ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
Сучасні каркаси web-додатків
Лісняк Андрій Олександрович

Лабораторна робота 7. Створення приватної сторінки редагування товарів із Pinia та Vue Router

Мета

Навчитися розподіляти локальний і спільний стан, організовувати вхід та відновлення сеансу через Pinia, обмежувати приватні переходи й застосовувати підтверджені зміни товару після відповіді API.

Теоретичні відомості

Інтерфейс застосунку складається з компонентів, які показують дані та реагують на дії користувача. Коли кілька компонентів використовують одні й ті самі дані, зміна в одному місці має узгоджено відображатися в усіх інших. Для цього потрібно визначити, де зберігається спільний стан і як компоненти його отримують та змінюють.

1. Проблема узгодження стану між компонентами

Значення, потрібне лише одному компоненту, може зберігатися в ньому. Наприклад, стан розкриття підказки не потрібно передавати всьому застосунку. Складність виникає, коли від того самого значення залежать кілька незалежних частин інтерфейсу.

Відомості про вхід користувача можуть одночасно впливати на шапку, доступні сторінки й дозволені дії. Якщо кожна частина зберігає власну копію цих відомостей, після виходу одна вже може показувати стан гостя, а інша — продовжувати показувати приватний вміст. Синхронізація копій потребує додаткових повідомлень і створює ризик пропустити оновлення.

Спільне джерело даних усуває потребу узгоджувати копії. Проте самого доступу до значення недостатньо. Потрібно також визначити правила зміни, похідні значення та момент очищення стану. Спосіб організації залежить від того, які компоненти використовують дані та як вони пов’язані.

2. Способи організації спільного стану

Для близьких компонентів стан можна перенести до спільного батьківського компонента. Нащадки отримують значення через props, а про потрібні зміни повідомляють подіями. Батьківський компонент змінює стан, після чого нові значення надходять нащадкам. За великої глибини дерева дані та події доводиться передавати через проміжні компоненти, яким вони не потрібні.

Механізм provide/inject дозволяє предку надати стан і функції його зміни віддаленим нащадкам без передавання через кожен проміжний компонент. Такий підхід придатний для пов’язаної групи компонентів. Область доступу

визначається місцем провайдера; провайдер рівня застосунку може надати залежність усім його компонентам.

Composable відокремлює логіку роботи зі станом у функцію. Саме повторне використання функції ще не означає спільних даних. Якщо ref створюється всередині кожного виклику, виклики отримують окремі значення. Спільний реактивний об'єкт можна створити на рівні модуля та повертати його з composable. У невеликому клієнтському застосунку цього може бути достатньо; правила зміни й очищення такого стану визначає власний код.

Коли спільних даних та операцій стає більше, їх можна організувати в окремі сховища за відповідальністю. Компоненти звертаються до потрібного сховища, а пов'язана зі станом логіка зосереджується поруч із даними. Це зменшує залежність доступу до даних від розташування компонентів.

Pinia реалізує цей підхід для Vue. Вона надає іменовані сховища, обчислювані значення, дії та інтеграцію з Vue DevTools для спостереження за змінами. Окремі сховища можуть відповідати за сеанс користувача й дані предметної області. Локальні значення залишаються у компонентах.

3. Створення сховища Pinia та його state

Сховище стану (store) — реактивний об'єкт зі спільними даними та пов'язаною з ними логікою.

`createPinia()` створює екземпляр Pinia, який підключається до Vue через `app.use(pinia)` перед монтуванням. Якщо початкова навігація використовує store, Pinia підключається перед маршрутизатором.

`defineStore()` отримує унікальний ідентифікатор та опис сховища. Він повертає функцію доступу, яку за домовленістю називають `use...Store`. Перший виклик цієї функції створює store. Наступні виклики в межах тієї самої Pinia повертають той самий екземпляр. Знищення компонента саме по собі не скидає його стан.

Pinia підтримує два способи опису сховища.

Option Store — сховище, описане об'єктом із полями state, getters та actions.

Setup Store — сховище, описане функцією, яка створює та повертає реактивні значення, обчислення й дії.

State — реактивні значення, які зберігає сховище.

У Setup Store ref задає state. Усі властивості власного реактивного стану повертаються зі сховища, щоб Pinia могла обробляти їх у Vue DevTools, плагінах і серверному рендерингу (SSR). Приховування state або повернення його як readonly порушує цю взаємодію. Службовий AbortController не є реактивним state.

