МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Економічний факультет
Кафедра економіко-математичного моделювання та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА/ПРОЄКТ

на здобуття освітнього ступеня бакалавра

на тему: «Розробка модулю групового навчання для освітньої платформи з елементами гейміфікації»

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Луцюк Максим Ігорович

Керівник: *викладач кафедри ЕММІТ*
Красюк Богдан Віталійович

Рецензент: *Front-end Developer "DOODLE", LLC*
Місай Володимир Віталійович

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри економіко-математичного моделювання та інформаційних технологій _____ (проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від «18» травня 2022 р.

Острог, 2022

Міністерство освіти і науки України
Національний університет «Острозька академія»

Факультет: *економічний*
Кафедра: *економіко-математичного моделювання та інформаційних технологій*
Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри економіко-математичного моделювання
та інформаційних технологій

_____ Ольга КРИВИЦЬКА
«____» _____20__ р.

З А В Д А Н Н Я
на кваліфікаційну роботу/проєкт студента

Луцюка Максима Ігоровича

*1. Тема роботи "*Розробка модулю групового навчання для освітньої платформи з елементами гейміфікації"

керівник проєкту Красюк Богдан Віталійович, викладач.

Затверджено наказом ректора НаУОА від 17 жовтня 2022 року №77.

2. Термін здачі студентом закінченої роботи/проєкту: 31 травня 2023 року.

3. Вихідні дані до роботи/проєкту: JetBrains WebStorm, Visual Studio Code, Postman, GitHub, Docker, PostgreSQL, PHP, Nginx, TypeScript, Vue 3.

4. Перелік завдань, які належить виконати: розробити схему бази даних, створити проекти клієнтської та серверної частини, розробити дизайн інтерфейсу, реалізувати функціонал для роботи з курсами, уроками, студентами, завданнями, поданнями завдань, запрошеннями. Провести тестування на доступність інтерфейсу. Розробити механізм взаємодії з API для клієнтської частини. Дослідити наявні способи створення автоматизованого розгортання додатків.

5. Перелік графічного матеріалу: рисунки.

6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Красюк Б. В.	01.12.2022	01.12.2022
2	Красюк Б. В.	01.12.2022	01.12.2022
3	Красюк Б. В.	01.12.2022	01.12.2022

7. Дата видачі завдання: 01.12.2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів	Примітка
1.	Затвердження теми роботи/проєкту	до 31.10.2022.	
2.	Постановка технічного завдання	до 01.11.2022	
3.	Розроблення схема бази даних, створення нових проектів та початкова конфігурація	до 01.12.2023	
4.	Розроблення дизайну клієнтської частини	до 14.02.2023	
5.	Верстка клієнтської частини, створення функціоналу серверної частини	до 14.03.2023	
6.	Інтеграція функціоналу у інтерфейс клієнтської частини	до 01.04.2023	
7.	Перевірка Accessibility клієнтської частини, внесення відповідний змін	до 07.05.2023	
8.	Тестування системи	до 14.05.2023	
9.	Попередній захист кваліфікаційної роботи/проєкту	до 18.05.2023	
10.	Здача кваліфікаційної роботи/проєкту на кафедру	до 31.05.2023	

Студент: _____ Максим ЛУЦЮК

Керівник кваліфікаційної роботи: _____ Богдан КРАСЮК

АНОТАЦІЯ
кваліфікаційної роботи/проєкту
на здобуття освітнього ступеня бакалавра

Тема: *Розробка модулю групового навчання для освітньої платформи з елементами гейміфікації*

Автор: *Луцюк Максим Ігорович*

Науковий керівник: *Красюк Богдан Віталійович, викладач.*

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *56 с., 8 рис., 26 джерел..*

Ключові слова: *онлайн-навчання, веб-сервіс, розроблення платформи, клієнт-серверна архітектура.*

Короткий зміст праці:

Завданням кваліфікаційної роботи/проєкту, було розроблення освітньої онлайн-платформи. Зокрема було розглянуто такі питання як взаємодія з API на клієнтській частині, процес проектування бази даних, переваги використання TypeScript та недоліки обраного стеку технологій й можливі вирішення цих проблем. При розробленні було використано клієнт-серверну архітектуру, що дає змогу в подальшому використати існуюче API для створення, наприклад, мобільного застосунку. Результатом роботи є готова, швидка та проста для подальшого покращення платформа для онлайн-навчання.

The task of the qualification work/project was to develop an online educational platform. In particular, such issues as interaction with the API on the client side, the database design process, the advantages of using TypeScript, the disadvantages of the selected technology stack, and possible solutions to these problems were considered. The development was based on a client-server architecture, which allows the existing API to be used in the future to create, for example, a mobile application. The result is a ready-made, fast, and easy-to-improve online learning platform

_____

ЗМІСТ

ВСТУП

В умовах в яких опинився світ через пандемію COVID-19 різко зріс попит на різноманітні інструменти для виконання буденних справ дистанційно. І однією зі сфер які найбільше потребували переведення в онлайн формат стала сфера освітніх послуг. І хоча на сьогодні карантинні обмеження не є настільки суворими якими вони були на початку пандемії, дистанційний формат навчання став доволі звичним, та став фундаментальним для усієї сфери. Саме тому попит на інструменти які дають змогу покращити ці процеси не зникає, а й росте, особливо зважаючи на інші ситуації.

Тому проблематика розробки навчальної онлайн платформи досі є актуальною. Зважаючи на ці обставини для теми кваліфікаційно роботи було обрано саме розроблення навчальної онлайн платформи.

Процес розроблення такої платформи можна розбити на наступні етапи:

- Аналіз ринку
- Збір вимог
- Планування функціоналу
- Створення концепції
- Створення дизайну платформи
- Розробка серверної та клієнтської частини
- Тестування
- Подальша підтримка продукту

Надалі у цій роботі висвітлені окремі елементи реалізації проекту, які були важливими для створення повноцінної системи.

РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Використані сервіси

1.1.1. Git

Git - це відкрита розподілена система керування версіями, призначена для швидкої та ефективного збереження, відстежування та керування змінами у програмному коді та інших файлах у проєктах різного рівня комплексності.

Використання цієї технології дає змогу розробникам програмного забезпечення одночасно працювати над спільною кодовою базою завдяки можливості ведення паралельних гілок які одночасно зберігають різні стани проекту.

Git може використовуватися як локально, зберігаючи репозиторій на комп’ютері без підключення до мережі інтернет, так і з віддаленими репозиторіями які зберігаються з використанням таких сервісів як GitHub (див. 1.1.2) чи GitLab.

На сьогодні є однією з найпопулярніших системою контролю версій, більшість інтегрованих середовищ розробки мають вбудовані інструменти для спрощення використання Git.

1.1.2. GitHub

GitHub - це вебсервіс, для зберігання та кооперативної роботи з віддаленими репозиторіями для системи контролю версій Git (див. 1.1.1).

Завдяки таким функціям як Pull Request’s чи Discussions, GitHub часто використовується для зберігання репозиторіїв відритого програмного забезпечення, адже це вирішує великий перелік проблем пов’язаних з підтримкою таких проектів:

- Issues – система контролю вад програмного забезпечення. В залежності від потреб та встановлених домовленостей в репозиторії можуть використовуватися також для створення запитів на додавання функціональності чи для розподілу завдань між розробниками. Додаткові

можливості такі як присвоювання користувачам, пріоритезація, система тегів, обговорення та багато інших роблять цей інструмент гнучким та корисним.

- Actions – це інструмент для автоматизації процесів що повинні запускатися за певних подій які відбуваються в репозиторії, наприклад тестування, збирання та публікація нової версії NPM пакету після чергового релізу.
- Pull Requests – механізм пропозицій внесення змін у відкриті репозиторії. Дозволяє зовнішнім користувачам вносити правки у кодову базу у клонованих репозиторіях та пропонувати авторам основного репозиторію прийняти ці правки у основний репозиторій.

Завдяки тісній інтеграції усіх інструментів між собою, використання цього сервісу позбавляє розробників виконання багатьох рутинних дій зо економить час та дозволяє сфокусуватися на розробці програмного забезпечення.

1.1.3. Api-клієнт Postman

Postman – це програмне забезпечення для розробки та тестування API. Цей інструмент реалізовує функціонал для створення та відправки запитів на API, створення автоматизованих тестів, генерування документації, аналіз відповідей сервера та інше.

Postman підтримує різні формати API, такі як REST API, GraphQL, gRPC, дозволяє працювати із заголовками запитів, використовувати змінні оточення та різні варіанти авторизації запитів (Рис. 1.1).

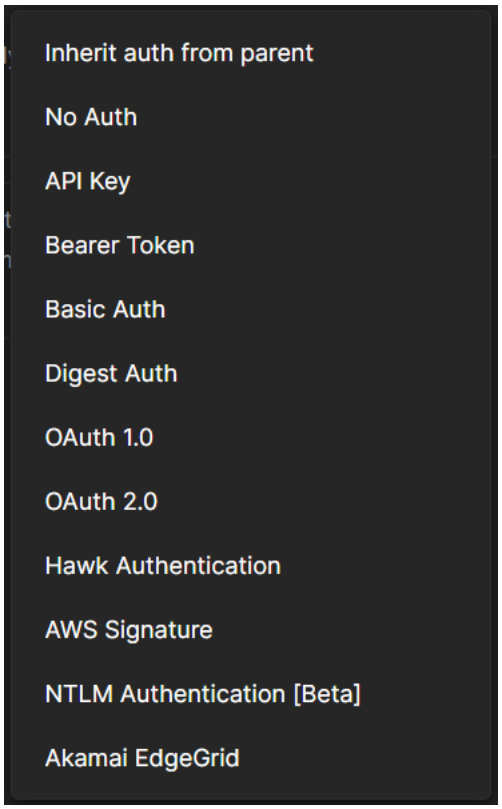

Рис. 1.1. Доступні варіанти авторизації запитів
Джерело: Фото інтерфейсу Postman

1.2. Мови програмування, бібліотеки, технології серверної частини

1.2.1. PHP 8.1

PHP - скриптова мова загального призначення з динамічною типізацію, яка створювалася для використання, здебільшого, у веб-розробці.

