МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки та програмної інженерії
Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
Катерина Нестеренко

“____”_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСНИКА ОСВІТНЬОГО СТУПЕНЯ
“ МАГІСТРА”

Тема: “Методика та застосунок для підвищення ефективності тестування програмних систем”

Виконавець: Мількевич Вікторія Володимирівна

Керівник: к.т.н. доцент Олександр Андрійович Ткаченко

Нормоконтролер: Гололобов Дмитро Олександрович ст.в

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки та програмної інженерії
Кафедра інженерії програмного забезпечення
Освітній ступінь магістр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Програмне забезпечення систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Катерина Нестеренко
"___" _____ 2023 р

ЗАВДАННЯ
на виконання дипломного проекту студента
Мількевич Вікторії Володимирівни

1. Тема проекту: «Методика та застосунок для підвищення ефективності тестування програмних систем»
затверджена наказом ректора від 29.09.2023 р. № 1994/ст

2. Термін виконання проекту: з 02.10.2023 р. до 31.12.2023 р.

3. Вихідні данні до проекту: розробка програмного продукту на основі .NET 6.0 та React.JS.

4. Зміст пояснювальної записки:
   1. Аналіз предметної області та дослідження ідеї.
   2. Опис вимог та методика підвищення ефективності тестування програмних систем
   3. Розробка та опис структури інформаційної системи.
   4. Оцінка та тестування програмного застосунку.

5. Перелік обов'язкових слайдів презентації:
   1. Мета роботи.
   2. Вимоги до системи.
   3. Архітектура системи.
   4. Робота системи.
   5. Майбутній розвиток.

6. Календарний план-графік

№ пор	Завдання	Термін виконання	Відмітка про виконання
1.	Оформлення перших 2-х сторінок пояснювальної записки (ПЗ) – титулка, завдання.	02.10.23 – 08.10.23	
2.	Ознайомлення з постановкою задачі та вивчення літератури. Написання 1 розділу, представлення керівнику. 1-ий нормо-контроль.	09.10.23– 15.10.23	
3.	Підготовка остаточного плану та узгодження його з керівником	16.10.23– 22.10.23	
4.	Обговорення та редагування першого розділу з керівником, підготовка до написання другого розділу	23.10.23– 29.10.23	
5.	Написання другого розділу, представлення керівнику, підготовка до написання третього розділу	30.10.23 – 05.11.23	
6.	Написання 3 розділу, представлення керівнику, форматування та виправлення існуючого тексту.	06.11.23 – 19.11.23	
7.	Загальне редагування та друк пояснювальної записки, графічного матеріалу. Отримання відгуку керівника, рецензії та заповнення листа про доброчесність. Розробка доповіді	20.11.23 – 03.12.23	
8.	Завершення написання ПЗ. Проходження нормо-контролю. Друк ПЗ. Отримання відгуку керівника. Підготовка презентації та доповіді на перед захист.	04.12.23 - 10.12.23	
9.	Передзахист каліф. Роботи. Отримання рецензії	11.12.23 - 17.12.23	
10.	Підготовка документів до захисту та здача їх секретарю ДЕК	18.12.23 - 24.12.23	
11.	Захист дипломної роботи перед ЕК	25.12.23 - 31.12.23	

Дата видачі завдання 02.10.2023

Керівник: к.т.н. доцент Олександр ТКАЧЕНКО

Завдання прийняв до виконання: Вікторія МІЛЬКЕВИЧ

Дата 02.10.2023

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Методика та застосунок для підвищення ефективності тестування програмних систем»: 103 с., 9 рис., 0 табл., 36 інформаційних джерел.

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, REACT.JS, .NET, C#

Об'єкт дослідження – методи та прийоми для підвищення ефективності тестування програмних систем з фокусом на вдосконаленні процесів управління життєвим циклом розробки програмного забезпечення.

Мета дипломної роботи – збільшення якості та надійності програмних систем через удосконалення методів тестування, що призведе до зменшення кількості виявлених помилок під час експлуатації програмного забезпечення.

Метод дослідження - методи управління процесами життєвого циклу розробки програмного забезпечення у гнучких методологіях, аналіз предметної області.

В ході проведення дослідження були розглянуті наступні аспекти: 1) існуючі практики тестування програмного забезпечення; 2) існуючі системи, що вирішують проблему тестування програмного забезпечення.

Результат даного дослідження можуть знайти застосування в індустрії розробки програмного забезпечення, сприяючи підвищенню якості продуктів та зменшенню витрат на їхнє подальше удосконалення.

Розробка та дослідження проводилися під управлінням ОС Windows 10. Розробка програми проводилася на платформі .NET 6.0, на мовах програмування C# та JavaScript.

ABSTRACT

Explanatory note to the thesis "Methodology and application for increasing the effectiveness of testing software systems": 103 pages, 9 figures, 0 tables, 36 information sources.

SOFTWARE TESTING, TEST AUTOMATION, REACT.JS, .NET, C#

Object of the research - methods and techniques for increasing the effectiveness of testing software systems with a focus on improving the processes of managing the software development life cycle.

Purpose of research is to increase the quality and reliability of software systems through the improvement of testing methods, which will lead to a decrease in the number of detected errors during the operation of the software.

Research method - methods of managing the life cycle processes of software development in flexible methodologies, analysis of the subject area.

During the research, the following aspects were considered: 1) existing software testing practices; 2) existing systems that solve the problem of software testing.

Result of this research can be used in the software development industry, contributing to the improvement of product quality and the reduction of costs for their further improvement.

The development and research was carried out under Windows 10 operating system. The program was developed on the .NET 6.0 platform, in the C# and JavaScript programming languages.

ЗМІСТ

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ

API - Appliccation Programming Interface

UI – User Interface

UX – User eXperience

HTTP – HyperText Transfer Protocol

XSS – Cross Site Scripting

ВСТУП

Головна мета даної дипломної роботи полягає в розробці методики та застосунку для ефективного оповіщення сигналу тривоги та формування маршруту до укриття в умовах реальної загрози, що виникає внаслідок війни на території України. Сучасна обстановка визначає необхідність створення надійних та ефективних систем, призначених для захисту населення у випадку надзвичайних ситуацій.

Наразі існує багато систем, які працюють на оповіщення населення про небезпеку, проте з різних причин вони можуть не працювати з деякими категоріями населення, що і вплинуло на створення та розвиток даної ідеї.

Дослідження предметної області в даній темі передбачає кінцевий продукт, проект, що використовує сучасні технології мапування та геолокації для надання точної інформації щодо потенційних загроз для користувачів та способів убезпечити себе. Інтерфейс та сповіщення бота будуть спрощеними і інтуїтивно зрозумілими, дозволяючи швидко реагувати на небезпеку та вживати необхідні заходи для захисту.

У роботі використані методи системного аналізу, такі як метод моніторингу, індукції та дедукції.

Окрім того, у роботі проводиться аналіз існуючих систем та додатків, що використовуються для оповіщення та навігації в екстрених ситуаціях. Цей огляд дозволить ідентифікувати сильні та слабкі сторони існуючих рішень, а також визначити напрямки подальших вдосконалень для нашого додатку з метою забезпечення максимальної ефективності та безпеки для користувачів.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ СИСТЕМ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Дослідження існуючих практик тестування програмного забезпечення

Тестування програмного забезпечення (ТПЗ) – це складний процес перевірки та оцінювання якості програмного продукту (ПЗ) з метою виявлення помилок, дефектів та інших аспектів, що можуть вплинути на його ефективність. Основною метою тестування є не лише виявлення проблем, а й переконання, що ПЗ працює коректно, відповідає вимогам та задовольняє очікування користувачів, забезпечуючи високий рівень надійності, безпеки та продуктивності. [1] Тестування включається в одну з ключових стадій циклу розробки програмного забезпечення. Починаючи з етапу аналізу, учасники процесу обговорюють вимоги до кінцевого продукту. [2] Мета цієї стадії – деталізація вимог до системи. Крім того, важливо забезпечити правильне розуміння завдань усіма учасниками та конкретизацію практичної реалізації кожної вимоги. На етапі визначення вимог формулюються Атрибути якості, що розширюють опис функціональності продукту та визначають характеристики, важливі для користувачів та розробників. [4] До таких атрибутів належать легкість використання, простота переміщення, цілісність, ефективність та стійкість до збоїв. Нефункціональні вимоги визначають зовнішні взаємодії системи та середовища, а також обмеження дизайну та реалізації. На стадії проектування (відомої як стадія дизайну та архітектури) програмісти та системні архітектори, керуючись вимогами, розробляють високорівневий дизайн системи. Технічні аспекти проектування обговорюються з усіма зацікавленими сторонами, включаючи замовника, і

визначають технології, завантаженість команди, обмеження, часові рамки та бюджет. Відповідно до уточнених вимог обираються оптимальні проектні рішення. Після затвердження вимог і дизайну продукту, наступною фазою життєвого циклу стає безпосередня розробка. Тут програмісти розпочинають написання коду програми відповідно до раніше визначених вимог. Системні адміністратори конфігурують програмне середовище, front-end програмісти вибудовують користувальницький інтерфейс програми і визначають її взаємодію з сервером. Окрім того, програмісти створюють Unit-тести для перевірки правильності роботи коду кожного компонента системи, проводять рев'ю написаного коду, формують білди і розгортають готове програмне забезпечення в програмному середовищі. Цей процес повторюється до досягнення вимог.

Після етапу реалізації наступає етап документації, хоча сама ця фаза є відносно умовною, оскільки процеси документації відбуваються на всіх етапах реалізації. [5] Подальше переходження настає на етап тестування.

Рис. 1.1. Життєвий цикл програмного забезпечення

Основні складові процесу тестування включають планування тестування, розробку тестових кейсів і сценаріїв, виконання тестів, аналіз

результатів і підготовку звітності. Важливо також ураховувати та поєднувати різні типи тестування, такі як функціональне, навантажувальне та інші, а також використовувати автоматизацію тестування для підвищення ефективності та повторюваності процесу.

Рис. 1.2. Життєвий цикл тестування

1.2. Види тестування програмного забезпечення

Згідно з даними ISTQB (International Software Testing Qualifications Board), вид тестування представляє собою засіб чіткого визначення мети конкретного рівня для програми або проєкту. [3] Цей вид тестування фокусується на конкретній меті, яка може охоплювати перевірку функцій, виконуваних компонентами або системою в цілому. Мета тестування може включати перевірку елементів нефункціонального тестування (надійність, зручність використання), структури та архітектури компонентів чи системи в цілому. Також тестування може бути спрямоване на визначені аспекти, такі як перевірка виправлення конкретного дефекту (підтверджувальне або повторне тестування) або перевірка випадкових змін в системі (регресійне тестування), що може виникнути внаслідок змін в системі.

Відповідно до потреб, процес тестування має бути організований відповідно. Отже, можна виділити чотири основних типи тестування програмного забезпечення:

- Функціональне тестування (Functional testing).
- Нефункціональне тестування (Non-functional testing).
- Структурне тестування (Structural testing).
- Тестування змін (Change related testing).

1.2.1. Функціональне тестування

Сьогодні важко переоцінити важливість функціонального тестування, оскільки ця дія спрямована на перевірку всіх функцій системи з метою підтвердження, що кожна функція програми працює відповідно до документації. [6] Елементи функціонального тестування включають:

- Підготовка тестових даних на основі описаної документації.
- Бізнес-вимоги, як частина функціонального тестування.
- Отримання результатів на основі специфікації.
- Проходження тест-кейсів.
- Аналіз фактичних та очікуваних результатів.

Функціональне тестування може бути проведене відповідно до специфікації, а також на основі бізнес-процесу, що відображає знання системи.

Переваги функціонального тестування:

- Ми "копіюємо" безпосереднє використання системи під час тестування.
- Тестування, як правило, проводиться в умовах близьких до реальних.

Недоліки:

- Існує ймовірність пропустити кілька помилок логіки програмного забезпечення під час перевірки функціоналу програми.

1.2.2. Нефункціональне тестування

Нефункціональне тестування спрямоване на оцінку тих аспектів програмного забезпечення, які, хоча можуть бути описані в документації, не відносяться безпосередньо до функцій програмних продуктів. [6] Воно включає такі підвиди:

- Тестування стабільності (Stability testing): Перевірка працездатності додатку при тривалому тестуванні з очікуваним рівнем навантаження.
- Юзабіліті тестування (Usability testing): Дослідження для визначення зручності використання ПЗ.
- Тестування ефективності (Efficiency testing): Перевірка обсягів коду і ресурсів QA, що використовуються програмою для виконання функцій.
- Тестування ремонтопридатності (Maintainability testing): Оцінка легкості підтримки працездатності системи.
- Перевірка портативності (Portability testing): Тестування можливості перенесення ПЗ з одного середовища на інше.
- Тестування "пра-витоків" (Baseline testing): Перевірка документації і специфікації, за якими будуть написані тест-кейси, включаючи тестування вимог.
- Приймальне тестування (Compliance/Acceptance testing): Перевірка продукту на відповідність критеріям готовності.
- Тестування документації (Documentation testing): Перевірка створеної в рамках тестування документації від майстера тест-плану до тест-кейсів.

- Тестування витривалості системи (Endurance testing): Тестування системи при високому навантаженні протягом тривалого періоду часу.
- Тестування навантаження (Load testing): Визначення поведінки ПЗ під очікуваним рівнем навантаження.
- Тестування продуктивності (Performance testing): Перевірка швидкості роботи ПЗ або його окремих функцій.
- Тестування сумісності (Compatibility testing): Тестування системи в різних середовищах.
- Тестування безпеки (Security testing): Перевірка безпеки та захищеності додатка.
- Об'ємне тестування (Volume testing): Тестування ПЗ з використанням баз даних певного розміру.
- Стрес тестування (Stress testing): Тестування системи в умовах обмеженості ресурсів комп’ютера.
- Тестування швидкості відновлення (Recovery testing): Визначення швидкості відновлення системи під час збою.
- Тестування локалізації, інтернаціоналізація (Localization testing): Перевірка відповідності мовних, культурних та релігійних норм.

Ці підвиди нефункціонального тестування допомагають забезпечити повноту та якість перевірки всіх аспектів програмного забезпечення.

1.2.3. Структурне тестування

Структурне тестування спрямоване на ретельне перевіряння структури системи чи компонента. [6] Цей підхід часто віднесений до категорій "білого" та "сірого" ящиків, оскільки дозволяє проникнути всередину системи або додатка для детального аналізу.

Різні методи структурного тестування допомагають забезпечити високий рівень покриття коду та зрозуміти, як система працює внутрішньо:

- Строкове покриття (Statement Coverage): Гарантує, що кожен оператор в програмі використовується хоча б один раз.
- Покриття шляху (Path Coverage): Спрямоване на задоволення критеріїв охоплення кожного логічного шляху в програмі.
- Покриття рішення (Branch Coverage): Визначає, чи має кожна умова розгалуження в програмі дійсні чи хибні значення.
- Покриття умови (Condition Coverage): Схожий на Branch Coverage, але перевіряє стан покриття для умовних і неумовних гілок.

Переваги структурного тестування:

- Можливість виявлення та видалення "зайвого" коду.
- Раннє виявлення потенційних помилок.
- Забезпечує більш ретельне тестування програмного забезпечення.
- Не вимагає великих витрат людино-годин.

Недоліки структурного тестування:

- Вимагає глибокого розуміння коду та інструментів тестування.
- Структурне тестування виявляється надзвичайно ефективним для виявлення помилок і покращення якості програмного забезпечення на ранніх етапах розробки.

1.2.4. Тестування змін

Важливо ретельно розглянути відмінності та межі між регресійним тестуванням (Regression testing) та повторним тестуванням (Retesting) при внесенні змін до системи.

Регресійне тестування (Regression testing):

Регресійне тестування виконується з метою перевірки працездатності існуючого функціоналу та виявлення відсутності сторонніх помилок після внесення змін або оновлення білда. [7] Це проводиться лише при додаванні нових функцій чи суттєвих змін у функціоналі системи. Регресійне тестування може виконуватися паралельно з повторним тестуванням. Тест-

кейси можуть бути автоматизовані, і ті, які були вже відмічені як "Passed", повинні пройти перевірку.

Ретестування (Retesting):

Ретестування виконується в тому ж самому середовищі та з тими ж даними, але на новому білді. [7] Це має вищий пріоритет і виконується перед регресійним тестуванням. Тест-кейси не можуть бути автоматизовані, і в рамках цього виду тестування перевіряються тільки тест-кейси зі статусом "Failed". Ретестування підтверджує виправлення помилок та коректну роботу функціоналу, забираючи менше часу на верифікацію та не вимагаючи нових налаштувань середовища тестування.

Переваги та недоліки:

Регресійне тестування:

Переваги:

- Підтверджує відсутність багів після додавання фічі або правки коду.
- Може бути автоматизоване, покращуючи ефективність.
- Допомагає поліпшити якість продукту.

Недоліки:

- Може бути трудомістким через велику кількість тест-кейсів.

Повторне тестування:

Переваги:

- Підтверджує виправлення помилок та коректну роботу функціоналу.
- Підвищує загальну якість продукту.
- Вимагає менше часу для верифікації.

Недоліки:

- Тест-кейси можуть виявлятися тільки після першого раунду тестування.
- Не може бути автоматизоване.
- Вимагає додаткового часу для проходження вже пройдених тест-кейсів.

- Ретельне управління обома типами тестування може ефективно забезпечити високий ступінь якості продукту під час розвитку та після внесення будь-яких змін.

1.3. Рівні тестування

Тестування на різних рівнях системи є важливою складовою усього життєвого циклу розробки та підтримки програмного забезпечення. [8] Визначення рівня тестування визначає область застосування тестів: це може бути окремий модуль, група модулів або вся система. Широкий охоплення тестування на різних рівнях є важливою передумовою для успішної реалізації та вдачі проекту.

Рівні тестування:

- Компонентне або Модульне тестування (Component testing or Unit testing):

Цей рівень спрямований на перевірку і валідацію окремих компонентів або модулів. Тут тестуються ізольовані частини програми, переконуючись, що кожна функція працює коректно.

- Інтеграційне тестування (Integration testing):

На цьому рівні перевіряється взаємодія між різними компонентами чи модулями. Метою є впевненість у правильному об'єднанні окремих частин системи.

- Системне тестування (System testing):

Система тестується в цілому, після інтеграції всіх компонентів. Мета полягає в підтвердженні, що система відповідає вимогам та працює як єдиний функціональний блок.

- Приймальне тестування (Acceptance testing):

На останньому рівні здійснюється перевірка системи замовником або кінцевим користувачем для визначення відповідності вимогам та готовності системи до впровадження. Запевняючи охоплення тестування на всіх рівнях,

забезпечуємо високу якість програмного продукту та ефективність його роботи на кожному етапі життєвого циклу.

1.3.1. Модульне тестування

Модульне тестування (Unit testing)– тестування кожної атомарної функціональності додатку окремо, в штучно створеному середовищі. [9] Саме потреба у створенні штучної робочого середовища для певного модуля, вимагає від тестувальника знань в автоматизації тестування програмного забезпечення, деяких навичок програмування. Дане середовище для деякого юніта створюється за допомогою драйверів і заглушок.

Драйвер – визначений модуль тесту, який виконує елемент, що ми тестуємо.

Заглушка – частина програми, яка симулює обмін даними із компонентом, що тестується, виконує імітацію робочої системи.

Заглушки необхідні для:

- імітації відсутніх компонентів для роботи даного елементу;
- подачі або повернення модулю певного значення, можливості надати тестеру самому ввести потрібне значення;
- відтворення певних ситуацій (виключення або інші нестандартні умови роботи елемента).

Перш за все, потрібно окреслити рамки, в яких юніт-тестування виправдано. По-перше, архітектура проекту повинна бути спроектована відповідно до ідей ООП (чіткий розподіл на класи, кожен з яких виконує свою певну функцію), що забезпечить систему грамотним розподілом на модулі. Також, модульне тестування має бути менш витратним при пошуку дефектів, ніж інші види тестів і повинне зменшувати час відладки коду. [9]

Що ж стосується безпосередньо переваг:

- модульне тестування мотивує програмістів писати код максимально оптимізованим, проводити рефакторинг (спрощення коду програми, не зачіпаючи її функціональність), так як за допомогою Юніт-

тестування можна легко перевірити працездатність розглянутого компонента.

- необхідність відділення реалізації від інтерфейсу (зважаючи на особливості модульного тестування), що дозволяє мінімізувати залежності в системі.
- документація Юніт-тестів може служити прикладом «живого документа» для кожного класу, що тестується даним способом. Модульне тестування допомагає краще зрозуміти роль кожного класу на тлі всієї програмної системи.
- також, при «розробці через тестування», яка активно використовується в екстремальному програмуванні, модульне тестування є одним з основних інструментів, що дозволяє розробляти продукт відповідно до вимог до даного модулю.