Pinia дозволяє прямий запис, наприклад `counter.count = 5`, і групування змін через `counter.$patch({ count: 10 })`. Функціональна форма `$patch(state => { state.count += 1 })` об'єднує зміни, зокрема операції з колекціями, в одну подію DevTools. Вибір actions для перевірки даних і збереження є правилом архітектури застосунку, а не забороною прямого запису.

4. Getters та actions

Getter — обчислюване значення, похідне від реактивних залежностей.

У Setup Store getter задається через computed. Результат кешується, доки залежності не зміняться. Наприклад, `doubleCount` залежить від `count`, тому нове значення count змінює подвоєний результат. Getter читає стан без HTTP-запитів та інших побічних ефектів.

Action — функція сховища, яка виконує операцію та може змінювати state.

Action приймає аргументи, повертає результат і може бути асинхронною. Отримання даних через API, перевірка відповіді та запис результату можуть бути частинами однієї action. Обробка помилки, заборона повторного запуску й скасування запиту реалізуються її кодом; Pinia не додає їх автоматично.

5. Використання store у компонентах

Мінімальний приклад двох компонентів використовує таке сховище.

```
import { computed, ref } from 'vue'
import { defineStore } from 'pinia'

export const useCounterStore = defineStore('counter', () => {
  const count = ref(0)
  const doubleCount = computed(() => count.value * 2)
  function increment() { count.value += 1 }
  function $reset() { count.value = 0 }
  return { count, doubleCount, increment, $reset }
})
```

Один компонент викликає increment, другий показує count і doubleCount. Обидва отримують store через `useCounterStore()`. Передавання props між цими компонентами не потрібне, оскільки вони використовують один екземпляр.

Store є реактивним об’єктом, тому його властивості читаються як `counter.count` і `counter.doubleCount` без `.value`. Усередині функції Setup Store локальний count залишається ref і потребує `count.value`.

Звичайна деструктуризація `const { count } = counter` копіює поточне числове значення та втрачає реактивний зв’язок. Для стану й getters використовується `storeToRefs(counter)`.

```
const counter = useCounterStore()
const { count, doubleCount } = storeToRefs(counter)
const { increment } = counter
```

Тепер count і doubleCount є refs. У JavaScript вони читаються через `.value`, а в шаблоні компонента розгортаються автоматично. Actions можна деструктурувати без storeToRefs, оскільки Pinia зв’язує їх зі сховищем. `storeToRefs()` пропускає функції та нереактивні властивості.

6. Скидання стану

В Option Store вбудована дія `$reset()` відновлює початкові значення через повторний виклик функції state. У Setup Store таку дію реалізують самостійно, як у прикладі зі count.

Очищення сеансу може додатково вилучати токен, дані користувача та підтверджені локальні зміни. Якщо потрібно скасувати запит, це явно додається до дії очищення. Саме скидання state не скасовує fetch.

7. Взаємодія кількох stores

Store може отримати інший store через його функцію доступу. Наприклад, сховище товарів читає сеанс зі сховища користувача перед приватним запитом. У Setup Store залежність отримується всередині функції сховища; у getter або action також можна викликати use...Store.

Взаємні залежності не повинні утворювати нескінченний цикл створення або обчислення. Якщо два Setup Store безпосередньо читають state одне одного під час ініціалізації, створення може не завершитися. Залежне читання переносять у computed або action без циклічного виклику.

8. Спостереження за змінами

`watch()` спостерігає за вибраним реактивним джерелом, наприклад `watch(() => counter.count, ...)`. Цього достатньо для реакції на зміну окремого значення.

`store.$subscribe()` спостерігає за state сховища. Callback отримує опис зміни та поточний стан. Поле mutation.type відрізняє direct, patch object і patch function. Після функціонального `$patch()` підписка за типової конфігурації повідомляє про згруповану зміну один раз. Метод повертає функцію відписки.

Підписка, створена під час setup компонента, автоматично очищується після його демонтажу. `{ detached: true }` зберігає її незалежно від компонента й потребує явного керування строком дії. Підписка, створена під час ініціалізації Setup Store, належить області дії сховища, а не першого компонента, який його отримав.

Ні watch, ні $subscribe самі не записують дані до Web Storage. Pinia також не відновлює state після reload автоматично. Збереження в браузері та відновлення є окремими операціями застосунку.

