МІНІСТЕРСТОВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри КІТ
_________ А. С. Савченко
«____» __________ 2020 р.

ДИПЛОМНИЙ РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

ЗА НАПРЯМОМ 122 «КОМП'ЮТЕРНІ НАУКИ»

Тема: «Система «Task manager» для здійснення управління ІТ-проектами»

Виконавець: студентка УС-211М Загурська Юліана Генріхівна
(студент, група, призвіще, ім'я, по батькові)

Керівник: доктор технічних наук, професор Зіатдінов Юрій Кашафович
(науковій ступень, вчене звання, призвіще, ім'я, по батькові)

Нормоконтролер ___________________ Райчев І. Е.
(підпис) (П.І.Б.)

КИЇВ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Навчально-науковий інститут комп’ютерних інформаційних технологій
Кафедра комп’ютерних інформаційних технологій
Освітній ступінь **Магістр**
Напрям 6.050101 Комп’ютерні науки
(шифр, найменування)

ЗАТВЕРДЖУЮ
Завідувач випускової кафедри

___________А. С. Савченко
« » _______________2020 р.

ЗАВДАННЯ

на виконання дипломного проекту студентки

Загурської Юліани Генріхівни
(прізвище, ім’я, по батькові)

1.Тема проекту: «Система «Task manager» для здійснення управління IT-проектами» затверджена наказом ректора № 2175/ст від 25.09.2019 р.

2. Термін виконання роботи: з 26.09.2019 р. по 08.02.2020 р.

3. Вихідні дані до роботи: розробка клієнт-серверної системи менеджменту задач на базі Spring-boot та Angular фреймворків.

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці): вступ, аналітичний огляд і постановка завдання, розробка ядра серверної частини у вигляді універсального алгоритму сортування карт та системи авторизації та автентифікації програмними засобами мови Java, розробка інтерфейсу користувача, висновок.

5. Перелік обов’язкового графічного матеріалу: огляд структури фреймворку Spring, огляд програмних засобів бібліотек мови Java.

6. Календарний план-графік:

	Етапи виконання дипломного проекту	Термін виконання етапів	Примітка
1	Аналіз літератури та джерел за темою дипломного проекту.		
2	Розробка та затвердження плану дипломного проекту.		
3	Проведення консультації з науковим керівником щодо створення першого розділу.		
4	Аналітичний огляд і постановка задачі.		
5	Порівняльний аналіз існуючих програмних реалізацій		
6	Розробка програмного модуля системи «Task manager»		
7	Висновки та оформлення пояснювальної записки дипломного проекту.		
8	Підписання необхідних документів у встановленому порядку.		
9	Підготовка до захисту та попередній захист дипломного проекту на випусковій кафедрі дипломного проекту		

7. Дата видачі завдання: 06.05.2020 р.

Керівник дипломної роботи: ____________________ Зіатдінов Ю. К.

(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання: _________________ Загурська Ю.Г..

(підпис випускника) (П.І.Б.)

Реферат

Пояснювальна записка до дипломного проекту «Система “Task-manager” для здійснення управління IT-проектами»: 101 с., 27 рис., 9 літературних джерел.

Ключові слова: JAVA, BACKEND, MVC, ANGULAR, REST,АРХІТЕКТУРНІ ШАБЛОНИ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Об’єкт дослідження: сучасні парадигми проектування, методи, технології створення клієнт-серверних додатків на базі *SOA* та *MVC* – патерну проектування.

Предмет дослідження: веб-додаток для управління *IT* – проектами.

Мета роботи: розробка системи планування задач – інструменту для організації діяльності на підприємстві та здійснення управління *IT*- проектами

Метод дослідження : дослідження існуючих концепцій проектування модульної архітектури в рамках сервісно-орієнтованого підходу.

Отримані результати та їх новизна: створено веб-додаток на базі тришарової клієнт-серверної архітектури із незалежними серверною та клієнтською частинами. У системі впроваджено можливість авторизації користувача та функціонал для запису та менеджменту проектної інформації.

Матеріали проекту можуть бути використані в розробках спеціалізованого програмного забезпечення та для управління проектною діяльністю на підприємстві у сфері IT – технологій.

Зміст

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 7

ВСТУП 8

РОЗДІЛ 1 СУЧАСНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ВЕБ-ДОДАТКІВ 9

1.1. Архітектура побудови веб-додатків 9

1.1.1. Клієнт-серверна архітектура 9

1.1.2. Основні компоненти веб-застосунків 11

1.1.3. Дворівнева архітектура 13

1.1.4. Трирівнева архітектура 14

1.1.5. REST – архітектура 14

1.2. Використання шаблонів проектування 18

1.2.1. Сервісно-орієнтована архітектура 18

1.2.2. Патерни проектування бекенду 22

1.2.3. Патерни проектування фронтенд 27

1.3. Інструменти проектування 30

1.3.1. Фреймворки як невід’ємна складова 30

1.3.2. Spring Boot 31

1.3.3. Angular 32

1.3.4. Docker 36

Висновки до розділу 1 39

РОЗДІЛ 2 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ ВЕБ – ДОДАТКУ 40

2.1. Інструменти backend – проектування 40

2.1.1. Мова як основний інструмент проектування 40

2.1.2. Принципи SOLID – проектування 42

2.1.3. Налаштування середовища Spring-boot 45

2.2. Організація роботи компонентів у Docker 47
2.2.1. Налаштування середовища Docker 48
2.2.2. Міграції баз даних 49
2.3. Архітектура модулів системи 52
2.3.1. Структура папки серверної частини проекту 53
2.3.2. Функції компонент ядра системи 55
2.3.3. Схема взаємодії компонентів на backend 58
2.4. Розробка модуля переміщення карт на екрані користувача 59
2.4.1. Постановка проблеми 59
2.5. Розробка модуля логіну 64
2.5.1. Архітектура модуля авторизації та автентифікації 64
2.5.2. Реалізація модуля авторизації 65
Висновки до розділу 2 75
РОЗДІЛ 3 РОЗРОБКА КЛІЄНТСЬКОЇ ЧАСТИНИ ВЕБ-ДОДАТКУ 76
3.1. Робота з середовищем фреймворку Angular 76
3.1.1. Налаштування середовища 76
3.2. Схема реалізації патерну MVC 78
3.2.1. Архітектура взаємодії клієнта та сервера 82
3.3. Розробка дизайну інтерфейсу 83
Висновки до розділу 3 89
ВИСНОВКИ 90
СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ. 91
ДОДАТОК А 92
ДОДАТОК Б 94
ДОДАТОК В 95

ДОДАТОК Г **97**

ДОДАТОК Ґ **100**

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

MVC — Model-View-Controller

SOA — Service-oriented architecture – Сервісно-орієнтована архітектура

DI — Dependency injection – впровадження залежностей

SOLID — Абревіатура п’ти базових принципів об’єктно орієнтованого програмування

REST — Representational State Transfer – передача репрезентативного стану

API — Application Programming Interface – прикладний програмний інтерфейс

ВСТУП

Показники та властивості програмного та апаратного забезпечення починаючи від часу створення першої електронної обчислювальної машини рухали колесо прогресу як до покращення якості та швидкодії у роботі із обчислювальними машинами – комп’ютерами, так і стосовно розвитку та швидкості передачі інформації.

Еволюція апаратного забезпечення створила нові виклики для інженерів у сфері розробки програмного забезпечення в усьому світі. Від першого файлу у гіпертекстовому форматі, відображеного у вікні терміналу *IBM* 2250 у шістдесятих роках минулого століття та протягом останніх тридцяти п’яти років ідея створенням глобальної мережі, інформаційного простору із можливістю безперервного обміну інформацією, а разом з цим і створення лаконічних, user-friendly та pixel-perfect – інтерфейсів веб-сторінок фактично розвернула нову еру спеціальностей та фахової діяльності.

Гарний веб-сайт із динамічним скролом сторінок, parallax–ефект, зручна форма зворотного зв’язку, декілька кліків і користувач миттєво знайшов необхідну інформацію, однак лише кваліфікований спеціаліст може дати визначення тим надскладним процесам, які відбуваються під капотом таких веб-сайтів.

Роки, десятки років розробок глобальної мережі та її безпосереднього вмісту у вигляді веб-ресурсів, мільйони годин, сотні тисяч спеціалістів та ентузіастів фактично дали життя таким сучасним технологіям у сфері розробки як фреймворки, бекенд, фронтенд та інші

Однак із розвитком технологій набув актуальності розвиток ефективної діяльності із високим коефіцієнтом корисної дії працівників у сфері розробки IT-продуктів.

Сфера розробки програмного забезпечення породила також і нові напрямки у сфері управлінської діяльності, а саме створення методологій керування та масштабування проектів. Нові виклики, що стояли та досі постають перед менеджерами топових компаній, таких як Oracle, Apple та будь-яких малих бізнесів

було ефективне управління діяльністю співробітників, менеджмент великих обсягів задач із необхідністю покроково документувати здійснені фікси, а також фіксувати взаємодію розробок підприємств у співставленні із новими версіями програм.

Після часткового впровадження технологій штучного інтелекту у різних сферах розробки можна сміливо сказати, що технології поступово досягають сингулярності . Однак чи досягнуло такої ж синхронності та такої ж технологічності людство у сфері управління кадрами? Якщо так то яким чином та завдяки яким інструментам?

Будь-яка методологія передбачає використання інструментів фіксування досягнень та прогресу підприємства, планування та управління роботою команди, відслідковування поточних задач планування та автоматизацію звітності.

Будь-якому проекту, стартапу чи навіть кав’ярні потрібна система для контролю ефективної діяльності на виробництві. Такою системою і є система менеджменту задач “Task-manager”

Якраз такими інструментами являються системи менеджменту задач – task-manager системи, що відносно потреб користувача являють собою десктопне або хмарне програмне забезпечення із зручним користувацьким інтерфейсом.

Таким чином набуває актуальності створення системи менеджменту задач, що матиме зручний користувацький інтерфейс та незалежне ядро програми.

РОЗДІЛ 1
СУЧАСНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ВЕБ-ДОДАТКІВ

1.1. Архітектура побудови веб-додатків

1.1.1. Клієнт-серверна архітектура

З розвитком програмних засобів проектування та безпосередньої розробки як веб-додатків так і будь-яких інших ресурсів, що передбачають наявність користувацького інтерфейсу для взаємодії із користувачем, розвивались та вдосконалювались і підходи до розробки систем як таких.

Веб-додаток – це інтерактивний ресурс, що надає користувачу можливість вводити, отримувати та керувати даними та таким чином вирішувати чималу кількість функцій та задач користувача в рамках конкретного ресурсу.

На відміну від звичайних веб-сайтів, що основним чином призначені для представлення тематичного контенту та за доцільністю являють собою більш інформаційний характер, веб-сервіси є більш ресурсномісткими майданчиками.

Перш за все веб-сервіс – це клієнт-серверний додаток, основна частина функціоналу котрого міститься на віддаленому сервері та власне запускається на сервері, а візуальна частина – веб-інтерфейс запускається та відображається у браузері (рис.1).

Створення веб-ресурсу передбачає розробку та використання наступних компонент:

- Розробка архітектури веб-додатку;
- Протоколи зв'язку;
- Сховища даних.

Всі сучасні веб-застосунки являються розподіленими системами.

Розподілена система – це система, компоненти якої розподілені між довільними вузлами та відносно своїх можливостей виконують ті чи інші функції.

Таким чином структура веб-додатків ділиться на дві частини – це клієнтська та серверна. Клієнт виконує запит та отримує відповідь, яка формується серверною частиною. Натомість серверна частина займається обробкою запиту, формуванням відповіді та відправкою відповіді клієнту. Таким чином для клієнта веб-застосунку формується відповідь у вигляді відповідної графічної форми *(HTML, XML)*

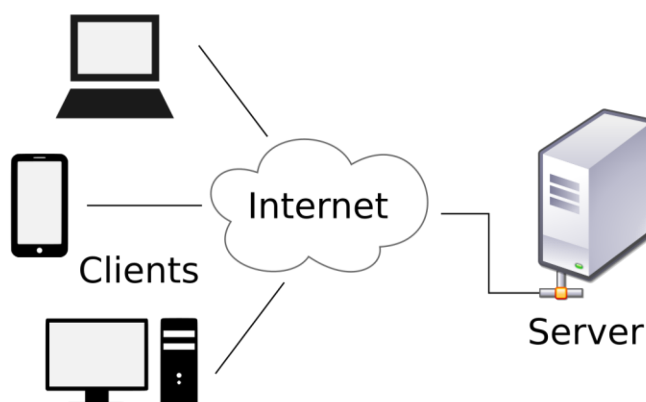


Рис. 1.1. Клієнт-серверна взаємодія

При детальному розгляді моделі клієнт-сервер, можна виділити що в цій моделі беруть участь два окремих процеси, тобто запущені програми, – одна відпрацьовує на клієнтській машині та інша на серверній машині.

Клієнтська частина, або просто клієнт - це «обличчя» додатка, тобто те, що бачить користувач, а серверна частина слугує центральним вузлом обчислень – мозком, який зберігає, обчислює та видає чималі обсяги даних.

За більшості умов один сервер може обробляти велику кількість (сотні чи тисячі) клієнтів одночасно.

Комунікація має форму клієнтського процесу, який надсилає повідомлення по мережі на серверний процес. Потім клієнтський процес чекає відповіді. Коли серверний процес отримує запит, він виконує запитувану роботу або шукає запитувані дані та повертає відповідь.

Така архітектура дозволяє розділити зони відповідальності між двома підсистемами і зробити їх більш незалежними.

Немає необхідності переписувати серверну частину під різні реалізації клієнта і, навпаки, при зміні внутрішньої логіки сервера - всі клієнти можуть продовжувати його використовувати, поки дотримується встановлений API.

Таким чином використання клієнт-серверної архітектури в рамках роботи веб-додатку передбачає використання в якості клієнта браузера, що завдяки протоколу прикладного рівня *HTTP* надсилає серверу запити *(HTTP-requst)* та отримує відповіді (*HTTP-response) (рис.2).*

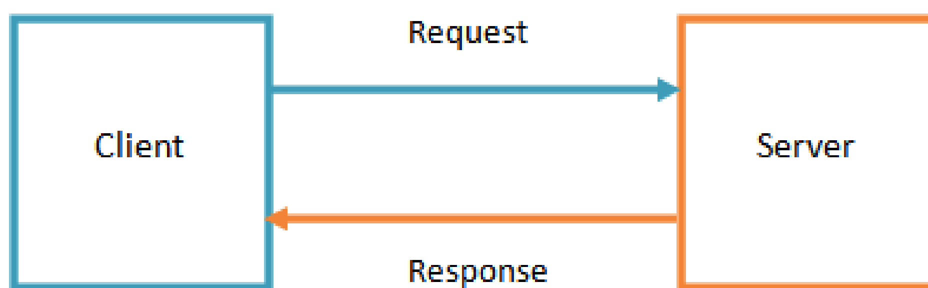


Рис. 1.2. Графічне зображення запиту та відповіді від сервера

1.1.2. Основні компоненти веб-застосунків

Відносно наявних характеристик, прийнято виділяти наступні типи веб-клієнтів[1] :

– За використання протоколу передачі даних засобами протоколів HTTP,WAP;
– За типом додатку який використовується – веб-браузер або будь-яка клієнтська програма;
– За типом рендерингу даних;
– За обробкою логіки застосунків.

Особливе значення варто приділити підходу до рендерингу даних. Прийнято розрізняти рендеринг на стороні клієнта та рендеринг на стороні сервера, також дані характеристика є широко відомі під формулюванням «товстий клієнт – тонкий сервер» та «тонкий клієнт – товстий сервер» відповідно.

Клієнт-серверна архітектура не є строго типізованою структурою, отже ролі клієнта та сервера можуть бути розподіленими відносно обраної концепції того чи іншого додатку, який розробляється, стеку технологій, мов програмування та фреймворків які прийнято використовувати для розробки.

Таким чином набуває актуальності питання розподілення ролей між компонентами веб-застосунку.

Підхід «товстий клієнт – тонкий сервер» має на меті розподілити рендеринг даних та обробку логіки того чи іншого додатку із їх реалізацією у більшій мірі на стороні клієнта. Засоби сучасних інтерпретованих мов програмування, а саме таких як JavaScript та похідних від них фреймворків здатні переносити виконання більшої частини бізнес-логіки додатків безпосередньо у браузер.

При рендерингу на стороні клієнта JavaScript, що відпрацьовує у веб-браузері користувача, відповідає за запит даних з сервера і взаємодією із веб-сторінкою. Наприклад, якщо користувач вводить в форму неприпустиме значення, буде достатньо коду на стороні клієнта, що оновить сторінку з повідомленням про помилку, без необхідності створювати нові запити на сервер і, таким чином, відпрацьовує без перезавантаження сторінки.

Натомість архітектура «тонкий клієнт – товстий сервер» являє собою класичний підхід у розробці веб-сервісів. Наявність «товстого серверу» передбачає реалізацію логіки веб-додатку, обробки даних та інших операції безпосередньо на стороні сервера.

При рендерингу саме на стороні сервера клієнт створює запит безпосередньо на сервер для кожної веб-сторінки чи після взаємодії користувача із наявними на ній компонентами. Таким чином при введенні невірних даних у форму на веб-ресурсі, код на стороні клієнта запросить нову сторінку з сервера.

1.1.3. Дворівнева архітектура

Поняття дворівневої архітектури є уособленням підходу «товстий клієнт-тонкий сервер». Будь-яка інформаційна система фактично повинна мати як мінімум три базові функціональні частини, а саме :

- Модулі зберігання даних;
- Обробка модулів даних;
- Інтерфейс представлення даних користувачу.

Кожна з перелічених функціональних частин має бути реалізована незалежно одна від одної. Не змінюючи середовища для зберігання і обробки даних система в комплексі має мати можливість видозмінювати користувацький інтерфейс та зберігаючи відображення таблиць конкретних даних у тому ж самому вигляді.

Таким чином зміна користувацького інтерфейсу, програми представлення даних чи зміна платформи для зберігання даних у вигляді сторонньої файлової системи не повинні втручатись чи змінювати дані як такі.

Як і кожна система чи методологія проектування, двошарова архітектура має власні як недоліки так і переваги.

Простота та мінімалізм архітектури фактично забезпечують розробника та користувача усім необхідним функціоналом. Розподіл даних на окремих носіях та можливість одночасної обробки даних є своєрідним сертифікатом безпеки та гарантією цілісності та непошкодженості даних. Однак необхідність визначення ролі поведінки проміжного вузла між сервером та клієнтом під назвою «обробки даних» як такої, недостатня потужність у підтримці бізнес-логіки додатку та необхідність оновлення клієнтської сторони часто ускладнює та перевантажує роботу сервісу до якого така архітектура застосовна.

1.1.4. Трирівнева архітектура

Поняття трирівневої архітектури є уособленням підходу «тонкий клієнт – товстий сервер».

У трирівневій архітектурі "тонкий" клієнт не перевантажений функціями обробки даних та виконує свою основну роль системи подання інформації, що надходить з сервера додатків. Такий інтерфейс можна реалізувати за допомогою стандартного стеку веб-технології - браузера, CGI і Java.

Таких підхід зменшує обсяг даних, переданих між клієнтом і сервером додатків, що дозволяє підключати клієнтські комп'ютери навіть по повільним лініях типу телефонних каналів. Крім того, клієнтська частина може бути настільки простою, що в більшості випадків її реалізують за допомогою універсального браузера.

Трирівнева архітектура клієнт-сервер дозволяє більш точно визначити ролі користувачів, так як вони отримують права доступу не до самої бази даних, а до певних функцій сервера додатків. Це підвищує захищеність системи (у порівнянні з звичайно архітектурою) не тільки від навмисного нападу, а й від помилкових дій персоналу.

1.1.5. REST – архітектура

REST - це абревіатура від Representational State Transfer. Це архітектурний стиль для розподілених гіпермедіа-систем і вперше був представлений Роєм Філдінгом у 2000 році у своїй знаменитій дисертації.

Як і будь-який інший архітектурний стиль, REST також має свої 6 керівних обмежень, які повинні бути задоволені, якщо інтерфейс потрібно називати RESTful [2].