1.3.2. Інтеграційне тестування

Інтеграційне тестування визначає себе як ключовий аспект у процесі тестування програмного забезпечення, спрямований на оцінку ефективності інтеграції різних програмних модулів. [10] У сучасному світі компанії регулярно використовують кілька програмних модулів, і саме завдяки інтеграції ці програми здатні співпрацювати, підвищуючи загальну ефективність та оптимізуючи робочі процеси.

Важливість інтеграційного тестування проявляється в його здатності забезпечити плавну інтеграцію програмних модулів, що робить їх взаємодію ефективною. Умови, коли кожен модуль розробляється різними розробниками із використанням різних логік програмування, вимагають тщательної перевірки для впевненості в гладкій інтеграції від самого початку.

Тестування інтеграції дає IT-спеціалістам можливість оцінити взаємодію різних модулів та внести необхідні зміни для оптимізації їхньої ефективності.

Сутність інтеграційного тестування полягає в оцінці процесу обміну даними між двома компонентами або програмними модулями, сприяючи виявленню та усуненню дефектів, що можуть виникнути внаслідок їх інтеграції.

Ключові аспекти інтеграційного тестування:

Стратегії інтеграційного тестування: Ці стратегії надають групам розробників та IT-спеціалістам засоби виявлення дефектів, які можуть виникнути внаслідок інтеграції двох або більше модулів програмного забезпечення. Крім того, вони дозволяють оцінити загальну придатність та функціонування об’єднаних елементів програмного забезпечення.

Час проведення інтеграційного тестування: Зазвичай інтеграційне тестування відбувається після модульного тестування, яке передбачає валідацію окремих модулів та блоків. Після того, як визначено, що кожен блок працює ізольовано, інтеграційне тестування оцінює, як усі блоки працюють у поєднанні.

Використовуючи інтеграційне тестування, забезпечуємо надійність та високу якість роботи програмного забезпечення в умовах взаємодії різних його складових.

Інтеграційне тестування визначається як послідовний процес, який вимагає від тестувальників поетапно інтегрувати модулі та проводити тестування на кожному етапі. Зміцнюючи взаємодію компонентів, цей процес гарантує ефективну роботу програмного забезпечення в цілому.

Тести інтеграції базуються на чітко визначеній специфікації інтерфейсу між тестованими компонентами. Автоматизація цих тестів важлива для їхнього регулярного виконання, щоб вчасно виявляти проблеми ще на етапі їхнього виникнення.

Цей тип тестування визначається як засіб перевірки того, чи працюють всі компоненти програми так, як очікувалося. Його мета - впевнитися, що інтеграція різних модулів і компонентів відповідає вимогам користувача та технічним критеріям.

Основні причини важливості інтеграційного тестування:

- Різна логіка розробників: Розробники можуть використовувати різну логіку при розробці модулів для одного програмного продукту. Інтеграційне тестування стає єдиним способом переконатися, що окремі модулі працюють належним чином разом.
- Зміна структури даних: Переміщення даних між модулями може змінити їхню структуру та викликати проблеми в роботі програми.
- Взаємодія зі сторонніми інструментами та API: Модулі можуть взаємодіяти з іншими інструментами та API. Інтеграційне тестування дозволяє перевірити правильність обміну даними та відповідність очікуванням.
- Оцінка ефективності змін: Інтеграційне тестування важливе для оцінки ефективності внесених змін, особливо якщо розробник вносить зміни без модульного тестування.
- Інтеграційне тестування є необхідним для забезпечення надійності, відповідності вимогам та високої якості функціонування багатокомпонентних програмних додатків.

1.3.3. Системне тестування

Системне тестування - це комплексне тестування програмного забезпечення, яке виконується на повній інтегрованій системі. Його мета - перевірити відповідність системи вихідним вимогам, як функціональним, так і не функціональним. [11] В процесі системного тестування можна виявити різноманітні дефекти, такі як неправильне використання системних ресурсів, непередбачені комбінації даних користувача, проблеми з сумісністю оточення, невідповідність функціональним вимогам та інші.

Цей вид тестування виконується за методом "Чорного ящика", фокусуючись на зовнішніх аспектах системи, не взаємодіючи з її внутрішнім складом. Рекомендується проводити його в середовищі, яке якнайбільше наближене до оточення кінцевого користувача.

Існують два підходи до системного тестування:

На базі вимог: Тестування відбувається відповідно до функціональних або нефункціональних вимог, для кожного з яких створюється тестовий прецедент.

На базі випадків використання: Тестування відбувається відповідно до варіантів використання продукту, на основі яких створюються користувальницькі прецеденти. Кожен з цих користувальницьких прецедентів визначає свій тестовий прецедент.

Також до системного тестування відносять альфа-тестування та бета-тестування, що надають можливість залучити кінцевих користувачів до перевірки продукту перед його фінальним випуском. У результаті, системне тестування стає ключовим етапом впевненості в надійності та якості програмного забезпечення.

1.3.4. Приймальне тестування

Приймальне тестування, відоме також як тестування на прийняття, є ключовим етапом на шляху до випуску готового продукту. [12] Цей вид тестування виконується на етапі здачі готового продукту або його частини замовнику. Його основною метою є визначення готовності продукту до використання, а це досягається за допомогою проходження тестових сценаріїв та випадків, побудованих на основі специфікації вимог до розроблюваного програмного забезпечення.

На завершальному етапі приймального тестування можуть мати місце два основних результати:

- Відправка проекту на доопрацювання.
- Ухвалення його замовником як виконаної задачі.

Цей етап тестування є останнім перед релізом та має свої особливості. Він зазвичай не настільки ретельний, як інші види тестувань, і фокусується переважно на основних функціях продукту. Однак важливість його ролі

важко переоцінити, оскільки він забезпечує остаточну перевірку готовності продукту перед його випуском.

Приймальне тестування може бути проведене замовником, групою тестувальників, що представляють інтереси замовника, або тестувальниками компанії-розробника. Вибір варіанта залежить від уподобань та потреб компанії-замовника. Важливо враховувати, що цей етап є важливим з точки зору задоволення вимог і очікувань замовника перед фінальним випуском продукту.

1.4. Методи тестування програмного забезпечення

Тестування програмного забезпечення можна проводити як вручну, так і з використанням автоматизованих інструментів. Автоматизація тестування - це процес використання програмних засобів для виконання тестових випадків, оцінки результатів і створення звітів. [13] Види тестування ПЗ за ступенем автоматизації:

- Ручне тестування - тестування, яке виконується вручну без використання автоматизованих інструментів.
- Напівавтоматичне тестування - тестування, за якого деякі кроки виконуються вручну, а деякі - з використанням автоматизованих інструментів.
- Повністю автоматичне тестування - це процес тестування програмного забезпечення, за якого тести запускаються автоматично без участі людини, що підвищує швидкість і надійність тестування.

1.4.1. Ручне тестування

Ручне тестування є важливим етапом в процесі виявлення помилок та забезпечення якості програмного забезпечення. Цей тип тестування включає виконання тестових кейсів тестувальниками вручну, без використання автоматизованих інструментів. [14] Хоча деякі можуть вважати його

простим, ручне тестування виявляється невід'ємною частиною виготовлення високоякісного програмного продукту.

Компанії використовують ручне тестування для ефективного виявлення помилок та проблем у своєму програмному забезпеченні. В цьому процесі тестувальники перевіряють функціональність програми, оцінюючи її відповідність вимогам та виявляючи труднощі, які можуть виникнути при реальному використанні.

На різних етапах розробки використовується ручне тестування для досягнення оптимальних результатів. Починаючи з етапу розробки базового функціоналу, розробники тестують кожну частину програми вручну, оцінюючи її роботу та виправляючи помилки.

Особливий акцент ручного тестування робиться на етапі розробки інтерфейсу користувача. Тут тестувальники перевіряють, як реальні користувачі взаємодіють з програмою, оцінюючи зручність меню та загальну ефективність системи.

З урахуванням того, що ручне тестування дозволяє виявити аспекти програми, недосяжні для автоматизованих засобів, його роль у розробці ПЗ важко переоцінити. Цей метод не тільки забезпечує виявлення дефектів, а й сприяє покращенню загальної якості та відповідності програмного продукту вимогам користувача.

Ручне тестування визначається як ідеальний метод для отримання глибокого розуміння продукту, оскільки воно включає аналіз великої кількості якісних даних та особистих думок, а не лише чистих кількісних показників.

Існують випадки, коли ручне тестування може займати забагато часу та зусиль, наприклад, у випадку тестування баз даних. Оскільки бази даних опрацьовують величезні обсяги інформації, ручне введення даних стає неефективним процесом. У таких ситуаціях використання автоматизованих систем виявляється найбільш оптимальним рішенням, оскільки вони здатні обробляти великі обсяги даних за короткий період часу.

Ще однією областю, де ручне тестування може бути менш ефективним, є навантажувальні тести. Наприклад, коли розробник вивчає, як його програмне забезпечення впорається зі значним навантаженням користувачів. У випадку онлайн-додатків та програм із серверами це може вимагати одночасного доступу до програми багатьох користувачів, що може призвести до великих трудовитрат. Автоматизовані системи тестування можуть виконувати цю задачу значно ефективніше та вартісно-ефективніше.

Отже, вибір між ручним та автоматизованим тестуванням залежить від конкретних вимог проекту та його характеристик, забезпечуючи оптимальний та раціональний підхід до тестування програмного забезпечення.

Процес ручного тестування включає різні аспекти, кожен з яких ефективно перевіряється завдяки специфічним завданням тестів. Визначимо деякі ключові аспекти ручного тестування та їхню раціональність.

Базова функціональність:

Один з перших етапів тестування – перевірка базової функціональності програмного забезпечення. Розглядаючи масштабність модулів коду, ручне тестування набагато ефективніше, оскільки дозволяє швидше оцінити, чи працює функціонал так, як очікується.

Прикладом є тестування програм для роботи з базами даних, де тестувальники вводять частину даних та порівнюють їх із сподіваними результатами. Ручне тестування на цьому етапі створює міцний фундамент для подальшого розвитку продукту.

Інтерфейс користувача:

Тестування інтерфейсу користувача вимагає уваги до функціональності та зручності для кінцевого користувача. Автоматизація важко передає естетичні та емоційні аспекти інтерфейсу, тому ручне тестування на цьому етапі необхідне.

Тестування включає перевірку, наскільки користувач може взаємодіяти з різними функціями та чи відповідає інтерфейс вимогам щодо зручності та естетики.

Таким чином, ручне тестування виявляється необхідним для глибокого оцінювання функціональності та емоційного враження від інтерфейсу користувача, створюючи раціональний підхід до забезпечення якості продукту.

Тестування на проникнення — це необхідний етап для визначення легкості доступу до програмного забезпечення ззовні за допомогою незаконних методів. Такий підхід спрямований на виявлення потенційних порушень безпеки та запобігання незаконному доступу до даних.

Автоматизація тестування програмного забезпечення здебільшого фокусується на виконанні конкретних завдань, вже вбудованих у програму. У той час як автоматизація ефективна для тестування функціональності, вона обмежена щодо виявлення нових областей для потенційних атак, що робить тестування безпеки значущим завданням для людського фахівця.

Наприклад, компанія може найняти етичного хакера для вивчення їхнього програмного забезпечення та виявлення можливих шляхів для зловмисників отримати доступ до конфіденційної інформації, що стало особливо важливим з прийняттям GDPR.

Дослідницьке тестування використовується для виявлення несподіваних функцій або помилок та виконується лише раз чи двічі в процесі "дослідження" програмного забезпечення. Цей вид тестування вимагає глибокого розуміння програми та швидкість виявлення непередбачених аспектів.

Ручне тестування найбільш підходить для цього завдання, оскільки створення автоматизованих тестових кейсів може зайняти багато часу, тоді як фахівець, вручну взаємодіючи з програмою, ефективно виявляє неочікувані рішення та потенційні проблеми.

Наприклад, розробник може використовувати ручне тестування для перевірки правильності інтеграції нової функції, шляхом одного тесту, що визначає коректність обробки даних в програмі.

У життєвому циклі ручного тестування, виявляється низка етапів, під час яких ручне тестування використовується для глибокої перевірки різних аспектів програмного пакету.

Деякі з етапів життєвого циклу ручних тестів включають в себе наступні:

Планування: систематичне створення тестових сценаріїв

Етап планування включає у себе детальне створення циклу тестування, оцінку вимог програми та розробку конкретних тестових сценаріїв. Важливо написати тестові кейси, які будуть зрозумілі ручним тестувальникам, і врахувати різні можливі варіації виконання тестів.

Перевірка: глибокий аналіз записів тестів

На етапі перевірки тестів виконується багаторазовий прогін тестових кейсів для отримання послідовних даних. Важливо не лише реєструвати результати, а й визначити причини відхилень від тестових кейсів. Аналіз зосереджується на пошуку помилок та врахуванні відмінностей, що можуть виникнути в різних варіантах тестування.

Аналіз: широкий огляд результатів та якісної інформації

Під час аналізу всіх результатів тестів визначаються потенційні проблеми у програмному забезпеченні. Тут важливо виходити за рамки функціональності і розглядати якісну інформацію, таку як дизайн додатку. Ручне тестування особливо цінне для генерації описових даних, що допомагають розробникам вносити невеликі корективи для покращення користувацького досвіду.

Реалізація: згідно змінам для вдосконалення

Використовуючи попередні звіти та аналіз, розробники впроваджують зміни у програмне забезпечення. Цей процес може бути тривалим, оскільки розробники експериментують з кодом для вирішення помилок з попередніх

версій. Ручне тестування забезпечує додаткову перевагу у взаємодії розробників із тестувальниками для правильного розуміння та впровадження коригувань.

Перезапуск планування: постійний цикл тестування

Під час виконання розробкою змін, заплануйте наступний набір тестів для перевірки останніх оновлень та спроб відтворити помилки з попередньої версії. Постійний цикл тестування гарантує постійне вдосконалення програмного забезпечення та підтримує його в актуальному стані. Ручне тестування, хоча і тривале, забезпечує значну ефективність завдяки гнучкості та безперервності повторних тестів.

Використання ручного тестування в розробці програмного забезпечення принесло безліч переваг, які охоплюють якість продукту і фінансовий вплив на компанію. Передоваги ручного тестування виявляються на кількох рівнях, від поліпшення якості програмного забезпечення до позитивного впливу на фінансовий стан компанії.

Завершення автоматизації тестування передбачає активну участь QA-аналітика, який входить у програмне забезпечення та розробляє тестові кейси, гарантуючи їх точне виконання при кожній ітерації. Незважаючи на користь автоматизації, важливість людського фактору полягає в здатності тестувальника помітити аномалії, які можуть залишитися непоміченими в іншому випадку.

Гнучкість ручних тестів дозволяє тестувальникам виявити проблеми, які автоматизація може упустити, і надає більше можливостей для виправлення виявлених недоліків. Ручне тестування також сприяє виявленню проблем у програмі, що може покращити загальний досвід користувача.

Якісна інформація, яку можуть надати ручні тестувальники, виявляється невартісною для компанії. Вони можуть виносити важливі аспекти, такі як відчуття “незграбності” меню, що стає важкою задачею для автоматизованих систем. Важливість такої інформації полягає в можливості негайно виправити виявлені недоліки і підвищити рівень якості додатку.

Уведення ручного тестування в бізнес-процеси дозволяє компаніям ефективно поліпшити своє програмне забезпечення, чого було б важко досягти, використовуючи виключно автоматизовані методи тестування. Використання ручних тестів у комбінації з автоматизованими підходами дозволяє досягти оптимального рівня ефективності і якості у випуску програмного забезпечення.

Обмеження, з якими стикаються деякі (хоча і не всі) платформи, включають неможливість роботи з такими платформами, як Linux, можливість роботи лише з певною мовою кодування та виконання лише певної кількості завдань.

При взаємодії з людьми у процесі тестування, технічні обмеження фактично втрачають свою суттєвість. Органічне спілкування з тестувальниками відкриває широкі можливості, де головним обмеженням стає їхня кваліфікація, а не технічні аспекти.

Створення стратегії тестування в цьому випадку стає більш гнучкою і докладною, оскільки ви можете розраховувати на людей для ретельного дослідження програмного забезпечення без компромісів.

Юзабіліті-тестування є ключовим етапом в оцінці того, наскільки продукт є "зручним для використання". Воно враховує не лише технічні аспекти функціональності, але й візуальний та тактильний досвід користувача. Ручне юзабіліті-тестування виходить за межі автоматизованих засобів та дозволяє отримати відгуки, які допомагають зробити продукт конкурентоспроможним.

Існують проблеми, пов'язані з використанням ручного тестування в якості інструменту забезпечення якості. Однак з усвідомленням цих проблем можна вдосконалити техніки ручного тестування, попереджаючи серйозні труднощі та підвищуючи стандарти програмного забезпечення.

Наведені вище вказівки підкреслюють, що взаємодія зі спеціалістами та використання ручного тестування дозволяють збагачувати якість програми та забезпечувати кращий досвід для кінцевого користувача.

Перша ключова проблема, яку необхідно вирішувати, полягає в рівні кваліфікації ручних тестувальників у команді. Важливо мати талановитих спеціалістів, оскільки вони більше виявлять помилки та гарантуватимуть належне функціонування програмного забезпечення. Кращі компанії завжди прагнуть залучати висококваліфікованих ручних тестерів для забезпечення високого рівня продуктивності.

Як тестувальник, важливо постійно вдосконалювати свої навички. Високий рівень кваліфікації дозволяє вам приносити більше користі компанії, оскільки ручне тестування виявляє більше помилок і покращує користувацький досвід. Відмінні ручні тести створюються тестувальниками, які інвестували час у вдосконалення своєї експертизи.

Ручне тестування стало стандартним процесом для компаній різних розмірів, але його вартість може бути значною. Наприклад, компанія з кількома висококваліфікованими тестувальниками може витрачати багато коштів на повторне тестування через оплату часу кожного з них. У випадку автоматизованих тестів ця проблема менше актуальна.

Ефективним рішенням цієї проблеми є передбачене планування. Чим більше часу ви витрачаєте на планування тестів та їх порядку виконання, тим менше ймовірно, що витрати на персонал зростатимуть через зайві тести.

Комп'ютери перевершують людей у багатьох завданнях, від шахового аналізу до фінансових операцій на фондовому ринку чи простого натискання кнопок. Та ж сама концепція відноситься і до тестування, де користувачі не завжди бажають читати всю інформацію та розбиратися в меню.

Ручне тестування може займати значно більше часу, ніж використання автоматизації тестування. Для протистояння цьому використовують комбінацію ручних і автоматизованих тестів, використовуючи тестувальників для експертизи та залишаючи ручну роботу там, де вона необхідна. Спрощення процесів особливо ефективно в ручному тестуванні, оскільки воно усуває зайві кроки.

Помилки виникають внаслідок людського фактору. Це може бути неправильний порядок виконання тесту чи неточний запис результатів через помилкове натискання кнопки миші. Такі помилки можуть призвести до серйозних проблем з точністю тестування програмного забезпечення.

Ручні тестувальники, які повторюють одні й ті ж завдання, можуть втомитися і частіше допускати помилки. Використання автоматизації дозволяє уникнути цього або надавати тестувальникам перерви для покращення уваги.

Менеджери також повинні розглядати можливості управління робочим навантаженням, щоб уникнути вигорання і проблем зі здоров'ям тестувальників.

Основні характеристики ручних тестів включають оптимізацію тестових кейсів. Інструкції для тестування оптимізовані з високим рівнем ефективності, що дозволяє команді тестувальників зекономити час і ресурси при виконанні завдань.

Завжди намагайтеся обмежувати розмір тестового кейсу там, де це можливо, щоб максимально ефективно використовувати наявні ресурси. Ручне тестування найкраще має більш зрозумілі метрики порівняно з автоматизацією тестування. Там, де автоматизація постійно генерує складну статистику та інформацію, ручні тести включають простіші метрики, що легко генеруються і займають менше часу для аналізу на більш пізніх етапах процесу.

Ручне тестування призводить до більш інтелектуальних звітів від команди тестувальників, оскільки тестувальники-люди можуть створювати гнучкі та індивідуальні звіти, додаючи інформацію, яку вони вважають корисною для команди розробників.

Стратегії повторного запуску у ручному тестуванні є гнучкішими, оскільки тестувальники можуть виконувати більше тестів за їх власною ініціативою, якщо вони вважають, що є щось, що потрібно дослідити.

Основна відмінність між ручним та автоматизованим тестуванням полягає в способі виконання. Ручне тестування повністю покладається на людину, яка виконує тестування, слідкує за тестовим кейсом до завершення та записує будь-яку інформацію, тоді як в автоматизованих тестах комп'ютерна програма відповідає за виконання тестових кейсів, спроектованих QA-аналітиком.