9. Автентифікація як приклад спільного стану

Після входу користувача змінюється одразу кілька частин застосунку. Шапка показує ім’я та кнопку виходу, приватні сторінки стають доступними, а запити до захищених ресурсів отримують дані для підтвердження сеансу. Якщо кожна частина окремо запам’ятовує факт входу, ці значення можуть розійтися після виходу або завершення сеансу.

Спільне сховище auth зберігає відомості про сеанс і дії для його зміни. Компоненти та маршрутизатор використовують цей store як єдине джерело

даних. Вхід і вихід змінюють його стан, а залежні частини інтерфейсу реагують на ці зміни без окремих повідомлень одна одній.

Автентифікація — перевірка того, ким є користувач.

Авторизація — визначення дозволених для користувача дій.

Access token — токен, який клієнт передає серверу для доступу до захищеного ресурсу протягом визначеного строку.

9.1. State та похідні значення

Сховище містить дані, потрібні кільком частинам застосунку.

Поле state	Призначення
user	Профіль користувача, отриманий після успішного входу або перевірки сеансу
accessToken	Токен для запитів до захищених ресурсів
status	Поточний стан входу або перевірки сеансу
error	Повідомлення про невдалу операцію

Пароль залишається у формі й передається action входу як аргумент. Він не потрібний шапці, маршрутизатору чи іншим сховищам, тому не зберігається у спільному state.

Окремого прапорця «користувач увійшов» недостатньо. Поки сервер перевіряє сеанс, результат ще невідомий; мережева помилка також не означає підтвердженого входу. Поле status може розрізняти idle (перевірка ще не почалася), restoring (триває перевірка), signing-in (триває вхід), guest, authenticated та unavailable (перевірку не завершено через помилку).

Getter isAuthenticated обчислюється зі status, accessToken та user. Він істинний лише для підтвердженого стану authenticated з токеном і профілем. Getter isBusy визначає, чи триває restoring або signing-in. Компоненти читають ці значення, а не підтримують власні копії прапорців.

Action login записує профіль і токен після успішного входу, restore перевіряє збережений сеанс, logout очищує його. Токен у sessionStorage переживає reload, але потребує перевірки сервером. До її завершення isAuthenticated залишається хибним. Відповідь 401 завершує сеанс, а мережева помилка залишає доступ закритим зі станом unavailable і можливістю повторити перевірку.

9.2. Метадані маршрутів і робота гардів

Стан сеансу відповідає на питання, чи підтверджено вхід користувача. Для рішення про перехід потрібна ще одна відомість — чи потребує цільова сторінка входу. Цю вимогу зберігають у конфігурації маршруту.

Поле meta є об’єктом додаткових відомостей у записі маршруту. Наприклад, власна властивість requiresAuth позначає потребу у вході. Її назва й

поведінка визначаються кодом застосунку. Сам Vue Router не перевіряє сеанс лише через наявність цієї властивості. Це метадані конфігурації, а не параметри URL.

У фрагменті масиву маршрутів компоненти вважаються вже імпортованими. Вимога входу задається для спільної приватної оболонки.

```
{
  path: '/manage/products',
  component: ManageLayout,
  meta: { requiresAuth: true },
  children: [
    { path: '', name: 'manage-products',
      component: ManageProductsView },
    { path: ':id/edit', name: 'manage-edit',
      component: ProductEditView },
  ],
}
```

Адреса /manage/products/12/edit зіставляється з батьківським і дочірнім записами. Масив to.matched містить обидва записи, а to.meta об’єднує їхні метадані від батьківського до дочірнього. Тому requiresAuth доступна й для списку, і для редактора. Значення дочірнього запису за тим самим ключем перевизначає батьківське; вкладені об’єкти не зливаються рекурсивно.

Навігаційний гард — функція, яка перевіряє перехід і може дозволити його продовження, скасувати або перенаправити на іншу адресу.

Глобальний beforeEach отримує to — ціль переходу та from — поточний маршрут. Асинхронний гард може очікувати перевірку сеансу; до завершення гардів перехід залишається непідтвердженим.

```
router.beforeEach(async to => {
  const auth = useAuthStore(pinia)
  await auth.restore()
  if (to.meta.requiresAuth && !auth.isAuthenticated) {
    return { name: 'login', query: { redirect: to.fullPath } }
  }
})
```

Спочатку гард отримує auth store та очікує restore. Під час початкової перевірки дія звертається до сервера за наявності токена; наступні виклики використовують уже отриманий результат. Без токена вона встановлює guest без HTTP-запиту. Кілька одночасних викликів можуть очікувати одну перевірку, якщо restore зберігає Promise поточного запиту.