Завдяки спрощеному (у порівнянні з деякими іншими мовами програмування) синтаксису, наявності великої кількості готових бібліотек та підтримці на переважній більшості веб-серверів, ця мова є простішою для новачків.

1.2.2. Laravel

Laravel - це відкритий фреймворк написаний на PHP з використанням бібліотек з екосистеми Symfony. Використовується для створення веб застосунків заснованих на концепції MVC (Model-View-Controller).

Для спрощення розробки Laravel пропонує перелік інструментів, таких як Eloquent ORM, Blade-шаблони, командний інструмент artisan та цілий перелік додаткових пакетів.

1.2.3. Apiato (Porto SAP)

Apiato (Porto SAP) - це реалізація архітектурного шаблону Porto для Laravel від автора цього ж шаблону. Цей пакет реалізовує запропоновані в архітектурі рішення, а також типовий основний функціонал веб застосунків, такий як авторизація-автентифікація, система прав та ролей користувачів, серіалізація об’єктів для відповідей API, та інше.

1.2.4. PostgreSQL

PostgreSQL - це об’єктно-реляційна система управління базою даних (Object-Relational Database Management System, ORDBMS). Вона розроблялася як високонадійна, ефективна та безпечна для роботи як зі структурованими, так і неструктурованими даними та з реалізацією стандартів SQL.

Має багатий функціонал, такий як оптимізація запитів за допомогою використання індексів, робота з JSON та XMl, робота з геоданими, повнотекстовий пошук та інше.

1.3. Мови програмування, бібліотеки, технології клієнтської частини

1.3.1. Vue.js

Vue.js 3 – JavaScript фреймворк з відкритим кодом для створення користувацьких інтерфейсів. Для роботи з даними він використовує реактивність та декларативний опис компонентів інтерфейсу.

Як і більшість інших сучасних JavaScript фреймворків, Vue.js для роботи з DOM-деревом сторінки використовує такий підхід, як Virtual DOM, зо дозволяє оптимізувати процес оновлення реального DOM-дерева веб-сторінки ігноруючи ті його частини, стан яких не змінився.

У версії Vue.js 3 весь внутрішній код було повністю переписано з використанням мови програмування TypeScript (див. 1.3.4), що надало розробникам усі переваги статичної перевірки типів.

Vue пропонує декілька варіантів використання фреймворку – від поступової інтеграції його у вже існуючий проект (додавання інтерактивності на статичний вебсайт, або ж використання його для створення окремих елементів інтерфейсу з усіма перевагами реактивних змінних), або повне створення веб-застосунку з використанням різноманітних елементів екосистеми Vue, таких як глобальне сховище, серверна генерація, чи навіть створення PWA (Progressive Web App).

Ключовою особливістю Vue є спосіб організації коду застосунку у вигляді однофайлових компонентів у файлах із розширенням `.vue`. Вони дозволяють зберігати у собі одночасно шаблон, стилі та логіку компонента (також є можливість створення власних секцій які потім компілюються на етапі збірки проекту).

1.3.2. Nuxt 3

Nuxt 3 - це метафреймворк на основі Vue 3, створений з фокусом на використання його як інструменту для розроблення додатків з генерацією сторінок на сервері (серверний рендер). Версія Nuxt 3 повністю створена з використанням TypeScript, та підтримує лише версію Vue. Цей перехід дозволив суттєво покращити безпеку, додатків та спростив сам процес розроблення.

Завдяки використання codegen підходу (це підхід, при якому на основі створених файлів автоматично генеруються відповідні файли з описами типів необхідних для роботи з цими файлами) відпала необхідність проводити відслідковування змін у назвах шаблонів, middleware, маршрутів, конфігурацій встановлених модулів та іншого. Після запуску серверу розробки Nuxt автоматично знаходить усі необхідні файли (Рис. 1.2) та генерує описи типів для них (Лістинг 1.1).

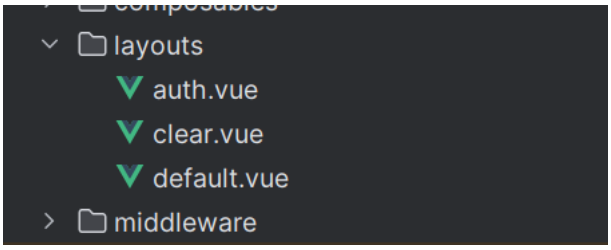

Рис. 1.2. Файли шаблонів у проекті у папці `layouts`
Джерело: фото структури проекту у середовищі розробки

На основі існуючих файлів Nuxt генерує типи для використання строгої типізації навіть для таких процесів як вибір шаблона сторінки (Лістинг 1.1).

Лістинг 1.1. Згенерований файл з типами з переліком наявних шаблонів

```
import { ComputedRef, Ref } from 'vue'
export type LayoutKey = "auth" | "clear" | "default"
declare module
"D:/Dev/versite/versite.frontend/node_modules/nuxt/dist/pages/runtime/com
posables" {
  interface PageMeta {
    layout?: false | LayoutKey | Ref<LayoutKey> | ComputedRef<LayoutKey>
  }
}
```

Надалі ці типи використовуються для підказок у середовищах розробки.

Рис. 1.3. Підказки у інтегрованому середовищі розробки з переліком доступних шаблонів сторінки
Джерело: Фото з інтегрованого середовища розробки

Окрім цього, Nuxt 3 конфігурує пакувальник Vite (див. 0 нижче), додає поліфіли (polyfills) для роботи в браузерах без підтримки нового синтаксису JS, має вбудований сервер для обробки запитів та інше.

Як було описано вище, метафреймворк розроблявся з фокусом на розроблення додатків з серверним рендером сторінок, але він також підтримує і інші режими роботи:

1. Універсальний режим, під час якого серверний рендер відбувається для першого запиту під час сесії, а наступні оновлення здійснюються на клієнті (Universal rendering). Це дозволяє проводити індексацію у пошукових сервісах та покращити час першого завантаження додатку.
2. Рендер повністю на стороні клієнта (Client-side only rendering). Використовується для додатків які не потребують індексації у пошукових системах або тих, для яких не є критичною швидкість завантаження першої сторінки.
3. Генерація повністю статичних сайтів (Full static site generation). Використовується для генерації вебсайтів, на яких рідко змінюється контент, та де він не залежить від стану іншого джерела інформації (наприклад бази даних). Наприклад для створення документації з певними інтерактивними елементами на сторінках.
4. Гібридний режим, поведінка може налаштовуватися окремо для маршрутів або груп маршрутів. Дозволяє налаштовувати поведінку окремих маршрутів або груп маршрутів (Лістинг 1.2). Доступні методи налаштування
    1) redirect – Визначає переадресацію на стороні сервера
    2) `ssr` – Вимикає генерацію сторінки на стороні сервера (серверний рендер) для окремих розділів сторінок, тобто, перемикає режим їх роботи на режим односторінкового додатку (Single Page Application, SPA).
    3) `cors` - Автоматично додає CORS заголовки.
    4) `headers` – Додавання заголовки на окремі розділи.
    5) `swr` – Додає заголовки для налаштування кешування та TTL на сервері або reverse proxy. Варто зазначити, що на таких сервісах як Netlify або Vercel відповіді з цих сторінок зберігаються у шарі CDN.

6) `static` – Має схожу поведінку як і `swr`, але не встановлює TTL, завдяки чому відповідь повністю кешується та не оновлюється до наступної публікації нової версії додатку.
7) `prerender` – Здійснює попередній рендер сторінок на етапі збірки проекту та включає їх як статичні ресурси.
8) `experimentalNoScripts` – Вимикає виконання скриптів для окремої секції.

Лістинг 1.2. Приклад налаштування маршрутів для додатка у гібридному режимі

```
export default defineNuxtConfig({
  routeRules: {
    '/blog/**': { swr: true },
    '/articles/**': { static: true },
    '/_nuxt/**': { headers: { 'cache-control': 's-maxage=0' } },
    '/admin/**': { ssr: false },
    '/api/v1/**': { cors: true },
    '/old-page': { redirect: '/new-page' },
    '/old-page2': { redirect: { to: '/new-page', statusCode: 302 } }
  }
});
```

1.3.3. Vite

Vite - це універсальний та модульний пакувальник для JavaScript, орієнтований на роботу з т.з. «Гарячим перезавантаженням модулів» (Hot Module Replacement, HMR). Vite є універсальним пакувальником, тому може використовуватися не лише з Vue, а й з іншими фреймворками, такими як React, Svelte, Solid, і не тільки. Завдяки модульності дозволяє легко та швидко створювати конфігурацію для будь якого проекту інкрементально інтегруючи різноманітні інструменти, наприклад компілятор для Vue, TypeScript та інші.

