МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
Кафедра комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач кафедри

Аліна САВЧЕНКО

"___"_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ДИПЛОМНА РОБОТА, ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ

"МАГІСТРА"

ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
"ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ"

Тема: "Платіжна система на базі фреймворків Angular та Spring"

Виконав: Мнишенко Ігор Володимирович

Керівник: к. т. н., доцент кафедри КІТ Климова Асія Сабирівна

Нормоконтролер: Ігор РАЙЧЕВ

Київ – 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *Комп'ютерних наук та технологій*

Кафедра *Комп'ютерних інформаційних технологій*

Галузь знань, спеціальність, освітньо-професійна програма: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології"

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Аліна САВЧЕНКО
«____»_____2023р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи студента

Мнишенко Ігоря Володимировича

(прізвище, ім'я, по батькові)

1. **Тема роботи:** «Платіжна система на базі фреймворків Angular та Spring» Затверджена наказом ректора від 29.09.2023 р. за № 1976/ст.
2. **Термін виконання роботи:** з 02.10.2023р. до 31.12.2022р.
3. **Вихідні дані до проекту:** теоретичні відомості веб-розробки платіжних систем, мова Java, мова Javascript, backend фреймворк Spring, фронтенд фреймворк Angular.
4. **Зміст пояснювальної записки:** вступ, аналіз предметної області та огляд фреймворків Angular та Spring, проектування веб-додатку, розробка веб-сервісу.
5. **Перелік обов'язкового ілюстративного матеріалу:** інформативні рисунки, графічні скріншоти роботи системи, слайди презентації в MS PowerPoint.

6. **Календарний план-графік**

№ п/п	**Завдання**	**Термін виконання**	**Підпис керівника**
1	Дослідження та аналіз предметної області.	02.10.23 – 05.10.23	
2	Проведення консультацій з науковим керівником	06.10.23 – 10.10.23	
3	Написання та підготовка розділу 1	11.10.23 – 13.10.23	
4	Написання та підготовка розділу 2	14.10.23 – 15.10.23	
5	Написання та підготовка розділу 3	16.10.23 – 20.10.23	
6	Оформлення пояснювальної записки	21.10.23 – 31.10.23	
7	Підготовка презентації та доповіді	01.11.23 – 20.11.23	
8	Створення доповіді та презентації	21.11.23 – 28.11.23	
9	Підготовка до захисту дипломної роботи	29.11.23 – 20.12.23	

7. **Дата видачі завдання:** 02 жовтня 2023р.

Керівник дипломної роботи: _____Асія КЛИМОВА

(підпис керівника)

Завдання прийняв до виконання: _____Ігор МНИШЕНКО

(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи “Платіжна система на базі фреймворків Angular та Spring”: містить 69 сторінок, 9 рисунків, 14 наукових джерел.

Ключові слова: ПЛАТІЖНІ СИСТЕМИ, WEB – ЗАСТОСУНОК, БЕЗПЕКА, ТЕСТУВАННЯ

Мета кваліфікаційної роботи: створення застосунку платіжної системи на базі фреймворків Angular та Spring.

Об’єкт дослідження: процес розробки та впровадження платіжної системи для сфери фінансових послуг.

Предмет дослідження: використання фреймворків Angular та Spring для створення безпечної та ефективної платіжної системи, вдосконалення користувацького інтерфейсу та дослідження заходів для забезпечення безпеки, конфіденційності і відповідності стандартам у сфері фінансових послуг.

Метод дослідження: методи порівняльного аналізу методів реалізації, обробка літературних джерел.

Результат проекту: Розроблений застосунок з платіжної системи на базі фреймворків Angular та Spring.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

SPA – Single Page Application (односторінковий застосунок)

MPA – Multi Page Application (багатосторінковий застосунок)

PWA – Progressive Web Application (прогресивний веб-додаток)

HTML – Hyper Text Markup Language (мова розмітки гіпертексту)

API – Application programming interface (інтерфейс прикладного програмування)

JSON – JavaScript Object Notation (текстовий формат обміну даними)

IDE – Integrated Development Environment (інтегроване середовище розробки)

URL – Uniform Resource Locator (уніфікований покажчик інформаційного ресурсу)

HTTP – Hypertext Transfer Protocol (протокол передачі гіпертексту)

API – application programming interface (Прикладний програмний інтерфейс)

ВСТУП

В сучасному світі, де інформаційні технології набувають все більшого значення, розробка та вдосконалення веб-застосунків стали важливим завданням для розвитку цифрового середовища. Веб-застосунки, здатні забезпечити відмінний користувацький досвід та високий рівень зручності взаємодії з користувачем, стають драйверами технологічного прогресу і відкривають широкі можливості для інновацій та бізнес-розвитку.

У цьому дипломному проекті ми розглядаємо тему "Платіжна система на базі фреймворків Angular та Spring", що належить до актуальних напрямків розробки веб-застосунків. Спрямована на розробку зручного та безпечного способу здійснення платежів в онлайн-середовищі, ця тема ставить перед собою важливі завдання, пов'язані з ефективним використанням фреймворків Angular та Spring для створення високоякісних веб-застосунків.

Перед нами стоїть завдання дослідити не лише технічні аспекти створення платіжної системи на базі цих фреймворків, але і вивчити їхні можливості щодо забезпечення безпеки, швидкості та зручності користувачів. Ми також розглянемо архітектурні рішення, які дозволять інтегрувати платіжну систему з іншими компонентами веб-додатку та забезпечити їх взаємодію на високому рівні.

Результати цього дипломного дослідження мають важливе значення для інженерів-програмістів, розробників веб-застосунків та всіх, хто цікавиться розробкою інноваційних веб-продуктів. Ми прагнемо не лише розкрити технічні деталі реалізації платіжної системи, а й підкреслити її важливість у сфері онлайн-бізнесу та споживчої сфери. Робота над цим проектом дозволить нам вдосконалити свої навички веб-розробки та сприятиме подальшому розвитку цієї сфери в майбутньому.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ФРЕЙМВОРКІВ ANGULAR ТА SPRING

1.1. Аналіз існуючих платіжних систем

У сучасному цифровому світі платіжні системи є невід'ємною складовою економічного та фінансового життя. Вони дозволяють ефективно здійснювати електронні та безготівкові платежі, що сприяє зручності і надійності фінансових операцій. Платіжні системи є основою електронної комерції, онлайн-банкінгу, а також різних фінтех-рішень.

Цей розділ присвячений аналізу різних платіжних систем, які існують у сучасному світі. Ми розглянемо різні типи платіжних систем, їхні основні характеристики та переваги. Крім того, ми проаналізуємо ключові гравці на ринку платіжних систем та їхню роль у забезпеченні фінансової стабільності та зручності для користувачів.

1.1.1. Типи платіжних систем

Платіжні системи поділяються на кілька основних типів, зокрема:

Банківські платіжні системи. Банківські платіжні системи включають у себе роботу банків та фінансових установ щодо обробки грошових транзакцій. Вони забезпечують послуги з переказу грошей, видачі платіжних карток, інтернет-банкінгу та інших фінансових операцій.

Електронні платіжні системи. Електронні платіжні системи дозволяють користувачам проводити оплату товарів і послуг через Інтернет.

Серед найвідоміших представників цього типу систем можна виділити PayPal, Apple Pay, Google Pay та інші.

Мобільні платіжні системи. Мобільні платіжні системи дозволяють користувачам проводити транзакції за допомогою мобільних пристроїв, таких як смартфони і планшети. Найпопулярніші представники цього типу систем - Apple Pay, Samsung Pay, та інші.

1.1.2. Основні характеристики платіжних систем

Платіжні системи мають ряд важливих характеристик, які визначають їхню ефективність та придатність для різних цілей. До ключових характеристик можна віднести:

Швидкість обробки транзакцій. Швидкість обробки транзакцій визначається часом, необхідним для завершення платіжної операції. Ця характеристика особливо важлива для онлайн-платежів та торгівлі в реальному часі.

Безпека. Безпека платіжних систем має критичне значення для захисту фінансових даних та особистої інформації користувачів. Вона включає в себе заходи з шифрування, аутентифікації та моніторингу.

Доступність. Доступність платіжних систем полягає в їхній доступності для користувачів у різних регіонах та часи доби. Це особливо важливо в глобальному інтернет-середовищі.

1.1.3. Головні гравці на ринку платіжних систем

На ринку платіжних систем існує багато гравців, які надають різноманітні платіжні послуги. До найбільших та найвпливовіших гравців можна віднести:

Visa та MasterCard. Visa та MasterCard є двома найбільшими мережами платіжних карток у світі. Вони дозволяють користувачам здійснювати безготівкові оплати у мільйонах точок обслуговування по всьому світу.

PayPal. PayPal є однією з найпопулярніших електронних платіжних систем, яка дозволяє користувачам здійснювати платежі через Інтернет без необхідності вводити фінансові дані на кожному сайті.

Apple Pay та Google Pay. Apple Pay та Google Pay є представниками мобільних платіжних систем, які використовують технологію NFC для проведення оплати з мобільних пристроїв.

1.1.4. Переваги та недоліки платіжних систем

Платіжні системи мають свої переваги і недоліки, які варто враховувати при їх виборі та використанні.

Переваги:

- **Зручність і швидкість**: Платіжні системи дозволяють здійснювати транзакції без необхідності відвідувати банк або використовувати готівку. Це робить оплату товарів і послуг швидшою та зручнішою;
- **Безпека**: Багато платіжних систем використовують сучасні технології шифрування та аутентифікації для забезпечення безпеки транзакцій;
- **Глобальність**: Деякі платіжні системи працюють на міжнародному рівні, дозволяючи здійснювати оплату в будь-якій країні світу;
- **Інновації**: Платіжні системи постійно вдосконалюються та впроваджують нові технології, що полегшують користувачам споживати товари і послуги.

Недоліки:

- **Вартість**: Деякі платіжні системи вимагають від користувачів платити комісії або збори за використання. Це може зробити оплату дорожчою;

- **Залежність від інтернету**: Багато електронних і мобільних платіжних систем вимагають доступу до Інтернету, що може бути проблематичним у деяких регіонах;
- **Ризик безпеки**: Існує ризик кібератак та шахрайства в сфері платежів, який може поставити під загрозу фінансові дані користувачів.

1.1.5. Майбутні тренди у платіжних системах

Платіжні системи постійно еволюціонують, а майбутні тренди включають:

- **Розширення використання криптовалют**: Криптовалюти, такі як Bitcoin, набувають популярності у світі платежів;
- **Використання штучного інтелекту**: AI використовується для виявлення фроду та покращення безпеки платежів;
- **Зростання мобільних платежів**: Мобільні платіжні системи продовжують рости і ставати більш популярними.

Аналіз наявних платіжних систем показує, що вони є ключовими компонентами сучасного фінансового та економічного ландшафту. Платіжні системи дозволяють ефективно та безпечно здійснювати грошові транзакції, полегшуючи спосіб споживання товарів і послуг. Розуміння характеристик та функцій різних платіжних систем є важливим для подальшого аналізу та розробки власної платіжної системи на базі фреймворків Angular та Spring, що буде розглянуто в наступних розділах дипломної роботи.

1.2. Огляд можливостей фреймворку Angular

1.2.1. Що таке Angular

Angular - це потужний фреймворк для розробки веб-застосунків, створений компанією Google. Він базується на мові програмування TypeScript і надає інструменти для створення масштабних та динамічних односторінкових додатків (Single Page Applications - SPA) та багатосторінкових веб-застосунків. Angular дозволяє розробникам будувати високопродуктивні та користувачам зручні додатки, завдяки ряду важливих можливостей.

1.2.2. Основні можливості Angular

Angular працює на основі компонентної архітектури, що спрощує розробку та підтримку додатків. Веб-застосунок розбивається на невеликі незалежні компоненти, кожен з яких відповідає за свою частину інтерфейсу та функціоналу. Це сприяє полегшенню спільної роботи розробників та дозволяє швидше реагувати на зміни.

Angular підтримує реактивне програмування з використанням RxJS, що дозволяє ефективно працювати з асинхронними операціями, такими як HTTP-запити чи обробка подій. Це допомагає створювати додатки з високою продуктивністю та швидкодією, зменшуючи навантаження на сервер і покращуючи користувацький досвід.

Angular надає потужний механізм для маршрутизації, що дозволяє створювати навігаційну структуру веб-додатку та визначати, який компонент відображати для кожного URL. Це особливо важливо для створення SPA, де зміна URL може завантажити нову частину додатку без перезавантаження сторінки.

Angular підтримує ін'єкцію залежностей, що дозволяє розробникам створювати сервіси та інші компоненти, які можна використовувати у всьому додатку. Це

спрощує структуру коду, робить його більш модульним та підтримує його перевикористання.