Далі гард читає to.meta.requiresAuth та auth.isAuthenticated. Якщо маршрут приватний, а вхід не підтверджено, повертається нова ціль login. Поточний перехід замінюється новим переходом на форму входу. В інших випадках функція завершується без поверненого значення, і маршрутизатор продовжує решту перевірок (рис. 1).

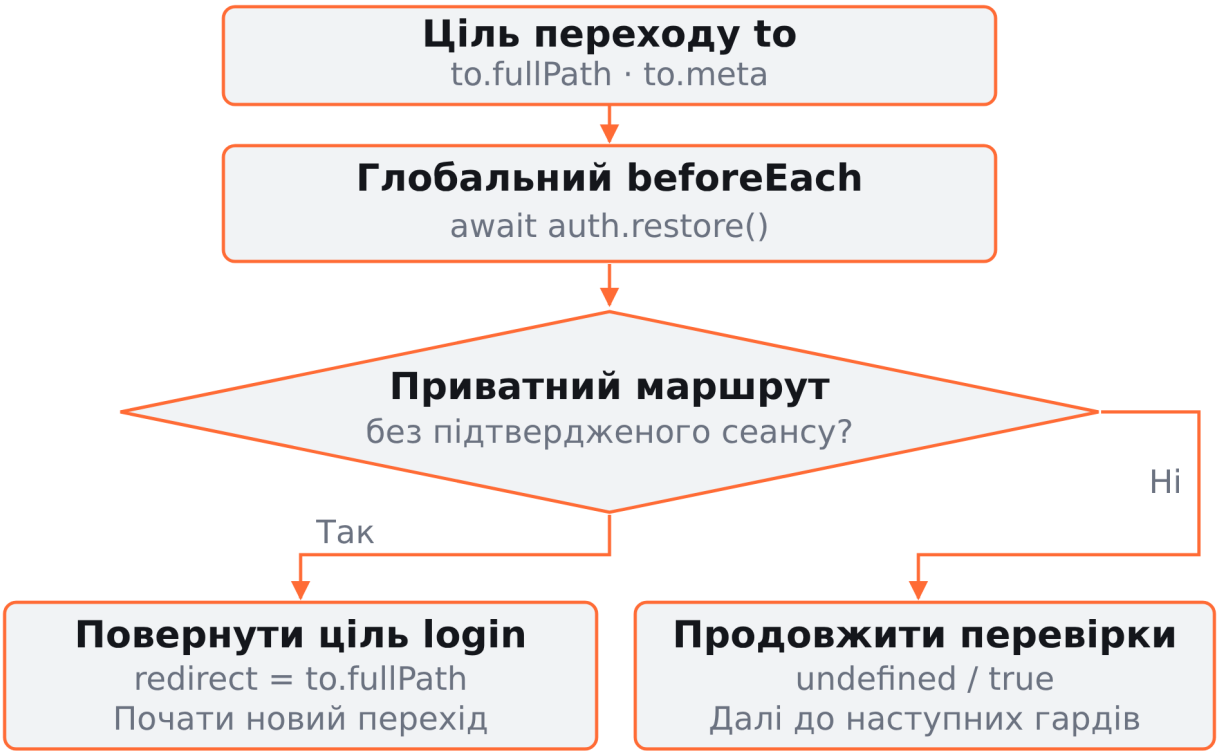

Рисунок 1 — Рішення глобального гарда за метаданими маршруту та станом Pinia.

Форма входу є публічним маршрутом, інакше перенаправлення може зациклитися. Поле query.redirect зберігає to.fullPath для повернення після входу. Перед переходом його перевіряють як дозволену внутрішню адресу; зовнішня чи невідома адреса не приймається. Після успішного входу router.replace відкриває перевірену ціль замість форми входу.

Store у гарді отримується після підключення Pinia або з явним аргументом її екземпляра, як у фрагменті вище. У main.js Pinia підключається перед маршрутизатором, щоб початкова навігація вже мала доступ до спільного стану.