Керівні принципи REST:

– Клієнт-сервер - відокремлюючи проблеми користувальницького інтерфейсу від проблем зберігання даних, ми покращуємо портативність користувальницького інтерфейсу на кількох платформах та покращуємо масштабованість, спрощуючи серверні компоненти.
– Без стану - кожен запит від клієнта до сервера повинен містити всю інформацію, необхідну для розуміння запиту, і не може скористатися будь-яким збереженим контекстом на сервері. Тому стан сесії повністю зберігається на клієнті.
– Кешованість - обмеження кешу вимагають, щоб дані у відповіді на запит неявно або явно позначали як кешовані або не кешовані. Якщо відповідь є кешованою, то клієнтський кеш має право повторно використовувати ці дані відповіді для наступних, рівнозначних запитів.
– Уніфікований інтерфейс - За допомогою застосування принципу загальної інженерії програмного забезпечення до компонентного інтерфейсу спрощується загальна архітектура системи та покращується видимість взаємодій. Щоб отримати єдиний інтерфейс, для керування поведінкою компонентів потрібно кілька архітектурних обмежень. REST визначається чотирма обмеженнями інтерфейсу: ідентифікація ресурсів; маніпулювання ресурсами через представництва; повідомлення з самоописанням; і, гіпермедіа як двигун стану застосування.

– Багатошарова система - шаровий стиль системи дозволяє архітектурі складатися з ієрархічних шарів, обмежуючи поведінку компонентів таким чином, що кожен компонент не може "бачити" за межами прямого шару, з яким вони взаємодіють.

Код на вимогу (необов'язково) - REST дозволяє розширити функціональність клієнта, завантаживши та виконавши код у вигляді аплетів чи сценаріїв. Це спрощує клієнтів, зменшуючи кількість функцій, необхідних для попередньої реалізації.

Ще одна важлива річ, пов'язана з REST, - це ресурсні методи, які використовуються для виконання бажаного переходу. Велика кількість людей неправильно ставляться до ресурсних методів до методів HTTP GET / PUT / POST / DELETE.

Рой Філдінг ніколи не згадував жодної рекомендації щодо того, який метод слід використовувати в якому стані. Все, що він наголошує, - це те, що це повинен бути єдиний інтерфейс. Якщо ви вирішите, що HTTP POST буде використовуватися для оновлення ресурсу - а не більшість людей рекомендує HTTP PUT - це добре, і інтерфейс програми буде RESTful.

В ідеалі все, що потрібно для зміни стану ресурсу має бути частиною відповіді API для цього ресурсу - включаючи методи та в якому стані вони залишать представлення.

Синхронна взаємодія – це стиль спілкування, коли один сервіс чекає, поки з'явиться відповідь від іншого. Його концептуальна простота дозволяє просту реалізацію, що робить її добре придатною для більшості ситуацій.

Синхронний зв'язок тісно пов'язаний з протоколом HTTP. Однак інші протоколи залишаються не менш розумним способом реалізації синхронного зв'язку.

Як механізм взаємодії за допомогою синхронних запитів і відповідей зазвичай використовуються HTTP і REST, особливо якщо спілкування служби ведеться за межами вузла Docker або кластера.

1.2. Використання шаблонів проектування

1.2.1. Сервісно-орієнтована архітектура

Оскільки архітектура проектування серверної частини відрізняється від проектування архітектури клієнтської частини, варто відзначити що до них можуть бути застосовні як однакові шаблони проектування так і шаблони, які суттєво відрізняються за принципом вирішуваних задач.

Сервісно-орієнтована архітектура (*SOA*) - це архітектурний підхід, при якому додатки використовують сервіси, доступні в мережі. У цій архітектурі надаються послуги для формування додатків за допомогою комунікаційного дзвінка через Інтернет.

Сервісно – орієнтована архітектура також відома як архітектура із модульним підходом до розробки програмного забезпечення, який заснований на використанні розрізнених, слабо зв’язаних замінних компонентів, оснащених стандартизованими інтерфейсами для взаємодії з звичних протоколам.

Сервіс – це програмний компонент, що реалізує завершену функцію подання або обробки даних. Основним відмінністю сервісу від звичайного компонента є стандартний та платформо-незалежний інтерфейс. Клієнт, що звертається до сервісу, не зобов’язаний нічого більше знати про деталі реалізованої служби.

Сервісно-орієнтована архітектура дозволяє компонувати бізнес-процеси з компонентів, виконуючись на різних платформах, представляти їх у вигляді служб та повторно використовувати в нових бізнес-процесах.

Принципи сервісно-орієнтованої архітектури:

– відокремлення бізнес-логіки прикладної системи від логіки презентації інформації;
– реалізація бізнес-логіки прикладної системи у вигляді деякої кількості програмних модулів)сервісів), які доступні користувачам та іншим модулям ззовні;

– "Користувач послуг", котрий може бути прикладною системою або іншим сервісом, має можливість випустити сервіс через інтерфейси, використовуючи відповідні комунікаційні механізми.

SOA дозволяє користувачам поєднувати велику кількість об'єктів із існуючих сервісів для формування додатків.

SOA охоплює набір принципів проектування, які структурують розвиток системи та забезпечують засоби для інтеграції компонентів у цілісну та децентралізовану систему.

Функціонування обчислювальних пакетів на основі *SOA* складається в набір сумісних сервісів, які можна інтегрувати в різні програмні системи, що належать до окремих бізнес-доменів.

У сервісно-орієнтованій архітектурі є дві основні ролі:

– Постачальник послуг;
– Споживач служби.

Постачальник послуг є обслуговуючим сервісом і організацією, яка надає одну чи кілька служб для використання іншими. Щоб рекламувати послуги, постачальник може опублікувати їх у реєстрі разом із договором на надання послуг, який визначає характер послуги, способи її використання, вимоги до послуги та стягувані плати.

До прикладу споживач служби може знайти метадані служби в реєстрі та розробити необхідні клієнтські компоненти для прив’язки та використання послуги.

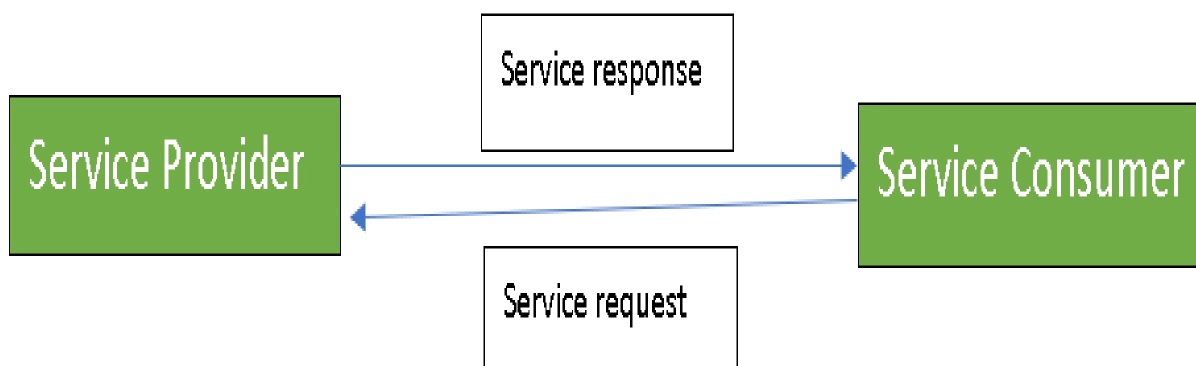


Рис.

Послуги можуть агрегувати інформацію та дані, отримані з інших служб, або створювати робочі процеси для задоволення запиту певного споживача послуги. Ця практика відома як службова оркестрація. Ще однією важливою схемою взаємодії є сервісна хореографія, яка є скоординованою взаємодією служб без єдиної точки контролю.

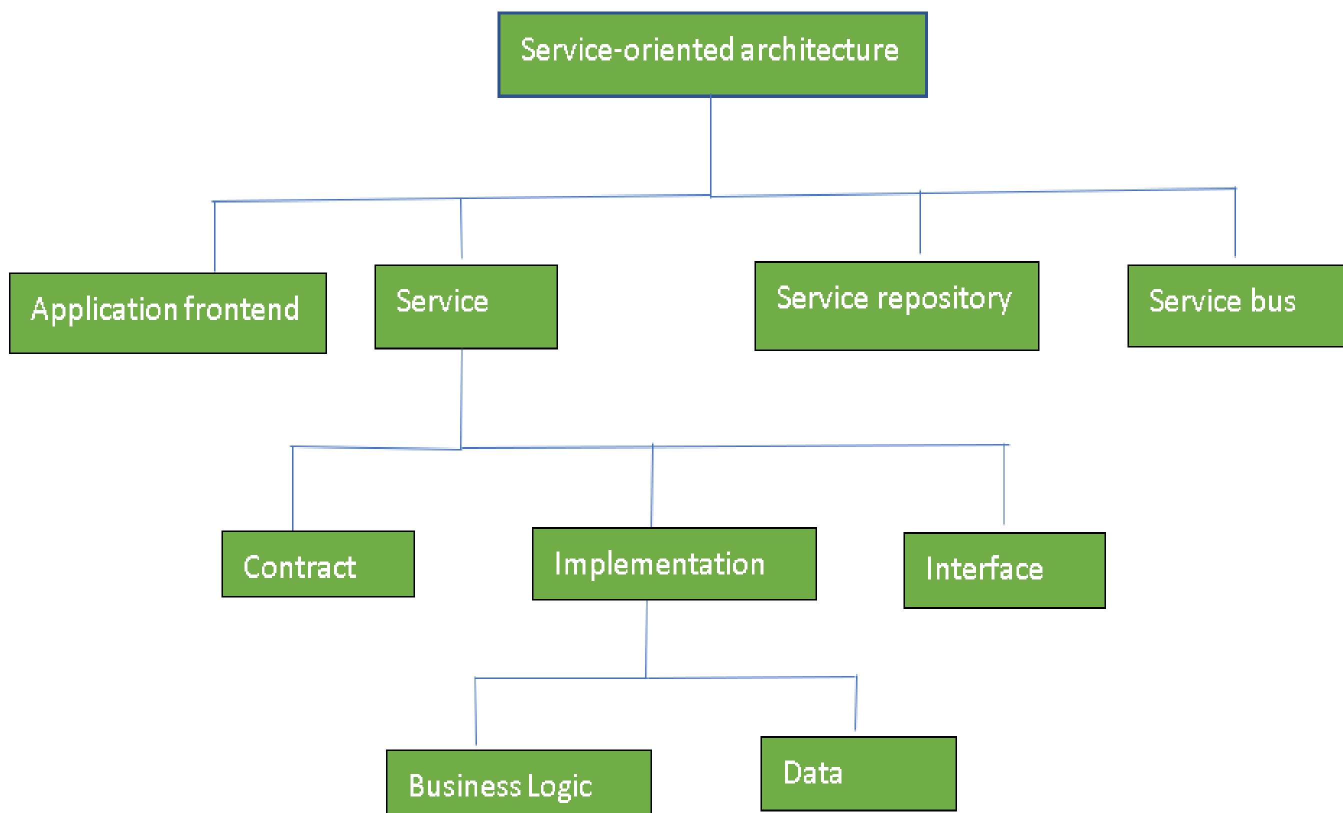


Рис.

Переваги сервісно – орієнтованої архітектури:

- Повторне використання сервісу. У *SOA* додатки створюється з існуючих сервісів. Отже, сервіси можна повторно використовувати для створення багатьох програм.
- Просте обслуговування. Оскільки послуги незалежні одна від одної, їх можна легко оновлювати та змінювати, не впливаючи на інші послуги.
- Незалежність від платформи. *SOA* дозволяє робити складне додаток, комбінуючи послуги, вибрані з різних джерел, незалежно від платформи.

– Доступність. Засоби *SOA* легко доступні для кожного за запитом.
– Надійність. Програми *SOA* надійніші, оскільки легко налагоджувати невеликі сервіси, а не величезні коди
– Масштабованість. Сервіси можуть працювати на різних серверах в межах певного середовища, що збільшує масштабованість

Недоліки сервісно – орієнтованої архітектури:

– Високі накладні витрати. Валідація вхідних параметрів послуг проводиться щоразу, коли служби взаємодіють, це знижує продуктивність, оскільки збільшує навантаження та час відповіді.
– Високі інвестиції. Для *SOA* потрібні величезні початкові інвестиції.
– Комплексне управління послугами. Коли служби взаємодіють, вони обмінюються повідомленнями із завданнями. кількість повідомлень може перейти в мільйони. Це стає громіздким завданням обробляти велику кількість повідомлень.

Принципам SOA відповідають такі патерни проектування як мікросервіси та модульність.

1.2.2. Патерни проектування бекенду

Розрізняють наступні архітектури :

- Монолітна архітектура;
- Мікросервіси;
- Модульна архітектура.

Застосування архітектури моноліт описує так зване одноярусне програмне забезпечення, в якому різні компоненти об'єднуються в єдину програму з однієї платформи.

Архітектура мікросервісів та модульна архітектура мають одну спільну рису. Дані архітектури працюють на досягнення спільної мети – зменшення якомога меншої зв’язності компонентів тієї чи іншої програми. Така поведінка в умах виходу з ладу одного із діючих сервісів певної програми здатна частково або повністю гарантувати безпеку даних [3].

Основна відмінність натомість полягає у розподіленні ролей відносно апаратного забезпечення. Оскільки архітектури являють собою окремі архітектурні одиниці, що надають користувачам сервіси, що використовуються через інтерфейси (у даному контексті і REST також), робота модулів розподіляється відносно однієї одиниці апаратного забезпечення, натомість робота мікросервісів є більш розподіленою та спроектована таким чином, щоб займати декілька сервісів.

Таким чином неозброєним оком стає помітно, що менша зв’язність компонент на багато легше спрощує масштабування архітектури.

Оскільки сервісно-орієнтована архітектура по суті є сукупністю служб, кожна з них має можливість «спілкуватись» одна з одною. Комунікація може включати в себе просту передачу даних двох або більше служб, що координують певну діяльність. Потрібні певні засоби підключення служб один до одного.

Мікросервіси – архітектура мікросервісів – це архітектурний стиль, який структурує додаток як сукупність невеликих автономних служб, модельованих навколо бізнес-домену, що володіють наступними характеристиками:

- висока спроможність та тестованість;

– Мала зв’язність;

– Незалежне розгортання;

– Можливість швидко, часто та надійно доставляти великі, складні програми

Архітектура мікросервісів є наступною:

– Хмарна інфраструктура

– Потенціал масштабованості. Кожна служба може масштабуватися окремо, не потрібно масштабувати весь додаток по горизонталі

– Самостійне розгортання. Послуги набагато легші, що полегшує їх розгортання індивідуально

– Самостійний розвиток. Різні розробники і навіть команди можуть відповідати за свою область

– Межі послуг. Послуги мають чіткі межі, що робить їх більш надійними та доступними

– Різноманітність. Послуги можуть бути побудовані за допомогою різних підходів, мов програмування, каналів зв'язку тощо.

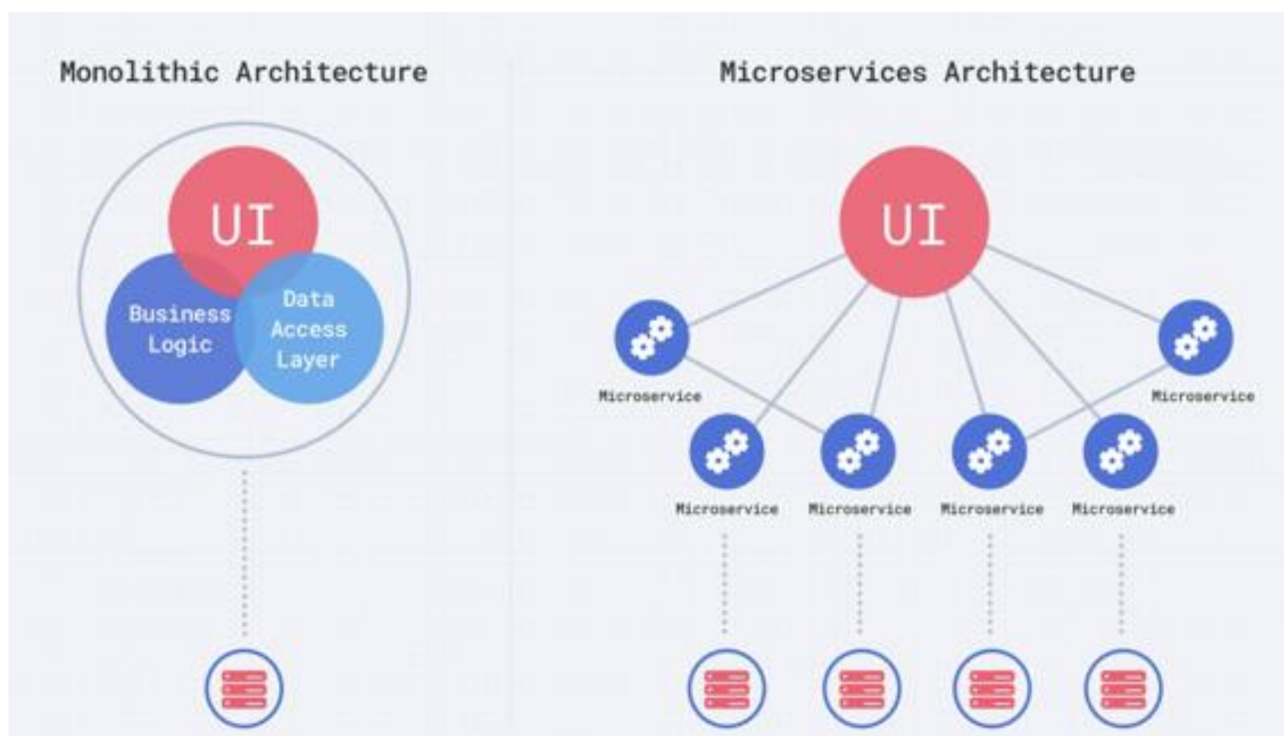


Рис.

Компоненти сервісу в архітектурі мікросервісів, як правило, одноцільові сервіси. Завдяки SOA, сервісні компоненти можуть варіюватися за розмірами від будь-яких маленьких прикладних служб до дуже великих корпоративних послуг.

Монолітний додаток повністю замкнутий в контексті поведінки. Під час роботи додаток може взаємодіяти з іншими службами або сховищами даних, проте уся поведінка реалізується у власному, одиничному процесі, а додаток зазвичай розгортається як один елемент. Для горизонтального масштабування такий додаток зазвичай цілком дублюється на декількох серверах або віртуальних машинах.

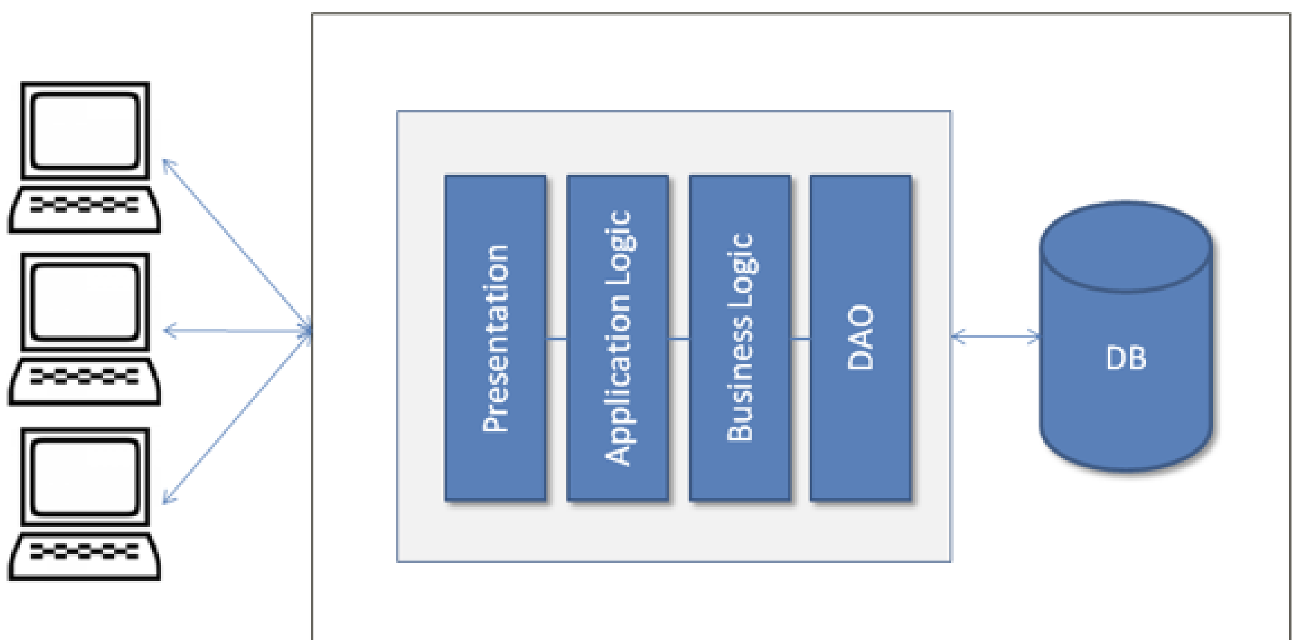


Рис. Моноліт

Термін «моноліт» означає – «усе складене в одне ціле», у якому різні компоненти об'єднуються в єдину програму з однієї платформи.

