набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв'язання задачі за скінченну кількість дій; система правил виконання дискретного процесу, яка досягає поставленої мети за скінченний час.

Архітектурні шаблони програмного забезпечення (*Software architectural patterns*) — це шаблони програмного забезпечення, що являють собою звіт «добрих практик» вирішення архітектурних проблем розробки програмного забезпечення.

Бібліотека в програмуванні (*library*) — збірка підпрограм або об’єктів, що використовуються для розробки програмного забезпечення.

Браузер, також *веб-переглядач* (*browser*) — програмне забезпечення для комп'ютера або іншого електронного пристрою, як правило, під'єднаного до Інтернету, що дає можливість користувачеві взаємодіяти з текстом, малюнками або іншою інформацією на гіпертекстовій веб-сторінці.

Веб-застосунок — розподілений застосунок, в якому клієнтом виступає браузер, а сервером — веб-сервер. Браузер може бути реалізацією так званих тонких клієнтів — логіка застосунку зосереджується на сервері, а функція брау-

зера полягає переважно у відображенні інформації, завантаженої мережею з сервера, і передачі назад даних користувача.

Модель-вигляд-контролер або Модель-вид-контролер (*MVC, Model-view-controller*) — архітектурний шаблон, який використовується під час проектування та розробки програмного забезпечення.

Препроцесор (*preprocessor*) — програма, яка виконує попередню обробку даних, для того, щоб вони могли використовуватись іншою програмою, наприклад, такою як компілятор.

Фреймворк (*Framework*) — інфраструктура програмних рішень, що полегшує розробку складних систем. Спрощено дану інфраструктуру можна вважати своєрідною комплексною бібліотекою.

Single Page Application(SPA) — Односторінковий додаток ,також відомий як одно сторінковий інтерфейс (англ. single-page interface, SPI) - це веб-застосунок чи веб-сайт, який вміщується на одній сторінці з метою забезпечити користувачу досвід близький до користування настільною програмою. В одно сторінковому додатку весь необхідний код - HTML, JavaScript, та CSS - завантажується разом із сторінкою , або динамічно довантажується за потребою, зазвичай у відповідь на дії користувача. Сторінка не оновлюється і не перенаправляє користувача на іншу сторінку в процесі роботи з нею. Взаємодія з одно сторінковим додатком часто включає в себе динамічний зв'язок з веб-сервером.

Компонентно-орієнтоване програмування (КОП) - парадигма програмування, яка спирається на поняття компонента – як незалежного модуля вихідного коду програми, призначеного для повторного використання і розгортання та реалізується у вигляді безлічі мовних конструкцій, об'єднаних за спільною ознакою і організованих відповідно до визначених правилами і обмеженнями.

РОЗДІЛ 1 ОГЛЯД КОМПОНЕНТНОГО ПІДХОДУ В СУЧАСНИХ JAVASCRIPT ФРЕЙМВОРКАХ ТА БІБЛІОТЕКАХ

1.1 Загальні відомості про компонентний підхід

Компонентно-орієнтоване програмування - парадигма програмування, яка спирається на поняття компонента – як незалежного модуля вихідного коду програми, призначеного для повторного використання і розгортання та реалізується у вигляді безлічі мовних конструкцій, об'єднаних за спільною ознакою і організованих відповідно до визначених правилами і обмеженнями [1].

Не дивлячись на свою відносну молодість - компонентно-орієнтоване програмування має свої особливості, які регулюють не тільки особливості мови, та і всієї екосистеми КОП. До таких особливостям слід віднести:

- Чітко виражену орієнтованість на модулі. Модуль є основною структурною одиницею.
- Роздільна компіляція модулів. Це призводить до заощадження обчислювальних і часових ресурсів.
- Сувора типізація, як всередині модуля, так і між модулями. Забезпечує надійну роботу компонентів в цілому.
- Неминучість динамічної збірки сміття. Для компільованих мов це важливий і незвичайний механізм.
- Суворе поділ частин модулів, призначених для взаємодії з іншими модулями, і приховані частини тільки для роботи всередині модуля.

Компонентно-орієнтоване програмування, в чомусь розширює, а в чомусь обмежує інші ідіоми програмування. Так, підтримується функціональний стиль, але обмежуються побічні ефекти [5]. Як такого в Компонентно-орієнтоване програмування немає поняття класу, але в той же час, структури даних, і прив'язані

до них процедури в рамках модуля - цілком описує поняття класу в ООП. Досить прості вихідні лексеми дозволяють будувати (і заохочують таку побудову) код, який добре узгодимо, сильно типізований, може бути розширений (як через вихідний код, так і через скомпільований в машинному вигляді).

Компонентно-орієнтоване програмування, перш за все, націлене на стійку динамічне середовище. Так, наприклад, можлива ситуація в ОС Linux, коли за допомогою спеціальних механізмів в ряді випадків можна внести виправлення прямо в працююче ядро, в компонентному Паскалі - динамічно замінити один працюючий модуль на інший не представляє складності (за винятком тих випадків, коли відбувається взаємодія з ресурсами, які Компонентний Паскаль контролювати, зі зрозумілих причин, - не може).

Також слід згадати про те, що в компільованих мовах автоматична прибирання сміття не така проста річ. У динамічних системах потреба в збирача сміття ще вище. Зазвичай, програміст сам повинен контролювати низько рівневі операції з пам'яттю. І тут порушується принцип абстракції. Використовуючи Компонентно-орієнтоване програмування можна побудувати систему, в якій прибирання сміття буде виконуватися автоматично і абсолютно точно. Це можна досягти кількома способами, і найдоступніший з них - підрахунок посилань на структуру даних. Якщо модуль, який обробляє будь-яку структуру вивантажено з пам'яті - очевидно, ці дані вже ніхто не зможе обробити).

1.2 Історія розвитку компонентного підходу

Об’єктно-орієнтоване програмування (ООП) виникло в результаті розвитку ідеології процедурного програмування, де дані і підпрограми (процедури, функції) їх обробки формально не пов'язані. Для подальшого розвитку об'єктно-орієнтованого програмування часто велике значення мають поняття події (так

зване подієво-орієнтоване програмування) і компонента (компонентне програмування, КОП).

Формування КОП від ООП сталося, коли виникло формування модульного від процедурного програмування: процедури сформувалися в модулі - незалежні частини коду до рівня збірки програми, так об'єкти сформувалися в компоненти - незалежні частини коду до рівня виконання програми. Взаємодія об'єктів відбувається за допомогою повідомлень. Результатом подальшого розвитку ООП, мабуть, буде агентно-орієнтоване програмування, де агенти - незалежні частини коду на рівні виконання. Взаємодія агентів відбувається за допомогою зміни середовища, в якій вони знаходяться.

Мовні конструкції, конструктивно не відносяться безпосередньо до об'єктів, але супутні їм для їх безпечної (виняткові ситуації, перевірки) і ефективної роботи інкапсулюються від них в аспекти (в аспектно-орієнтованому програмуванні) [1]. Предметно-орієнтоване програмування розширює поняття об'єкт за допомогою забезпечення більш уніфікованого і незалежного взаємодії об'єктів.

В даний час кількість прикладних мов програмування (список мов), що реалізують об'єктно-орієнтовану парадигму, є найбільшим по відношенню до інших парадигм. В області системного програмування до сих пір застосовується парадигма процедурного програмування, і загальноприйнятою мовою програмування є мова С. Хоча при взаємодії системного і прикладного рівнів операційних систем помітний вплив стали надавати мови об'єктно-орієнтованого програмування. Наприклад, однією з найбільш поширених бібліотек мультиплатформенного програмування є об'єктно-орієнтована бібліотека Qt, написана на мові C++.

Компонентно-орієнтований підхід з'явився в 1987 році , коли Ніклаус Вірт [1] запропонував для мови «Оберон» патерн написання блоків. Даний патерн сформувався при вивченні проблеми «тендітних» базових класів, що виникає при побудові об'ємної ієрархії класів. Патерн полягав в тому, що компонент

компілюється окремо від інших, а на стадії виконання - необхідні компоненти підключаються динамічно.

У 1989 році - Бертран Мейер запропонував ідею єдиного взаємодії між викликається і викликає компонентами. Ця ідея втілилася у вигляді готових рішень: CORBA, COM, SOAP. Згодом, підтримка з боку мови здійснилася в «компонентний Паскалі».

Ситуація зі впровадженням компонентно-орієнтованого підходу, - як обмеження для існуючих парадигм програмування, - подібна до появою структурного програмування, яке обмежувало невпорядковані переходи управління за допомогою оператора «GOTO» (утрудняє аналіз алгоритму програми для вже існуючих мов і не привносить нових конструкцій).

1.3 Розвиток компонентів у Вебі

Сучасні веб-додатки настільки ж складні, як і будь-які інші програмні додатки, і часто створюються декількома людьми, об'єднують зусилля для створення фінального продукту. В таких умовах, щоб підвищити ефективність, природно шукати правильні способи поділу роботи на дільниці з мінімальними перетинами між людьми і підсистемами. Впровадження компонентного підходу (в цілому) - це те, як зазвичай вирішується таке завдання. Будь-яка компонентна система повинна зменшувати загальну складність через надання ізоляції, або природних бар'єрів, що приховують складність одних систем від інших. Хороша ізоляція також полегшує повторне використання та впровадження сервісних парадигм [2].

Спочатку складність веб-додатків в основному регулювалася з боку сервера за рахунок розділення програми на окремі сторінки, що вимагало від користувача відповідним чином переходити в браузері з однієї сторінки на іншу. З впровадженням AJAX і пов'язаних технологій розробники змогли відмовитися

від потреби робити «переходи» між різними сторінками веб-додатки. Для типових сценаріїв на кшталт читання пошти або новин очікування користувачів змінилися. Наприклад, після логіна в пошту, ви можете «користуватися поштовим додатком» з одного і того ж адреси (URL) і знаходиться на цій сторінці цілий день (т.зв. Single-Page Applications, SPA). Логіка клієнтських веб-додатків в таких ситуаціях істотно ускладнюється, іноді вона навіть стає складніше, ніж на серверній стороні. Можливим вирішенням даної складності може бути подальший поділ на компоненти і ізоляція логіки всередині однієї сторінки або документа.

Мета веб-компонентів в зменшенні складності за рахунок ізоляції пов'язаних груп коду на HTML, CSS і JavaScript для виконання загальної функціональності в межах контексту однієї сторінки.

Так як веб-компоненти повинні зв'язати воєдино HTML, CSS і JavaScript, необхідно враховувати існуючі моделі ізоляції, притаманні кожній з технологій, так як вони впливають на сценарії і цілісність веб-компонентів [4]. Ці незалежні моделі ізоляції включають:

- Ізоляція стилів в CSS.
- JavaScript і області видимості (замикання).
- Ізоляція глобального об'єкта.
- Інкапсуляція елементів (iframe).

Для ізоляції стилів CSS в рамках сьогоднішньої платформи не існує ідеального і природного способу розбити їх на компоненти (хоча інструменти на зразок Sass можуть істотно допомогти). Компонентна модель повинна пропонувати механізм для ізоляції однієї підмножини CSS від іншої так, що правила не будуть впливати один на одного. До того ж, стилі компонента повинні застосовуватися тільки до безпосередніх частинах компонента і більш ні до чого іншого.

Всередині таблиці стилів CSS-правила застосовуються до документу, використовуючи селектори. Селектори завжди розглядаються як потенційно прийнятні до всього документа, тому їх область застосування, по суті, глобальна. Глобальне застосування призводить до реальних конфліктів, коли кілька людей, які працюють над проектом, змішують разом свої CSS-файли. З перетинаннями і повтореннями селекторів можна боротися в чіткому порядку (наприклад, каскади, специфічність, порядок проходження ісходних кодів) для вирішення конфліктів, однак, такі дії, цілком ймовірно, - зовсім не те, чого хотіли розробники. Є багато потенційних способів вирішення цієї проблеми. Просте рішення - перенести елементи і пов'язані стилі, які беруть участь у формуванні компонента з основного документа в інший документ (тіньовий документ) так, що вони більше не будуть «реагувати» на чужі селектори. Це призводить до другої проблеми: тепер, коли ми їх розмежували, як деякий стиль може перетнути кордон (для управління зовні компонента)? Очевидне можливе рішення - це явно використовувати JavaScript, але це виглядає якось жахливо: покладатися на JavaScript для передачі стилів через кордон, що здається швидше прогалиною в CSS.

Щоб передати стилі через кордон компонента ефективним чином і при цьому захистити структуру компонента (наприклад, дозволити свободу зміни структури без впливу стилів), існує два загальних підходи, до яких багато хто схиляється: «часткова» стилізація з використанням псевдо-елементів та кастомні властивості (раніше відомі як «змінні» CSS). Якийсь час також розглядався супер-потужний крос-граничний селектор '>>>' (визначений в CSS Scoping), але сьогодні він загальновизнаний не найвдалішою ідеєю, так як легко порушує ізоляцію компонентів.

Часткова стилізація дозволить авторам компонента створювати власні псевдо-елементи для стилізації, таким чином, виставляючи назовні зовнішнього світу тільки частина своєї внутрішньої структури. Це схоже на модель, яку бра-

узери використовують для виставлення «частин» нативних елементів управління. Для цілісності даного сценарію авторам також знадобиться деякий спосіб обмеження набору стилів, які можуть застосовувати до псевдо-елементу. Додаткове дослідження цієї «часткової моделі», що базується на псевдо-елементах, може привести до появи зручних стилістичних примітивів, хоча опрацювання деталей ще потребують зусиль. Подальша робота над часткової моделлю також повинна раціоналізувати стилізацію рідних елементів управління в браузерах.

Кастомні властивості дозволять авторам описувати значення стилів, які вони хочуть повторно використовувати в таблицях стилів (визначаються як власні імена властивостей з подвійним тире в якості префікса). Кастомні властивості успадковуються через під-дерево документа, дозволяючи селекторам перевизначать значення кастомними властивості для конкретного під-дерева без порушення інших піддерев. Кастомні властивості також зможуть успадковуватися через кордони компонентів, надаючи елегантний механізм стилізації компонентів, який при цьому уникає розкриття внутрішньої структурної природи компонента. Кастомниє властивості оцінювалися при розробці різних компонентних фреймворків в Google і, за звітами, дозволяють покрити більшість потреб стилізації.

Для повноти картини, області видимості і ізоляція CSS - це не така чорно-біла область, як могло здатися. Насправді, кілька минулих і поточних підходів пропонують варіанти обмеження сфери застосування та ізоляції з різною застосовністю до веб-компонентів.

CSS пропонує деякі обмежені форми ізоляції селектор в специфічних сценаріях. Наприклад, правило @media групує набір селекторів разом і застосовує їх при настанні умов, відповідних медіа-контексту (наприклад, розмір або дозвіл вьюпорту, або медіа-тип - друк і т.п.); правило @page визначає деякі стилі, які можна застосовувати лише в контексті друку; правило @supports об'єднує разом селектори для застосування тільки, коли реалізована підтримка специфіч-

ною *CSS-функціональності* - нова форма визначення наявності функціональності в CSS); запропоноване правило @document групує селектори для застосування тільки тоді, коли документ, в якому завантажені стилі, відповідає умовам.

Області видимості в CSS (спочатку написані як частина роботи над веб-компонентами) пропонують спосіб обмежувати можливість застосування CSS-селекторів всередині одного HTML-документа. Специфікація вводить нове правило @scope, яке дозволяє селектору визначити корінь області застосування і далі призводить до того, що застосування всіх селекторів всередині правила @scope буде працювати тільки в поддереве цього кореня (а не на всьому документі). Специфікація дозволяє вказувати корінь області декларативно в HTML (наприклад, запропонований <style scoped> атрибут, поки реалізований тільки в Firefox; ця функціональність раніше була доступна в Chrome в якості експериментальної, але після була цілком вилучена). Деякі аспекти цієї функціональності (наприклад,: scope, визначений у Selectors L4) також можуть застосовуватися для відносної оцінки селектор в новому API запитів в специфікації DOM.

Тут важливо зазначити, що @scope встановлює тільки одне-спрямовану ізоляцію кордонів: селектори, що містяться всередині @scope, обмежені цією областю, в той час як будь-які інші селектори (поза @ scope) можуть спокійною проникати всередині @scope (хоча вони можуть бути по різному впорядковані каскадом стилів). Це кілька невдалий дизайн, так як він не надає обмеження області та ізоляції від будь-яких стилів, які не перебувають в підмножині @scope - весь CSS повинен як і раніше «добре стикуватися», щоб уникнути стилізації всередині чужого правила @scope.

Інша пропозиція обмеження видимості - це стримування в CSS. Стримування області видимості - це в меншій мірі про ізоляцію стилів і селектор і в більшій про ізоляцію «композиції». Усередині властивості «contain» поведінку деяких можливостей CSS, які мають природне успадкування (в сенсі застосовності від батьківського до дочірньому елементу в документі, наприклад, лічиль-

ники) буде блоковано. Основне застосування цього для розробників полягає в тому, щоб вказати, що деякі елементи передбачають суворе «стримування», так що композиція, застосовна до цього елементу і його піддерево ніколи не буде зачіпати композицію інших елементів документа. Ці обіцянки стримування дозволяють браузерам оптимізувати композицію і отрисовку так, що «нова» композиція утримуваного піддерева потребує тільки поновлення цього піддерева, а не всього документа.

Увесь JavaScript-код, який включений на сторінку, має доступ до одного і того ж глобальному об'єкту. Як і інші мови програмування, JavaScript має області видимості, які надають певний рівень «приватності» для коду функції. Ці лексичні області видимості використовуються для ізоляції змінних і функцій від решти глобального оточення. «Модульний шаблон» в JavaScript, популярний сьогодні (використовує лексичні області видимості), еволюціонував з потреби безлічі фреймворків на JavaScript «співіснувати» в єдиному глобальному вимірі без того, щоб «наступати на п'яти» один одному (залежачи при цьому від порядку завантаження).

Лексичні області видимості в JavaScript - це односпрямована ізоляція кордонів: код всередині області може мати доступ як до внутрішнього вмісту, так і до вмісту будь-якої батьківської області аж до глобальної, в той час код зовні області не має доступу до її вмісту. Важливим принципом є те, що односпрямований спосіб ізоляції віддає перевагу коду всередині області, тобто захищає його. Код всередині лексичної області має можливість захищати / ховати себе від решти оточення (або не робити цього).

Внесок, який лексичні області видимості JavaScript вносять в реалізацію веб-компонента, відповідає вимозі мати спосіб «закриття» компонента так, що її вміст може бути розумно приватним.

Для деякого коду може бути небажаним, щоб він мав загальний доступ до глобального оточення, як це було описано вище. Наприклад, розробник програ-

ми може не довіряти якомусь коду на JavaScript, хоча він і надає істотну цінність. Характерний випадок - реклама і рекламні фреймворки. З міркувань безпеки, необхідно, щоб не довірений код виконувався в окремому чистому скриптовій оточенні (зі своїм власним глобальним об'єктом). Щоб досягти такої поведінки сьогодні (без включення в гру iframe елементів), розробники можуть використовувати Воркер. Втім, недолік цього рішення в тому, що Воркер не мають доступу до елементів, тобто UI.

Є ряд міркувань, які потрібно враховувати при проектуванні компонентів з підтримкою ізоляції глобального об'єкта - особливо якщо ізоляція буде мати на увазі захищені кордони (докладніше нижче). На сьогодні ми очікуємо, що ізольовані компоненти не будуть повністю доступні до тих пір, поки базовий набір специфікацій веб-компонентів не буде зафіксовано (тобто, це «відкладено до наступної версії»). Однак, якщо ми витратимо деякий час на дослідження того, як ізольовані компоненти можуть виглядати, це може направити в правильне русло поточну роботу. На деякі пропозиції дійсно варто звернути увагу.

Ізоляція глобального об'єкта - це важливий нереалізований сценарій для веб-компонентів. А поки ми працюємо над реалізацією, можна, наприклад, покладатися на найуспішніший і поширений на сьогодні спосіб привнесення компонентності в веб: iframe-елемент.

Iframe-елементи і їхні близькі родичі: елементи object, frameset і імперативний API windows.open () - вже надають можливість працювати в ізольованим піддерево елементів. Однак, якщо компоненти мають на увазі роботу всередині одного документа, iframe включає в собі цілий HTML-документ; як якщо б два окремих веб-додатки були розміщені спільно, просто одне всередині іншого. У кожного унікальні адресу документа, глобальне оточення для скриптів і область видимості CSS; кожен документ повністю відділений від іншого.