9.3. Використання спільного стану

Форма входу викликає login, показує error і використовує isBusy. Шапка читає user та isAuthenticated і викликає logout. Приватна оболонка показує вміст лише за істинного isAuthenticated. Усі ці споживачі отримують один auth store через useAuthStore(). Якщо стан деструктурується в компоненті, реактивний зв’язок зберігається через storeToRefs().

Action logout очищує user, accessToken і збережений токен та встановлює guest. Зміна спільного стану прибирає ім’я з шапки й закриває приватний вміст. Пов’язані сховища очищують дані поточного сеансу. Незавершені запити скасовуються, а пізні відповіді перевіряються, щоб не відновити вже завершений сеанс. Цю поведінку реалізує код застосунку.

Приватні запити передають токен зі store в `Authorization: Bearer …`. Відповідь 401 для поточного сеансу запускає його очищення. Якщо сеанс

завершується на вже відкритій сторінці, застосунок приховує приватний вміст і виконує відповідний перехід; одного гарда переходів для цього недостатньо.

Store керує станом клієнтського інтерфейсу. Наявність токена, його розбір або дозвіл гарда не замінюють перевірку доступу сервером. Локальний вихід не відкликає виданий сервером токен. Refresh token та серверне відкликання сеансу є окремими механізмами.

Функціональні вимоги

- Публічний каталог, пошук із q, сторінки товарів, відгуки й крихти роботи 6 залишаються працездатними
- Гість, який напряму відкрив приватний список або редактор, потрапляє до форми входу; приватні дані не показуються до підтвердження сеансу
- Неправильні облікові дані показують помилку без створення сеансу; успішний вхід оновлює шапку та відкриває запитану дозволену приватну адресу
- Reload перевіряє збережений access token через профіль; неприйнятний токен вилучається, мережева помилка дозволяє повторити перевірку без відкриття приватного вмісту
- Приватний список показує назву, ціну, залишок і посилання редагування товарів; доступне завантаження наступної та попередньої сторінок
- Редагуються назва, ціна, залишок та опис; назва після trim має 1–120 символів, опис — 1–2000, ціна є додатним скінченним числом, залишок — цілим невід'ємним числом; порожні числові поля не приймаються
- До успішного PATCH чернетка не змінює підтверджені дані; помилка збереження лишає чернетку, а скасування відновлює поточні підтверджені значення без HTTP-запиту
- Під час входу й збереження повторне натискання не створює паралельної операції; незбережені зміни потребують підтвердження перед виходом із редактора або зміною id
- Після успіху список керування, публічна сторінка, її крихта та вже завантажені картки показують підтверджені зміни поточного сеансу; інтерфейс пояснює їх втрату після reload або виходу
- Вихід і 401 приватного API очищують сеанс і локальні підтверджені зміни; пізні відповіді не повертають очищений стан та не переривають новий сеанс
- За 360×800, 768×1024, 1440×900 і 844×390 немає горизонтального переповнення та обрізаних полів; форми доступні з клавіатури, нові дії мають область натискання щонайменше 44×44 px
- Видимі ПІБ, група й відомості про середовище виконавця збережені

Технічні вимоги

- lab7 створено на основі власного lab6 без копіювання node_modules, dist і вкладеного .git; використано JavaScript, Composition API та script setup
- Pinia 3.0.4 встановлено локально з точним номером версії; Vue узгоджено з 3.5.43, Vue Router — з 4.6.4; package-lock.json збережено
- Один createPinia підключено перед маршрутизатором; auth і products є окремими Setup Store; state задається через ref, похідні значення — computed, операції — actions; для деструктуризації стану використано storeToRefs
- Пароль і чернетка форми є локальними; auth містить токен, перевірений профіль, стан і помилку, products — лише підтверджені зміни за id; власні дії очищення реалізовано явно
- Маршрути login, manage-products і manage-edit належать адресам /login, /manage/products і /manage/products/:id/edit; приватну оболонку позначено meta.requiresAuth, глобальний beforeEach очікує початкову перевірку сеансу
- Ціль повернення з query перевіряється за дозволеними внутрішніми маршрутами; зовнішні й невідомі адреси та сама форма входу не використовуються як цілі повернення
- Вхід виконується через POST /auth/login, профіль через GET /auth/me; приватне читання та PATCH використовують /auth/products і заголовок Authorization Bearer; токени й паролі не передаються в URL
- У sessionStorage зберігається лише access token; наявність токена не замінює серверної перевірки, пароль і refresh token не зберігаються; credentials omit задає обрану модель із Bearer без cookie
- HTTP-модуль перевіряє response.ok, відрізняє 401 від інших помилок і підтримує AbortSignal; завантаження очищується при зміні джерела та демонтуванні
- Збереження перевіряє актуальність операції й сеансу перед зміною store; завершення сеансу скасовує операції й очищує словник підтверджених змін; вони не записуються в Web Storage
- Форми мають пов’язані підписи, повідомлення про помилки та стан операції; використано onBeforeRouteLeave, onBeforeRouteUpdate й очищуваний beforeunload для незбережених змін
- Збережено оформлення каталогу, нові елементи використовують BEM; бібліотеки готової авторизації та автоматичного збереження Pinia не застосовуються