Деякі платформи автоматизованого тестування також генерують власні звіти для користувачів, обмежуючи час, який потрібно витратити на збір усіх даних з експерименту. Такі платформи можуть зосередитися на створенні виправлень проблем, які можуть виникнути в процесі тестування програмного забезпечення.

Існують фундаментальні відмінності між ручним і автоматизованим тестуванням, і ці концепції базуються на абсолютно різних принципах для ефективної роботи. Однак вони можуть успішно взаємодіяти в тандемі над різними проектами розвитку. Використовуючи автоматизоване тестування для важких завдань і ручні методи тестування для ситуацій, що потребують більшої гнучкості, можна суттєво прискорити процеси тестування.

Одна з найбільших помилок у тестуванні полягає в уявленні, що є тільки бінарний вибір між ручним і автоматизованим тестуванням. В реальності успішна команда забезпечення якості може використовувати обидві стратегії, доповнюючи одна одну і вибираючи найбільш підходящий підхід для конкретного контексту тестування.

1.4.2. Автоматизоване тестування

Автоматизоване тестування – це процес використання програмних інструментів, які запускають нещодавно розроблене програмне забезпечення або оновлення через низку тестів для виявлення потенційних помилок кодування, вузьких місць та інших перешкод продуктивності. [15] Засоби автоматизації тестування програмного забезпечення виконують такі функції:

- Впровадження та виконання тестів: Автоматизовані інструменти виконують набір тестів для перевірки функціональності та продуктивності програмного забезпечення.
- Аналіз результатів: Засоби автоматизації аналізують результати виконання тестів, ідентифікуючи помилки, виключення та інші проблеми.
- Порівняння результатів з очікуваними результатами: Автоматизовані тести порівнюють фактичні результати з очікуваними для виявлення розходжень.
- Формування звіту про продуктивність програмного забезпечення розробки: Засоби автоматизації генерують звіти, які можна використовувати для оцінки продуктивності та якості програмного забезпечення.

Під час тестування нового програмного забезпечення або оновлень, ручне тестування може бути дорогим і виснажливим. Тому автоматизовані тести стають ефективним інструментом, зменшуючи витрати і час.

Автоматизовані тести допомагають виявляти збої швидше з меншим ризиком людської помилки. Їх можна легко запускати кілька разів для кожної зміни або до досягнення бажаних результатів.

Процес автоматизації також прискорює виведення програмного забезпечення на ринок, дозволяючи ретельно тестувати певні області для вирішення проблем перед переходом до наступних етапів.

Модульне тестування передбачає розбиття програмного забезпечення на легкозасвоювані одиниці для виявлення будь-яких помилок або проблем із продуктивністю.

Модульне тестування є важливою складовою процесу розробки програмного забезпечення, спрямованою на виявлення та виправлення помилок на ранніх етапах перед глибоким інтеграційним тестуванням. Цей вид тестування забезпечує перевірку кожного окремого компонента

програмного забезпечення, гарантуючи, що найдрібніші деталі працюють коректно.

На початкових етапах розробки модульне тестування дозволяє виявляти та вирішувати проблеми, запобігаючи їх поширенню на більш пізні етапи розробки. Це полегшує уникнення виправлення помилок на пізніших етапах, коли вони можуть стати більш складними та витратними для виправлення.

Модульне тестування має перевагу проводитися часто, оскільки воно гарантує, що всі компоненти працюють правильно перед їхньою інтеграцією в єдиноцільне програмне забезпечення. Після вдалого модульного тестування настає час для інтеграційних тестів, які перевіряють взаємодію всіх компонентів в рамках програми чи системи. Це дозволяє впевнитися, що всі елементи працюють як цілісна система та взаємодіють правильно між собою та з зовнішніми службами.

Важливою частиною інтеграційного тестування є перевірка правильної взаємодії між компонентами та їхня здатність взаємодіяти з іншими частинами програмного забезпечення чи зовнішніми сервісами. Це може включати в себе створення бази даних для інтеграційного тестування, де перевіряються всі можливі сценарії взаємодії.

Оскільки модульне тестування спрямоване на усунення більшості помилок на ранніх етапах, інтеграційне тестування може бути проведене менш часто. Це дозволяє ефективно використовувати час та ресурси для підтримки високої якості програмного забезпечення.

Тестування прикладного програмного інтерфейсу (API) важливо для перевірки взаємодії між програмними компонентами та забезпечення їхньої правильної роботи у різних умовах. Деякі основні види тестування API включають:

- Валідаційне тестування: Перевірка правильності вхідних та вихідних даних API. Це включає в себе перевірку форматів даних, кодування та інші аспекти даних, що пересилаються через API.

- Функціональне тестування: Перевірка правильності функціональності API. Тестування різних методів API та їхніх можливостей відповідно до специфікацій.
- Тестування безпеки: Оцінка заходів безпеки API, виявлення та виправлення потенційних уразливостей, таких як атаки на введення та виведення даних.
- Тестування навантаження: Визначення стійкості та продуктивності API під час обробки великої кількості запитів.

Тестування інтерфейсу користувача (GUI) фокусується на взаємодії з програмним забезпеченням через різні інтерфейси, такі як операційні системи та браузери. Важливо перевірити функціональність, візуальний дизайн, продуктивність та зручність використання для кінцевих користувачів.

Автоматизація тестування інтерфейсу користувача дозволяє ефективно відтворювати досвід користувача та гарантує, що програмне забезпечення відповідає очікуванням користувача. Засоби автоматизації можуть включати сценарії тестування, які відображають реальні використання програмного забезпечення та взаємодію з іншими системами.

Оскільки попередні етапи тестування (модульне, інтеграційне, API) вже виявили та усунули багато проблем, тестування інтерфейсу користувача стає менш трудомістким і фокусується на досвіді кінцевого користувача. Автоматизація допомагає економити час та ресурси, спростовуючи процес тестування і підтримуючи високу якість програмного забезпечення.

Автоматизація тестування визначається як процес використання програмних інструментів для виконання тестів на програмному забезпеченні з метою виявлення помилок та забезпечення його належної роботи. Головною метою цього процесу є виявлення та усунення помилок до переходу проекту до іншої фази розробки або випуску для кінцевих користувачів. Успішна автоматизація тестування може суттєво скоротити час, необхідний для виправлення помилок, та забезпечити належну

функціональність програмного забезпечення. Вибір відповідних інструментів для автоматизації тестування є ключовим етапом у цьому процесі. Ефективні інструменти автоматичного тестування відрізняються наступними характеристиками:

- Простота використання: Інструмент повинен бути легким у використанні, навіть для тих, хто не має глибоких знань у програмуванні.
- Підтримка різних операційних систем, браузерів і пристроїв: Інструмент повинен мати можливість тестувати програмне забезпечення на різних платформах та середовищах.
- Комплект інструментів: Інструмент повинен мати повний пакет необхідних засобів для ефективної перевірки того, що вам потрібно.
- Підтримка мов сценаріїв та простота використання для тих, хто не програмує: Інструмент повинен бути придатним для використання навіть для тих, хто не володіє глибокими навичками програмування.
- Можливість багаторазового використання для багатьох тестів і змін: Інструмент повинен забезпечувати легкість повторного використання для різних тестових сценаріїв і змін.
- Підтримка великих наборів даних з різних джерел: Інструмент повинен дозволяти використовувати різноманітні дані для ефективної перевірки на основі даних.

Процес автоматизації тестування, який ґрунтується на використанні данних, має на меті не лише прискорення тестування, але й зменшення витрат, що робить його важливим у багатьох випадках. Ось декілька сценаріїв, де автоматизація тестування на основі даних може бути корисною:

- Повторне тестування: Будь-які тести, які потребують повторюваності та регулярної перевірки, виграють від автоматизованого тестування, оскільки воно забезпечує швидше виконання порівняно з ручним тестуванням.

- Тести високого ризику: Автоматизація дозволяє ізолювати потенційні точки збою та виправити їх, перш ніж внесенням змін до коду. Це допомагає уникнути уповільнення циклу розробки, якщо тест не виявить проблем.
- Тривалі тести: Тестування вручну може забирати багато часу та призводити до помилок. Автоматизація тестів дозволяє ефективно зменшити час, необхідний для проведення тестів, і знизити ризик пропуску важливих помилок.
- Багатогранні програми: Коли програмне забезпечення взаємодіє з іншими програмами чи середовищем, автоматизація допомагає виявити потенційні конфлікти та уникнути можливих проблем взаємодії.

1.5. Аналіз існуючих рішень

Слід розглянути вже існуючи системи автоматизації тестування та їх переваги й недоліки.

1.5.1. Selenium

Selenium представляє собою великий open source проект, конкретно, фреймворк для автоматизації браузера, в межах якого розробляється ряд програмних продуктів для автоматизованого тестування, що зазвичай використовуються для тестування веб-додатків. Офіційне джерело та документацію можна знайти за посиланням - https://www.seleniumhq.org/

"Selenium Core," основа проекту або "Selenium 1.0," спочатку виникло як бібліотека JavaScript з назвою "JavaScriptTestRunner," створена Джейсоном Хаггінсом у компанії ThoughtWorks у 2004 році. Ця бібліотека була призначена для запуску тестів у браузері для сайту, написаного на Python. Згодом до розвитку проекту приєдналися інші розробники, а з 2007 року Хаггінс та його колеги продовжили роботу над Selenium в Google. З'явилося

багато версій, включаючи довгоочікувану версію Selenium 3.0, яка визначила напрямок до Selenium 4.0 та 5.0.

Завдяки їхній праці на сьогодні Selenium представляє собою сукупність багатьох бібліотек, написаних на різних мовах програмування, хоча головним чином на кросплатформенній мові Java.

Серед численних програм, предназначених для автоматизації тестування, особливий акцент слід зробити на серії програмних продуктів від Selenium. Цей набір включає в себе такі компоненти як Selenium IDE, Selenium Grid, Selenium RC, Selenium Server та Selenium WebDriver.

Selenium WebDriver – це гнучкий інструмент для автоматизації тестування веб-проєктів, який базується на наборі бібліотек для різних мов програмування, таких як Java, .Net (C#), Python, Ruby, PHP, Perl та JavaScript.

Цей інструмент є платформонезалежним і підтримує операційні системи Windows, macOS та Linux. Крім того, він взаємодіє із найпоширенішими браузерами, такими як Google Chrome, Firefox, Safari, Edge, Internet Explorer, а також з деякими браузерами без графічного інтерфейсу.

Selenium WebDriver використовується найчастіше для реалізації різних видів тестування, включаючи регресійне та функціональне тестування[16]. Його універсальність у підтримці різних мов програмування та браузерів робить його ефективним інструментом для автоматизації тестування веб-додатків.

Процес автоматизації тестування веб-проєктів дуже простий. Замість того, щоб вручну виконувати одні й ті ж самі дії у браузері, використовують webdriver браузера. Цей webdriver спілкується з браузером і виконує необхідні дії, використовуючи скрипти з бібліотеки Selenium. Наприклад, це може бути пошук елементів, перехід за посиланнями, збір великих об'ємів даних (парсинг), натискання кнопок та інші дії. Все це відбувається за допомогою JSON Wire Protocol, що забезпечує взаємодію між бібліотекою та веб-драйвером.

Цей інструмент є потужним і дуже популярним серед тестувальників, але, як і будь-яке інше програмне забезпечення, має свої переваги та недоліки.

Переваги:

- Інтеграція та Кросплатформність: Підтримка великої кількості мов програмування та можливість використання на різних операційних системах.
- Простота використання: Легкий синтаксис команд і зручність у створенні скриптів за допомогою бібліотек.
- Автоматизація: Можливість виконання авто-тестів без участі людини і у будь-який зручний час.
- Відкритий вихідний код: Безкоштовний продукт із відкритим вихідним кодом, що дозволяє спільноті активно співпрацювати у його розвитку.

Недоліки:

- Необхідність програмування: Вимагає наявності навичок програмування для ефективного використання.
- Обмежений функціонал: Має обмежений функціонал порівняно з платними аналогами.
- Обмеження в тестуванні графічних та Flash-елементів: Не може бути використаний для тестування графічних елементів та Flash-об'єктів.
- Можливі дефекти у бібліотеках: Іноді можуть виникати проблеми або дефекти у самих бібліотеках, що впливає на їхню ефективність.

1.5.2. Appium

Appium – це потужний інструмент для кросплатформного автоматизованого мобільного тестування, який відкриває безліч можливостей для тестувальників у сфері мобільного розроблення. Забезпечуючи широкий спектр можливостей, Appium є не тільки інструментом для автоматизації

тестування, але й важливим фактором для розширення міжплатформенної сумісності в області мобільних додатків. [17]

Його кросплатформність означає, що один і той же тестовий сценарій може легко та ефективно працювати як на платформі iOS, так і на Android. Використовуючи Appium, розробники та тестувальники можуть максимально оптимізувати свій час та ресурси, забезпечуючи високу ефективність та надійність тестових процесів на обох популярних мобільних платформах.

Однією з ключових переваг Appium є його відкритість та гнучкість. Завдяки цим якостям, інструмент легко інтегрується з різними технологіями та мовами програмування, надаючи користувачам великий простір для вибору оптимального середовища для їхніх потреб. Appium визначає новий стандарт у сфері автоматизованого мобільного тестування, роблячи його ефективним та доступним для розробників на різних етапах розробки мобільних додатків.

Переваги використання Appium:

- Доступ до баз даних та внутрішніх API: Appium дозволяє тестувальникам та користувачам легко отримувати доступ до баз даних тестового коду та внутрішніх API, спрощуючи взаємодію з внутрішніми компонентами додатку.
- Можливість використання різних мов програмування: здатність використовувати будь-яку мову програмування, яка сумісна з веб-драйвером (наприклад, Java, Objective-C, JavaScript), дозволяє писати тестові приклади на мові, яка найбільше відповідає конкретним потребам користувача.
- Підтримка будь-якого фреймворку: Appium підтримує будь-який фреймворк, надаючи гнучкість у виборі та використанні тестових інструментів.
- Легка налаштування на різних платформах: легкість налаштування Appium на різних платформах дозволяє ефективно використовувати його на різноманітних операційних системах.

- Підтримка різних мов програмування: Appium підтримує різні мови програмування, включаючи Ruby, Java, PHP, Node, та Python, надаючи можливість вибору мови, яка найбільше відповідає вимогам проекту.
- Необхідності в модифікації додатку: використання Appium не вимагає модифікації самого додатку, забезпечуючи його стабільність та незалежність від тестового процесу.
- Використання дротового протоколу Selenium Webdriver JSON: можливість використовувати дротовий протокол Selenium Webdriver JSON дозволяє ефективно взаємодіяти з браузерами та забезпечує стандартизований підхід до автоматизації тестування.
- Необхідності повторної компіляції на інших платформах: використання Appium не вимагає повторної компіляції мобільного додатку для роботи на інших платформах, що прискорює процес розгортання тестів.
- Інтеграція з іншими інструментами: за допомогою Java можливо легко інтегрувати Appium з іншими інструментами, розширюючи його функціональність та забезпечуючи гармонійну роботу в комплексі з іншими інструментами розробки та тестування.

Серед недоліків можна відзначити той факт, що автоматичні тести можуть припинити свою ефективну роботу при оновленні операційної системи. [18] Це означає, що після кожного оновлення ОС необхідно чекати на відповідне оновлення інструментів для розробки автоматизованих сценаріїв. Такий підхід вимагає часу та утримання сумісності інструментів з новими версіями операційних систем.

Ще одним недоліком є складність написання автоматизованих сценаріїв. Наприклад, для написання одного сценарію для веб-додатку в середньому потрібно близько 8 годин, а для мобільного додатку - приблизно 20 годин. Це може створювати значний трудоголізм для розробників та

тестувальників, особливо при необхідності швидкого оновлення або написання нових тестових сценаріїв.

Також важливо відзначити нестабільну роботу багатьох інструментів, що може призвести до непередбачуваних проблем у процесі автоматизації тестування. Це вимагає постійного вивчення та вдосконалення вибраних інструментів для забезпечення їхньої надійної та стабільної роботи в різних умовах.

Висновки

В цьому розділі було досліджено та порівняно основні види тестування, його методики, а також системи, що використовуються для тестування систем на різніх платформах та операційних системах

РОЗДІЛ 2
ОПИС ВИМОГ ДО СИСТЕМИ

Вимоги до програмного забезпечення (англ. Software Requirements) - це опис того, як повинна функціонувати або які властивості повинно мати програмне забезпечення. Вимоги є важливою частиною процесу розробки програмного забезпечення і визначають, як програма повинна взаємодіяти з користувачем, іншими програмами та системами. Вимоги до програмного забезпечення зазвичай встановлюються на етапі аналізу вимог і є важливим інструментом для забезпечення того, що розроблене програмне забезпечення відповідає потребам користувачів і вимогам бізнесу.

Аналіз вимог є важливим етапом у розробці програмного забезпечення, спрямованим на збір, визначення, систематизацію та формалізацію вимог до системи. Цей процес допомагає розробникам і стейкхолдерам зрозуміти, що має бути реалізовано в програмному продукті. Основні кроки аналізу вимог включають наступне:

- Збір вимог: на цьому етапі проводиться активна взаємодія зі стейкхолдерами, такими як користувачі, клієнти та менеджери, з метою збору вимог для подальшої розробки програмного продукту. Також розглядається документація та існуючі системи для збору додаткової інформації.
- Аналіз вимог: на цьому етапі вимоги систематизуються та уточнюються, враховуючи пріоритети та важливість кожної вимоги. Також проводиться аналіз для виявлення можливих конфліктів чи непорозумінь між різними стейкхолдерами.
- Документування вимог: на цьому етапі вимоги записуються в документах, що будуть використовуватися для наступних етапів проекту. Забезпечується зрозумілість та повнота опису вимог.
- Валідація вимог: на цьому етапі вимоги перевіряються на відповідність бізнес-цілям та можливостям технічної реалізації. Також важливо залучити стейкхолдерів для підтвердження чи коригування вимог.
- Управління змінами вимог: на цьому етапі встановлюються механізми управління змінами для ефективного врахування будь-яких змін у вимогах, які можуть виникнути під час розробки.

Аналіз вимог є ітеративним процесом, інформація збирається, аналізується та переглядається протягом різних етапів розробки програмного забезпечення. Цей процес допомагає створити чітке розуміння того, що потрібно в розроблюваному продукті та як він повинен функціонувати.

2.1. Вимоги до системи

Вимоги до системи є визначенням основних параметрів та характеристик, які має мати інформаційна система для досягнення своїх цілей та відповідання потребам користувачів чи бізнес-завданням. Ці вимоги формулюються на початкових етапах розробки системи та служать основою для подальшого проектування, розробки, тестування та впровадження

програмного продукту чи іншої інформаційної системи. Вимоги до системи деталізують функціонал, операційні можливості, аспекти безпеки, продуктивність та інші аспекти, які є ключовими для успішної реалізації інформаційного проекту. Одним із важливих аспектів є чіткість та конкретність вимог, які допомагають уникнути непорозумінь та забезпечують зручний фундамент для подальшого процесу розробки.

Функціональні вимоги до системи визначають конкретні функції та операції, які повинна виконувати система для задоволення потреб користувачів чи бізнес-цілей. Ці вимоги описують, як система має взаємодіяти з користувачем, обробляти та зберігати дані, виконувати обчислення та інші дії, необхідні для функціонування. Бізнес-логіка та операційні вимоги формулюють основні принципи, за якими система повинна працювати, враховуючи конкретні бізнес-процеси та вимоги користувачів. Вимоги цього типу є важливим керівним документом, який визначає функціональний обсяг та можливості системи з метою її ефективної реалізації.

2.1.1. Бізнес-логіка

Бізнес-логіка визначається як набір правил, процедур і процесів, які визначають, як конкретна організація функціонує та взаємодіє зі своїм оточенням. Це сукупність логічних інструкцій та умов, що визначають, які дії має виконати система у різних сценаріях. Бізнес-логіка часто моделює бізнес-процеси та вимоги і визначає, як вони повинні бути автоматизовані в програмному забезпеченні. Вона відображає специфічні правила та умови, які дозволяють системі вирішувати конкретні завдання та досягати мети бізнесу.

Для забезпечення ефективності тестування програмних систем важливо дотримуватись правильної бізнес-логіки. Перш за все, бізнес-логіка повинна бути чіткою та зрозумілою, щоб тестувальники могли правильно інтерпретувати та перевіряти її. Також важливо мати повний набір вимог до

бізнес-логіки, які включають різні сценарії та умови, щоб впевнитись, що всі можливі випадки були враховані. Забезпечення правильності та вичерпності бізнес-логіки сприяє вчасному виявленню та виправленню помилок, а також полегшує підтримку системи в подальшому.