Iframe - це на сьогодні найуспішніша (і єдина широко впроваджена) форма компонентізаціі в інтернеті. Iframe дозволяє здійснювати взаємодію різних

веб-додатків. Наприклад, багато сайтів використовують iframe саме як форму компонента для всіляких сценаріїв від реклами до логіна користувачів. Однак iframe зіткнувся з низкою викликів та з'явилися деякі способи з цими викликами боротися [3]:

- JavaScript-код всередині одне HTML документа може потенційно вторгнутися в межі ізоляції іншого документи (наприклад, через властивість contentWindows у iframe елемента). Така можливість порушення кордону може бути необхідною потребою, але вона також представляє собою ризик з точки зору безпеки, коли вміст iframe містить чутливу інформацію, якої не хотілося б ділитися. Сьогодні небажані порушення можуть регулюватися політиками загального джерела: документи з URL з одного джерела можуть порушувати кордону за замовчуванням, в той час, як документи з різних джерел мають обмежені можливості взаємодії один з одним.
- Порушення кордону - це не єдиний ризик безпеки. Використання атрибута <iframe sandbox> накладає подальші обмеження на iframe з інших джерел з тим, щоб захистити хост-оточення від небажаних скриптів, спливаючих вікон, зміни навігації та інших можливостей, доступних в iframe.
- CSS-стилі зовнішнього документи не можуть застосовуватися до внутрішнього документу. Таке архітектурне рішення слід принципом ізоляції. Однак ізоляція стилів створює суттєву прогалину в інтеграції iframe як компонент (в рамках загального джерела походження). HTML адресує цю проблему за допомогою запропонованого атрибута <iframe seamless> для iframe з загальним джерелом. Такий атрибут «безшовні» видаляє ізоляцію стилів контенту фрейму; бесшовно включені документи беруть копію стилів хостового документа і відображаються, як якщо б їх обмежень iframe-елемента, в який вони включені, не було.

З хорошими політиками безпеки і можливість вставляти фрейм безшовна, використання iframe в якості моделі для компонентів здається вельми привабливим рішенням. Однак декількох властивостей бажаних в моделі веб-компонентів все ж не вистачає:

1. Глибока інтеграція. Iframe обмежує (і в основному повністю відключає) інтеграцію і взаємодію моделей між хостом і фреймовим документом. Наприклад, щодо хоста: фокус і модель виділення незалежна, а передача подій ізольована одним або іншим документом. Для компонентів, які передбачають більш близьку інтеграцію, підтримка такого поведінки не можлива без впровадження деякого «агента» в хостової документі, який би прокидаємо відомості через кордон.

2. Розмноження глобальних об'єктів. Для кожної сутності iframe, створеної на сторінці, буде присутній унікальний глобальний об'єкт. Глобальний об'єкт і пов'язана з ним повна система типів не дешева в створенні і може привести до споживання великої кількості пам'яті і надмірного навантаження на браузер. Множинні копії одного і того ж компонента, що використовуються на одній сторінці, не обов'язково повинні бути ізольовані один від одного, на практиці наявність загального глобального об'єкта може бути бажано, особливо, якщо вони повинні підтримувати деякий загальний стан.

3. Модель використання контенту хоста. Iframe не дозволяє повторного використання тематичної моделі хост-елемента всередині фреймової документа. (Для простоти: тематична модель елемента - це його підтримуване піддерево елементів і тексту.) Наприклад, select-елементи має контентну модель, що включає option-елементи. Select-елемент, реалізований у вигляді компонента, захоче деяким чином взаємодіяти з дочірніми елементами.

4. Вибіркова стилізація. Безшовний iframe не працює з документами з різних джерел. Є явні ризики безпеки, якби це було дозволено. Основна проблема в тому, що «безшовність» контролюється хостом, а не фреймовим документом

(фреймовий документ хащами є жертвою атак). Для компонента двозначна можливість включення «безшовні» (є чи ні) може бути занадто дорогою; компоненти швидше хотіли б вибірково приймати рішення, які стилі від хоста застосовні до їх вмісту (замість автоматичного успадкування всіх стилів, тобто це те, як працює безшовність). В цілому, питання, що має стилізований, має вирішуватися самим компонентом.

5. Виставлення API. Багато сценаріїв для веб-компонентів на увазі створення повноцінних кастомних елементів з власними виставленим набором API, семантикою відображення і управлінням життєвим циклом. Використання iframe обмежує розробника до роботи в рамках API iframe, з усім його особливостями. Наприклад, ви не можете впливати на параметри самого iframe і його життєвий цикл.

1.4 Сучасні веб-компоненти

Використовуючи концепції, описані в XBL в якості стартової точки, монолітна компонентна система була розбита на колекцію будівельних блоків для компонентів. Ці будівельні блоки дозволили веб-розробникам експериментувати з окремими корисними можливостями до того, як спільне бачення для веб-компоненти буде повністю визначено. Компонентність самого підходу і розробка окремих корисних можливостей дозволили просунутися ближче до успіху. Практично кожен може знайти в веб-компонентах щось корисне для свого застосування.

Ця нова хвиля веб-компонентів привела до формування набору конкретних прикладів використання, що пояснюють, як існуючі вбудовані елементи працюють в рамках сьогоднішньої веб-платформи. У теорії, веб-компоненти дозволять розробникам прототипувати нові типи HTML елементів з тією ж точніс-

тю і характерними рисами, як і у нативних елементів (на практиці забезпечення доступності в HTML є сьогодні особливо важким у досягненні).

Ясно, що повний набір всіх технологій, необхідних для покриття всіх сценаріїв використання веб-компонентів, не буде реалізований в браузерах відразу. Розробники браузерів працюють разом, щоб узгодити базовий набір технологій, який можна Консистентне реалізувати перш, ніж рухатися до додаткових сценаріями.

Сучасне покоління веб-компонентів включає:

1. Кастомні елементи (Custom Elements). Кастомні елементи визначають точку розширення HTML-парсеру, щоб він міг розпізнавати імена нових «кастомних елементів» і надати їм автоматично необхідну об'єктну модель на JavaScript. Кастомні елементи не створюють кордонів компонентів, але зате надають браузеру спосіб приєднувати API і поведінки до авторських елементів [4]. Браузери, які не підтримують кастомні елементи, можуть їх симулювати (через поліфілії) з деякою точністю, використовуючи події і спостерігачі за змінами і підлаштовуючи прототип. Правильне планування і розуміння наслідків - це ключовий елемент наших майбутніх зустрічей.

2. Атрибут "*is*". Він захований всередині специфікації кастомних елементів, але надає критичну функціональність - можливість вказати, що вбудований елемент повинен отримати ім'я кастомного елементу і нові API. У звичайному випадку кастомний елемент має в основі якийсь загальний елемент; за допомогою "is" за основу може бути взяти нативний елемент (наприклад, <input is = "custom-input">). Хоча ця функціональність представляє відмінний спосіб успадкувати переваги від вбудованого рендеринга, доступності, парсинга і т.п., синтаксис цієї функціональності сприймається скоріше як хак, і є думка, що, можливо, примітиви для доступності та стілазація нативних елементів - це більш відповідний в довгостроковому плані шлях стандартизації.

3. Тіньовий DOM (Shadow DOM). Надає імперативний API для створення окремого дерева елемента, яке може бути приєднано (одноразово) до хост-елементу. Такі «тіньові» нащадки заміняють «реальних» нащадків при відображенні документа. Тіньовий DOM також надає механізм використання тематичної моделі хост-елементи, використовуючи нові slot-елементи (запропоновані нещодавно), вирішує завдання з цільовим елементом у подій і додає відкриті / закриті режими операцій (також недавно додані). Ця відносно тривіальна ідея має дивно велика кількість сторонніх ефектів у всьому, починаючи з моделі фокуса і виділення і закінчуючи композицією і поширення (для тіньового DOM всередині тіньових DOM).

4. Області видимості CSS визначають різні псевдо-елементи, релевантні стилізації тіньового DOM, включаючи: host, :: content (скоро, можливо, стане :: slot), і колишній ">>>" (пронизливий комбінатор тіньового DOM), який тепер офіційно дезавуював.

5. Template елемент. Він включений для повноти картини, ця функціональність раніше була частиною веб-компонентів, а зараз є частиною рекомендації HTML5. Шаблонний елемент вносить концепцію інертності (дочірні елементи шаблону не призводять до завантаження ресурсів, не реагують на призначений для користувача введення і т.п.). Це спосіб декларативно створити в HTML незв’язане піддерево елемента. Шаблон може використовуватися для різних завдань: від власне зразків шаблонів і зв'язування даних до надання контенту для тіньового DOM.

6. HTML Imports. Визначає декларативний синтаксис для «імпорту» (запит, отримання або парсинг) HTML в документ. Запити на імпорт (використовуючи link-елемент з rel="import") виконують скрипти імпортованого документа в контексті хостовой сторінки (таким чином, маючи доступ до того ж глобальному об'єкту і станом). HTML, JavaScript і CSS частини веб-компонента можуть, відповідно, бути завантажені одним використанням імпорту.

7. Кастомні властивості. Як описано детальніше вище, кастомні властивості, описані поза компонента і доступні для використання всередині компонента - це проста і зручна модель для стилізації компонентів сьогодні. З огляду на це, ми включили кастомні властивості в перше покоління технологій для веб-компонентів.

1.5 Шаблон проектування MVC

Model-View-Controller (MVC, «модель-представлення-контролер», «модель-вид-контролер») - схема використання декількох шаблонів проектування, за допомогою яких модель додатки, призначений для користувача інтерфейс і взаємодія з користувачем розділені на три окремих компонента таким чином , щоб модифікація одного з компонентів надавала мінімальний вплив на інші. Дана схема проектування часто використовується для побудови архітектурного каркаса, коли переходять від теорії до реалізації в конкретній предметній області. Основна мета використання цієї Концепція полягає в відділенні бізнес-логіки (моделі) від її візуалізації (уявлення, виду). За рахунок такого поділу підвищується можливість повторного використання. Найбільш корисне застосування даної концепції в тих випадках, коли користувач повинен бачити ті ж самі дані одночасно в різних контекстах і / або з різних точок зору. Зокрема, виконуються наступні завдання:

До однієї моделі можна приєднати кілька видів, при цьому не зачіпаючи реалізацію моделі. Наприклад, деякі дані можуть бути одночасно представлені у вигляді електронної таблиці, гістограми і кругової діаграми.

Не торкаючись реалізацію видів, можна змінити реакції на дії користувача (натискання мишею на кнопці, введення даних), для цього досить використовувати інший контролер.

Ряд розробників спеціалізується тільки в одній з областей: або розробляють графічний інтерфейс, або розробляють бізнес-логіку. Тому можливо добитися того, що програмісти, які займаються розробкою бізнес-логіки (моделі), взагалі не будуть інформовані про те, яке уявлення буде використовуватися.

Концепція MVC дозволяє розділити дані (модель), уявлення і обробку дій (вироблену контролером) користувача на три окремих компоненти:

Модель (англ. Model):

- Надає знання: дані та методи роботи з цими даними.
- Реагує на запити, змінюючи свій стан.
- Не містить інформації, як ці знання можна візуалізувати.

Подання, вид (англ. View) - відповідає за відображення інформації (візуалізацію). Часто в якості уявлення виступає форма (вікно) з графічними елементами;

Контролер (англ. Controller) - забезпечує зв'язок між користувачем і системою: контролює введення даних користувачем і використовує модель і уявлення для реалізації необхідної реакції.

Важливо відзначити, що як уявлення, так і контролер залежать від моделі; проте модель (активна) не залежить ні від уявлення, ні від контролера. Тим самим досягається призначення такого поділу: воно дозволяє будувати модель незалежно від візуального представлення, а також створювати кілька різних уявлень для однієї моделі.

Для реалізації схеми «Model-View-Controller» використовується досить велика кількість шаблонів проектування (в залежності від складності архітектурного рішення), основні з яких - «спостерігач», «стратегія», «компоновник»:

Найбільш типова реалізація - в якій вид відділений від моделі шляхом встановлення між ними протоколу взаємодії, що використовує «апарат подій» (позначення «подіями» певних ситуацій, що виникають в ході виконання програми, - і розсилка повідомлень про них всім тим, хто підписався на отримання)

: при кожному особливому зміні внутрішніх даних в моделі (позначеному як «подія»), вона сповіщає про нього ті залежні від неї уявлення, які передплатили такого оповіщення - і уявлення оновлюється. Так використовується шаблон «спостерігач»;

При обробці реакції користувача - уявлення вибирає, в залежності від потрібної реакції, потрібний контролер, який забезпечить ту чи іншу зв'язок з моделлю. Для цього використовується шаблон «стратегія», або замість цього може бути модифікація з використанням шаблону «команда»;

Для можливості однотипного поводження з підоб'екти складноскладеного ієрархічного виду - може використовуватися шаблон «компоновник». Крім того, можуть використовуватися і інші шаблони проектування - наприклад, «фабричний метод», який дозволить поставити за замовчуванням тип контролера для відповідного виду.

1.6 Фреймворк AngularJS

AngularJS це JavaScript-фреймворк з відкритим програмним кодом, який розробляє Google. Призначений для розробки одно сторінкових застосунків, що складаються з одної HTML сторінки з CSS і JavaScript [8]. Його мета — розширення браузерних застосунків на основі шаблону Модель-вид-контролер (MVC), а також спрощення їх тестування та розробки.

Фреймворк працює зі сторінкою HTML, що містить додаткові атрибути і пов'язує області вводу або виводу сторінки з моделлю, яка являє собою звичайні змінні JavaScript. Значення цих змінних задаються вручну або отримуються зі статичних або динамічних JSON-даних.

AngularJS спроектований з переконанням, що декларативне програмування найкраще пасує для побудови інтерфейсів користувача та опису програмних компонентів, в той час як імперативне програмування пасує для опису бізнес-

логіки. Фреймворк адаптує та розширює традиційний HTML, щоб забезпечити двосторонню прив'язку даних для динамічного контенту, що дозволяє автоматично синхронізувати модель та вид. У результаті AngularJS зменшує роль DOM-маніпуляцій з метою підвищення продуктивності і спрощення тестування.

У цьому є свої плюси й мінуси. Однак, оскільки плюси явно переважують мінуси, вибір вже зроблений. Тим більше, що на даний момент ніхто не сумнівається в можливостях Google створювати якісні інструменти розробки.

Фреймворк працює з HTML, що містить додаткові користувальницькі атрибути, які описуються директивами, і пов'язує введення або виведення області сторінки з моделлю, що представляє собою звичайні змінні JavaScript. Значення цих змінних задаються вручну або витягуються з статичних або динамічних JSON-даних.

Позиція AngularJS по роботі з даними та іншими інженерними концепціями відрізняється від таких фреймворков, як Backbone.js і Ember.js. Ми задовольняємося вже відомому нам HTML, а Angular самостійно його покращує. Angular обновляє DOM при будь-яких змінах моделі, які живуть у чистих об'єктах JavaScript з метою зв'язку з даними. Коли оновлюється модель, Angular оновлює об'єкти, які містять актуальну інформацію про стан програми.

Двостороння зв'язок даних одна із особливостей Angular, вона надає синхронізацію між шарами моделі і виду. Зміни моделі передаються в вид, а зміни виду автоматично відбиваються в моделі. Таким чином, модель стає актуальним джерелом даних про стан програми.

Angular використовує прості об'єкти JavaScript для синхронізації моделі і виду, в результаті чого оновлювати будь-який з них легко і приємно. Angular перетворює дані в JSON і найкраще спілкується методом REST. За допомогою такого підходу простіше будувати фронтенд-додатки, тому що весь стан додатки зберігається в браузері, а не передається з сервера по шматочках, і немає побоювання, що стан буде зіпсовано або втрачено [7].

Пов'язуємо ми ці значення через вираження Angular, які доступні у вигляді керуючих шаблонів. Також ми можемо пов'язувати моделі через атрибут під назвою ng-model. Angular використовує свої атрибути для різних API, які звертаються до ядра Angular.

У додатку на одну сторінку (SPA) або весь необхідний код (HTML, CSS та JavaScript) викликається за одне завантаження сторінки, або потрібні ресурси підключаються динамічно і додаються до сторінці в разі потреби, зазвичай у відповідь на дії користувача. Сторінку не перезавантажується під час роботи, не передає управління іншій сторінці, хоча сучасні технології з HTML5 дозволяють одному з додатків працювати на декількох логічних сторінках. Взаємодія з SPA часто відбувається за допомогою фонового спілкування з сервером.

У старіших додатках, коли стан програми зберігалося на сервері, траплялися відмінності між тим, що бачить користувач і тим, що зберігалося на сервері. Також відчувався брак стану програми в моделі, так як всі дані зберігалися в шаблонах HTML і динамічними не були. Сервер готував статичний темплейт, користувач вводив туди інформацію і браузер відправляв її назад, після чого відбувалася перезавантаження сторінки і Backend оновлював стан. Будь-який не збережений стан втрачався, і браузеру потрібно було викачувати всі дані після оновлення сторінок заново.

Часи змінилися, браузер зберігає стан додаток, складна логіка і фреймворки набули популярності. AngularJS зберігає стан в браузері і передає зміни при необхідності через Ajax (HTTP) з використанням методом GET, POST, PUT і DELETE. Краса в тому, що сервер може бути незалежний від Frontend, а Frontend - від сервера. Ті ж самі сервера можуть працювати з мобільними додатками з абсолютно іншим Frontend. Це дає нам гнучкість, так як на Backend ми працюємо з JSON-даними будь-яким зручним нам способом на будь-якому сервері.

У Angular є різні API, але структура програми зазвичай одна і та ж, тому майже всі програми будуються подібним чином і розробники можуть включатися в проект без зусиль. Також це дає передбачувані API і процеси налагодження, що зменшує час розробки і швидке прототипування. Angular побудований навколо можливості тестування («testability»), щоб бути найбільш простим як в розробці, так і в тестуванні.

Всі додатки створюються через модулі. Модуль може залежати від інших, або бути одиночним. Модулі служать контейнерами для різних розділів програми, таким чином роблячи код придатним для повторного використання. Для створення модуля застосовується глобальний Object, простір імен фреймворка, і метод module.

У додатку є один модуль app `angular.module('app', []);`. Другим аргументом йде [] - зазвичай цей масив містить залежності модуля, які нам потрібно підключити. Модулі можуть залежати від інших модулів, які в свою чергу теж можуть мати залежності. Для створення Controllers, Directives, Services і інших можливостей нам треба послатися на існуючий модуль.

Одне з основних понять в програмуванні - область видимості. У Angular область видимості - це один з головних об'єктів, який робить можливим цикли двостороннього зв'язку даних і зберігає стан додатка. $Scope - досить хитрий об'єкт, який не тільки має доступ до даних і значенням, але і надає ці дані в DOM, коли Angular рендерить наш додаток.

Уявіть, що $scope - це автоматичний міст між JavaScript і DOM, який зберігає синхронізовані дані. Це дозволяє простіше працювати з шаблонами, коли ми використовуємо при цьому синтаксис HTML, а Angular рендерить відповідні значення $scope. Це створює зв'язок між JavaScript і DOM. Загалом, $scope грає роль ViewModel. $Scope використовується тільки всередині Контролерів. Там ми прив'язуємо дані Контролера до Виду. Щоб це відобразилося в DOM, ми повинні приєднати Контролер до HTML і повідомити Angular, куди вставляти зна-

чення. $Scope використовується тільки всередині Контролерів. Там ми прив'язуємо дані Контролера до Виду.

Перед вами концепція області видимості Angular, що підкоряється деяким правилам JavaScript в плані лексичних областей видимості. Зовні елемента, до якого приєднаний Контролер, дані виходять за межі області видимості - так само, як змінна вийшла б за область видимості, якщо б ми послалися на неї зовні її області видимості.

Ми можемо прив'язати будь-які типи JavaScript $scope. Таким чином ми беремо дані від сервісу, спілкується з сервером, і передаємо їх у View, шар презентації. Чим більше ми створимо Контролерів і зв'язків з даними, тим більше з'являється областей видимості.

Контролер дозволяє взаємодіяти виду і моделі. Це те місце, де логіка презентації синхронізує інтерфейс з моделлю. Мета Контролера - приводити в дію зміни в Моделі і Виду. У контролер Angular зводить разом бізнес-логіку і логіку презентації. Ng-controller пов'язує область видимості і екземпляр контролера, і забезпечує доступ до даних і методів контролера з DOM. Контролер приймає два аргументи - ім'я, за яким на нього можна посилатися, і функцію зворотного виклику. При цьому насправді це буде функція, яка описувала тіло контролера.

Мета *контролера* - інтерпретувати бізнес-логіку моделі і перетворювати її в формат презентації. У Angular це можна робити по-різному, в залежності від того, які ми отримуємо дані. Контролер спілкується з сервісом, і передає дані в тому ж, або зміненому форматі в наш Вид через об'єкт $scope. Коли вид оновлен, логіка контролера також оновлюється, і її можна передавати назад на сервер через сервіс.