1.2.3. Чому варто використовувати Angular у розробці веб-застосунків

Angular має декілька переваг, які роблять його відмінним вибором для розробки веб-застосунків:

- Велике та активне спільнота: Angular має велику спільноту розробників та активно оновлюється. Це означає, що ви можете легко знайти відповіді на свої запитання та рішення для проблем;
- Завдяки TypeScript, код більш безпечний та робустий: Використання TypeScript допомагає уникнути багатьох типових помилок під час розробки та покращує зрозумілість коду;
- Масштабованість: Angular розроблений з урахуванням потреб масштабних проектів, що дозволяє легко розширювати та підтримувати додатки зі зростанням їх складності;
- Широкий функціонал: Фреймворк надає велику кількість готових інструментів та бібліотек, що спрощують розробку та забезпечують можливість швидко створювати різноманітні функції та ефективний інтерфейс.

1.2.4. Використання Angular при розробці веб-застосунків

Angular ідеально підходить для розробки веб-застосунків будь-якої складності. Він може використовуватися для створення:

- Односторінкових додатків (SPA), які завантажуються один раз та динамічно оновлюють контент;

- Багатосторінкових веб-застосунків зі складними маршрутизаційними структурами;
- Передових додатків з реактивним інтерфейсом та асинхронною взаємодією.

Ми розглянули ключові аспекти фреймворку Angular, його можливості та переваги. Angular є потужним інструментом для створення високоякісних веб-застосунків, які можуть задовольняти навіть найвищі вимоги щодо продуктивності та користувацького досвіду. Використовуючи Angular, розробники мають можливість ефективно створювати сучасні веб-додатки та розвивати їх з часом.

1.3. Огляд можливостей фреймворку Spring

1.3.1. Огляд фреймворку Spring

Фреймворк Spring є одним з найбільш популярних і впливових фреймворків для розробки веб-застосунків та програмного забезпечення на мові програмування Java. Він був розроблений компанією Pivotal Software (раніше SpringSource) та став важливим інструментом для Java-розробників у всьому світі.

1.3.2. Основні можливості Spring

Spring надає розробникам численні можливості та інструменти для створення потужних веб-застосунків. Основні можливості фреймворку включають:

Інверсія управління та контейнер IoC

Spring використовує принцип інверсії управління (Inversion of Control - IoC), що дозволяє визначати, які об'єкти повинні створюватися та як вони повинні взаємодіяти, без прив'язки до конкретного класу. Це полегшує створення та управління об'єктами в додатку.

Ін'єкція залежностей

Spring сприяє ін’єкції залежностей (Dependency Injection - DI), що дозволяє об'єктам отримувати всі необхідні залежності ззовні, а не створювати їх самостійно. Це робить додаток більш модульним та покращує його підтримку та розширюваність.

Модульність та розширюваність

Spring дозволяє розробникам використовувати тільки необхідні модулі та розширювати функціональність фреймворку за потреби. Це полегшує налаштування та підтримку додатку.

Аспектно-орієнтоване програмування (AOP)

Spring підтримує аспектно-орієнтоване програмування (Aspect-Oriented Programming - AOP), що дозволяє розділити логіку програми на окремі аспекти, такі як журналювання, без втручання в основний код. Це спрощує управління певними аспектами додатку та підвищує його модульність.

Підтримка інтеграції з іншими технологіями

Spring надає розширення та інтеграцію з різними технологіями, такими як системи управління базами даних, фреймворки для веб-розробки, технології месседжінгу та інші. Це дозволяє розробникам ефективно взаємодіяти з іншими компонентами системи.

1.3.3. Використання Spring при розробці веб-застосунків

Spring є ідеальним інструментом для розробки різноманітних веб-застосунків. Розробники можуть використовувати Spring для створення та розробки веб-застосунків будь-якої складності, забезпечуючи підтримку для обробки HTTP-запитів, маршрутизації, обробки форм та багато іншого.

RESTful веб-сервісів. Spring дозволяє створювати RESTful веб-сервіси для забезпечення взаємодії з клієнтами та іншими додатками. Його можна

використовувати для створення API, які відповідають сучасним стандартам веб-розробки.

Систем мікросервісів. Spring добре підходить для розробки систем мікросервісів, де функціональність розділяється на окремі сервіси, що взаємодіють між собою. Фреймворк надає інструменти для створення, конфігурації та керування цими сервісами.

1.3.4. Переваги використання Spring

Spring має безліч переваг, які роблять його важливим інструментом для розробки веб-застосунків:

Велика та активна спільнота. Spring має велику та активну спільноту розробників, яка надає підтримку та розвиває фреймворк. Це означає, що ви можете легко знайти відповіді на свої запитання та рішення для проблем.

Висока модульність та розширюваність. Spring дозволяє використовувати тільки ті компоненти, які вам потрібні, і легко розширювати функціональність за потреби.

Інтеграція з іншими технологіями. Spring надає розширення та інтеграцію з різними технологіями, що спрощує взаємодію з іншими компонентами системи.

Висока продуктивність та безпека. Spring допомагає підвищити продуктивність та забезпечити безпеку додатків, надаючи ефективні засоби для роботи з даними та забезпечуючи захист від вразливостей.

Ми детально розглянули фреймворк Spring, його ключові можливості та переваги, які роблять його важливим інструментом для розробки веб-застосунків на мові програмування Java. Spring надає розробникам потужні інструменти та підтримку для створення високоякісних додатків, що задовольняють найвищі вимоги щодо продуктивності та надійності.

Висновки до розділу 1

У цьому розділі дипломної роботи був проведений аналіз предметної області, що стосується платіжних систем, а також проведено огляд двох ключових технологічних фреймворків - Angular та Spring. Аналіз платіжних систем надав глибокого розуміння їхнього функціоналу, основних характеристик, гравців на ринку та майбутніх трендів. Цей аналіз визначив актуальність дослідження та розробки платіжної системи.

Огляд фреймворку Angular розкрив основні можливості цієї технології, відзначено її переваги та приведено приклади використання. Цей огляд підкреслив потужний інструментарій Angular для розробки веб-застосунків та підкреслив важливість використання цього фреймворку в контексті розробки платіжних систем.

Огляд фреймворку Spring також розкрив його ключові можливості та переваги в контексті розробки веб-застосунків. Spring визначено як потужний фреймворк для розробки серверної частини платіжних систем, зокрема, для забезпечення безпеки та обробки транзакцій.

Загальною метою дослідження та розробки є створення безпечної та ефективної платіжної системи з використанням фреймворків Angular та Spring. Відповідно до результатів аналізу та огляду, обрані технології є відмінними засобами для досягнення цієї мети.

Основними висновками з цього розділу є підтвердження актуальності теми дипломної роботи, обґрунтування вибору фреймворків Angular та Spring, а також визначення шляхів подальшого дослідження та реалізації платіжної системи з використанням цих технологій.

РОЗДІЛ 2. ТЕОРЕТИЧНІ ОСНОВИ СТВОРЕННЯ ВЕБ-ДОДАТКУ ПЛАТІЖНОЇ СИСТЕМИ

2.1. Принципи розробки веб-застосунків з використанням Angular

Angular пропонує низку ключових принципів, які слугують основою ефективної розробки веб-застосунків. Один з них – це уніфікована компонентна архітектура, яка дозволяє поділити додаток на самодостатні компоненти. Це полегшує підтримку та дозволяє розробникам працювати над окремими частинами без впливу на інші.

Інверсія управління в Angular забезпечує, що фреймворк відповідає за управління об'єктами та контролює їх життєвий цикл. Це дозволяє знизити залежності та робить код меншим та більш зрозумілим.

Компонентний підхід у веб-розробці використовує розбиття додатку на компоненти. Кожен компонент відповідає за конкретну частину інтерфейсу та функціоналу, забезпечуючи таким чином модульність та легкість управління.

Для полегшення розробки та роботи з проектом, Angular рекомендує використання Angular CLI. Цей інструментарій автоматизує багато завдань, таких як створення компонентів, модулів, а також забезпечує зручний інтерфейс для керування проектом.

Загалом, принципи розробки веб-застосунків з використанням Angular наголошують на модульності, чіткій структурі та ефективній взаємодії між компонентами, що сприяє швидкій та зручній розробці високоякісних додатків.

2.1.1. Архітектурні концепції Angular та їх вплив на розробку

Angular визначає унікальні архітектурні концепції, які додають структурну красу та ефективність до розробки веб-застосунків.

Метакомпонентність: В основі Angular лежить концепція метакомпонентності, де кожен компонент може виступати як шаблон для створення нових компонентів. Це сприяє високому рівню перевикористання та дозволяє розширювати функціонал з легкістю.

Обслуговування подій та станів: Angular пропонує потужні механізми обробки подій та відстеження стану додатка. Забезпечуючи реактивний підхід до програмування, ці концепції дозволяють розробникам створювати відгукні та ефективні додатки.

Декларативна конфігурація: Angular використовує декларативний підхід до конфігурації компонентів та їхньої поведінки. Це полегшує читання та розуміння коду, а також зменшує ймовірність помилок у процесі розробки.

Ін'єкція залежностей на рівні компонентів: Angular дозволяє ін'єкцію залежностей на рівні компонентів, що сприяє високому рівню міжкомпонентної взаємодії та дозволяє ефективно обмінюватись даними та функціоналом.

Розширюваність та масштабованість: Архітектурні рішення Angular розроблені з урахуванням потреб масштабованих проектів. Це дозволяє створювати додатки, які легко розширюються та підтримуються зі зростанням їхньої складності.

Архітектурні концепції Angular формують технічний фундамент для створення динамічних та ефективних веб-застосунків, спрощуючи розробку та підтримку кодові бази.

2.1.2. Компонентний підхід у веб-розробці з Angular

Компонентний підхід є ключовою характеристикою веб-розробки з використанням Angular, принесенням значного поліпшення до структури та обслуговування коду.

Суть компонентів: У світі Angular кожен елемент інтерфейсу представлений як компонент — самостійний та незалежний будівельний блок. Це дозволяє використовувати компоненти як метакомпоненти, щоб легко розширювати та перевикористовувати їх в різних частинах додатка.

Ізольованість та реюзабельність: Кожен компонент має власний контекст виконання та ізольований від інших. Це підвищує стійкість системи та дозволяє ефективно управляти станом. Компонентний підхід також дозволяє легко використовувати компоненти в різних частинах додатка або навіть у різних проектах.

Спрощене тестування: Ізольованість компонентів полегшує процес тестування. Розробники можуть тестувати кожен компонент незалежно, переконуючись в його коректній роботі перед інтеграцією в загальний код.

Єдина відповідальність: Компоненти мають чітко визначену відповідальність, що спрощує розподіл обов'язків та сприяє більшій чіткості в структурі додатка. Кожен компонент відповідає за свою частину інтерфейсу та поведінки.

Забезпечення взаємодії: Компоненти взаємодіють через визначені інтерфейси та обмінюються даними. Це дозволяє створювати розширювані та модульні додатки, які легко підтримувати та розвивати.

Компонентний підхід Angular привносить в розробку веб-додатків модульність, ізольованість та ефективність, сприяючи створенню високоякісних та сучасних інтерфейсів.

2.1.3. Використання Angular CLI для полегшення розробки проекту

Angular CLI (Command Line Interface) є невід'ємною складовою розробки веб-застосунків з використанням Angular, надаючи розробникам потужний інструментарій для ефективного управління та підтримки проектів.

Автоматизація створення проектів: Angular CLI пропонує команду для швидкого створення нового проекту з використанням передових налаштувань та структури каталогів. Це спрощує початковий процес ініціалізації проекту та надає готову базу для подальшого розвитку. Генерація компонентів та модулів: CLI дозволяє створювати нові компоненти, модулі та інші будівельні блоки за допомогою простих команд. Це значно прискорює розробку та виключає ручне створення файлів, дозволяючи розробникам швидко експериментувати зі структурою свого додатка. Автоматична конфігурація Webpack та інших інструментів: Angular CLI автоматично налаштовує інструменти, такі як Webpack, для оптимізації та збільшення продуктивності додатка. Це дозволяє розробникам уникнути складних конфігурацій та фокусуватися на самому коді. Вбудована система тестування: CLI надає інтегровану систему тестування для автоматизації та виконання юніт-тестів. Це допомагає забезпечити стабільність та надійність коду в процесі розробки. Легкість використання командного рядка: Angular CLI робить роботу з проектом простою завдяки інтуїтивно зрозумілому та легкому використанні інтерфейсу командного рядка. Це полегшує комунікацію та взаємодію розробників з проектом. Оновлення та розширення: CLI надає зручний механізм для оновлення Angular та його залежностей, а також для розширення функціоналу за допомогою плагінів. Це дозволяє тримати проекти в актуальному стані та використовувати нові можливості фреймворку.