У разі змін у бізнес-логіці, слід актуалізувати відповідні вимоги та документацію. Чіткість і консистентність документації, пов'язаної з бізнес-логікою, полегшує процес тестування та сприяє співпраці між командами розробників і тестувальників. Важливо також визначити відповідальності за оновлення бізнес-логіки та взаємодію зі змінами в інших частинах системи, щоб уникнути конфліктів та забезпечити її стабільність під час тестування та експлуатації.

Врахування бізнес-логіки в процесі тестування допомагає забезпечити, що програмне забезпечення відповідає реальним бізнес-потребам та вимогам. Це сприяє створенню ефективних тест-кейсів, які враховують усі можливі сценарії використання та забезпечують високий рівень якості продукту під час його розгортання та експлуатації.

Дотримання певних принципів бізнес-логіки у застосунку може значно підвищити ефективність тестування програмних систем. Ось кілька важливих аспектів бізнес-логіки, які варто враховувати:

- Бізнес-логіка повинна бути чіткою та легко зрозумілою: тестувальники повинні мати чітке уявлення про те, як система повинна функціонувати, щоб створювати ефективні тест-кейси та виявляти можливі помилки.
- Забезпечення повноти вимог до бізнес-логіки: всі можливі сценарії використання повинні бути визначені, і відповідні тест-кейси повинні покривати всі ці сценарії для впевненості в стабільності та надійності системи.
- При змінах у бізнес-логіці слід актуалізувати відповідну документацію: якщо виникають зміни у вимогах чи бізнес-процесах,

цю інформацію слід вносити в документацію та відразу попереджати тестувальників.

- Переконання в тестуванні всіх можливих сценаріїв використання: кожен сценарій повинен бути перевірений для впевненості в тому, що система поводить себе правильно навіть у найскладніших умовах.
- Стриманість внесення змін та спрощення бізнес-логіки, коли це можливо: зміни в бізнес-логіці можуть впливати на тест-кейси, тому важливо уникати непотрібної складності та зберігати стабільність вимог.
- Чітко визначити відповідальність за оновлення бізнес-логіки та документації: це дозволяє уникнути конфліктів та забезпечити однозначність в процесі змін.

Дотримання цих принципів сприяє покращенню ефективності тестування програмних систем, спрощує процес виявлення та виправлення помилок, а також полегшує підтримку та розвиток системи в подальшому.

2.1.2. Вимоги користувача

Вимоги користувача визначають очікування, потреби та специфічні вимоги, які має враховувати програмний продукт для задоволення користувачів. Ці вимоги формулюються на ранніх етапах розробки для створення системи, яка відповідає очікуванням та вимогам кінцевих користувачів. Вимоги користувача можуть бути як функціональними, так і нефункціональними, описуючи, як система має працювати та якими характеристиками вона повинна володіти для задоволення потреб користувачів.

Функціональні вимоги користувача стосуються конкретних функцій та операцій, які користувачі очікують від системи. Наприклад, це може бути можливість реєстрації та входу в систему, додавання та редагування інформації, виконання певних завдань тощо. Нефункціональні вимоги

можуть включати швидкодію, зручність використання, безпеку, масштабованість та інші характеристики, які визначають якість системи з погляду користувача.

Для підвищення ефективності тестування програмних систем важливо дотримуватись конкретних вимог користувача. По-перше, вимоги повинні бути чіткими та однозначними, щоб тестувальники могли належним чином їх інтерпретувати. По-друге, вони повинні бути вимірюваними та перевірюваними, щоб забезпечити можливість створення ефективних тест-кейсів та критеріїв прийняття.

Також важливо забезпечити повноту вимог та врахування всіх можливих сценаріїв використання, щоб тестування включало у себе усі аспекти функціональності та властивостей системи. Документація вимог користувача повинна бути відомою та легко доступною для всіх учасників процесу тестування, щоб уникнути непорозумінь та забезпечити однозначність вимог.

Уникання непередбачуваних змін у вимогах під час процесу розробки та визначення відповідальностей за актуалізацію вимог може сприяти стабільності тестування та забезпечити зручність підтримки системи в подальшому. Дотримання цих підходів допомагає створити ефективний тестовий процес, який відображає реальні потреби та очікування користувачів.

Дотримання певних принципів та вимог користувача у застосунку може значно покращити ефективність тестування програмних систем. Ось декілька ключових аспектів, які слід враховувати:

- Чіткість та однозначність вимог: вимоги користувача повинні бути чіткими, зрозумілими та однозначними. Тестувальники повинні чітко розуміти очікування користувачів щодо функціональності та характеристик системи для ефективного визначення тест-кейсів та критеріїв прийняття.

- Міряність та перевірюваність: вимоги повинні бути міряними та перевірюваними. Вони повинні мати конкретні критерії успішності, які можна перевірити під час тестування. Це дозволяє створювати об'єктивні тест-кейси та легко визначати, чи відповідає система вимогам.
- Повнота та зручність документації: документація вимог користувача повинна бути повною та зручною в користуванні. Тестувальники повинні мати доступ до докладної та актуальної інформації для ефективного розуміння функціональних та нефункціональних вимог.
- Специфікація сценаріїв використання: сценарії використання повинні бути деталізовані та специфіковані. Вони допомагають створити тест-кейси, які враховують усі можливі шляхи взаємодії користувачів з системою.
- Стабільність та управління змінами: уникання раптових змін у вимогах під час розробки та чітке управління змінами. При непередбачуваних змінах вимог можуть виникати непорозуміння та витрати на переробку тест-кейсів.
- Взаємодія із зацікавленими особами: активна взаємодія зі зацікавленими особами, включаючи користувачів. Регулярний зворотний зв'язок дозволяє виправляти помилки на ранніх етапах і підтримує відповідність вимогам.

2.1.3. Функціональні вимоги

Функціональні вимоги - це частина вимог до системи, яка описує конкретні функції, операції та функціональність програмного забезпечення. Ці вимоги визначають, як система повинна взаємодіяти з користувачем, іншими системами чи компонентами, які функції повинна виконувати, і яким чином. Вони формулюють основні здатності програми та визначають, яким чином вона повинна вирішувати бізнес-завдання чи проблеми користувачів. Для підвищення ефективності наведений загальний перелік функціональних

вимог, які можуть бути важливими для забезпечення ефективності тестування:

1. Функції тестування:
   - Автоматизація тестів: забезпечення можливості автоматизації тестових сценаріїв для прискорення процесу тестування та зменшення його трудомісткості.
   - Масштабованість тестів: здатність ефективно виконувати тестування при збільшенні обсягу функціональності або обсягу даних.
2. Звітність та моніторинг:
   - Генерація звітів: можливість автоматично генерувати детальні та зрозумілі звіти з результатами тестів.
   - Моніторинг продуктивності: функціонал для відстеження та аналізу продуктивності під час тестування.
3. Управління конфігурацією:
   - Зберігання конфігурацій: можливість зберігати конфігурації тестового середовища та відновлення їх для відтворення умов тестування.
4. Інтеграція та сумісність:
   - Інтеграція з іншими інструментами: здатність інтегруватися з іншими інструментами розробки, тестування та управління проектом.
   - Тестування сумісності: можливість визначати та тестувати сумісність з різними операційними системами, браузерами чи апаратними платформами.
5. Зручність для користувача:
   - Інтуїтивний інтерфейс: забезпечення зручного та інтуїтивно зрозумілого інтерфейсу для створення та виконання тестових сценаріїв.

- Можливість швидкого запуску тестів: забезпечення можливості швидкого запуску окремих тестів або груп тестів.

6. Тестування безпеки:

- Забезпечення тестів на безпеку: функціональність для проведення тестів на вразливості та аспекти безпеки програмного продукту.

Дотримання цих принципів допомагає зробити тестування більш ефективним, забезпечуючи точні та об'єктивні критерії для оцінки якості функціональності програм.

2.1.4. Нефункціональні вимоги

Нефункціональні вимоги визначають аспекти якості та характеристики системи, які не пов'язані безпосередньо з конкретними функціональностями програмного продукту, але визначають його загальну ефективність, безпеку, доступність та інші аспекти. Ці вимоги визначають "якість" системи та те, як користувачі взаємодіють із нею.

- Продуктивність: нефункціональні вимоги до продуктивності стосуються швидкодії та ефективності системи в різних умовах завантаження. Це може включати час відгуку, швидкість обробки запитів, витрати ресурсів та інші метрики. Наприклад, система повинна забезпечувати швидкий час завантаження сторінок або ефективну обробку великого обсягу даних.
- Безпека: нефункціональні вимоги до безпеки визначають рівень захищеності системи від несанкціонованого доступу, атак та витоків інформації. Це може включати вимоги до криптографічної стійкості, контролю доступу, аудиту безпеки та інших заходів для забезпечення конфіденційності, цілісності та доступності даних.
- Надійність: надійність визначає, наскільки стабільною та безперебійною є система. Нефункціональні вимоги до надійності можуть включати метрики такі, як середній час між відмовами

(mtbf), відсоток часу роботи без відмов (uptime) та можливість відновлення після помилок.

- Доступність: доступність визначає, наскільки легко та ефективно користувачі можуть отримати доступ до системи. Це може включати вимоги до часу відновлення після відмови (rto), забезпечення резервних копій даних та інші заходи для забезпечення доступності системи для користувачів у різних умовах.

Нефункціональні вимоги є важливою частиною специфікації проекту, оскільки вони визначають параметри якості та ефективності, які визначають узгодженість системи з очікуваннями користувачів та бізнес-вимогами.

2.2. Вимоги до безпеки

Вимоги до безпеки програмного забезпечення є невід'ємною частиною процесу розробки, визначаючи стандарти та умови, які програма повинна відповідати для забезпечення найвищого рівня захисту від потенційних загроз та атак. Ці вимоги визначаються в контексті конкретної системи, її призначення та особливостей взаємодії з іншими компонентами.

Вимоги до безпеки програмного забезпечення охоплюють такі ключові аспекти, як аутентифікація, авторизація, шифрування даних, контроль доступу, моніторинг та виявлення інцидентів, а також відновлення після порушень. Аутентифікація та авторизація визначають, як користувачі отримують доступ до системи та її ресурсів. Шифрування забезпечує конфіденційність даних під час їх передачі. Контроль доступу встановлює права та обмеження на рівні користувача. Моніторинг та виявлення інцидентів спрямовані на вчасне виявлення порушень безпеки, а процеси відновлення дозволяють системі швидко відновити нормальну роботу після інциденту.

Основне призначення вимог до безпеки полягає в забезпеченні конфіденційності, цілісності та доступності даних та ресурсів програми. Конфіденційність гарантує, що лише авторизовані користувачі мають доступ

до конфіденційної інформації. Цілісність забезпечує, що дані залишаються недоторканими та невідмінними від несанкціонованих модифікацій. Доступність гарантує, що програма може надавати свої послуги у визначений час та в обсягах, які визначені бізнес-потребами.

Вимоги до безпеки тісно пов'язані з іншими етапами розробки програмного забезпечення. Вони враховуються при проектуванні архітектури, встановленні механізмів тестування та визначенні стратегій впровадження. Забезпечення безпеки включає в себе також регулярне оновлення та аудит системи на предмет виявлення та виправлення можливих вразливостей.

Вимоги до безпеки програмного забезпечення визначають набір умов, стандартів та процедур, які програмне забезпечення повинно виконувати для забезпечення високого рівня захисту від ризиків та загроз безпеці. Для веб-застосунків, які взаємодіють з користувачами через Інтернет, вимоги до безпеки є критично важливим елементом, оскільки вони спрямовані на захист від можливих атак, зловживань та витоків інформації.

Основні аспекти вимог до безпеки для веб-застосунків:

1. Аутентифікація та авторизація:
    - Аутентифікація: забезпечення ефективного процесу перевірки ідентифікації користувачів, часто включаючи використання паролів, біометричних методів та двофакторної аутентифікації.
    - Авторизація: визначення прав доступу користувачів до різних ресурсів та функцій відповідно до їхніх ролей та повноважень.
2. Захист введених даних:
    - Зашифрування передачі даних між користувачем та сервером за допомогою протоколів https для захисту конфіденційної інформації, такої як логіни, паролі та особисті дані.
3. Захист від кросс-сайт атак:

- Вимоги до фільтрації та обробки введених даних для запобігання кросс-сайт скриптінгу (xss) та кросс-сайт запросам піддавання (csrf).

4. Механізми захисту бази даних:
   - Захист баз даних від sql-ін'єкцій та інших атак на безпеку даних.
5. Моніторинг та ведення журналу:
   - Вимоги до систем моніторингу та журналювання для вчасного виявлення та реагування на можливі порушення безпеки.
6. Захист від атак на безпеку:
   - Захист від різноманітних атак, таких як внедрення коду, витоки інформації та деніал сервіс атаки (dos).
7. Відновлення після порушень:
   - Вимоги до процедур відновлення після порушень, включаючи резервне копіювання даних та можливості відновлення системи після інцидентів безпеки.
8. Оновлення та патчі:
   - Вимоги до системи оновлень та вчасного встановлення патчів для виправлення виявлених вразливостей.
9. Бізнес-контингентність:
   - Вимоги до планів бізнес-контингентності, включаючи відновлення послуг та зберігання резервних копій даних.
10. Захист від інших атак:
    - Вимоги до захисту від інших атак, таких як внедрення шкідливих файлів, атаки на сесії та інші потенційні загрози.

Вимоги до безпеки для веб-застосунків варіюються залежно від характеру даних, які обробляються, і взаємодії з іншими системами чи користувачами. Ці вимоги спрямовані на захист конфіденційності, цілісності та доступності даних та забезпечення високого рівня довіри до веб-

застосунку. Серед них варто розглянути наступні вимоги яких варто дотримуватись для забезпечення веб-застосунку безпекою:

- Контроль доступу та аутентифікація: забезпечення ефективної системи контролю доступу та аутентифікації є ключовою безпековою вимогою. Це включає в себе встановлення чітких політик аутентифікації, використання сильних паролів, двофакторної аутентифікації та обмеження доступу до конфіденційних ресурсів лише авторизованим користувачам.
- Шифрування даних та захист від кросс-сайт атак: застосунок повинен використовувати шифрування для захисту конфіденційних даних під час їх передачі мережею. Додатково, важливо вбудовувати заходи захисту від кросс-сайт атак (xss) та кросс-сайт запитів піддавання (csrf), щоб запобігти впровадженню зловмисного коду та несанкціонованих дій на сторінках веб-застосунку.
- Захист від sql-ін'єкцій та інших атак на базу даних: важливо забезпечити валідацію та екранування введених даних для запобігання sql-ін'єкціям. Додатково, застосунок повинен використовувати підготовлені вирази та параметризовані запити для запобігання атакам на базу даних. Це допоможе уникнути витоків чутливої інформації та несанкціонованого доступу.

Ці три безпекові вимоги становлять фундаментальний базис для забезпечення безпеки веб-застосунку та захисту від різних видів загроз.

2.2.1. Аутентифікація та авторизація

Аутентифікація та авторизація - це дві ключові складові систем безпеки, які визначають, як система взаємодіє з користувачами та яким чином контролює доступ до своїх ресурсів. Аутентифікація визначає, чи є користувач легітимним у системі, тобто чи він той, за кого видає себе. Авторизація визначає рівень доступу, який має користувач після того, як він успішно аутентифікувався.

Безпекові вимоги до аутентифікації включають в себе використання сильних методів перевірки ідентичності, таких як паролі великої довжини, двофакторна аутентифікація або біометричні методи. Авторизація вимагає чітких політик та ролей, щоб забезпечити, що користувачі отримують лише той рівень доступу, який необхідний для їхніх обов'язків.

Для ефективної аутентифікації, система повинна використовувати криптографічно безпечні хеш-функції для зберігання паролів, а також механізми для виявлення та запобігання атакам, таким як перебор паролів. Додатково, використання двофакторної аутентифікації, такої як коди або апаратні токени, підвищує рівень безпеки.

Для авторизації, система повинна використовувати чітко визначені ролі та політики, щоб призначати та контролювати рівень доступу для кожного користувача. Авторизаційні механізми повинні бути вбудовані в рівень додатку, і система повинна виявляти та реагувати на намагання несанкціонованого доступу.

Обидві ці процеси повинні бути добре задокументовані та піддаватися періодичній перевірці та оновленню. Розуміння контексту додатку, чутливості даних та потреб користувачів є ключовим для правильної реалізації аутентифікації та авторизації. Важливо також регулярно аудитувати систему для виявлення та усунення можливих вразливостей.

2.2.2. Захист від Кросс-Сайт атак

Крос-сайт атаки (XSS) є формою загрози веб-безпеці, де зловмисник впроваджує зловісний код в веб-сторінку, спрямовану на виконання у браузері користувача. Ці атаки можуть виникати через введення тексту, URL-параметри або вразливості в коді сторінки, порушуючи конфіденційність та цілісність веб-застосунку. Існують різні види крос-сайт атак:

- Крос-сайт скриптінг (XSS) - збоку клієнта. Це враження великі ризики безпеки, коли зловмисник вбудовує скрипти прямо в веб-сторінку, використовуючи нефільтровані дані введення користувача.

- Крос-сайт скриптінг (XSS) - збоку сервера. Зловмисник використовує вразливості серверного коду для внесення зловісного коду, який потім відправляється користувачам, порушуючи безпеку веб-застосунку.
- Крос-сайт фреймінг (XSF). Ця атака включає вбудовування веб-сторінки в інший фрейм або iframe, що може порушити контекст та безпеку веб-застосунку.

Крос-сайт атака (XSS) — це процес, коли зловмисник вставляє зловісний код у веб-сторінку, яку потім виконує браузер користувача. Основні способи реалізації включають в себе вбудовування скриптів у введення користувача, використання несанкціонованих URL-параметрів та атаки на вразливості в коді веб-сторінки. Це може призвести до викрадання сесій, зміни вмісту сторінок або перенаправлення на інші шкідливі ресурси.

Наступні заходи сприяють зменшенню ризиків XSS-атак та забезпеченню веб-застосунку високого рівня безпеки.

- Валідація та екранування даних: ефективне застосування валідації та екранування введених даних перед їх обробкою у веб-застосунку. Це допомагає уникнути впровадження зловісного коду через нефільтровані вхідні дані.
- HTTP-заголовок content security policy (CSP): встановлення політики безпеки контенту, яка обмежує джерела, з яких можна завантажувати ресурси. Це запобігає виконанню зловісних скриптів з інших джерел.
- Використання HTTPS: захист від атак посереднього перегляду та модифікації даних шляхом використання шифрування трафіку за допомогою протоколу HTTPS.
- Захист від крос-сайт фреймінгу (XSF): використання заголовків, таких як `x-frame-options`, щоб контролювати те, як ваш сайт вбудовується в інші фрейми, уникнутий атак xsf.

- Використання безпечних бібліотек та фреймворків: використання безпечних фреймворків та бібліотек, які мають захист від XSS-атак та регулярно оновлюються.
- Захист кукі-файлів: встановлення атрибутів для кукі-файлів, таких як `HttpOnly` та `secure`, для зменшення ризику витоку інформації та забезпечення безпеки сесій.

Також існує шифрування даних що відіграє важливу роль у захисті конфіденційності інформації, особливо під час передачі через мережі на пристроях. Цей процес забезпечує перетворення зрозумілої інформації в кодовану форму, яку може розкодувати лише особа або система з правильним ключем. Симетричне шифрування використовує один і той самий ключ для шифрування та розшифрування, тоді як асиметричне використовує пару ключів: публічний і приватний. Використання шифрування даних допомагає уникнути несанкціонованого доступу та забезпечує таємницю передачі чутливої інформації.

Шифрування застосовується в різних контекстах для забезпечення безпеки інформації. Одним з найпоширеніших використань є шифрування даних на зберігальних пристроях, таких як жорсткі диски або USB-накопичувачі, що зменшує ризик втрати конфіденційної інформації при втраті або крадіжці пристрою. У сфері передачі даних, застосування протоколів шифрування, таких як SSL/TLS в інтернет-з'єднаннях, дозволяє захищати від перехоплення та зміни інформації під час передачі між клієнтом і сервером. Шифрування також знаходить застосування в захисті електронної пошти, фінансових транзакцій та інших сферах, де зберігання чи передача конфіденційної інформації є критичною. Є два види шифрування:

- Симетричне шифрування: використовує один і той самий ключ як для шифрування, так і розшифрування даних. Сучасні алгоритми, такі як AES, забезпечують високий рівень безпеки.