Контролери схожі на класи, але використання їх через об'єкти $scope не схоже на використання класів. Вони пропонували використовувати ключове слово this замість $scope. Розробники Angular ввели таку можливість в рамках синтаксиса controllerAs, де контролер оформляється у вигляді примірника, що

зберігається в змінної - приблизно так само, як використання new зі змінною для створення нового об'єкта. У кожного створюваного $scope тобто об'єкт $parent. Без використання controllerAs нам треба б було використовувати посилання на $parent для будь-яких методів з області видимості $parent, і $parent. $Parent для області на рівень вище, і так далі. Тепер же ми можемо просто використовувати ім'я змінної.

Сервіси дозволяють зберігати дані моделі і бізнес-логіку, наприклад, спілкування з сервером через HTTP. Часто плутають сервіси і фабрики - відмінність їх у тому, як створюються відповідні об'єкти. Важливо пам'ятати, що всі сервіси - сінглтони, на кожну ін'єкцію є тільки один сервіс. За угодою імена Сервісів даються в стилі Паскаля, тобто «my service» пишеться як «MyService». Метод створює новий об'єкт, який спілкується з Backend і надає інструменти для роботи з бізнес-логікою. *Сервіс* - це об'єкт constructor, який викликають через ключове слово new, в зв'язку з чим наша логіка зв'язується з сервісом за допомогою ключового слова this. Сервіс створює об'єкт-сінглтон.

Фабричні методи повертають об'єкт або функцію, тому ми можемо використовувати замикання, або повертати об'єкт host, до якого можна прив'язувати методи. Можна створювати приватну і публічну області видимості. Всі фабрики стають сервісами, тому ми так їх і називаємо.

Вирази Angular - це схожі на JavaScript сніппети, які можна використовувати в шаблонах, щоб проводити умовні зміни в DOM - як в елементах, так і їх властивості, а також в тексті. Вони живуть всередині посилань {{}} і виконуються в межах $scope. Там немає циклів або if / else, і ми не можемо викидати виключення. Можна робити тільки дрібні операції або викликати значення властивостей $scope. Наприклад, `{{value}}` - це вираз. У виразах можна використовувати логічні оператори на кшталт || і &&, або тернарний оператор `value? true: false`. З їх допомогою можна створювати гнучкі шаблони і оновлювати змінні без перезавантаження сторінок.

Є два види *Директив* - одні працюють зі зв'язками всередині Angular, а інші ми створюємо самі. Директива може робити, що завгодно - надавати логіку для заданого елемента, або сама бути елементом і надавати шаблон з логікою всередині себе. Ідея Директив в розширенні можливостей HTML.

Хоча вбудовані Директиви додають функціональності в HTML, іноді необхідно додати свої функції для подальшого розширення можливостей. Директиви що настроюються - одна з найскладніших концепцій в API Angular, тому що вони не схожі на звичні програмні концепції. Вони виступають в ролі способу Angular реалізувати концепції Веб найближчого майбутнього - настроюються елементи, тіньовий DOM, шаблони і імпорт HTML. Давайте поступово вивчимо директиви, розбивши їх на шари.

Елементи, що налаштовуються, використовуються у випадках повторного використання коду або шаблонного коду, коли ми можемо оголосити один елемент, а код, пов'язаний з ним, буде автоматично приєднаний до нього.

У Angular є чотири способи використання директив - елементи що настроюються, атрибути що настроюються, імена класів і коментарі. Останніх двох краще намагатися уникати, тому що в них легко заплутатися, і до того ж, у коментарів є проблеми з IE. Найбезпечніший і кросбраузерності спосіб - атрибути що настроюються. Давайте розглянемо ці способи в наступному порядку: Element, Attribute, Class, Comment. У Директив є властивість restrict, через яке можна обмежити їх використання одним з цих способів. За замовчуванням, Директиви використовуються через 'EA', що означає Element, Attribute. Інші варіанти - C для класів і M для коментарів.

Тіньовий DOM працює так, що всередині певних частин звичайного DOM документа містяться вкладені документи. Вони підтримують HTML, CSS і області видимості JavaScript. У тіньовому DOM можна визначати як чистий HTML, так і контент, який буде в нього імпортований.

Крім цього особливість AngularJS це фільтри які обробляють інформацію і видають певні набори даних, ґрунтуючись на будь-яку логіку. Це може бути що завгодно, від форматування дати в читаний формат, до списку імен, які починаються на задану букву. Подивимося на популярні вбудовані фільтри. Їх можна використовувати або в DOM через символ вертикальної риси | у виразах, які парсит AngularJS, або через сервіс $filter, який можна використовувати в JavaScript коді замість HTML.

Цілі розробки:

- Відділення DOM-маніпуляції від логіки додатка, що покращує тестований код.
- Ставлення до тестування як до важливої частини розробки. Складність тестування безпосередньо залежить від структурованості коду.
- Роздільна клієнтської і серверної сторони, що дозволяє вести розробку паралельно.
- Проведення розробника через весь шлях створення програми: від проектування користувальницького інтерфейсу, через написання бізнес-логіки, до тестування.

Переваги AngularJS

- На AngularJS легко почати писати програми.
- Двохнаправлене зв’язування даних.
- Модульність що дозволяє відокремити логіку.
- Динамічний застосунок.

Недоліки AngularJS

- Дуже плоха швидкодія виконання за стосунку.
- Велика частина функціоналу з AngularJS не використовувалося на практиці.

- AngularJS працював безпосередньо з DOM, а не с VirtualDOM.
- Низький рівень абстракції, в AngularJS практично вся логіка була зав'язана на HTML.
- Важко бути експертом в AngularJS, якщо сам фреймворк величезний.
- Сучасний Web вимагає нового підходу.

1.7 Фреймворк Angular 2.0+

Angular 2 дуже сильно відрізняється від нинішньої версії, переосмислено майже все. Багато з відмінностей випливають з того, що він використовує TypeScript з анотаціями та типами.

TypeScript - мова програмування, представлений Microsoft в 2012 році і позиціонується як засіб розробки веб-додатків, що розширює можливості JavaScript. Розробником мови TypeScript є Андерс Хейлсберг який створив раніше Turbo Pascal, Delphi і C #. Специфікації мови відкриті і опубліковані в рамках угоди Open Web Foundation Specification Agreement.

TypeScript є назад сумісним з JavaScript і компілюється в останній. Фактично, після компіляції програму на TypeScript можна виконувати в будь-якому сучасному браузері або використовувати спільно з серверної платформою Node.js. Код експериментального компілятора, який транслює TypeScript в JavaScript, поширюється під ліцензією Apache. Його розробка ведеться в публічному репозиторії через сервіс GitHub.

TypeScript відрізняється від JavaScript можливістю явного статичного призначення типів, підтримкою використання повноцінних класів (як в традиційних об'єктно-орієнтованих мовах), а також підтримкою підключення модулів, що покликане підвищити швидкість розробки, полегшити читабельність коду, рефакторинг і повторне його використання, допомогти здійснювати пошук

помилок на етапі розробки та компіляції, і, можливо, прискорити виконання програм.

TypeScript виник через передбачувані недоліків JavaScript в великомасштабних додатках як в Microsoft, так і у інших користувачів JavaScript. Проблеми з розробкою складних програм на JavaScript привели до необхідності полегшення розробки компонентної мови [12].

Розробники TypeScript шукали рішення, яке не порушуватиме сумісність зі стандартом і його крос-платформної підтримкою. Знаючи, що тільки стандарт ECMAScript пропонує підтримку в майбутньому для програмування на базі класів (Class-based programming), TypeScript був заснований на цьому припущенні. Це призвело до створення компілятора JavaScript з набором синтаксичних мовних розширень, збільшеним на основі пропозиції, яке трансформує розширення в JavaScript. У цьому сенсі TypeScript є уявленням про те, що очікувати від ECMAScript 6. Унікальний аспект не в реченні, а в додаванні в TypeScript статичної типізації, що дозволяє статично аналізувати мову, полегшуючи оснащення і IDE підтримку.

TypeScript це розширення мови ECMAScript 5. Додані наступні опції:

- Анотації типів і перевірка їх узгодження на етапі компіляції.
- Вивід типів.
- Класи.
- Інтерфейси.
- Типи, що перераховуються.
- Узагальнене програмування.
- Модулі.
- Скорочений синтаксис «стрілок» для анонімних функцій.
- Розширені можливості пошуку і параметри за замовчуванням.

Додаток Angular 2 тепер складається з компонентів і вдає із себе їх дерево. Ідея схожа на Web-components, навіть розмітка Angular 2 компонентів поміщається в Shadow DOM. Причому якщо ви зберетеся використовувати в своєму додатку Web-components, або, наприклад, Polymer - то синтаксис нічим не буде відрізнятися від використання ваших власних Angular 2 компонентів.

Самі компоненти являють собою ES6 класи з анотаціями, ніякого спеціального синтаксису як, наприклад, для директив у версії 1.x не потрібно. Сервіси тепер теж стали звичайними класами, а завдяки підтримці типізації в AtScript, ін'ектіровать їх можна по типу, без використання синтаксису. $ Inject або ngAnnotate.

Компонентний підхід AngularJS 2.0 дозволяє працювати над частинами рішення паралельно, в тому числі і різними командами. В силу специфіки філософії та архітектури AngularJS2.0 як не можна краще підходить для х команд, які працюють над складними додатками.

Спрощений синтаксис і передача даних між окремими частинами програми, забезпечує незалежність компонентів, творці AngularJS 2.0 полегшили створення каркаса додатки і написання коду. В результаті розробник отримує прості інструменти, за допомогою яких він може робити складні фічи.

Слабко зв’язана архітектура - це ознака хорошого тону в розробці додатків, а Angular 2.0 як фреймворк саме цей тон і задає. Версія 2.0 дозволяє писати код, який легко підтримувати, а його компоненти повторно використати.

Після виходу другої версії Angular 2.0, через рік вийшло вже 2 нові версії які покращують його функціонал, якщо у Angular 4.0 були невеликі зміни, такі як зменшення розміру, та покращення роумінгу, то версія 5 яка вийшла у кінці осені 2017 року покращила його дуже сильно [13].

У попередніх версіях використовуючи інкрементальну збірку Angular-компілятор (ngc) перекомпільовував всі файли при кожній зміні, що уповільнювало процес розробки. Зараз у Angular 5.0 представляє поліпшену watch-опцію,

яка перекомпілюються тільки те, що потрібно. Це означає, що для середніх проектів час компіляції скоротилося з 12-14 секунд до 2-3 секунд.

Тепер є можливість використання режиму інкрементальної збірки спільно з AOT-компіляцією, завдяки чому можна знаходити проблемні ділянки коду програми ще на стадії розробки. Найчастіші помилки, які виникали, це робота з Lazy Modules в різних режимах збірки. У AngularJS і в Angular 2 (спочатку) була тільки JIT-компіляція, в майбутньому був доданий новий тип компіляції, в більш старших версіях Angular режим AOT буде включений за замовчуванням, зараз для цього потрібно або вказувати спеціальний прапор (Angular CLI), або використовувати спеціальний loader для Webpack.

Також була оптимізована збірка. Починаючи з Angular 5.0.0 production builds, створені за допомогою Angular CLI, тепер будуть застосовувати поліпшені оптимізації коду за замовчуванням.

Build optimizer - це інструмент, що входить в поставку Angular CLI, та дозволяє зменшити розмір вашої програми, використовуючи семантичний аналіз всієї програми [14].

Був проведений тест на тестових застосунках на різних версіях Angular 2.0-5.0, результати якого показані на Рис 1.

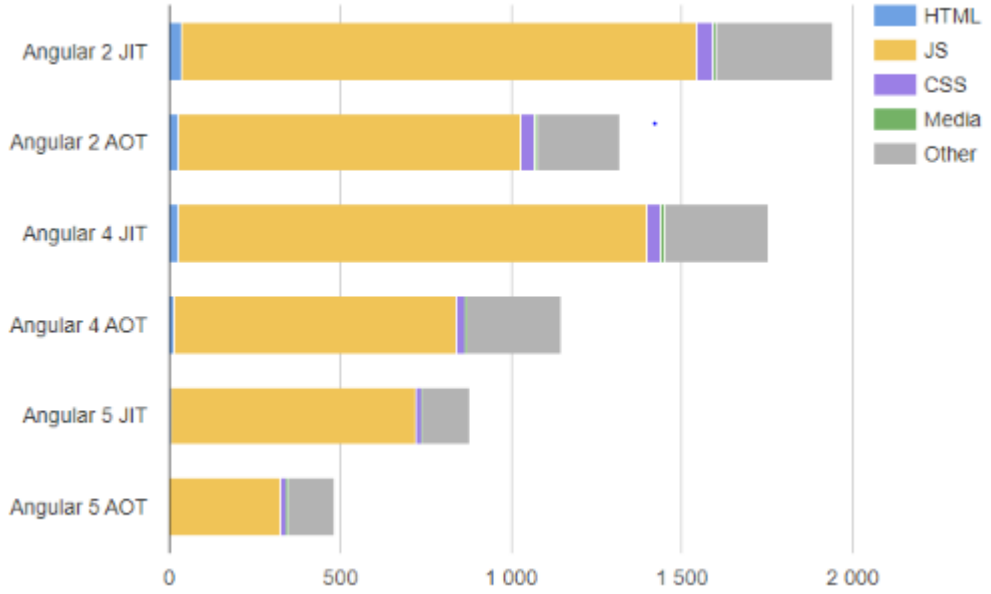

Рис. *1. Загальний час відтворення(мс)*

Переваги Angular 5.0

- Створення компонентного застосунку однією командою.
- Найшвидший Javascript Framework на сьогоднішній час.
- Використання TypeScript.
- Використання VirtualDOM.

Недоліки AngularJS 5.0

- Важкий поріг входу.

1.8 Бібліотека ReactJS

React.js, здебільшого називають *React* - відкрита JavaScript бібліотека для створення інтерфейсів користувача, яка покликана вирішувати проблеми часткового оновлення вмісту веб-сторінки, з якими стикаються в розробці односторінкових застосунків. Розробляється Facebook, Instagram и спільнотою індивідуальних розробників.

React дозволяє розробнику створювати великі веб-застосунки, які використовують дані, котрі змінюються з часом, без перезавантаження сторінки. Його мета полягає в тому, щоб бути швидким, простим, масштабованим. React обробляє тільки користувацький інтерфейс у застосунках. Це відповідає Видові у шаблоні модель-вид-контролер (*MVC*), і може бути використаний у поєднанні з іншими JavaScript бібліотеками або в великих фреймворках MVC, таких як AngularJS. Він також може бути використаний з React на основі надбудов, [9, 11], щоб піклуватися про частини без користувацького інтерфейсу побудованого веб-застосунку.

React надається у вигляді бібліотеки JavaScript з компілятором JSX і пов'язаних з ними інструментів розробки. React полегшує створення багаторазових швидкодіючих компонентів уявлення, які можна використовувати для створення сучасних веб-інтерфейсів. Дотримуючись практичних рекомендацій, легко створювати такі прості в обслуговуванні багаторазові компоненти.

Для оптимізації швидкодії компоненти React спочатку перетворюються в керовану модель Virtual DOM. Ви декларативно описуєте ієрархію компонентів свого призначеного для користувача веб-інтерфейсу і вводите її в віртуальну модель DOM React. Потім React піклується про синхронізацію вашого призначеного для користувача інтерфейсу з фактичної DOM-моделі браузера на відповідні моменти часу. Реалізація моделі Virtual DOM React самодостатня і не залежить від браузера; її навіть можна використовувати для рендеринга на стороні сервера. Virtual DOM виконує оптимізоване порівняння свого внутрішнього стану з DOM-моделями браузерів і вносить мінімальні зміни, необхідні для збереження узгодженості призначеного для користувача інтерфейсу.

Цей підхід абстрактного режиму вирішує великий клас проблем продуктивності, пов'язаних з безпосередньою зміною елементів DOM браузера при кожному візуальному переході (саме так працюють популярні бібліотеки компонентів для користувача інтерфейсу, такі як jQuery UI).

На відміну від аналогічних бібліотек, React не застосовує стандартний стиль управління станом і розробки призначеного для користувача інтерфейсу «модель-уявлення-контролер» (MVC). Замість цього, React займається виключно створенням уявлень, і учасники проекту пояснюють, чому *MVC* - не найкращий спосіб створення складних веб-інтерфейсів [6]. Однак в React немає нічого, що завадило б використовувати конструкцію MVC; багато розробників на початку роботи з React вставляють новий код в рівень представлення існуючих великих проектів на основі MVC.

JSX - це перетворювач і компілятор, що сприймає розширений синтаксис JavaScript. Знайома HTML-подібна форма запису JSX значно спрощує кодування і обслуговування компонентів React. Це спрощення особливо очевидно при оголошенні співвідношень між глибоко вкладеними компонентами. JSX дозволяє розмістити логіку призначеного для користувача інтерфейсу разом з відповідним структурним описом в одному файлі, що допомагає підвищити продуктивність праці і зменшити кількість помилок в великих проектах. При використанні інших платформ може знадобитися синхронізувати в три рази більше файлів: файли шаблону, обробника коду JavaScript і структурного опису HTML. JSX може виконуватися в браузері або самостійно. Перетворювач, вбудований в браузер, допомагає при розробці, тому що після зміни JSX-коду результати видно відразу. При випуску промислових товарів для кращої продуктивності в лінію складання необхідно включити окремий перетворювач.

Елементи - це об'єкти JavaScript, які представляють HTML-елементи. Їх не існують в браузері. Вони описують DOM-елементи, такі як h1, div, або section.

Компоненти - це елементи React, написані розробником. Зазвичай це частини призначеного для користувача інтерфейсу, які містять свою структуру і функціональність. Наприклад, такі як NavBar, LikeButton, або ImageUploader.

Компонування компонентів - наріжний камінь багаторазових компонентів для користувача інтерфейсу. React дозволяє легко компонувати існуючі компо-

ненти React разом з нативними HTML-елементами, отримуючи більш складні компоненти.

Flux - це один з підходів до побудови додатків з використанням компонентів React. Він визначає архітектуру з одностороннім потоком даних, яка вирішує проблеми, пов'язані з взаємодіючими мережами MVC, що мають місце в складних призначених для користувача інтерфейсів, і покращує експлуатаційну технологічність бази коду в довгостроковому плані.

Якщо коротко, загально змінюваний стан додатку, що використовується при рендерингу компонента, передається на більш високі рівні. Традиційні контролери замінені на контролери-вистави. Тепер операції традиційної моделі виконуються за допомогою передачі дій в відповідні сховища через двоелементний диспетчер. Дії - це пакети даних, які декларують виконувану операцію і пов'язані з нею дані. Тісно пов'язані виклики методів перетворюються діями в потік слабо пов'язаних даних. Сховища і їх взаємозалежності обробляються через диспетчер слабко пов'язаним чином. Проектувальники структур можуть помітити подібність зі структурами Command and Chain of Responsibility, а системним інженерам це нагадає маршалинг і сериализацию.

Контролери-вистави ніколи не передають стан додатку один одному - прямо або побічно. Вони реєструють свою зацікавленість у змінах даних у сховищах, а сховища в разі змін повідомляють уявлення про те, що їм потрібно забрати дані і оновлюють свій власний керований стан. Це єдиний потік даних, що входить в ці уявлення.

Relay доповнює Flux можливістю вилучення даних з сервера. Ключова ідея Relay - можливість для кожного компонента React самостійно вказувати свої власні вимоги по вилученню даних. Як правило, це відноситься до даних, що використовуються для візуалізації самого компонента. Relay дозволять статично оголошувати вимоги компонента до даних в тому ж файлі JSX, де знаходиться логіка призначеного для користувача інтерфейсу. Це значно скорочує

характерний безлад в файлах при роботі зі складними, повноцінними додатками. Це також дозволяє негайно перевіряти коректність і синхронізацію в процесі розробки, перезавантажуючи браузер після редагування файлу JSX [9].

У Relay застосовується важлива технологія GraphQL - вільно компонована декларативна мова опису запитів довільно оформлених даних. GraphQL дозволяє високорівневим компонентам складати вимоги своїх власних компонентів - незалежно від змісту самих вимог - і збирати комбінований запит до сервера. Результати GraphQL - запитів зберігаються в сховищі Flux general common, де уявлення, що підписалися, повідомляють про оновлення даних.

React Native замінить DOM браузера для платформи Android або iOS. Це дозволить розробникам React оптимізувати свої додатки для мобільних пристроїв.

У React Native код JavaScript виконується в своєму системному потоці з застосуванням власного інтерпретатора. З нативної платформою код React з'єднує так званий високопродуктивний асинхронної пакетний міст - погоджуючи нативні компоненти UI з конкретною реалізацією призначеного для користувача інтерфейсу. Можна також створювати спеціальні нативні методи для доступу JavaScript через цей міст. HTML-елементи в React замінюються нативними уявленнями і UI-віджетами на мобільній платформі. React Native буде підтримувати стилізацію нативних компонентів через підмножина CSS поверх JavaScript.