Angular CLI відкриває перед розробниками можливості швидкого та ефективного впровадження проектів, роблячи процес розробки зручним та продуктивним.

2.2. Основні аспекти розробки на платформі Spring

Платформа Spring — це потужний інструментарій для розробки сучасних, надійних та масштабованих Java-застосунків. У цьому розділі розглянемо ключові аспекти розробки на платформі Spring, які допомагають розробникам забезпечити ефективність, надійність та гнучкість своїх додатків.

Контейнер IoC (Inversion of Control): Одним з основних принципів Spring є інверсія управління, що дозволяє контейнеру Spring керувати життєвим циклом та залежностями об'єктів. Це полегшує конфігурацію та підтримку додатків.

Аспектно-орієнтоване програмування (AOP): Spring надає можливості для впровадження аспектів, що дозволяє винести загальні аспекти, такі як логування чи безпека, в окремі модулі. Це спрощує обслуговування та покращує читабельність коду.

Керування транзакціями: Spring забезпечує механізми керування транзакціями, що робить його ідеальним для розробки додатків з високим рівнем надійності та цілісності даних.

Модель програмування для роботи з базами даних: Spring Framework має вбудовану підтримку для роботи з реляційними базами даних, надаючи високорівневий та об'єктно-реляційний підхід до взаємодії з даними.

Spring Boot для швидкого старту проектів: Spring Boot дозволяє швидко створювати самостійні, автономні застосунки з мінімальними конфігураціями. Це

Підтримка тестування: Spring надає зручні інструменти для юніт-тестування та інтеграційного тестування, що дозволяє розробникам створювати надійний та стабільний код.

Використання платформи Spring в розробці додатків дозволяє розробникам швидко та ефективно створювати високоякісні застосунки, забезпечуючи їх гнучкість та легкість управління.

2.2.1. Огляд основних концепцій та архітектурних принципів Spring

У цьому розділі ми розглянемо ключові концепції та архітектурні принципи, які формують основу платформи Spring і грають важливу роль у розробці надійних та ефективних додатків.

Спрощена конфігурація з Spring Boot:

Spring Boot пропонує анотаційну конфігурацію, що дозволяє розробникам зосередитися на функціональності додатку, а не на витраті часу на складні конфігураційні файли. Автоматична конфігурація спрощує створення та підтримку проектів.

Інтеграція з іншими фреймворками та технологіями:

Spring легко інтегрується з іншими технологіями, надаючи розробникам вибір використання різноманітних інструментів. Це включає інтеграцію з Apache Kafka, RabbitMQ, та іншими рішеннями для обробки повідомлень.

Аспекти безпеки в Spring Security:

Spring Security — це модуль, що додає високорівневий захист для додатків, включаючи аутентифікацію, авторизацію та захист від атак. Розширена система безпеки дозволяє розробникам забезпечувати найвищий рівень безпеки для своїх додатків.

Модульність з Spring Modules:

Spring дозволяє розробникам будувати модульні додатки, розділяючи їх на логічні блоки. Це полегшує управління та розширення функціоналу додатку з часом.

Використання Spring Data для роботи з базами даних:

Spring Data спрощує взаємодію з базами даних, надаючи анотаційний підхід та автоматичне створення репозиторіїв. Це полегшує інтеграцію додатків з різними системами управління базами даних.

Міжнародна підтримка в Spring:

Spring надає підтримку для міжнародизації, дозволяючи розробникам створювати додатки, які можуть працювати в різних мовах та регіонах без великих труднощів.

Ці концепції та принципи допомагають розробникам створювати гнучкі, надійні та модульні додатки з використанням платформи Spring.

2.2.2. Використання Spring Boot для швидкого старту розробки

Spring Boot є високорівневим рішенням, спрямованим на полегшення розробки додатків на основі платформи Spring. Його головною метою є надання простого та швидкого способу створення робочих прототипів, мікросервісів та інших аспектів сучасного програмного об'єкту.

Автоматична конфігурація:

Spring Boot вражає своєю здатністю автоматично конфігурувати додаток. Замість ручного визначення кожної деталі, розробник може скористатися анотаціями та стандартними конвенціями для швидкого налаштування.

Вбудований контейнер:

Однією з ключових переваг Spring Boot є вбудований веб-сервер, що дозволяє запускати додаток без необхідності конфігурування зовнішнього веб-сервера. Це значно спрощує процес розгортання та тестування.

Автоматичне управління залежностями:

Spring Boot використовує систему управління залежностями Maven або Gradle, автоматично розв'язуючи проблеми залежностей для вас. Це дозволяє ефективно управляти бібліотеками та їх версіями без зайвого головоболю.

Велика кількість стартових шаблонів:

Spring Boot пропонує велику кількість стартових шаблонів (starter templates), які містять базовий код та конфігурацію для швидкого старту. Це дозволяє розробникам вибирати необхідні компоненти для свого проекту.

Моніторинг і управління:

Завдяки актуаторам, Spring Boot забезпечує вбудовані засоби моніторингу та управління додатком. Розробники можуть легко отримувати дані про стан додатку та виконувати ряд управлінських дій.

Висока ступінь гнучкості:

Spring Boot дозволяє використовувати або налаштовувати свої засоби, щоб розробники могли забезпечити необхідну гнучкість та контроль над додатком.

Використання Spring Boot полегшує та прискорює процес розробки, дозволяючи фокусуватися на функціональності та вдосконаленні додатків, замість витрати часу на налаштування та конфігурацію.

2.2.3. Інтеграція Spring з базами даних та іншими сервісами

Інтеграція Spring з базами даних та іншими сервісами є важливим етапом у розробці додатків, оскільки дозволяє забезпечити доступ до даних та інших ресурсів. Spring Framework пропонує ефективні механізми для здійснення цієї інтеграції.

Spring Data дозволяє легко інтегрувати додатки з різними системами управління базами даних, такими як MySQL, PostgreSQL, або MongoDB. Використовуючи анотації та конфігураційні файли, розробники можуть здійснювати операції читання, запису та оновлення даних без зайвого коду.

Spring дозволяє легко взаємодіяти з іншими мікросервісами чи зовнішніми API. Використовуючи бібліотеку Spring Cloud, можна реалізувати механізми обслуговування, такі як обнаруження, балансування навантаження та контроль доступу.

Spring Framework забезпечує декларативний підхід до управління транзакціями. Це дозволяє визначати межі транзакцій, використовуючи анотації, і гнучко керувати їхнім життєвим циклом.

Spring Integration дозволяє легко інтегрувати різноманітні системи та платформи, створюючи підтримку для патернів обміну повідомленнями та обробки подій.

Spring Security дозволяє ефективно забезпечувати безпеку додатків при інтеграції з різними джерелами даних та зовнішніми сервісами.

Spring Actuator забезпечує можливості моніторингу та журналювання, що спрощує відстеження взаємодії додатка з базами даних та іншими сервісами.

Інтеграція Spring із базами даних та іншими сервісами надає розробникам ефективні інструменти для розширення функціональності своїх додатків та забезпечення взаємодії з різними джерелами даних.

2.3. Засоби забезпечення якості коду та тестування в Angular та Spring

Забезпечення якості коду та відповідність функціональності вимогам є ключовим аспектом розробки веб-застосунків на платформах Angular та Spring. Обидва фреймворки надають розширені інструменти для забезпечення високої якості коду та ефективного тестування.

Тестування в Angular:

Angular сприяє розробці тестованих додатків за допомогою платформи Jasmine. Розробники можуть писати юніт-тести, інтеграційні тести та e2e-тести, щоб переконатися в працездатності кожного компонента та його взаємодії. Angular Testing Module дозволяє створювати імітації та ефективно взаємодіяти з залежностями під час тестування.

Інструменти забезпечення якості в Angular:

Angular CLI надає вбудовані інструменти для перевірки якості коду. Наприклад, команда ng lint використовує TSLint для аналізу та перевірки стилю коду, що допомагає уникати потенційних помилок та забезпечує відповідність стандартам коду.

Тестування в Spring:

Spring Framework підтримує різноманітні види тестів, включаючи юніт-тести, інтеграційні тести та тести з використанням фреймворка Spring Boot. Анотації, такі як @RunWith(SpringRunner.class) та @SpringBootTest, полегшують конфігурацію та виконання тестів.

Інструменти забезпечення якості в Spring:

Spring підтримує використання інструментів, таких як FindBugs, Checkstyle, та PMD, для аналізу коду на предмет можливих помилок та відповідності стандартам. Використання цих інструментів допомагає уникнути деяких типових проблем та підвищити якість коду.

Інтеграція з системами CI/CD:

Як Angular, так і Spring можуть інтегруватися з системами Continuous Integration (CI) та Continuous Deployment (CD), що дозволяє автоматизувати процеси тестування та впровадження змін. Jenkins, Travis CI та GitLab CI є популярними засобами для цього.

Моніторинг та відстеження:

Інтеграція моніторингу та відстеження дозволяє вчасно виявляти проблеми. Spring Actuator та Angular Performance Explorer - це приклади інструментів, які забезпечують метрики та інформацію про продуктивність.

Загальна філософія тестування та засоби забезпечення якості в Angular та Spring роблять розробку додатків ефективною, а код надійним та високоякісним.

2.3.1. Юніт-тестування та інтераційне тестування в Angular

Юніт-тестування та інтеграційне тестування в Angular є ключовим етапом розробки для забезпечення високої якості коду та функціональності веб-застосунків.

Angular використовує фреймворк тестування Jasmine для написання юніт-тестів. Розробники можуть створювати тести для окремих компонентів, сервісів та

модулів, щоб переконатися в їхній правильній роботі. Застосунок Angular Testing Module дозволяє легко імітувати залежності та проводити ізольовані тести.

Інтеграційне тестування в Angular спрямоване на перевірку взаємодії між компонентами та модулями. З використанням Karma та Protractor можна автоматизувати тестування на різних рівнях. Karma використовується для виконання юніт-тестів, а Protractor - для e2e-тестування.

Angular CLI надає зручний інтерфейс для використання різноманітних засобів тестування. Такі інструменти, як Karma, Protractor, і Jest, інтегровані в CLI, що полегшує конфігурацію та запуск тестів.

Angular дозволяє використовувати імітацію та mock-об'єкти для ізольованого тестування. Це забезпечує можливість перевірки компонентів без реального з'єднання з сервером або іншими частинами додатку.

Завдяки HttpClientTestingModule та HttpTestingController, розробники можуть легко взаємодіяти з HTTP запитами під час юніт-тестування та забезпечити стабільність та передбачуваність результатів.

Angular надає засоби для автоматизації тестування через CI/CD системи, такі як Jenkins або GitLab CI. Це дозволяє автоматично виконувати тести при зміні коду та забезпечує раннє виявлення помилок.

Інтеграція юніт-тестування та інтеграційного тестування в розробку Angular гарантує надійність та стабільність веб-застосунків, а також допомагає розробникам ефективно виявляти та виправляти помилки.

2.3.2. Використання JUnit та інших інструментів для тестування Spring

В розробці на платформі Spring використання ефективних інструментів для тестування є важливою частиною процесу, спрямованою на забезпечення якості коду та функціональності додатків.

JUnit - це основний інструмент для юніт-тестування у світі Java та розширеної екосистеми Spring. Розробники можуть створювати тести для окремих методів, класів чи навіть цілих модулів, перевіряючи їхню коректність та взаємодію.

Spring Framework надає спеціальний модуль Spring Test, який допомагає використовувати анотації, такі як @RunWith(SpringRunner.class) та @SpringBootTest, для автоматизації конфігурації тестового середовища та завантаження контексту Spring. Для ізольованого тестування та мокування залежностей Spring використовує бібліотеку Mockito. З її допомогою розробники можуть створювати імітації об'єктів та контролювати їхню поведінку під час тестування.

Spring Boot дозволяє легко налаштовувати інтеграційні тести за допомогою анотацій, таких як @DataJpaTest для роботи з базою даних або @SpringBootTest для тестування всього додатка. Це спрощує виявлення проблем із взаємодією компонентів.

Використання test slices дозволяє створювати контекст тестування з обмеженим набором компонентів, що полегшує проведення швидких ізольованих тестів певних шарів додатка.

Додатково, Spring інтегрується з іншими інструментами для тестування, такими як TestNG, AssertJ, та Hamcrest, що надає розробникам більше можливостей у створенні деталізованих та ефективних тестів.

Використання JUnit та інших інструментів тестування у поєднанні з можливостями Spring допомагає розробникам створювати стабільні, надійні та ефективні додатки, забезпечуючи високий рівень тестового покриття та якість коду.