Необхідне обладнання та матеріали

- Власний завершений Vue-проєкт лабораторної роботи 6
- Node.js 24 LTS, npm, Git і редактор коду
- Vue 3.5.43, Vue Router 4.6.4 та Pinia 3.0.4

- Браузер із DevTools, Network та емуляцією розмірів екрана
- Доступ до навчальних Auth і Products API DummyJSON

Хід роботи

1. Підготовка проєкту

Скопіюйте власний завершений проєкт lab6 до нового каталогу lab7 у корені репозиторію mjsf-mag. Не копіюйте node_modules, dist і вкладений каталог .git. Збережіть персональні відомості на сторінці.

Із lab7 виконайте окремими командами:

```
npm ci
npm install --save-exact vue@3.5.43 vue-router@4.6.4 pinia@3.0.4
npm run dev -- --port 5177 --strictPort
```

Відкрийте http://localhost:5177. Перевірте каталог, пошук із q, сторінку товару, відгуки та крихти. Початковий результат має відповідати власній роботі 6. Dev-сервер залиште запущеним; інші команди виконуйте в окремому терміналі з каталогу lab7.

2. Власники стану та структура

У README опишіть відповідальність модулів. Пароль належить формі входу, чернетка — редактору. Сеанс є спільним для шапки, гарда та приватних запитів. Підтверджені зміни товарів також спільні для кількох представлень. Пошук і id продовжують надходити з маршруту, як у роботі 6.

Додайте до src такі модулі:

```
src/
├── stores/
│   ├── index.js
│   ├── auth.js
│   └── products.js
├── api/
│   ├── http.js
│   └── productDraft.js
├── auth/
│   └── session.js
├── composables/
│   └── usePrivateResource.js
├── router/
│   └── guards.js
├── layouts/
│   └── ManageLayout.vue
└── views/
    ├── LoginView.vue
    ├── ManageProductsView.vue
    └── ProductEditView.vue
```

У stores/index.js створіть один createPinia. У main.js підключіть його перед router і до монтування застосунку. Опишіть auth і products через defineStore у формі Setup Store; почніть із порожнього об’єкта повернення. Викличте обидва use...Store у setup компонента App, щоб побачити два сховища у DevTools. Стан і дії додаються на етапах 4, 7 та 8. Локальні поля форм до цих сховищ не переносяться.

3. Запити до навчального API

У http.js реалізуйте запит JSON із перевіркою response.ok. Помилка має зберігати HTTP-статус. Підтримайте AbortSignal і необов’язковий токен у заголовку Authorization. Для передавання тіла використовуйте JSON.stringify та Content-Type application/json. У цій моделі задайте credentials omit, оскільки клієнт явно передає Bearer token.

Базова адреса API — https://dummyjson.com. Використайте такі операції:

Операція	**Метод та шлях**	**Результат**
Вхід	POST /auth/login	Профіль і accessToken
Перевірка сеансу	GET /auth/me	Профіль за переданим токеном
Приватний список	GET /auth/products?limit=12&skip=0	Сторінка товарів
Приватний товар	GET /auth/products/:id	Дані редактора
Збереження	PATCH /auth/products/:id	Імітація оновленого товару

Для входу використайте публічні навчальні дані username emilys, password emilyspass та expiresInMins 30. Неправильний пароль у цьому API дає 400. Приватний запит без прийнятного токена дає 401.

Одержані токени й пароль не виводьте в консоль, URL або звіт. У Network перевіряйте наявність заголовка та статус операції без копіювання значення токена. Публічні запити каталогу роботи 6 залиште на звичайних /products.

