Лабораторна робота №3

### Створення репозиторіїв та сервісів

Кішкін Олександр.

8.1212

**1. Виконати клонування та запуск проекту:** <https://bitbucket.org/munspel/ex2-todo-app-services/src/master/>

Розпаковали архів.

Для створення локального Docker-образа с ім’ям vue-lab3 и тегом latest створили наступні кроки:

1. **Створили Dockerfile:** Створили текстовий файл с ім’ям Dockerfile (без розширення) в кореневий каталог нашого проекту. Dockerfile визначає, який образ буде створено. Створили такий Dockerfile для Vue.js додатку:

Dockerfile

 # Використовуйте образ Node.js в якості базового образа

FROM node:18

# Встановите робочий каталог всередині контейнера

WORKDIR /app

# Скопіюйте файли package.json и package-lock.json и встановите залежності

COPY package\*.json ./

RUN npm install

# Скопіюйте решту файлів додатків

COPY . .

# Опублікуйте порт, який використовує ваша програма  
EXPOSE 8080

# Команда для запуска додатку

CMD ["npm", "run", "serve"]

 Зі**брали Docker-образ:** В командному рядку чи в терміналі перейшли в директорію, де знаходиться наш Dockerfile, і виконали наступну команду, щоб зібрати Docker-образ:

 docker build -t vue-lab3:latest .

Де:

* -t вказує ім’я і тег образа (vue-lab3:latest).
* . означає поточний каталог, де знаходиться Dockerfile.

 **Дочеклися завершения сборки:** Docker будє виконувати кроки, вказані в Dockerfile, щоб створити образ. Це може заняти дещо часу в залежності від ускладнення нашого додатку і інтернет-з’єднання.

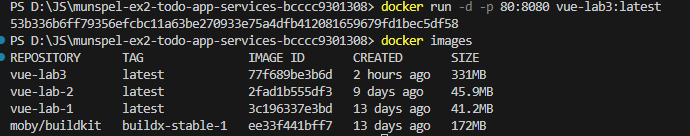
 **Перевірили створений образ:** Мы можемо перевірити, що образ vue-lab3:latest   
бил успішно створено шляхом виконання команди:

docker images

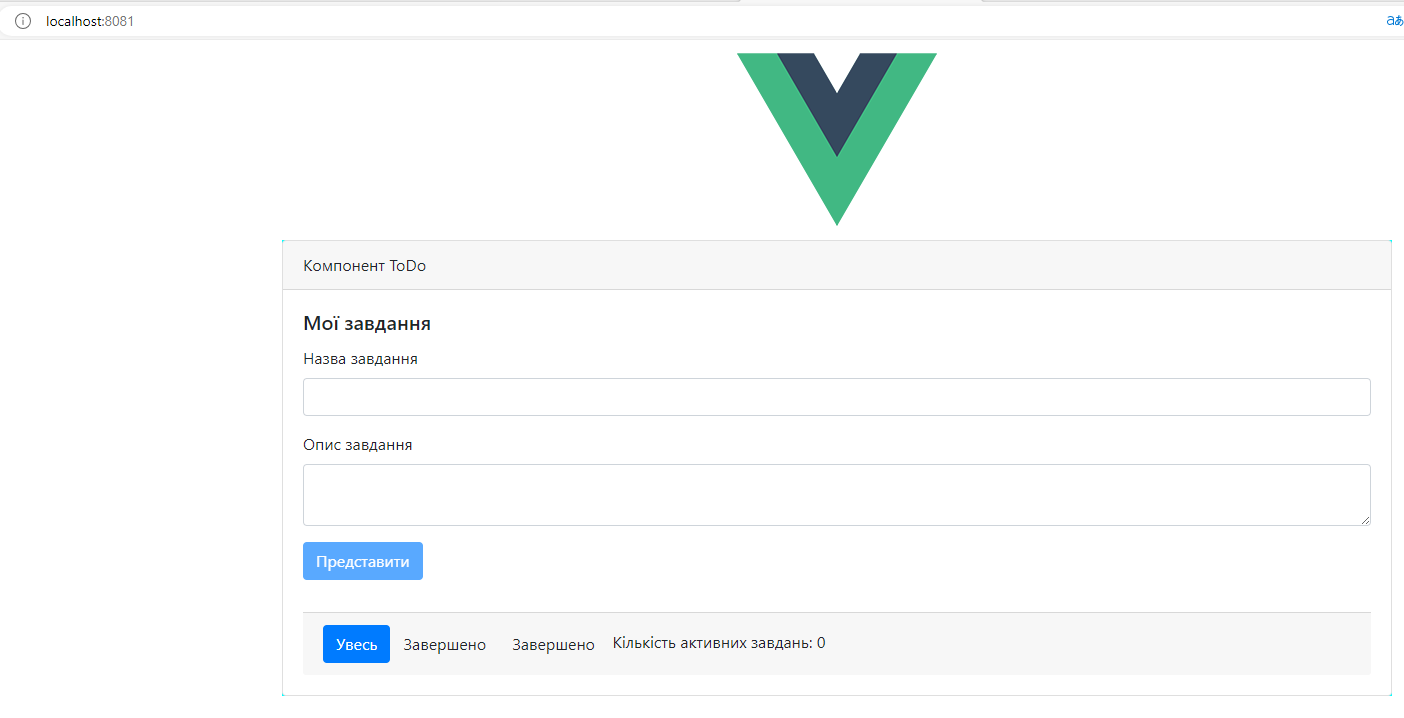
Це покаже список всіх доступних образів, і ми побачимо наш новий образ  
 vue-lab3 с тегом latest.