2.4. Принципи безпеки в розробці веб-застосунків

Забезпечення безпеки є критичним аспектом розробки веб-застосунків, оскільки вони піддаються різноманітним загрозам та атакам. Дотримання ключових

принципів безпеки є важливим завданням для захисту конфіденційності, цілісності та доступності даних.

Захист від атак на основі введення:

Всі дані, які надходять від користувачів, повинні бути оброблені та перевірені на належність. Використання механізмів фільтрації та валідації введених даних, таких як input validation та output encoding, допомагає запобігти атакам на основі введення (Injection Attacks).

Керування ідентифікацією та автентифікацією:

Важливо використовувати надійні механізми ідентифікації користувачів та перевірки їх автентичності. Використання сильних паролів, двоетапної аутентифікації та механізмів управління сесіями допомагає захищати доступ до системи.

Захист від міжсайтового скриптування (XSS):

Для уникнення атак XSS важливо коректно обробляти та екранувати дані, які виводяться на веб-сторінці. Використання безпечних API та фреймворків, а також регулярних оновлень для попередження нових уразливостей, допомагає у забезпеченні безпеки.

Захист від міжсайтового запросу (CSRF):

Для запобігання атакам CSRF важливо використовувати захисні механізми, такі як токени CSRF, які перевіряють автентичність та дозволяють визначати, чи є запит відповідним.

Шифрування та захист транспортного рівня:

Використання шифрування для захисту конфіденційності даних, особливо при передачі їх через мережу, є необхідним. Протокол HTTPS, який використовує SSL або TLS, гарантує шифрування та цілісність даних між клієнтом та сервером.

Регулярні аудити та моніторинг:

Проведення регулярних аудитів безпеки та встановлення систем моніторингу дозволяє вчасно виявляти та реагувати на можливі загрози та атаки. Це важливий аспект для підтримання безпеки веб-застосунків в актуальному стані.

Загальний підхід до безпеки, включаючи вищезазначені принципи, допомагає забезпечити надійність та захищеність веб-застосунків від сучасних загроз та атак.

2.4.1. Засоби забезпечення безпеки на рівні Angular

Angular надає ряд важливих засобів для ефективного забезпечення безпеки веб-застосунків. Ці засоби враховують високі стандарти безпеки та допомагають у виявленні та запобіганні потенційним загрозам.

Angular використовує вбудовані механізми для захисту від атак XSS (Cross-Site Scripting) та інших загроз безпеки. Всі дані, які виводяться на веб-сторінці, автоматично екрануються, що унеможливлює виконання шкідливого коду на клієнтському боці. Angular підтримує строгу політику безпеки контенту (CSP), яка дозволяє визначити, звідки може завантажуватись контент. Це зменшує ризик атак XSS, обмежуючи джерела, з яких може виконуватись скриптовий код. Angular використовує об'єктну модель безпеки, де доступ до об'єктів та їхній зміні контролюється. Це дозволяє уникнути використання несанкціонованих операцій та забезпечує внутрішню безпеку додатків. Angular забезпечує можливість встановлення заголовків безпеки для HTTP запитів, таких як CORS (Cross-Origin Resource Sharing). Це дозволяє обмежити доступ до ресурсів лише для певних джерел, покращуючи безпеку додатка. Активна спільнота розробників Angular та організація Google регулярно випускають оновлення та патчі безпеки. Розробники повинні слідкувати за новими версіями фреймворку та вчасно впроваджувати їх для захисту від відомих уразливостей.

Використання цих засобів на рівні Angular є важливою складовою безпеки веб-застосунків, забезпечуючи надійну захист від різноманітних загроз та атак.

2.4.2. Механізми безпеки Spring для захисту серверної частини

Spring Framework надає потужні механізми для забезпечення безпеки серверної частини веб-застосунків. Ці механізми враховують різноманітні аспекти безпеки та дозволяють розробникам ефективно захищати свої додатки.

Spring Security. Spring Security є ключовим модулем для забезпечення аутентифікації та авторизації на рівні серверної частини. Він дозволяє визначати права доступу до ресурсів та обмежувати їх використання тільки авторизованими користувачами. Конфігурація Spring Security забезпечує гнучкість та можливість інтеграції з різними системами аутентифікації.

Захист від атак. Spring Framework включає заходи безпеки, спрямовані на запобігання атакам, таким як SQL-ін'єкції, переповнення буфера та інші типи атак. Використання параметризованих запитів, валідація введених даних та інші техніки допомагають у зменшенні ризиків безпеки.

Вбудовані механізми шифрування. Spring пропонує вбудовані інструменти для шифрування конфіденційної інформації, такої як паролі користувачів. Використання стійких алгоритмів шифрування допомагає у збереженні конфіденційності та інтегрітету даних.

Сертифікати та HTTPS. Spring дозволяє налаштовувати та використовувати сертифікати для забезпечення безпеки передачі даних між клієнтом та сервером за допомогою протоколу HTTPS. Це є важливим аспектом у сучасних веб-додатках для захисту конфіденційної інформації.

Моніторинг та журналювання. В Spring можна використовувати інструменти для моніторингу безпеки, включаючи журналювання подій та аудит безпеки. Це дозволяє вчасно виявляти та реагувати на потенційні загрози.

Використання цих механізмів у Spring дозволяє розробникам створювати безпечні та надійні серверні компоненти для веб-застосунків.

2.5. Кращі практики розробки платіжних систем

Розробка платіжних систем вимагає особливої уваги до безпеки, ефективності та надійності. Нижче представлені кращі практики, які допомагають створити ефективні та безпечні платіжні рішення.

Забезпечення відповідності стандартам безпеки, таким як PCI DSS (Payment Card Industry Data Security Standard), є обов'язковим етапом розробки платіжної системи. Ці стандарти встановлюють вимоги до обробки, зберігання та передачі конфіденційної платіжної інформації. Платіжні системи повинні надійно захищати особисті дані користувачів. Застосування сучасних методів шифрування та анонімізації гарантує конфіденційність особистих інформаційних даних. Для підвищення безпеки, системи повинні використовувати двофакторну аутентифікацію. Це може включати в себе SMS-підтвердження, використання аутентифікаційних додатків або біометричні методи. Системи моніторингу дозволяють вчасно виявляти незвичайну активність або аномалії в платіжних операціях. Застосування алгоритмів машинного навчання допомагає автоматично розпізнавати підозрілі взаємодії. Для забезпечення ефективної роботи платіжної системи важливо оптимізувати її продуктивність. Використання кешування, асинхронних операцій та оптимізація баз даних дозволяють забезпечити швидку обробку платежів. Проведення регулярних аудитів безпеки допомагає виявляти та усувати можливі порушення безпеки. Аудити включають перевірку відповідності стандартам безпеки та аналіз журналів подій.

Врахування цих кращих практик в розробці платіжних систем допомагає забезпечити їхню безпеку, ефективність та надійність, що є критично важливим для успішного функціонування в умовах сучасного фінансового середовища.

2.5.1. Оптимізація продуктивності веб-застосунків

Оптимізація продуктивності веб-застосунків визначається низкою ключових факторів, що включають швидкодію, відзивчивість та ефективність використання ресурсів. Для досягнення оптимальної продуктивності використання Angular вирізняється певними підходами та інструментами.

Angular дозволяє використовувати асинхронне завантаження ресурсів за допомогою підгружених модулів. Це полегшує завантаження лише необхідних елементів, зменшуючи час завантаження сторінки.

Використання кешу для зберігання певних ресурсів, таких як зображення, стилі та скрипти, дозволяє прискорити повторні завантаження сторінок для користувачів.

Angular пропонує ефективні механізми для роботи з DOM, зокрема, використання Shadow DOM для ізоляції компонентів. Це сприяє уникненню зайвого перерендерингу та покращує швидкість відображення змін.

Angular підтримує концепцію лінивої загрузки модулів, що дозволяє завантажувати компоненти лише при їхньому реальному виклику. Це покращує ефективність завантаження додатка.

Використання Content Delivery Network (CDN) для розповсюдження ресурсів сприяє їхньому швидкому та надійному доставленню до кінцевих користувачів.

Підтримка Angular для HTTP-запитів дозволяє ефективно взаємодіяти з сервером, зменшуючи кількість та об'єм передаваних даних, що позитивно впливає на продуктивність.

Загальною метою оптимізації продуктивності веб-застосунків за допомогою Angular є створення швидкодіючих та ефективних додатків, які забезпечують задоволення користувачів та мінімізують час завантаження сторінок.

2.5.2. Управління залежностями та версіонуванням

Ефективне управління залежностями та версіонуванням є ключовим аспектом розробки веб-застосунків як на платформі Angular, так і в середовищі Spring. Використання правильних інструментів та підходів грає важливу роль у забезпеченні стабільності, безпеки та актуальності програмного продукту.

Angular використовує систему керування пакетами npm (Node Package Manager) для управління залежностями. Розробники можуть визначити необхідні бібліотеки та їх версії в файлі package.json. Використання npm дозволяє зручно встановлювати, оновлювати та видаляти залежності. Angular використовує стратегію Semantic Versioning (SemVer) для версіонування. Кожен номер версії складається з трьох частин: major.minor.patch. Зміна major вказує на несумісні зміни API, minor означає додані функціональні можливості, а patch вказує на виправлення помилок. У світі Spring, інструмент Maven часто використовується для управління залежностями та версіонуванням. Інформація про залежності та їх версії визначається в файлі pom.xml. Maven автоматично завантажує необхідні бібліотеки з центрального репозиторію. Аналогічно Angular, в Spring також використовується стратегія Semantic Versioning. Кожен номер версії складається з трьох частин: major.minor.patch. Це полегшує визначення суттєвості змін та їх впливу на розробку.

Ефективне управління залежностями та версіонуванням забезпечує проект стабільністю, гнучкістю та легкістю супроводу. Застосування зазначених підходів дозволяє розробникам ефективно працювати над проектами, забезпечуючи їхню актуальність та високу якість.

2.6. Розробка масштабованих та ефективних систем

Розробка масштабованих та ефективних систем є важливою задачею для забезпечення високої продуктивності та задоволення потреб сучасних користувачів.

У контексті веб-застосунків на платформі Angular та в середовищі Spring існують ключові підходи та стратегії, спрямовані на забезпечення масштабованості та ефективності.

Angular дозволяє розробникам будувати масштабовані веб-застосунки завдяки своїй компонентній архітектурі. Розділення функціональності на невеликі та незалежні компоненти сприяє легкості масштабування. Крім того, використання асинхронного програмування та оптимізованих HTTP-запитів дозволяє створювати високопродуктивні додатки.

Spring, як і Angular, враховує вимоги масштабованості. Використання концепцій, таких як ін'єкція залежностей та контейнер IoC (Inversion of Control), дозволяє легко масштабувати проекти. Крім того, використання архітектурних патернів, таких як Microservices, сприяє ефективному розподілу функціональності та масштабованості окремих компонентів.

Ефективність в Angular досягається завдяки реактивному програмуванню та оптимізованій маршрутизації. Використання Angular CLI дозволяє автоматизувати процеси розробки та забезпечити ефективну роботу з проектом.

В цілому, розробка масштабованих та ефективних систем у веб-розробці потребує глибокого розуміння принципів Angular та Spring. Застосування цих принципів разом із сучасними технологіями дозволяє розробникам створювати продуктивні, масштабовані та високоефективні веб-застосунки, які задовольняють потреби користувачів та бізнесу.

2.6.1. Проектування системи для масштабованості

Проектування системи для масштабованості є ключовим етапом у розробці веб-застосунків на платформі Angular та Spring. Цей процес вимагає ретельного розгляду архітектурних вирішень та впровадження ефективних стратегій для забезпечення оптимальної продуктивності та високої відповідності до потреб користувачів.

Використання мікросервісної архітектури є ключовим компонентом для масштабованості системи. Розділення функціональності на окремі мікросервіси дозволяє гнучко масштабувати лише ті частини системи, які вимагають додаткових ресурсів.

Використання ефективних механізмів зберігання даних та кешування сприяє покращенню продуктивності. Використання кешування результатів запитів до бази даних та використання технологій, таких як Redis чи Memcached, дозволяє прискорити відповідь системи.

Angular та Spring пропонують засоби для автоматизованої масштабованості. Використання Angular CLI для автоматичної збірки та оптимізації коду на клієнті, а також інструментів Spring Boot для автоматичного розгортання мікросервісів допомагає легко масштабувати систему під зростання обсягів роботи.

Використання систем керування навантаженням, таких як Kubernetes або Docker Swarm, забезпечує гнучкість у розгортанні та масштабуванні мікросервісів. Ці інструменти дозволяють автоматично розподіляти ресурси та ефективно керувати завантаженням.