За словами розробників, додатки, створені за допомогою React Native, зможуть точно відтворювати тонкі нюанси інтерфейсу, які у користувачів асоціюються тільки з нативними додатками - існуючі сьогодні альтернативи на основі WebViews або HTML5 не можуть цього забезпечити.

Переваги ReactJS

- Зв'язування JavaScript і HTML в JSX робить компоненти простими для розуміння.
- Можливість рендерити ReactJS на сервері.

Недоліки ReactJS

- Не підтримує роботу с AJAX, Promises та систему подій.
- React досить великий, враховуючи те, як мало ви від нього отримуєте, включаючи погану підтримку кросбраузерності.
- Погана та незрозуміла документація.

1.9 Фреймворк Vue.JS

Vue - це сучасний фреймворк для створення користувацьких інтерфейсів. На відміну від фреймворків-монолітів, Vue створений придатним для поступового впровадження [15]. Його ядро в першу чергу вирішує завдання рівня уявлення (view), що спрощує інтеграцію з іншими бібліотеками та існуючими проектами. З іншого боку, Vue повністю підходить і для створення складних одно сторінкових додатків (SPA, Single-Page Applications), якщо використовувати його спільно з сучасними інструментами та додатковими бібліотеками [16].

Іншою важливою концепцією Vue є компоненти. Ця абстракція дозволяє збирати великі програми з менших шматочків. Компоненти є придатні до повторного використання об'єкти. Якщо подумати, майже будь-який інтерфейс може бути представлений як дерево компонентів. Компоненти - це одна з найпотужніших можливостей Vue. Компоненти розширюють базові HTML-елементи, дозволяючи інкапсулювати повторно використовуваний код. Не вдаючись в подробиці, можна сказати, що компоненти - це призначені для користувача елементи, до яких компілятор Vue прив'язує певну поведінку. У деяких ви-

падках компоненти також можна задати за допомогою нативних елементів, розширених спеціальним атрибутом is. У Vue, компонент - це, по суті, екземпляр Vue з попередньо встановленими опціями.

Компоненти Vue досить схожі на призначені для користувача елементи, які є частиною специфікації W3C Web Components. Справа в тому, що синтаксис компонентів Vue і навмисно слідує цій специфікації. Зокрема, компоненти Vue реалізують API слоти і спеціальний атрибут is [17]. Разом з тим, є і кілька ключових відмінностей:

- Специфікація Web Components все ще перебуває в статусі чернетки і не реалізована ні в одному з браузерів. Компоненти Vue, навпаки, не вимагають ніяких поліфілів і стійко працюють у всіх підтримуваних браузерах (IE9 і вище). При необхідності компоненти Vue можуть бути обгорнуті в нативні призначені для користувача елементи.
- Компоненти Vue надають важливі можливості, недоступні в простих користувальницьких елементах. Найважливіші з них: крос-компонентна передача даних, комунікація з використанням призначених для користувача подій і інтеграція з інструментами створення збірок.

1.10 Висновки з розділу

Проаналізувавши літературу про компонентний підхід, та його впровадження у Вебі, були розглянуті існуючі Javascript фреймворки та бібліотеки, проведено аналіз їх переваг та недоліків.

На основі цього аналізу буде створено програмний застосунок, використовуючи Angular 5.0, який якнайкраще демонструє реалізацію компонентного підходу з дуже гарною оптимізацією.

РОЗДІЛ 2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ ДЛЯ КОМПОНЕНТНОГО ПІДХОДУ ДО СТВОРЕННЯ ЗАСТОСУНКІВ

2.1 Клієнтська частина

2.1.1 Структура застосунку Angular 5.0, та створення компонентів

Angular являє собою фреймворк для створення клієнтських додатків у форматі HTML та JavaScript або мови типу TypeScript, яка компілюється на JavaScript. Фреймворк складається з декількох бібліотек, деякі з яких є основними а деякі - необов'язковими.

Застосунок Angular складається з HTML-шаблонів з ангуляризованою розміткою, компонентів класів, які керують цими шаблонами, та додають логіку до сервісів та викликаються залежностями сервісів та компонентів у модулях. На Рис. 2 показана повна структура застосунку Angular.

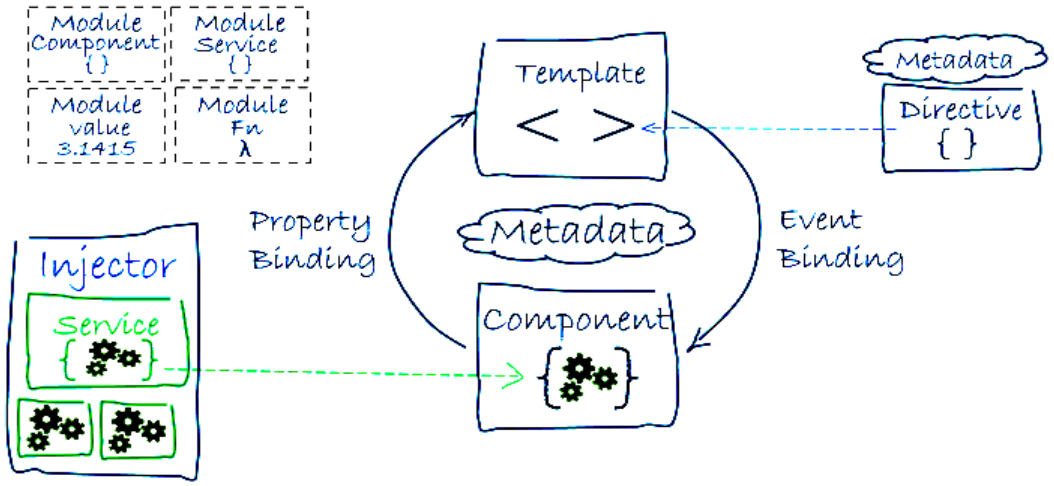

Рис. *2 Структура застосунку Angular*

Додаток Angular складається з окремих *модулів*. Як правило, додатки складаються з декількох модулів. І кожен додаток Angular як мінімум має один кореневий модуль (*root module*), який, за замовчуванням, називається *AppModule*.

Для роботи модуля йому необхідні ряд бібліотек, тому на початку файлу йде їх підключення. Ім'я кожної бібліотеки Angular починається з префікса @angular.

Бібліотеки встановлюються через пакетний менеджер npm і імпортуються за допомогою директиви import. Наприклад, імпорт функціональності декоратора NgModule з бібліотеки @angular/core: `import { NgModule } from '@angular/core'`.

Тобто для роботи застосунку треба імпортувати всі модулі і класи, які використовує даний модуль. Зокрема, для AppModule необхідні:

- NgModule: функціональність декоратора NgModule, без якої ми не зможемо створити модуль;
- BrowserModule: модуль, необхідний для роботи з браузером;
- FormsModule: модуль, необхідний для роботи з формами html і, зокрема, з елементами input. (Так як клас компонента працює з подібними елементами, то ми зобов'язані також імпортувати цей модуль);
- AppComponent: функціональність кореневого компонента додатка.

Безпосередньо сам модуль представлений класом AppModule, який на перший погляд нічого не робить і не містить ніякого функціоналу: `export class AppModule { }`. Однак в Angular модуль це не просто клас. Кожен модуль повинен визначатися з декоратором @NgModule. NgModule представляє функцію-декоратора, яка приймає об'єкт, властивості якого описують метадані модуля. Найбільш важливі властивості:

- declarations: класи уявлень (view classes), які належать модулю. Angular має три типи класів уявлень: компоненти (components), директиви (directives), канали (pipes);
- exports: набір класів уявлень, які повинні використовуватися в шаблонах компонентів з інших модулів;

- imports: інші модулі, класи яких необхідні для шаблонів компонентів з поточного модуля;
- providers: класи, створюють сервіси, що використовуються модулем;
- bootstrap: кореневої компонент, який викликається за замовчуванням при завантаженні програми.

Одним з ключових елементів програми є компоненти. *Компонент,* що управляє відображенням уявлення на екрані має наступний вигляд.

Лістинг *1*

```
import { Component } from '@angular/core';
@Component({
    selector: 'my-app',
    template: `<label>Name:</label>
            <input [(ngModel)]="name" placeholder="name">`
})
export class AppComponent {
    name: "";
}
```

Як ми бачимо з лістингу 1 сам клас компонента складається лише з його оголошення та змінної name.

Щоб клас міг використовуватися в інших модулях, він визначається з ключовим словом export. У самому ж класі визначена лише одна змінна, яка в якості значення зберігає деякий рядок.

Для створення компонента необхідно імпортувати функцію декоратора @Component з бібліотеки @angular/core. Декоратор @Component дозволяє ідентифікувати клас як компонент.

Якщо не застосовувати декоратор @Component до класу AppComponent, то клас AppComponent компонентом б не був. Декоратор @Component як параметр приймає об'єкт з конфігурацією, яка вказує фреймворку, як працювати з компо-

нентом і його поданням. За допомогою властивості template шаблон являє шматок розмітки HTML з вкрапленнями коду Angular. Фактично шаблон це і є уявлення, яке побачить користувач при роботі з додатком.

Кожен компонент повинен мати один шаблон. Однак необов'язково визначати шаблон безпосередньо за допомогою властивості template. Можна винести шаблон у зовнішній файл з розміткою html, а для його підключення використовувати властивість templateUrl.

Шаблон може бути однорядковим або багаторядковим. Якщо шаблон багато строковий, то він полягає в косі лапки (`), які варто відрізняти від стандартних ординарних лапок (').

Також в прикладі вище встановлюється властивість selector, який визначає селектор CSS. В елемент з цим селектором Angular буде додавати уявлення компонента. Наприклад, в прикладі вище селектор має значення my-app. Відповідно якщо html-сторінка містить елемент <my-app> </ my-app>, то саме цей елемент буде використовуватися для рендеринга уявлення компонента.

Іншими елементами Angular є *директиви* які визначають набір інструкцій, що застосовуються при рендерингу html-коду. Директива представляє клас з директивними метаданими. У TypeScript для прикріплення метаданих до класу застосовується декоратор @Directive.

У Angular є три типи директив:

- Компоненти: компонент по суті також є директивою, а декоратор @Component розширює можливості декоратора @Directive за допомогою додавання функціоналу по роботі з шаблонами.
- Атрибутивні: вони змінюють поведінку вже існуючого елемента, до якого вони застосовуються. Наприклад, ngModel, ngStyle, ngClass.
- Структурні: вони змінюють структуру DOM за допомогою додавання, зміни або видалення елементів hmtl. Наприклад, це директиви ngFor і ngIf.

Останнім важливим елементом є *сервіси* в Angular які представляють досить широкий спектр класів, що виконують деякі специфічні завдання, наприклад, логуваня, роботу з даними і тощо.

На відміну від компонентів і директив сервіси не працюють з уявленнями, тобто з розміткою html, не роблять на неї прямого впливу. Вони виконують строго певне і досить вузьке завдання.

Стандартні завдання сервісів:

- Надання даних до додатку. Сервіс може сам зберігати дані в пам'яті, або для отримання даних може звертатися до будь-якого джерела даних, наприклад, до сервера.
- Сервіс може представляти канал взаємодії між окремими компонентами програми.
- Сервіс може інкапсулювати бізнес-логіку, різні обчислювальні завдання, завдання по логуванню, які краще виносити з компонентів. Тим самим код компонентів буде зосереджений безпосередньо на роботі з поданням. Крім того, тим самим ми також можемо вирішити проблему повторення коду, якщо нам буде потрібно виконати одну і ту ж задачу в різних компонентах і класах.

2.1.2 CSS-препроцессор

Препроцесор компілює код в CSS, що працює однаково у всіх браузерах.

CSS препроцесор – це програма, яка компілює написаний код (з використанням спец. синтаксису) в чистий CSS код. При цьому, препроцесор надає розробнику нові можливості та функції.

Синтаксис

Найважливіше, при використанні CSS препроцесора - це розуміти синтаксис. На щастя, він є однаковий для всіх трьох CSS-препроцесорів, які будуть розглянуті, а саме це: SASS, LESS, Stylus.

Sass & LESS

Як Sass, так і LESS використовують стандартний CSS синтаксис. Це робить їх надзвичайно простими для розуміння. SASS використовує розширення .scss, LESS – .less.

Stylus

Синтаксис Stylus є більш різноманітним. Використовуючи розширення файлу .styl, Stylus допускає стандартний CSS синтаксис, але також пропонує інші варіанти, де дужки, двокрапки та крапки з комою є необов'язковими. Також можна використовувати декілька варіантів в одному стилі.

Змінні

Змінні можна оголошувати і використовувати у всій таблиці стилів. Вони можуть приймати будь-яке, використовуване в CSS значення (напр. кольори, цифри чи текст).

Вкладеність

При використанні вкладеності в звичайному CSS, доводиться багато разів переписувати ім'я батьківського елемента. Це не досить зручно. Препроцесор вирішує цю проблему: ми можемо писати стиль дочірнього елемента всередині дужок батьківського компонента. Всі три препроцесори використовують однаковий синтаксис для вкладених селекторів. Також, за допомогою символу & можна посилатися на батьківський селектор.

Домішки

Домішки це функції, в яких можна окремо виділити певні властивості і повторно використовувати їх кілька разів у всьому CSS файлі. Коли домішка викликається з CSS селектора, препроцесор розпізнає аргумент домішки, і стилі, що розташовані всередині домішки, застосовуються до селектора.

Спадкування

Спадкування - це здатність одних CSS селекторів наслідувати властивості інших. LESS не підтримує спадкування, так як Sass та Stylus. Замість додавання

декількох селекторів до одного набору властивостей, він розглядає спадкування, як домішку без аргументів і імпортує стилі в кожен селектор. Недоліком є те, що властивості повторюються в компільованому CSS.

Color функції

Color функції – це вбудовані функції, що дозволяють змінювати колір при компіляції. Може бути корисно для створення градієнтів, інших відтінків кольорів при наведенні курсору.

Операції

Препроцесор надає можливість використання математичних розрахунків в CSS коді.

Vendor Prefixes

Особливо необхідно використовувати препроцесор при створенні Vendor Prefixes – це економить багато часу і зусиль.

Відомі особливості

CSS препроцесор має деякі свої особливості використання. Ось деякі найбільш поширені:

- *Повідомлення про помилки*

  Якщо ви писали CSS протягом великого періоду часу, то, я впевнений, у вас були ситуації, коли ви розуміли, що десь є помилки, та не могли знайти їх. CSS препроцесори повідомляють про помилки. Якщо з вашим кодом щось не так, препроцесор повідомить вам де, і можливо, чому виникає помилка в коді.

- *Коментарі*

  При компіляції CSS, будь-який коментар, оголошений подвійним слешем, буде видалений(напр. //comment), а кожен коментар «слеш-зірочка» (напр. /* comment */) залишається. Тож, використовуйте подвійний слеш для коментарів, які необхідні при роботі з кодом до

компіляції, а слеш з зірочкою для коментарів, які хочете бачити в коді CSS.

Отже, кожен CSS препроцесор (SASS, LESS і Stylus), має свій унікальний спосіб досягнення однієї і тієї ж мети – використання корисних, не підтримуваних функцій CSS, зберігаючи сумісність браузера і чистоту коду. Найбільш ефективним для мене є SASS, тому власне я його і обрав для реалізації моєї роботи.

2.2 Серверна частина

При виборі мови для реалізації серверної частини web-застосунку розглядалися три найбільш популярні та відомі мови програмування: PHP, C# (ASP.NET Web API) та JavaScript (Node.js).

2.2.1 Серверна мова PHP

PHP (англ. PHP: Hypertext Preprocessor — PHP: гіпертекстовий препроцесор), попередня назва: Personal Home Page Tools — скриптова мова програмування, була створена для генерації HTML-сторінок на стороні веб-сервера. PHP є однією з найпоширеніших мов, що використовуються у сфері веб-розробок (разом із Java, .NET, Perl, Python, Ruby). PHP підтримується переважною більшістю хостинг-провайдерів. PHP — проект відкритого програмного забезпечення.

PHP інтерпретується веб-сервером у HTML-код, який передається на сторону клієнта. На відміну від скриптової мови JavaScript, користувач не бачить PHP-коду, бо браузер отримує готовий html-код. Це є перевага з точки зору безпеки, але погіршує інтерактивність сторінок. Але ніщо не забороняє використо-

вувати PHP для генерування і JavaScript-кодів які виконуються вже на стороні клієнта.

PHP — мова, код якої можна вбудовувати безпосередньо в html-код сторінок, які, у свою чергу, будуть коректно оброблені PHP-інтерпретатором. Обробник PHP просто починає виконувати код після відкриваючого тегу (<?php) і продовжує виконання до того моменту, поки не зустріне закриваючий тег (?>).

Велика різноманітність функцій PHP дає можливість уникати написання багаторядкових функцій, призначених для користувача, як це відбувається в C або Pascal.

Особливості:

- *Наявність інтерфейсів до багатьох баз даних*
  - в PHP вбудовані бібліотеки для роботи з MySQL, PostgreSQL, mSQL, Oracle, dbm, Hyperware, Informix, InterBase, Sybase.
  - через стандарт відкритого інтерфейсу зв'язку з базами даних (Open Database Connectivity Standard — ODBC) можна підключатися до всіх баз даних, до яких існує драйвер.
- *Традиційність*

  Мова PHP здається знайомою програмістам, що працюють в різних областях. Багато конструкцій мови запозичені з C, Perl. Код PHP дуже схожий на той, який зустрічається в типових програмах на C або Pascal. Це помітно знижує початкові зусилля при вивченні PHP. PHP — мова, що поєднує переваги Perl і C та спеціально спрямована на роботу в Інтернеті, мова з універсальним і зрозумілим синтаксисом. І хоча PHP є досить молодою мовою, вона здобула таку популярність серед web-програмістів, що в наш час є мало не найпопулярнішою мовою для створення веб-застосунків (скриптів).

- *Наявність сирцевого коду та безкоштовність*

  Стратегія Open Source, і розповсюдження початкових текстів програм в масах, безсумнівно справили благотворний вплив на багато проектів, в першу чергу — Linux хоч і успіх проекту Apache сильно підкріпив позиції прихильників Open Source. Сказане відноситься і до історії створення PHP, оскільки підтримка користувачів зі всього світу виявилася дуже важливим чинником в розвитку проекту PHP.

  Ухвалення стратегії Open Source і безплатне розповсюдження початкових текстів PHP надало неоціненну послугу користувачам. Окрім цього, користувачі PHP в усьому світі є свого роду колективною службою підтримки, і в популярних електронних конференціях можна знайти відповіді навіть на найскладніші питання.

- *Ефективність*

  Ефективність є дуже важливим чинником у програмуванні для середовищ розрахованих на багато користувачів, до яких належить і web. Важливою перевагою PHP є те, що ця мова належить до інтерпретованих. Це дозволяє обробляти сценарії з достатньо високою швидкістю. За деякими оцінками, більшість PHP-сценаріїв (особливо не дуже великих розмірів) обробляються швидше за аналогічні їм програми, написані на Perl. Проте хоч би що робили розробники PHP, виконавчі файли, отримані за допомогою компіляції, працюватимуть значно швидше — в десятки, а іноді і в сотні разів. Але продуктивність PHP достатня для створення цілком серйозних веб-застосунків.

2.2.2 Платформа ASP.NET Web API

Платформа Web API ASP.NET дозволяє з легкістю створювати служби HTTP для широкого діапазону клієнтів, включаючи браузери і мобільні пристрої. Web-API ASP.NET ідеально підходить для розробки RESTful додатків на платформі .NET Framework.

Web API представляє інший спосіб побудови програми ASP.NET дещо відмінний від ASP.NET MVC. Web API представляє собою веб-службу, яка може взаємодіяти з різними застосунками. При цьому застосунок може бути веб-додатком ASP.NET, або може бути мобільним або звичайним десктопним застосунком.

Також треба відзначити, що платформа Web API 2 не є частиною фреймворка ASP.NET MVC і може бути задіяна як у зв'язці з MVC, так і в поєднанні з Web Forms. Тому в Web API є своя система версій. Так, перша версія з'явилася з .net 4.5. А разом з .NET 4.5.1 і MVC 5 вийшла Web API 2.0.

Особливості:

- Дозволяє створювати мережеві API-інтерфейси, які підтримують велику кількість різних типів вмісту, в тому числі XML, JSON і т. Д.
- Використовує основні протоколи і формати, такі як HTTP, WebSockets, SSL, JQuery, JSON і XML. Відсутня підтримка протоколів високого рівня, таких як надійний обмін повідомленнями і транзакції.
- HTTP працює через «запит-відповідь», але підтримуються додаткові шаблони через інтеграцію SignalR і WebSockets.
- Є різні способи опису Web API - від автоматично формованих HTML-сторінок довідки з описом фрагментів до структурованих метаданих для інтеграції API в OData.
- Поставляється разом з платформою .NET Framework, але має відкритий код, доступний також по зовнішніх каналах як незалежне завантаження.