Ми створили локальний Docker-образ с ім’ям vue-lab3:latest, и мы можемо запустить контейнер, використовуя команду docker run, як ми робили раніше:

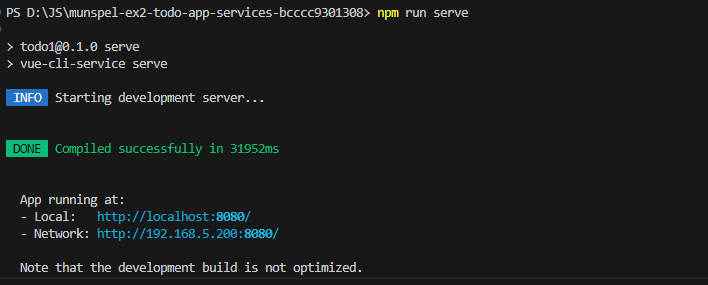
docker run -d -p 80:8080 vue-lab3:latest



Запустили на виконання командою:  
 docker run -d -p 8081:8080 vue-lab3:latest  
Відкрили в браузері по адресу: <http://localhost:8081/>



Виконали команду для запуску проєкту:



1. **Проаналізувати код з попереднього пункту та додаткові матеріали  виконати (використовуючи TypeScript):**

Код представляє Vue.js компонент для створення і управління задачами (to-do list).   
  
Розберемо його по частинам:

1. **Шаблон (Template)**:
   * Основний контейнер компонента має клас "todo".
   * Всередені нього знаходиться карточка ("card"), що складається із заголовка ("card-header"), тела ("card-body") и підвал ("card-footer").
   * В тілі карточки розташована форма для вводу нової задачі, а також список задач.
   * В формі є поля для вводу заголовка и описання задачі, а також кнопка "Submit" для додавання нової задачі.
2. **Скрипт (Script)**:
   * Імпортується модель "Task" із файлу "../models/task".
   * Експортується Vue.js компонент с ім’ям "ToDo".
   * В компоненте визначаються наступні дані:
     + model: об’єкт задачі, з яким звязиваються поля вводу форми.
     + taskList: масив задач, який буде відображатися в списку.
     + filterStatus: рядок, використанням для фільтрації задач по статусу (завершене, незавершене).
     + loading: флаг для відслідковування стану загрузки даних.
   * В методе mounted виконується загрузка задач із сервісу і флаг loading установлюється в true в час загрузки и в false після завершення завантаження.
   * В методе submit виконується додавання нової задачі. Вона зберігається через сервіс, потім додається в масив taskList, и об’єкт model скидаєтеся до нової пустої задачі.
   * В блоку watch відслідковуєтся змінення змінної message, но в коді відсутня змінна message, так що цей блок не виконує ніяких дій.
   * В блоку computed визначені обчислені властивості:
     + messageLength: обчислює дліну рядка, представлений змінній message (яка відсутня в даному контексте).
     + isValid: перевіряє, що model.title і model.description не є пустими рядками.
     + filteredTaskList: фільтрує задачі в масиві taskList в залежності від значення filterStatus.
3. **Стили (Style)**:
   * Стилі обмеженні область компонента с атрибутом "scoped" и встановлюють фоновий колір для елемента с класом "todo".