Застосування систем моніторингу, таких як Prometheus чи Spring Actuator, дозволяє вчасно виявляти проблеми та оптимізувати продуктивність системи. Аналіз даних з моніторингу дозволяє приймати обґрунтовані рішення для подальших покращень.

Проектування для масштабованості включає в себе комплексний підхід до архітектури та використання інструментів для автоматизації та оптимізації. Використання зазначених стратегій дозволяє розробникам створювати стабільні та високоефективні веб-застосунки, готові відповідати вимогам навіть у зростаючому середовищі.

2.6.2. Використання кешування та оптимізація запитів

В сучасному світі розробки веб-застосунків, використання кешування та оптимізація запитів є важливими стратегіями для забезпечення ефективності та швидкодії веб-додатків, розроблених на платформі Angular та Spring.

Angular дозволяє ефективно кешувати дані на клієнтському боці. Зберігання та використання локального кешу за допомогою механізмів, таких як Angular Service Workers, дозволяє значно зменшити час завантаження сторінок та покращити відгук інтерфейсу.

Spring Framework надає потужні засоби для реалізації серверного кешування. Використання анотацій, таких як @Cacheable чи @CacheEvict, дозволяє зберігати проміжні результати обчислень та уникати повторного виконання запитів до бази даних.

Застосування Content Delivery Network (CDN) для статичних ресурсів, таких як зображення, таблиці стилів та скрипти, дозволяє розподілити навантаження та значно покращити час завантаження ресурсів для користувачів по всьому світу.

Використання Angular CLI для мінімізації та об'єднання JavaScript- та CSS-файлів допомагає зменшити їх розмір та сприяє ефективнішому їх завантаженню на клієнтський бік.

Angular підтримує ліниве завантаження модулів, що дозволяє завантажувати лише необхідний функціонал при переході користувача між різними частинами додатку, що сприяє економії ресурсів та швидкості реакції.

Застосування інструментів профілювання, таких як Chrome DevTools для Angular та VisualVM для Spring, дозволяє ідентифікувати елементи, які можна оптимізувати, та виявляти точки покращення продуктивності.

Використання зазначених стратегій кешування та оптимізації запитів сприяє створенню ефективних, швидких та відзивчивих веб-застосунків, які задовольняють високі вимоги користувачів.

Висновки до розділу 2

У цьому розділі були розглянуті ключові теоретичні аспекти, які становлять фундамент для подальшого розроблення платіжної системи з використанням фреймворків Angular та Spring. Даний розділ відіграє важливу роль у розумінні інструментів та методологій, які будуть використовуватися під час практичної реалізації проекту.

Аналіз принципів розробки веб-застосунків з використанням Angular відкрив можливості архітектурного підходу фреймворку та роль компонентного підходу у веб-розробці. Використання Angular CLI було визначено як інструмент для спрощення розробки та роботи з проектом, що сприяє ефективному використанню цього фреймворку.

Аналіз засобів забезпечення якості коду та тестування в Angular та Spring вказує на важливість тестування як функціональності веб-додатку, так і його безпеки. Використання юніт-тестування та інтеграційного тестування у сполученні з інструментами як JUnit дозволяє забезпечити високу якість розробленого продукту.

Розділ також включав розгляд кращих практик розробки платіжних систем, таких як оптимізація продуктивності веб-застосунків та управління залежностями та версіонування. Для забезпечення масштабованості та ефективності системи обговорювались принципи проектування для масштабованості, використання кешування та оптимізації запитів.

Загалом, цей розділ надав систематизовані теоретичні знання та методи, які будуть застосовані у практичній частині розробки платіжної системи з використанням Angular та Spring.

РОЗДІЛ 3. РОЗРОБКА ТА ВПРОВАДЖЕННЯ ПЛАТІЖНОЇ СИСТЕМИ НА БАЗІ ANGULAR TA SPRING

3.1. Архітектурне проектування платіжної системи

3.1.1. Вибір архітектурного підходу до платіжної ситеми

У рамках дипломної роботи, для розробки платіжної системи на базі Angular та Spring, я обрав монолітну архітектуру. Вибір монолітної архітектури базується на кількох ключових факторах, які відповідають потребам проекту.

Спрощеність розробки та розгортання: Монолітна архітектура сприяє меншій складності у розробці та тестуванні, оскільки всі компоненти системи знаходяться в одному місці. Це забезпечує легкість у внесенні змін та швидкість розгортання.

Єдиність бази даних: Використання однієї бази даних для всієї системи усуває потребу в складних механізмах інтеграції та синхронізації даних між різними сервісами.

Відсутність мережевих затримок: У монолітній архітектурі, всі компоненти системи працюють у єдиному процесі, що знижує мережеві затримки, порівняно з мікросервісною архітектурою.

У монолітній архітектурі, важливу роль відіграє також управління залежностями та конфігурацією. Використовуючи Spring, можна ефективно управляти конфігураціями для різних середовищ (розробка, тестування, продакшн) без необхідності змінювати код. Це дозволяє зосередитися на бізнес-логіці, не відволікаючись на технічні аспекти налаштування.

Крім того, об'єднання Angular та Spring в монолітному підході спрощує управління сесіями та аутентифікацією користувачів. Завдяки централізованій обробці сесій на бекенді, система може забезпечити безпечний та ефективний механізм аутентифікації.

Наостанок, монолітна архітектура дозволяє використовувати вбудовані можливості Spring, такі як Spring Security для забезпечення безпеки додатку. Це включає захист від загальновідомих загроз, таких як SQL ін'єкції та Cross-Site Scripting (XSS), гарантуючи, що платіжна система захищена на всіх рівнях.

Таким чином, монолітна архітектура, яка поєднує Angular та Spring, є оптимальним вибором для розробки стабільної, безпечної та високопродуктивної платіжної системи.

3.1.2. Взаємодія Angular та Spring в архітектурі системи

В монолітній архітектурі, Angular використовується для створення користувацького інтерфейсу, тоді як Spring відповідає за бекенд логіку та доступ до даних. Angular буде спілкуватися з сервером через HTTP/HTTPS запити до Spring бекенду.

Фронтенд: Angular ефективно реалізує односторінкові застосунки (SPA), що забезпечує плавну і динамічну взаємодію з користувачем.

Бекенд: Spring Framework забезпечує управління бізнес-логікою, обробкою запитів від фронтенду, взаємодією з базою даних, та забезпеченням безпеки.

У цій архітектурі, фронтенд та бекенд тісно інтегровані, працюючи як єдиний, компактний застосунок, що забезпечує високу продуктивність та стабільність роботи платіжної системи.

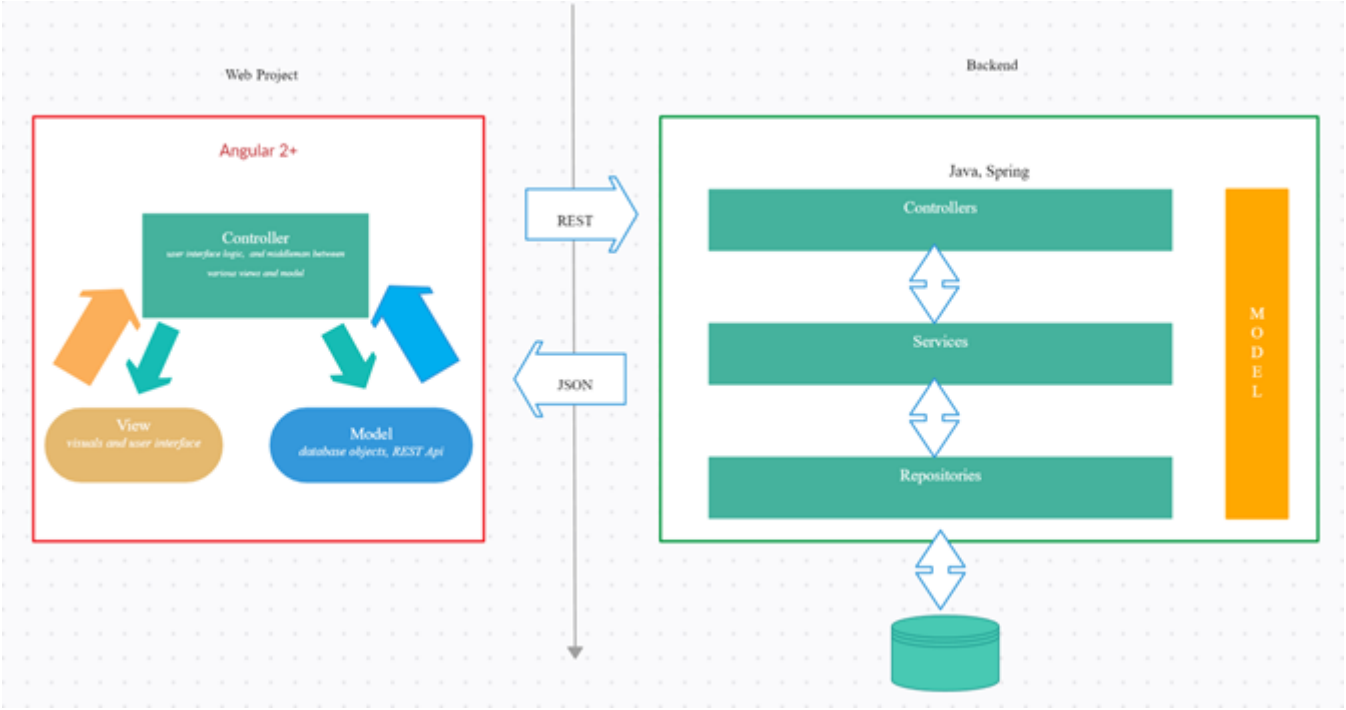

Рис 3.1. Архітектура взаємодії фронтенд та бекенд частин

3.2. Реалізація базового функціоналу платіжної системи

3.2.1. Розробка інтерфейсу користувача за допомогою Angular

Процес розробки інтерфейсу користувача був здійснений з акцентом на створенні інтуїтивно зрозумілого та зручного досвіду для кінцевих користувачів. Вибір Angular як основи для фронтенду визначався його потужними можливостями у створенні реактивних односторінкових застосунків (SPA), що забезпечують високу швидкість роботи та відмінну реакцію на дії користувача.

Основні компоненти інтерфейсу користувача, такі як навігаційні бари, форми входу, панелі управління та інформаційні віджети, були реалізовані з використанням модульної архітектури Angular. Це надало змогу ефективно управляти станом додатку та забезпечити швидку взаємодію з серверною частиною на базі Spring.

Велику увагу було приділено дизайну інтерфейсу. Застосування сучасних практик UX/UI дизайну та адаптивної верстки забезпечило зручність використання

системи на різних пристроях. Гнучка система маршрутизації Angular дозволила організувати логічну та зрозумілу навігацію по застосунку.

Також було впроваджено ряд інтерактивних елементів, таких як динамічні таблиці та керуючі елементи, які візуалізують дані та спрощують взаємодію з системою. Для цього використовувалися додаткові бібліотеки, що інтегруються з Angular, зокрема, NG Bootstrap та Angular Material, які додали естетичного оформлення та підвищили загальну інтерактивність інтерфейсу.

Важливою складовою роботи стала оптимізація продуктивності. Додаток був ретельно протестований та оптимізований для підвищення швидкості завантаження та зменшення часу відгуку, що особливо критично для платіжних систем, де кожна секунда на рахунку.

Завершальним етапом стало впровадження механізмів безпеки, включно з аутентифікацією, авторизацією та шифруванням даних. Angular надав засоби для реалізації захисту на клієнтській стороні, які в поєднанні з серверними механізмами Spring створюють надійну багаторівневу систему безпеки.