Переваги:

– Простота в розробці. На початку проекту набагато простіше перейти на монолітну архітектуру.

- Простий для тестування. Наприклад, ви можете реалізувати тестування, щоб просто запустити додаток і протестувати інтерфейс користувача з Selenium.
- Простий у розгортанні. Ви повинні скопіювати упаковану програму на сервер.
- Легко масштабувати горизонтально, виконавши кілька копій за балансиром навантаження.

Недоліки:

- Технічне обслуговування. Якщо додаток занадто великий і складний, щоб його зрозуміти цілком, складно вносити зміни швидко і правильно.
- Розмір програми може сповільнити час запуску.
- Ви повинні повторно розмістити всю програму під час кожного оновлення.
- Монолітні програми також можуть бути складними для масштабування, коли різні модулі мають суперечливі вимоги до ресурсів.
- Надійність - помилка в будь-якому модулі (наприклад, витік пам'яті) може потенційно знизити весь процес. Більше того, оскільки всі екземпляри програми однакові, ця помилка впливає на доступність всієї програми
- Незалежно від того, наскільки легкими можуть здатися початкові етапи, у монолітних додатках виникають труднощі із застосуванням нових та прогресивних технологій. Оскільки зміни мов чи рамок впливають на всю програму, вона вимагає зусиль для ретельної роботи з деталями програми, отже, це дорого враховує як час, так і зусилля.

Таким чином обидва підходи мають свої плюси і мінуси, але це залежить від кожного сценарію або вимог до продукту чи проекту та вибору, який ви обираєте. Оскільки монолітний підхід найкраще підходить для легких застосувань,

рекомендується спочатку застосовувати монолітний підхід і залежно від потреб та вимог поступово переходити до підходу мікросервісів.

1.2.3. Патерни проектування фронтенд

Модель Controller View або MVC, як прийнято називати, - це модель дизайну програмного забезпечення для розробки веб-додатків. Шаблон контролера подання моделі складається з наступних трьох частин –

- Модель - це найнижчий рівень структури, що відповідає за збереження даних.
- Перегляд. Він несе відповідальність за показ усіх або частини даних користувачеві.

- Контролер - це програмний код, який контролює взаємодію між Модель та Переглядом.

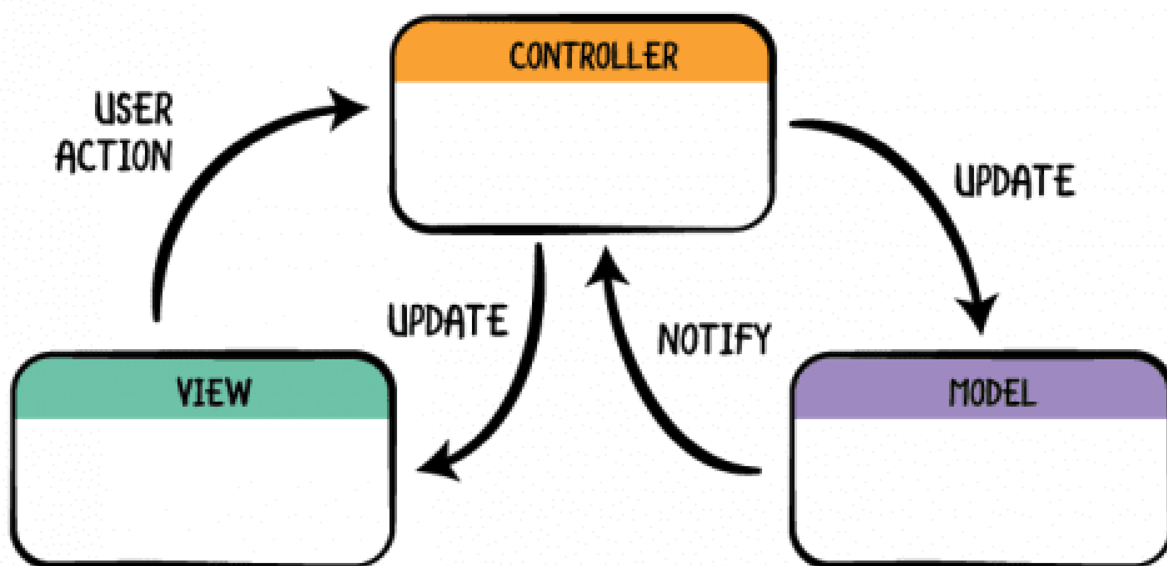


Рис.

MVC популярний тим, що він ізолює логіку програми від рівня інтерфейсу користувача та підтримує розділення проблем. Контролер отримує всі запити на додаток, а потім працює з моделлю, щоб підготувати будь-які дані, необхідні для перегляду. Потім представлення використовує дані, підготовлені контролером, для створення остаточної презентабельної відповіді. Абстракцію MVC можна графічно представити наступним чином[].

Модель відповідає за управління даними програми. Він відповідає на запит від перегляду та на вказівки контролера щодо оновлення.

Вид. Подання даних у певному форматі, викликане рішенням контролера представити дані. Вони являють собою шаблонні системи на основі скриптів, такі як JSP, ASP, PHP і дуже легко інтегруватися з технологією AJAX.

Контролер відповідає на введення користувача та здійснює взаємодію з об'єктами моделі даних. Контролер отримує вхід, перевіряє його, а потім виконує бізнес-операції, що змінюють стан моделі даних.

Що таке архітектура MVC на Java?

Конструкції моделей, засновані на архітектурі MVC, відповідають схемі дизайну MVC і при розробці програмного забезпечення вони відокремлюють логіку програми від користувальницького інтерфейсу. Як випливає з назви, шаблон MVC має три шари, які є:

Модель - представляє бізнес-рівень програми

Перегляд - визначає презентацію програми

Контролер - Керує потоком програми

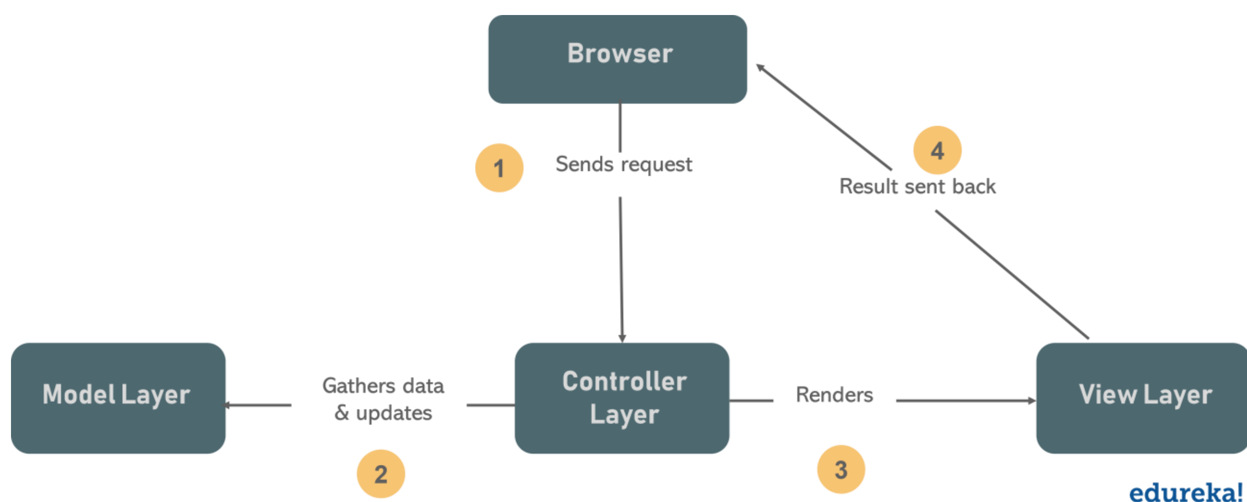


Рис.

У контексті програмування Java Модель складається з простих класів Java, у представленнях відображаються дані, а контролер складається з сервлетів. Це розділення призводить до того, що запити користувачів обробляються наступним чином:

Браузер на клієнті надсилає запит на сторінку до контролера, присутнього на сервері. Контролер виконує дію виклику моделі, тим самим, отримуючи потрібні їй дані у відповідь на запит. Потім контролер надає отримані дані для перегляду. Перегляд надається та відправляється назад клієнту для перегляду браузера

Переваги архітектури MVC на Java

MVC-архітектура пропонує безліч переваг для програміста при розробці програм, які включають:

- Кілька розробників можуть працювати з трьома шарами (Модель, Перегляд та Контролер) одночасно
- Пропонує покращену масштабованість, що доповнює здатність програми зростати
- Оскільки компоненти мають низьку залежність один від одного, їх легко підтримувати
- Модель може бути повторно використана за допомогою декількох переглядів, що забезпечує повторне використання коду
- Прийняття MVC робить додаток більш виразним і зрозумілим
- Розширення та тестування програми стає простим

1.3. Інструменти проектування

1.3.1. Фреймворки як невід'ємна складова

Фреймворк —це перш за все інфраструктура програмних рішень, що полегшує розробку складних систем.

Фреймворк – це програмний каркас, абстракція, в якій програмне забезпечення, що забезпечує загальну функціональність, може вибірково змінюватися додатковим кодом, написаним користувачем, забезпечуючи таким чином програмне забезпечення, яке стосується додатків.

Він забезпечує стандартний спосіб створення та розгортання програм і являє собою універсальне програмне середовище для багаторазового використання, яке забезпечує особливу функціональність як частину більшої програмної платформи для полегшення розробки програмних програм, продуктів та рішень. Рамки програмного забезпечення можуть включати в себе програми підтримки, компілятори, бібліотеки кодів, набори інструментів та інтерфейси прикладного програмування (API), які об'єднують всі різні компоненти, щоб забезпечити розробку проекту чи системи.

Backend або script-side, яка працює в архітектурі request-response, складається з API, баз даних, фреймворків, службових працівників та операційної системи.

У наші дні розробники мають можливість користуватись такими сервісами як Docker, Vagrant та BAAS, які надають складні системи для автоматичного розгортання сценаріїв резервного копіювання на будь-який сервер або хмару.

З огляду на важливість та кільеість сучасних бекенд-фреймворків, абсолютно природньою в даних обставинах є наявність вибору. У GitHub та подібних платформах є ряд каркасів, які додають цінності розвитку бекенду.

Архітектура роботи бекенду будь-якого динамічного додатка зображена на наступному рисунку (рис.).

Рис.

1.3.2. Spring Boot

Spring Boot - це відкритий код на базі Java, який використовується для створення мікросервісу. Він розроблений Pivotal Team і використовується для створення автономних та готових Spring – додатків.

Рис.

Spring Boot – це хороша платформа для розробників Java та, власне, для розробки автономного і виробничого класу Spring – додатка, який досить просто

запустити. Spring Boot дає можливість розпочати роботу з мінімальними конфігураціями без необхідності цілого налаштування Spring.

Spring Boot - це корисний проект, метою якого є спрощення створення додатків на основі Spring. Він дозволяє найбільш простим способом створити web-додаток, вимагаючи від розробників мінімум зусиль по його налаштуванні і написання коду[6].

Особливості та переваги:

- надає гнучкий спосіб налаштування Java Beans, конфігурацій XML та транзакцій з базами даних;
- Забезпечує потужну пакетну обробку та керує кінцевими точками REST;
- У Spring Boot все налаштовано автоматично, ніякі конфігурації вручну не потрібні;
- Він пропонує Spring – додаток на основі анотацій;
- Полегшує управління залежністю;
- Він включає вбудований контейнер сервлетів.

Spring Boot володіє великим функціоналом, але його найбільш значущими особливостями є управління залежностями, автоматична конфігурація і вбудовані контейнери сервлетів.

1.3.3. Angular

Angular - це платформа та фреймворк для побудови клієнтських додатків у HTML та TypeScript. Angular використовує TypeScript. Він реалізує основну та додаткову функціональність як набір бібліотек TypeScript, які ви імпортуєте у свої програми [4].

На рисунку зображені основні рішення, запропоновані фреймворком Angular (рис.).

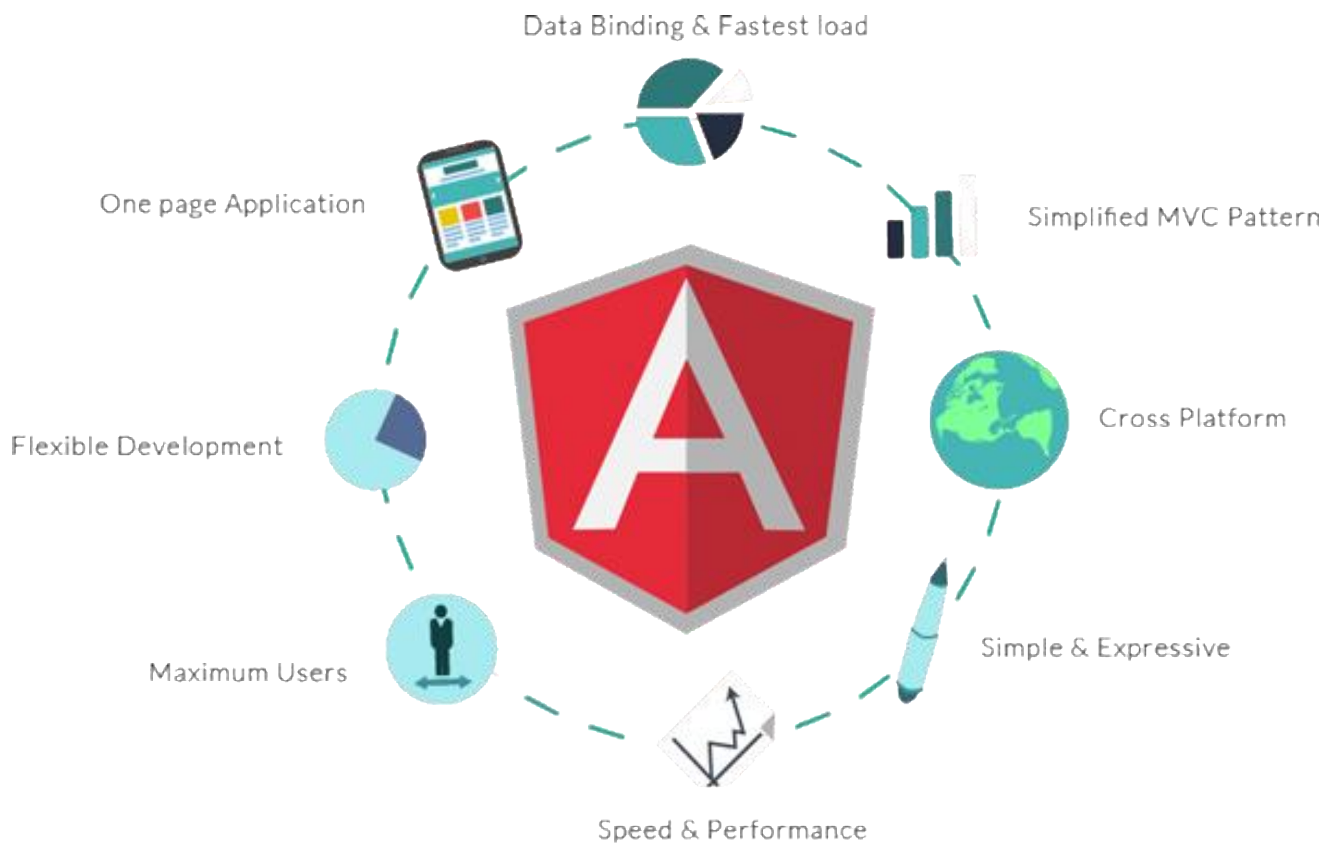


Рис.

Angular - це фреймворк для побудови клієнтських додатків у HTML та JavaScript та TypeScript, яка компілюється в JavaScript.

Angular складається з декількох бібліотек, деякі з них основні, а деякі необов'язкові. Основними складовими частин програми Angular є NgModules, які забезпечують контекст компіляції компонентів. NgModules збирають відповідний код у функціональні набори; Кутовий додаток визначається набором NgModules. У додатку завжди є принаймні кореневий модуль, який дозволяє завантажувати, і, як правило, має ще багато функціональних модулів[4].

Компоненти визначають представлення даних, що представляють собою набори елементів екрану, які можна кутовий вибирати та змінювати відповідно до логіки та даних вашої програми.

Компоненти використовують сервіси, які надають певні функціональні можливості, безпосередньо не пов'язані з переглядами. Постачальники послуг можуть бути введені в компоненти як залежності, що робить ваш код модульним, багаторазовим та ефективним.

Діаграма архітектури визначає вісім основних будівельних блоків програми Angular(рис.).

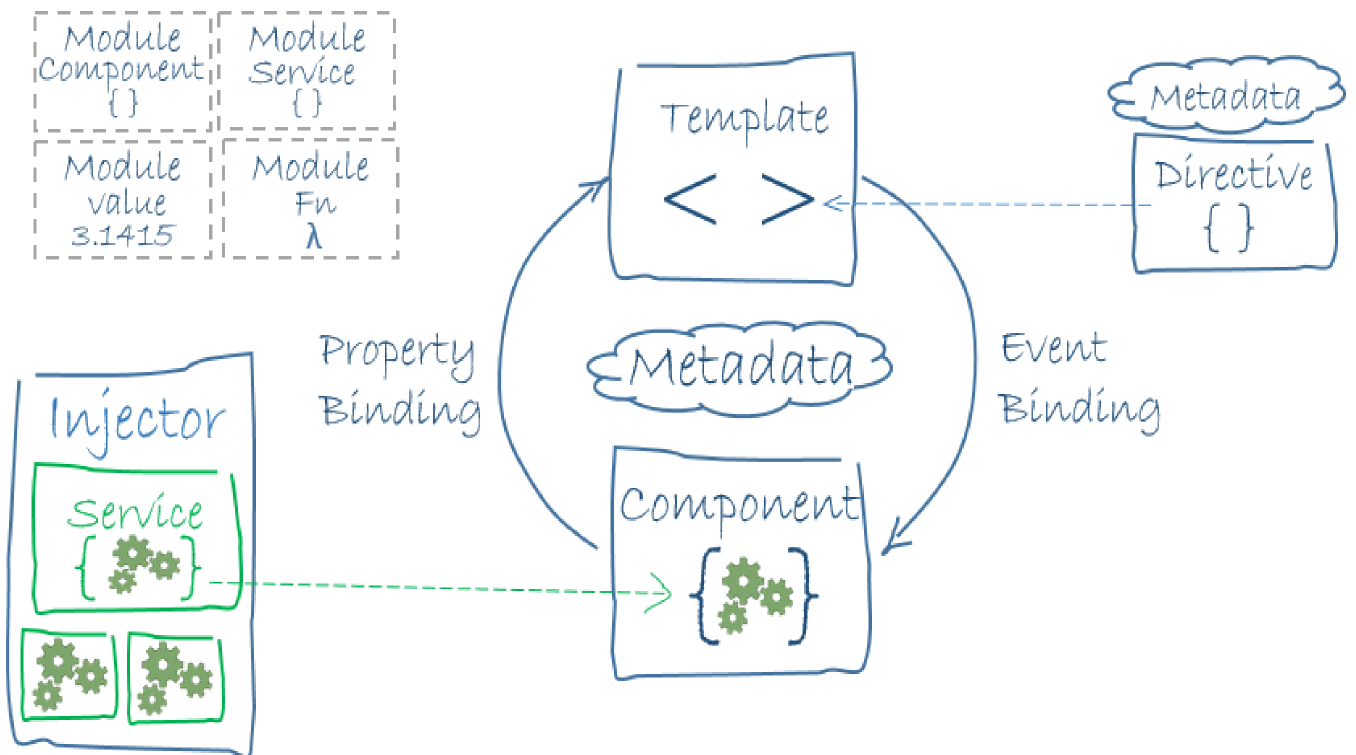


Рис.

Angular – додатки є модульними, а Angular має власну модульну систему, що називається Angular– модулі або NgModules.

І компоненти, і сервіси - це просто класи, в яких є декоратори, які позначають їх тип та надають метадані, які вказують Angular, як ними користуватися.