Цей компонент створює інтерфейс для вводу, відображення и фільтрації задач (to-do items). Коли користувач вводить нову задачу и натискає "Submit", вона зберігається і відображається в списку задач. Користувач також може фільтровати задачі по їх статусу (завершене, незавершене).

* **Розробити типи сутностей відповідно до потреб попередньої роботи**

<div class="form-group">  
 <label for="task-type">Task type</label>  
 <select v-model="model.type" class="form-control" id="task-type">  
 <option value="urgent">Невідкладна</option>  
 <option value="important">Важлива</option>  
 <option value=" ordinary">Звичайна</option>  
 <option value=" insignificant">Незначна</option>  
 </select>  
</div>

* **Розробити інтерфейс репозиторію для виконання операцій (Create, Read, Update,  Delete)**

export interface IRepository<T> {  
 // Створення нового запису (Create)  
 save(model: T): Promise<T>;  
 // Отримання запису за ідентифікатором (Read)  
 find(id: number): Promise<T | null>;  
 // Отримання всіх записів (Read)  
 fetch(): Promise<T[]>;  
 // Оновлення існуючого запису за ідентифікатором (Update)  
 update(id: number, model: T): Promise<T>;  
 // Видалення запису за ідентифікатором (Delete)  
 delete(id: number): Promise<void>;  
}

* **Реалізувати інтерфейс репозиторію для роботи з локальним сховищем (LocalStorage)**

Методи **Create –** реалізован метод SAVE, Додає новий запис у репозиторій і повертає доданий запис.

**Read –** FIND Знаходить запис за ідентифікатором і повертає його або null, якщо запис не знайдено.

В файлі ToDo.js в TodoLocalStorageRepository додаємо метод **Delete.**

Цей метод виконує наступні дії:

1. Отримує поточний список елементів з локального сховища за допомогою this.\_\_getItems().
2. Використовує метод findIndex() для знаходження індексу запису з вказаним id.
3. Якщо запис знайдений (індекс відмінний від -1), він видаляється з масиву items за допомогою splice(), а потім оновлений масив зберігається в локальному сховищі за допомогою localStorage.setItem.
4. Метод повертає Promise, який завжди резолвиться навіть, якщо запис з вказаним id не був знайдений.

* //---------------------------------
* async delete(id) {
* return new Promise((resolve) => {
* let items = this.\_\_getItems();
* const index = items.findIndex((item) => item.id === id);
* if (index !== -1) {
* items.splice(index, 1);
* localStorage.setItem("todos", JSON.stringify(items));
* resolve();
* } else {
* resolve();
* }
* });
* }
* async update(id, model) {
* return new Promise((resolve) => {
* let items = this.\_\_getItems();
* const index = items.findIndex((item) => item.id === id);
* if (index !== -1) {
* // Замінюємо існуючий запис на новий model
* items[index] = model;
* localStorage.setItem("todos", JSON.stringify(items));
* resolve(model);
* } else {
* // Якщо запис з вказаним id не знайдено, повертаємо null або генеруємо помилку
* resolve(null);
* }
* });
* }
* //---------------------------------

В файлі ToDo.js в TodoLocalStorageRepository додаємо метод Update.

Цей метод виконує наступні дії:

1. Отримує поточний список елементів з локального сховища за допомогою this.\_\_getItems().
2. Використовує метод findIndex() для знаходження індексу запису з вказаним id.
3. Якщо запис знайдений (індекс відмінний від -1), він замінює існуючий запис на новий model в масиві items.
4. Оновлений масив зберігається в локальному сховищі за допомогою localStorage.setItem.
5. Метод повертає Promise, який резолвиться новим model у разі успішного оновлення. Якщо запис з вказаним id не був знайдений, метод може повертати null або генерувати помилку, залежно від нашого вибору.

**3. Розмістити проект у системі контролю версій (github.com, bitbucket.org) з іменем ропозиторію mjsf-mag-lab3.**

Для розміщення проекту у системі контролю версій виконали наступні кроки:

Увійшли до облікового запису GitHub.

Натиснули на значок "+" в правому верхньому кутку екрана та вибрали "New repository" (Новий репозиторій).