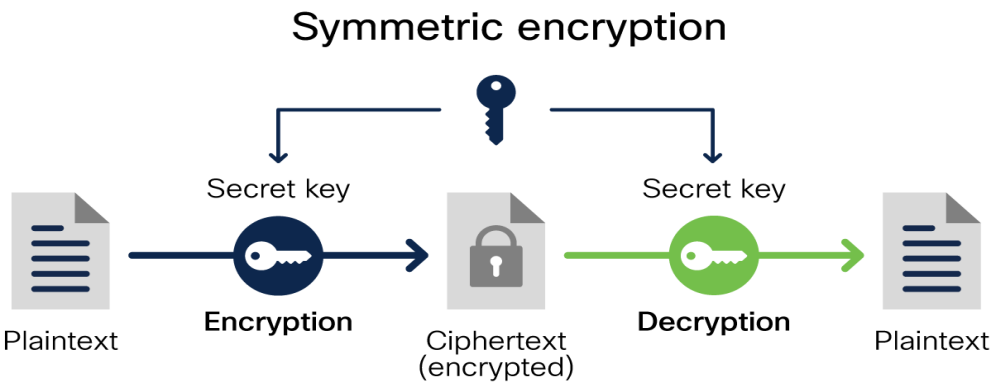

Рис. 2.1 Схема роботи симетричного шифрування

- Асиметричне шифрування: застосовує два ключі: публічний та приватний, що дозволяє безпечно обмінюватись даними. RSA є одним з найпоширеніших асиметричних алгоритмів.

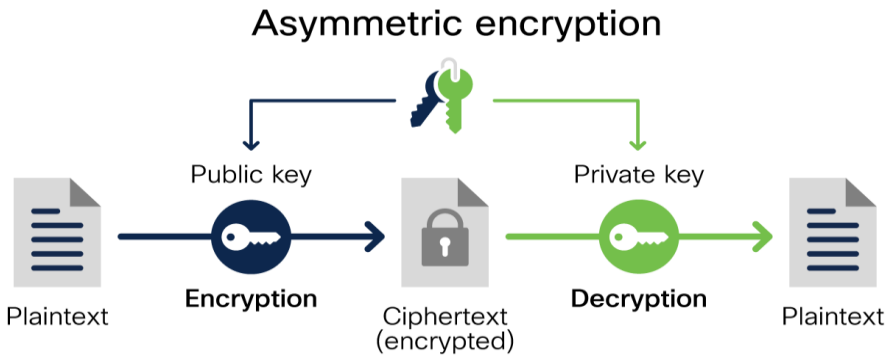

Рис. 2.2 Схема роботи асиметричного шифрування

Веб-додатки часто використовують шифрування для забезпечення безпеки та конфіденційності. Один із основних аспектів цього застосування полягає в захисті передачі даних між клієнтом і сервером за допомогою протоколу HTTPS (SSL/TLS). Це забезпечує шифрування інформації, що передається через мережу, запобігаючи можливість перехоплення та читання конфіденційних даних.

Додатково, шифрування використовується для захисту сесій користувачів, уникання атак XSS через використання атрибута `HttpOnly` для кукі-файлів та забезпечення безпеки даних в базі даних від несанкціонованого доступу. Цей комплексний підхід сприяє стійкості та надійності веб-додатків у цифровому середовищі.

2.2.3. Захист від SQL-ін'єкцій

База даних - це структурована колекція даних, організованих таким чином, щоб легко здійснювати пошук, оновлення та управління інформацією. Вона може включати таблиці, які зберігають дані, і взаємозв'язки між цими таблицями. Бази даних використовуються для ефективного зберігання та обробки інформації в різних областях, від бізнесу до науки.

SQL - це стандартна мова запитів для роботи з реляційними базами даних. Вона дозволяє виконувати різні операції, такі як вибірка (SELECT), вставка (INSERT), оновлення (UPDATE) та видалення (DELETE) даних. SQL дозволяє ефективно маніпулювати даними в базах даних і використовується широкою кількістю реляційних систем управління базами даних (СУБД).

SQL-ін'єкції - це вид атак, який полягає в введенні зловмисного SQL-коду в поля вводу веб-додатків, що взаємодіють з базами даних. Зловмисники використовують цей метод для виконання несанкціонованих операцій у базі даних або отримання конфіденційної інформації. Найчастіше це відбувається через форми вводу на веб-сайтах, де злоумисник вводить SQL-код, який потім виконується в контексті бази даних. Відсутність належного фільтрування та валідації введених даних може створити вразливість для таких атак. Захист від SQL-ін'єкцій включає в себе коректне використання параметризованих запитів та валідацію введених даних.

Один з ключових методів захисту від SQL-ін'єкцій - це використання параметризованих запитів. Замість конкатенації строкових значень для формування SQL-запиту, веб-розробники повинні використовувати параметри, які передаються до бази даних. Це дозволяє системі правильно

інтерпретувати дані, уникати можливості виконання зловісного коду введеними користувачем даними та підвищує безпеку запитів.

Важливо валідувати та екранувати введені дані перед їх використанням у SQL-запитах. Валідація допомагає перевірити коректність введених даних, а екранування забезпечує правильне форматування та обробку символів, які можуть викликати SQL-ін'єкції. Врахування цих заходів виключає можливість використання спеціальних символів, що можуть завдати шкоди базі даних.

Принцип найменших прав передбачає, що користувальницькі облікові записи або ролі мають лише ті дозволи, які необхідні для виконання їхніх функцій. Якщо користувальникові не потрібно виконувати операції запису, то його обліковий запис не повинен мати відповідних дозволів. Це обмеження прав користувачів зменшує можливість успішних атак SQL-ін'єкцій, оскільки навіть якщо злоумисник отримає доступ до системи, йому буде важко виконати деякі небезпечні операції.

Дотримання практик захисту від SQL-ін'єкцій відіграє ключову роль у підтримці веб-застосунків і забезпеченні їх безпеки та надійності. Ось як це може допомогти:

- Забезпечення безпеки даних: захист від SQL-ін'єкцій дозволяє забезпечити конфіденційність та цілісність даних в базі даних. Врахування параметризованих запитів та валідація введених даних запобігає несанкціонованому доступу до інформації та недозволеним змінам.
- Запобігання втрати довіри користувачів: коли веб-застосунок захищений від SQL-ін'єкцій, користувачі можуть бути впевнені в безпеці своїх даних. Це сприяє підтримці довіри користувачів до сервісу та позитивному сприйняттю веб-застосунку.
- Зменшення ризику втрати фінансів та репутації: веб-застосунки, які дотримуються практик захисту від SQL-ін'єкцій, зменшують ризик фінансових втрат та негативного впливу на репутацію. Уникнення

втрати конфіденційних фінансових даних чи особистої інформації користувачів сприяє стабільності та довірі до веб-застосунку.

- Виключення зниження продуктивності та доступності: атаки SQL-ін'єкцій можуть призвести до втрати продуктивності та доступності веб-застосунку. Захист від цих атак дозволяє уникнути втрати ресурсів, пов'язаних з непередбаченими атаками та відновленням працездатності системи.
- Відповідність законодавчим вимогам: забезпечення безпеки та конфіденційності даних у веб-застосунку допомагає відповідати законодавчим вимогам та стандартам, таким як GDPR чи інші регулятивні рамки. Це особливо важливо в сучасному середовищі, де охорона приватності користувачів має велике значення.

2.3. Вимоги до інтерфейсу

Інтерфейс - це засіб спілкування людини з комп'ютером. Критерії оцінки інтерфейсу повинні охоплювати три основних аспекти: простота освоєння і запам'ятовування операцій системи, швидкість досягнення цілей завдання розв'язуваної за допомогою системи, суб'єктивна задоволеність при експлуатації системи. [19]

Інтерфейс забезпечує зв'язок між користувачем і процесом виконання деякого завдання. Інтерфейс дає можливість користувачеві визначити, які завдання зробити активними, передати їм дані і отримати результати обробки. Інтерфейс містить дві компоненти: процеси введення-виведення і процес діалогу. Користувач взаємодіє з інтерфейсом, посилає вхідні дані, приймає вихідні. Інтерфейс ініціює процес виконання завдання. В процесі діалогу відбувається обмін повідомленнями. До основних пристроїв введення-виведення відносяться клавіатура, миша, дисплей, сканери, мовне введення і інші. Вхідні повідомлення можуть являти собою команди і дані. Введення може виконуватися зазначенням, виділенням і безпосередньо введенням. Вихідні повідомлення представляють підказки, інформацію про

стан, повідомлення про помилки, довідки. Діалог може управлятися системою (меню) і може управлятися користувачем.

Вимоги до інтерфейсу програмного забезпечення визначають стандарти та характеристики, які повинен відповідати користувацький інтерфейс програми. Ці вимоги орієнтовані на створення зручного, ефективного та зрозумілого інтерфейсу для полегшення ефективної взаємодії користувача з програмою. Вони включають в себе такі аспекти, як ергономіка та зручність використання, сумісність із користувацькими преференціями, відповідність дизайну та стилю, доступність та універсальність, а також ефективну взаємодію та зворотний зв'язок. Забезпечення естетичної привабливості та інтуїтивності навігації також входить у вимоги до інтерфейсу для підтримки позитивного досвіду використання користувачами. Дизайн інтерфейсів складається з двох основних частин:

2.3.1. UI Design

У галузі промислового дизайну взаємодії людини з комп’ютером інтерфейс користувача (UI) — це простір, де відбувається взаємодія між людьми та машинами. Мета цієї взаємодії полягає в тому, щоб забезпечити ефективну роботу та контроль машини з боку людини, в той час як машина одночасно передає інформацію, яка допомагає операторам приймати рішення. Приклади цієї широкої концепції інтерфейсів користувача включають інтерактивні аспекти комп’ютерних операційних систем, ручних інструментів, засобів керування оператором важкого обладнання та керування процесами. Міркування щодо дизайну, які застосовуються під час створення інтерфейсів користувача, пов’язані з такими дисциплінами, як ергономіка та психологія, або включають такі дисципліни.

Загалом, метою проектування інтерфейсу користувача є створення інтерфейсу користувача, який робить легким, ефективним і приємним (зручним для користувача) керування машиною таким чином, щоб отримати

бажаний результат (тобто максимальну зручність використання). Загалом це означає, що оператор повинен забезпечити мінімальний вихід для досягнення бажаного результату, а також те, що машина мінімізує небажані результати для користувача.

Інтерфейси користувача складаються з одного або кількох рівнів, включаючи інтерфейс людина-машина (HMI), який зазвичай поєднує машини з фізичним вхідним обладнанням (таким як клавіатури, миші або ігрові панелі) та вихідним обладнанням (таким як комп'ютерні монітори, динаміки та принтери). Пристрій, який реалізує HMI, називається пристроєм людського інтерфейсу (HID). Користувальницькі інтерфейси, які обходяться без фізичного руху частин тіла як проміжного кроку між мозком і машиною, не використовують пристрої введення чи виведення, окрім лише електродів; їх називають інтерфейсами мозок–комп'ютер (BCI) або інтерфейсами мозок–машина (BMI).

Процес проектування інтерфейсу користувача — це послідовність кроків та етапів, спрямованих на створення ефективного та зрозумілого дизайну для взаємодії користувача з продуктом, веб-сайтом чи програмою. Цей процес включає такі ключові етапи:

1. Розуміння контексту:
    - Дослідження користувачів та бізнес-потреб: вивчення характеристик цільової аудиторії та вимог бізнесу для визначення функціональних та дизайнерських вимог.
    - Аналіз конкурентів: оцінка дизайнерських рішень конкурентів для визначення трендів та виробництва конкурентноспроможного інтерфейсу.
2. Розробка інформаційної архітектури:
    - Створення структури сайту/продукту: визначення логічної структури інформації та організації контенту для полегшення навігації користувача.

- Створення схеми потоку роботи: моделювання послідовності дій користувача для оптимізації взаємодії та досягнення бажаного результату.

3. Створення прототипу:
    - Дизайнерський прототип: створення інтерактивного макету, що демонструє основні елементи інтерфейсу та взаємодію користувача.
    - Тестування прототипу: збирання фідбеку від користувачів для вдосконалення дизайну перед реалізацією.
4. Вибір дизайну та кольорової палітри:
    - Графічний дизайн: розробка естетичного інтерфейсу, включаючи вибір шрифтів, графічних елементів та кольорової палітри.
    - Створення зразків (UI Kit): розробка набору стандартних елементів для забезпечення консистентності.
5. Розробка програмного забезпечення:
    - Інтеграція дизайну: перенесення графічного дизайну та інтерфейсу в реальний код.
    - Тестування та оптимізація: перевірка функціональності та вирішення можливих проблем, оптимізація продукту для покращення продуктивності та швидкості.
6. Тестування та збір фідбеку:
    - A\B-тестування: порівняння ефективності різних варіантів інтерфейсу.
    - Проведення тестування користувачів: оцінка реакції та взаємодії реальних користувачів з продуктом.
7. впровадження та моніторинг:
    - Запуск продукту: введення розробленого інтерфейсу в експлуатацію.

- Моніторинг та оновлення: відстеження ефективності та внесення змін для вдосконалення продукту з плином часу.

Цей цикл дозволяє забезпечити ефективну інтерфейсну взаємодію з користувачем та підтримує вдосконалення продукту на всіх етапах його життєвого циклу.

Ключові принципи та практики дизайну інтерфейсу користувача визначають ефективність, зручність та естетичність продукту для оптимальної взаємодії з користувачем. Основні принципи та практики включають:

1. Простота та ясність:
    - Мінімалізм: зменшення зайвих елементів та подробиць для полегшення сприйняття.
    - Ясність та логічність: інтуїтивне розташування елементів та послідовність дій.
2. Ефективність використання:
    - Швидкість доступу: забезпечення швидкого доступу до ключових функцій та інформації.
    - Мінімізація кроків: спрощення процесів та зменшення кількості кроків для виконання завдань.
3. Консистентність:
    - Стандартизація елементів: використання однакового стилю та розташування елементів для підтримки консистентності.
    - Використання UI Kit: застосування набору стандартних елементів для однакового дизайну.
4. Доступність:
    - Різнобарвність: врахування потреб користувачів із обмеженими можливостями через використання контрастних кольорів та адаптивних інтерфейсів.

- Доступ до вмісту: забезпечення можливості отримання інформації користувачами з різними потребами.

5. Взаємодія та зворотний зв'язок:
    - Зручність введення: забезпечення ефективного та комфортного введення даних (наприклад, через екран сенсорного дисплею).
    - Повідомлення та підказки: надання користувачеві зрозумілого зворотного зв'язку та допоміжних підказок.
6. Гнучкість та адаптабельність:
    - Адаптивний дизайн: створення інтерфейсу, який ефективно адаптується до різних розмірів екранів та пристроїв.
    - Персоналізація: надання можливості користувачам налаштовувати деякі аспекти інтерфейсу згідно з власними вподобаннями.
7. Графічний дизайн та естетика:
    - Сучасний вигляд: використання сучасних графічних рішень для створення естетично приємного інтерфейсу.
    - Єдність стилю: збереження єдності стилю в усьому інтерфейсі.

Ці принципи та практики є основою успішного дизайну інтерфейсу користувача, сприяючи вдосконаленню взаємодії та задоволенню користувачів. Розробка інтерфейсу користувача є ключовим елементом успішного продукту, веб-сайту або програми. Високоякісний дизайн інтерфейсу є важливим для забезпечення позитивного досвіду користувача (UX). Для досягнення цього мета важливо володіти основами дизайну інтерфейсу користувача, використовувати передові практики та інструменти, які дозволять створити ефективний дизайн, що відповідає потребам користувачів. Ключовими принципами є простота, послідовність і доступність, а також забезпечення зворотного зв'язку та створення візуальної ієрархії.

2.3.2. UX Design

Взаємодія з користувачем (UX) — це те, як користувач почувається під час взаємодії з продуктом, системою чи послугою. Він включає уявлення людини про корисність, легкість використання та ефективність. Покращення взаємодії з користувачем є важливим для більшості компаній, дизайнерів і творців під час створення та вдосконалення продуктів, оскільки негативний досвід користувача може зменшити використання продукту та, отже, будь-який бажаний позитивний вплив; навпаки, проектування, спрямоване на прибутковість, часто суперечить цілям етичної взаємодії з користувачем і навіть завдає шкоди. Взаємодія з користувачем суб'єктивна. Однак атрибути, які складають користувацький досвід, є об’єктивними.

Першою вимогою до зразкового досвіду користувача є відповідність точним потребам клієнта, без суєти чи турботи. Далі йдуть простота та елегантність, які створюють продукти, якими приємно володіти та користуватися. Справжня взаємодія з користувачем виходить далеко за рамки надання клієнтам того, що вони кажуть, що вони хочуть, або надання функцій контрольного списку. Щоб досягти високої якості користувальницького досвіду в пропозиціях компанії, має бути безперебійне злиття послуг багатьох дисциплін, включаючи інженерію, маркетинг, графічний і промисловий дизайн, а також дизайн інтерфейсу.

Важливо відрізняти загальний досвід користувача від користувальницького інтерфейсу (UI), навіть якщо UI є надзвичайно важливою частиною дизайну. Як приклад розглянемо веб-сайт із оглядами фільмів. Навіть якщо користувальницький інтерфейс для пошуку фільму ідеальний, UX буде поганим для користувача, який хоче отримати інформацію про невеликий незалежний випуск, якщо основна база даних містить лише фільми великих студій.

Ми також повинні розрізняти UX і юзабіліті: згідно з визначенням юзабіліті, це атрибут якості інтерфейсу користувача, який визначає, чи є система легкою для вивчення, ефективною у використанні, приємною тощо.

Знову ж таки, це дуже важливо, і знову ж таки загальний досвід користувача є ще ширшим поняттям.

Ключові принципи та практики дизайну користувальницького досвіду (User Experience Design) визначають способи створення задоволення та позитивного враження від взаємодії користувача з продуктом чи сервісом. Основні принципи та практики включають:

1. Розуміння користувача:
    - Дослідження користувачів: глибоке вивчення потреб та поведінки цільової аудиторії для адаптації продукту до їхніх очікувань.
    - Створення персонажів користувачів: розробка образів користувачів для кращого розуміння їхніх уявлень та потреб.
2. Інформаційна архітектура:
    - Організація інформації: створення логічної структури для забезпечення зручної навігації та знаходження інформації.
    - Створення схем потоку роботи: моделювання послідовності дій користувача для оптимізації взаємодії.
3. Дизайн інтерфейсу:
    - Мінімалізм та простота: забезпечення лаконічного та зрозумілого дизайну для полегшення сприйняття інтерфейсу.
    - Використання кольорів та типографії: створення приємного вигляду за допомогою гармонійних кольорів та шрифтів.
4. Ефективність використання:
    - Прискорення завдань: спрощення процесів та зменшення кількості кроків для виконання завдань.
    - Налаштування: надання можливостей користувачам налаштовувати інтерфейс відповідно до їхніх потреб.
5. Доступність:

- Адаптивний дизайн: розробка інтерфейсу, що ефективно адаптується до різних пристроїв та екранів.
- Доступ до вмісту: забезпечення можливості взаємодії для користувачів із різними потребами.

6. Взаємодія та зворотний зв'язок:
    - Емоційна зв'язаність: створення позитивних емоцій та зв'язаності користувача з продуктом.
    - Зручність введення: покращення ефективності та зручності введення даних.
7. Тестування та оптимізація:
    - Тестування користувачів: залучення реальних користувачів для отримання фідбеку та виявлення можливих вдосконалень.
    - Аналіз метрик: відстеження показників використання для постійного вдосконалення продукту.
8. Інтеграція задоволення та видовища:
    - Графічний дизайн: використання естетичних елементів для створення привабливого вигляду.
    - Підсилення бренду: впровадження елементів, що відображають бренд та його цінності.

Ці принципи та практики допомагають дизайнерам UX створювати продукти, які не лише відповідають потребам користувачів, але й створюють позитивний та задовільний досвід взаємодії.

Дотримання усіх вище вказаних принципів дизайну користувальницького досвіду (UX) може бути досягнуте за допомогою ряду конкретних практик:

- Інтерв'ю та анкетування користувачів: проведення інтерв'ю та анкетування для отримання важливої інформації про потреби та очікування користувачів.

- Створення прототипів: розробка прототипів для інтерактивного тестування та збору фідбеку від користувачів на ранніх етапах.
- Картографування користувальницького досвіду: створення карт користувальницького досвіду для візуалізації всього циклу взаємодії користувача з продуктом.
- Тестування користувачів: регулярне проведення тестів з реальними користувачами для оцінки ефективності та зручності взаємодії.
- Картографування потоку роботи: створення схем потоку роботи для аналізу та оптимізації послідовності дій користувача.
- Застосування сучасних методів дизайну: використання сучасних методів дизайну, таких як Design Thinking чи Lean UX, для стимулювання інновацій та врахування потреб користувачів.
- Аналіз зворотного зв'язку: систематичний аналіз отриманого зворотного зв'язку від користувачів та впровадження відповідних вдосконалень.
- Використання Design Systems та UI Kits: застосування готових дизайн-систем та UI Kit для створення консистентного та єдновиглядного інтерфейсу.
- Моніторинг аналітики: використання аналітичних інструментів для відстеження поведінки користувачів та оцінки ефективності дизайну.
- Співпраця між командами: активна співпраця між дизайнерами, розробниками та іншими членами команди для забезпечення інтеграції ux в усі етапи розробки.