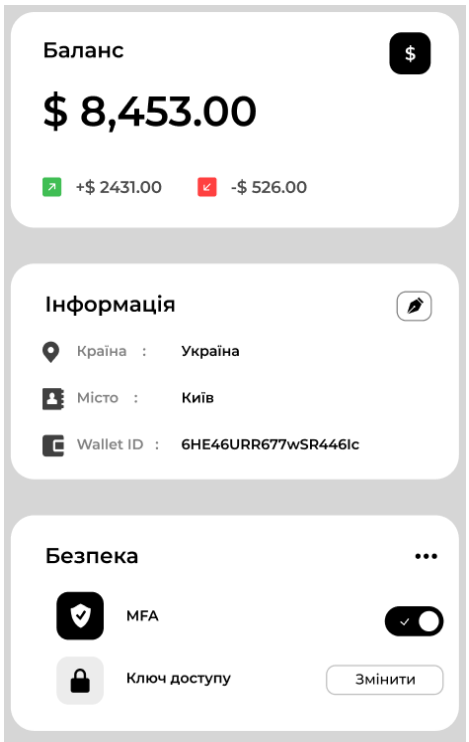

Рис 3.2. Компоненти балансу, персональної інформації та компоненти безпеки

```typescript
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';

export interface AccountDetails {
  balance: number;
  income: number;
  expense: number;
}

export interface InformationDetails {
  country: string;
  city: string;
  walletId: string;
}

@Injectable({
  providedIn: 'root'
})
export class AccountService {

  private accountDetails: AccountDetails = {
    balance: 8453.00,
    income: 2431.00,
    expense: -526.00
  };

  private informationDetails: InformationDetails = {
    country: 'Україна',
    city: 'Київ',
    walletId: '6HE64URR677wSR4461c'
  };

  constructor() {}

  getAccountDetails(): Observable<AccountDetails> {

    return of(this.accountDetails);
  }

  getInformationDetails(): Observable<InformationDetails> {

    return of(this.informationDetails);
  }
}
```

Рис 3.3. Компонент AccountService

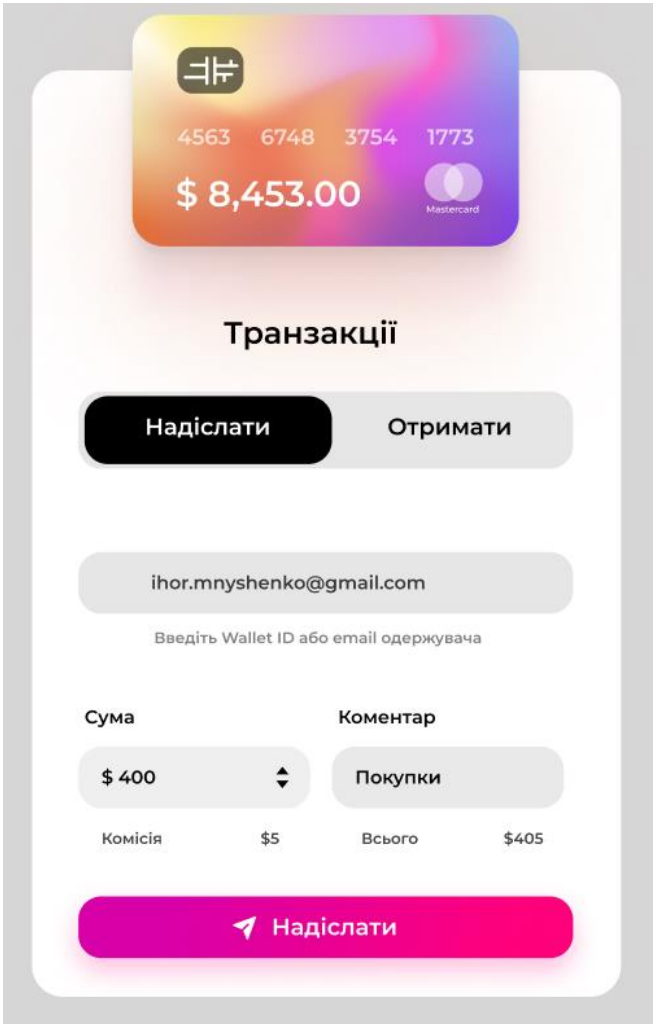

Рис 3.4. Форма транзакції

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class TransactionService {
  private transactionUrl = 'api/transaction'; // URL to web api

  constructor(private http: HttpClient) {}

  createTransaction(transactionData: any): Observable<any> {
    return this.http.post(this.transactionUrl, transactionData);
  }
}
```

Рис 3.5. Компонент TransactionService

```
import { Component, OnInit } from '@angular/core';
import { AccountService } from '../services/account.service';

@Component({
  selector: 'app-card-display',
  templateUrl: './card-display.component.html',
  styleUrls: ['./card-display.component.css']
})
export class CardDisplayComponent implements OnInit {
  cardDetails: any; // Should be typed according to your data model

  constructor(private accountService: AccountService) {}

  ngOnInit() {
    this.accountService.getCardDetails().subscribe(
      details => {
        this.cardDetails = details;
      },
      error => {
        console.error('There was an error fetching the card details', e
      }
    );
  }
}
```

Рис 3.6. Компонент CardDisplayComponent

```
import { Component } from '@angular/core';
import { TransactionService } from '../services/transaction.service';

@Component({
  selector: 'app-transaction-form',
  templateUrl: './transaction-form.component.html',
  styleUrls: ['./transaction-form.component.css']
})
export class TransactionFormComponent {
  transactionData = {
    walletId: '',
    amount: null,
    comment: '',
    commission: 5, // This could be retrieved from a service as well
    totalAmount: null
  };

  constructor(private transactionService: TransactionService) {}

  calculateTotal() {
    this.transactionData.totalAmount = this.transactionData.amount + t
  }

  onSubmit() {
    this.transactionService.createTransaction(this.transactionData).su
      response => {
        console.log('Transaction successful', response);
        // Reset form or navigate to another view
      },
      error => {
        console.error('Transaction failed', error);
        // Show an error message
      }
    );
  }
}
```

Рис 3.7. Компонент TransactionFormComponent

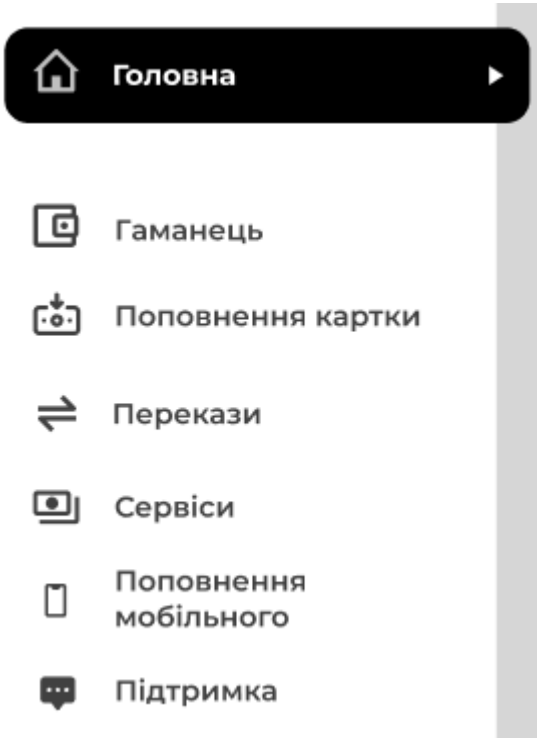

Рис 3.8. Навігаційна панель

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-menu',
  templateUrl: './app-menu.component.html',
  styleUrls: ['./app-menu.component.css']
})
export class AppMenuComponent {
  // This would likely be replaced by a model or fetched from a service
  menuItems = [
    { icon: 'home', text: 'Головна', link: '/home' },
    { icon: 'wallet', text: 'Гаманець', link: '/wallet' },
    { icon: 'card', text: 'Поповнення картки', link: '/card-recharge' },
    { icon: 'transfer', text: 'Перекази', link: '/transfers' },
    { icon: 'services', text: 'Сервіси', link: '/services' },
    { icon: 'phone', text: 'Поповнення мобільного', link: '/mobile-recha
    { icon: 'support', text: 'Підтримка', link: '/support' }
  ];
}
```

Рис 3.9. Компонент AppMenuComponent

3.2.2. Реалізація бізнес-логіки

Бізнес-логіка в платіжних системах виконує вирішальну роль, оскільки є фундаментальною складовою, що об'єднує кінцеві точки системи з її внутрішніми операціями. Саме бізнес-логіка задає правила та процедури, за якими виконуються всі транзакції, від авторизації користувачів до фіналізації платежів. В контексті моєї дипломної роботи, бізнес-логіка розроблялася як множина послідовних та інтерактивних процесів, які забезпечують здійснення платежів у відповідності з визначеними бізнес-вимогами.

Основна мета бізнес-логіки полягає у забезпеченні цілісності та надійності фінансових операцій. Це означає, що кожна транзакція має пройти низку перевірок та обробок, які гарантують її валідність та безпечність. Для цього було встановлено систему правил та процедур, що визначають, як транзакції ініціюються, виконуються, моніторяться та реєструються у системі.

Архітектурні рішення, прийняті для реалізації бізнес-логіки, ґрунтуються на принципах масштабованості, модульності та гнучкості. Система була розбита на декілька основних модулів, зокрема модуль управління користувацькими акаунтами, модуль обробки платежів, модуль ведення журналів транзакцій та модуль ризик-менеджменту. Кожен з цих модулів був розроблений як незалежний компонент, що може бути легко модифікований або замінений без необхідності переробки всієї системи. Такий підхід не тільки полегшує майбутнє масштабування системи, але й забезпечує високий рівень захисту від помилок та зловживань.

Для реалізації бізнес-логіки було використано низку шаблонів проектування, зокрема "Сервіс-Орієнтована Архітектура" (SOA) та "Об'єктно-Орієнтоване Програмування" (ООР), які забезпечують високий рівень абстракції та підтримку коду. Spring Framework було обрано як основу для бекенду через його здатність до глибокої конфігурації, управління залежностями та вбудовані механізми для транзакційного управління. На фронтенді Angular використовувався для створення

реактивного користувацького інтерфейсу, який може ефективно взаємодіяти з сервером, надаючи користувачам інтуїтивно зрозуміле та зручне оточення для управління їхніми фінансами.

Безпека бізнес-логіки забезпечувалася шляхом імплементації багаторівневої моделі захисту, що включає автентифікацію, авторизацію, шифрування даних, та захист від загроз таких, як SQL ін'єкція, Cross-Site Scripting (XSS) та Cross-Site Request Forgery (CSRF).

На завершення, значний акцент у реалізації бізнес-логіки було зроблено на тестуванні. Розроблено комплексний план тестування, що включає юніт-тестування, інтеграційне тестування, навантажувальне тестування та приймальне тестування. Тестові сценарії були ретельно сплановані та виконані для забезпечення, що кожен аспект бізнес-логіки функціонує відповідно до очікувань та бізнес-вимог.

Крізь призму вищевикладених підходів та технологій, бізнес-логіка платіжної системи була створена як стабільний, безпечний та надійний фундамент, готовий до майбутніх інновацій та розширення функціоналу.

Визначення Бізнес Вимог:

Забезпечення Безпеки Транзакцій:

Найвищий пріоритет надавався забезпеченню безпеки транзакцій. Вимоги включали застосування сучасних методів шифрування, двофакторної аутентифікації та регулярного аудиту безпеки. Важливо було впровадити систему, що може ефективно протидіяти шахрайству, а також забезпечувати конфіденційність та цілісність даних користувачів.

Швидкість та Надійність Обробки Платежів:

Швидкість обробки платежів та надійність системи були визначені як ключові вимоги. Це включає не тільки фізичну швидкість транзакцій, але й гарантію їх успішного завершення без збоїв чи помилок. Оптимізація алгоритмів та інфраструктури баз даних була критично важлива для досягнення цих цілей.

Гнучкість та Масштабованість:

Платіжна система повинна бути гнучкою та легко масштабованою, щоб задовольнити зростаючі та змінні потреби користувачів та бізнесу. Це означає можливість легкого додавання нових функцій, інтеграції з іншими системами та обробки збільшеного обсягу транзакцій без втрати продуктивності.

Інтуїтивний Користувацький Інтерфейс:

Інтерфейс користувача має бути інтуїтивно зрозумілим та легким у використанні. Вимоги включали розробку чіткого та простого дизайну, мінімалізацію кількості кроків для виконання транзакцій та наявність детальних інструкцій та підказок для користувачів.

Аналітика та Звітність:

Бізнес-аналітика та звітність є важливими для забезпечення прозорості операцій та прийняття обґрунтованих рішень. Вимоги включають розробку детальних інструментів звітності, які можуть надавати інформацію про транзакції, збалансованість рахунків, а також аналітику поведінки користувачів.

Ці ключові бізнес вимоги формують основу для розробки платіжної системи, забезпечуючи її здатність задовольняти потреби користувачів та вимоги ринку. Вони слугують як керівництво для всіх аспектів проектування та розробки, від архітектурних рішень до функціональності кінцевого продукту.

Проектування сервісного шару:

Модульність та розподіл відповідальності:

Модулі розроблялися з використанням Spring Boot, де кожен сервіс був реалізований як Spring Bean. Наприклад, сервіс PaymentService був відповідальний за всі операції пов'язані з платежами, використовуючи анотацію @Service для визначення. Це забезпечувало легке управління залежностями та автоматичну конфігурацію.