Заповнили інформацію про репозиторій:

Repository name (Ім'я репозиторію): mjsf-mag-lab3

Натиснули кнопку "Create repository" (Створити репозиторій).

Після створення репозиторію, побачили URL-адресу репозиторію, яку можна використовувати для клонування репозиторію на наш комп'ютер.

Перейшли до локального каталогу нашого проекту на комп'ютері, де розташований код нашого проекту.

Відкрили термінал (або командний рядок) і перейшли до цього каталогу.

 Використали команди Git, щоб ініціалізувати новий Git-репозиторій, додати всі файли до коміту, створити коміт та пов'язати репозиторій на GitHub з нашим локальним репозиторієм.

git init  
git add .  
git commit -m "Перший коміт"  
git branch -M main

перший раз

git remote add origin <https://github.com/KishkinA/mjsf-mag-lab3.git>

настиупні рази

git remote set-url origin https://github.com/KishkinA/mjsf-mag-lab3.git

git push -u origin main

1. **Відправити посилання на репозиторій у якості звіту з виконаної роботи.**

**https://github.com/KishkinA/mjsf-mag-lab3**

**Контрольні питання**

1. **Що таке паттерн репозиторій і для чого він використовується в розробці програмного забезпечення?**

Паттерн репозиторій (Repository Pattern) є одним з шаблонів проектування, який використовується в розробці програмного забезпечення для розділення логіки доступу до даних від інших частин програми. Головна мета цього паттерна - надати абстракцію для роботи з базою даних або іншими джерелами даних, щоб інші частини програми не залежали від конкретних деталей роботи з даними.

Основні компоненти паттерна репозиторій включають в себе:

1. **Репозиторій (Repository)**: Це інтерфейс або клас, який визначає методи для звернення до даних, такі як додавання, оновлення, видалення і запити на вибірку даних. Репозиторій приховує конкретний спосіб доступу до даних, надаючи абстракцію.
2. **Реалізація репозиторію (Repository Implementation)**: Це конкретний клас, який реалізує інтерфейс репозиторію. Він містить реалізацію методів доступу до даних, як правило, взаємодіє з базою даних або іншим джерелом даних.
3. **Сутності (Entities)**: Це об'єкти, які представляють дані, які зберігаються в базі даних. Вони можуть бути використані як об'єкти даних, з якими працює репозиторій.

Причини використання паттерна репозиторій в розробці програмного забезпечення включають:

1. **Розділення відповідальностей**: Паттерн дозволяє відокремити логіку доступу до даних від логіки додатку. Це полегшує розширення та зміну методів доступу до даних без впливу на інші частини програми.
2. **Підтримка множинних джерел даних**: Репозиторій може бути налаштований для роботи з різними джерелами даних, такими як бази даних, веб-служби, зовнішні API тощо, без змін в інших частинах програми.
3. **Тестування**: Використання репозиторію полегшує одиниць тестування, оскільки можна легко створити моки (mocks) або підроблені об'єкти для тестування логіки додатку без доступу до реальних даних.
4. **Покращення читабельності коду**: Розділення логіки доступу до даних від інших частин програми може полегшити розуміння та обслуговування коду.

Паттерн репозиторій є корисним інструментом для дотримання принципів чистої архітектури та покращення модульності та розширюваності програмного забезпечення.

1. **Які компоненти включаються в реалізацію паттерну репозиторій?**

Реалізація паттерну репозиторій включає кілька компонентів, які допомагають взаємодіяти з джерелами даних і надають абстракцію для доступу до даних. Основні компоненти в реалізації паттерну репозиторій включають:

1. **Репозиторій (Repository Interface або Abstract Repository)**: Це інтерфейс або абстрактний клас, який визначає методи для доступу до даних, такі як додавання, оновлення, видалення і запити на вибірку даних. Цей інтерфейс повинен бути загальним для всіх типів сутностей, з якими працює репозиторій.
2. **Реалізація репозиторію (Repository Implementation)**: Це конкретний клас, який реалізує інтерфейс або абстрактний клас репозиторію. Цей клас містить реалізацію методів, які взаємодіють з конкретним джерелом даних, таким як база даних. Реалізація репозиторію може також включати логіку перетворення даних між форматами, якщо це необхідно.
3. **Сутності (Entities)**: Це об'єкти, які представляють дані, які зберігаються в базі даних або іншому джерелі даних. Сутності зазвичай відображають структуру даних і містять поля, які відображають стовідношення між різними об'єктами даних.
4. **Фабрика сутностей (Entity Factory, Entity Factory Method)**: Інколи використовується фабрика сутностей для створення нових об'єктів сутностей перед додаванням їх до репозиторію. Це може бути корисним, якщо існують специфічні правила або додаткові дії, пов'язані з створенням сутностей.
5. **Модель даних (Data Model)**: Це представлення даних в програмі, яке відображає структуру та зв'язки між сутностями. Модель даних використовується для взаємодії з репозиторієм та для передачі даних між компонентами програми.
6. **Класи запитів (Query Classes)**: Деякі реалізації паттерну репозиторій можуть включати класи або об'єкти запитів, які дозволяють створювати складні запити до джерела даних через репозиторій.