1.3.4. Typescript

Typescript – це відкрита мова програмування яка розширює можливості JavaScript шляхом додавання строгої типізації на етапі написання коду та компіляції. Але оскільки

TypeScript є лише розширенням до JavaScript, то на етапі збірки проектів, компілятор перетворює код написаний на TypeScript на JavaScript.

Статична типізація дозволяє визначати типи даних та функції на етапі написання коду, що зменшує кількість помилок пов’язаних з несумісністю типів, що полегшує розробку як на невеликих проектах, так і на великих проектах в особливості.

Переважна більшість сучасних бібліотек та фреймворків написані з використанням TypeScript, ба більше, більшість новітнього функціоналу у таких фреймворках як Nuxt 3 чи подібних створений лише завдяки використанню TypeScript (наприклад, підказки для існуючих шаблонів сторінок, як було показано на Рис. 1.3 та у розділі 1.3.2 в цілому).

1.3.5. Yarn

Yarn - це пакетний менеджер для JavaScript розроблений компанією Facebook, у порівнянні з стандартним пакетним менеджером npm (Node Package Manager) має кращі показники швидкодії та надійності.

1.3.6. Pinia

Pinia - це повністю типізована бібліотека для створення глобального об’єкту стану (також відомого як сховища) для вебзастосунків на базі Vue. Бібліотека має багато спільного з архітектурою Flux, який передбачає створення централізованих сховищ які зберігають глобальний стан додатку, таке компоненти які його використовують. Pinia є офіційною бібліотекою для створення сховища у Vue 3, та була створена на заміну VueX.

1.3.7. VueUse

VueUse - це бібліотека типових допоміжних composable функцій для Vue.js. Має більш ніж 200 функцій різного призначення, наприклад для роботи з реактивністю (`useRefHistory`, `createInjectionState`), з елементами на сторінці (`useDraggable`, `useIntersectionObserver`), з функціями браузера (`usePerformanceObserver`, `useClipboard`), з сенсорами пристрою

(`useBattery`, `useGeolocation`) та навіть для роботи із зовнішніми сервісами, такими як Firebase (`useFirestore`, `useRTDB`).

1.3.8. Tailwind CSS

Tailwind CSS - це відкритий CSS фреймворк, що використовує парадигму т.з. “Utility classes”, де CSS класи описуються атомарні CSS властивості, тобто, не обмежують користувача певною структурою HTML-документа, а повністю незалежні. Це дає змогу швидко та ефективно створювати інтерфейс без необхідності написання власного CSS коду, що суттєво економить час та дозволяє зосередитись на інших процесах.

1.3.9. Popper.js

Popper.js – це JavaScript бібліотека що спрощує та робить вкрай гнучким процеси позиціонування елементів на сторінці відносно інших елементів. Бібліотека також дозволяє встановлювати додаткові зв’язки та залежності, наприклад, від розміру екрану, поведінку при прокручуванні сторінки.

Також важливо те, що бібліотека не накладає обмеження на використання певного фреймворку, тому може бути використана у будь якому проекті.

1.4. Висновки до розділу

У цьому розділі було розглянуто використаний стек технологій та описано переваги використання кожного з його складових. Описано як основні технології (наприклад Nuxt 3 як основний фреймворк для клієнтської частини), так і додаткові бібліотеки (наприклад popper.js, яка використовується для позиціонування та прив’язки елементів на сторінці).

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Розроблення модулів

У попередній курсовій роботі було описано початковий процес проектування бази даних. За час розробки структура БД була частково змінена задля реалізації функціоналу.

На етапі планування було обрано модульну серверну архітектуру Porto, а оскільки саме від серверної частини в більшій мірі залежать усі процеси роботи з даними, то і опис реалізованих модулів буде відбуватися відповідно до існуючих моделей.

2.1.1. Course

Описує навчальний курс. Може бути створений користувачем.

Поля:

- `id` – Унікальний ідентифікатор. Первинний ключ.
- `title` – Заголовок курсу.
- `description` – Короткий опис для студентів. Може бути пустим (null).
- `author_id` – id автора курсу. Зовнішній ключ.

Зв’язки з іншими моделями:

- Курс – Автор. Кожен курс повинен мати користувача-автора який створив цей курс. Користувач може бути автором багатьох курсів.
- Курс – Уроки. Урок належить одному курсу, курс може мати багато уроків. Створювати уроки може лише автор курсу.
- Курс – Студенти. Автор може реєструвати на своєму курсі студентів. Студент створюється для кожного курсу окремо.

2.1.2. User

Для користувачів які реєструються в системі створюються записи в цій таблиці.

Поля:

- `id` – Унікальний ідентифікатор. Первинний ключ.
- `name` – Реальне ім’я користувача.
- `nickname` – Унікальний нік в системі.
- `email` – Електронна пошта за якою відбувалася реєстрація.
- `avatar` – Посилання на зображення профілю.
- `password` – Захешований пароль.
- `email_verified_at` – Дата підтвердження електронної пошти.
- `remember_token` – Системний токен який використовується для авторизації.

Зв’язки з іншими таблицями:

- Користувач(Автор) – Авторські курси. Користувач може створювати курси на яких він автоматично встановлюється як автор та якими він може керувати.
- Користувач – Запрошення. Отримані запрошення на реєстрацію на курс в ролі студента.
- Користувач – Студенти. Студенти з різних курсів у списках яких знаходиться користувач.

2.1.3. Student

Задля можливості для авторів курсів зберігати інформацію про студентів до того як вони приймуть запрошення було створено модель Student. Автор може вести курс та ставити оцінки студентам навіть якщо реальний користувач ще не зареєструвався на це місце (не прийняв запрошення).

Поля:

- `id` – Унікальний ідентифікатор. Первинний ключ.

- `name` – Ім’я студента яке встановлює автор курсу для розпізнавання студентів до того, як реальний користувач зареєструється на це місце.
- `course_id` – id курсу учасником якого є цей студент. Зовнішній ключ.
- `user_id` – id користувача який зареєструвався на місце цього студента. Може бути пусте (`null`), якщо реєстрація не відбулась. Зовнішній ключ.

Зв’язки:

- Студент – Курс. Відображає курс на якому зареєстрований цей студент.
- Студент – Користувач. Користувач який зареєструвався на місце цього студента.
- Студент – Запрошення. Відправлене запрошення для реєстрація користувача на місце цього студента.
- Студента – Подання. Подання які створював студент. Оскільки студент належить лише одному курсу, то і завдання на які студент створює подання теж належать цьому курсу.

2.1.4. Invitation

Після створення курсу та додавання першого студента автор курсу має змогу відправити на електронну пошту запрошення для реєстрації на місце обраного студента.

Поля:

- `id` – Унікальний ідентифікатор. Первинний ключ.
- `student_id` – id студента на місце якого було відправлено запрошення. Зовнішній ключ.
- `receiver_id` – id користувача якому було відправлено запрошення. Встановлюється при створенні, якщо вказана електронна пошта належить існуючому користувачу, інакше пусте. Зовнішній ключ.
- `email` – Електронна пошта, встановлюється якщо користувача із вказаною адресою не знайдено, інакше пусте.

- `is_hidden` – Використовується для приховання небажаних запрошень зі списку сповіщень.

Зв’язки:

- Запрошення – Студент. Студент на місце якого відправлено запрошення.
- Запрошення – Користувач. Користувач якому відправлено запрошення (може бути відсутній).

2.1.5. Lesson

Для публікації навчальних матеріалів та завдань на курсі створюються уроки. Задля оптимізації виконання запитів до БД, дані пов’язані з уроком було розділено на дві моделі, урок та текстовий матеріал.

Поля:

- `id` – Унікальний ідентифікатор. Первинний ключ.
- `title` – Заголовок уроку.
- `date` – Дата проведення уроку.
- `course_id` – id курсу якому належить цей урок. Зовнішній ключ.

Зв’язки:

- Урок – Курс. Створення уроку відбувається на сторінці уроків курсу. Виконати цю дію може лише автор курсу.
- Урок – Матеріал. При створенні уроку автоматично створюється запис в базі з відповідним матеріалом до цього уроку. Вміст уроку який автор може створити у спеціальному текстовому редакторі на сторінці редагування уроку потрапляє саме у цю таблицю.
- Урок – Завдання. Автор може створити завдання для студентів які будуть прив’язані до певного уроку.

2.1.6. Lesson Material

Використовується для зберігання текстового матеріалу уроку. Має зв’язок із відповідним уроком.

Поля:

- `id` – Унікальний ідентифікатор. Первинний ключ.
- `content` – Текстовий матеріал уроку. Зберігається у форматі HTML.
- `lesson_id` – id уроку якому належить цей матеріал. Зовнішній ключ.

Зв’язки:

- Матеріал – Урок. Урок якому належить цей матеріал.

2.1.7. Assignment

Використовується для створення завдань для студентів. Подані роботи які створюють студенти зберігаються у окремій таблиці з посиланням на завдання на яке була подана та робота.

Поля:

- `id` – Унікальний ідентифікатор. Первинний ключ.
- `title` – Заголовок до завдання.
- `description` – Необов’язковий додатковий опис до завдання.
- `max_mark` – Максимальна оцінка за виконання цього завдання.
- `deadline` – Кінцева дата здачі завдання.
- `lesson_id` – id уроку до якого прив’язане це завдання. Зовнішній ключ.

Зв’язки:

- Завдання – Урок. Урок до якого прив’язане завдання.
- Завдання – Подання. Виконані завдання які подали студенти.