Метадані класу компонентів асоціюють його з шаблоном, який визначає представлення. Шаблон поєднує звичайний HTML з кутовими директивами та розміткою, що дозволяють Angular змінювати HTML, перш ніж відображати його для відображення.

Метадані для класу обслуговування надають інформацію, яку Angular потребує, щоб зробити її доступною для компонентів через введення залежності (DI).

Компоненти програми зазвичай визначають багато представлень, розташованих ієрархічно. Angular надає послугу маршрутизатора, щоб допомогти вам визначити шляхи навігації серед представлень. Маршрутизатор забезпечує складні навігаційні можливості в браузері.

1.3.4. Docker

Docker – це відкрита платформа для розробки, доставки і експлуатації додатків. Використовуючи контейнери Docker, можна розгортати, копіювати, переносити і робити резервні копії інформації швидше і легше, ніж за допомогою віртуальної машини [5].

Мережа Docker побудована на Container Network Model (CNM), яка дозволяє будь-кому створити свій власний мережевий драйвер. Таким чином, у контейнерів є доступ до різних типів мереж і вони можуть підключатися до декількох мереж одночасно .

Docker ізолює контейнери, змушуючи їх працювати як єдиний процес. Якщо оточення додатка складається з X одночасних процесів, Docker запустить X контейнерів, кожен зі своїм процесом. На відміну від Docker, LXC контейнери можуть запускати безліч процесів.

Архітектура мережевої архітектури Docker із двома ізольованими мережами контейнерів зображена на рисунку нижче (рис.)

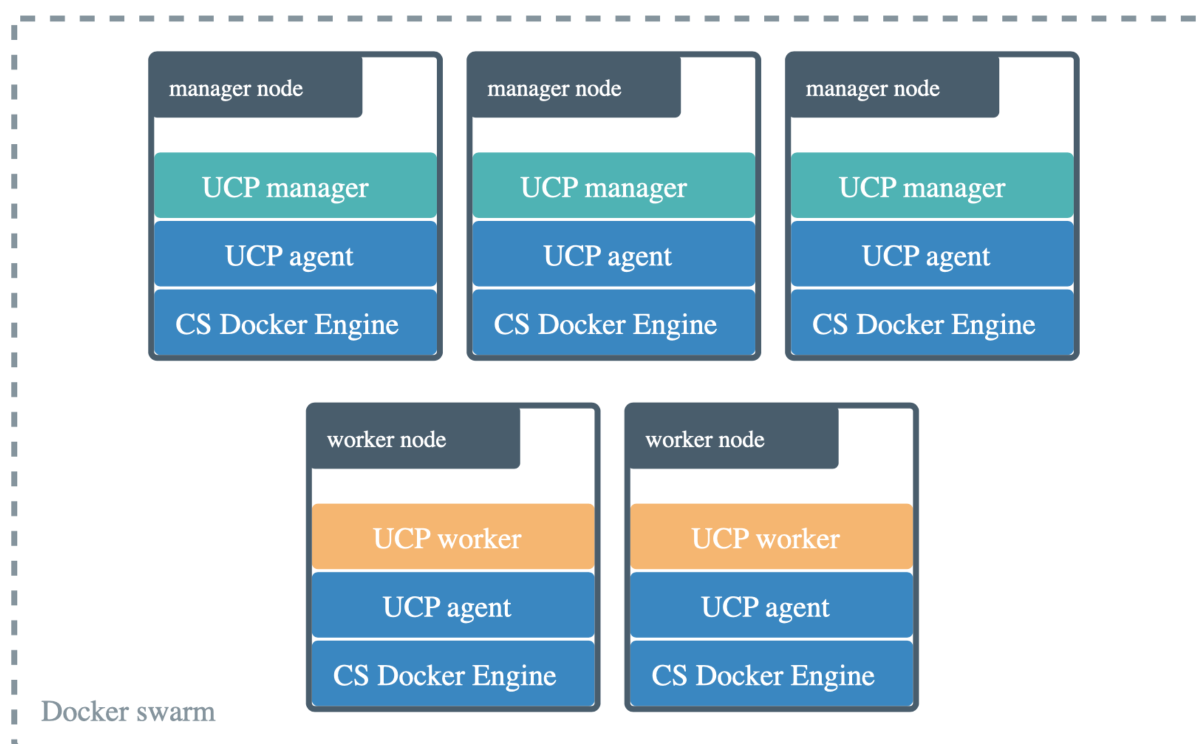


Рис.

Контейнеризація – це «легка» віртуалізація і ізоляція ресурсів, яка дозволяє запускати додаток і необхідний йому мінімум системних бібліотек в повністю стандартизованому контейнері, який взаємодіє з хостом за допомогою певних інтерфейсів, отримуючи доступ до апаратних ресурсів.

Контейнер – це екземпляр образу Docker. Контейнер відповідає за виконання програми, процесу чи служби. Він складається з вмісту образу Docker, середовища виконання та стандартного набору інструкцій. При масштабуванні додатку, можна створити декілька екземплярів контейнера з одного образу.

У контейнерах міститься все, що потрібно для роботи програми. Кожен контейнер створюється з образу. Контейнери можуть бути створені, запущені, зупинені, перенесені або видалені. Кожен контейнер ізольований і є безпечною платформою для додатка.

Контейнер складається з операційної системи, користувацьких файлів і метаданих. Образ Docker включає у себе інформацію як запустити контейнер, тобто, який процес запустити при старті контейнера та інші конфігураційні дані. Docker образ доступний тільки для читання. Коли docker запускає контейнер, він створює рівень для читання / запису зверху образу (використовуючи union file system), в якому може бути запущений додаток.

При проектуванні розподілених систем працюють з Docker-контейнерами, мережеве взаємодія стає вкрай важливими. Сервіс-орієнтована архітектура, безперечно, спирається на взаємодію між компонентами для коректного функціонування системи в цілому.

Коли Docker запускає контейнер, створюється новий віртуальний інтерфейс і йому призначається адреса в діапазоні підмережі моста. IP-адреса пов'язана із внутрішньою мережею контейнера, надаючи шлях для мережі контейнера до мосту docker0 на системі хоста. Docker автоматично створює правила в iptables для забезпечення переадресації та налаштовує NAT для трафіку з docker0 в зовнішню мережу.

Усі контейнери всередині своєї мережі можуть вільно бачити один одного і спілкуватися між собою, а зовні до них можна достукатися лише прив'язавши контейнерні порти до портів хоста, наприклад, увесь трафік хост-машини з порта 80 можна перенаправляти на будь-який доступний порт контейнера.

Висновки до розділу 1

У даному розділі було розглянуто сучасні підходи до проектування клієнт-серверних додатків різних рівнів та їх типи архітектури. Сучасні інструменти проектування, які в повному та вільному обсязі доступні для розробок як клієнтських, так і серверних систем, надають можливість створювати масштабовані та гнучкі системи для здійснення управління та організації діяльності на підприємстві.

Технології контейнеризації та потужні патерни проектування backend за рахунок розширюваних бібліотек та фреймворків здатні вповні забезпечити розробку клієнт-серверної системи по типу «тонкий клієнт – товстий сервер», що являє собою класичний підхід до побудови програмного забезпечення веб-застосунку.

РОЗДІЛ 2
РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ ВЕБ – ДОДАТКУ

2.1. Інструменти backend – проектування

2.1.1. Мова як основний інструмент проектування

Розробка клієнтської сторони , також відомої як *frontend* та серверної сторони – *backend* веб-застосунків мають фактично протилежно-направлені вектори розвитку.

Оскільки розробка системи менеджменту задач «Task manager» передбачала розробку як серверної так і клієнтської частини, невід'ємну частину розробки серверної частини складало дослідження засобів програмування об'єктно орієнтованої мови *Java* та фреймворку *Spring Boot.*

Java – це статично-типізована мова, яка надає повний набір інструментів для роботи з веб – додатками зі сторони сервера.

Розробники Java у офіційному технічному описі достоїнств мови визначили наступні :

– Проста;
– Об'єктно орієнтована;
– Розподілена;
– Надійна;
– Безпечні;
– Не залежить від архітектури комп'ютера;
– Переносима;
– Інтерпретована;
– Високопродуктивна;
– Багатопоточна;
– Динамічна.

Щоб жити у світі електронної комерції та розповсюдження, технологія Java повинна забезпечити розвиток безпечних, високопродуктивних та дуже надійних додатків на кількох платформах у різнорідних, розподілених мережах[7].

Мова Java надає розробнику велику бібліотеку програм для передачі даних через протокол TCP/IP, HTTP та FTP. Додатки на Java здатні відкривати об’єкти та отримувати до них доступ через мережу з такою ж легкістю, як і в локальній файловій системі, використовуючи URL для адресації.

Мова Java призначена для написання програм, які повинні надійно працювати за будь-яких умов. Головна увага в цій мові приділяється ранньому виявленню можливих помилок, контролю в процесі виконання програми, а також усунення ситуацій, які можуть викликати помилки. Єдина суттєва відмінність мови Java від мови C++ у моделі вказівників, прийнятної в Java, яка виключає можливість запису в довільно обрану область пам’яті та пошкодження даних.

Мова Java призначена для використання в мережевому чи розподільчому середовищі. Через це більшість уваги було приділено безпеці. Java дає можливість створювати системи, захищені від вірусів та несанкціонованого доступу[8].

В рамках автоматизації розробки та постійного її вдосконалення відносно базових парадигм програмування, а саме таких як ООП – об’єктно-орієнтоване програмування, мова Java роками міцнішала та збагачувалась новими методологіями та підходами задля досягнення результатів у програмування у найефективніший спосіб.

Java є об’єктно-орієнтованою мовою програмування, що, у свою чергу, має на меті програмування на цій мові із застосуванням об’єктно орієнтованого стилю.

Java є основою ряду платформ, призначених для розробки програмного забезпечення і забезпечуючих ефективну роботу на системах з різною чіп-архітектурою. Java допомагає вирішувати завдання розробникам серверних, клієнтських та вбудованих систем.

Одна із сильних сторін віртуальної машини Java, завжди була її здатність з легкістю жонглювати декількома потоками. JVM оптимізована для великих багатоядерних машин, і вона без проблем може керувати сотнями потоків. Завдяки

цій здатності, на JVM з'явилися і інші мови - створюються крос-компілятори і емулятори, що працюють поверх JVM. Ці магічні можливості використовуються багатьма веб-сайтами з високою відвідуваністю.

2.1.2. Принципи SOLID – проектування

Однак окрім необхідності слідувати об’єктно–орієнтованій парадигмі проектування, у сучасному світі розробки існує чимало інших критеріїв створення програмного забезпечення.

Часто при розробці якоїсь програми перед розробником стоїть лише одна мета – програма повинна працювати правильно. Однак коректність – це лише частина того, що робить програму хорошою. Інша, не менш важлива частина, полягає в тому, що програма має бути здійсненою.

У світі розробки не існує загальноприйнятого терміну, який характеризував би ту чи іншу архітектуру як взірцеву. Створення архітектури програми часто вимагає гнучкого підходу та комбінування різних методологій та практик. Однак як показує практика, критерії «хорошої» архітектури все ж таки існують.

До характеристик взірцевої архітектури можна віднести наступні:

- Гнучкість системи;
- Ефективність системи;
- Масштабованість системи;
- Тестованість системи;
- Можливість повторного використання системи;
- Супроводжуваність системи.

Дані характеристики найкращим чином описані у загальновідомому принципі в об’єктно–орієнтованому програмуванні під назвою *SOLID*.

Хороше програмне забезпечення навмисне розроблене таким чином, щоб передбачити та уникнути несподіваних взаємодій.

Об'єкт - це програмна модель об'єктів реального світу або абстрактних понять, що представляє собою сукупність змінних, які задають перебуваючи -ня об'єкта, і пов'язаних з ними методів, що визначають поведінку об'єкта.

Клас - це прототип, що описує змінні і методи, що визначають ха- рактеристики об'єктів даного класу.

Принципи проектування класів

Для того, щоб уникнути найбільш часто виникаючих проблем при проектуванні об'єктно-орієнтованих додатків, були розроблені принципи, яким рекомендується слідувати при розробці.

SOLID (single-responsibility, open-closed, Liskov substitution, interface segregation, dependency inversion) – це абревіатура, що безпосередньо відображає назви всіх основних принципів, які вона описує[9].

Принципи *SOLID* передбачають:

- Принцип єдиного зв’язку (SRP - Single Responsibility Principle);
- Принцип відкритості та закритості(OCP - Open-Closed Principle);
- Принцип підстановки Лісков(LSP - Liskov Substitution Principle);
- Принцип розділення інтерфейсу (ISP - Interface Segregation Principle);
- Принцип інверсії залежностей (DIP - Dependency Inversion Princi- ple).

Слідування даним принципам у розробці дає можливість створювати гнучке та розширюване програмне забезпечення.

Формулювання про принцип відкритості та закритості наголошує на тому що класи, модулі, функції системи та інші складові мають бути відкритими до розширення, але закритими для модифікації.

Таким чином в рамках слідування принципу open/closed має бути можливість розширити чи змінити поведінку системи без внесення змін до вже наявних компонент. Тобто програмні компоненти мають бути відкриті для масштабування, але закриті для внесення змін.

Що стосується принципу єдиного обов’язку та принципу підстановки Лісков – дані твердження наголошують на необхідності можливості замінити об’єкти в програмі нащадками, без втручання у вже існуючий код, а також на необхідності виконувати кожним об’єктом лише один обов’язок.

Клас повинен мати єдиний обов'язок. При цьому не повинно виникати більше однієї причини для зміни класу.

Класи повинні бути відкриті для розширення, але закриті для модифікації. Успадковує клас повинен доповнювати, а не заміщати поведінку базового класу. При цьому заміна в коді об'єктів класу-предка на об'єкти класу-нащадка не приведе до змін в роботі програми.

Замість одного універсального інтерфейсу краще використовувати багато спеціалізованих.

Залежно всередині системи будуються на рівні абстракцій. Модулі верх- нього рівня не повинні залежати від модулів нижнього рівня. Абстракції не повинні залежати від деталей, а навпаки, деталі повинні залежати від абстракцій.

Принцип підвищення зв'язності

Можливості підключення (cohesion) - це міра сфокусованості обов'язків класу. Вважається що клас має високий ступінь зв'язності, якщо його обов'язком ності тісно пов'язані між собою і він не виконує величезних обсягів раз- народного роботи.

Високий ступінь зв'язності (high cohesion) дозволяє домогтися кращої розпізнаваності коду класу та підвищити ефективність його повторного використання.

Клас з низьким ступенем зв'язності (low cohesion) виконує багато різних функцій або незв'язаних між собою обов'язків. Такі класи созда- вать небажано, оскільки це може призвести до виникнення наступних про- блем:

- Труднощі розуміння.
- Складність при повторному використанні.
- Складність підтримки.
- Ненадійність, постійна схильність до змін.

Класи зі слабкою зв'язністю, як правило, є занадто «абстрактними» або виконують обов'язки, які можна легко розподілити між другі- ми об'єктами.

Елемент з низьким ступенем пов'язаності (або слабким зв'язуванням, low coupling) залежить від не дуже великого числа інших елементів і має сліду- ющие властивості:

- Мале число залежностей між класами.
- Слабка залежність одного класу від змін в іншому.
- Високий ступінь повторного використання класів.

А ось тут про те які типові критерії успішної серверної частини але докладніше
Чи кожна архітектура має бути RESTful ? ні

Чому? Що нам дає співставність архітектури нашої системи із вимогами рест?

2.1.3. Налаштування середовища Spring-boot

Система «Task manager» уповні побудована на основі Spring-фреймворку.

В якості автоматизованого інструменту зборки і розвертання проекту використовується Spring Boot, який являється частиною фреймворку.

Spring Boot автоматично налаштовує програму на основі залежностей, які додано до проекту, використовуючи анотацію @EnableAutoConfiguration. Наприклад, якщо база даних MySQL знаходиться на вашому класі, але ви не налаштували жодного з'єднання з базою даних, то Spring Boot автоматично налаштовує базу даних в пам'яті.

Точкою входження програми для завантаження є клас, що містить анотацію @SpringBootApplication та основний метод.

Spring Boot автоматично сканує всі компоненти, що входять до проекту, використовуючи анотацію @ComponentScan.

Поводження з управлінням залежностей є складним завданням для великих проектів. Spring Boot вирішує цю проблему, надаючи набір залежностей для зручності розробників.

Наприклад, за необхідності використовувати Spring та JPA для доступу до бази даних, достатньо включити залежність spring-boot-starter-data-jpa у проект.

Важливим моментом є те, що всі початкові програми Spring Boot дотримуються однакової схеми іменування spring-boot-starter- *, де * вказує, що це тип програми.

Залежність виконавчого механізму Spring Boot Starter використовується для контролю та керування вашою програмою. Його код показаний на рисунку(рис.)

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Рис.

Залежність безпеки Spring Boot Starter використовується для безпеки Spring.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Рис.

Веб-залежність Spring Boot Starter використовується для написання кінцевих точок відпочинку. Його код показаний нижче

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Рис.

Автоматичне налаштування Spring Boot автоматично налаштовує програму Spring на основі залежностей від JAR, які додаються у проект. Наприклад, якщо база даних MySQL знаходиться на шляху використовуваного класу, але ви не налаштували жодного з'єднання до бази даних, то Spring Boot auto конфігурує базу даних в пам'яті.

Для цього потрібно додати основний анотацію @EnableAutoConfiguration або анотацію @SpringBootApplication до вашого основного файлу класу. Тоді ваша програма Spring Boot буде автоматично налаштована.

2.2. Організація роботи компонентів у Docker

Docker - це програмна платформа для швидкої збірки, налагодження та розгортання додатків за допомогою контейнерів.

Основні компоненти програмної платформи зображені на рисунку нижче(рис.) .

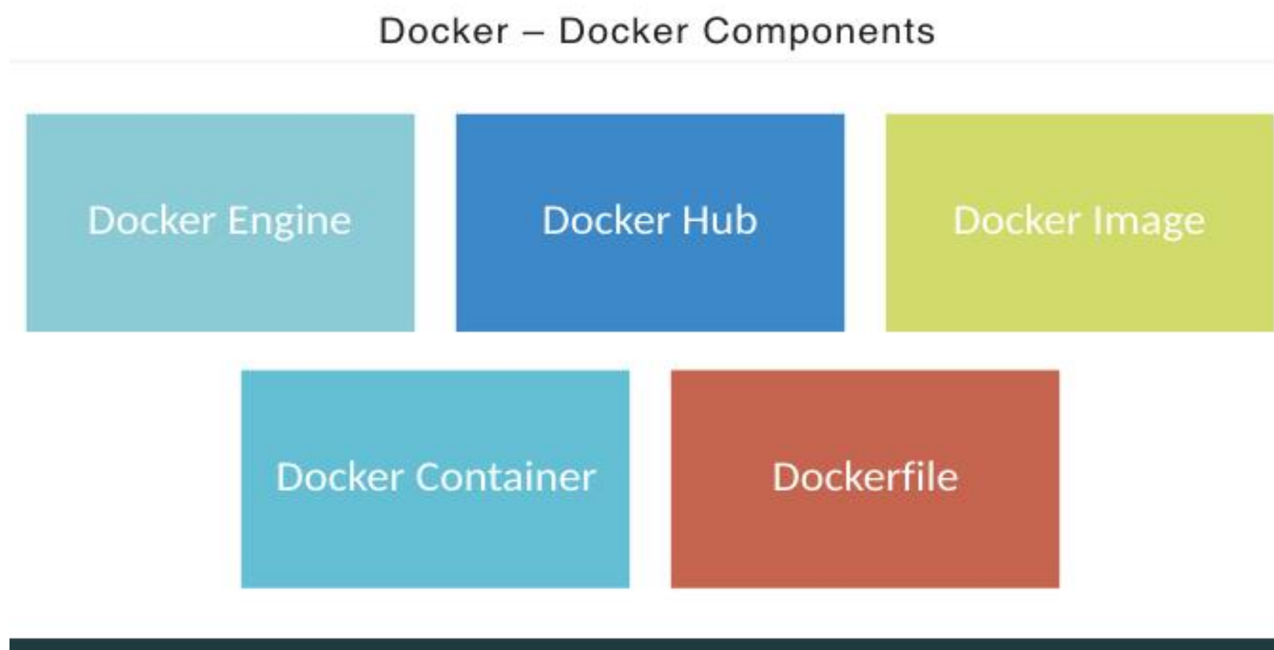


Рис.