Загальна ідея реалізації паттерну репозиторій полягає в тому, щоб ізолювати логіку доступу до даних від інших частин програми і надати абстракцію, яка дозволяє замінювати джерела даних без впливу на користувачів цих даних.

1. **Яка роль репозиторію в паттерні репозиторій?**

Роль репозиторію (Repository) в паттерні репозиторій полягає в наданні абстракції для доступу до даних і управління сутностями (entities) у програмі. Репозиторій виконує кілька важливих функцій:

1. **Ізоляція даних від користувача**: Репозиторій приховує деталі доступу до даних, такі як SQL-запити до бази даних, від інших частин програми. Це означає, що решта програми може взаємодіяти з репозиторієм, не знаючи, з яким джерелом даних він працює. Це полегшує зміну джерела даних без змін в інших частинах програми.
2. **Управління сутностями**: Репозиторій включає методи для додавання, оновлення, видалення та запитів на вибірку сутностей (entities). Він відповідає за створення, читання, оновлення та видалення сутностей з джерела даних, а також за виконання пошукових запитів.
3. **Підтримка мови бізнес-логіки**: Репозиторій може включати логіку, пов'язану з обробкою даних перед збереженням або після отримання їх із джерела даних. Наприклад, він може валідувати дані перед їх збереженням, перетворювати дані в формат, зрозумілий користувачеві, або виконувати інші дії, пов'язані з бізнес-логікою програми.
4. **Тестування**: Репозиторій дозволяє легко створювати модульні тести, оскільки його можна підробити або підмінити під час тестування. Це допомагає визначити, чи правильно взаємодіє програма з джерелом даних без реального з'єднання з базою даних або іншими джерелами.
5. **Підтримка множинних джерел даних**: Репозиторій може бути налаштований для роботи з різними джерелами даних, такими як реляційні бази даних, NoSQL-сховища, веб-служби, зовнішні API тощо, завдяки чому різні компоненти програми можуть співпрацювати з однаковим інтерфейсом репозиторію.

Узагальнюючи, репозиторій відокремлює логіку доступу до даних від решти програми, надаючи абстракцію для роботи з даними і спрощуючи розробку, тестування та обслуговування програмного забезпечення.

1. **Яка роль моделі в паттерні репозиторій?**

Модель в контексті паттерну репозиторій відіграє роль представлення даних або сутностей (entities), які зберігаються в базі даних або іншому джерелі даних. Модель визначає структуру та формат цих даних, а також зв'язки між різними сутностями. Роль моделі в паттерні репозиторій включає такі аспекти:

1. **Представлення даних**: Модель визначає, як дані повинні бути представлені у програмі. Це може включати в себе властивості (поля) та методи для роботи з цими даними. Наприклад, якщо ви маєте сутність "Користувач" (User), модель цієї сутності визначає, які поля є в користувача (наприклад, ім'я, електронна пошта, пароль) і які методи можуть бути використані для роботи з цією сутністю (наприклад, валідація даних, обробка подій тощо).
2. **Валідація і перетворення даних**: Модель може містити логіку для валідації та перетворення даних перед їх збереженням у базі даних або після отримання їх із джерела даних. Наприклад, модель може перевіряти, чи всі обов'язкові поля заповнені, чи дані відповідають певним правилам (наприклад, унікальність електронної адреси користувача) та виконувати інші перетворення даних.
3. **Зв'язки між сутностями**: Модель може визначати зв'язки між різними сутностями. Наприклад, якщо в програмі є сутності "Користувач" і "Замовлення", модель може визначати, як ці дві сутності пов'язані (наприклад, один до багатьох: один користувач може мати багато замовлень).
4. **Зручний доступ до даних**: Модель надає зручний інтерфейс для отримання та редагування даних сутностей. Використання моделі спрощує роботу з даними і дозволяє зосередитися на бізнес-логіці програми.
5. **Підтримка декількох джерел даних**: Модель може бути налаштована так, щоб підтримувати роботу з різними джерелами даних (наприклад, реляційною базою даних та NoSQL-сховищем) без змін в інших частинах програми.