2.1.8. Assignment Submission

Описує подання виконаних завдань студентами.

Поля:

- `id` – Унікальний ідентифікатор. Первинний ключ.
- `assignment_id` – id завдання на яке було створено це подання. Зовнішній ключ.
- `student_id` – id студента який створив це подання. Зовнішній ключ .
- `content` – Текстовий матеріал подання у форматі HTML.
- `mark` – Оцінка яку встановлює автор курсу після перевірки подання.
- `is_checked` – Поле яке відображає стан перевірки подання. Після перевірки автором автоматично встановлюється значення true.

Зв’язки:

- Подання – Завдання. Завдання на яке було створене це подання.
- Подання – Студент. Студент який виконав роботу та створив це подання.

2.2. Висновки до розділу

У цьому розділі було описано розроблені модулі серверної частини та їхні складові, а саме:

- Поля моделей – характеристики сутностей, які описують їх, їх поточний стан, або ж використовуються для опису реляційних зв’язків з іншими сутностями бази даних (наприклад поля `author_id` в сутності Course, яка зв’язує кожен курс з користувачем який його створив)
- Зв’язки – пов’язані сутності та тип зв’язку

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Оновлення механізму взаємодії з API

Основним завданням у створенні будь-якої інформаційної системи, є робота з даними. На початку створення цього проекту, необхідно було прийняти декілька рішень, такі як де та в якому форматі будуть зберігатися дані, та як буде відбуватися взаємодія з ними. Було вирішено створити окремо серверну частину, та клієнтську частину, які будуть взаємодіяти між собою за допомогою REST API.

3.1.1. Механізми виконання запитів у Nuxt 3

Оскільки було вирішено використовувати Nuxt.js 3 для створення клієнтської частини, який має вбудовані базові механізми для роботи з таким варіантом API – composable функції `useFetch` (Лістинг 3.1) або `useAsyncFetch` (Лістинг 3.2).

Лістинг 3.1. Використання `useFetch` для виконання запиту.

```
<script setup lang="ts">
const { data: count } = await useFetch('/api/count');
</script>

<template>
  <div>
    Page visits: {{ count }}
  </div>
</template>
```

Nuxt.js 3 пропонує декілька варіантів виконання запитів, але варто розуміти, що `useFetch` викликає в собі той самий `useAsyncData`, але лише додає певний «синтаксичний цукор». Якщо порівняти Лістинг 3.1 та Лістинг 3.2, то помітимо, що `useFetch` приймає лише один параметр з адресою API на яку необхідно виконати запит, в той час як `useAsyncData` приймає першим параметром певне значення `'count'`. Це повинен бути унікальний ключ, який функція `useAsyncData`

використовує для кешування відповіді від сервера та дедуплікації запитів (механізм для запобігання повторного виконання ідентичних запитів).

Лістинг 3.2. Використання `useAsyncData` для виконання запиту.

```
<script setup lang="ts">
const { data: count } = await useAsyncData('count', () =>
$fetch('/api/count'));
</script>

<template>
  <div>Page visits: {{ count }}</div>
</template>
```

Оскільки Nuxt 3 створений з використанням TypeScript, то передбачені способи опису типів даних які повинен повернути сервер у випадку успішного виконання запиту. Для цього необхідно викликати ці функції з generic-параметром (Лістинг 3.3).

Лістинг 3.3. Сигнатура перевантаженої функції `useAsyncData`

```
import { NuxtApp } from 'nuxt/dist/app/nuxt';
import { AsyncData, AsyncDataOptions, KeysOf, PickFrom } from
'nuxt/dist/app/composables/asyncData';

export declare function useAsyncData<
  ResT,
  DataE = Error,
  DataT = ResT,
  PickKeys extends KeysOf<DataT> = KeysOf<DataT>
>(
  handler: (ctx?: NuxtApp) => Promise<ResT>,
  options?: AsyncDataOptions<ResT, DataT, PickKeys>
): AsyncData<
  PickFrom<DataT, PickKeys>,
  DataE | null
>;
export declare function useAsyncData<
  ResT,
  DataE = Error,
  DataT = ResT,
  PickKeys extends KeysOf<DataT> = KeysOf<DataT>
```

```
>(
  key: string,
  handler: (ctx?: NuxtApp) => Promise<ResT>,
  options?: AsyncDataOptions<ResT, DataT, PickKeys>
): AsyncData<
  PickFrom<DataT, PickKeys>,
  DataE | null
>;
```

Як можна помітити на попередньому лістингу, функція `useAsyncData` є перевантаженою, та передавання ключа є необов'язковим, але в деяких ситуаціях це може спричинити неочевидні проблеми (наприклад, усі запити окрім найпершого не виконуються, а функція повертає результат виконання першого запиту).

3.1.2. Недоліки попередньої реалізації механізму взаємодії з API

Документація до фреймворку Nuxt 3 надає приклад використання функцій `useFetch` та `useAsyncData` (Лістинг 3.1, Лістинг 3.2), але хоч такий підхід і є максимально спрощений, що непогано для прикладу в документації, але фреймворк не пропонує ніяких рішень для групування даних що описують окремі запити (адреса, метод, тип даних відповіді, тип даних запиту).

Для вирішення цієї проблемо було створено певні принципи роботи з API:

1. Компоненти не повинні зберігати у собі інформацію про адреси API, методи доступу до них, формати даних які вони вимагають та формати даних відповідей з сервера. Порушення цього принципу призводить до дублювання коду, невідповідності даних у різних частинах додатку, а отже – до збільшення часу що необхідний для розширення та подальшої підтримки проекту.
2. Задля покращення організації, код групується по модулях.
3. Типи даних, що описують формат даних які повертає сервер при успішному виконанні запиту зберігаються у папці `responses` кожного модуля.

4. Типи даних які описують формат даних які сервер вимагає для виконання запиту зберігаються у папці `requests` кожного модуля.
5. Для кожної адреси на сервері створюється окремий файл який експортує об'єкт типу `Endpoint` (Лістинг 3.4) який описує тип даних запиту, відповіді, зберігає тип запиту (GET, POST, тощо) та адресу. Ці об'єкти зберігаються у папці `endpoints` кожного модуля.
6. Для виконання запиту необхідно викликати метод `useApiFetch` або `useApiAsyncData` та передати два параметра:
   1) Об'єкт типу Endpoint
   2) Параметри запиту відповідного типу

Лістинг 3.4. Опис типу Endpoint

```
import { RouterMethod } from 'h3';

export interface Endpoint<ResT, ReqT> {
  method: Uppercase<RouterMethod>;
  url: string;
}
```

Загалом, виконання запиту виглядало наступний чином (Лістинг 3.5).

Лістинг 3.5. Приклад виконання запиту

```
import { passwordLoginEndpoint, PasswordLoginRequest } from '~/api';

const loginForm: PasswordLoginRequest = reactive({
  email: '',
  password: ''
});
const { data, error } = await useApiAsyncData(passwordLoginEndpoint,
loginForm);
```

Однак, такий варіант реалізації мав деякі суттєві недоліки:

1. Як можна помітити (Лістинг 3.5), виконання запиту вимагає написання чималої кількості коду (імпорт адреси, виклик методу, передача адреси як першого обов’язкового параметра).
2. Код стає важче читати, розуміння та навігація по коду вимагає додаткових зусиль.
3. Необхідно слідкувати за тим, щоб назви адрес усі були різні.
4. Задля запобігання конфліктів імен зі збереженням спрощеного імпорту, необхідно іменувати усі об’єкти адрес із суфіксом `*Endpoint`.
5. Відсутнє групування в автодоповненні імпортів за модулями – об’єкти адрес із одного модуля знаходять порізно у списку автодоповнення при імпорті.

Також суттєвим недоліком було те, що на момент створення проекту виявилось, що реалізований механізм не підтримував виконання запитів на адреси які не потребують додаткових даних для виконання (запити з пустим тілом).

Було вирішено оновити механізм з урахуванням усіх потреб.

3.1.3. Формування уявлення про бажаний вигляд виклику API

У першій версії реалізації механізму взаємодії з API запити групувались у репозиторії, але було прийняте рішення відмовитись від цього підходу задля можливості вибирати метод для виконання запиту у компоненті, а не централізовано у репозиторії.

Але це прийняття цього рішення стало причиною (іноді частковою) для усіх 5 перелічених раніше недоліків. Тому було вирішено повернутися до концепції групування об’єктів адрес у репозиторіях, але дозволити обирати метод для виконання запиту у місці виклику.

Було сформовано декілька варіантів бачення того, як повинні виглядати виклики для виконання запитів, але обрано було наступний (Лістинг 3.6).