Docker виконує наступні функції :

– Ізоляція
– Docker виділяє єдину програму та її залежності як від ОС, так і від інших контейнерів
– Docker переконує, що кожен контейнер має власні ресурси, ізольовані від інших контейнерів
– Переносність
– Docker інкапсулює все, що потрібно для запуску програми

– Будь-який хост із встановленою програмою виконання докера може запустити контейнер докера
– Продуктивність - додатки, що містять контейнери, використовують набагато менше пам'яті, ніж віртуальні машини
– Стандартизація - докер забезпечує послідовне середовище (ви можете локально запускати абсолютно те саме додаток - з точно такими ж залежностями, конфігурацією середовища та інфраструктури - те саме, що і у виробничому середовищі)

2.2.1. Налаштування середовища Docker

Докер використовується для контейнеризації середовищ, а у рамках розробки системи «Task manager» в даному випадку використовується для контейнеризації середовищ проектування серверної частини, клієнтської частини та бази даних.

Перш за все створюється файл конфігурації усіх контейнерів (рис.).

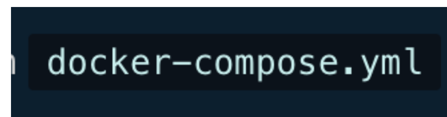


Рис.

Compose - це інструмент для визначення та запуску багатоконтейнерних програм Docker. За допомогою Compose ви використовуєте файл YAML для налаштування служб своєї програми. Потім за допомогою однієї команди ви створюєте та запускаєте всі служби зі своєї конфігурації. Щоб дізнатися більше про всі функції Compose, перегляньте список функцій.

Ініціація Compose - це трикроковий процес:

– Визначення середовища програми за допомогою Dockerfile, щоб воно могло бути відтворене в будь-якому місці.
– Визначення сервіси, які складають програму в docker-compose.yml, щоб вони могли працювати разом в ізольованому середовищі.

Запуск команди docker-compose і Compose запускає та запускає весь ваш додаток.

`docker-compose up` `docker-compose.yml`

Рис.

Docker запобігає конфлікту середовищ, у випадку розробки системи менеджменту задач – запобігає конфлікту БД.

2.2.2. Міграції баз даних

Міграція схеми виконується над базою даних коли необхідно оновити, або повернути схему бази даних до якоїсь новішої чи старішої версії.

Міграція бази даних - в контексті корпоративних програм - означає переміщення даних з однієї платформи на іншу. Процес міграції баз даних може включати кілька етапів та ітерацій - включаючи оцінку поточних баз даних та майбутніх потреб компанії або замовника, міграцію схеми та нормалізацію, переміщення даних, тестування.

Кожного разу при увімкненні докер-контейнері відбувається видалення всіх схем баз даних. Саме для цього було додано схему міграції (рис.), яка при кожній ініціації проекту дозволяє накочувати таблиці у БД.

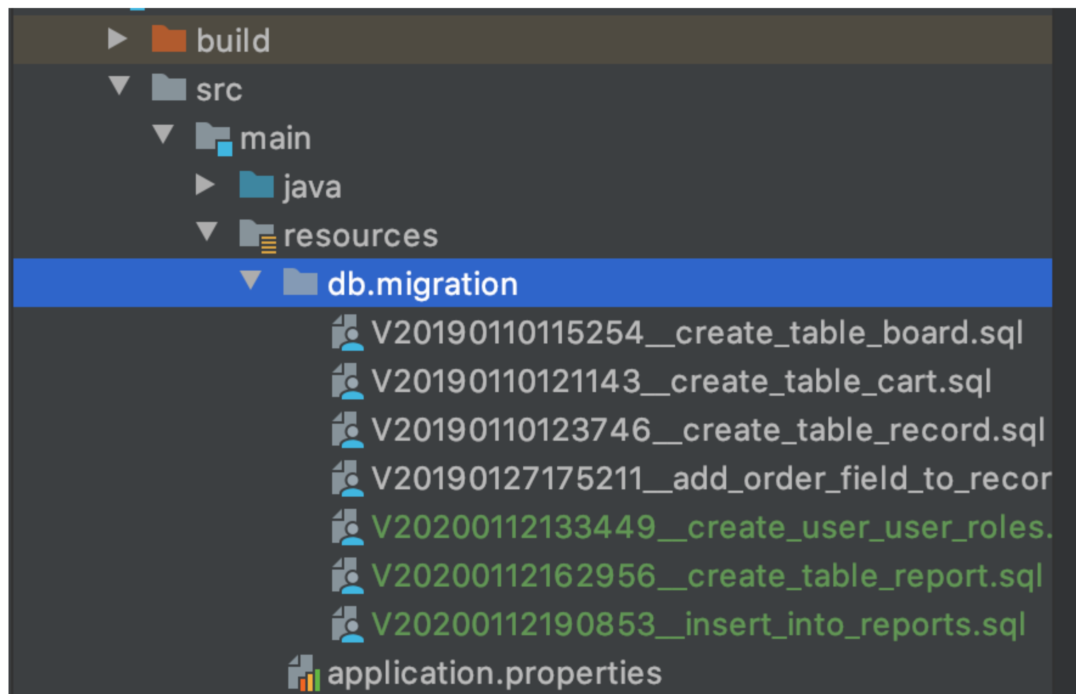


Рис.

Використання міграції дає можливість працювати із проектом незалежно від апаратного забезпечення, адже для передбачення видалення таблиць БД працюють міграції, які в даному проекті реалізовані за допомогою сервісу Flyway. Таким чином автоматизація запуску проекту та підняття контейнерів спрощується завдяки використанню міграцій, адже достатньо підняття контейнера з базою даних та запустити середовище.

Сервіс самостійно накочує міграції із файлу db.migration і створює власну таблицю , в якій записуються всі записані міграції та їх чек–сума.

Важливо відзначити, що у розробці також використовується патерн «Dependency injection» – впровадження залежностей.

Впровадження залежностей – це один із патернів проектування програмного забезпечення, який передбачає надання зовнішньої залежності програмному компоненту, завдяки використанню «інверсії управління» для розв'язання залежностей.

Таким чином під впровадженням залежностей мається на увазі делегування управління залежностями сторонньому контексту, у випадку системи «Tasl manager» – це Spring framework.

Оскільки DI імплементується за допомогою інтерфейсів, що знижує coupling компонентів, натомість самі ж компоненти додатку стаються менш зв'язними та, у свою чергу, спрощується масштабування та тестування додатку.

2.3. Архітектура модулів системи

Додаток «Task manager» представляє собою набір слабо зв'язаних (*low-coupling*) компонентів, які взаємодіють між собою за допомогою інтерфейсів. Даний підхід до організації компонентів серверної частини системи свідчить про використання модульної архітектури.

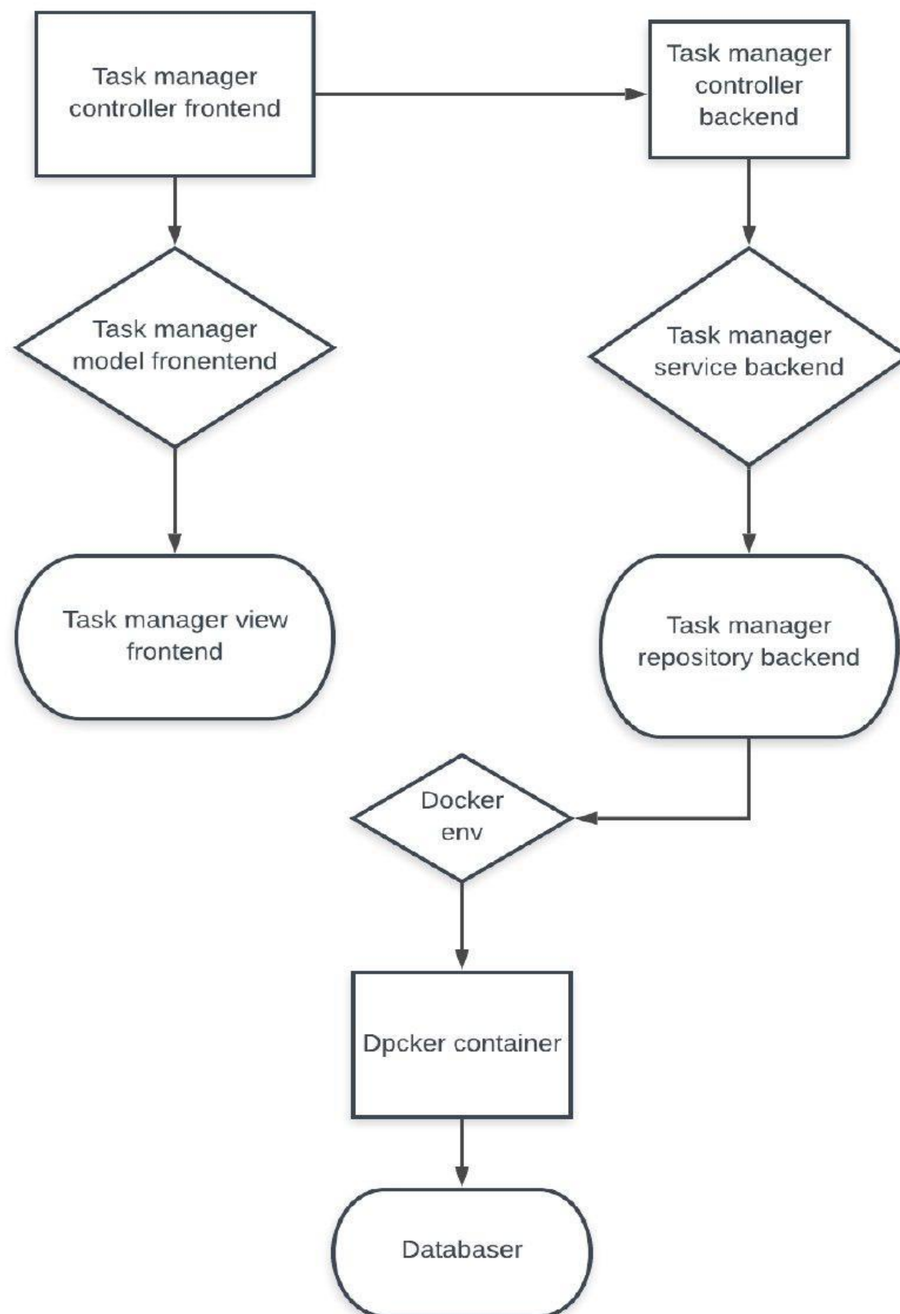


Рис.

2.3.1. Структура папки серверної частини проекту

Папка проекту перш за все вміщує точку входу у програму. Точкою входу є файл під назвою «EntryPoint.java»,який знаходиться у папці main/src/java/com/taskmanager.

```
1   package com.taskmanager;
2
3   import ...
5
6   @SpringBootApplication(scanBasePackages = "com.taskmanager")
7   public class EntryPoint {
8
9
10      public static void main(String[] args) { SpringApplication.run(EntryPoint.class, args); }
13  }
14
```

Рис.

Як вже було зазначено раніше є два підходи до побудови архітектури – це моноліт та мікросервіси. Різниця між монолтною архітектурою та мікросервісами в тому, що вся логіка програми у випадку монолітної архітектури знаходиться в одному місці, натомість мікросервісний передбачає розосередження лоігки додатку на різних програмних носіях та навіть апаратному забезпечення (окремі сервери).

Переваги мікросервісів над монолітною архітектурою основним чином полягають в тому, що таку архітектуру як моноліт важче масштабувати. Саме тому для реалізації системи «Task manager» було обрано модульну архітектуру.

Модульна архітектура є проміжною між мікросервісами та монолітом. Таким чином зв'язність компонентів зменшується, що, у свою чергу, я позитивним критерієм.

Окремо важливим питанням є те, як організована структура взаємодії модулів між собою, так зване спілкування. В рамках модульної архітектури клас з модуля А не можуть на пряму взаємодіяти з класами модуля Б.

Для того, щоб запобігти модульного використання зв'язаності, так званої жорсткої прив'язки одного модуля до іншого, було застосовано рішення – окремий модуль API, який являється сполучною ланкою між двома модулями.

Папка core являється модулем та вміщає базовий функціонал додатку. Core побудований базі Spring MVC, що дає чітке розділення між архітектурними шарами. У багаторівневій модульній архітектурі - один модуль може містити декілька підмодулів. В даному випадку модуль реалізовано в одинарному екземплярі.

Core вміщує три архітектурний прошарки :

- Контролер (Controller);
- Сервіс (Service);
- Репозиторій (Repository).

Кожен шар має власну зону відповідальності. Рівень організації шарів можна назвати таки, що шари нижнього рівня нічого не знають про шари верхнього рівня, отже кожен шар працює лише в межах власного архітектурного прошарку.

Таким самим чином влаштовані компоненти – компоненти нижнього рівня нічого не знають про компоненти верхнього рівня:

- Контролер на пряму взаємодіє з «сервісом», але в той же час нічого не знає про папку репозиторій.
- Сервіс взаємодіє із репозиторієм, але репозиторій нічого не знає ні про контролер ні про сервіс.
- Для репозиторія контролер та сервіс являються компонентами верхнього рівня, отже взаємодія закрита.

2.3.2. Функції компонент ядра системи

Контролер займається тим, що приймає запити і не містить ніякої логіки. Основною задачею контролера є надати точку входу у програму зовнішнім сервісам.

Важливим є те, що контролер використовує REST – архітектуру , що означає, що кожен контролер містить набір адрес, так званих «url-edpoint», які приймають та віддають запити. Відправка та прийом запитів відбувається за допомогою таких методів, як GET, POST, PUT, DELETE та інших.

Рис.

Сервіс являється проміжним шаром та використовує контролер для зберігання бізнес-логіки. Окрім цього однією із основних функцій сервісу є використання репозиторію для роботи з базою даних.

Рис.

Репозиторій є спеціальною категорією класів, яка спрощує роботу з базою даних в рамках архітектури Spring MVC.

Оскільки для спрощення роботи з бащою даних був використаний один із компонентів Spring під назвою Spring-data, який є надбудовою над фреймворком *Hibernate*, який є одним із найбільш широковживаних фреймів в Java, основне завдання якого проектувати поля з бази даних (вибірку з БД) на Java об'єкт.

Завдяки використанню Spring-data в рамках використання цього додатку майже не доводилось писати sql-запити та працювати із вибіркою даних на пряму. Таким чином при запиті методів у репозиторій на виході ми отримуємо об’єкт або колекції об’єктів.

Варто зазначити, що крім цього Spring-data дозволяє писати нативні sql -запити, які необхідні у випадках коли ми не можемо отримати необхідні дані вбудованими засобами Spring-data.

У додаток також була додана можливість працювати з вибіркою з бази даних на пряму. Це було реалізовано за допомогою вбудованого компонента Spring під назвою *jdbc-template.*

Так як в сервісах використовується інтерфейс ми можемо легко підмінити реалізацію і використовувати не Spring-data, а *jdbc-template*.

Окрім перелічених архітектурних прошарків, папка Core вміщує ще декілька важливих одиниць проекту. Перелік вмісту папки подано на рисунку нижче.

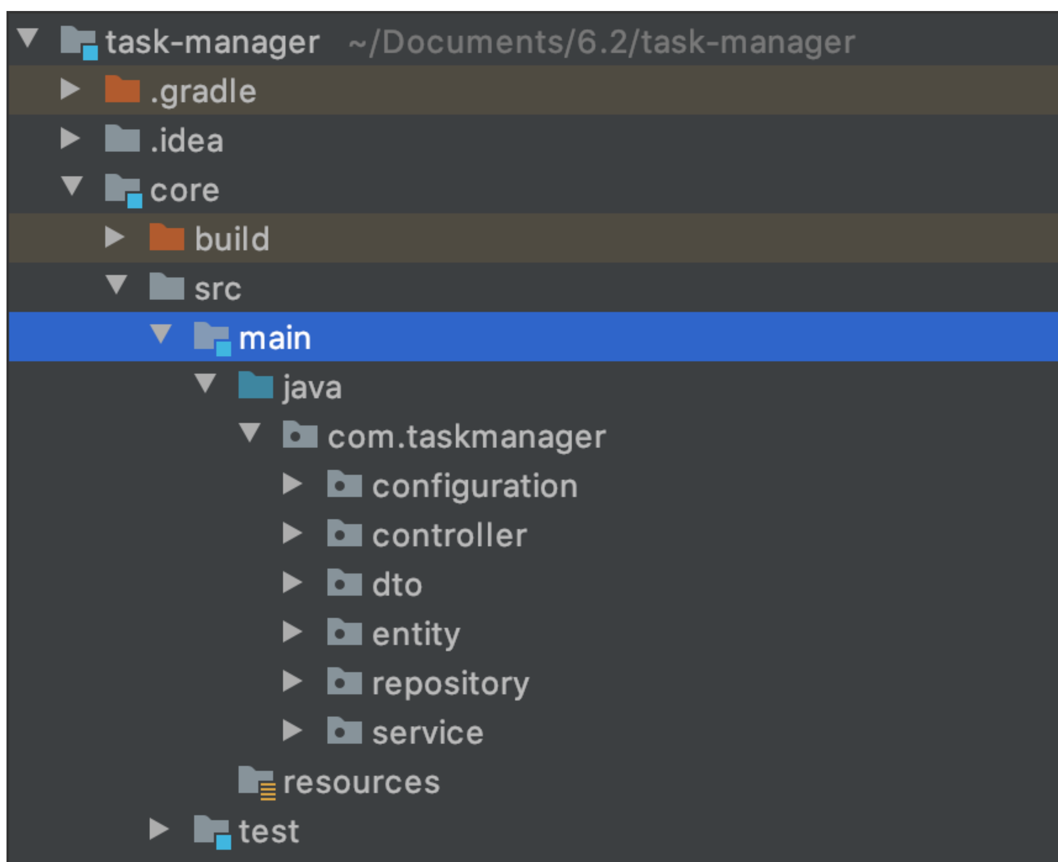


Рис.

Наступним важливим елементом вмістилища папки Core є Entity, що являється об’єктною репрезентацією таблиці із бази даних.

Entity

Працюючи з базою даних за допомогою стандартних інструментів Java, наприклад написання запитів через звичайний jdbc на виході ми отримуємо просто набір полів із яких необхідно сформувати Java-об’єкт, в той час як Hibernate робить цю роботу за нас.

При створенні Entity (сутоності) в якості полів повинні бути вказані колонки баз даних, крім цього обов'язковий атрибут id, таким чином Hibernate сканує такі

класи, помічені анотацією Entity і створює об’єктну репрезентацію таблиць з БД за нас.

При використанні Spring-data робота з базою даних стає ще простіше, так як описуючи прості методи в інтерфейсі, помічені анотації, репозиторій, як вихідних результатів цих методів ми отримуємо об'єкт або колекції об'єктів.

DTO

Так як ми працюємо з тришаровою архітектурою, де кожен нижній шар нічого не знає про верхній шар, кожен з шарів повинен використовувати свої об’єкти для передачі даних.

Тобто при отриманні з репозиторію об'єкта типу Entity ми не можемо використовувати його в контролері.

Таким чином, для цього був придуманий шаблон під назвою DTO -data transfer object, який можна використовувати в сервісі або в контролері.

DTO по суті являє собою об'єкт який містить поля Entity, але також він може містити поля інших об'єктів Entity, що використовується у випадках коли наприклад при запиті ендпоінту на якому-небудь контролері нам потрібно отримати дані не з однієї таблиці а з 1-4, і ці дані можуть не бути пов'язані між собою, що означає що DTO може містити поля із 3-4 і більше сутностей.

Mapper

Для того щоб конвертувати Entity в DTO були використані спеціальні маппери, які реалізовані як статично утилітарні класи.

Таке рішення було вибрано з тієї причини що популярні бібліотеки які використовуються в мові Java для таких цілей можуть конфліктувати з іншими бібліотеками, наприклад map-stract. який конфліктує з Lombok.

Однією з найважливіших складових будь-якого Java класу є гетери і сеттери які дозволяють отримати доступ до полів класу, в тому випадку якщо поля приватні.

Гетери і сеттери представляють собою шаблонний код, який можна згенерувати за допомогою будь-якої сучасної IDE. Однак є й інший спосіб, використання спеціальної бібліотеки Lombok, яка при компіляції проекту сама генерує гетери і сеттери.

Принцип роботи бібліотеки дуже простий, для того щоб Lombok генерував необхідні методи досить поставити над класом анотацію, крім цього Lombok вміє генерував конструктор як прості так і з усіма полями класу, що досягається за допомогою анотацій.