Ці практики сприяють створенню високоякісного та ефективного користувальницького досвіду, а також дозволяють адаптувати продукт до змінних потреб користувачів та ринкових умов.

Висновки

У другому розділі проекту були уточнені ключові вимоги до розроблюваної системи. Уважно проаналізовані бізнес-вимоги, вимоги користувача, а також функціональні та нефункціональні вимоги, включаючи вимоги до інфраструктури, на якій має функціонувати продукт. Зокрема, особлива увага приділена забезпеченню надійності системи, що вимагало детального розгляду вимог до безпеки, таких як використання аутентифікації та авторизації, захист від Кросс-Сайт атак та захист від SQL-Ін'єкцій. Докладно розглянуті принципи дизайну інтерфейсу (UI) та користувальницького досвіду (UX), оскільки вони визначають ефективність та зручність взаємодії користувача з продуктом. Особливий акцент робився на збалансованості між функціональністю та естетикою, наголошуючи на необхідності створення гармонійного дизайну. Загалом, чітко визначені вимоги служать основою для подальшого проектування та розробки системи, забезпечуючи створення зручного та ефективного інструменту для користувачів при збереженні високого рівня безпеки.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ ТЕСТУВАННЯ

3.1. Архітектура застосунку

Для цього застосунку була використана трирівнева архітектура. Трирівнева архітектура є найбільш поширеною концепцією організації взаємодії в інтернет-система. [20] Виникла внаслідок розширення дворівневої архітектури, де серверна частина була розділена на дві ключові складові: шар логіки та шар даних. Ця еволюція архітектурного підходу стала найбільш широко використовуваною та ефективною для забезпечення оптимальної взаємодії у мережі Інтернет.

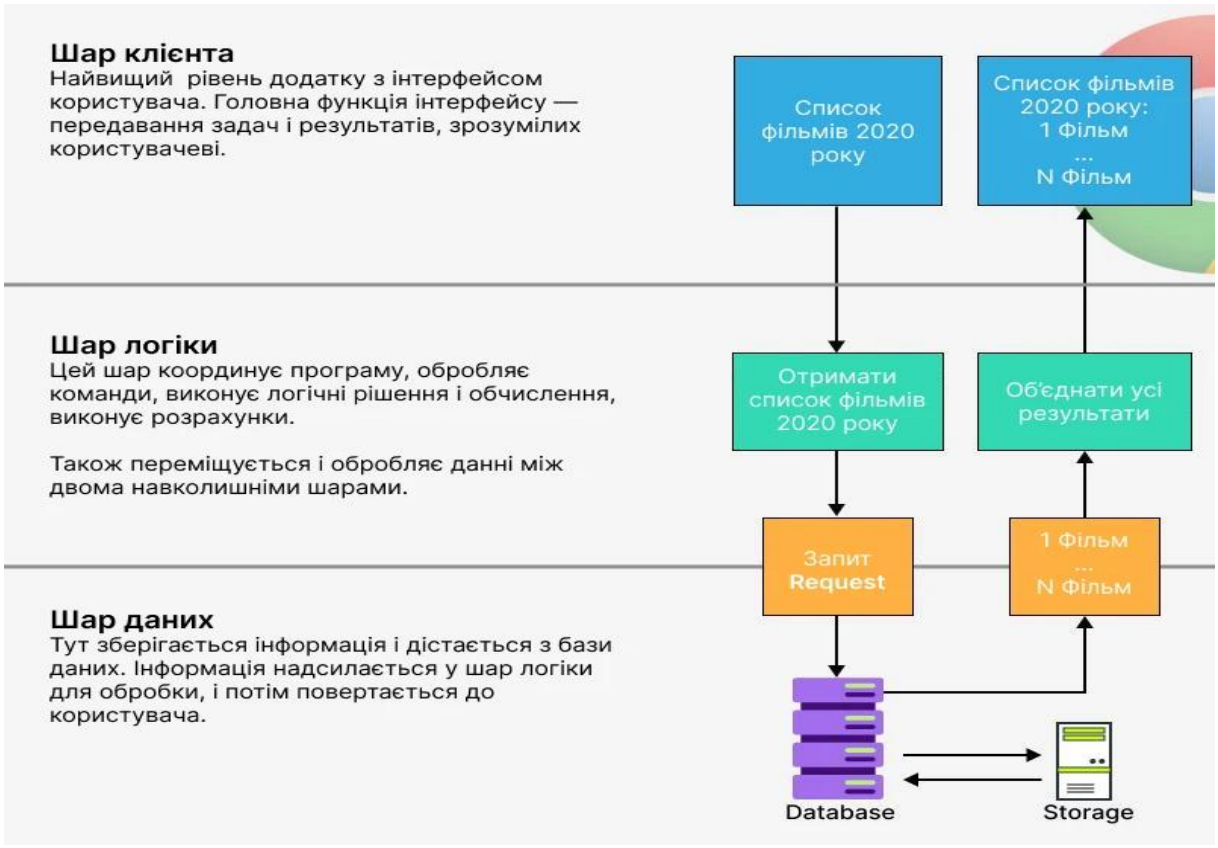

Рис. 3.1. Схематичне зображення трирівневої архітектури

Шар клієнта відноситься до складової "розподіленої програми", що відповідає за взаємодію з користувачем. У цьому шарі не допускається

присутність бізнес-логіки або зберігання критично важливих даних. Також необхідно уникати прямого взаємодії цього шару з базою даних і обмежити цю взаємодію лише через бізнес-логіку. Незважаючи на це, шар клієнта включає в себе певну логіку. По-перше, це взаємодія з користувачем через інтерфейс, валідація введених даних та робота з локальними файлами. Сюди також входить все, що стосується авторизації користувача та шифрування даних під час спілкування з сервером. По-друге, тут реалізована проста бізнес-логіка. Наприклад, у випадку інтернет-магазину, коли надходить список товарів, клієнтський шар може їх відсортувати та відфільтрувати. Також тут враховано елементарне зберігання даних, таке як кешування, обробка кукі для залогінених користувачів тощо.

Шар бізнес-логіки розташований на другому рівні і включає значну частину бізнес-логіки системи. Відокремлюються лише фрагменти, які експортуються на клієнта, а також елементи логіки, взаємодія яких здійснюється з базою даних (збережені процедури та тригери).

Нерідко сервери бізнес-логіки несуть не лише основну логіку, але й вирішують завдання масштабування: код розбивається на частини і розподіляється між різними серверами. Деякі надзвичайно популярні сервіси можуть функціонувати на десятках серверів, а їх обсяг навантаження регулює load balancer.

Проектування серверних програм часто враховує можливість легко запустити ще одну копію сервера, яка почне взаємодіяти з іншими копіями. Таким чином, навіть під час написання серверного коду немає гарантій, що статичний клас буде запущено у єдиному екземплярі.

Шар даних відповідає за зберігання даних і розташовується на окремому рівні, зазвичай реалізованому за допомогою систем управління базами даних (СУБД). З'єднання з цим компонентом здійснюється лише з рівня сервера застосунків.

Відділення шару даних у повноцінний третій рівень стало результатом численних факторів, причиною яких, безперечно, є зростання навантаження на сервер.

По-перше, бази даних вимагали значних ресурсів пам'яті та процесорного часу для ефективної обробки даних. Це призвело до їх виносу на окремі сервери. При збільшенні навантаження на бекенд легко було створити дублікати і підняти декілька копій одного сервера. Однак дублювання бази даних було непрактичним, оскільки база продовжувала залишатися єдиною і нероздільною складовою системи.

По-друге, бази даних отримали власну бізнес-логіку, підтримуючи збережені процедури, тригери і власні мови, такі як PLSQL. Навіть з'явилися програмісти, які писали код, який виконується всередині СУБД. Логіку, не пов'язану з даними, виносили в бекенд і запускали паралельно на десятках серверів. Все, що було критично пов'язано з даними, залишалося всередині СУБД. Проблеми зі збільшеним навантаженням тут вирішувалися різними методами:

- Кластер бази даних: Група серверів БД, які зберігають одні й ті ж дані і синхронізують їх за певним протоколом.
- Шардування: Дані діляться на логічні блоки і розподіляються на різні сервери БД, що може бути складним у випадку змін у структурі БД.
- Підхід NoSQL: Зберігання даних у БД, спеціально спроектованих для масштабного зберігання великих обсягів інформації, часто як файлові сховища з обмеженим функціоналом порівняно з реляційними базами даних.
- Кешування даних: З'явилися СУБД для кешування, які зберігали результати лише в пам'яті, замість простого кешу на рівні бази даних.

Це розмаїття серверних технологій вимагало спеціалізованих команд або окремих фахівців для ефективного управління, що призвело до винесення шару даних в окремий рівень.

Як окремий випадок трирівневої архітектури, у цьому застосунку було використано архітектуру MVC. Архітектура MVC (Model-View-Controller) дозволяє розділити проект на три ключові частини: модель, представлення та контролер, забезпечуючи можливість модифікувати кожен компонент незалежно від інших. [21]

Додатки, побудовані за принципом MVC, складаються з трьох важливих частин. Давайте розглянемо кожну з них більш детально.

Модель надає дані та методи їх обробки, такі як запити до бази даних та перевірка коректності введених даних. Модель не залежить від представлення та контролера (не має взаємодії з користувачем), і вона має доступ до даних та може їми керувати.

Основні риси моделі:

- Бізнес-логіка додатку.
- Містить інформацію про себе і не має інформації про контролер та представлення.

Для деяких проектів це є шаром даних (DAO, БД, XML-файл), а для інших - менеджером бази даних, набором об'єктів або просто логікою додатку.

Представлення відповідає за відображення необхідних даних з моделі та їхнє надсилання користувачеві. Представлення не обробляє введені дані користувача.

Основні риси представлення:

- Реалізує відображення даних, отриманих від моделі.
- У деяких випадках представлення може містити код, що реалізує певну бізнес-логіку.

Прикладами представлення можуть бути HTML-сторінка, WPF-форма, Windows Forms тощо.

Контролер забезпечує зв'язок між системою та користувачем, контролює та направляє дані від користувача до системи та навпаки, використовуючи модель та представлення для реалізації необхідної дії.

Основні риси контролера:

- Визначає, які представлення повинні бути відображені в даний момент.
- Події представлення можуть впливати тільки на контролер; контролер може впливати на модель та визначати інше представлення.
- Можливість створення декількох представлень для одного контролера.

Також цей застосунок дотримується монолітної архітектури. Монолітна архітектура представляє собою традиційний підхід до розробки програмного забезпечення, де весь додаток розробляється як єдина технологічна система. [22] У цій архітектурі всі компоненти додатку взаємодіють один з одним, а розгортання відбувається на одному сервері або групі серверів. Монолітна архітектура має свої переваги. Зазвичай вона простіша в розробці та тестуванні, оскільки всі компоненти додатку взаємодіють безпосередньо, що сприяє швидкому виявленню та вирішенню проблем. Крім того, такі додатки можуть бути менш складними в управлінні, оскільки вони працюють на єдиній технологічній платформі.

Проте монолітні додатки можуть стикатися з проблемами масштабування, оскільки розширення системи може бути ускладненим через взаємозалежність компонентів. Крім того, складна структура монолітних додатків може утруднити розробку та підтримку.

Отже, монолітна архітектура є ефективною для невеликих та середніх проектів із стабільним функціоналом, в той час як мікросервісна архітектура виявляється більш відповідною для великих та складних проектів, які можуть зростати в майбутньому та потребують гнучкості у розширенні та масштабуванні.

3.2. Технології, що застосовуються

Щоб створити цей додаток, було використано систему управління базами даних PostgreSQL. Backend частину було реалізовано з використанням .NET Core та ASP .NET Core та ASP .NET Core Web API. Також, для взаємодії з системою управління базами даних було використано Entity Framework Core. Для тестування бізнес-логіки backend частини було використано бібліотеку тестування nUnit та бібліотеку Moq.

Frontend частина використовує HTML 5 та CSS 3. Було застосовано CSS бібліотеку Tailwind для спрощення створення елементів дизайну та CSS пре-процесорна мова SASS. Логіку відображення було реалізовано з React 18.2.0 з використанням React Router для навігації між сторінками та React Redux для зберігання стану застосунку.

3.2.1. Система управління базами даних PostgreSQL

PostgreSQL - це потужна об'єктно-реляційна система баз даних з відкритим вихідним кодом, яка використовує і розширює мову SQL у поєднанні з багатьма функціями, що дозволяють безпечно зберігати і масштабувати найскладніші робочі навантаження даних. Витоки PostgreSQL сягають 1986 року як частина проекту POSTGRES в Каліфорнійському університеті в Берклі і має більш ніж 35-річну історію активного розвитку на основній платформі. [23]

PostgreSQL завоювала міцну репутацію завдяки своїй перевіреній архітектурі, надійності, цілісності даних, потужному набору функцій, розширюваності та відданості спільноти розробників з відкритим вихідним кодом, що стоїть за цим програмним забезпеченням, постійному створенню ефективних та інноваційних рішень. PostgreSQL працює на всіх основних операційних системах, є ACID-сумісною з 2001 року і має потужні доповнення, такі як популярний розширювач геопросторових баз даних

PostGIS. Не дивно, що PostgreSQL стала реляційною базою даних з відкритим вихідним кодом, яку обирають багато людей та організацій.

PostgreSQL постачається з багатьма функціями, покликаними допомогти розробникам створювати додатки, адміністраторам захищати цілісність даних і створювати відмовостійкі середовища, а також допомагати вам керувати даними незалежно від того, наскільки великий чи малий набір даних ви використовуєте. Крім того, що PostgreSQL є безкоштовною і з відкритим кодом, вона дуже легко розширюється. Наприклад, ви можете визначати власні типи даних, створювати власні функції і навіть писати код на різних мовах програмування без перекомпіляції бази даних.

PostgreSQL намагається відповідати стандарту SQL там, де така відповідність не суперечить традиційним можливостям або може призвести до поганих архітектурних рішень. Багато можливостей, що вимагаються стандартом SQL, підтримуються, хоча іноді з дещо відмінним синтаксисом або функціями. З часом можна очікувати подальших кроків у напрямку відповідності. Станом на випуск версії 16 у вересні 2023 року, PostgreSQL відповідає принаймні 170 з 179 обов'язкових функцій для відповідності стандарту SQL:2023 Core. На момент написання цієї статті жодна реляційна база даних не відповідає цьому стандарту повністю.

Говорячи мовою баз даних, PostgreSQL використовує модель клієнт/сервер. Сеанс PostgreSQL складається з наступних взаємодіючих процесів:

Процес сервера, який керує файлами бази даних, приймає з'єднання з базою даних від клієнтських додатків і виконує дії з базою даних від імені клієнтів. Програма сервера бази даних називається postgres.

Клієнтська (інтерфейсна) програма користувача, яка хоче виконувати операції з базою даних. Клієнтські програми можуть бути дуже різноманітними за своєю природою: клієнт може бути текстово-орієнтованим інструментом, графічною програмою, веб-сервером, який звертається до бази даних для відображення веб-сторінок, або спеціалізованим інструментом для

обслуговування бази даних. Деякі клієнтські програми постачаються з дистрибутивом PostgreSQL; більшість розробляються користувачами.

Як це характерно для клієнт-серверних додатків, клієнт і сервер можуть знаходитися на різних хостах. У такому випадку вони взаємодіють через мережеве з'єднання TCP/IP. Ви повинні мати це на увазі, тому що файли, до яких можна отримати доступ на клієнтській машині, можуть бути недоступні (або можуть бути доступні лише під іншими іменами) на машині сервера бази даних.

3.2.2. .NET Core та ASP .NET Core

.NET Framework, вимовляється як "дот-нет", представляє собою програмну технологію, розроблену компанією Microsoft як платформа для створення як звичайних програм, так і веб-застосунків. У багатьох аспектах це продовження ідей та принципів, вкладених у технологію Java. Однією з ключових концепцій .NET є можливість взаємодії служб, написаних на різних мовах. Навіть якщо ця можливість акцентується як особливість .NET від Microsoft, платформа Java також володіє подібними можливостями. .NET - це крос-платформова технологія, яка в даний час має реалізації для платформ Microsoft Windows, FreeBSD (розроблена Microsoft), а також варіант для ОС Linux у проєкті Mono (згідно з угодою між Microsoft і Novell), а також DotGNU. [24]

Забезпечення захисту авторських прав є ключовим аспектом середовища виконання (CLR - Common Language Runtime) для програм .NET. Компілятори для .NET розробляються різними компаніями для різних мов на вільній основі.

.NET розділяється на дві основні частини - це середовище виконання (фактично віртуальна машина) та інструментарій розробки. Так само, як і в технології Java, середовище розробки .NET генерує байт-код, призначений для виконання віртуальною машиною. Мова, яку ця машина використовує в .NET, відома як CIL (Common Intermediate Language) або MSIL (Microsoft

Intermediate Language), або просто IL. Використання байт-коду дозволяє досягти крос-платформенності на рівні скомпільованого проєкту (в термінах .NET: збірка), а не на рівні вихідного тексту, як у мові C, наприклад. Перед запуском збірки в середовищі виконання (CLR), байт-код перетворюється в машинний код цільового процесора вбудованим JIT-компілятором (just in time, компіляція на льоту).

.NET Core є безкоштовним крос-платформовим фреймворком з керованим кодом, який підтримується на операційних системах Windows, Linux і Mac OSX. У відміну від .NET Framework, вихідний код .NET Core повністю відкритий і доступний за наступним посиланням https://github.com/dotnet/core [Архівовано 11 лютого 2016 року у Wayback Machine.]. Цей фреймворк включає у себе CoreCLR - повністю крос-платформну реалізацію CLR, віртуальну машину, що відповідає за виконання програм в середовищі .NET. CoreCLR постачається з оптимізованим компілятором "just-in-time" RyuJIT. .NET Core також включає у себе CoreFX, яка є частковим відгалуженням FCL (стандартної бібліотеки класів .NET Framework).

Навпаки до .NET Framework, .NET Core має свій власний API, хоча і ділиться підмножиною API .NET Framework. [25] Також, .NET Core включає CoreRT - оптимізовану для інтеграції в AOT (компіляція перед виконанням) бібліотеку. Версія бібліотеки .NET Core використовується для UWP (універсальна платформа Windows), яка була створена Microsoft і вперше представлена в Windows 10. Метою цієї платформи є надання підтримки для створення універсальних додатків Windows, які запускаються як на Windows 10, так і на Windows 10 Mobile, без змін в коді. Інтерфейс командного рядка .NET Core служить точкою входу для операційних систем і надає розробникам послуги, такі як компіляція і управління пакетами. .NET Core підтримує чотири крос-платформові сценарії: веб-аплікації ASP.NET Core, консольні додатки, бібліотеки і додатки UWP (універсальна платформа Windows).

Важливо зазначити, що .NET Core не включає реалізації Windows Forms або WPF, які використовуються для створення стандартного графічного інтерфейсу для настільних ПК на Windows. Додатково, .NET Core є модульним, що означає, що розробники працюють з пакетами NuGet замість збірок. На відміну від .NET Framework, який отримує оновлення через службу Windows Update, .NET Core покладається на свого менеджера пакетів.

ASP.NET Core є технологією для розробки веб-додатків на платформі .NET, що активно розвивається компанією Microsoft. [26] Мови програмування, які використовуються для розробки додатків на ASP.NET Core, включають C# та F#.