Лістинг 3.6. Оновлений вигляд виконання запиту.

```
import { courses } from 'api/repositories';

const { data } = await courses.all.fetch();
```

Такий варіант взаємодії з API вирішує усі попередні недоліки:

1. Кількість необхідного коду для виклику API скорочується завдяки таким рішенням:
    1) Відпадає необхідність додавання суфікса `*Endpoint`, адже усі об’єкти тепер знаходяться виключно всередині репозиторіїв, тому при дотриманні простих домовленостей контекст повністю описує призначення полів репозиторіїв.
    2) Завдяки вилученню репозиторіїв зі глобального імпорту `'api'`, і реімпорт репозиторіїв у окремому файлі `api/repositories.ts` відпадає також потреба додавання надлишкових суфіксів і до об’єктів репозиторіїв.
    3) Виклик composable-функцій `useFetch` та `useAsyncData` відбувається через виклик методів об’єктів `Endpoint`, тому і скорочується назва (`useApiFetch` стало `fetch`, `useApiAsyncData` стало `asyncData`).
2. Завдяки групуванню по репозиторіях та скороченню назв читабельність коду радикально покращується.
3. Слідкувати потрібно тепер лише за тим, щоб не дублювалися назви репозиторіїв, але оскільки в більшості випадків на один модуль припадає один репозиторій, то і це не є проблемою.
4. Об’єкти Endpoint завдяки групуванню у репозиторії не лише позбулися суфікса `*Endpoint`, а і можуть мати коротші та виразніші назви.
5. Об’єкти Endpoint групуються у репозиторії, тому пошук необхідного метода розбитий на два рівні – перший це вибір репозиторія, а другий це вибір необхідного метода. Підказки інтегрованого середовища розробки значно спрощують це процес.

3.1.4. Реалізація нового механізму

Оскільки на `Endpoint` об'єктах має бути можливість викликати методи, для створення добре підходять класи. TypeScript підтримує створення generic-класів, тому також залишається типізація.

У попередній реалізації для отримання generic-типів з об'єкта як параметра composable-функції необхідно було створювати додаткові допоміжні типи (Лістинг 3.7).

Лістинг 3.7. Допоміжні типи для екстракції generic-типів із параметра функції

```
type EndpointResT<E extends Endpoint<any, any>> = E extends
Endpoint<infer ResT, any> ?
  ResT : never;
type EndpointReqT<E extends Endpoint<any, any>> = E extends Endpoint<any,
infer ReqT> ?
  ReqT : never;
```

Завдяки використанню класів відпадає необхідність використання цих допоміжних типів, адже виклик методів `useFetch` чи `useAsyncData` відбуватиметься всередині класу де є прямий доступ до цих generic-типів.

3.1.4.1. Використання нестатичних адрес

Ще однією з проблем попередньої реалізації була неможливість виклику тих адрес, у яких параметр передається як частина URL-адреси (наприклад, `/courses/27`). Тому до двох попередніх generic-параметрів `ResT` (тип даних відповіді з сервера) та `ReqT` (тип даних які сервер потребує для виконання запита) додається ще один `RouteParamsT` який описуватиме об'єкт із необхідними параметрами (Лістинг 3.8).

Лістинг 3.8. Створений клас `Endpoint` з трьома generic-типами

```
export class Endpoint<
  ResT,
  ReqT extends Record<string, any> | undefined = undefined,
  RouteParamsT = undefined
> { }
```

Обов'язковим типом є лише `ResT`, адже відповідь з сервера при успішному виконанні буде завжди, а тіло запиту (`ReqT`) може бути пусте (наприклад, запити на отримання усіх даних) як і URL параметри, які теж не завжди потрібні (запит на створення ресурсу, який зазвичай має адресу в форматі `/courses/`).

Для додавання можливості використання нестатичних URL-адрес тип поля `url` було змінено зі string на `((params: RouteParamsT) => string) | string`. Тобто, у випадку якщо адреса на яку повинен здійснюватися запит статична, то у це поле записується звичайний рядок. У випадку ж якщо адреса нестатична і складається з параметрів, то тоді необхідно передати анонімну функцію яка приймає об'єкт із generic-типом `RouteParamsT` і яка повертатиме `string` (рядок). Також, оскільки ці поля не повинні змінюватися після створення об'єкта і не повинні бути доступні для доступу ззовні, було встановлено модифікатори `private` та `readonly` (Лістинг 3.9).

Лістинг 3.9. Поля класу `Endpoint`.

```
private readonly method: Uppercase<RouterMethod>;
private readonly url: ((params: RouteParamsT) => string) | string;
```

Оскільки поля відтепер приватні та доступні лише для читання, то встановлювати їх початковий стан можливо лише з конструктора.

3.1.4.2. Створення конструктора класу

Для передавання параметрів в конструктор було обрано використовувати паттерн об'єкт параметрів. Для цього створюється один параметр, але який буде об'єктом, поля якого будуть реальними параметрами.

Оскільки для можливості використовувати нестатичні URL-адреси полю `url` встановлено union-тип `((params: RouteParamsT) => string) | string`, то потрібно щоб тип параметра конструктора був відповідним. Але особливість в тому, що те, якого типу має бути цей параметр залежить від переданого generic-типу `RouteParamsT`. Якщо generic-тип було передано, то адреса нестатична і повинна

генеруватися в залежності від переданих параметрів під час виклику. Інакше, якщо generic-тип не було передано, то його тип буде мати стандартне значення `undefined`. В такому випадку коректним типом параметра буде `string` – статичний рядок який не залежить від переданих параметрів.

Для цього було використано механізм який має назву умовні типи. Він дозволяє створювати типи які залежать від певної умови. Умовні типи визначаються з використанням ключового слова `extends`. Тому тип параметра url має наступний вигляд (Лістинг 3.10).

Лістинг 3.10. Тип параметра `url` конструктора класу `Endpoint`.

```
RouteParamsT extends undefined
  ? string
  : (params: RouteParamsT) => string
```

Тут відбувається перевірка чи є generic-тип `RouteParamsT` підтипом до типу `undefined`, якщо так, то generic-тип має стандартне значення, отже `url` є статичним і його тип повинен бути `string` (Рис. 3.1).

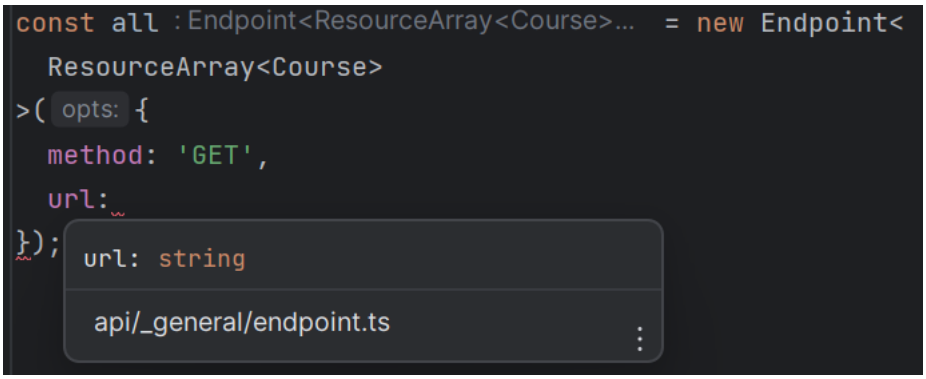

Рис. 3.1. Підказки типів без переданого generic-типу `RouteParamsT`
Джерело: Фото з інтегрованого середовища розробки

Інакше, якщо `RouteParamsT` не є підтипом для `undefined`, то це має бути `(params: RouteParamsT) => string` (Рис. 3.2).

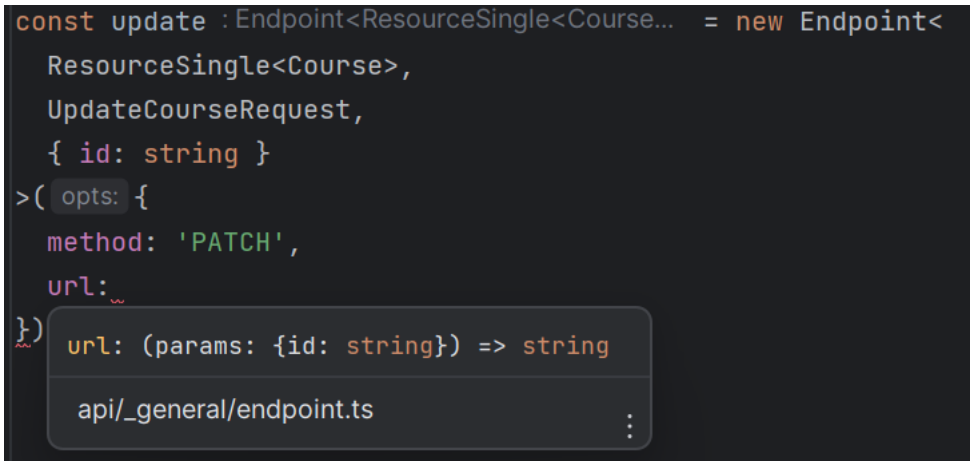

Рис. 3.2. Підказки типів з переданим generic-типом `RouteParamsT`
Джерело: Фото з інтегрованого середовища розробки

В результаті клас з конструктором має наступний вигляд ().

Лістинг 3.11. Вигляд класу `Endpoint` з полями та конструктором

```
export class Endpoint<
  ResT,
  ReqT extends Record<string, any> | undefined = undefined,
  RouteParamsT = undefined
> {
  private readonly method: Uppercase<RouterMethod>;
  private readonly url: ((params: RouteParamsT) => string) | string;

  constructor(opts: {
    method: Uppercase<RouterMethod>;
    url: RouteParamsT extends undefined
      ? string
      : (params: RouteParamsT) => string;
  }) {
    this.method = opts.method;
    this.url = opts.url;
  }
}
```

3.1.4.3. Методи для виконання запиту

Для виконання запиту на адресу API описану в об’єкті було додано два метода, `fetch` та `asyncData` (назви обрані відповідно до того, які composable-функції вони

використовують для виконання запиту). Оскільки відрізняються вони лише методом який використовуються для виконання запиту, розглянемо лише одну з них – `asyncData` (Лістинг 3.12).