Загалом, роль моделі в паттерні репозиторій полягає в тому, щоб представляти структуру та формат даних, які зберігаються в базі даних або інших джерелах даних, і надавати зручний інтерфейс для роботи з цими даними. Модель допомагає забезпечити єдність у представленні даних і спрощує розробку та обслуговування програми.

1. Як паттерн репозиторій допомагає відокремити логіку доступу до даних від бізнес-логіки програми?

Паттерн репозиторій (Repository Pattern) допомагає відокремити логіку доступу до даних від бізнес-логіки програми за допомогою декількох ключових механізмів:

1. **Абстракція для доступу до даних**: Репозиторій надає абстракцію для доступу до даних, що означає, що решта програми може взаємодіяти з репозиторієм через загальний інтерфейс, не знаючи, як саме відбувається доступ до даних в середині репозиторію. Це важливо, оскільки це дозволяє ізолювати логіку доступу до даних від інших частин програми і забезпечує велику гнучкість у виборі та зміні джерел даних.
2. **Підтримка розділення відповідальностей**: Репозиторій допомагає відокремити завдання доступу до даних від завдань бізнес-логіки. Бізнес-логіка програми може концентруватися на обробці та маніпулюванні даними, не звертаючи увагу на деталі доступу до даних, що спрощує розробку та обслуговування коду.
3. **Можливість зміни джерела даних без змін в бізнес-логіці**: Репозиторій дозволяє легко замінювати джерело даних, якщо це необхідно. Наприклад, ви можете спроектувати репозиторій для роботи з реляційною базою даних, а потім, якщо вимоги змінилися, замінити його на репозиторій, який працює з NoSQL-сховищем без змін в бізнес-логіці програми. Це зроблює код більш масштабованим і адаптивним.
4. **Підтримка модульного тестування**: Репозиторій полегшує модульне тестування, оскільки його можна легко підробити (застосувати мок-об'єкти) під час тестування. Це дозволяє вам перевіряти коректність бізнес-логіки, не залежачи від реальних джерел даних.
5. **Читабельність коду**: Розділення логіки доступу до даних від бізнес-логіки полегшує читабельність та обслуговування коду, оскільки кожна частина коду відповідає лише за свою відповідальність.

В результаті використання паттерну репозиторій допомагає створити більш модульний, масштабований та гнучкий код, де логіка доступу до даних відокремлена від бізнес-логіки, і зміни у джерелах даних можуть бути внесені з мінімальним впливом на інші частини програми.

1. **Як паттерн репозиторій сприяє тестованості програми?**

Паттерн репозиторій сприяє тестованості програми завдяки декільком ключовим механізмам:

1. **Ізоляція доступу до даних**: Один з основних аспектів паттерну репозиторій - це ізоляція логіки доступу до даних від інших частин програми. Це дозволяє створити мок-об'єкти або підроблені реалізації репозиторіїв для тестування бізнес-логіки без реального з'єднання з базою даних або іншими джерелами даних. Таким чином, ви можете створити тести, які фокусуються на коректності бізнес-логіки, не звертаючи уваги на деталі доступу до даних.
2. **Легкість створення мок-об'єктів**: Так як репозиторій надає абстракцію для доступу до даних, ви можете легко створити мок-об'єкти або підроблені реалізації репозиторіїв, які повертають контрольовані дані для тестування. Це дозволяє вам відокремити тестування бізнес-логіки від реальної бази даних.
3. **Зручність модульного тестування**: Розділення логіки доступу до даних від бізнес-логіки дозволяє створювати модульні тести для окремих компонентів програми. Ви можете тестувати окремі методи бізнес-логіки, не турбуючись про взаємодію з реальними джерелами даних. Це спрощує виявлення та виправлення помилок і підтримує тестову суіту незалежно від реальних даних.
4. **Підтримка автоматизованого тестування**: Завдяки ізоляції доступу до даних та зручним мок-об'єктам, ви можете створювати автоматизовані тести, які можна виконувати автоматично під час розробки. Це допомагає забезпечити стабільність програми під час змін і додавання нового функціоналу.
5. **Можливість швидко виявляти помилки**: Завдяки тестам, які тестують бізнес-логіку програми незалежно від реальних джерел даних, ви можете швидко виявити та виправити помилки без великих зусиль.