Історія ASP.NET почалася з виходу першої версії, призначеної спочатку для роботи виключно в середовищі Windows на веб-сервері IIS. Однак вже в 2014 році відбулися значущі зміни, визначаючи новий етап у розвитку платформи. Компанія Microsoft взяла курс на створення кросплатформової технології ASP.NET як відкритого проекту. Ця еволюція платформи отримала назву ASP.NET Core, яку Microsoft офіційно утримує й донині. Перший реліз оновленої платформи відбувся у червні 2016 року. Тепер ASP.NET Core працює не лише на Windows, але й на MacOS та Linux. Вона стала більш легковажною, модульною, легше конфігурується та відповідає сучасним вимогам. Поточна версія ASP.NET Core, яка описана у цьому посібнику, була випущена разом із релізом .NET 7 у листопаді 2022 року. ASP.NET Core тепер повністю відкритий фреймворк. Поточну архітектуру платформи ASP.NET Core можна виразити так:

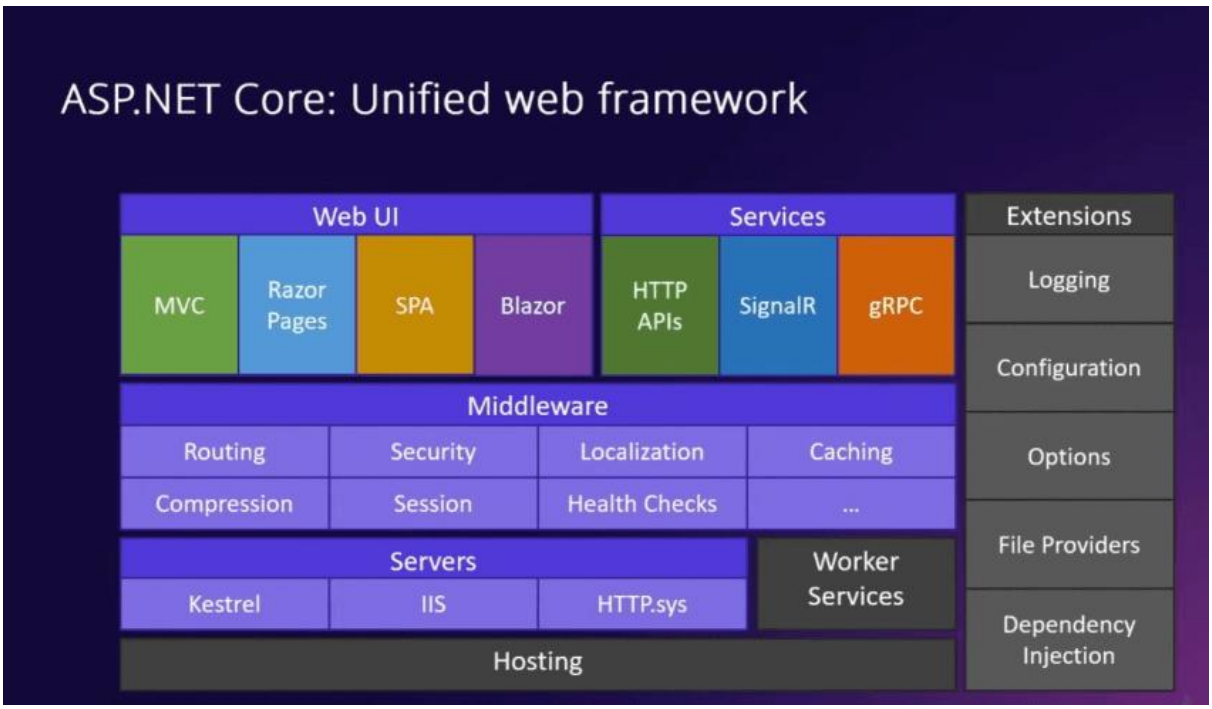

Рис. 3.2. Архітектура ASP .NET Core

На верхньому рівні розташовані різні моделі взаємодії з користувачем в архітектурі ASP.NET Core. Ці моделі включають технології побудови інтерфейсу користувача та обробки введення користувача, такі як MVC, Razor Pages, SPA (Single Page Application - односторінкові програми з використанням Angular, React, Vue) та Blazor. До цих технологій також відносяться послуги у вигляді вбудованих HTTP API, бібліотек SignalR або сервісів GRPC.

Усі ці технології базуються на чистому ASP.NET Core та/або взаємодіють з ним. ASP.NET Core включає різні вбудовані компоненти middleware, які використовуються для обробки запиту. Крім того, технології вищого рівня взаємодіють з різними розширеннями, такими як розширення для логування, конфігурації тощо. На нижньому рівні додаток ASP.NET Core працює в рамках певного веб-сервера, такого як Kestrel, IIS або бібліотека HTTP.sys. Опираючись на цю коротку архітектуру, розглянемо моделі розробки програм ASP.NET Core:

- Базовий ASP.NET Core: Підтримує всі основні аспекти, необхідні для роботи сучасного веб-додатка, такі як маршрутизація,

конфігурація, логування та робота з різними системами баз даних. Доданий концепт Minimal API - мінімізована спрощена модель для спрощення розробки та написання коду програми.

- ASP.NET Core MVC: Представляє модель побудови програми навколо трьох основних компонентів - Model (моделі), View (уявлення) та Controller (контролери). Моделі відповідають за роботу з даними, контролери - за логіку обробки запитів, а уявлення - за візуальну складову.
- Razor Pages представляє модель, у якому за обаботку запиту відповідають спеціальні сутності – сторінки Razor Pages. Кожну окрему таку сутність можна порівнювати з окремою веб-сторінкою.
- ASP.NET Core Web API представляє реалізацію патерну REST, у якому кожному за типу http-запиту (GET, POST, PUT, DELETE) призначений окремий ресурс. Такі ресурси визначаються як методів контролера Web API. Ця модель особливо підходить для односторінкових програм, але не тільки.
- Blazor представляє фреймворк, який дозволяє створювати інтерактивні програми як на стороні сервера, так та на стороні клієнта та дозволяє задіяти на рівні браузера низькорівневий код WebAssembly.

Інтерфейси прикладного програмування (API) - це, по суті, HTTP-сервіси, які використовуються для простого та централізованого зв'язку між додатками.

Корпорація Майкрософт за допомогою фреймворку ASP.NET надає способи створення веб-інтерфейсів API, доступ до яких можна отримати з будь-якого клієнта, наприклад, браузерів, десктопних додатків і мобільних пристроїв.

ASP.NET Web API можна використовувати для створення REST і SOAP сервісів. [27]

Нижче наведено деякі переваги роботи з ASP.NET Web API:

- Він працює так само, як і HTTP, використовуючи стандартні дієслова HTTP, такі як GET, POST, PUT, DELETE для всіх CRUD-операцій.
- Повна підтримка маршрутизації.
- Відповідь генерується у форматі JSON та XML за допомогою MediaTypeFormatter.
- Його можна розміщувати на IIS, а також автоматично розміщувати за межами IIS.
- Підтримує прив'язку та перевірку шаблонів.
- Підтримує шаблони URL і методи HTTP.
- Має просту форму введення залежностей.
- Може бути версійним.

ASP.NET Core Web API в основному складається з одного або декількох класів контролерів, які є похідними від ControllerBase. Клас ControllerBase надає багато методів і властивостей, корисних для роботи з HTTP-запитами.

3.2.3. Entity Framework Core

Entity Framework Core - це нова версія Entity Framework після EF 6.x. Це відкрита, легка, розширювана і крос-платформна версія технології доступу до даних Entity Framework. [28] Entity Framework - це фреймворк об'єктно-реляційного відображення (O/RM). Це розширення ADO.NET, яке надає розробникам автоматизований механізм доступу та зберігання даних у базі даних. EF Core призначений для використання з додатками .NET Core. Однак його також можна використовувати зі стандартними програмами на базі фреймворку .NET 4.5+. EF Core підтримує два підходи до розробки: 1) спочатку код 2) спочатку база даних. EF Core в основному орієнтований на підхід "спочатку код" і надає незначну підтримку підходу "спочатку база даних", оскільки візуальний дизайнер або майстер для моделі БД не

підтримується в EF Core. При підході "спочатку код" API EF Core створює базу даних і таблиці за допомогою міграції на основі угод і конфігурації, передбачених у ваших доменних класах. Цей підхід корисний при доменно-орієнтованому проектуванні (DDD). При підході "спочатку база даних" EF Core API створює доменні та контекстні класи на основі існуючої бази даних, використовуючи команди EF Core. Цей підхід має обмежену підтримку в EF Core, оскільки він не підтримує візуальний конструктор або майстер.

3.2.4. nUnit та Moq

NUnit - це середовище модульного тестування з відкритим вихідним кодом для .NET додатків. Його було перенесено з мови Java (бібліотека JUnit). [29] Перші версії NUnit були написані на J#, але потім весь код був переписаний на C# з використанням таких нововведень .NET, як атрибути. Існують також відомі розширення оригінального пакету NUnit, багато з яких також мають відкритий вихідний код. NUnit.Forms доповнює NUnit інструментами для тестування елементів інтерфейсу користувача Windows Forms. NUnit.ASP виконує ту ж саму задачу для елементів інтерфейсу в ASP.NET.

Moq є найпопулярнішою та найзручнішою бібліотекою для створення моків та стабів для .NET.

3.2.5. HTML, CSS, Tailwind та SASS

HTML - скорочення від "HyperText Mark-up Language" - перекладається як "Мова розмітка гіпертексту" (Гіпертекст - це текст, що не послідовно зв'язаний з іншими документами, тобто у вас є змога з першої сторінки документу перейти на останню). [30] Іншими словами HTML - це мова розмітки, або ще один спосіб зберігання інформації. За допомогою HTML ти позначаєш текст, вказуючи своєму веб-переглядачу, як він має розуміти позначений текст, так само як і на жорсткому диску інформація зберігається в блоках, кластерах, секторах, доріжках і тільки за допомогою, такої,

визначеної структури твій комп'ютер розуміє, що треба, а що не треба зчитувати.

У HTML текст позначається за допомогою теґів. Кожен HTML документ буде складатися з деякої групи елементів, де кожен елемент буде визначатися (починатися та закінчуватися) певним теґом (Для деяких елементів кінцевий теґ не є обов'язковим). Теґ — це назва елементу, записана у кутових дужках (< >)

Кожен HTML теґ має свою унікальну назву з визначеним синтаксисом, яка записується латинськими літерами і не чутливий до регістру. Елементи являють собою базові компоненти розмітки HTML. Кожен елемент має дві основні властивості: атрибути та вміст (контент). Існують певні настанови щодо кожного атрибута та контенту елемента, які треба виконувати задля того, щоб HTML-документ був визнаний валідним.

У елемента є початковий тег, який має вигляд <element-name>, та кінцевий тег, який має вигляд </element-name>. Атрибути елемента записуються в початковому тегу одразу після назви елемента, контент елемента записується між його двома тегами. Наприклад: <element-name element-attribute="attribute-value">контент елемента</element-name>.

Деякі елементи, наприклад br, не містять контенту, тож і не мають кінцевого тегу. Елемент може не мати початкового та кінцевого тегу (наприклад, елемент head), проте він завжди буде представлений в документі. Елементи структурної розмітки застосовують для опису семантики тексту, іншими словами ці елементи описують призначення тексту свого контенту. Вони не зазначають ніякого спеціального (візуального) відтворення тексту, проте більшість браузерів мають стандартні стилі форматування для кожного елемента. Для подальшого стилізування тексту рекомендується використовувати Каскадні таблиці стилів (CSS).

CSS (скорочення від Cascading Style Sheets, що перекладається як каскадні таблиці стилів) представляє собою спеціалізовану мову стилів, якою визначається зовнішній вигляд документів, а саме, як елементи веб-сторінки

повинні бути відображені. [31] Зазвичай CSS використовується для форматування документів, які написані мовами розмітки даних, такими як HTML, XHTML та XML. HTML відповідає за організацію інформації та створення структури документу, в той час як CSS відповідає за його оформлення, встановлюючи унікальні стилі для кожного елемента. Файл CSS містить правила форматування для всіх елементів на сторінці. Це дозволяє уникнути повторювання коду, що робить його більш зрозумілим. Одну таблицю стилів можна легко використовувати на безлічі сторінок, що суттєво прискорює процес верстки.

У кожному правилі у файлі міститься селектор і блок оформлення. Селектор визначає, до якої саме частини документа застосовується конкретне правило. Блок оформлення, у свою чергу, складається з пар «властивість: значення», які розміщуються в фігурних дужках і завершуються крапкою з комою. При наявності CSS на сторінці браузер кешує її, що призводить до зменшення часу завантаження сайту при наступних візитах.

CSS використовується веб-розробниками та користувачами для визначення кольорів, шрифтів, верстання та інших аспектів вигляду веб-сторінок. Однією з ключових переваг є можливість відокремлення вмісту сторінки (найчастіше HTML, XML або інша мова розмітки) від її вигляду (що описується в CSS). Це розділення поліпшує сприйняття та доступність контенту, надає більше гнучкості та контролю над відображенням у різних умовах, робить контент більш структурованим і зрозумілим, а також усуває непотрібні повторення. CSS також дозволяє адаптувати контент до різних умов відображення (на екрані монітора, мобільного пристрою, у роздрукованому вигляді, на екрані телевізора, пристроях з підтримкою шрифту Брайля чи голосових браузерах та інших). Один і той самий HTML або XML документ може відображатися по-різному в залежності від використаного CSS, надаючи гнучкість та можливість адаптації контенту до різноманітних умов.

Tailwind CSS - це, по суті, фреймворк CSS для швидкого створення користувацького інтерфейсу. Це низькорівневий CSS-фреймворк, який легко налаштовується і надає вам всі необхідні будівельні блоки. Крім того, це чудовий спосіб написання вбудованих стилів і досягнення чудового інтерфейсу без написання жодного рядка власного CSS.

Як ми знаємо, існує багато фреймворків CSS, але люди завжди обирають швидкий і простий фреймворк для вивчення і використання в проекті. Tailwind має безліч вбудованих функцій і стилів для користувачів на вибір, а також використовується для зменшення необхідності писати CSS-код і створення красивого користувацького інтерфейсу. [32] Це допоможе вам подолати складне завдання. Tailwind CSS створює невеликі утиліти з визначеним набором опцій, що дозволяють легко інтегрувати існуючі класи безпосередньо в HTML-код.

Переваги Tailwind CSS

- Легко налаштовується.
- Дозволяє створювати складні адаптивні макети.
- Адаптивність та розробка - це просто.
- Легко створювати компоненти.

Недоліки Tailwind CSS

- Відсутні заголовки та навігаційні компоненти.
- Потрібен час, щоб навчитися реалізовувати вбудовані класи.

Sass (англ. Syntactically Awesome Stylesheets) — це скриптова метамова, яка перетворюється в каскадні таблиці стилів (CSS). Цей інструмент був розроблений Гемптоном Кетліном та Наталі Вейзенбаум з метою підвищення рівня абстракції коду та спрощення файлів CSS. [33] Мова Sass має два синтаксиси:

- SASS (оригінальний): Відрізняється відсутністю фігурних дужок. В цьому синтаксисі вкладені елементи реалізовані за допомогою відступів, а правила відокремлюються переведенням рядка.

- SCSS (новий): Використовує фігурні дужки (подібно до CSS).

Файли з синтаксисом SASS мають розширення .sass, а файли зі синтаксисом SCSS — .scss. SASS розширює CSS, надаючи кілька механізмів, які доступні в більш традиційних мовах програмування, зокрема в об'єктно-орієнтованих мовах, але які відсутні у CSS. Інтерпретатор SASS транслює SASSScript у блоки правил CSS. У підсумку, SASS — це синтаксичний цукор для CSS, призначений для полегшення його написання та розуміння.

SASS - це надзвичайно ефективний спосіб покращення CSS. Завдяки використанню змінних, вкладеності та міксинів, ви можете досягти кращих результатів порівняно з чистим CSS. Замінивши CSS на SASS, ви отримуєте можливість використовувати код повторно, уникнувши необхідності писати одні й ті самі фрагменти знову і знову. Таким чином, використання SASS дозволяє зробити ваш код більш консистентним та зручним, роблячи вас більш сміливими у власних програмах.

3.2.6. React, Router та Redux

React JS — це відкрита бібліотека JavaScript, призначена для розробки інтерфейсів користувача. Створена компанією Facebook, вона швидко здобула популярність серед розробників з усього світу. [34] React дозволяє ефективно створювати застосунки з високою продуктивністю та масштабованістю. Однією з ключових концепцій у React JS є компоненти, що представляють собою незалежні блоки коду, відповідні за рендеринг певних частин користувацького інтерфейсу. React використовується для розробки користувацького інтерфейсу веб-додатків з метою створення інтерактивних, динамічних та відзивчивих інтерфейсів для користувачів. Frontend на React дозволяє ефективно створювати багатофункціональні та інтерактивні застосунки з швидким рендерингом і плавним переходом між сторінками. Основні переваги використання React включають:

- Використання віртуального DOM: Застосування віртуального DOM та ефективного алгоритму оновлення дозволяє робити мінімальні зміни в реальному DOM, покращуючи продуктивність застосунків.
- Компонентна архітектура: React ґрунтується на компонентній архітектурі, що дозволяє розбити інтерфейс на незалежні компоненти. Це спрощує розробку, тестування та підтримку коду.
- Односторонній потік даних: Пропагація даних у React відбувається в одному напрямку, що сприяє простоті та передбачуваності управління станом додатків.
- Активна спільнота розробників: Велика та активна спільнота розробників, яка використовує React, забезпечує наявність безлічі ресурсів, бібліотек та інструментів для розробки.
- Універсальність та масштабованість: React підходить для розробки проектів будь-якого масштабу, забезпечуючи можливості легкого розширення та перевикористання компонентів.

Загалом використання React дозволяє розробникам ефективно будувати потужні та швидкі інтерфейси, полегшуючи роботу з компонентами та станом додатків, і отримувати широку підтримку від спільноти розробників. Цей фреймворк ідеально підходить для командної розробки завдяки своєму дотриманню UI та шаблону робочого процесу.

Розробники використовують різноманітні інструменти для розробки, тестування, налагодження та оптимізації додатків на основі React. Вони обирають інструменти в залежності від потреб проекту та власних уподобань. Ось деякі з найпоширеніших інструментів:

- React Developer Tools: Це браузерне розширення, яке надає розробникам інструменти для інспектування, налагодження та профілювання React-додатків у веб-браузері. Воно дозволяє переглядати ієрархію компонентів, аналізувати їх стан та властивості, а також перевіряти швидкість оновлення компонентів.

- Редактори коду: Visual Studio Code, Sublime Text або Atom. Ці редактори мають розширення, плагіни та корисні функції, які полегшують роботу з React.
- Create React App: Цей набір інструментів допомагає швидко налаштувати новий проект, забезпечує зручний шаблон проекту та автоматично налаштовує середовище розробки.
- Бандлери модулів: Webpack або Parcel для збірки та пакування всього React-коду та його залежностей в один або кілька файлів JavaScript. Це дозволяє ефективно керувати залежностями, зменшує розмір файлів та поліпшує завантаження застосунку.
- Babel: Дозволяє використовувати нові функції та синтаксис JavaScript, які ще не підтримуються всіма браузерами.
- Фреймворки тестування: Jest, React Testing Library або Enzyme для написання тестів та перевірки функціональності компонентів.
- Git та SVN: Для керування кодом, роботи з репозиторіями та спільної розробки.

Вибір інструментів залежить від конкретних вимог проекту, і розробники мають можливість підібрати оптимальний набір для своєї роботи.

React Router — це бібліотека для реалізації маршрутизації в веб-додатках, побудованих з використанням бібліотеки React. [35] Вона дозволяє розробникам створювати односторінкові додатки з динамічною маршрутизацією, тобто без повторного завантаження сторінки при навігації користувача.

Основні поняття та можливості React Router:

- Маршрути: Розробники можуть визначити маршрути для різних частин додатка. Кожен маршрут пов'язаний з конкретним компонентом React, який буде відображений, коли користувач потрапляє на цей маршрут.

- Динамічна маршрутизація: Маршрутизація відбувається динамічно, коли користувач взаємодіє з додатком. Це означає, що зміна маршруту не призводить до повторного завантаження всієї сторінки.
- Односторінковий додаток (SPA): React Router дозволяє створювати SPA, де контент динамічно змінюється без перезавантаження сторінки.
- Управління історією браузера: Роутер дозволяє використовувати кнопки «Назад» та «Вперед» браузера, а також зберігає стан додатка при оновленні сторінки.
- Захист від спалаху білого екрана: Завдяки динамічній маршрутизації, React Router запобігає спалаху білого екрана, який може виникнути при повторному завантаженні сторінки.
- Вбудовані компоненти: React Router надає ряд вбудованих компонентів, таких як BrowserRouter, Route, Link, які спрощують роботу з маршрутизацією.

React Router дозволяє створювати динамічні та зручні для користувача додатки, де навігація відбувається швидко та безперервно.

Redux є потужною бібліотекою для управління станом додатків у JavaScript-додатках, особливо в контексті бібліотеки React. [36] Давайте розглянемо основні концепції Redux:

- Store (Сховище): Це централізоване сховище, яке містить стан вашого додатку. Стан представляє собою об'єкт, який визначає усі дані та стани, необхідні для роботи програми.
- Actions (Дії): Це об'єкти, які представляють зміни стану додатка. Дії містять інформацію про те, що трапилось, і вони є єдиною джерело інформації для зміни стану.
- Reducers (Редуктори): Це функції, які визначають, як стан додатка змінюється у відповідь на дії (actions). Кожен редуктор відповідає за

частину стану і визначає, як зміни в цій частині мають впливати на загальний стан.

Процес взаємодії в Redux виглядає наступним чином:

- Користувач взаємодіє з компонентами, що викликає дії.
- Дії подаються до Redux Store.
- Редуктори обробляють ці дії і оновлюють стан в сховищі.
- Зміни в стані відображаються на компонентах, які використовують цей стан, тригеруючи оновлення інтерфейсу користувача.