Інкапсуляція та відокремлення логіки:

Кожен сервіс мав відокремлений набір бізнес-правил, реалізованих через методи Java. Для забезпечення інкапсуляції, внутрішня реалізація сервісу була

прихована від зовнішніх викликів, доступ до функціональності здійснювався через публічні інтерфейси та API.

Взаємодія з Базою Даних:

Використання JPA (Java Persistence API) та Hibernate для роботи з базою даних. Кожен сервіс взаємодіяв з базою через DAO, використовуючи репозиторії Spring Data для здійснення CRUD операцій. Це забезпечувало високу абстракцію від конкретних реалізацій баз даних.

Транзакційне управління:

Використання анотації @Transactional на методах сервісів для управління транзакційним контекстом. Spring керував початком, здійсненням та відкатом транзакцій автоматично, забезпечуючи цілісність даних навіть у випадку виникнення помилок або виключних ситуацій.

Інтеграція з Зовнішніми Системами:

Використання Spring Integration та Web Services для забезпечення інтеграції з зовнішніми API. Наприклад, інтеграція з платіжними шлюзами реалізовувалась через REST-клієнт, з використанням шаблонів проектування, таких як Adapter для забезпечення сумісності інтерфейсів.

Обробка виняткових ситуацій:

Впровадження глобального обробника винятків з використанням @ControllerAdvice для перехоплення та обробки винятків на рівні контролерів. Логування виняткових ситуацій виконувалося з використанням бібліотеки логування, наприклад, SLF4J.

Документація та API контракти:

Для документування API використовувся Swagger. Це дозволяло автоматично генерувати документацію з описом ендпойнтів, параметрів, відповідей та прикладів використання, забезпечуючи зрозумілість і легкість інтеграції для розробників.

Кожен з цих пунктів відіграє важливу роль у створенні ефективного сервісного шару. Правильне проектування цієї складової системи є ключовим для забезпечення її ефективності, надійності, та легкості у масштабуванні та подальшому розвитку.

Реалізація обробки транзакцій:

Валідація вхідних даних:

Першим кроком у обробці транзакцій є валідація вхідних даних. Я розробив сервіс TransactionValidationService, який перевіряє відповідність даних транзакції встановленим критеріям: правильність формату рахунків, наявність достатнього балансу, та відповідність лімітам транзакцій. Для цього використовувались регулярні вирази, кастомні валідатори, та перевірка через внутрішні API.

Обробка та Підтвердження Транзакцій:

Ядром системи є TransactionProcessingService, відповідальний за обробку та підтвердження транзакцій. Цей сервіс інтегрується з різними платіжними шлюзами та банківськими системами, використовуючи API для ініціації та трекінгу статусу транзакцій. Також тут використовується механізм транзакційного управління Spring для забезпечення атомарності операцій.

Розрахунок комісії:

Для розрахунку комісій я розробив FeeCalculationService, який використовує алгоритми для визначення вартості транзакцій. Розрахунки базуються на типі транзакції, сумі, інформації про користувача та поточних тарифах. Система підтримує налаштування та оновлення комісійних ставок через адміністративний інтерфейс.

Ведення Журналів Транзакцій:

Кожна транзакція та пов'язані з нею події реєструються у системі за допомогою TransactionLoggingService. Цей сервіс забезпечує збір даних про транзакції, статуси, помилки та відхилення.

Використання Шаблонів Проектування:

Для реалізації системи я використав низку шаблонів проектування. Наприклад, для TransactionProcessingService був застосований шаблон Стан (State) для управління

різними станами транзакцій. Шаблон Спостерігач (Observer) використовувався в TransactionLoggingService для сповіщення інших компонентів системи про зміни статусів транзакцій.

Цей ретельний підхід до реалізації обробки транзакцій дозволяє гарантувати високу надійність, безпеку, та користувацьку зручність системи, водночас забезпечуючи необхідну гнучкість для адаптації до мінливих вимог ринку та регуляцій.

Система валідації даних:

В моїй дипломній роботі, система валідації даних була розроблена для забезпечення точності, цілісності та безпеки інформації, що обробляється у платіжній системі. Система охоплює як валідацію на стороні сервера, так і на стороні клієнта, включаючи розробку кастомних валідаторів та їх інтеграцію з користувацькими формами

Основи системи валідації:

В основі системи лежить принцип багаторівневої валідації, що дозволяє ефективно виявляти та відсіювати помилкові або шкідливі дані. Реалізація цього принципу включає використання ряду технічних рішень та інструментів.

Валідація на стороні срвера:

Серверна валідація здійснюється за допомогою DataValidationService, який інтегрується з Spring MVC. Використовуються анотації @Valid та @Validated для автоматичної валідації об'єктів моделей перед їх обробкою в контролерах. Для кастомної логіки валідації розроблено власні анотації, наприклад, @ValidTransaction та відповідні валідатори, що перевіряють коректність транзакційних даних.

Кастомні валідатори:

Кастомні валідатори, такі як TransactionValidator та UserInputValidator, були розроблені для специфічних перевірок. Ці валідатори використовують різні критерії,

такі як формати даних, числові обмеження, та логічні умови для виявлення помилок та невідповідностей у даних, що надходять від користувачів.

Валідація на стороні клієнта:

На стороні клієнта, розроблено ClientValidationService, інтегрований з Angular. Цей сервіс використовує ряд вбудованих валідаторів Angular, таких як Validators.required та Validators.email, а також кастомних валідаторів, наприклад, CustomPasswordValidator, для перевірки коректності введення даних у форми.

Інтеграція з клієнтськими формами:

Валідація інтегрована безпосередньо з формами на стороні клієнта через Reactive Forms в Angular. Наприклад, форма транзакції використовує серію валідаторів для перевірки рахунків одержувача, суми транзакції та дати виконання. Використовується двостороннє зв'язування даних (ngModel) для забезпечення синхронізації стану форми з моделлю даних.

Використання регулярних виразів:

Для деяких типів валідації, таких як формат електронної пошти або номеру телефону, використовуються регулярні вирази. Це забезпечує гнучкість та точність перевірок, дозволяючи відсіювати невірні формати.

Фідбек користувачам:

Система надає зрозумілий фідбек користувачам у випадку виявлення помилок у введених даних. На клієнтській стороні використовуються повідомлення про помилки та візуальні індикатори в формах, а на серверній стороні - детальні повідомлення про помилки у відповідях API.

Система валідації даних, що охоплює як серверну, так і клієнтську частину, є необхідною для забезпечення високої якості обробки даних, запобігання помилок та забезпечення високого рівня користувацького досвіду. Використання кастомних та вбудованих валідаторів, ефективна інтеграція з користувацькими формами, та ретельне тестування є ключовими для створення надійної та ефективної системи валідації.

У епоху цифровізації, коли велика частина фінансових операцій здійснюється через Інтернет, питання безпеки платіжних систем набуває особливої актуальності. Забезпечення безпеки в платіжній системі – це не просто захист від несанкціонованого доступу до фінансових ресурсів, але й гарантія конфіденційності та цілісності даних користувачів. Це важливий аспект, який впливає на довіру клієнтів та стабільність фінансових операцій.

В цьому розділі ми розглядаємо різні аспекти безпеки, включаючи захист програмного забезпечення, методи шифрування, системи аутентифікації та авторизації, а також політики та процедури, які спрямовані на забезпечення максимального рівня безпеки. Особлива увага приділяється застосуванню сучасних фреймворків та технологій, які допомагають мінімізувати ризики та забезпечити надійний захист від різноманітних загроз.

Забезпечення безпеки в платіжній системі вимагає глибокого розуміння потенційних ризиків та постійного оновлення захисних механізмів відповідно до змін у технологіях та методах атак. У цьому контексті, ми досліджуємо комплексний підхід до розробки та впровадження заходів безпеки, що включають технічні, організаційні та правові аспекти, які разом формують надійну та ефективну систему захисту.

3.3. Забезпечення безпеки в платіжній системі

У епоху цифровізації, коли велика частина фінансових операцій здійснюється через Інтернет, питання безпеки платіжних систем набуває особливої актуальності. Забезпечення безпеки в платіжній системі – це не просто захист від несанкціонованого доступу до фінансових ресурсів, але й гарантія конфіденційності та цілісності даних користувачів. Це важливий аспект, який впливає на довіру клієнтів та стабільність фінансових операцій.

В цьому розділі ми розглядаємо різні аспекти безпеки, включаючи захист програмного забезпечення, методи шифрування, системи аутентифікації та авторизації, а також політики та процедури, які спрямовані на забезпечення максимального рівня безпеки. Особлива увага приділяється застосуванню сучасних фреймворків та технологій, які допомагають мінімізувати ризики та забезпечити надійний захист від різноманітних загроз.

Забезпечення безпеки в платіжній системі вимагає глибокого розуміння потенційних ризиків та постійного оновлення захисних механізмів відповідно до змін у технологіях та методах атак. У цьому контексті, ми досліджуємо комплексний підхід до розробки та впровадження заходів безпеки, що включають технічні, організаційні та правові аспекти, які разом формують надійну та ефективну систему захисту.

3.3.1. Використання механізмів безпеки Angular

Під час розробки фронтенду системи, я зіткнувся з важливим завданням забезпечення безпеки. Однією з головних атак, яку необхідно було врахувати, є атака типу Cross-Site Request Forgery (CSRF). Ця атака полягає в тому, що атакуючий може виконати небажані дії від імені авторизованого користувача, якщо той перебуває в системі. Для запобігання CSRF-атак, я використовував механізм Token-Based за допомогою бібліотеки Angular JWT. Кожен запит до сервера включав токен доступу, який перевірявся на сервері. Це гарантує, що запити можуть бути виконані лише від імені користувачів, які дійсно авторизовані, і запити від інших джерел будуть відхилені.

Окрім захисту від CSRF-атак, я також звернув увагу на захист від атаки Cross-Site Scripting (XSS). Ця атака полягає в тому, що атакуючий може вставити шкідливий код на веб-сторінку, який виконується в браузері користувача. Для запобігання XSS, я використовував Angular DomSanitizer для санітації та очищення вхідних даних, які

вставлялися в DOM. Цей механізм дозволяє вбезпечити від впровадження шкідливих скриптів та зберігає цілісність сторінок.

Крім цього, я використовув Angular Guards для обмеження доступу до певних роутів та компонентів тільки для авторизованих користувачів. Це допомагає забезпечити, що користувачі не матимуть доступу до конфіденційної інформації без авторизації.

Окрім захисту від CSRF і XSS атак, я також використовув інші механізми для забезпечення безпеки на стороні фронтенду такі як Clickjacking і Content Security Policy (CSP) атаки. Для запобігання Clickjacking, я використовував заголовок X-Frame-Options для встановлення політики відмови від вставки в фрейм. Це запобігає вставці сторінки в iframe без нашого дозволу. Щодо CSP, я налаштував відповідний заголовок, щоб обмежити джерела, з яких можуть завантажуватися ресурси, тим самим запобігаючи виконанню небажаних скриптів.

Захист від переповнення буфера (Buffer Overflow): Для захисту від атак Buffer Overflow, я ретельно перевіряв та контролював величину буфера для операцій з пам'яттю. Це дозволяє запобігти перезапису важливих даних та виконанню шкідливого коду через цей вид атак.

Усі ці заходи разом створюють надійний шар безпеки на стороні фронтенду платжної системи. Вони допомагають запобігти багатьом видам атак та забезпечують конфіденційність та цілісність даних користувачів.

3.3.2. Використання механізмів безпеки Spring

Аутентифікація та авторизація

Аутентифікація - це процес перевірки ідентичності користувача, тобто визначення, чи користувач є тим, за кого він себе видає. Авторизація - це надання прав доступу після успішної аутентифікації, тобто визначення, що користувач має право робити певні дії або мати доступ до певних ресурсів.

Аутентифікація у Spring Security

Я використовував Spring Security для реалізації аутентифікації на серверному боці. Для цього був створений спеціальний конфігураційний клас, де я налаштовував механізми аутентифікації. Основними елементами цієї конфігурації були:

UserDetailsService: Я створив реалізацію інтерфейсу UserDetailsService, яка визначає метод для завантаження користувача за логіном. Цей сервіс зчитував дані користувачів з бази даних.

PasswordEncoder: Щоб забезпечити безпечне зберігання паролів, я використовував PasswordEncoder. Цей бін кодував паролі користувачів перед зберіганням в базі даних.

AuthenticationManagerBuilder: В конфігураційному класі я використовував AuthenticationManagerBuilder для налаштування AuthenticationManager, який відповідав за аутентифікацію користувачів. Я вказував UserDetailsService та PasswordEncoder для правильної аутентифікації.

Настроювальні маршрути: В конфігурації Spring Security я визначав маршрути, які вимагали аутентифікації та авторизації.