Лістинг 3.12. Публічний метод `asyncData` із класу Endpoint.

```
public asyncData(
  ...[options]: ConditionallyRequiredOptions<
  AsyncDataOptions<ResT, ReqT, RouteParamsT>
>
) {
  const url =
    this.url instanceof Function
      ? this.url(options!.routeParams as RouteParamsT)
      : this.url;

  const key = `${this.method}-${url}`;

  return useApiAsyncData<ResT>(
    url,
    {
      method: this.method,

      ...options
    },
    key
  );
}
```

Окремої уваги заслуговує незвичний формат запису параметрів з певними нестандартними типами. Детальний розбір описано у розділі

Оскільки URL-адреса може бути як статичним рядком, так і методом який генерує адресу залежно від переданих параметрів, перед викликом необхідно перевірити чи поле `url` є функцією, і якщо так, то передати у неї необхідні параметри та отримати в результаті її виконання рядок з адресою.

Як вже було зазначено у розділі 3.1.1, при використанні composable-функції `useAsyncData` для правильного виконання запиту необхідно передавати унікальний ключ для різних запитів. Для спрощення процесу розробки, ключ генерується автоматично із методу запиту та адреси.

В кінці виконання методу виконання запиту передається у composable-функцію `useApiAsyncData` яка є обгорткою для стандартної composable-функції `useAsyncData` але встановлює глобальні параметри такі як токен авторизації, базову адресу API та деякі необхідні заголовки. Результат її виконання повертається користувачу.

3.1.4.4. Умовно обов’язкові параметри для методів

В процесі реалізації методів для виконання запитів було знайдено суттєвий недолік – якщо створити метод який приймає один обов’язковий аргумент з об’єктом параметрів, то для виконання деяких запитів цей об’єкт може бути пустим, що не є зручно. Спершу для вирішення цього було змінено аргумент на необов’язковий, для цього у TypeScript необхідно після назви параметра функції поставити `?` (Лістинг 3.13).

Лістинг 3.13. Метод з використанням необов’язкового параметра

```
public asyncData(options?: AsyncDataOptions<ResT, ReqT, RouteParamsT>) {
    // ...
}
```

При такому використанні змінюється тип параметра – він перетворюється на union-тип який складається із встановленого типу параметра та `undefined`, у цьому випадку тип параметра змінився на `AsyncDataOptions<ResT, ReqT, RouteParamsT> | undefined`. Але, недоліком цього рішення є те, що якщо не передати параметр у виклик тих запитів де є обов’язкові параметри TypeScript не покаже помилки.

Для вирішення цієї проблеми необхідно було створити умовно обов'язковий параметр, де умовою того чи буде цей параметр обов'язковим буде наявність у типі цього параметра обов'язкових полів.

Спершу необхідно було реалізувати умовно-обов'язкові поля у об'єкті параметрів. Для цього було створено тип `ConditionalProperty` (Лістинг 3.14).

Лістинг 3.14. Опис типу, що додає в об'єкт поля залежно від виконання певної умови

```
export type ConditionalProperty<Base, Key extends string, PropertyT> =
Base &
  (PropertyT extends undefined
    ? { [key in Key]?: never }
    : { [key in Key]: PropertyT });
```

Цей тип приймає 3 generic-типи:

1. `Base` – базовий тип, об'єкт у який необхідно умовно додати поле
2. `Key` – ключ об'єкта, назва поля яке необхідно додати при виконання певної умови
3. `PropertyT` – тип поля яке буде додане за вказаним ключем `Key`.

Умовою додавання поля із вказаним ключем `Key` та типом `PropertyT` є те, чи `PropertyT` не є підтипом до `undefined`. Інакше, це поле стає необов'язковим і йому встановлюється спеціальний тип `never`. У TypeScript цей тип використовуються для полів які не можуть мати значення, або для ситуацій які ніколи не можуть статися. З точки зору дискретної математики, цей тип можна описати як пусту множину, тобто множину яка не містить жодного елемента. Саме тому полям із цим типом не може бути встановлене жодне існуюче значення.

Завдяки використанню цього типу, якщо, наприклад, запит який має статичну адресу та не потребує параметрів для побудови URL-адреси отримає цей параметр, TypeScript сповістить про помилку несумісності типів, адже будь-яке значення не може бути встановлене.

Використовуючи цей тип до стандартних параметрів composable-функції `useAsyncData` додаються типізовані та умовно-обов'язкові поля `body` та `routeParams` (Лістинг 3.15).

Лістинг 3.15. Додавання в параметри типізованих полів `body` та `routeParams`.

```
type AsyncDataOptions<ResT, ReqT, RouteParamsT> = ConditionalProperty<
  ConditionalProperty<
    Omit<AsyncHttpOptions<ResT>, 'method' | 'body'>,
    'body',
    ReqT
  >,
  'routeParams',
  RouteParamsT
>;
```

Із стандартного типу `AsyncHttpOptions` окрім поля `body` викидається за допомогою вбудованого типу `Omit` також поле `method`, адже метод запиту встановлюється автоматично із поля `method` класу `Endpoint` (Лістинг 3.12).

Відтепер для досягнення бажаної поведінки залишається вирішити одну проблему – зробити параметр options обов'язковим лише тоді, коли у ньому є обов'язкові поля. Нажаль, в TypeScript немає вбудованого механізму який дозволив би це зробити, тому для цього було створено наступний тип (Лістинг 3.16).

Лістинг 3.16. Тип що дозволяє отримати умовно-обов'язковий параметр функції

```
type ConditionallyRequiredOptions<OptionsT extends object> =
  keyof RequiredFieldsOnly<OptionsT> extends never
    ? [OptionsT?]
    : [OptionsT];

export type RequiredFieldsOnly<T extends object> = {
  [K in keyof T as T[K] extends Required<T>[K] ? K : never]: T[K];
};
```

Тип `RequiredKeysOnly` повертає підтип від переданого generic-типу `T`, який включає в себе лише обов'язкові поля.

Оператор `keyof` дозволяє отримати union-тип, який складається з переліку ключів полів переданого об'єкта. У випадку використання цього оператора на об'єкті який не має полів, результатом його виконання буде порожня множина – тип `never`. Саме використання цієї поведінки лежить в основі роботи типу `ConditionallyRequiredOptions` (Лістинг 3.16). Для перевірки на наявність обов'язкових полів використовується перевірка чи список обов'язкових полів переданого об'єкті є підтипом до типу `never` (`keyof RequiredFieldsOnly<OptionsT> extends never`). Якщо умова виконується, то повертається кортеж з одним необов'язковим елементом з типом `OptionsT`, інакше, ідентичний кортеж, але з обов'язковим елементом.

Для перетворення цього типу у параметри метода створення масиву з один параметром та передавання їх як залишкових параметрів (Лістинг 3.12). Завдяки цьому запити в яких немає обов'язкових параметрів можуть бути викликані без передавання опцій (Рис. 3.3).

```
const { data } = await courses.all.asyncData();
```

Рис. 3.3. Виклик методу `all` який не має обов'язкових параметрів без помилок.
Джерело: Фото інтегрованого середовища розробки

В той же час, якщо виконувати запити які потребують додаткові параметри для виконання та не передати ці параметри, то буде отримано помилку (Рис. 3.4).

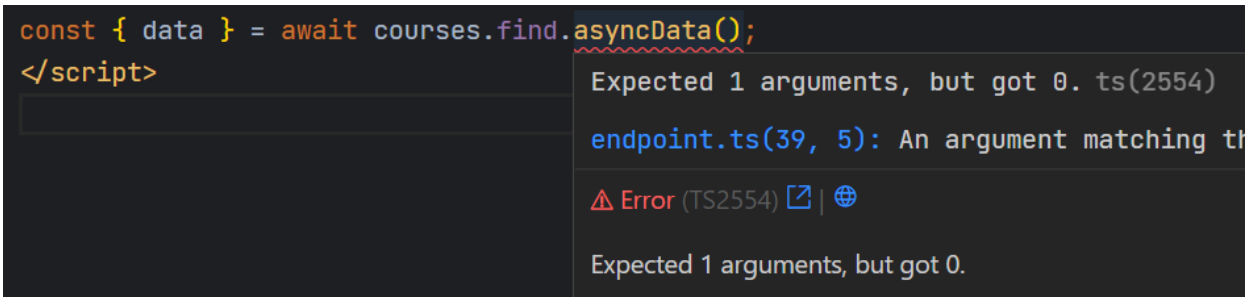

Рис. 3.4. На методі який має обов'язкові параметри необхідно передавати об'єкт з опціями.

Джерело: Фото інтегрованого середовища розробки

Якщо ж передати об’єкт з опціями, але не встановити усі обов’язкові, то буде отримано звичну помилку про відсутність обов’язкового поля (Рис. 3.5).