4. Сховище сеансу й форма входу

У auth створіть accessToken, user, status та error. Обчислюйте isAuthenticated за підтвердженим станом сеансу, токеном і профілем. Стани мають відрізняти початкову перевірку, гостя, вхід, успіх і недоступність перевірки через мережу.

Додайте публічний маршрут /login із LoginView до чинної конфігурації. Форма викликає login; повернення до початкової адреси додається на етапі 5. Дія login надсилає облікові дані й після успіху оновлює спільний стан. Пароль лишається в LoginView і очищується після передавання запиту. Поки дія триває, повторний запуск заблокований. Помилка показується у формі; невдалий вхід не створює токена.

У sessionStorage зберігайте лише accessToken під власним ключем. Дія restore перевіряє його через /auth/me перед приватною навігацією. Кілька

одночасних викликів очікують одну запущену перевірку. За 401 очистіть сеанс; за мережевої помилки залиште можливість повторити перевірку, але не вважайте сеанс підтвердженим. Додайте відповідну кнопку.

У шапці покажіть ім’я після входу та дію виходу. Для деструктуризації state і getter використайте storeToRefs. Logout очищує токен, профіль і sessionStorage. Пізня відповідь login або restore після виходу не повинна повернути сеанс.

На /login перевірте неправильний і правильний пароль: помилку або підтверджений профіль у шапці. Перевірки reload і підробленого токена виконайте після підключення гарда на етапі 5.

5. Приватні маршрути та повернення після входу

Доповніть конфігурацію маршрутів. /login уже створено на етапі 4; додайте до нього приватні маршрути з таблиці:

Шлях	**Ім’я**	**Доступ**
/login	login	Форма входу
/manage/products	manage-products	Після підтвердження сеансу
/manage/products/:id/edit	manage-edit	Після підтвердження сеансу

Створіть запис /manage/products із ManageLayout і meta.requiresAuth true. У children порожній шлях показує список, :id/edit — редактор. Поки що додайте до цих двох представлень заголовки, а в редакторі покажіть id із маршруту; дані та поля додаються на наступних етапах. Оболонка містить RouterView і показує приватний вміст лише для підтвердженого сеансу.

У глобальному beforeEach одержіть auth із потрібного екземпляра Pinia та дочекайтеся restore. Якщо ціль потребує входу, поверніть маршрут login із query.redirect, що містить to.fullPath. Після успіху форма відкриває цю ціль через replace. Приймайте лише внутрішні цілі, які router.resolve зіставляє з manage-products або manage-edit; для решти відкривайте список.

Окремо обробіть завершення сеансу на вже відкритій приватній сторінці. За 401 приберіть її вміст і відкрийте вхід, а після звичайного виходу — публічний каталог. Гард втрати чернетки не повинен утримувати редактор після завершення сеансу.

Відкрийте /manage/products/12/edit у новій сесії браузера без входу. Спочатку має з’явитися LoginView, а після успіху — приватне представлення з id 12. Після reload виконується одна перевірка /auth/me. Підроблений токен у sessionStorage не відкриває це представлення. Зовнішній URL у redirect не повинен виконуватися.

6. Приватне читання й список товарів

У auth реалізуйте дію request, яка передає поточний токен приватному API та очищує той самий сеанс за 401. Помилка попереднього токена не повинна очищувати новий сеанс, якщо за час очікування користувач увійшов знову.

У usePrivateResource організуйте реактивне читання через цю дію. Стан data, isLoading та error належить composable. При зміні адреси очистіть попередні дані; через очищення watch скасуйте старе завантаження. Застаріла відповідь не застосовується після зміни джерела або демонтування.

ManageProductsView показує по 12 товарів із назвою, ціною, залишком і RouterLink до manage-edit відповідного id. Додайте попередню й наступну сторінки через skip. Поки триває запит, заблокуйте повторне перемикання. Покажіть завантаження та помилку з повтором.

Доповніть власний контракт крихт роботи 6. Приватна оболонка задає рівень «Керування», редактор додає реактивну назву товару. Порожній дочірній шлях не створює дубліката.

7. Форма редагування та збереження

ProductEditView читає id із маршруту й завантажує приватний товар. Некоректний id не повинен створювати запит; 404 показує окремий стан. Створіть локальну чернетку з title, price, stock і description. Не використовуйте сам завантажений об’єкт як модель полів.