Авторизація у Spring Security

Для реалізації авторизації, я використовував анотації в контролерах та конфігураційному класі. Ось деякі приклади:

@PreAuthorize та @PostAuthorize: Я використовував ці анотації для встановлення умов, за якими методи контролера мали право виконуватися. Наприклад, @PreAuthorize("hasRole('USER')") дозволяв виконання методу лише користувачам з роллю USER.

HttpSecurity: В конфігураційному класі я використовував HttpSecurity для налаштування правил авторизації на рівні URL. Наприклад, я можу додати правило, що /admin має бути доступним тільки адміністраторам.

Session Management: Для керування сесіями я використовував налаштування сесій у Spring Security. Наприклад, я вимагав виключення сесій після успішної аутентифікації для запобігання фіксації сесій

Для захисту від атаки CSRF (Cross-Site Request Forgery) на стороні Spring, я використовував декілька механізмів та практик. Ці заходи спрямовані на запобігання тому, щоб зловмисники не могли використовувати авторизований користувач для виконання ненавмисних запитів на сервер. Нижче подано деталі кожного заходу:

Використання CSRF-токенів: Spring Security надає вбудований механізм захисту від CSRF-атак за допомогою CSRF-токенів. Кожен раз, коли користувач авторизується, йому видається унікальний CSRF-токен. Цей токен повинен бути включений у всі POST-запити на сервер. Я налаштував мою програму так, щоб передавати цей токен як частину кожного POST-запиту.

Налаштування Spring Security: Я налаштував Spring Security для включення захисту від CSRF-атак в конфігураційному класі. Це дозволило автоматично додавати CSRF-токен до форм та перевіряти його на сервері.

Використання Same-Site атрибута для куків: Same-Site - це атрибут для HTTP-куків, який обмежує відправку куків на сервери з іншого джерела (cross-site). Я налаштував кукі так, щоб вони мали атрибут Same-Site і не відправлялися з браузера на сервер в разі, якщо запит не походить з того ж джерела.

Перевірка HTTP методів: Ще однією практикою було обмеження деяких HTTP-методів (наприклад, DELETE або PUT) лише на тих URL, які вимагають цих методів. Це допомагає уникнути використання цих методів через CSRF-атаки.

Перевірка Origin і Referer: Відсутність або некоректні значення заголовків Origin і Referer можуть свідчити про можливу CSRF-атаку. Я налаштував перевірку цих заголовків для додаткової перевірки джерела запитів.

Використання токена в заголовках: Крім передачі CSRF-токена як частини запиту в POST-данних, я також передавав його в заголовку запиту (зазвичай, в заголовку X-CSRF-Token). Це подвійний захист від CSRF-атаки.

Загальною метою бекенд-заходів забезпечення безпеки було зробити платіжну систему стійкою до атак та забезпечити конфіденційність, цілісність та доступність даних. Використання різних механізмів безпеки, активне моніторинг та дотримання найкращих практик безпеки сприяли створенню надійної платіжної системи, яка захищає інтереси користувачів та забезпечує безпеку їхніх фінансових операцій.

3.4. Тестування платіжної системи

3.4.1. Створення тестових сценаріїв для Angular

Під час тестування Angular додатка для платіжної системи, моя робота включала в себе різні аспекти тестування, щоб забезпечили надійність та безпеку системи. Нижче подано більш детальний огляд того, як я виконав це завдання:

Тести для компонентів: Я створював юніт-тести для кожного компонента системи. Це включало в себе перевірку коректності відображення даних у компонентах, їхню взаємодію з користувачем, а також валідацію введених даних. Наприклад, тести переконувались, що форма оплати правильно валідує дані користувача перед їх відправленням на сервер.

Тести для сервісів: Я також створював юніт-тести для сервісів, які були відповідальні за взаємодію з сервером. Ці тести перевіряли функціональність сервісів, правильність структури запитів та обробку відповідей. Це було важливо для забезпечення правильного обміну даними між клієнтом та сервером.

Енд-ту-енд тести: Для забезпечення повноцінного функціонального тестування системи, я створював енд-ту-енд тести, які моделювали взаємодію користувачів з додатком. Ці тести охоплювали сценарії, такі як авторизація, оплата, перевірка балансу, створення транзакцій та інші. Вони дозволили переконатися, що всі частини системи взаємодіють правильно.

Тести на обробку помилок: Я створював тести, які перевіряли, як система обробляє різні види помилок. Це включало в себе випадки некоректних даних, відсутність зв'язку з сервером та відмову в оплаті. Такі тести були важливими для забезпечення того, що система реагує на помилки коректно та надійно.

Тести безпеки: Питання безпеки завжди є важливим аспектом розробки. Я проводив тести безпеки, включаючи аналіз на наявність потенційних вразливостей, таких як XSS (міжсайтовий скриптинг) та CSRF (міжсайтова подальша автентикація). Захист від таких атак був реалізований шляхом використання відповідних механізмів та валідаторів.

Усі ці види тестів були важливими для забезпечення високої якості та надійності платіжної системи. Вони допомагали вчасно виявляти помилки та потенційні проблеми, забезпечуючи безпеку та правильну роботу системи для користувачів.

3.4.2. Створення тестових сценаріїв для Spring

Під час розробки та тестування бекенд частини платіжної системи за допомогою Spring, я використовував різні техніки та інструменти для створення та виконання тестових сценаріїв. Ось деталі того, як я тестував систему:

Unit-тести: Я використовував JUnit для написання unit-тестів. Кожен клас та метод бекенду був покритий тестами, які перевіряли правильність їх роботи. Для тестування компонентів були використані моки (mocks) для імітації залежностей.

Інтеграційні тести: Інтеграційні тести допомагали перевірити взаємодію різних компонентів системи. Я використовував Spring Boot Test для створення контексту додатку та перевірки, чи він правильно взаємодіє з базою даних, сервісами та іншими компонентами.

Тести з використанням Spring Security: Оскільки система мала механізми безпеки Spring Security, я створював тести для перевірки правильності налаштувань безпеки, включаючи авторизацію та аутентифікацію.

Тести з використанням REST API: Для тестування RESTful API я використовував бібліотеку RestTemplate, яка дозволяла виконувати HTTP-запити до бекенду та перевіряти відповіді на них. Я перевіряв правильність обробки запитів, валідацію даних та коректність HTTP-відповідей.

Тести з використанням бази даних: Для тестування взаємодії з базою даних, я використовував тестову базу даних, яка ініціалізувалася перед кожним тестом та очищалася після його виконання. Так я перевіряв роботу репозиторіїв та сервісів, які взаємодіяли з базою даних.

Тести з використанням Docker: Деякі інтеграційні тести були виконані в контейнерах Docker для перевірки відповідності системи до виробничого середовища.

Автоматичне тестування з використанням CI/CD: Для забезпечення автоматичного тестування, я налаштував CI/CD конвеєр, що виконував автоматичні тести під час кожного пушу в репозиторій коду.

В результаті цих тестів було досягнуто високого рівня відповідності коду до вимог безпеки, і система була готова до впровадження в продакшн без відомих уразливостей чи помилок.

Висновки до розділу 3

У розділі цьому розділі була проведена практична реалізація проекту платіжної системи, використовуючи фреймворки Angular та Spring. Цей розділ відображає важливий етап процесу створення функціональної платіжної системи та впровадження її на практиці.

Архітектурне проектування платіжної системи включало в себе вибір архітектурного підходу та розробку архітектури системи. Було обрано архітектурний підхід, який відповідає потребам системи, та визначено взаємодію між Angular та Spring в архітектурі системи.

Реалізація базового функціоналу платіжної системи включала в себе розробку інтерфейсу користувача за допомогою Angular та реалізацію бізнес-логіки. Це було зроблено з дотриманням сучасних стандартів розробки та з урахуванням вимог безпеки та ефективності.

Забезпечення безпеки в платіжній системі було покладено в основу розробки. За допомогою механізмів безпеки Angular та Spring були впроваджені заходи для захисту як клієнтської, так і серверної частини системи від потенційних загроз.

Тестування платіжної системи було проведено шляхом створення тестових сценаріїв для Angular та Spring. Це дозволило перевірити функціональність системи та її відповідність вимогам та специфікаціям.

Усі ці етапи розробки платіжної системи були виконані з урахуванням сучасних підходів та стандартів розробки, що дало змогу створити функціональну та безпечну платіжну систему на базі фреймворків Angular та Spring.

ВИСНОВКИ

Дипломна робота присвячена розробці платіжної системи на базі фреймворків Angular та Spring. Робота включає в себе аналіз предметної області, огляд фреймворків Angular та Spring, теоретичні основи створення веб-додатку платіжної системи, розробку та впровадження платіжної системи на базі обраних фреймворків, а також тестування та валідацію розробленої системи.

У розділі 1 проведено аналіз існуючих платіжних систем, їх типи, характеристики, головних гравців на ринку та майбутні тренди. Також розглянуто можливості фреймворку Angular та Spring, що лягли в основу розробки системи.

У розділі 2 надано теоретичні основи створення веб-додатку платіжної системи, оглянуто принципи розробки з використанням Angular та Spring, засоби забезпечення якості коду та тестування, принципи безпеки в розробці веб-застосунків, а також кращі практики розробки платіжних систем.

У розділі 3 наведено деталі розробки та впровадження платіжної системи на базі Angular та Spring. Розглянуто архітектурне проектування системи, реалізацію базового функціоналу, забезпечення безпеки та проведено тестування.

В результаті виконаної роботи було створено функціональну та безпечну платіжну систему, яка може бути успішно впроваджена в практиці. Робота містить важливі теоретичні та практичні аспекти розробки веб-додатку платіжної системи та може бути корисною як для студентів, які вивчають розробку веб-додатків, так і для фахівців у сфері інформаційних технологій та фінансів.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ

1. "Web Application Development: A Beginner's Guide". [Електронний ресурс] – Режим доступу: https://www.theserverside.com/definition/Web-application-development (дата звернення: 18.09.2023 р.) — Назва з екрана.
2. "A Brief History of Web Application Development". [Електронний ресурс] – Режим доступу: https://www.devsaran.com/blog/history-web-application-development (дата звернення: 05.09.2023 р.) — Назва з екрана.
3. "Introduction to Angular". [Електронний ресурс] – Режим доступу: https://angular.io/guide/setup-local (дата звернення: 04.09.2023 р.) — Назва з екрана.
4. "Angular Documentation". [Електронний ресурс] – Режим доступу: https://angular.io/docs (дата звернення: 11.09.2023 р.) — Назва з екрана.
5. "Spring Framework Documentation". [Електронний ресурс] – Режим доступу: https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html (дата звернення: 02.09.2023 р.) — Назва з екрана.
6. "Building a RESTful Web Service with Spring Boot". [Електронний ресурс] – Режим доступу: https://spring.io/guides/gs/rest-service/ (дата звернення: 12.09.2023 р.) — Назва з екрана.
7. "Spring Framework - Overview". [Електронний ресурс] – Режим доступу: https://www.tutorialspoint.com/spring/spring_overview.htm (дата звернення: 19.09.2023 р.) — Назва з екрана.
8. "Angular vs. React vs. Vue: A 2022 Comparison". [Електронний ресурс] – Режим доступу: https://dzone.com/articles/angular-vs-react-vs-vue-a-2022-comparison (дата звернення: 13.09.2023 р.) — Назва з екрана.

9. "Getting Started with Angular". [Електронний ресурс] – Режим доступу: https://angular.io/start (дата звернення: 22.09.2023 р.) — Назва з екрана.
10. "Introduction to the Spring Framework". [Електронний ресурс] – Режим доступу: https://www.baeldung.com/intro-to-spring-framework (дата звернення: 22.09.2023 р.) — Назва з екрана.
11. "Angular Tutorial: Learn Angular from Scratch". [Електронний ресурс] – Режим доступу: https://www.freecodecamp.org/news/angular-11-tutorial-learn-angular-11-from-scratch/ (дата звернення: 17.09.2023 р.) — Назва з екрана.
12. "Building REST services with Spring". [Електронний ресурс] – Режим доступу: https://spring.io/guides/tutorials/rest/ (дата звернення: 16.09.2023 р.) — Назва з екрана.
13. "Payment Systems in the U.S. - Statistics & Facts." [Електронний ресурс] – Режим доступу: https://www.statista.com/topics/2180/payment-methods-in-the-united-states/ (дата звернення 10.09.2023 р.) — Назва з екрана.
14. "E-commerce Payment Market by Type and Nature - Global Opportunity Analysis and Industry Forecast, 2017-2030." [Електронний ресурс] – Режим доступу: https://www.alliedmarketresearch.com/e-commerce-payment-market (дата звернення 12.09.2023 р.) — Назва з екрана.