Крім цього Lombok вміє генерувати «equels = hashcode» в тих випадках коли необхідно перевизначити ці методи в класі.

2.3.3. Схема взаємодії компонентів на backend

Схема взаємодії компонентів представляє взаємодію таких компонентів як Boardservice, який отримує на вході сутність(entity) із бази даних від репозиторія і передає на контролер DTO.

У свою чергу з бази даних ми отримуємо entity, а на контролер віддаємо DTO.

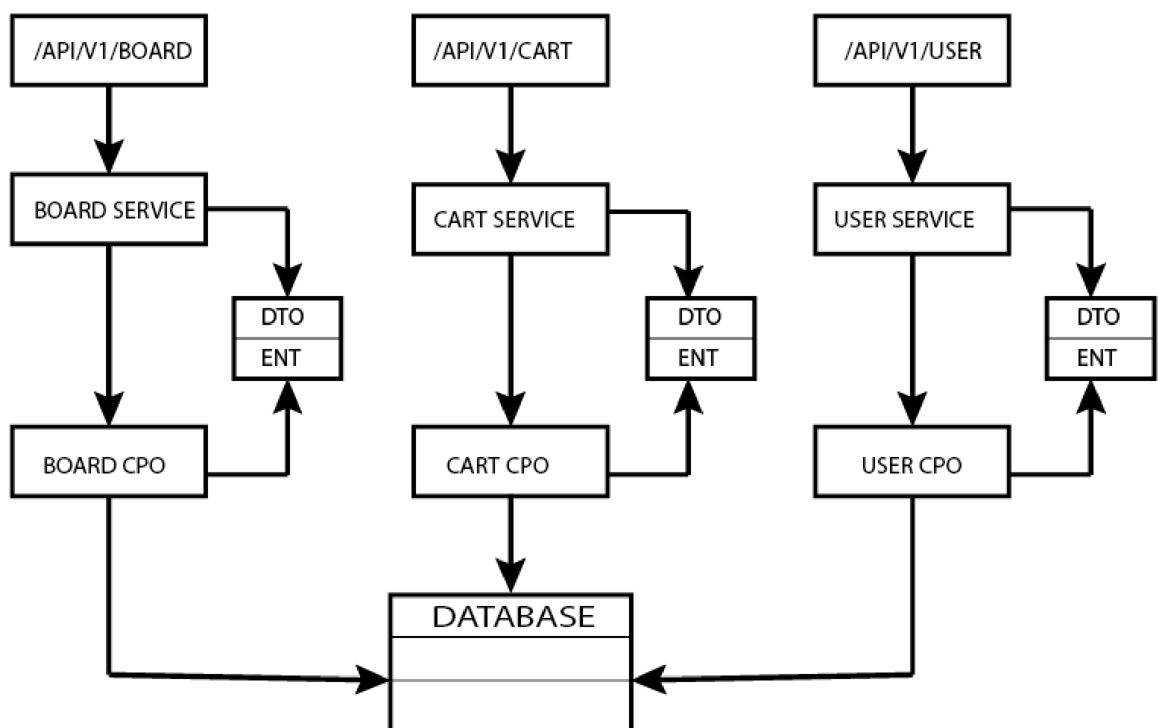


Рис. взаємодія леєрів на беку

2.4. Розробка модуля переміщення карт на екрані користувача

2.4.1. Постановка проблеми

Під час вирішення питання переміщення карт на користувацькому інтерфейсі з метою забезпечити приємний «user experience» було знайдено наступні проблеми:

- Як передати найменшу кількість даних для сортування?
- Як відслідкувати переміщення записів у колонках?
- Пошук алгоритму, за мінімальну кількість викликів бази даних, дозволив би змінити сортування.

В рамках веб-сервісу task-manager, в кожній картці (cart) можна створювати записи (cart records). Кількість записів не обмежена, отже створювати можна в необмеженній кількості.

Як це влаштовано? Кожна cart record зв’зана з cart по id cart. У кожної cart-record в базі данних і на стороні UI є параметр «record-order», який задає розташування один cart-record відносно іншого. При зберіганні cart record у базу, на бекенд передається серіалізованний об’єкт cartRecordDto, який містить необходні для заповнення в базі данних поля для cart record, а також cart id – ідентифікатор колонки, до якої буде прив’язанний запис.

.

```
saveCartRecord(){
  this.cartRecordService.saveCartRecord(this.data.cartId, this.cartRecordDto).
  subscribe(() => this.dialogRef.close());
}
```

Рис. Збереження нового запису код клієнтської сторони

При активації контрольної точки у CartRecordController, відбувається збереження нового запису на стороні сервера.

```
@PostMapping("{cartId}")
@CrossOrigin(origins = "http://localhost:4200")
public CartRecordDto saveCartRecord(@PathVariable Long cartId,
                                    @RequestBody CartRecordDto cartRecordDto){
    return cartRecordService.saveCartRecord(cartRecordDto, cartId);
}
```

Рис. Збереження нового запису зі сторони серверу

Після цього йде виклик методу saveCartRecord, який в свою чергу викликає імплементацію метода saveCartRecord через інтерфейс CartRecordService. Знаходження відповідної імплементації відбувається завдяки фреймворку Spring (фреймворк, який став уособленням ентерпрайз розробки, завдяки великій кількості рішень, що доступні після встановлення фреймоврку), який був використанний у проекті і його модулю, що є реалізацією принципу Dependency inversion, що дозволяє використовувати абстракції по типу інтерфейсів замість їх реалізацій.

Далі, в самому методі відбувається конвертація DTO об'єкту, яким можна користуватись тільки на рівні контролерів і сервісів, у об'єкт CartRecordEntity, який вже можна зберігати у базу, використовуючи фреймворк Hibernate, який являється найбільшим у Java розробці фреймворк, що використовується для реалізації ORM. Object relation mapping – проектування таблиць баз данних на POJO – plain old Java object.

```
public CartRecordDto saveCartRecord(CartRecordDto cartRecordDto, Long cartId) {
    CartRecord cartRecord = CartRecordCustomMapper.mapToCartRecord(cartRecordDto);
    Cart cart = new Cart();
    cart.setId(cartId);
    cartRecord.setCart(cart);
    Optional <CartRecord> cartRecordForOrder = cartRecordRepository.findTopByCartIdOrderByRecordOrderDesc(cartId);
    Long recordOrder = cartRecordForOrder.map(cartRecord1 -> cartRecord1.getRecordOrder() + 2).orElse( other: 2L);
    cartRecord.setRecordOrder(recordOrder);
    cartRecordRepository.save(cartRecord);
    return cartRecordDto;
}
```

Рис.

Далі, відбувається підготовка об'єкта до зберігання у базу. Обєкту cart record присвоюються id cart, до якого буде прив'язанний це запис. Крім того, відбувається виклик самого верхнього запису в колонці з бази з сортуванням по спаданню cart record.

У запису із бази береться cart order, до нього додається 2 і це стає індексом сортування нового елементу в списку, як він буде відображатись на ui. Отже,

здійснбється перевірка – чи існують елементи в колонках cart, якщо ні, елементу присвоюються індекс сортування record_order у базі, recordOrder у Entity 2.

Також, важливо додати, що всі індекси діляться без залишку на 2 (перший елемент має індекс 2, наступний 4 і т.д).

```
Long recordOrder = cartRecordForOrder.map(cartRecord1 -> cartRecord1.getRecordOrder() + 2).orElse( other: 2L);
cartRecord.setRecordOrder(recordOrder);
```

Рис.

Наступним кроком відбувається зберігання елементу у базу даних, з подальшим його відображенням у списку на ui.

Зміна порядку cart record при перетаскуванні їх одне відносно одного колонці відбувається по такому алгоритму:

З фронтенду відбувається виклик контрольної точки “/change-order” методу changeRecordsOrder, що є частиною контролера CartRecordController

```
public updateRecord(cartId: number, params: CartRecordDto): Observable<CartRecordDto>{
  return this.http.put<CartRecordDto>(this.url + `/${cartId}`, params);
}
```

Рис. Зміна порядку сортування клієнтська сторона

```
@PutMapping("/change-order")
@CrossOrigin(origins = "http://localhost:4200")
public void changeOrder(@RequestBody CartRecordQuery cartRecordQuery){
    cartRecordService.changeRecordsOrder(cartRecordQuery.getCartRecordId(),
            cartRecordQuery.getChangeIndex(), cartRecordQuery.getCartId());
}
```

Рис. Зміна порядку сортування сторона сервера

Виклик відбувається за допомогою Put метода. На вхід подаються такі параметри: ідентифікатор запису cart record, якому буде змінена позиція, ідентифікатор колонки cart, до якого цей запис прив’язаний, на скільки позицій перемістити запис відносно інших записів. Після цього відбувається виклик метода сервіса changeRecordsOrder, в якому власне і відбувається сортування.

Саме пересортування відбувається по такому алгоритму:

```
public void changeRecordsOrder(Long cartRecordId, Integer changeIndex, Long cartId) {
    CartRecord changed = cartRecordRepository.getOne(cartRecordId);
    Optional<CartRecord> cartRecordChanging = cartRecordRepository.
            findByRecordOrderAndAndCartId( recordOder: changed.getRecordOrder() + changeIndex*2, cartId);
    cartRecordChanging.ifPresent(cartRecord -> saveOrders(cartRecord, changed));
}
```

Рис. Алгоритм пересортування

Спочатку, з бази данних вибирається запис по cart record id. Далі, з бази данних вибирається запис елемента, з ким буде відбуватись зміна record_order. Далі, відбувається виклик приватного метода saveOrders(рис. .

```
private void saveOrders(CartRecord changing, CartRecord changed) {
    Long forChanged = changed.getRecordOrder();
    changed.setRecordOrder(changing.getRecordOrder());
    cartRecordRepository.save(changed);
    changing.setRecordOrder(forChanged);
    cartRecordRepository.save(changing);
}
```

Рис. Виклик приватного методу

У приватному методі saveOrders у записів міняється record_order і записи з новим індексом сортування записються у базу. Основні проблеми, що виникли під час написання алгоритму:

- Як передати найменшу кількість данних для зміни сортування: Вирішено – визначені ключеві характеристики cart record і ті з них, які необхідні для зміни сортування.
- Як відслідкувати переміщення записів у колонках: Вирішено – бібліотека на Angular (один з найпоширенніших фреймворків, що використовується для створення клієнтських додатків), що дозволяє відслідковувати переміщення курсора.
- Сам алгоритм, який за мінімальну кількість викликів бази даних, дозволив би змінити сортування: Вирішено оптимізація таблиці cart records, всі виклики бази данних в одній транзакації.

2.5. Розробка модуля логіну

2.5.1. Архітектура модуля авторизації та автентифікації

Архітектура влаштована наступним чином. При введені логіну та паролю користувачем на головній сторінці додатку, посилається запит на контролер. Логін та пароль зберігаються у зашифрованому вигляді у базі даних. Сама система містить ідентифікатор ролі для кожного користувача.

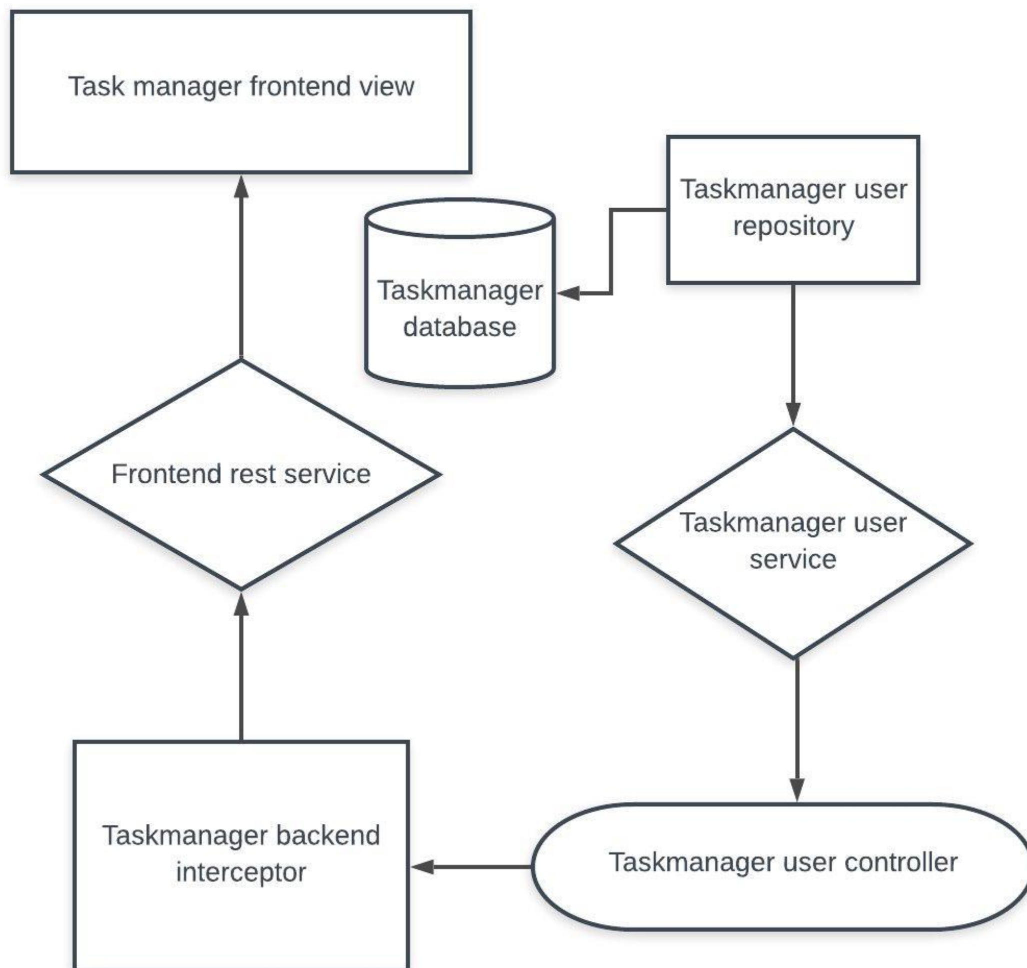


Рис. Схема архітектури модуля авторизаці та аутентифікації

При запиті на контролер відсилається запит у базу даних, яка здійснює перевірку– чи існує такий користувач із вказаним паролем та логіном у таблиці.

Якщо існує то на основі цих даних компонент Spring-security формує спеціальний об'єкт під назвою principal, в якому зберігаються ідентифікатор користувача та його роль. Після цього кожного разу при запиті інших контролерів , Spring-security перехоплює ці запити та перевіряє чи у користувача наявні права доступу до цих контролерів. Згідно ідентифікатору ролі якщо користувач не має прав доступу – відбувається переадресація на сторінку помилки.

2.5.2. Реалізація модуля авторизації

Система автентифікація-авторизації в проекті task-manager реалізована на основі компоненту фреймворку Spring – Spring security. Таке рішення було вибрано по причині доступності більшості функцій, що потрібні для реалізації такої системи типу «out of box».

Перед стартом веб-сервісу були створені міграції, які додали в базу даних таблицю users, roles, і user_roles, яка по-суті є зв'язковою і не зберігає ніякої інформації, крім ідентифікаторів користувачів і їх ролей. Такі рішення було використано по причині зв'язку таблиці користувачів до таблиці ролей «Один до Багатьох» (тобто у одного користувача може бути багато ролей). Крім того, при доданні поля роль в таблицю користувачів була б порушена нормалізація таблиці «користувач».

Коли користувач заходить на головну сторінку, перед ним з'являється 2 кнопки: Sign in (залогінитись), Registration (реєстрація).

Рис. Головна сторінка веб-сервісу task-manager

При переході на строрінку реєстрації перед користувачем відкривається з 4 полями – ім'я, email, пароль і роль.

Registration

Enter your name

Enter email

Enter password

enter role

Register

Рис. Форма реєстрації веб-сервісу task-manager

При введенні данних в поля і нажатті на кнопку register йде збір даних в об’єкт registrationDto, що містить всі поля форми і передача його на сервер через функцію saveUser ()

```
params: RegistrationDto = new RegistrationDto();

constructor(private registrationService: RegistrationRestService,
            private router: Router) {
}

ngOnInit(): void {
}

saveUser() {
  this.registrationService.saveUser(this.params).subscribe(() => {
    this.router.navigate(['/login']);
  });
}
}
```

Рис.

Це в свою чергу веде до активації контрольної точки “/registration” в AuthorizationController, що знаходиться на сервері. Далі, через виклик метода

saveUser(), що знаходиться в UserSecurutyService, відбувається перевірка – чи існує користувач з таким email і якщо існує, на клієнт передається помилка.

Створення нового користувача на серверній частині виглядає наступним чином (рис.).

```
public void saveUser(RegistrationDto registrationDto) {
    userRepository.findByEmail(registrationDto.getEmail()).orElseThrow(()
            -> new IllegalArgumentException("User with email already exists"));
    User user = new User();
    user.setPassword(bCryptPasswordEncoder.encode(registrationDto.getPassword()));
    user.setEmail(registrationDto.getEmail());
    user.setUserName(registrationDto.getName());
    Role role = new Role();
    role.setRole(registrationDto.getRole());
    user.setRoles(new ArrayList<>(
            Collections.singletonList(role)
    ));
    userRepository.save(user);
}
```

Рис. Створення нового користувача – серверна частина

Далі, створюється новий об'єкт користувача, якому записується пароль. Пароль в базі даних зберігається не в простому вигляді, а в зашифрованому, для шифрування паролів використовується клас BCryptPasswordEncoder, що є частиною компонента Spring security. Клас хешує паролі по алгоритму SHA2, такі паролі є криптостійкими. Для того, що об'єкт bcryptPasswordEncoder без конфліктів підтягувався Spring context-ом, класс був об'явлений як Bean у классі конфігурації

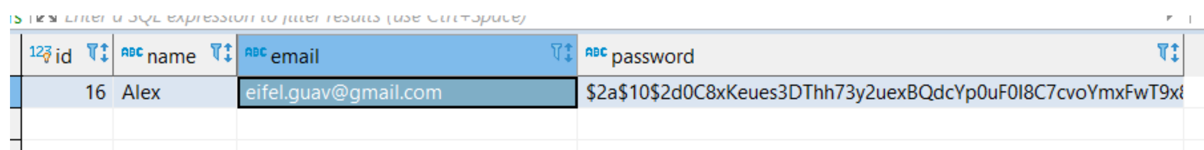
Об'явлення класу BCryptPasswordEncoder як Bean у конфігураційному класі представлено на зображенні.

```
@Bean
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Рис.

Також користувачу додається роль, яка була передана з клієнтської сторони і відбувається запис користувача в базу даних. Ідентифікатор користувача

генерується автоматично, завдякий використанню типу данних Serial у Primary key, , що працює схожим з Auto Increment чином у базах MySQL.



id	name	email	password
16	Alex	eifel.guav@gmail.com	$2a$10$2d0C8xKeues3DThh73y2uexBQdcYp0uF0I8C7cvoYmxFwT9x

Рис. Приклад зареєстрованого користувача в інтерфейсі DBeaver

Після вдалої реєстрації користувача, на клієнтській стороні йде переадресація на сторінку login, де користувач має ввести email і пароль, які він вводив при реєстрації.

Сторінка логіну веб-застосунку task-manager представлена на наступному зображенні.

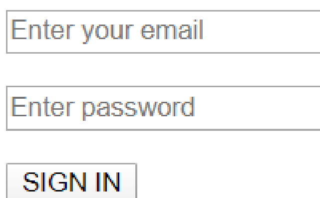


Рис. Сторінка логіну веб-застосунку task-manager

Після введення email і password, дані збираються в об’єкт userLoginDto і передаються на сервер в функцію login().

Передача даних користувача на сервер на клієнтській стороні зображена на рисунку.