2.2.3 Платформа Node.js

Платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript. Засновником платформи є Раян Дал (Ryan Dahl). Платформа Node.js перетворила мову JavaScript, що в основному використовувалась в браузерах на мову загального використання з великою спільнотою розробників.

Функції платформи не обмежені створенням серверних скриптів для веб, платформа може використовуватися і для створення звичайних клієнтських і серверних мережевих програм. Для забезпечення виконання JavaScript-коду використовується розроблений компанією Google рушій V8.

Для забезпечення обробки великої кількості паралельних запитів у Node.js використовується асинхронна модель запуску коду, заснована на обробці подій в не блокуючому режимі та визначенні обробників зворотних викликів (callback). Як способи мультиплексування з'єднань підтримується epoll, kqueue, /dev/poll і select. Для мультиплексування з'єднань використовується бібліотека libuv, для створення пулу потоків (thread pool) задіяна бібліотека libeio, для виконання DNS-запитів у не блокуючому режимі інтегрований c-ares. Всі системні виклики, що спричиняють блокування, виконуються всередині пула потоків і потім, як і обробники сигналів, передають результат своєї роботи назад через неіменовані канали (pipe).

За своєю суттю Node.js схожий на фреймворки Perl AnyEvent, Ruby Event Machine і Python Twisted, але цикл обробки подій (event loop) у Node.js прихований від розробника і нагадує обробку подій у веб-застосунку, що працює в браузері. При написанні програм для Node.js необхідно враховувати специфіку подієво-орієнтованого програмування

Особливості:

- асинхронна однопотокова модель виконання запитів;

- неблокуючий ввід/вивід;
- система модулів CommonJS;
- рушій JavaScript Google V8;
- Для управління модулями використовується пакетний менеджер; npm (node package manager).

2.2.4 Порівняльна характеристика мови PHP та платформ ASP.NET WEB API (C#) і Node.js (JavaScript)

На таблиці 1 представлена порівняльна характеристика функціональних можливостей мов програмування PHP, C# та JavaScript.

Таблиця *1*

Порівняльна характеристика мови PHP та платформ WEB API і Node.js

Характеристика	PHP	ASP.NET WEB API (C#)	Node.js (JavaS-cript)
Багатозадачність	+	+	+
RESTful	+	+	+
Динамічна типізація	+	+	+
Вільне ПЗ	+/-	+	+
Рекомендований обсяг пам’яті	256 MB	100 MB	3 kB
Розширення / плагіни	+	+	+
Атомарність	+	+	+
Ізольованість	+	+	+
Час запуску про-грами	1 ms	25 ms	?

Характеристика	PHP	ASP.NET WEB API (C#)	Node.js (JavaS-cript)
Об'єктно-реляційна проекція (ORM)	+	+	+
Лямбда-вирази	+	+	+
Підтримка Websocket	+	+	+
Підтримка подій	+	+	+
Масштабованість	+	+	+
Безпека типів	+	+	-
Спадкування	+	+	+

де

+ — можливість присутня;

- — можливість відсутня;

-/+ — можливість підтримується дуже обмежено;

? — нема даних.

З таблиці 2.1 видно, що засоби дуже подібні між собою, але ASP.NET WEP API 3 має дуже гарну швидкість обробки запитів, та дуже зрозумілий інтерфейс, саме він був обран для розробки тестового застосунку.

2.2.5 Дослідження Entity Framework

Entity Framework являє спеціальну об'єктно-орієнтовану технологію на базі фреймворка .NET для роботи з даними. Якщо традиційні засоби ADO.NET дозволяють створювати підключення, команди та інші об'єкти для взаємодії з базами даних, то Entity Framework являє собою більш високий рівень абстракції, який дозволяє абстрагуватися від самої бази даних і працювати з даними неза-

лежно від типу сховища. Якщо на фізичному рівні ми оперуємо таблицями, індексами, первинними і зовнішніми ключами, але на концептуальному рівні, який нам пропонує Entity Framework, ми вже працює з об'єктами.

Перша версія Entity Framework - 1.0 вийшла ще в 2008 році і представляла дуже обмежену функціональність, базову підтримку ORM (object-relational mapping - відображення даних на реальні об'єкти) і один єдиний підхід до взаємодії з бд - Database First. З виходом версії 4.0 у 2010 році багато чого змінилося - з цього часу Entity Framework став рекомендованої технологією для доступу до даних, а в сам фреймворк були введені нові можливості взаємодії з бд - підходи Model First і Code First.

Додаткові поліпшення функціоналу пішли з виходом версії 5.0 в 2012 році. І нарешті, в 2013 році був випущений Entity Framework 6.0, що володіє можливістю асинхронного доступу до даних.

Центральної концепцією Entity Framework є поняття сутності або entity. Сутність представляє набір даних, асоційованих з певним об'єктом. Тому дана технологія передбачає роботу не з таблицями, а з об'єктами і їх наборами.

Будь-яка сутність, як і будь-який об'єкт з реального світу, має низку властивостей. Наприклад, якщо сутність описує людини, то ми можемо виділити такі властивості, як ім'я, прізвище, зріст, вік, вага. Властивості необов'язково представляють прості дані типу int, а й можуть представляти більш комплексні структури даних. І у кожної сутності може бути одна або кілька властивостей, які будуть відрізняти цю сутність від інших і будуть унікально визначати цю сутність. Подібні властивості називають ключами.

При цьому суті можуть бути пов'язані асоціативної зв'язком один-до-багатьох, один-к-одному і багато-до-багатьох, подібно до того, як в реальній базі даних відбувається зв'язок через зовнішні ключі.

Відмінною рисою Entity Framework є використання запитів LINQ для вибірки даних з БД. За допомогою LINQ ми можемо не тільки отримувати певні

рядки, що зберігають об'єкти, з БД, а й отримувати об'єкти, пов'язані різними асоціативними зв'язками.

Іншим ключовим поняттям є Entity Data Model. Ця модель зіставляє класи сутностей з реальними таблицями в БД. Entity Data Model складається з трьох рівнів: концептуального, рівень сховища і рівень зіставлення (маппінга).

На концептуальному рівні відбувається визначення класів сутностей, які використовуються в додатку.

Рівень сховища визначає таблиці, стовпці, відносини між таблицями і типи даних, з якими порівнюється використовувана база даних.

Рівень зіставлення (маппінга) служить посередником між попередніми двома, визначаючи зіставлення між властивостями класу суті і стовпцями таблиць. Таким чином, ми можемо через класи, визначені у додатку, взаємодіяти з таблицями з бази даних.

Entity Framework передбачає три можливі способи взаємодії з базою даних:

- Database first: Entity Framework створює набір класів, які відображають модель конкретної бази даних.
- Model first: спочатку розробник створює модель бази даних, по якій потім Entity Framework створює реальну базу даних на сервері.
- Code first: розробник створює клас моделі даних, які будуть зберігатися в БД, а потім Entity Framework за цією моделлю генерує базу даних і її таблиці.

2.3 Системи керування базою даних

СКБД повинна забезпечувати реляційну модель роботи з даними. Сама модель має на увазі певний тип зв'язку між сутностями з різних таблиць. Щоб

зберігати і працювати з даними, такий тип СКБД повинен мати певну структуру (таблиці). У таблицях кожен стовпець може містити дані різного типу. Кожен запис складається з безлічі атрибутів (стовпців) і має унікальний ключ, що зберігається в тій же таблиці - всі ці дані взаємопов'язані між собою, як описано в реляційній моделі.

Було розглянуто чотири основні розповсюджуваних СКБД:

- SQLite - дуже потужна вбудована система керування.
- MySQL – одна з найпопулярніших і поширених СКБД.
- PostgreSQL - найбільш просунута СКБД.
- Microsoft SQL - найпопулярніша СКБД.

SQLite

База даних легко вбудовується в застосунки. Так як ця система базується на файлах, то вона надає досить широкий набір інструментів для роботи з нею, в порівнянні з мережевими СКБД. При роботі з цією СКБД звернення відбуваються безпосередньо до файлів (в ці файлах зберігаються дані), замість портів і гнізд в мережевих СКБД. Саме тому SQLite дуже швидка, а також потужна завдяки технологіям обслуговуючих бібліотек.

Переваги SQLite

- Файлова структура - вся база даних складається з одного файлу, тому її дуже легко переносити на різні машини.
- Використовувані стандарти - хоча може здатися, що ця СКБД примітивна, але вона використовує SQL. Деякі особливості опущені (праве зовнішнє об'єднання або для кожного оператора), але основні все-таки підтримуються.
- Відмінна при розробці та тестуванні - в процесі розробки застосунків часто з'являється необхідність масштабування. SQLite пропонує все, що

необхідно для цих цілей, так як складається всього з одного файлу і бібліотеки написаної на мові С.

Недоліки SQLite

- Відсутність системи користувачів - більші СКБД включають в свій склад системи керування правами доступу користувачів. Зазвичай застосування цієї функції не так критично, так як ця СКБД використовується в невеликих застосунках.
- Відсутність можливості збільшення продуктивності - знову, виходячи з проектування, досить складно вичавити щось більш продуктивне з цієї СКБД.

MySQL

MySQL - це найпоширеніша повноцінна серверна СКБД. MySQL дуже функціональна, вільно розповсюджена СКБД, яка успішно працює з різними сайтами і веб-застосунками. Навчитися користуватися цією СКБД досить просто, так як на просторах інтернету ви легко знайдете величезну кількість інформації.

Незважаючи на те, що в ній не реалізований весь SQL функціонал, MySQL пропонує досить багато інструментів для розробки застосунків. Так як це серверна СКБД, застосунки для доступу до даних, на відміну від SQLite працюють зі службами MySQL.

Переваги MySQL

- Простота в роботі - MySQL встановити досить просто. Існують різноманітні програми, наприклад графічний інтерфейс, дозволяє досить легко працювати з БД.
- Багатий функціонал - MySQL підтримує більшість функціоналу SQL.
- Безпека - велика кількість функцій забезпечують безпеку, які по замовчуванню підтримуються.

- Масштабованість - MySQL легко працює з великими обсягами даних і легко масштабується.
- Швидкість - спрощення деяких стандартів дозволяє MySQL значно збільшити продуктивність.

Недоліки MySQL

- Відомі обмеження - за задумом в MySQL закладені деякі обмеження функціонала, які іноді необхідні в особливо вимогливих застосунках.
- Проблеми з надійністю - через деякі способи обробки даних MySQL (зв'язку, транзакції, аудити) іноді поступається іншим СКБД по надійності.
- Повільна розробка - MySQL хоча технічно відкрите ПЗ, існують скарги на процес розробки.

PostgreSQL

PostgreSQL є найбільш професійним з усіх трьох розглянутих нами СКБД. Вона вільно розповсюджується і максимально відповідає стандартам SQL. PostgreSQL або Postgres намагаються повною мірою використовувати ANSI / ISO SQL стандарти своєчасно з виходом нових версій.

Від інших СКБД PostgreSQL відрізняється підтримкою затребуваного об'єктно-орієнтованого і/або реляційного підходу до баз даних. Наприклад, повна підтримка надійних транзакцій, тобто атомарність, послідовність, ізоляційні, міцність (атомарність, узгодженість, ізольованість, довговічність (ACID).) Завдяки потужним технологіям Postgre дуже продуктивна. Паралельність досягнута не за рахунок блокування операцій читання, а завдяки реалізації управління різноманітним паралелізмом (MVCC), що також забезпечує відповідність ACID. PostgreSQL дуже легко розширювати своїми процедурами, які називаються збережені процедури. Ці функції спрощують використання постійно повторюваних операцій.

Хоча PostgreSQL і не може похвалитися великою популярністю на відміну від MySQL, існує досить велика кількість застосунків, що полегшують роботу з PostgreSQL, незважаючи на всю потужність функціоналу. Зараз досить легко встановити цю СКБД використовуючи стандартні менеджери пакетів операційних систем.

Переваги PostgreSQL

- Відкрите ПЗ, що відповідає стандарту SQL - PostgreSQL - безкоштовне ПЗ з відкритим вихідним кодом. Ця СКБД є дуже потужною системою.
- Велике співтовариство - існує досить велика спільнота, в якій ви запросто знайдете відповіді на свої питання.
- Велика кількість доповнень - незважаючи на величезну кількість вбудованих функцій, існує дуже багато доповнень, що дозволяють розробляти дані для цієї СКБД і керувати ними.
- Розширення - існує можливість розширення функціоналу за рахунок збереження своїх процедур.
- Об'єктність - PostrgreSQL це не тільки реляційна СКБД, але також і об'єктно-орієнтована з підтримкою успадкування і багато іншого.

PostgreSQL Недоліки

- Продуктивність - при простих операціях читання PostgreSQL може значно уповільнити сервер і бути повільніше своїх конкурентів, таких як MySQL.
- Популярність - за своєю природою, популярністю ця СКБД похвалитися не може, хоча і є досить велика спільнота.
- Хостинг - в силу вище перерахованих факторів іноді досить складно знайти хостинг з підтримкою цієї СКБД.

MicrosoftSQL

SQL Server є надійною базою даних для будь-яких цілей, може продовжувати розширюватися в міру наповнення інформацією, без помітного зменшення швидкодії операцій із записами в розрахованому на багато користувачів режимі. Користувачі можуть бути додані шляхом модернізації обладнання. В останньому тесті підтримувалося до 4600 користувачів бази даних.

Забезпечується максимальна безпека. Дані захищені від несанкціонованого доступу за рахунок інтеграції мережевої безпеки з сервером безпеки. Оскільки безпека на рівні користувача, користувачі можуть мати обмежений доступ до запису даних, тим самим захищаючи їх від модифікації або пошуку, вказавши доступ на рівні користувача привілеєм. Крім того, з даними, що зберігаються на окремому сервері, сервер працює як шлюз, який обмежує несанкціонований доступ.

SQL Server обробляє запити від користувачів і тільки відправляє користувачеві результати запиту. Таким чином, мінімальна інформація передається по мережі. Це покращує час відгуку і усуває вузькі місця в мережі. Це також дозволяє використовувати SQL Server в якості ідеальної бази даних для інтернет.

Технічне обслуговування SQL Server дуже просте і не вимагає великих знань. Можливі зміни в структурі даних а так само резервне копіювання під час роботи сервера, без зупинки.

Два основних мови розробки додатків використовується для добування інформації з даних SQL Server. Це C ++ і C#. Ці мови є частиною Visual Studio.Net, інтегрованого середовища розробки Microsoft. Купівля додатків, розроблених за допомогою цих продуктів гарантує, що програмне забезпечення буде модернізуватися і розширюватися і розвиватися в майбутньому.

SQL Server є додатком бази даних при роботі на. Net, новітні розробки Microsoft. Вибравши Microsoft SQL Sever в якості бази даних інформації для

компанії, додаток може розширюватися і адаптуватися в міру зміни бізнес-клімату.

Отже, оскільки надійність та цілісність даних є найбільш пріоритетним у якісних за стосунках та з серверної частини ми використовуємо ASP.NET , моїм вибором стала СКБД MicrosoftSQL.

Переваги MicrosoftSQL

- Простота адміністрування.
- Можливість підключення до Web.
- Швидкодія і функціональні можливості механізму сервера СУБД.
- Наявність засобів віддаленого доступу.

MicrosoftSQL Недоліки

- Складність роботи з ієрархічними структурами.
- Невідповідність реляційної моделі даних.

2.4 Висновки з розділу

Під час виконання даного розділу було розглянуто, проаналізовано та досліджено всі найсучасніші методи розробки ПЗ та обрано для реалізації наступні технології:

1. Серверна мова програмування C#;
2. Серверна платформа ASP.NET WEB API 3;
3. Серверний фреймворк Entity Framework;
4. Клієнтська мова програмування TypeScript;
5. Клієнтський JavaScript-фреймворк Angular 5;
6. CSS-препроцессор SASS;
7. Клієнтський SASS-фреймворк Bootstrap 4;
8. СКБД MSSQL.

РОЗДІЛ 3 ПРОЕКТ ПРОГРАМНОЇ СИСТЕМИ TASK MANAGER

3.1 Функціональні вимоги до тестового застосунку

Завданням роботи є дослідження компонентного підходу у сучасних Javascript фреймворках та бібліотеках, а також повторного використання компонентів у застосунку. Для досягнення мети було прийнято рішення розробити Task Manager для корпоративних цілей, який якнайкраще покаже використання компонентного підходу у обраному фреймворку – Angular. Крім цього потрібно розробити таку архітектуру застосунку, щоб компоненти могли повторно використовуватися.

Задача також вимагає розробки комплексу програмних засобів для клієнтської частини, який дозволить створювати редаговані таблиці для тасків з функціоналом drag-and-drop інтерфейсу, а для серверної частини буде використовувати RESTful асінхроні запити, та зберігати данні у БД.

Отже виходячи з поставлених цілей вирішено розробити програмний продукт, який дозволить регіструватися новим користувачам, для яких буде створено редаговану таблицю з завданнями, усі рядки та стовпчики якої можуть редагуватися, можливість створювати задачі для кожного користувача з drag-and-drop інтерфейсом, та зберігати усі данні до БД. Для більшою зручності створена сторінка усіх завдань з створеним фільтром, щоб при великій кількості завдань була можливість відфільтрувати та знайти лише потрібні нам. Також необхідна сторінка з усіма користувачами з можливістю редагувати їх данні, та видаляти для адміністратора застосунку.

3.2 Вимоги до тестового застосунку

Мінімальні вимоги до апаратного забезпечення клієнта:

- Двохядерний процесор з тактовою частотою 2.0 GHz
- Оперативна пам’ять ємністю не менше 1024 Мб
- Монітор із розподільною здатністю 1280x960
- Інтернет підключення зі швидкістю 20 Мбіт/с

Рекомендовані вимоги до апаратного забезпечення клієнта:

- Чотирьохядерний процесор з тактовою частотою 2.0 GHz
- Оперативна пам’ять ємністю не менше 4096 Мб
- Монітор із розподільною здатністю не менше ніж 1920x1080
- Інтернет підключення зі швидкістю 50 Мбіт/с

Мінімальні вимоги до апаратного забезпечення сервера:

- Багатоядерний процесор з архітектурою x64
- Оперативна пам’ять ємністю не менше 8192 Мб
- Жорсткий диск 100 Гб
- Інтернет підключення зі швидкістю 100 Мбіт/с

Мінімальні вимоги до програмного забезпечення клієнта:

- Google Chrome 60+

Мінімальні вимоги до програмного забезпечення сервера:

- Операційна система Windows/Linux

Вимоги до користувача

Від користувачів вимагається базові знання роботи з веб-браузером.

Вимоги до надійності

При роботі з програмою не повинно виникати ніяких збоїв. Якщо клієнтський застосунок не може приєднатися до серверного, користувач повинен отримати повідомлення про це.

3.3 Хостінг застосунку на основі Microsoft Azure Web Sites

Згідно з тенденціями, усі важкі обчислення прийнято переносити до дата-центрів та хмарних сервісів. Саме тому було вирішено спроектувати систему у стилі клієнт - серверної архітектури. Для хостінгу нашого за стосунку був обран хотсінг Microsoft Azure Web Sites.

Microsoft Azure Web Sites - це платформа, основана на хмарних обчисленнях для розміщення веб-сайтів, створених та керованих корпорацією Майкрософт. Microsoft Azure Web Sites - це платформа як служба (PaaS), яка дозволяє публікувати веб-додатки, що працюють на різних платформах і написані різними мовами програмування (.NET, node.js, PHP, Python та Java). Microsoft Azure Web Sites стали доступними в першій версії попереднього перегляду в червні 2012 року, і офіційна версія ("Загальної доступності") була оголошена в червні 2013 року. Microsoft Azure Web Sites спочатку були названі Windows Azure Web Sites, але було перейменовано в зв’язку з ребрендінгу Azure у березні 2014 року.

Користувачі з підпискою Microsoft Azure можуть створювати веб-сайти та розгортати вміст і код на веб-сайтах. Microsoft Azure Web Sites підтримують інструменти для швидкого створення веб-сайту, який дозволяє користувачеві створювати порожній сайт або створювати сайт на основі одного з декількох доступних попередньо налаштованих зображень із галереї веб-сайтів.

У рамках створення веб-сайту URL-адресу сайту присвоюється субдомен azurewebsites.net. На різних рівнях підписки хостингу є можливість мати для веб-сайту декілька доменів. На деяких рівнях платників, користувач має додаткову можливість завантаження сертифіката SSL та налаштування його сайту для прив'язки до HTTPS.