У підсумку, паттерн репозиторій полегшує тестування програми, дозволяючи відокремити логіку доступу до даних від бізнес-логіки, створювати мок-об'єкти для тестування та створювати модульні, автоматизовані тести, що сприяє стабільності та надійності програми.

1. **Які переваги повторного використання коду надає використання паттерну репозиторій?**

Використання паттерну репозиторій має кілька переваг, які сприяють повторному використанню коду та поліпшують структуру програми:

1. **Відокремлення доступу до даних від бізнес-логіки**: Репозиторій ізолює логіку доступу до даних від бізнес-логіки програми. Це дозволяє повторно використовувати бізнес-логіку без змін в інших частинах програми, що сприяє більшому повторному використанню.
2. **Загальний інтерфейс для доступу до даних**: Репозиторій надає загальний інтерфейс для роботи з даними, незалежний від конкретного джерела даних (наприклад, бази даних). Це полегшує створення загальних компонентів, які можуть бути використані в різних частинах програми.
3. **Модульність та розширюваність**: Репозиторій дозволяє додавати нові методи та функціональність без впливу на існуючий код. Це зрощує розширення програми та сприяє її модульності.
4. **Підтримка різних джерел даних**: Репозиторій може бути налаштований для роботи з різними джерелами даних, такими як реляційні бази даних, NoSQL-сховища, веб-служби тощо. Це сприяє більшому повторному використанню коду, оскільки одні й ті ж методи можуть бути використані для різних джерел даних.
5. **Підтримка тестування**: Використання репозиторію полегшує модульне тестування, оскільки його можна легко підробити або замінити під час тестування. Це сприяє виявленню помилок та забезпечує більше повторне використання коду під час тестування.
6. **Зниження дублювання коду**: Репозиторій дозволяє зберігати логіку доступу до даних в одному місці, замість дублювання її в різних частинах програми. Це зменшує кількість коду та сприяє підтримці програми.

Загалом, використання паттерну репозиторій сприяє більшому повторному використанню коду та поліпшує якість та структуру програми, зрощуючи її розширення та підтримку.

1. **Як паттерн репозиторій допомагає абстрагувати джерела даних?**

Паттерн репозиторій (Repository Pattern) допомагає абстрагувати джерела даних, такі як бази даних, веб-служби, зовнішні API тощо, шляхом створення загального інтерфейсу для доступу до даних та використання цього інтерфейсу в програмі. Ось як він це робить:

1. **Загальний інтерфейс репозиторію**: Паттерн репозиторій визначає загальний інтерфейс або абстрактний клас для репозиторію, який включає методи для додавання, оновлення, видалення та запитів на вибірку даних. Цей інтерфейс надає єдиний спосіб взаємодії з даними для різних джерел.
2. **Конкретні реалізації репозиторію**: Для кожного конкретного джерела даних (наприклад, бази даних) створюються конкретні реалізації репозиторію, які реалізують інтерфейс репозиторію. Ці реалізації включають логіку доступу до конкретного джерела даних, таку як виконання SQL-запитів або виклик веб-служб.
3. **Сутності (Entities)**: Репозиторій включає сутності, які представляють дані, з якими працює програма. Сутності можуть бути агностичними до джерела даних, і репозиторій відповідає за перетворення сутностей на дані відповідного формату для конкретного джерела даних.
4. **Фабрика сутностей (Entity Factory)**: В реалізаціях паттерну репозиторій може використовуватися фабрика сутностей для створення нових об'єктів сутностей перед додаванням їх до репозиторію. Це допомагає забезпечити, що сутності мають правильну структуру та дані перед збереженням їх у джерелі даних.
5. **Модель даних (Data Model)**: Паттерн репозиторій допомагає визначити модель даних, яка представляє дані в програмі та визначає структуру і зв'язки між сутностями. Ця модель даних допомагає взаємодіяти з репозиторієм та передавати дані між компонентами програми.