Цей підхід робить управління станом передбачуваним та простим для відстеження, що особливо важливо у великих додатках. Redux також дозволяє вам використовувати розширення, такі як часові маніпуляції, щоб відслідковувати та аналізувати зміни стану в часі.

Redux використовує сховище (Store) як свій центральний елемент. Сховище — це об’єкт, що вміщує глобальний стан вашого додатка, виконуючи ключову роль у зберіганні та забезпеченні доступу до необхідних даних. Використовуючи структуру даних, подібну до дерева, кожна частина стану має свій унікальний шлях, аналогічний шляху до файлу у файловій системі. Таким чином, взаємодія з даними в сховищі здійснюється за допомогою ключів, кожен з яких вказує на конкретну складову стану. Ця структура дозволяє дотримуватися ієрархічного підходу та легко керувати даними в контексті вашого додатка. Actions у Redux представляють собою об'єкти, які визначають події або сигнали в додатку. Вони служать для сповіщення про виникнення подій у вашому застосунку і використовуються для ініціювання змін у стані Redux. Коли щось відбувається у додатку, створюється action, який містить інформацію про виникнення події. Після цього цей action надсилається до редуктора (reducer), який обробляє його та вносить відповідні зміни у глобальний стан додатку. Actions грають ключову роль у системі Redux, де вони допомагають забезпечити передбачувані та контрольовані зміни в стані.

Reducers у Redux представляють собою функції, які реагують на дії (Actions) та визначають, як має змінюватися стан програми. Вони є важливою частиною механізму зміни стану в Redux. Редуктори відповідають за оновлення глобального стану додатку відповідно до отриманих дій. Кожен редуктор виконує логіку, специфічну для типу дії, і повертає новий стан. Через те, що редуктори є чистими та передбачуваними функціями, процес зміни стану стає надійним і легко зрозумілим. Редуктори грають ключову роль у впорядкуванні та управлінні станом застосунку, сприяючи його послідовному оновленню та дотриманню принципів Redux-архітектури.

Redux базується на кількох фундаментальних принципах, які покликані зробити управління станом додатка більш ефективним і передбачуваним. Розглянемо ці принципи:

- Передбачуваність стану програми: один з ключових принципів Redux полягає в тому, що стан застосунку є передбачуваним. Це означає, що в будь-який момент часу ви можете точно знати, як виглядає стан вашого застосунку і як він зміниться у відповідь на конкретні дії.
- Імутабельність і створення нових екземплярів стану: цей принцип передбачає, що стан програми не змінюється безпосередньо. Замість цього, кожна зміна стану призводить до створення нового екземпляра стану. Це дозволяє зберігати історію змін, спрощує відлагодження та робить код більш декларативним.

Ці принципи грають ключову роль у підтримці чистоти та порядку в управлінні станом застосунку, забезпечуючи його ефективну та передбачувану роботу.

Висновки

Процес проектування привів до вибору вищезазначених технологій для розробки програми. Саме вони є оптимальними для вирішення даної проблеми.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА ДЕМОНСТРАЦІЯ РОБОТИ ПЗ

4.1. Огляд системи

Веб-застосунок розроблено відповідно до сучасних технологій на базі .NET 6.0 та React.JS і використовується для автоматизації тестування. Він підтримує ручні та автоматичні тести для веб-сайтів і програм.

Ключовими властивостями є:

- Тестування доступне для різних версій Chrome, Safari, Firefox та IE, включаючи старіші версії.
- Відладка також реалізована за допомогою вбудованих засобів розробника. Це онлайн-інструмент крос-браузерного тестування.
- Тестування можна проводити як для локальних, так і для внутрішніх серверів: користувачі можуть перевіряти дизайн внутрішнього сервера або локального HTML у віддалених браузерах за допомогою безпечних локальних установок тестування.
- Автоматичні знімки екрана: генеруються автоматичні знімки екрана веб-сторінок у різних браузерах, що допомагає у візуальному порівнянні та регресійному тестуванні.

4.2. Демонстрація тестів

На наступних рисунках можна побачити інтерфейс програми, тести, що виконуються у ній, користувацькі групи тестів та статистику по тестах, що виконувалися повільніше інших або які не проходили найчастіше.

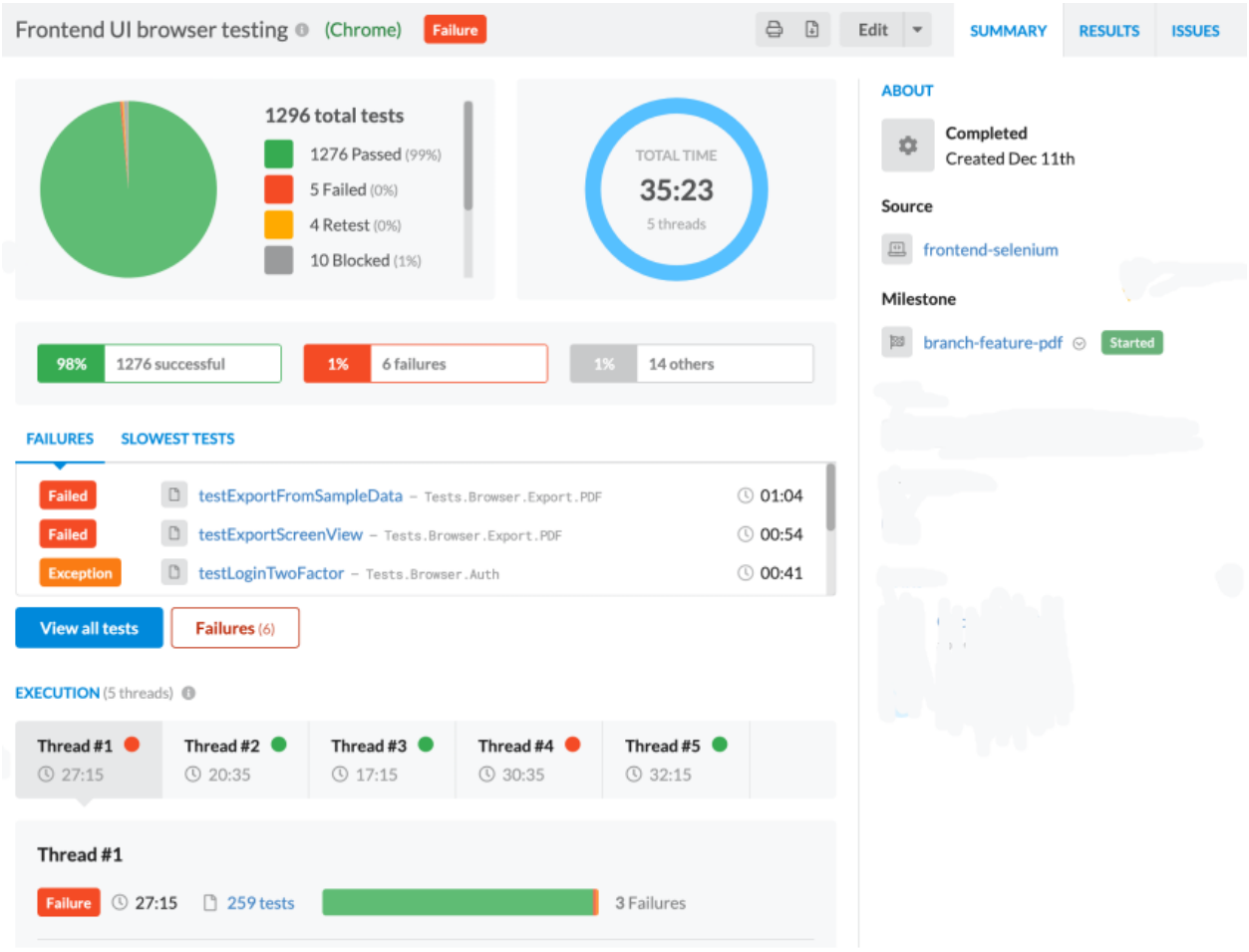

Рис. 4.1. Тести UI

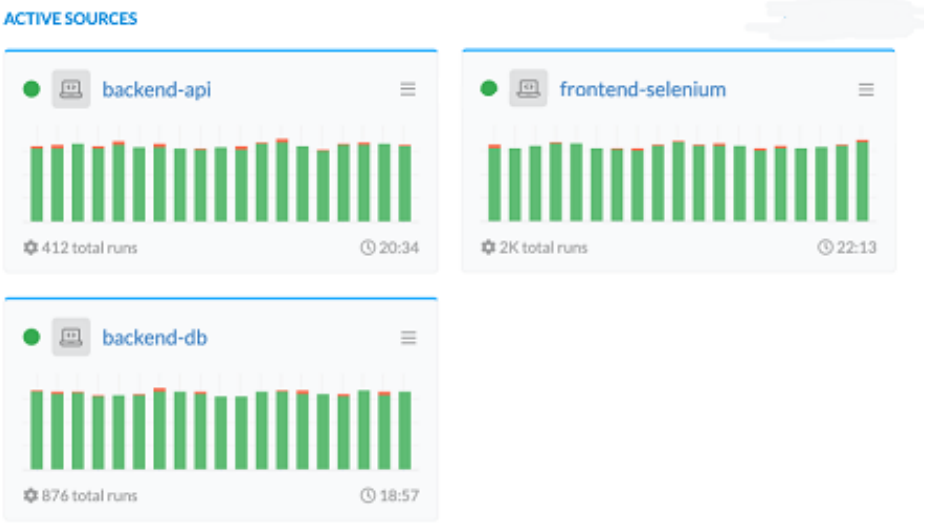

Рис. 4.2. Групування тестів

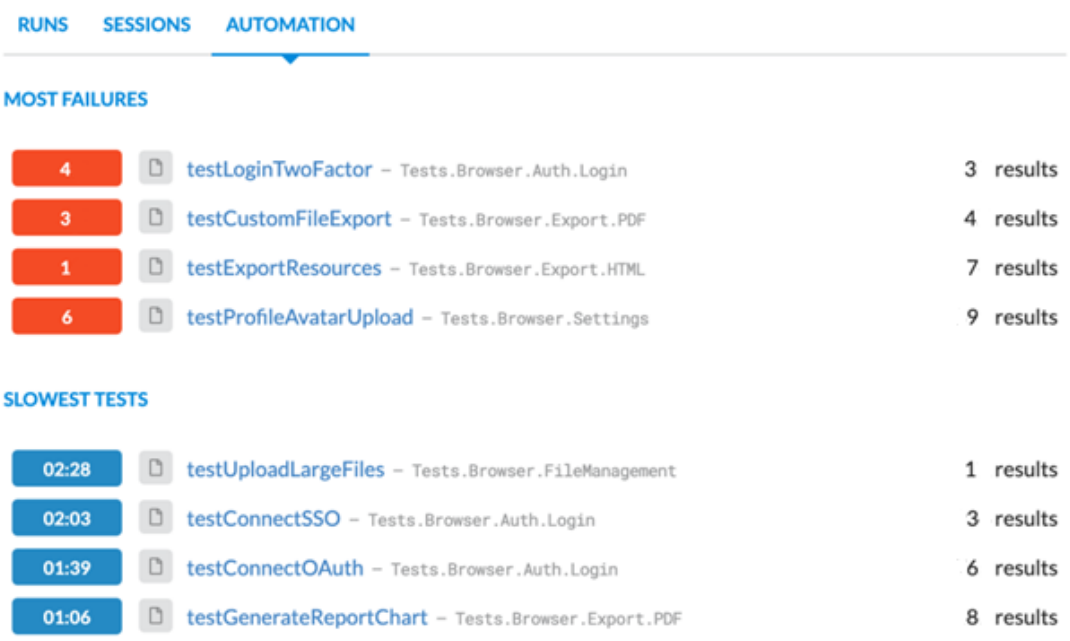

Рис. 4.3. Статистика тестів

Висновки

В цьому розділі було продемонстровано веб-додаток та перелічено його властивості.

Додаток може бути модифікований доданням можливості для ручних тестувальників створення автоматизованих тестів. Наприклад, запис дій тестувальника і їх конвертація у Javascript код, що утворить автоматизований тест.

ВИСНОВКИ

В ході дослідження, що висвітлене у даній дипломній роботі, були проаналізовані ключові аспекти тестування програмних систем з метою підвищення їх ефективності. Проведений порівняльний аналіз існуючих методик та технологій дозволив систематизувати та визначити оптимальні підходи для вдосконалення процесу тестування.

Отримані результати були чітко структуровані, що дозволило встановити конкретні вимоги до методики та застосунку для досягнення максимальної ефективності в тестуванні програмних систем. Обрана архітектура методики відповідає високим стандартам та гарантує надійність результатів.

Детально описана реалізація обраних технологій розкриває принципи їхньої роботи та показує їхню ефективність у практиці. Останній етап дослідження, що включає тестування та демонстрацію розробленої методики, свідчить про успішне впровадження в практику, а також розкриває можливості для подальших модифікацій та удосконалень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Тестування програмного забезпечення [Електронний ресурс] // foxminded.ua – Режим доступу до ресурсу: https://foxminded.ua/testuvannia-prohramnoho-zabezpechennia/
2. Стадії циклу розробки ПЗ [Електронний ресурс] // qalight.ua – Режим доступу до ресурсу: https://qalight.ua/baza-znaniy/stadiyi-tsiklu-rozrobki-pz/
3. Test types [Електронний ресурс] // istqb.org – Режим доступу до ресурсу: https://glossary.istqb.org/en_US/term/test-type-4-2
4. Аналіз вимог до програмного забезпечення [Електронний ресурс] / О. Л. Козак // Тернопільський Національний Економічний Університет. – 2011. – Режим доступу до ресурсу: http://dspace.wunu.edu.ua/retrieve/14135/FCIT_kKN_sPZS_dAVPZ_%20LEC.pdf
5. Формування якісної технічної документації до програмного забезпечення [Електронний ресурс] / О. В. Марковець, А. І. Синько // Національний Університет \"Львівська Політехніка\". – 2021. – Режим доступу до ресурсу: https://visnyk.vntu.edu.ua/index.php/visnyk/article/download/2613/2469
6. Огляд видів тестування [Електронний ресурс] // qatestlab.com – Режим доступу до ресурсу: https://training.qatestlab.com/blog/technical-articles/review-the-types-of-testing/
7. Що таке регресійне тестування? Визначення та основні інструменти [Електронний ресурс] // visuresolutions.com – Режим доступу до ресурсу: https://visuresolutions.com/uk/what-is-regression-testing-definition-and-top-tools/
8. Рівні тестування [Електронний ресурс] / qalearning.com.ua – Режим доступу до ресурсу: https://qalearning.com.ua/theory/lectures/material/testing-levels/

9. Модульне тестування [Електронний ресурс] / ЛНТУ // Луцький Національний Технічний Університет – Режим доступу до ресурсу: https://elib.lntu.edu.ua/sites/default/files/elib_upload/%D0%A2%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F/page14.html
10. Що таке інтеграційне тестування? Глибоке занурення в типи, процеси та впровадження [Електронний ресурс] // zaptest.com – Режим доступу до ресурсу: https://www.zaptest.com/uk/%D1%89%D0%BE-%D1%82%D0%B0%D0%BA%D0%B5-%D1%96%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%B5-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B3%D0%BB%D0%B8%D0%B1
11. Системне тестування програмного забезпечення [Електронний ресурс] // jak.koshachek.com – Режим доступу до ресурсу: https://jak.koshachek.com/articles/sistemne-testuvannja-programnogo-zabezpechennja.html
12. Що таке User Acceptance Testing?(приймальне тестування) [Електронний ресурс] // highload.today – Режим доступу до ресурсу: https://highload.today/uk/user-acceptance-testing-uat-prijmalne-testuvannya-ta-jogo-tsili/#_User_Acceptance_Testing
13. Види тестування програмного забезпечення [Електронний ресурс] // lemon.school – Режим доступу до ресурсу: https://lemon.school/blog/vydy-testuvannya-programnogo-zabezpechennya
14. Ручне тестування – що це таке, типи, процеси, підходи, інструменти та інше! [Електронний ресурс] // zaptest.com – Режим доступу до ресурсу: https://www.zaptest.com/uk/%D1%80%D1%83%D1%87%D0%BD%D0%B5-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%89%D0%BE-%D1%86%D0%B5-

%D1%82%D0%B0%D0%BA%D0%B5-%D1%82%D0%B8%D0%BF%D0%B8-%D0%BF%D1%80%D0%BE%D1%86

15. Що таке автоматизація тестування? Без жаргонів, простий посібник [Електронний ресурс] // zaptest.com – Режим доступу до ресурсу: https://www.zaptest.com/uk/%D1%89%D0%BE-%D1%82%D0%B0%D0%BA%D0%B5-%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D0%B7%D0%B0%D1%86%D1%96%D1%8F-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B1%D0%B5%D0%B7
16. Selenium WebDriver як інструмент для автоматизованого тестування [Електронний ресурс] // qatestlab.com – Режим доступу до ресурсу: https://training.qatestlab.com/blog/technical-articles/selenium-webdriver/
17. Appium Documentation [Електронний ресурс] // appium.io – Режим доступу до ресурсу: https://appium.io/docs/en/2.2/
18. Аналіз переваг та недоліків автоматизації тестування мобільних додатків. Аналіз інструмента Appium. [Електронний ресурс] / В. Ю. Коцюбинський, Я. В. Марущак, Ю. Д. Дрожнікова // Вінницький Національний Технічний Університет. – 2020. – Режим доступу до ресурсу: https://www.researchgate.net/profile/Yaroslava-Marushchak-2/publication/342165650_ANALIZ_PEREVAG_TA_NEDOLIKIV_AVTOMATIZACII_TESTUVANNA_MOBILNIH_DODATKIV_ANALIZ_INSTRUMENTA_APPIUM/links/5ee696c2458515814a5e8d15/ANALIZ-PEREVAG-TA-NEDOLIKIV-AVTOMATIZACII-TESTUVANNA-MOBILNIH-DODATKIV-ANALIZ-INSTRUMENTA-APPIUM.pdf
19. Шаблон проектування MVC [Електронний ресурс] // Ю. К. Поліщук // Уманський Державний Педагогічний Університет імені Павла Тичини – Режим доступу до ресурсу: https://informatika.udpu.edu.ua/?page_id=5450

20. Трирівнева архітектура [Електронний ресурс] // javarush.com – Режим доступу до ресурсу: https://javarush.com/ua/quests/lectures/ua.questservlets.level14.lecture01
21. Монолітна архітектура ПЗ [Електронний ресурс] // qalight.ua – Режим доступу до ресурсу: https://qalight.ua/baza-znaniy/shho-take-monolitna-arhitektura/
22. Інтерфейс користувача. Вимоги, складові частини. Класифікація. Вимоги до діалогу [Електронний ресурс] // um.co.ua – Режим доступу до ресурсу: http://um.co.ua/8/8-7/8-76154.html
23. PostgreSQL: Documentation [Електронний ресурс] // postgresql.org – Режим доступу до ресурсу: https://www.postgresql.org/docs/current/tutorial-install.html
24. .NET [Електронний ресурс] // dotnet.microsoft.com – Режим доступу до ресурсу: https://dotnet.microsoft.com/en-us/
25. What is .NET? Introduction and overview [Електронний ресурс] // learn.microsoft.com – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/core/introduction
26. ASP.NET documentation [Електронний ресурс] // learn.microsoft.com – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-8.0
27. ASP.NET Core for Beginners: Web APIs [Електронний ресурс] // telerik.com – Режим доступу до ресурсу: https://www.telerik.com/blogs/aspnet-core-beginners-web-apis
28. Entity Framework Core [Електронний ресурс] // entityframeworktutorial.net – Режим доступу до ресурсу: https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx
29. NUnit [Електронний ресурс] // nunit.org – Режим доступу до ресурсу: https://nunit.org/

30. HTML Підручник [Електронний ресурс] // w3schoolsua.github.io – Режим доступу до ресурсу: https://w3schoolsua.github.io/html/index.html#gsc.tab=0
31. Український веб-довідник [Електронний ресурс] // css.in.ua – Режим доступу до ресурсу: https://css.in.ua/
32. Tailwind CSS [Електронний ресурс] // tailwindcss.com – Режим доступу до ресурсу: https://tailwindcss.com/
33. Sass Introduction [Електронний ресурс] // w3schools.com – Режим доступу до ресурсу: https://www.w3schools.com/sass/sass_intro.php
34. Що таке React JS? Як почати вивчати Реакт? Навички для react developer [Електронний ресурс] // cases.media – Режим доступу до ресурсу: https://cases.media/article/sho-take-react-js-yak-pochati-vivchati-reakt-navichki-dlya-react-developer
35. Основи React Router [Електронний ресурс] // fpm.dnu.dp.ua – Режим доступу до ресурсу: http://fpm.dnu.dp.ua/2019/12/04/react-router/
36. Для чого і коли використовується Redux [Електронний ресурс] // foxminded.ua – Режим доступу до ресурсу: https://foxminded.ua/shcho-take-redux/