Microsoft Azure Web Sites реалізовані як веб-сайти, які динамічно створюються за вимогою на серверах під керуванням Windows Server 2012 та IIS 8.0. Коли клієнт публікує запит на веб-сайт, Microsoft Azure Web Sites динамічно

підтримує сайт на одному з віртуальних машин Azure, вказуючи на його вміст, що зберігається в контейнерах Azure Storage. Віртуальні машини Azure розгортаються у групах під назвою «Stamps», які можуть містити сотні таких машин. Microsoft розгортає «Stamps» в своїх центрах обробки даних Azure у всьому світі і додає більше «Stamps», коли зростає попит.

Основна перевага чому був обран Microsoft Azure Web Sites це безкоштовне та швидке розгортання нашого затонку який був створений на основні Web Api. При реєстрації треба лише вказати номер валютної картки з якої знімається 1 долар, та дозволяє створити новий веб-сайт або розгорнути існуючий на хостінгу Microsoft Azure Web Sites.

3.3 Програмна реалізація

Програмна реалізація застосунку ділиться на дві частини: серверну і клієнтську. Клієнтська сторона застосунку в свою чергу ділиться на три складові: сторінка гріда тасків усіх користувачів, сторінка усіх існуючих тасків з можливістю фільтрування, сторінка користувачів та сторінка для редагування гріда тасків.

Серверний застосунок включає:

- базу даних;
- реалізацію запитів до бази даних.

Створена база даних зберігає наявну інформацію про всіх користувачів, усі таски та редаговану таблицю тасків, замовлень.

У клієлнтській частині при створенні нового застосунку Angular створую основний модуль який підключається до index.html, та дерево усіх елементів, до цього модуля підлкюченні усі залежності створенних елементнів які будуть використовуватись у застосунку. Данний лістинг показує залежності усіх елементів у головному модулі `tm-app.module`.

Лістинг *2*

Головний модуль tm-app.module

```
    import { BrowserModule } from '@angular/platform-
browser';
    import { RouterModule, Router } from '@angular/router';
    import { NgModule } from '@angular/core';
    import { FormsModule, ReactiveFormsModule } from
'@angular/forms';
    import { HttpModule, Http } from '@angular/http';
    import { BrowserAnimationsModule } from
'@angular/platform-browser/animations';
    import {
      DateAdapter,
      MatButtonModule,
      MatCheckboxModule,
      MatDatepickerModule,
      MatDialogModule,
      MatIconModule,
      MatInputModule,
      MatListModule,
      MatMenuModule,
      MatNativeDateModule,
      MatRippleModule,
      MatSelectModule,
      MatSnackBarModule,
      MatToolbarModule,
      MatTooltipModule
    } from '@angular/material';
    import { SqueezeBoxModule } from 'squeezebox';
    import { AvatarModule } from 'ngx-avatar';
    import { MomentModule } from 'angular2-moment';
    import { DragulaModule } from 'ng2-dragula';
    import { NgPipesModule } from 'ngx-pipes';
    import { ResourceModule, ResourceGlobalConfig } from
'ngx-resource';
    import { Ng2Webstorage } from 'ngx-webstorage';
    import {LoadingModule, ANIMATION_TYPES} from 'ngx-
loading';
```

```
    import { API_URL } from './tm-app.constants';
    import { TM_ROUTES } from './tm-app.routes';
    import { TmAppComponent } from './tm-app.component';
    import {TmCardComponent}from'./components/widgets/tm-
card';
    import{TmFooterComponent}from'./components/widgets/tm-
footer';
    import{TmGridComponent}from './components/widgets/tm-
grid';
    import { TmLayoutComponent } from
'./components/widgets/tm-layout';
    import { TmLogoComponent } from './components/widgets/tm-
logo';
    import { TmNavbarComponent } from
'./components/widgets/tm-navbar';
    import { TmUserScheduleComponent } from
'./components/widgets/tm-user-schedule';
    import { TmTasksListComponent } from
'./components/widgets/tm-tasks-list';
    import { TmUsersListComponent } from
'./components/widgets/tm-users-list';
    import { TmGridOptionsComponent } from
'./components/widgets/tm-grid-options';
    import { TmGridHeadersListComponent } from
'./components/widgets/tm-grid-headers-list';
    import { Tm404Component } from './components/pages/tm-
404';
    import { TmSignInComponent } from './components/pages/tm-
sign-in';
    import { TmSignUpComponent } from './components/pages/tm-
sign-up';
    import { TmScheduleComponent } from
'./components/pages/tm-schedule';
    import { TmSettingsComponent } from
'./components/pages/tm-settings';
    import { TmTasksComponent } from './components/pages/tm-
tasks';
    import { TmUsersComponent } from './components/pages/tm-
users';
```

```
    import { TmConfirmModalComponent } from
'./components/modals/tm-confirm';
    import { TmGridHeaderModalComponent } from
'./components/modals/tm-grid-header';
    import { TmTaskModalComponent } from
'./components/modals/tm-task';
    import { TmUserModalComponent } from
'./components/modals/tm-user';
    import { TmEditGridHeaderDirective } from
'./directives/tm-edit-grid-header';
    import { TmRemoveGridHeaderDirective } from
'./directives/tm-remove-grid-header';
    import { TmEditTaskDirective } from './directives/tm-
edit-task';
    import { TmRemoveTaskDirective } from './directives/tm-
remove-task';
    import { TmUnassignTaskDirective } from './directives/tm-
unassign-task';
    import { TmEditUserDirective } from './directives/tm-
edit-user';
    import { TmRemoveUserDirective } from './directives/tm-
remove-user';
    import { TmUniqueEmailValidatorDirective } from
'./directives/tm-unique-email-validator';
    import { TmSameValidatorDirective } from
'./directives/tm-same-validator';
    import { TmDateAdapter } from './services/tm-date-
adapter';
    import { TmSharedService } from './services/tm-shared';
    import { TmAuthService } from './services/tm-auth';
    import { TmGridOptionsService } from './services/tm-grid-
options';
    import { TmMainMenuService } from './services/tm-main-
menu';
    import { TmCommentsService } from './services/tm-
comments';
    import { TmPrioritiesService } from './services/tm-
priorities';
    import { TmTasksService } from './services/tm-tasks';
```

```
    import { TmUsersService, ITmUser } from './services/tm-
users';
        import { TmWeeksService } from './services/tm-weeks';
```

Кожен модуль повинен визначатися з декоратором @NgModule. Тож NgModule представляє функцію-декоратора, та приймає об'єкти, властивості якого описують метадані модуля (Лістинг 3).

Лістинг *3*

Метадані для модуля tm-app.module

```
    @NgModule({
      imports: [
        BrowserModule,
        RouterModule.forRoot(TM_ROUTES, { useHash: true }),
        FormsModule,
        ReactiveFormsModule,
        MatButtonModule,
        MatCheckboxModule,
        MatDatepickerModule,
        MatDialogModule,
        MatIconModule,
        MatInputModule,
        MatListModule,
        MatMenuModule,
        MatNativeDateModule,
        MatRippleModule,
        MatSelectModule,
        MatSnackBarModule,
        MatToolbarModule,
        MatTooltipModule,
        BrowserAnimationsModule,
        SqueezeBoxModule,
        AvatarModule,
        MomentModule,
        DragulaModule,
        NgPipesModule,
        ResourceModule.forRoot(),
        Ng2Webstorage.forRoot({ prefix: 'tm', separator: '.',
caseSensitive: true }),
```

```
    LoadingModule.forRoot({
      animationType: ANIMATION_TYPES.threeBounce,
      backdropBackgroundColour: 'rgba(0, 0, 0, .21)',
      backdropBorderRadius: '0px',
      primaryColour: '#595a5c',
      secondaryColour: '#00a94f',
      tertiaryColour: '#595a5c'
    })
  ],
  declarations : [
    TmAppComponent,
    TmCardComponent,
    TmFooterComponent,
    TmGridComponent,
    TmLayoutComponent,
    TmLogoComponent,
    TmNavbarComponent,
    TmUserScheduleComponent,
    TmTasksListComponent,
    TmUsersListComponent,
    TmGridOptionsComponent,
    TmGridHeadersListComponent,
    Tm404Component,
    TmSignInComponent,
    TmSignUpComponent,
    TmScheduleComponent,
    TmSettingsComponent,
    TmTasksComponent,
    TmUsersComponent,
    TmConfirmModalComponent,
    TmGridHeaderModalComponent,
    TmTaskModalComponent,
    TmUserModalComponent,
    TmEditGridHeaderDirective,
    TmRemoveGridHeaderDirective,
    TmEditTaskDirective,
    TmRemoveTaskDirective,
    TmUnassignTaskDirective,
    TmEditUserDirective,
```

```
    TmRemoveUserDirective,
    TmUniqueEmailValidatorDirective,
    TmSameValidatorDirective
  ],
  providers: [
    { provide: DateAdapter, useClass: TmDateAdapter },
    TmSharedService,
    TmAuthService,
    TmGridOptionsService,
    TmMainMenuService,
    TmCommentsService,
    TmPrioritiesService,
    TmTasksService,
    TmUsersService,
    TmWeeksService
  ],
  entryComponents: [
    TmConfirmModalComponent,
    TmGridHeaderModalComponent,
    TmTaskModalComponent,
    TmUserModalComponent
  ],
  bootstrap: [ TmAppComponent ]
})
```

Тож щоб показати компонентний підхід були створенні компоненти та дерективи, за допомогою яких був побудований наш застосунок. Дерево елементів застосунку (Рис. 3).

Рис. *3 Дерево елементів застосунку*

Першими компонентами застосунку є модульні вікна та як видно з попереднього рисунку їх 4.

Компонента *tm-confirm* модальне вікно яке створення для підтвердження зберігання, видалення, зміни даних.

Компонента *tm-grid-header* модальне вікно для редагування таблиці завдань.

Компонента *tm-task* модальне вікно для створення або редагування завдань.

Компонента *tm-user* модальне вікно для створення або редагування користувачів на сторінці Users для адміністратора.

Наступні компоненти знаходяться у папці pages, це компоненти для усіх сторінок завдань менеджеру.

Компонента *tm-404* сторінка яка показує що нема такого url та видає помилку.

Компонента *tm-schedule* сторінка Task Manager з таблицією завдань усіх користувачів.

Компонента *tm-settings* сторінка для редагування таблиці завдань.

Компонента *tm-sign-in* сторінка для авторизації.

Компонента *tm-sign-up* сторінка для реєстрації.

Компонента *tm-tasks* сторінка усіх завдань.

Компонента *tm-users* сторінка усіх користувачів.

Наступні компоненти знаходяться у папці widgets вони розширюють основні можливості html та написан основний функціонал застосунку.

Компонента *tm-card* яка створює новий html елемент акордіон, який дозволяє згортати та розгортати данні які записані у ньому.

Компонента *tm-footer* створює нижній колонтитул на всіх сторіках.

Компонента *tm-grid* створює редагуємо таблицю у якій створюється завдань та реалізує drag-and-drop функціонал для завдань у таблиці.

Компонента *tm-grid-header-list* компонента для виклику подій у таблиці завдань.

Компонента *tm-grid-options* яка відображає існуючі рядків та колонки таблиці завдань, та їх можливість редагувати або видаляти

Компонента *tm-logo* відображує логотип у верхньому меню.

Компонента *tm-navbar* створює меню для переходу між сторінками у застосунку.

Компонента *tm-tasks-list* яка відображає усі створенні завдання, та створений фільтр для більш легкого знаходження потрібного нам задачі.

Компонента *tm-user-schedule* відображує на сторінці Task Manager усіх користувачів, та завантажує їх завдання.

Компонента *tm-user-list* яка відображує список усіх користувачів для їх редагування чи видалення.

3.3 Інтерфейс застосунку

Робота застосунку починається з вікна авторизації (Рис. 4), яке дозволяє залогінітьcя вже існуючому користувачу.

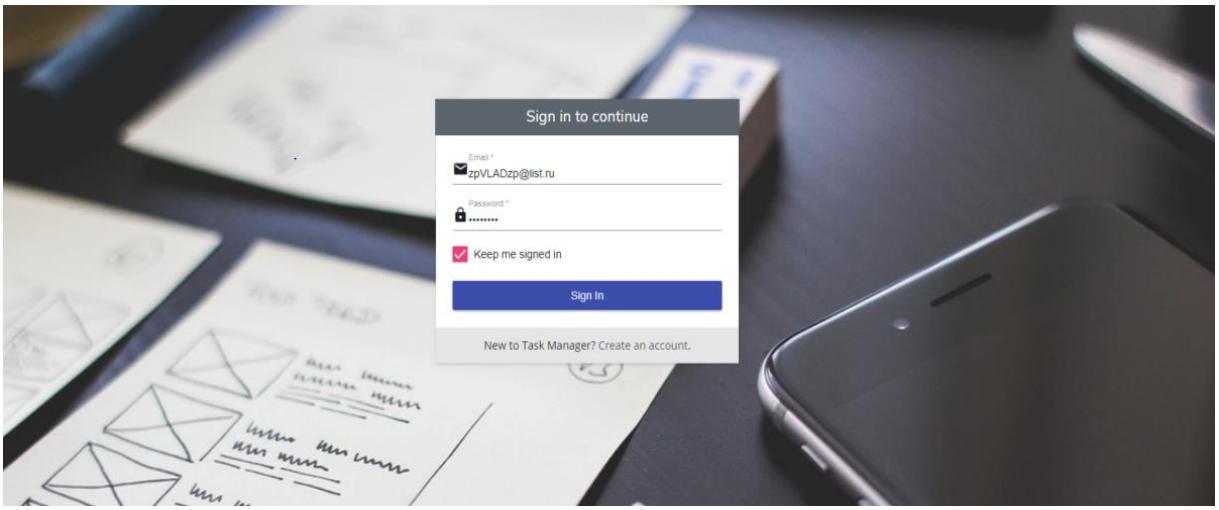

Рис. *4 Інтерфейс вікна авторизації*

Або зареєструватися новому користувачу (Рис. 5), у цьому вікні кожне поле перевіряєця валідаціє, та не дає зареєструватися якщо вводяться не валідні данні.

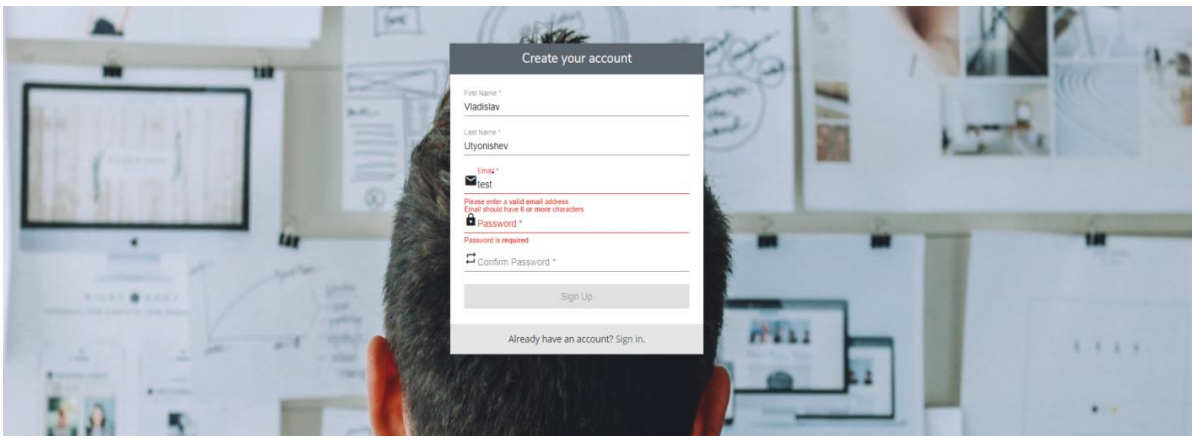

Рис. *5 Інтерфейс реєстрації нового користувачи*

Після авторизації користувач бачить першу сторінку застосунку "*Task Manager*" яка показує список усіх зареєстрованих користувачів та редагуєму таблицю з тасками (Рис. 6). Крім цього є можливість переносити таски за допомогою drag-and-drop интерфейсу, та данні таску(время на його виконнаня чи дата виконання) при такому редагуванні також змінюються та зберігаються у базі данних (Рис. 7).

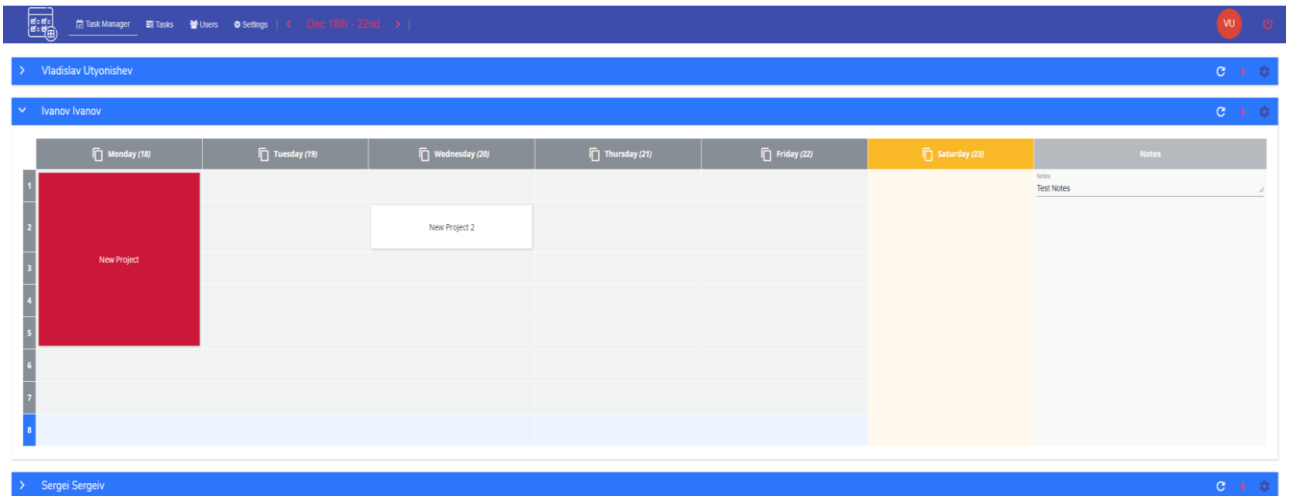

Рис. *6 Список таблиц тасків існуючих користувачів*

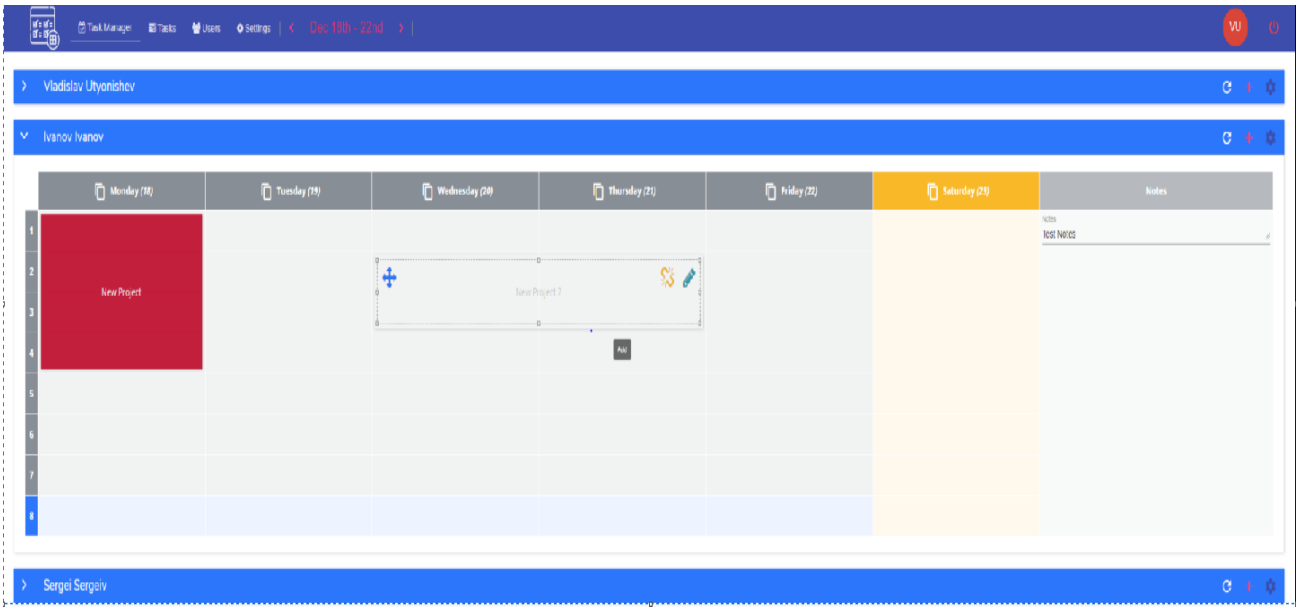

Рис. *7 Список таблиц тасків існуючих користувачів*

При натиснені на одну з клітин таблиці відкривається модальне вікно яке дозволяє створювати нові таски для користувача (Рис. 8). При натиснені на вже створенний таск показуєьться модальне вікно для редагування данного таску (Рис. 9). У поля дати виконаня таску є перевірка на валідність дат, та не дає створювати не коректні данні та зберігати їх.