```
params: UserLoginDto = new UserLoginDto();

ngOnInit(): void {
}

login() {
  this.loginService.loginUser(this.params)
```

Рис. Передача даних користувача на сервер

Дані, зібрані для авторизації серіалізуються з об’єкту в Json і передаються на сервер.

Основні компоненти Spring security:

1. Перший і основний компонент – класс конфігурації, який анотується @Configuration і @EnableWebSecurity – при сканування классів через механізм рефлексії Spring визначає, що налаштування потрібно брати з цього класу. Також, цей клас має наслідувати інший WebSecurutyCofigurerAdapter і переоприділяти метод configure

```
@Override
protected void configure(HttpSecurity httpSecurity) throws  Exception {
```

Рис.

В цьому методі задаються основні налаштування. Більше детально:

```
httpSecurity.httpBasic().disable()
        .authorizeRequests()
        .antMatchers( ...antPatterns: "/login").permitAll()
        .antMatchers( ...antPatterns: "/register").permitAll()
        .antMatchers( ...antPatterns: "/board").hasRole("DEVELOPER")
        .antMatchers( ...antPatterns: "/admin").hasRole("ADMIN")
```

Рис.

У налаштуваннях задано доступність контрольних точок різним користувачам. Наприклад, контрольні точки, що починаються з “/login” або “/register” достпні будь-якому користувачу, контроль точки, що починаються з “/board” доступні тільки авторизованним користувачам, що мають роль Developer, а контроль точки “/admin” доступні тільки адміністраторам системи.

2. Другий компонент – це клас, що імплементує інтерфейс UserDetailsService, з якого імплементується єдиний наявний в інтерфейсі метод loadUserByUserName(String userName), у якості username можна передавати інший ідентифікатор, наприклад, email.

```java
@AllArgsConstructor
@Service
public class CustomUserDetailService implements UserDetailsService {

    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepository.findByEmail(username).orElseThrow(()-> new UsernameNotFoundException("User not found"));

        return new org.springframework.security.core.userdetails.User(
                user.getUserName(),
                user.getPassword(),
                getUserRoles(user.getRoles())
        );
```

Рис. Імлементація інтерфейсу UserDetailsService

У якості вихідних параметрів методу виступає об'єкт UserDetails, з ролями, який Spring Security використовує для ідентифікації користувача.

3. Третій компонент – UserRepository, який власне і отримує користувача в з бази данних. У веб-сервісі Task-manager був використаний Spring JPA, що сильно спрощує роботу з базою данних.

```java
@Repository
public interface UserRepository extends JpaRepository<User, Integer> {

    Optional<User> findByEmail(String email);
    User findByUserName(String userName);
    Optional<User> findByEmailAndPassword(String email, String password);
}
```

Рис. Імплементація інтерфейсу UserRepository

Покроково як працює система: при введені даних користувачем і відправці їх на сервер, Spring security на основі введених даних створює UserDetails і в залежності від вибраного способу, зберігає цей об'єкт в контексті. Також, створюється сессія або токен, в який записується ідентифікатор авторизованого користувача (в веб-сервісі task-manager це реалізовано через сесію). При кожному наступному запиті контрольних точок, система перехоплює ці виклики, за допомогою так званих «фільтрів».

Тут варто зупинитись детальніше. Spring security, Spring MVC – компоненти, які реалізовані на основі сервлетів, що є стандартом в Java, сервлети, по-суті і

представляють собою ті контролери, з якими взаємодіє клієнт. Крім сервлетів, ще існують фільтри, які перехоплюють запити і при виконані якихось умов (наприклад, чи авторизований користувач), передають управління далі на сервлет.

Крім того, в Spring security і Spring MVC існує поняття Dispatcher Servlet, що по-суті являє собою єдину точку входу, і який переадресовує виклики на потрібні контроллери/фільтри.

Наступним чином, Spring security порівнює дані з поточної сесії і дані, що були записані в UserDetails і далі віддає/забороняє виклик потрібної контрольної точки. Крім того, в веб-сервісі task-manager був реалізований додатковий функціонал у вигляді об’єкту з кодом

```
public UserLoginDto findByEmailAndPassword(UserLoginDto userLoginDto) {
    Optional <User> user = userRepository.findByEmail(userLoginDto.getEmail());
        if (!user.isPresent()) {
            return new UserLoginDto( email: null,  password: null,  responseCode: 404);
        }
```

Рис. Додаткова перевірка сторона серверу

Якщо користувач не знайдений, на клієнт передається з кодом 404. Клієнт в свою чергу перевіряє відповідь від серверу

```
const responseCode = response.responseCode;
if (responseCode === 404 || responseCode === 400) {
  this.router.navigate(['/']);
  this.snackBar.open('Email or password incorrect', null, {
    duration: 2000,
  });}
```

Рис. Додаткова перевірка сторона клієнта

В залежності від відповіді переадресовує на потрібну сторінку на клієнті.

Переваги використаної системи авторизації/автентифікації – відносна простота і надійність.

Також, один з найпопулярніших нині способів створення такої системи – JWT токени, які представляють собою зашифровану на стороні серверу строку, в яку шифруються дані про користувача, його роль і інше.

Токени підписуються 2 ключами – приватним і публічним. Приватний ключ може зберігатись різними способами: у вигляді файлу, в InMemory базі даних (тип бази даних, що не використовує фізичне сховище, а існує тільки в оперативній пам’яті), наприклад Redis. Такий такий передається в header. При генерації токену, він підписується публічним ключем, для розшифрування токену потрібна пара публічний-приватний ключ.

Це Access Token, у якого є термін життя, який задається конфігураціями. Для перегенерації Access Token використовується Refresh token, який зберігає інформацію про час життя Access Token у випадку закінчення цього терміну дозволяє перегенерувати його. Варто додати, що при генерації нового Access Token відбувається також і генерація Refresh token. Останній також передається в Header при запитах на сервер.

Також досить розповсюдженою є 2-факторная система автентифікації. Принцип її роботи такий:

1. При вводі логіна і пароля генерується тимчасовий токен (час життя у якого до 20 хвилин), і користувач переадресовується на сторінку, з QR-кодом, полем, де він має ввести код, що прийшов в смс або прийняти дзвінок з номеру і нажати якусь цифру.
2. При вводі корретних коду/додаткових даних генурується пара Access-Refresh token і користувач отримує можливість зайти в систему.

Найпростіша схема 2-факторної аутентифікації

Основа перевага систем, в яких використовується 2-факторна аутентифікація – нівелювання можливості зламу акаунту користувача, тому що, по-суті, треба отримати доступ одразу до 2 пристроїв.

Також, не зайвим буде згадати про сучасний стандарт протокол авторизації Oath2.0.

В основі ідеї протоколу є те, що користувач не надає свої дані додатку, тобто додаток їх не зберігає, що прибирає ймовірність зламу акаунта користувача.

Інша ключова ідея – передача даних, тільки у зашфированому вигляді.

Це забезпечує гарантію користувачам, що всі маніпуляції з даними будуть проводитись тільки в заданих користувачем рамках.

Одним із прикладів реалізації протоколу і також архітектурного принципу *Single Sign In*(про який вже згадували вище) є *SAML* – security assertion markup language – по суті мова розмітка, реалізовано за допомогою *XML*, що дозволяє декларативно обмінюватись даними між постачальниками авторизаційних даних (під постачальниками мається на увазі, наприклад, *JWT*-токени).

```
<saml2p:Response xmlns:saml2p="urn:oasis:names:tc:SAML:2.0:protocol"
                 Destination="http://localhost:8888/simplesamlphp/www/module.php/saml/sp/saml2-acs.php/example-okta-com"
                 ID="id79474354816085061715642415"
                 InResponseTo="_37b09eebbdb6d226cef76a0a7b6f30fb032ce950a3"
                 IssueInstant="2017-01-31T22:28:44.788Z"
                 Version="2.0"
                 xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 >
```

Рис. Приклад коду написанного на SAML

Висновки до розділу 2

Розвиток програмних бібліотек навколо об’єктно-орієнтованої мови Java створив масштабну екосистему, яка дозволяє використовуючи засоби фреймворків вирішувати чимало проблем.

В даному розділі було розроблено та детально описано процес створення серверної частини веб-додатку в рамках обраної архітектури MVC та за використання функціоналу фреймворку Spring-boot та його складових компонент.

У якості програмного забезпечення для контейнеризації було вибрано Docker, як сучасний і широко застосований продукт для керування контейнерами і образами.

Розроблений модуль авторизації та автентифікації також було забезпечено високим рівнем захищеності, що надає можливість сміливого використання сервісу у масштабованому варіанті.

РОЗДІЛ 3
РОЗРОБКА КЛІЄНТСЬКОЇ ЧАСТИНИ ВЕБ-ДОДАТКУ

3.1. Робота з середовищем фреймворку Angular

3.1.1. Налаштування середовища

Налаштування локального середовища для роботи із фреймворком перш за все вимагає наявності Node.js та менеджера пакетів npm[].

Програми Angular, Angular CLI та Angular залежать від особливостей та функціональних можливостей, що надаються бібліотеками, які доступні у вигляді пакетів npm. Завантаження та встановлення npm передбачає наявність менеджера пакетів npm.

Щоб встановити CLI за допомогою npm, необхідно виконати таку команду:

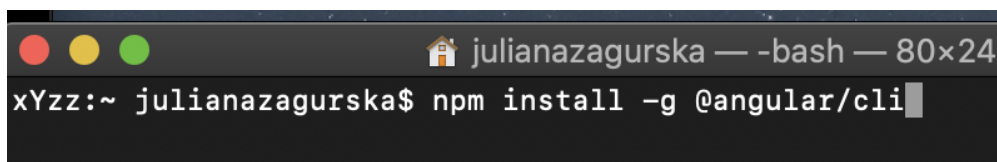


Рис.

Подальша робота передбачає створення нової робочої області. При введенні команди ng new необхідно задати ім’я створюваного додатку. Після ініціації створення нового Angular-додатку самим фреймворком буде запропоновано додати роутинг сторінок та можливість встановити мінімальні компоненти для провадження стилів(рис.).

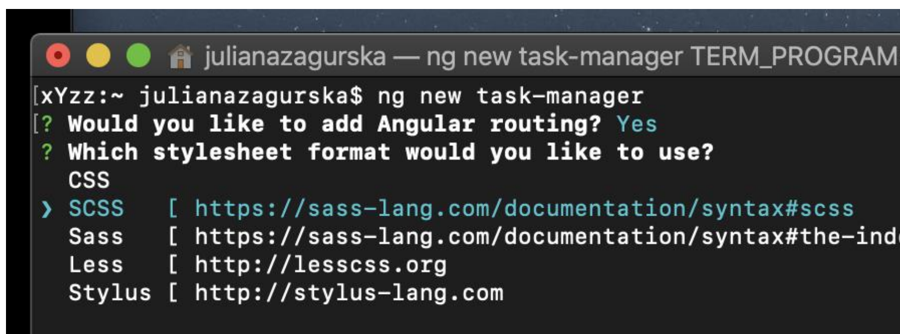


Рис.

Наступним етапом буде запуск додатку. Команда *ng serve* запускає сервер, переглядає необхідні файли та відновлює додаток під час внесення змін до цих файлів.

У папці проекту додаток представлений як папка під назвою Frontend, що зображена на рисунку нижче (рис.).

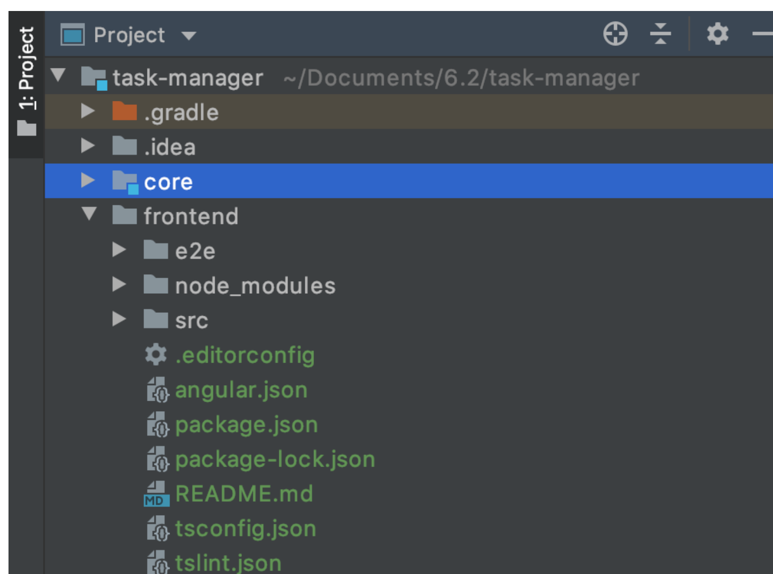


Рис.

Папка Frontend містить такий важливий пекедж як rest, в якому знаходяться контролери. Контролери відповідають за прийом і відправку даних. Аналогічно, кожен пекедж кожного з модулів серверної частини містить папку controller, отже так само володіє функціоналом прийому та відправки даних на бекенді.

Однак контролери на фронтенді, дослівно, нічого не знають про бекенд, адже все що потрібно контролерам із фронтенду це адреса « url» на яку вони «стукають».

3.2. Схема реалізації патерну MVC

Вся взаємодія на клієнтській стороні побудована фактично на тих самих принципах, що і на стороні сервера, адже фронтенд-частина базується на принципах Model, View, Controller (рис.).

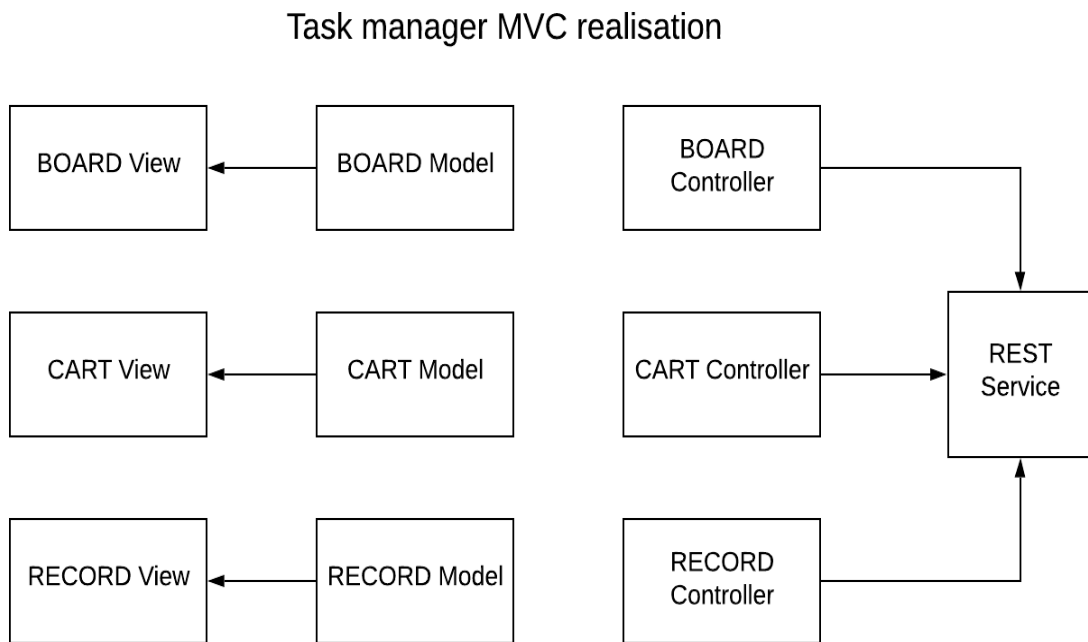


Рис. Схема архітектури роботи Angular в рамках MVC

Як і на стороні сервера, наступні контролерипь відповідають за відправку даних на сервер і їх прийом і обробку з серверної сторони:

– BoardController
– UserController
– CartController
– CartRecordController

Так, як для реалізаціїї клієнтської сторони був використанний Angular з Typescript (мова, що додає типізацію у JS), як і на стороні сервера, на стороні клієнта можна оперувати об’єктами.

Окремо, хотілось би зупинитись на інтерцепторі на клієнтській стороні, що конвертує *JSON*, який приходить з сервера в об’єкт, яким типізується функція, виклик якої перехоплює інтерцептор

```
get<T>(url: string, params?) {
  return super.get<T>(url, this.parseParams(params));
}
```

Рис. Функція інтерцептора

```
parseParams(data) {
  let params = new HttpParams();

  if (data) {
    Object.keys(data).forEach(key => {
      if (data[ key ] != null && data[ key ] !== '') {
        params = params.append(key, data[ key ]);
      }
    });
  }

  return { params };
}
```

Рис. Парсинг параметрів

Так само і об’єкти, що відправляються на сервер конвертуються цією ж функцією інтерцептора в *JSON*.

Наступними йдуть сервіси:

- Board service
- Cart service
- Cart record service
- User service

Дані сервіси взаємодіють з моделями (під моделями маються на увазі об'єкти, що відправляються на/з контроллерів). Такими моделями до прикладу є:

– BoardDto
– CartDto
– CartRecordDto

Також є допоміжні моделі, які потрібні, наприклад, для здійснення якихось маніпуляцій з даними, яскравим прикладом такої моделі є CartRecordQuery, що використовується для зміни порядку сортування Cart record всередині Cart

```
export class CartRecordQuery{

  cartRecordId: number;
  cartId: number;
  changeIndex: number;

  constructor(data?: CartRecordQuery){
    if (data){
      this.cartRecordId = data.cartRecordId;
      this.cartId = data.cartId;
      this.changeIndex = data.changeIndex;
    }
  }
}
```

Рис.

Останнім шаром є *View (Board view, Cart record view і інші)*, з якими взаємодіє користувач. Тільки View і ніякий інший шар окрім View не може взаємодіяти з користувачем.

Технічно – це звичайний HTML, який за допомогою 2-сторонньої прив'язки (одна з особливостей Angular framework, коли данні, що вводяться/відображаються в формах у View відразу прив'язаються до змінних у сервісах) дозволяє організувати взаємодію View і моделей.

```
<div mat-dialog-content>
<p><input matI="text" [(ngModel)]="cartRecordDto.title" placeholder="Enter title"></p>
<p><textarea [(ngModel)]="cartRecordDto.description" placeholder="Enter description"></textarea></p>
<p><input type="text" [(ngModel)]="cartRecordDto.comment" placeholder="Enter comment"></p>
<p><input type="datetime-local" [(ngModel)]="cartRecordDto.deadline"></p>
<p>Deadline</p>
<button (click)="processForm()">Save record</button>
</div>
```

Рис. Приклад *View* – компонент створення нового запису у карті

3.2.1. Архітектура взаємодії клієнта та сервера

Користувача через шар View передає певні дані, які віддаються на модель. Модель через сервіс передається на контроллер, що вже безпосередньо взаємодіє з Rest сервісом на стороні бекенду.

Модель є доволі простою, легка в масштабуванні, кожен компонент і шар логічно відділений від інших. Крім того, така модель дозволяє замінити будь-який компонент, без необхідності зупинки всього сервісу.

При зміні Rest-сервісу, з яким взаємодіє шар контролерів, достатньо просто змінити посилання на головний лінк, адже згідно стандартів, самі контрольні точки не можуть змінюватись.

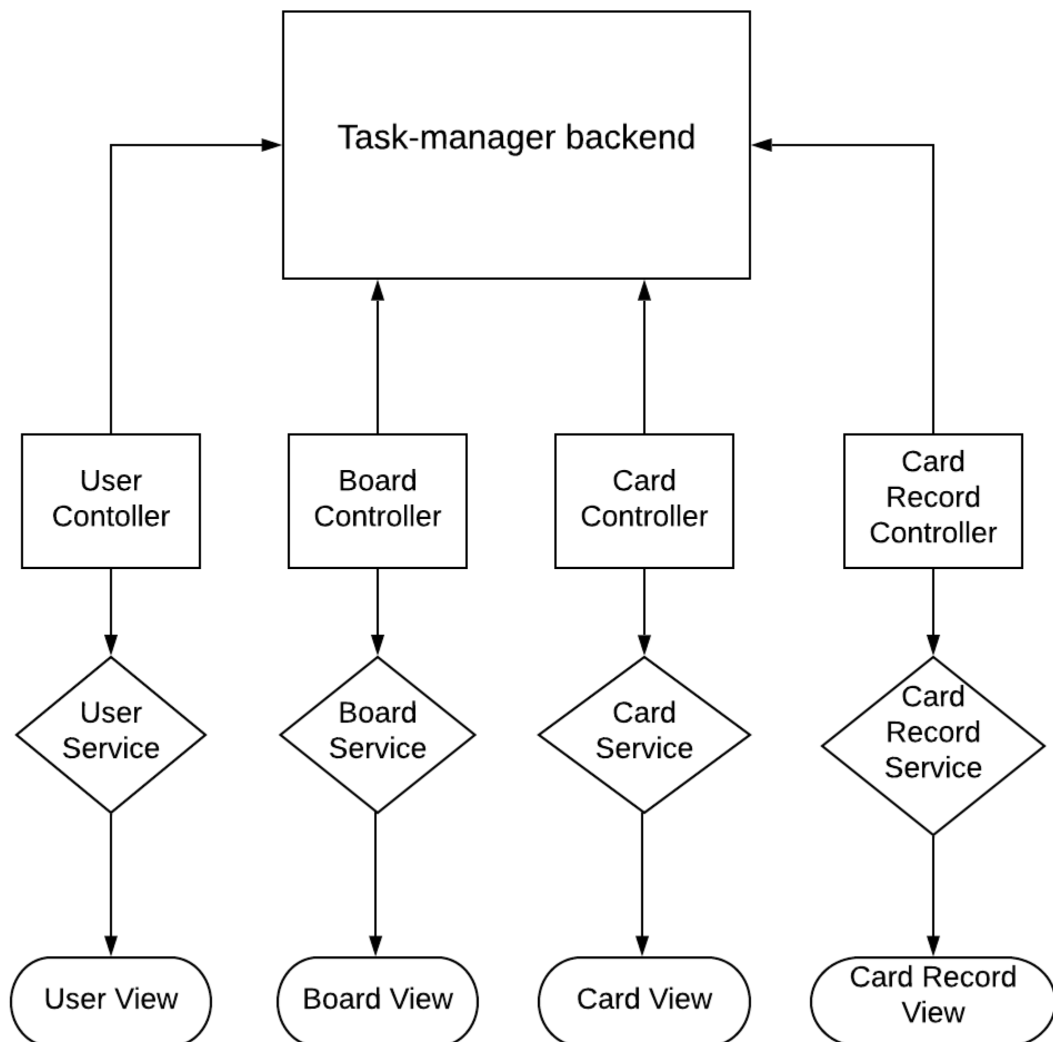


Рис. Схема архітектури взаємодії клієнта та сервера

3.3. Розробка дизайну інтерфейсу

Верстка мінімалістичного інтерфейсу в рамках фреймворку Angular виглядає наступним чином(рис.). На зображенні продемонстровано верстку компоненту Board.