Завдяки цим механізмам паттерн репозиторій дозволяє відокремити джерела даних від бізнес-логіки програми та надає можливість легко змінювати або додавати нові джерела даних, не впливаючи на решту програми. Така абстракція сприяє гнучкості та повторному використанню коду.

1. **Як зміна джерела даних впливає на клієнтський код при використанні паттерну репозиторій?**

Однією з головних переваг паттерну репозиторій є те, що він дозволяє змінювати джерело даних без великого впливу на клієнтський код. Це досягається завдяки ізоляції логіки доступу до даних у репозиторії та створенню загального інтерфейсу для доступу до даних. Ось як зміна джерела даних впливає на клієнтський код при використанні паттерну репозиторій:

1. **Зміна реалізації репозиторію**: Якщо ви вирішите змінити джерело даних (наприклад, перейти від реляційної бази даних до NoSQL-сховища), вам потрібно буде створити нову реалізацію репозиторію для нового джерела даних. Однак клієнтський код не буде змінюватися, оскільки він взаємодіє лише з інтерфейсом репозиторію, а не з конкретною реалізацією.
2. **Використання іншої фабрики підключень**: Якщо джерело даних вимагає іншого методу підключення чи автентифікації, ви можете створити нову фабрику підключень для нового джерела даних і використовувати її в новій реалізації репозиторію. Це не вплине на клієнтський код, оскільки клієнти взаємодіють лише з інтерфейсом репозиторію.
3. **Зміна моделі даних**: Якщо структура даних в новому джерелі відрізняється від попереднього, реалізація репозиторію відповідає за перетворення даних між моделлю даних програми і моделлю даних нового джерела. Клієнтський код залишається незмінним, оскільки він взаємодіє лише з моделлю даних програми.
4. **Зміна джерела даних в реальному часі**: Паттерн репозиторій дозволяє змінювати джерело даних в реальному часі без впливу на клієнтський код. Наприклад, ви можете спроектувати програму так, щоб вона автоматично переходила з одного джерела даних на інше в разі відмови чи зміни умов доступу.

Загальна ідея полягає в тому, що клієнтський код взаємодіє з репозиторієм через абстрактний інтерфейс, і реалізація репозиторію відповідає за роботу з конкретним джерелом даних. Це дозволяє легко міняти або доповнювати джерела даних без великих змін в клієнтському коді і сприяє більшій гнучкості та адаптивності програми.

1. **Як паттерн репозиторій допомагає знизити ризик помилок при роботі з даними?**

Паттерн репозиторій допомагає знизити ризик помилок при роботі з даними завдяки декільком ключовим механізмам і підходам:

1. **Ізоляція доступу до даних**: Репозиторій ізолює логіку доступу до даних від інших частин програми, включаючи бізнес-логіку. Це дозволяє зберігати всю логіку доступу до даних в одному місці, і якщо помилки з'являються, вони зазвичай легше виявляються і виправляються, оскільки не потрібно перевіряти кожен виклик до джерела даних у всьому коді.
2. **Модульне тестування**: Паттерн репозиторій полегшує модульне тестування, оскільки реалізація репозиторію може бути підроблена або замінена під час тестування. Це дозволяє створити тести, які перевіряють коректність роботи бізнес-логіки без реального з'єднання з базою даних чи іншими джерелами даних.
3. **Автоматичне тестування**: Більшість логіки доступу до даних у реалізації репозиторію може бути протестована автоматично. Це дозволяє створити набір тестів, які перевіряють різні аспекти роботи репозиторію і забезпечують його стабільність.
4. **Загальний інтерфейс для доступу до даних**: Використання загального інтерфейсу репозиторію робить код більш читабельним і зменшує ризик помилок, оскільки клієнти взаємодіють з репозиторієм через однаковий інтерфейс, незалежно від конкретного джерела даних.
5. **Підтримка транзакцій**: Реалізація репозиторію може включати підтримку транзакцій, що дозволяє забезпечити консистентність даних в деяких випадках, коли це необхідно.
6. **Легкість обслуговування**: Всі зміни та вдосконалення логіки доступу до даних можуть бути здійснені в одному місці, що полегшує обслуговування та підтримку коду.

Загалом, паттерн репозиторій сприяє меншому ризику помилок при роботі з даними завдяки чіткій структурі коду, модульному тестуванню, автоматичному тестуванню та ізоляції логіки доступу до даних від бізнес-логіки. Це допомагає покращити надійність та стабільність програми.