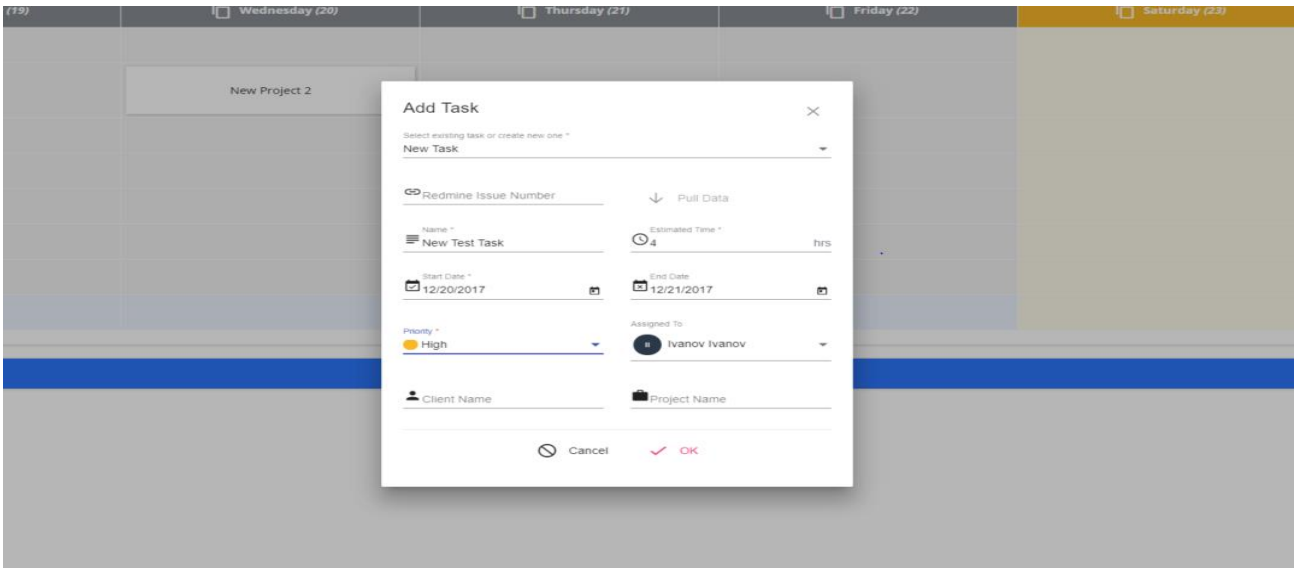

Рис. *8 Модальне вікно для створення нового таску*

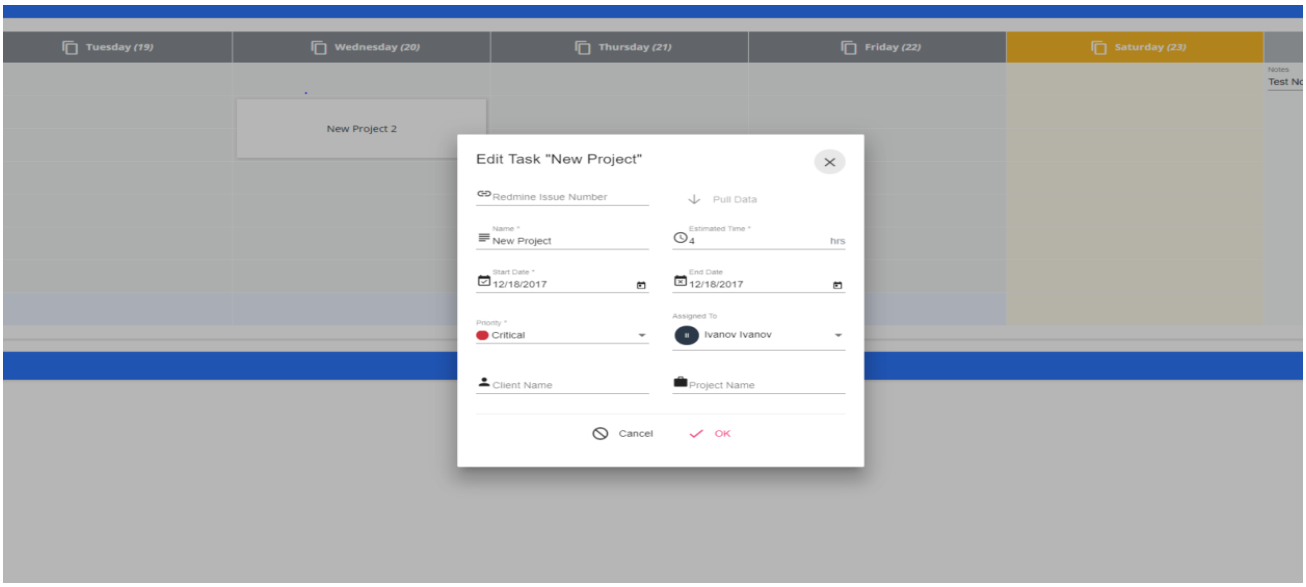

Рис. *9 Модальне вікно для редагування існуючого таску*

Друга сторінка застосунку це "*Tasks*" на цій сторінці показується список усіх тасків на кожну неділю (Рис. 10).

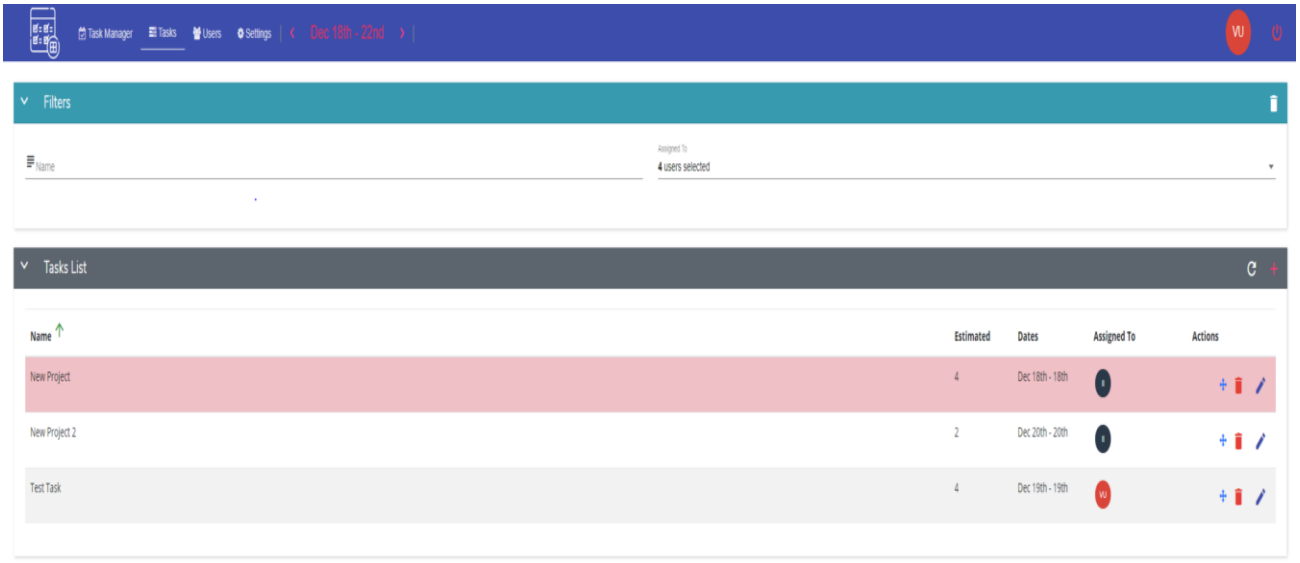

Рис. *10 Сторінка усіх тасків на кожну неділю*

Крім Цього є можливість фільтрувати таски для обранних користувачів, не назначенних тасків, та по назві таску (Рис. 11).

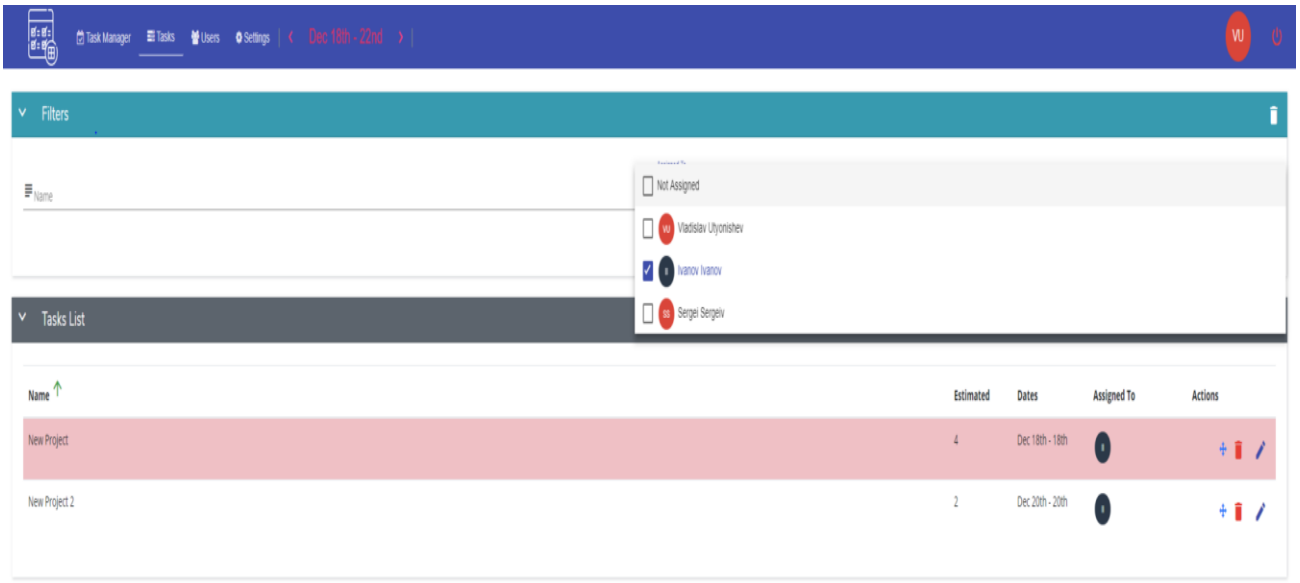

Рис. *11 Фільтр для списка тасків*

Третя сторінка застосунку це “*Users*” яка показує список існуючих користувачів з їх данними, та для адміна можливість видалення користувачів, чи змінни іх паролів (Рис. 12).

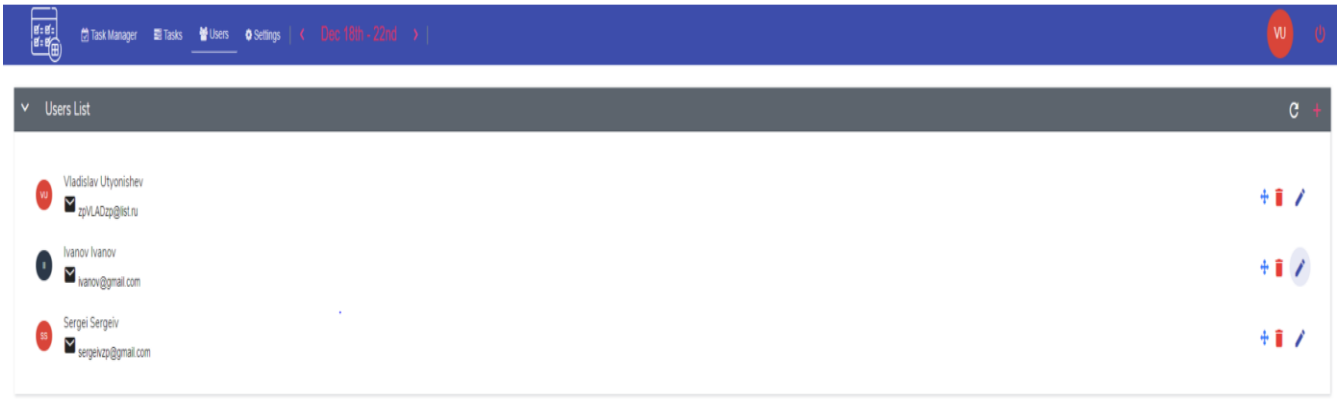

Рис. *12 Список усіх користувачів*

Остання сторінка за стосунку це “*Setting*”, ця сторінка дозволяю редагувати колонки та стовбці таблиць з тасками, створювати нові, чи видаляти існуючі.

За допомогою сторінки “*Setting*” можна створювати таблицю з тасками повністю на свій лад (Рис. 13).

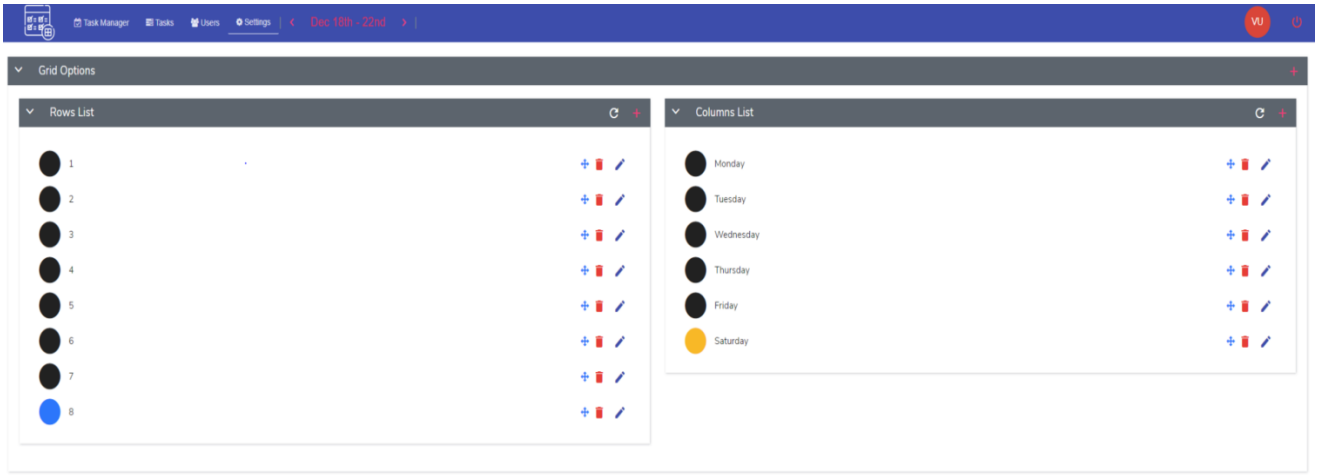

Рис. *13 Редагування таблиці тасків*

3.4 Висновки з розділу

При роботі над даним розділом були сформульовані вимоги до тестового застосунку та був проведений огляд та аналіз засобів для хостінгу веб-застосунку, який був використаний в результаті для Task Manager, а також його переваги і можливості.

Також був створений застосунок на основі Angular 5.0, який якнайкраще реалізує компонентний підхід, та має найкращу продуктивність серед усіх JavaScript фреймворків чи бібліотек. Був проведенний повний огляд інтерфейсу застосунку, його можливості та функціонал.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ ПОВТОРНОГО ВИКОРИСТАННЯ РОЗРОБЛЕНИХ КОМПОНЕНТІВ

4.1 Дослідження можливості повторного використання компонентів у створеному застосунку

Сучасні веб-додатки настільки ж складні, як і будь-які інші програмні додатки, і часто створюються декількома людьми, об'єднують зусилля для створення фінального продукту. В таких умовах, щоб підвищити ефективність, природно шукати правильні способи поділу роботи на дільниці з мінімальними перетинами між людьми і підсистемами. Впровадження компонентного підходу (в цілому) - це те, як зазвичай вирішується таке завдання. Будь-яка компонентна система повинна зменшувати загальну складність через надання ізоляції, або природних бар'єрів, що приховують складність одних систем від інших. Хороша ізоляція також полегшує повторне використання та впровадження сервісних парадигм.

Використовуючи Angular 5.0 який являється модульним та створений с компонентів які підключаються до кожної сторінки тільки коли нам потрібно дуже добре допомагає рішати такі проблеми. Крім цього він використовує TypeScript який добавляє для звичайного Javascript класи з модифікаціями(public, private, protected) що дозволяє управляти ізоляцією наших класів та компонентів.

У застосунку Task Manager компоненти мають слабку зв’язність між собою, що дозволяє використовувати повторно їх у нашому застосунку. Компонента tm-card, яка створює html елемент акордіон з любим текстом використовується майже на кожній сторінці, є як найкращим прикладом компоненти, яка повторно використовується та не потрібно кожного разу писати такий елемент (Лістинг 4).

Лістинг *4*

Тимплейт компоненти tm-card.component.html

```
    <div [ngClass]="getClasses()" [ngStyle]="getStyles()"
class="card-wrap">
      <div class="card-header pointer" matRipple *ngIf="title
&& collapsible"
           [ngClass]="getHeaderClasses()"
[ngStyle]="getHeaderStyles()"
           (click)="collapseToggle()">
        <div class="pull-right">
          <ng-content select="[card-content=actions]"></ng-
content>
        </div>
        <h5>
          <i *ngIf="collapsible" [class.rotate-
90]="!isCollapsed" class="fa fa-fw fa-lg fa-angle-right">
          </i>
          {{ title }}
        </h5>
      </div>
      <div class="card-header" *ngIf="title && !collapsible"
           [ngClass]="getHeaderClasses()">
        <div class="pull-right">
          <ng-content select="[card-content=actions]"></ng-
content>
        </div>
        <h5>
          <i *ngIf="collapsible" class="fa fa-fw"></i>
          {{ title }}
```

```
        </h5>
      </div>
      <div class="card-content" *ngIf="!collapsible ||
!isCollapsed"
           [ngClass]="getContentClasses()"
[ngStyle]="getContentStyles()"

[@collapseContentAnimation]="getCollapseContentAnimationState(
)">
        <ng-content select="[card-content=body]"></ng-
content>
      </div>
    </div>
```

Та компонента *tm-card.component.ts*, яка розширую можливості *tm-card.component.html* темплейту (Лістинг 4).

Лістинг *4*
Компонента tm-card.component.ts

```
    import { Component, Input, trigger, transition, style,
animate } from '@angular/core';

    @Component({
      selector: 'tm-card',
      templateUrl: 'tm-card.component.html',
      styleUrls: ['tm-card.component.scss'],
      animations: [
        trigger(
          'collapseContentAnimation', [
```

```
            transition('void => expanded', [
              style({ height: '0', paddingTop: '0',
paddingBottom: '0', opacity: 0 }),
              animate(500, style({ height: '*', paddingTop:
'*', paddingBottom: '*', opacity: 1 }))
            ]),
            transition('expanded => void', [
              style({ height: '*', paddingTop: '*',
paddingBottom: '*', opacity: 1 }),
              animate(500, style({ height: '0', paddingTop:
'0', paddingBottom: '0', opacity: 0 }))
            ])
          ]
        )
      ]
    })

      public getContentClasses() {
        return {
          'text-right': this.align === 'right',
          'text-center': this.align === 'center',
          [this.contentCustomClass]: true
        };
      }

      public getStyles() {
        return {
          'background-color': this.customBgColor,
          'background-image': `url('${this.bgImage}')`,
          'border-color': this.customBgColor,
```

```
      'color': this.customColor
    };
  }

  public getHeaderStyles() {
    return {
      'background-color': this.headerCustomBgColor,
      'color': this.headerCustomColor
    };
  }

  public getContentStyles() {
    return {
      padding: this.indents
    };
  }

  public collapseToggle() {
    this.isCollapsed = !this.isCollapsed;
  }

  public getCollapseContentAnimationState() {
    if (this.collapsible && !this.isCollapsed) {
      return 'expanded';
    }
  }
}
```

Крім цього у Angular є елементи такі як дерективи, що дозволяють розширювати можливості нашого коду, та повторно використовувати частини деякого

коду. Так у Task Manager створена деректива *tm-unique-email-validator* яка використовується на сторінках реєстрації та авторизації, що підключається через залежності та перевіряє валідацію полів імейлу (Лістинг 5).