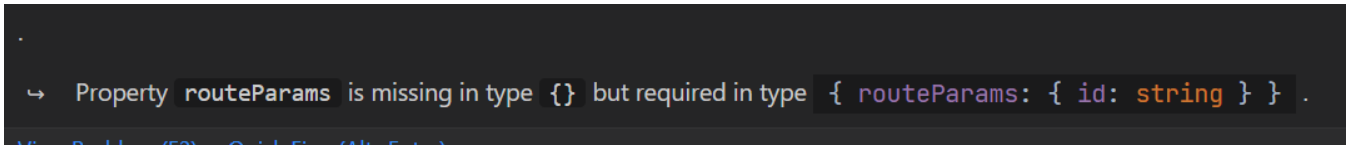

Рис. 3.5. Помилка про відсутність обов’язкового поля
Джерело: Фото інтегрованого середовища розробки

3.2. Недоліки обраної архітектури та стеку технологій

3.2.1. Причини заміни архітектури

На початку планування розробки серверної частини було прийнято рішення використати замість стандартної архітектури Laravel архітектуру Porto. Цей архітектурний шаблон пропонує також його готову реалізація на Laravel яка має назву Apiato яку розробив автор Porto. Причини цього рішення були наступні:

- Стандартна архітектура Laravel пропонує зберігати весь код що відповідає за обробку певного запиту у методах відповідного контролера. При такому підході у класів контролерів з’являється багато відповідальностей (формування запитів до БД, фільтрація, сортування, пагінація, відправка подій, сповіщень, формування та форматування відповіді, тощо).
- Групування відбувалося за типом сутностей, а не за бізнес-логікою. При подальшому розвитку проекту це погіршує орієнтацію по ньому (наприклад, велика кількість моделей які не пов’язані між собою, але усі знаходяться у одній папці).
- Apiato одразу після створення нового проекту реалізовує велику частину функціоналу, та проводить більшість типових налаштувань проекту, що інакше доводиться виконувати вручну, наприклад:

- Налаштування системи авторизації з використанням пакету Laravel Passport.
- Додає автогенератор документації для API.
- Налаштування Rate Limit для обмеження кількості запитів до API
- Додає готову реалізацію параметрів запиту для фільтрації, сортування, повнотекстового пошуку та додавання у відповідь зв’язків моделей.

3.2.2. Знайдені проблеми під час розробки

Очікувалося, що використання Apiato дозволить виправити проблеми використання Laravel, але, під час розробки було помічено деякі проблеми які по-різному вплинули на досвід розробки.

3.2.2.1. Кількість додаткових файлів

Задля розділення коду Porto вводить нові сутності для зберігання коду, що відповідає за безпосередню роботу з даними та виконання бізнес-логіки, такі я Action (Дія), Sub-Action (Піддія) так Task (Завдання). Але в більшості випадків, таке розділення є надлишковим та збільшує кількість файлів.

Особливо гостро постає проблема передавання даних між цими сутностями. Оскільки PHP це мова із динамічною типізацією і Laravel більшість часу свого існування розроблявся без строгої типізації, то занадто часто виникають проблеми із несумісністю типів. В купі з модифікованим обробником помилок від Apiato, пошук причини та виправлення таких проблем вимагає надмірних витрат часу.

3.2.2.2. Пріоритет обробки маршрутів

На відміну від, наприклад, маршрутизатора для Vue, Laravel не має підтримки обробки пріоритету маршрутів, а натомість вибір маршруту під час обробки запиту відбувається за принципом «перший відповідний». Наприклад, якщо буде описано два маршрути `/courses/{id}` та `/courses/my`, то запит на адресу `/courses/my` буде оброблений першим правилом (через те, що воно описане першим). В стандартному

підході Laravel пропонує змінити порядок оголошення маршрутів у файлі. Але Apiato використовує маршрутизацію, в якій кожен маршрут описаний в окремих файлах, через що порядок їх реєстрації залежить від назви файлу. Така поведінка призводить до складних в пошуку та вирішенні проблем (для коректної обробки необхідно прописувати додаткові правила за допомогою регулярних виразів).

3.2.2.3. Виконання запитів до БД

Такий функціонал як параметри запиту для фільтрації, сортування, додавання пов'язаних моделей створено з використанням пакету `l5-repository`. Хоч цей пакет і реалізовує чималу кількість додаткового функціоналу, але попри своє давнє існування та популярність має документацію вкрай низької якості, повну відсутність підказок в IDE та надзвичайно погіршує прості процеси.

3.2.2.4. Затримки у виправленні помилок

Наразі проект Apiato підтримує лише його автор, і ця підтримка є доволі неактивною. Тому використання на проектах із запланованим довгим терміном підтримки не рекомендується.

3.2.2.5. Підказки типів та полів

Для вирішення проблем з підказками типів та полів при використанні Laravel, було створено деякі допоміжні інструменти:

- пакети які аналізують міграцій бази даних та генерують додаткові файли з прописаними PHPDoc (наприклад `laravel-ide-helper`)
- плагіни для інтегрованих середовищ розробки (`Laravel Idea`)

При розробці проекту із використанням Laravel без Apiato, пакетів та плагінів достатньо для часткового вирішення проблем із підказками у інтегрованому середовищі розробки. Apiato використовує певні пакети для зміни стандартної поведінки певних

сутностей та має модифіковану файлову структуру файлів проекту, через що проблеми з підказками типів та полів з’являються знову.

3.2.3. Існуючі аналоги та підходи

Незважаючи на переваги які надає використання Apiato, недоліки настільки суттєві, що його використання здебільшого ускладнило розробку. На відміну від Nuxt 3 та TypeScript, використання яких очевидно покращило процес розробки. Тому було вирішено провести дослідження існуючих аналогів з урахуванням отриманого досвіду.

Все більш популярним стає використання TypeScript для розробки також і серверної частини проекту. Node.js дає змогу запускати JavaScript як на сервері, так і в консолі, тому і для JavaScript були створені різноманітні бібліотеки для раніше типових для серверів завдань (наприклад, робота з базою даних, робота з файлами, тощо).

Для створення сервера найбільш популярними рішеннями є Express.js та Nest.js.

- Express.js – це бібліотека для легкого створення веб-серверів на Node.js. Серед його основних функцій є маршрутизація, обробка запитів та відповідей, обробка помилок, робота з middleware, тощо. Головною перевагою використання є можливість максимально швидко створити та запустити сервер з пустого проекту.
- Nest.js – це повноцінний фреймворк орієнтований на розробку масштабованих та підтримуваних проектів. Він повністю базується на TypeScript та пропонує об’єктно-орієнтований до розробки серверних додатків. Має вбудовану підтримку багатьох функцій, наприклад WebSocket та GraphQL. Має вбудовані механізми для розробки веб-застосунків з використанням модульної архітектури та мікросервісів.

Іншим варіантом є використання бібліотеки h3, яка є вбудованим сервером у Nuxt 3. Він дозволяє створювати сторінки для обробки запитів на внутрішнє API. В такому випадку усі серверні файли є недоступними для клієнта, а відповідають лише за обробку запитів на сервері. Такий підхід, при якому код для серверної частини та

клієнтської зберігається у одному репозиторії називається «monorepo» (mono repository). Він дає змогу при розробці full-stack застосунків використовувати єдині описи типів як на сервері, так і на клієнті, що суттєво спрощує процес розроблення, та дозоляє уникнути помилок пов'язаних із невідповідністю типів.

Для цього непоганим рішенням є використовувати бібліотеку tRPC. tRPC – це бібліотека яка забезпечує зручний та ефективний спосіб взаємодії між сервером та клієнтом за допомогою RPC (Remote Procedure Call). Вона надає простий та повністю типізований API, дозволяє використовувати різні транспортні протоколи (HTTP, WebSocket), використовує покращені методи для серіалізації та десеріалізації.

Перевагою tRPC є також використання бібліотеки Zod. Це бібліотека для валідації даних, яка завдяки використанню TypeScript може повертати інтерфейси для даних на основі створеної схеми. Це дозволяє описувати схему валідації даних (Лістинг 3.17) використовуючи лише бібліотеку Zod, та автоматично отримувати строго типізовані сигнатури для методів виклику API.

Лістинг 3.17. Приклад створення процедури з використанням tRPC та Zod

```
import { object, string } from 'zod'
import { publicProcedure, router } from '../trpc'

export const appRouter = router({
  example: publicProcedure.input(
    object({
      text: string().nullish()
    })
  ).query(({ input }) => {
    return {
      message: `Message: ${input?.text}`,
      time: new Date()
    }
  })
})
```

Виклик цієї процедури на стороні клієнта виглядає наступний чином (Лістинг 3.18).

Лістинг 3.18. Виклик процедури на стороні клієнта

```
<script setup lang="ts">
const { $client } = useNuxtApp()

const { data } = await $client.example.useQuery({
  text: 'Text from client'
})
</script>

<template>
  <div>
    tRPC Data: "{{ data?.message }}" send at "{{ data?.time }}".
  </div>
</template>
```

3.2.4. Рішення для Nuxt 3

Для Nuxt 3 розробляється набір розробки (development kit) під назвою Sidebase. Він створений для швидкого старту розробки та конфігурації проектів з використанням tRPC та Prisma (ORM для TypeScript). В набір також входять також власні бібліотеки:

- `nuxt-auth` – це бібліотека для додавання авторизації та реєстрації як за допомогою електронної пошти та паролю, так і великого переліку зовнішніх провайдерів, таких як Google чи GitHub
- `nuxt-session` – це бібліотека для налаштування зберігання даних користувача між різними запитами та додавання спеціальних middleware для роботи зі станом користувача

3.3. Висновки до розділу

У цьому розділі було розглянуто процес оновлення механізму взаємодії з API. Процес створення цього механізму було описано у попередній роботі, але через певні недосконалості потребував змін. Окремої уваги було надано складнощам з якими довелось зустрітися в процесі реалізації та причинам які призвели до обраних рішень. В

результаті було створено зручний у використанні завдяки повній статичній типізації з використанням TypeScript.