```
<div style="background-color: crimson" *ngFor="let board of boardDto" class="text-center">
  <div style="...">

    <h1 style="...">{{board.title}}</h1>

  </div>
  <p style="..."><button (click)="deleteBoard(board.id)">Delete Board</button></p>
  <p style="..."><button (click)="getOne(board.id)">Open board</button></p>

</div>


  <input type="text" [(ngModel)]="boardName" placeholder="Enter Board title"><br>
<button (click)="saveBoard()" type="submit"
        style="...">Save Board</button>
```

Рис.

Забезпечення компонентів логікою відбувається за рахунок вказування адрес на початку файлу кожного компоненту(рис.) . Таким чином серверна частина постачає нас багатим функціоналом, який відпрацьовує відносно потреб та взаємодії із сайтом користувача.

```
import { Component, OnInit, Injectable } from '@angular/core';
import {UserLoginDto} from '../rest/login/userLoginDto';
import {LoginRestService} from '../rest/login/login-rest.service';
import {Router} from '@angular/router';
import {MatSnackBar} from '@angular/material/snack-bar';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.scss'],
  providers: [LoginRestService]
})
```

Рис.

Наступними не менш важливими елементами є розробка інтерфейсної частини сторінки авторизації та автентифікації. Стилі усього проекту виконані за допомогою

SCSS – препроцесора, який володіє вищим рівнем абстракції ,являється скриптовою мовою та інтерпретований у таблиці стилів CSS.

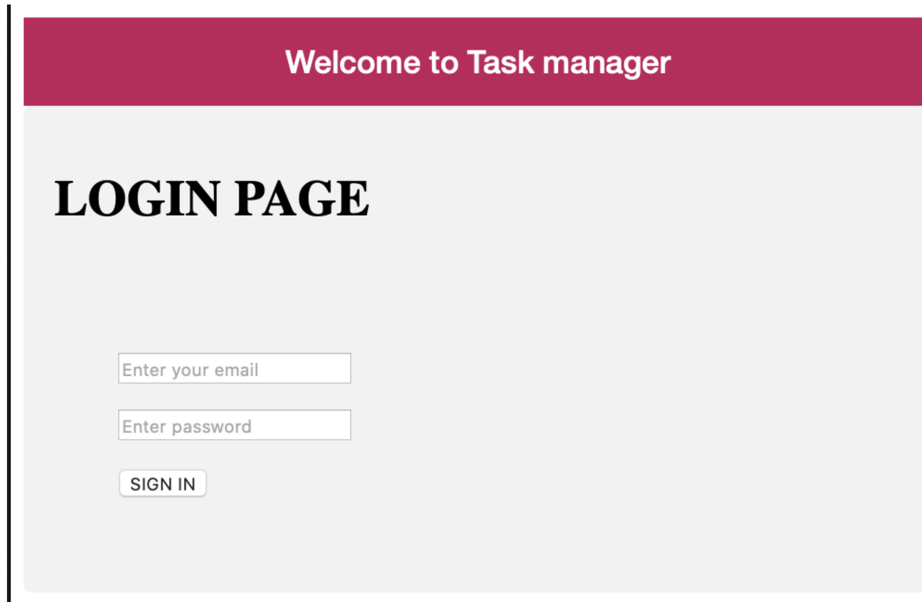


Рис.

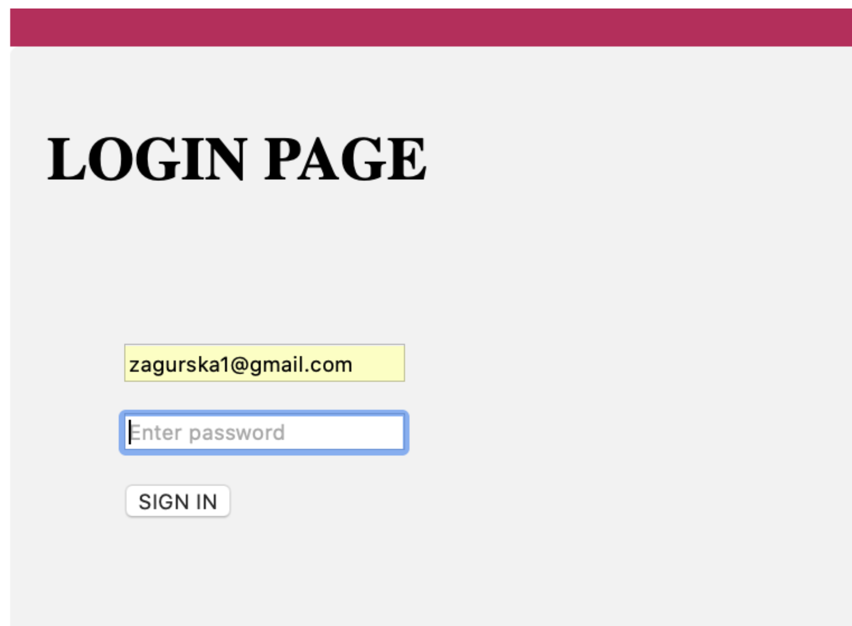


Рис.

Як було описано у розділі розробки серверної частини, завдяки функціоналу сторінки користувача має можливість створювати картки, додавати опису до карт, переміщати карти між собою та виставляти довільний порядок карт в рамках екрану.

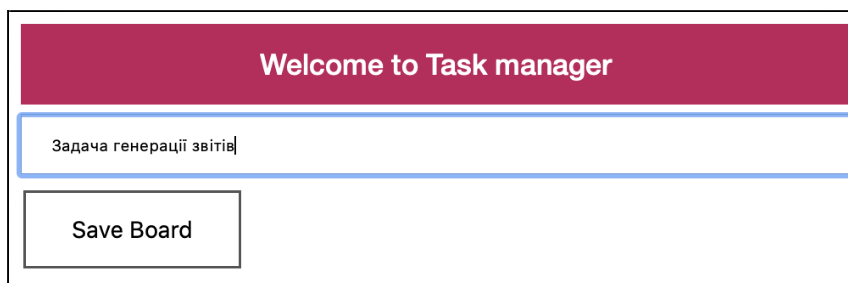


Рис.

На наступних трьох рисунках продемонстровано позиціонування карт у стандартному вигляді (одна за одною) та у довільному вигляді відносно бажання користувача. Як вже було наголошено, реалізація саме цього алгоритму сортування карт була найважчим етапом розробки.

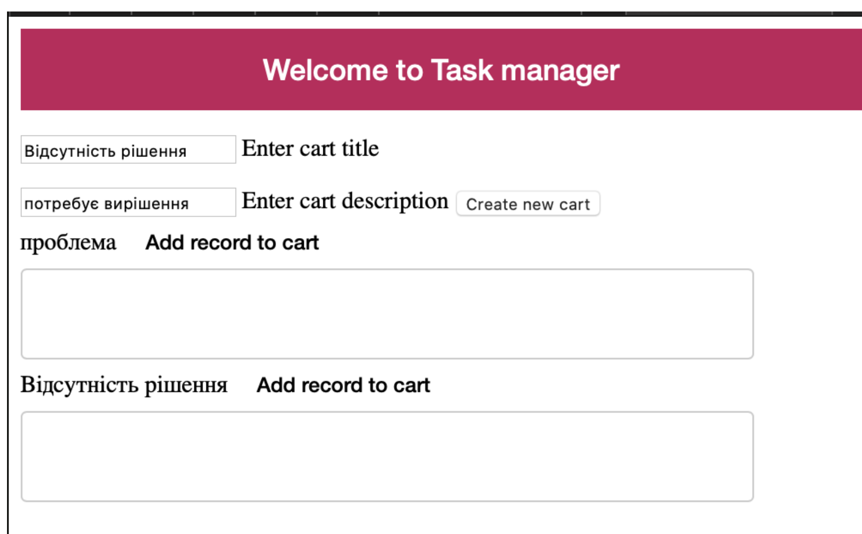


Рис.

Welcome to Task manager

Відсутність рішення Enter cart title

потребує вирішення Enter cart description Create new cart

Відсутність рішення Add record to cart

проблема Add record to cart

Рис.

Welcome to Task manager

Відсутність рішення Add record to cart

проблема Add record to cart

Рис.

Усі картки відносно часу створення задач відображаються на головній сторінці користувача у послідовності зверху вниз.

Кожна картка має дві базових кнопки для видалення чи введення додаткової інформації з приводу задач користувача, що він здійснює(рис.) .

Welcome to Task manager

Задача генерації звітів

Delete Board

Open board

Задача генерації звітів

Save Board

Рис.

Enter title

Enter description

Enter comment

Deadline

Save record

Рис.

На наступному зображенні можна побачити реалізацію даної опції на стороні клієнта.

```
  ngOnInit(): void {
    this.getAll();
  }

  private getAll() {
    this.boardRestService.getAll().subscribe(resp => {
      this.boardDto = resp.map(resp => new BoardDto(resp));

    });
  }

  saveBoard() {
    let boardDto = new BoardDto();
    boardDto.title = this.boardName;
    this.boardRestService.saveBoard(boardDto).subscribe(() => {
      this.getAll();
    });
  }

  deleteBoard(boardId:number){
    this.boardRestService.deleteBoard(boardId).subscribe(() => {
      this.getAll();
    });

  }

  getOne(boardId : number){
    this.router.navigate(['/' + boardId]);
  }
}
```

Рис.

Окрім функціоналу, з яким користувач взаємодіє було також додано опцію відображення статусу після взаємодії користувача із кнопками «видалити» чи «створити».

Рис.

Висновки до розділу 3

У даному розділі було докладно описано схему архітектури шаблону MVC в рамках роботи фреймворку Angular а також описано спосіб прив’язки та взаємодії контролерів клієнтської частини із контролерами серверної частини.

Продемонстровано реалізацію інтерфейсу користувача засобами фреймворку Angular, що реалізовані у вигляді зв’язок компонент.

ВИСНОВКИ

В результаті розробки системи менеджменту задач «Task manager» було розроблено повноцінний додаток, що представляє собою клієнт-серверну взаємодію по типу « тонкий клієнт – товстий сервер».

Додаток представляє собою повноцінні та незалежні модулі клієнтської та серверної частини. У системі впроваджено можливість авторизації користувача та функціонал для запису та менеджменту проектної інформації.

В ході роботи було проаналізовано типи архітектур та сучасні патерни проектування система.

У роботі докладно описано, яким чином розроблялось програмне рішення, та з яких частин воно складається.

Розроблено програмне забезпечення, що надає можливість здійснювати менеджмент задач.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Andrew S. Tanenbaum, David J. Wetherall, Computer Networks 5th Edition,*2010.– 960 с.
2. *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools – Iuliana Cosmina,*2017. – 849 с.
3. Микросервисные паттерны проектирования [Electronic resource]. Access mode: https://habr.com/ru/company/piter/blog/275633/ .
4. *angular.io* [Електронний ресурс]. – Режим доступу https://angular.io
5. *docker.com* [Електронний ресурс]. – Режим доступу: https://docs.docker.com/v17.12/
6. *Spring in Action. Fifth Editi on – Craig Walls October 2018 Publisher: Manning Publications*, *520 pages printed in black & white,*2018. – 520 с.
7. *Clean Code: A Handbook of Agile Software Craftsmanshi Paperback,Publisher: Prentice Hall; 1 edition,English,*2008. – 464 с.
8. Java серверные приложения / Равиль Мухамедзянов. – М.:СЛОН-Р, 2010. –336 с.
9. Кей Хорстман, Java – Библиотека профессионала, десятое издание, 2016. – 864 с.

ДОДАТОК А

Код програми

```
package com.taskmanager.controller;
import com.taskmanager.dto.BoardDto;
import com.taskmanager.service.BoardService;
import org.springframework.web.bind.annotation.*;

import java.util.List;
@RestController
@RequestMapping(value = "/api/v1/board")
public class BoardController {
    private final BoardService boardService;
    public BoardController(BoardService boardService) {
        this.boardService = boardService;
    }

    @PostMapping
    @CrossOrigin
    public void saveBoard(@RequestBody BoardDto boardDto){
        boardService.saveBoard(boardDto);
    }
    @GetMapping
    @CrossOrigin
    public List<BoardDto> getAll()
    {
        return boardService.getAll();
    }
    @DeleteMapping("{boardId}")
    @CrossOrigin
    public void deleteBoard(@PathVariable Long boardId){
```

```
        boardService.deleteBoardById(boardId);
    }
    @GetMapping("{boardId}")
    public BoardDto getOne(@PathVariable Long boardId){
        return boardService.getById(boardId);
    }

}
```

ДОДАТОК Б

Код програми

```
package com.taskmanager.controller;
import com.taskmanager.dto.CartDto;
import com.taskmanager.service.CartService;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping(value = "/api/v1/cart")
public class CartController {

    private final CartService cartService;

    public CartController(CartService cartService) {
        this.cartService = cartService;
    }
@GetMapping("{boardId}")
    @CrossOrigin(origins = "http://localhost:4200")
    public List<CartDto> getAllByBoardId(@PathVariable Long boardId){
        return cartService.getAllByBoardId(boardId);
    }
    @PostMapping("/{boardId}")
    @CrossOrigin(origins = "http://localhost:4200")
    public CartDto createCart(@PathVariable Long boardId,
                              @RequestBody CartDto cartDto){
        return cartService.saveNewCart(cartDto, boardId);
    }
}
```

ДОДАТОК В

Код програми

```
package com.taskmanager.controller;

import com.taskmanager.dto.CartRecordChangeCartDto;
import com.taskmanager.dto.CartRecordDto;
import com.taskmanager.dto.CartRecordQuery;
import com.taskmanager.service.CartRecordService;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/v1/cart-record")
public class CartRecordController {

    private final CartRecordService cartRecordService;

    public CartRecordController(CartRecordService cartRecordService) {
        this.cartRecordService = cartRecordService;
    }

    @PostMapping("{cartId}")
    @CrossOrigin(origins = "http://localhost:4200")
    public CartRecordDto saveCartRecord(@PathVariable Long cartId,
                                   @RequestBody CartRecordDto cartRecordDto){
        return cartRecordService.saveCartRecord(cartRecordDto, cartId);
    }

    @PutMapping("{cartId}")
    @CrossOrigin(origins = "http://localhost:4200")
    public CartRecordDto updateCartRecord(@PathVariable Long cartId,
```

```
                                @RequestBody CartRecordDto cartRecordDto){

        return cartRecordService.saveCartRecord(cartRecordDto, cartId);
    }

    @GetMapping("{cartRecordId}")
    public Long getAllByCartIdRecord(@PathVariable Long cartRecordId){
        return cartRecordService.countAllByCartId(cartRecordId);
    }

    @PutMapping("/change-order")
    @CrossOrigin(origins = "http://localhost:4200")
    public void changeOrder(@RequestBody CartRecordQuery cartRecordQuery){
        cartRecordService.changeRecordsOrder(cartRecordQuery.getCartRecordId(),
            cartRecordQuery.getChangeIndex(), cartRecordQuery.getCartId());
    }

    @PutMapping("/change-cart")
    @CrossOrigin(origins = "http://localhost:4200")
    public void changeCart(@RequestBody CartRecordChangeCartDto
cartRecordChangeCartDto){
        cartRecordService.changeCart(cartRecordChangeCartDto);
    }
}
```

ДОДАТОК Г

Код програми

```
import { Component, OnInit, Injectable } from '@angular/core';
import {UserLoginDto} from '../rest/login/userLoginDto';
import {LoginRestService} from '../rest/login/login-rest.service';
import {Router} from '@angular/router';
import {MatSnackBar} from '@angular/material/snack-bar';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.scss'],
  providers: [LoginRestService]
})
export class LoginComponent implements OnInit {

  constructor(private loginService: LoginRestService,
              private router: Router,
              public snackBar: MatSnackBar) {
  }

  params: UserLoginDto = new UserLoginDto();

  ngOnInit(): void {
  }

  login() {
    this.loginService.loginUser(this.params).subscribe(response => {
      const responseCode = response.responseCode;
      if (responseCode === 403 || responseCode === 400) {
```

```
this.router.navigate(['/']);
this.snackBar.open("Email or password incorrect", null, {
  duration: 2000,
});}
else {
this.router.navigate(['/board']);
this.snackBar.open("Welcome!", null, {
  duration: 2000,
});
}
});
}

}
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {LoginRoutingModule} from './login.routing';
import {RouterModule} from "@angular/router";
import {LoginComponent} from "./login.component";
import {FormsModule} from "@angular/forms";
import {MatButtonModule, MatCardModule, MatInputModule, MatListModule,
MatToolbarModule} from "@angular/material";
import {HttpClientModule} from "@angular/common/http";
import {RestModule} from "../modules/rest.module";
import {MatSnackBar, MatSnackBarModule} from "@angular/material/snack-bar";


@NgModule({
 exports: [
  LoginComponent
```

```
  ],
  declarations: [
    LoginComponent
  ],
  imports: [
    CommonModule,
    LoginRoutingModule,
    RestModule,
    HttpClientModule,
    MatButtonModule,
    MatCardModule,
    MatInputModule,
    MatListModule,
    MatToolbarModule,
    FormsModule,
    RouterModule,
    MatSnackBarModule
  ]
})
export class LoginModule { }
```

ДОДАТОК Ґ

```
import { Component, OnInit } from '@angular/core';
import {RegistrationDto} from "../rest/registration/registration.dto";
import {RegistrationRestService} from "../rest/registration/registration-rest.service";
import {Router} from "@angular/router";

@Component({
  selector: 'app-registration',
  templateUrl: './registration.component.html',
  styleUrls: ['./registration.component.scss'],
  providers: [RegistrationRestService]
})
export class RegistrationComponent implements OnInit {

  title = 'frontend';
  params: RegistrationDto = new RegistrationDto();

  constructor(private registrationService: RegistrationRestService,
              private router: Router) {
  }

  ngOnInit(): void {
  }

  saveUser() {
    this.registrationService.saveUser(this.params).subscribe(() => {
      this.router.navigate(['/login']);
    });
  }
}
```

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {RegistrationRoutingModule} from "./registration.routing";
import {RouterModule} from "@angular/router";
import {RegistrationComponent} from "./registration.component";
import {FormsModule} from "@angular/forms";
import {MatButtonModule, MatCardModule, MatInputModule, MatListModule, MatToolbarModule} from "@angular/material";
import {HttpClientModule} from "@angular/common/http";
import {RestModule} from "../modules/rest.module";


@NgModule({
  exports: [
    RegistrationComponent
  ],
  declarations: [
    RegistrationComponent
  ],
  imports: [
    CommonModule,
    RegistrationRoutingModule,
    RestModule,
    HttpClientModule,
    MatButtonModule,
    MatCardModule,
    MatInputModule,
    MatListModule,
    MatToolbarModule,
    FormsModule,
```

```
    RouterModule,
  ]
})
export class RegistrationModule { }
```