Лістинг *5*

Деректива tm-unique-email-validator.ts

```
    @Directive({
      selector: '[tm-unique-email-validator][formControlName], [tm-unique-email-validator][ngModel], [tmUniqueEmailValidator][formControlName], [tmUniqueEmailValidator][ngModel]',
      providers: [
        {
          provide: NG_ASYNC_VALIDATORS,
          useExisting: forwardRef(() => TmUniqueEmailValidatorDirective),
          multi: true
        }
      ]
    })
    export class TmUniqueEmailValidatorDirective implements Validator {
      @Input() public userId?: number;
      private _uniqueEmailValidateTimer: number;
      constructor(private _tmUsersService: TmUsersService) {
      }
      public validate(abstractControl: AbstractControl): Promise<{[key: string]: any}> {
```

```
        return
this.validateUniqueEmail(abstractControl.value);
      }
      public validateUniqueEmail(email: string) {
        return new Promise(resolve => {
          if (this._uniqueEmailValidateTimer) {
            clearTimeout(this._uniqueEmailValidateTimer);
          }
          const timer: any = setTimeout(() => {
            this._tmUsersService.checkEmailIsUnique({   email:
email, userId: this.userId })
              .$observable.toPromise()
              .then(()  =>  resolve(null),  ()  =>  resolve({
emailNotUnique: true }));
          }, 700);
          this._uniqueEmailValidateTimer = timer as number;
        })
      }
    }
```

4.2 Висновки з розділу

У даному розділі було досліджено повторне використання компонентів у створенному застосунку Task Manager, що зменшить час на розробку, та можливе їх впровадження у ішних застосунках.

РОЗДІЛ 5 ОХОРОНА ПРАЦІ ТА ТЕХНОГЕННА БЕЗПЕКА

5.1 Характеристика потенційних небезпечних та шкідливих виробничих факторів

У зв'язку з впровадженням нових технологічних процесів і інтенсифікацією існуючих спостерігається посилення впливу різних виробничих факторів, таких як шум, вібрація, різні види електромагнітних випромінювань, ультразвук, пил, органічні і неорганічні сполуки. Тому проблема профілактики професійних захворювань і гострих отруєнь набуває особливого значення.

В Конституції України виділені спеціальні статті, які стверджують рівноправність на працю, охорону праці та охорону навколишнього середовища.

Згідно ст. 22 Закону України "Про охорону праці" одним з найбільш важливих підрозділів колективного договору повинні бути комплексні заходи щодо досягнення встановлених нормативів безпеки, гігієни праці та виробничого середовища, підвищення існуючого рівня охорони праці, запобігання випадкам виробничого травматизму, професійних захворювань і аварій.

Запорука успіху в розвитку охорони праці, в рішенні проблеми гігієни праці, фізіології праці, в забезпеченні безпечних умов праці, ліквідації професійних захворювань і виробничого травматизму, профілактики отруєнь та шкідливого впливу на працюючих фізичних факторів виробничого середовища полягає в турботі уряду щодо охорони здоров'я працівників.

Законодавчо-правовими актами з охорони праці та навколишнього середовища є: Конституція України, Законодавства України про охорону здоров'я, Основи земельного законодавства України, Закон про трудові колективи, Закон України про охорону праці та ін.

Фактори, що впливають на продуктивність праці людини в процесі трудової діяльності, можна поділити наступним чином:

Психофізіологічні - фізична, нервово-психічне навантаження, монотонність, ритм праці. Вони диктуються конкретним змістом трудової діяльності, характером даного виду праці.

Санітарно-гігієнічні умови - мікроклімат, стан повітря, шум, освітлення та інше, визначаються зовнішньої виробничим середовищем та санітарно-побутовим обслуговуванням.

Соціально-психологічним - характеризують взаємовідносини в трудовому колективі і створюють відповідний психологічний настрой.

Завданням організації є - приведення перерахованих факторів в оптимальний стан для підвищення працездатності і зміцнення здоров'я службовців.

Людини працюючого на ПК постійно або періодично діють наступні небезпечні і шкідливі фактори:

- невідповідність нормам параметрів мікроклімату;
- підвищений рівень статичної електрики при неправильно спроектованої робочій зоні;
- небезпечний рівень напруги в електричному ланцюзі, замикання якого може відбутися через тіло людини;
- широкий спектр випромінювання від дисплея, який включає рентгенівську, ультрафіолетову і інфрачервону області, а так само широкий діапазон електромагнітних випромінювань інших частот;
- підвищений рівень електромагнітних випромінювань;
- відсутність або нестача природного світла;
- недостатня освітленість робочої зони;
- небезпека виникнення пожежі.

При роботі на ПК органи зору користувача витримує більше навантаження з одночасним постійним напруженим характером праці, що призводить до порушення функціонального стану зорового аналізатора і центральної нервової системи.

На ЕПТ часто накопичується електростатичний заряд. У момент включення напруженість поля миттєво зростає до максимуму, а потім поступово зменшується до стабільного рівня. Після виключення монітора реєструють негативну напруженість поля, яка поступово зменшується.

Деякі комп'ютери та мережеве обладнання є потенційними джерелами цілого ряду звукових коливань як чутного, так і ультразвукового діапазону. Цей шум справляє негативний вплив на функціональний стан користувачів. Відомо, що шум несприятливий для людини, особливо при тривалому впливі.

5.2 Заходи з поліпшення умов праці

Для створення і автоматичної підтримки в лабораторії оптимальних значень температури, вологості і швидкості руху повітря в холодну пору року використовується водяне опалення, в теплу пору року застосовується кондиціонування повітря. Кондиціонер є вентиляційною установкою, яка за допомогою приладів автоматичного регулювання підтримує в приміщенні задані параметри повітряного середовища [20].

У зв'язку з тим, що природне освітлення лабораторії здійснюється через віконні отвори і є дуже слабким, на робочому місці має застосовуватися також штучне освітлення. Штучне освітлення створюють електричним джерелом світла, яке включають в міру необхідності, регулюють інтенсивність світлового потоку і його спрямованість.

Як заходи щодо зниження рівня шуму можна запропонувати наступне:

- облицювання стелі та стін звукопоглинальним матеріалом (знижує шум на 6-8 дБ);
- установка звукопоглинального кожуха;
- установка в лабораторії обладнання, що виробляє мінімальний шум;
- раціональне планування приміщення.

Для того щоб знизити рівень шуму до оптимального значення, в лабораторії рекомендується застосовувати звукопоглинальне покриття стін.

Ослаблення шуму повітро(газо)проводів досягають плавністю руху повітряного потоку, плавними переходами в місцях зміни напрямку трубопроводу, застосуванням глушників.

5.3 Виробнича санітарія

Оптимальні норми температури повітря в робочій зоні виробничих приміщень в холодний і перехідний періоди року при I категорії робіт (легкої) - 20 ... 23^{o}C, відносна вологість - 60 ... 40%, швидкість руху повітря - 0,2 м / сек. У теплий період року температура - 22 ... 25 ° С, відносна вологість - 60 ... 40%, швидкість руху повітря - 0,2 м / сек.

Для забезпечення вищевказаних параметрів передбачено застосування вентиляції - природної (аерації і провітрювання), механічною (загально обмінної, місцевої припливної та витяжної), кондиціювання повітря.

У приміщенні забезпечений приплив свіжого повітря, кількість якого визначається техніко-економічним розрахунком і вибором схеми системи вентиляції. Мінімальна витрата повітря визначається з розрахунку 50 ... 60 м3 / г на одну людину, але не менше двократного повітрообміну в годину.

Системи опалення та системи кондиціонування встановлені так, щоб ні теплий, ні холодне повітря не направлявся на людей. На виробництві створено

динамічний клімат з певними перепадами показників. Температура повітря в поверхні підлоги і на рівні голови не відрізняється більш ніж на 5 градусів. У виробничих приміщеннях крім природної вентиляції передбачають приточно-витяжна вентиляція.

Розрахунок характеристики вентиляційної системи

Основним параметром, що визначає характеристики вентиляційної системи, є кратність обміну, тобто скільки разів за годину зміниться повітря в приміщенні.

Зробимо розрахунок для приміщення:

- V_1 - об'єм повітря необхідний для обміну;
- V_2 - об'єм робочого приміщення.

Для розрахунку приймемо такі розміри робочого приміщення:

- довжина $B = 7{,}35$ м;
- ширина $A = 4{,}9$ м;
- висота $H = 4{,}2$ м.

Відповідно обсяг приміщення дорівнює:

$$V_2 = A * B * H = 151{,}263 \text{ м}^3 \quad (5.1)$$

Необхідний для обміну об'єм повітря V1вент визначимо виходячи з рівняння теплового балансу:

$$V_1 * C * (t_{ух} - t_{пр}) * Y = 3600 * Q_{над} \quad (5.2)$$

де $Q_{над}$ - надлишкова теплота (Вт)

$C = 1000$ - питома теплопровідність повітря (Дж / КГК)

$Y = 1{,}2$ - щільність повітря (мг / см);

$t_{ух}$ - температура повітря, що йде, ^{o}C;

$t_{пр}$ - температура повітря приходить ^{o}C;

Температура повітря, що йде визначається за формулою:

$$t_{ух} = t_{рм} + (H - 2) * t \quad (5.3)$$

де t = 1-5 градусів - перевищення t на 1 м висоти приміщення;

$t_{рм}$ = 25 градусів - температура на робочому місці;

H = 4,2 м - висота приміщення;

$t_{пр}$ = 18 ^{o}C.

Тоді $t_{ух}$ = 25 + (4,2 - 2) 2 = 29,4

далі:

$$Q_{над} = Q_{над.1} + Q_{над.2} + Q_{над.3} \quad (5.4)$$

де Qнад.1 - надлишок тепла від електрообладнання та освітлення.

$$Q_{над.1} = E * P \quad (5.5)$$

де E - коефіцієнт втрат електроенергії на тепловідвід (E = 0,55 для освітлення);

P - потужність, P = 40 Вт * 15 = 600 Вт

Тоді $Q_{над.1}$ = 0,55 * 600 = 330 Вт

$Q_{над.2}$ - теплопостачання від сонячної радіації:

$$Q_{над.2} = m * S * k * Q_{с} \quad (5.6)$$

де m - число вікон, приймемо m = 4;

S - площа вікна, $S = 2,3 * 2 = 4,6$ м^2;

k - коефіцієнт, що враховує скління. Для подвійного склінням $k = 0,6$;

$Q_с = 127$ Вт / м - тепло надходить від вікон.

Тоді $Q_{над.2} = 4,6 * 4 * 0,6 * 127 = 1402$ Вт

$Q_{над.3}$ - тепловиділення людей

$$Q_{над.3} = n * q \qquad (5.7)$$

де $q = 80$ Вт / чол., n - число людей, наприклад, $n = 15$.

$Q_{над.3} = 15 * 80 = 1200$ Вт

В результаті отримуємо $Q_{над} = 330 + 1402 + 1200 = 2932$ Вт

Тоді з рівняння теплового балансу слід:

$V_1 = 3600 * 2932/1000 * (29,4 - 18) = 926$ м^3.

Розрахунок кратності:

Кратність = $V_1 / V_2 = 6.12$

Оптимальним варіантом є кондиціювання повітря, тобто автоматична підтримка його стану в приміщенні відповідно до певних вимог (задана температура, вологість, рухливість повітря) незалежно від зміни стану зовнішнього повітря і умов в самому приміщенні.

При невеликій забрудненості зовнішнього повітря кондиціювання приміщень здійснюється зі змінними витратами зовнішнього повітря.

З метою підтримки заданих параметрів мікроклімату в машинному залі використовується автоматичне регулювання УКВ. У офісі, як правило, застосовують одностороннє бічне природне освітлення. Причому світло проєми з метою зменшення сонячної інсоляції влаштовані з північної, північно-східної або північно-західною орієнтацією. Робочі місця програмістів розташовані подалі від вікон і таким чином, що віконні прорізи перебувають збоку. Якщо екран дисплея звернений до віконного отвору, необхідні спеціальні екрануючі пристрої.

Вікна рекомендується забезпечувати світло розсіювальні шторами, регульованими жалюзі або сонцезахисної плівкою з металізованим покриттям.

У тих випадках, коли одного природного освітлення недостатньо, влаштовують суміщене освітлення. При цьому додаткове штучне освітлення застосовують не тільки в темний, але і в світлий час доби.

Найбільш прийнятними для приміщень офісу, є люмінесцентні лампи ЛБ (білого світла) і ЛТБ (тепло-білого світла) потужністю 20, 40 або 80 Вт.

Для виключення засвічення екранів дисплеїв прямими світловими потоками світильники загального освітлення мають в своєму розпорядженні збоку від робочого місця, паралельно лінії зору програміста і стіні з вікнами.

Для робочих місць програмістів характерна наявність декількох видів шумів. Технічні засоби створюють механічний шум, установки кондиціювання, вентиляційне обладнання - аеродинамічний, перетворювачі напруги - електромагнітний.

5.4 Електробезпека

У приміщенні відсутні травмонебезпечні місця, відкритих струмоведучих частин немає. Від розподільного щитка проводка (стандартне однофазну напругу 220 В, 50 Гц) йде по стінах до комутаційних панелях, звідки вона розходиться до робочих місць, всі кабелі зібрані в пучки і додатково захищені коробами, які виступають як додатковий шар ізоляції. Все обладнання, з яким працюють люди розраховані на низьку напругу, і відносяться за ступенем небезпеки ураження електричним струмом в категорії «без підвищеної небезпеки», згідно з ПУЕ. Їх живлення здійснюється від стандартних розеток з третім контактом для примусового заземлення.

Характеристика електромережі: змінна однофазне напругу 220 В, 50 Гц.

Для електробезпеки досить буде вжити таких заходів:

- Налагодження та ремонт електромережі повинні проводити тільки фахівці-електрики, а ремонт пристроїв - гарантійні майстерні виробників,
- При виявленні несправності обладнання необхідно зупиняти роботу на ньому і не продовжувати роботу, поки несправність не усунуто,
- Регулярно перевіряти стан електрокабелів харчування і цілісність ізоляції повідомити всіх працівників про правила роботи з довіреною їм технікою і перевіряти дотримання ними вищевказаних правил.

5.5 Пожежна безпека

Як відомо, пожежа може виникнути при взаємодії горючих речовин, окислювача і джерел запалювання. У приміщеннях присутні всі 3 основні фактори, необхідні для виникнення пожежі.

Для відводу теплоти у офісі постійно діє потужна система кондиціювання. Тому кисень, як окислювач процесів горіння, є в будь-якій точці приміщення.

Джерелами запалювання на ВЦ можуть виявитися електронні схеми ЕОМ, прилади, застосовувані для технічного обслуговування, пристрої електроживлення, кондиціонери повітря, де в результаті різних порушень утворюються перегріті елементи, електричні іскри і дуги, здатні викликати загоряння горючих матеріалів.

Для більшості технологічних процесів у офісі встановлена категорія пожежної небезпеки В (у виробництві звертаються тверді спаленні речовини і матеріали).

На випадок пожежі передбачені шляхи евакуації працівників — проходи, проїзди, евакуаційні виходи.

Коридори і проходи, призначені для евакуації, мають можливо меншу довжину і мінімальна кількість поворотів. На всьому протязі проходу немає порогів або проміжних ступенів.

Важливу роль в забезпеченні безпечного виходу людей грає протидимна захист евакуаційних шляхів. У будинках заввишки до 9 поверхів незадимлюваність сходових клітин на час евакуації досягається їх ізоляцією від підвалів, горищ та поверхів. Для цього влаштовані відокремлені входи в підвали, вхід на сходову клітку з поверхів здійснюють через тамбур - шлюз з підпором повітря, відокремлюють горища від сходових клітин перекриттями з негорючих матеріалів.

Для відводу надлишкової теплоти служать системи вентиляції і кондиціювання повітря. Однак вони представляють додаткову пожежну небезпеку, так як, з одного боку, вони забезпечують подачу кисню - окислювача в усі приміщення, а з іншого - при виникненні пожежі поширюють вогонь і продукти горіння по всіх приміщеннях і пристроїв, з якими пов'язані повітроводи.

Для тепло- і звуко- поглинаючої ізоляції систем вентиляції застосовують такі негорючі матеріали, як мати з мінеральної вати, скловолокна, стіліта.

Напруга до електроустановок подається по кабельних лініях, які представляють особливу пожежну небезпеку.

Однією з найбільш важливих завдань пожежної профілактики є захист будівельних конструкцій від руйнування і забезпечення їх достатньої міцності в умовах впливу високих температур при пожежі.

У машинних залах кабельні лінії прокладають під технологічними знімними полами, які виконують з негорючих матеріалів з межею вогнестійкості не менше 0.5ч.

Протидимний захист будівель, спрямована на незадимлюваність евакуаційних шляхів, окремих приміщень і видалення продуктів горіння в певному напрямку, є першочерговим завданням пожежної профілактики.

Для виявлення початкової стадії загоряння, швидкого і точного оповіщення служби пожежної охорони про час і місце виникнення пожежі встановлені системи автоматичної пожежної сигналізації (АПС). Вони можуть самостійно пускати в хід установки пожежогасіння, коли пожежа не досяг великих розмірів. Системи АПС складаються з пожежних сповіщувачів, ліній зв'язку та приймних пультів (станцій).

Для подачі сигналу про пожежу встановлені пожежні сповіщувачі. Залежно від контрольованого фактора, вони діляться на теплові, димові, світлові, комбіновані.

5.6 Висновки з розділу

В даному розділі були розглянуті питання і заходи з техніки безпеки, а також враховані небезпечні і шкідливі фактори з метою створення безпечної експлуатації робочих місць в обраному приміщенні.

ВИСНОВКИ

В рамках випускної магістерської роботи була виконана наступна робота:

1. Виконано огляд компонентного підходу та його розвиток у вебі. Оглянуті сучасні JavaScript фреймворки та бібліотеки, проаналізовані їх переваги та недоліки, та обрано Angular для використання у розробці тестового застосунку.
2. Оглянуто структуру Angular 5.0, створення компонентів з його допомогою, та переваги у компонентному підході.
3. Досліджено популярні засоби розробки тестового застосунку, для серверної частини, та обрано C# ASP.NET Web API.
4. Сформульовано функціональні вимоги до тестового застосунку Task Manager з використанням компонентного підходу, та повторного використання компонентів.
5. Створено Task Manager, використовуючи обрані технології. Оглянуто створені компоненти, та розроблений інтерфейс застосунку.
6. Досліджено повторне використання компонентів у розробленому застосунку, що дозволяє зберегти час при розробці.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Компонентний підхід [Електронний ресурс]. – режим доступу https://en.wikipedia.org/wiki/Component-based_software_engineering

2. Nir Kaufman, Thierry Templier Angular 2 Components. – Packt Publishing, 2016. – 168 p.

3. Chris Buckett Web Components in Action version 2 Packt Publishing, – 32p.

4. Внедрение компонентого подхода в вебе: обзор веб-компонентов [Електронний ресурс]. – режим доступу https://habrahabr.ru/company/microsoft/blog/264791/

5. Технологии программирования. Компонентный подход. В.В Кулямин – 65 с.

6. Michele Bertoli React Design Patterns and Best Practice – 332p.

7. AngularJS – MVC архітектура [Електронний ресурс]. – режим доступу http://thewebland.net/development/javascript/angularjs/angularjs-mvc/

8. AngularJS Tutorial: A Comprehensive 10000 World Guide [Електронний ресурс]. – режим доступу https://www.airpair.com/angularjs

9. Стефанов Стоян React.js. Быстрый старт. — СПб.: Питер, 2017. — 304 с.

10. Компонентный подход к созданию приложения с помощью AngularJS 1.5 & 2.0 [Електронний ресурс]. – режим доступу http://bogdanov-blog.ru/komponentnyj-podhod-k-sozdaniyu-prilozheniya-s-pomoshhyu-angularjs-1-5-2-0/

11. Getting Started With React.js [Електронний ресурс]. – режим доступу https://code.tutsplus.com/courses/getting-started-with-reactjs

12. Matthew Scarpino Building Web Components with TypeScript and Angular 4. – Quiller Technologies LLC, 2017

13. Nir Kaufman, Thierry Templier Angular 2 Components. – Packt Publishing, 2016. – 168 p.

14. Adam Freeman Pro Angular. Second Edition. – Apress, 2017. – 788 p.
15. Olga Filipova Learning Vue.js 2. – Packt Publishing, 2016. – 382 p.
16. Vue.js – реактивный фронтенд фреймворк для людей [Електронний ресурс]. – режим доступу https://komelin.com/ru/articles/vuejs-reaktivnyy-frontend-freymvork-dlya-lyudey
17. Загальні відомості про Vue.js [Електронний ресурс]. – режим доступу https://ru.vuejs.org/v2/guide/index.html
18. Sql преимущества и недостатки [Електронний ресурс]. – режим доступу https://www.softbusiness.net/Default.aspx?id=29
19. Навакатікян О., Кальниш В., Стрюков С. Охорона праці користувачів комп'ютерних відеодісплейних терміналів./ О. Навакатікян. – К., 1997. – 400с.
20. Кожемякін Г., Новокщонова О. Методичні вказівки до виконання розділу дипломних проектів (робіт)./ Г. Кожемякін, О. Новощокова – Запоріжжя ЗДІА 2011. – 22.с
21. Жидецький В. Охорона праці користувачів комп'ютерів./ В. Жидецький. – Львів: Афіша, 2000 – 170 с.