Також було розглянуто недоліки обраної архітектури серверної частини (наприклад, проблеми з коректним визначенням пріоритетних маршрутів чи суттєве погіршення процесу виконання нестандартних запитів до БД). В корені проблемою є відсутність строгої типізації в PHP, тому було описано можливі аналоги що позбавлені критичних недоліків обраного серверного стеку технологій.

ВИСНОВКИ

Під час роботи над цим кваліфікаційним проектом було проведене дослідження та вивчення нових технологій спочатку теоретично, а потім і з практичним використанням набутих знань. Після роботи можна зробити наступні висновки:

1. Строга типізація з використанням TypeScript надає безперечні переваги, такі як статична перевірка коду, підсвітка можливих типів та значень полів, змінних, тощо. Окремо варто виділити додаткові можливість які надає використання TypeScript, наприклад ті, що описані в розділі 3.1 (умовні типи).
2. Використання PHP для створення проектів вважаю невдалим вибором, через відсутність строгої типізації, що призводить до великої кількості помилок, які можливо було уникнути за наявності статичної перевірки типів ще до запуску коду (підсвітка помилок у середовищі розробки).
3. Використання бібліотеки Apiato для створення серверної частини було також невдалим вибором через те, що нівелювало певні ключові переваги використання Laravel.
4. На початку розробки проекту Nuxt 3 був у нестабільній версії, але з часом були реалізовані усі обіцяні функції. Але, варто зазначити, що у випадку якщо швидкість розробки критична (наприклад при розробці стартапу), то використання нестабільних бібліотек та фреймворків з великою вірогідністю призведе до втрат часу та ситуацій, в яких доведеться вносити масові правки через зміни в критичних елементах.
5. Непоганою альтернативою для використання замість Laravel та Apiato для розробки серверної частини може бути вбудований сервер Nuxt, tRPC та Prisma (для цього стеку також існує проект Sidebase який створює та підтримує додаткові пакети)

Отже, результатом виконання цього кваліфікаційного проекту є створені та налаштовані проекти клієнтської та серверної сторони з реалізованим функціоналом який дозволяє повноцінно використовувати платформу, та отримані знання з використання сучасних технологій розробки програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Документація для фреймворку Vue.js. URL: https://vuejs.org. (Дата звернення: 02.05.2022 р.).
2. Документація для мови програмування php. URL: https://www.php.net.
3. Документація для фреймворку Laravel версії 9. URL: https://laravel.com/docs/9.x. (Дата звернення: 02.05.2022 р.).
4. Специфікація архітектурного шаблону Porto. URL: https://github.com/Mahmoudz/Porto. (Дата звернення: 02.05.2022 р.).
5. Документація для бібліотеки Apiato. URL: https://apiato.io. (Дата звернення: 02.05.2022 р.).
6. Чистий код: створення і рефакторинг за допомогою Agile / пер. з англ. І. Бондар-Терещенко. – Харків, 2019. 448 с.
7. Чиста архітектура: Мистецтво розроблення програмного забезпечення / пер. з англ. І. Бондар-Терещенко. – Харків, 2019. 368 с.
8. Документація для технології Docker. URL: https://docs.docker.com. (Дата звернення: 02.05.2022 р.).
9. Документація для інструменту ESLint. URL: https://eslint.org/docs. (Дата звернення: 02.05.2022 р.).
10. Документація для інструменту Prettier. URL: https://prettier.io/docs/en/index.html. (Дата звернення: 02.05.2022 р.).
11. Документація для фреймворку Nuxt. URL: https://nuxt.com. (Дата звернення: 02.05.2022 р.).
12. Crazy, Powerful TypeScript Tuple Types. URL: https://levelup.gitconnected.com/crazy-powerful-typescript-tuple-types-9b121e0a690c. (Дата звернення: 02.05.2022 р.).
13. Документація для мови програмування TypeScript. URL: https://www.typescriptlang.org. (Дата звернення: 02.05.2022 р.).

14. Документація по використанню пакетного менеджера yarn. URL: https://yarnpkg.com. (Дата звернення: 02.05.2022 р.).
15. Документація по використанню бібліотеки Pinia. URL: https://pinia.vuejs.org. (Дата звернення: 02.05.2022 р.).
16. Using TypeScript Mapped Types Like a Pro. URL: https://javascript.plainenglish.io/using-typescript-mapped-types-like-a-pro-be10aef5511a. (Дата звернення: 02.05.2022 р.).
17. A Complete Guide To TypeScript's Never Type. URL: https://www.zhenghao.io/posts/ts-never. (Дата звернення: 02.05.2022 р.).
18. Understanding infer in TypeScript. URL: https://blog.logrocket.com/understanding-infer-typescript/. (Дата звернення: 02.05.2022 р.).
19. Understanding Conditional Types in TypeScript. URL: https://www.syncfusion.com/blogs/post/understanding-conditional-types-in-typescript.aspx. (Дата звернення: 02.05.2022 р.).
20. Документація для бібліотеки VueUse. URL: https://vueuse.org. (Дата звернення: 02.05.2022 р.).
21. GitHub репозиторій JavaScript бібліотеки ofetch. URL: https://github.com/unjs/ofetch. (Дата звернення: 02.05.2022 р.).h
22. Документація для бібліотеки TipTap. URL: https://tiptap.dev. (Дата звернення: 02.05.2022 р.).
23. Документація для фреймворку TailwindCSS. URL: https://tailwindcss.com. (Дата звернення: 02.05.2022 р.).
24. Документація для бібліотеки Iconify. URL: https://iconify.design. (Дата звернення: 02.05.2022 р.).
25. Документація для технології SASS. URL: https://sass-lang.com. (Дата звернення: 02.05.2022 р.).

26. Adding ESLint and Prettier to Nuxt 3. URL: https://dev.to/tao/adding-eslint-and-prettier-to-nuxt-3-2023-5bg. (Дата звернення: 02.05.2022 р.).

ДОДАТКИ

Додаток А. Повний лістинг класу Endpoint

```
import { RouterMethod } from 'h3';
import { AsyncHttpOptions } from '~/types/fetch/fetch';
import { ConditionalProperty, RequiredFieldsOnly } from '~/types/object';

type AsyncDataOptions<ResT, ReqT, RouteParamsT> = ConditionalProperty<
  ConditionalProperty<
    Omit<AsyncHttpOptions<ResT>, 'method' | 'body'>,
    'body',
    ReqT
  >,
  'routeParams',
  RouteParamsT
>;

type ConditionallyRequiredOptions<OptionsT extends object> =
  keyof RequiredFieldsOnly<OptionsT> extends never
    ? [OptionsT?] // They can be optional
    : [OptionsT]; // Otherwise, required

export class Endpoint<
  ResT,
  ReqT extends Record<string, any> | undefined = undefined,
  RouteParamsT = undefined
> {
  private readonly method;
  private readonly url: ((params: RouteParamsT) => string) | string;

  constructor(opts: {
    method: Uppercase<RouterMethod>;
    url: RouteParamsT extends undefined
      ? string
      : (params: RouteParamsT) => string;
  }) {
    this.method = opts.method;
    this.url = opts.url;
  }
```

```
  public asyncData(
    ...[options]: ConditionallyRequiredOptions<AsyncDataOptions<ResT,
ReqT, RouteParamsT>>
  ) {
    const url =
      this.url instanceof Function
        ? this.url(options!.routeParams as RouteParamsT)
        : this.url;

    const key = `${this.method}-${url}`;

    return useApiAsyncData<ResT>(
      url,
      {
        method: this.method,

        ...options
      },
      key
    );
  }

  public fetch(
    ...[options]: ConditionallyRequiredOptions<AsyncDataOptions<ResT,
ReqT, RouteParamsT>>
  ) {
    const url =
      this.url instanceof Function
        ? this.url(options!.routeParams as RouteParamsT)
        : this.url;

    return useApiFetch<ResT>(url, {
      method: this.method,

      ...options
    });
  }
}
```

Додаток В. Оновлена схема бази даних

```
Project project_name {
  database_type: 'PostgreSQL'
}

Table courses {
  id int [pk, increment]
  title varchar
  description varchar [null]
  author_id int [ref: > users.id]
}
Table users {
  id int [pk, increment]
  name varchar
  nickname varchar
  email varchar
  avatar varchar
  password varchar
  email_verified_at datetime
  remember_token varchar
}
Table students {
  id int [pk, increment]
```

```
  name varchar [null, Note: 'Author-only student name']
  course_id int [ref: > courses.id]
  user_id int [null, ref: > users.id]
}
Table invitations {
  id int [pk, increment]
  student_id int [ref: - students.id]
  receiver_id int [null, ref: > users.id]
  email varchar [null]
  is_hidden boolean [default: false]

  Note: '''
    If on invitation create user with provided email exists,
    set email to null, receiver_id to this user id.
  '''
}
Table lessons {
  id int [pk, increment]
  title varchar
  date datetime
  course_id int [ref: > courses.id]
}
Table lesson_materials {
```

```
  id int [pk, increment]
  content text
  lesson_id int [ref: - lessons.id]
}
Table assignments {
  id int [pk, increment]
  title varchar
  description text [null]
  max_mark int [null]
  deadline datetime [null]
  lesson_id int [ref: > lessons.id]
}
Table assignment_submissions {
  id int [pk, increment]
  assignment_id int [ref: > assignments.id]
  student_id int [ref: > students.id]
  content text
  mark int [null]
  is_checked boolean [default: false]
}
```

Додаток С. Діаграма бази даних