У productDraft.js опишіть нормалізацію й перевірку:

- title після trim містить 1–120 символів;
- price є скінченним числом більшим за 0;
- stock є цілим числом від 0, порожнє поле не приймається;
- description після trim містить 1–2000 символів.

Помилки покажіть біля пов’язаних полів через aria-describedby і aria-invalid; після спроби некоректного збереження переведіть фокус до першого поля з помилкою. Запиту PATCH у такому випадку немає.

Дія save у products одержує id та копію чернетки. Передайте лише чотири редаговані поля; id не редагується і не надходить із поля форми. Поки запит очікує відповіді, заблокуйте поля й повторне збереження. За помилки залиште чернетку. «Скасувати зміни» відновлює поточні підтверджені значення без запиту.

Порівнюйте чернетку з початковими значеннями. За незбережених змін onBeforeRouteLeave і onBeforeRouteUpdate запитують підтвердження їх втрати. Beforeunload окремо обробляє reload або закриття документа; обробник вилучається після демонтування. Вигляд цього попередження визначає браузер.

8. Спільний результат і завершення сеансу

Після успішного PATCH запишіть підтверджені поля до словника products за id. Перед записуванням перевірте, чи актуальні операція та сеанс. Зміна локального поля до відповіді API не повинна оновлювати словник.

Створіть спосіб одержати відображуваний товар як поєднання прочитаних даних і підтверджених змін поточного сеансу. Використайте його у списку керування, публічній сторінці й завантажених картках. Назва крихти також має оновитися. Відповідь /auth/products не зберігає зміни на сервері, тому новий GET сам по собі не дасть цього результату.

У приватному розділі поясніть, що зміни діють до reload або виходу. Не записуйте словник у Web Storage. Серверний пошук продовжує працювати з початковими даними DummyJSON; локальні зміни не перебудовують його набір.

Власна дія reset очищує словник і скасовує збереження. Викликайте її після завершення сеансу. Пізня відповідь PATCH після виходу не повинна повернути старий стан або змінити дані нового користувача.

9. Перевірка поведінки та адаптивності

Підготуйте таблицю з колонками «Сценарій», «Очікуваний результат», «Фактичний результат», «URL», «HTTP-статус» і «Кількість запитів». Виконайте такі перевірки:

1. Пряма приватна адреса гостя відкриває вхід без показу редактора; після успіху відновлюється ціль.
2. Неправильний пароль показує помилку; швидке повторне натискання не створює другого POST.
3. Reload із чинним токеном виконує /auth/me; підроблений токен вилучається й не дає доступу.
4. Тимчасово додайте delay=1500 до запиту /auth/me, виконайте reload і дочекайтеся початку цього запиту в Network. Увімкніть Offline. Після помилки приватний вміст закритий, а токен збережений для повтору. Поверніть Online, повторіть перевірку сеансу та приберіть діагностичну затримку.
5. Приватний список перегортається, а редагування відкриває правильний id; некоректний id не створює запиту, відсутній товар дає окремий стан.
6. Порожня назва, нульова ціна, дробовий або порожній залишок і порожній опис показують помилки без PATCH; фокус переходить до першої помилки.
7. Зміна поля не оновлює список; скасування відновлює дані без запиту. Відмова у підтвердженні виходу лишає чернетку відкритою.
8. Успішний PATCH оновлює приватний список, публічну сторінку й крихту; після reload повертаються серверні значення, але сеанс перевіряється знову.
9. Помилка PATCH не змінює підтверджених даних і не стирає чернетку; після відновлення мережі збереження можна повторити.
10. Увімкніть затримку, почніть збереження й одразу вийдіть. Пізня відповідь не повинна повернути сеанс або зміни; повторний вхід починає роботу без попередніх локальних змін.
11. Для перевірки 401 тимчасово передайте в HTTP-модулі для одного запиту /auth/products заголовок Authorization: Bearer invalid замість одержаного токена. Значення токена у store не змінюйте.

Переконайтеся, що клієнт завершує сеанс, потім приберіть діагностичну зміну.

12. Перевірте вхід, список і редактор за 360×800, 768×1024, 1440×900 і 844×390; пройдіть поля й дії через Tab, Shift+Tab та Enter.

За потреби додавайте delay до запиту DummyJSON в межах, дозволених API. У Network відрізняйте запит JSON від завантаження коду компонента. Збережіть знімки з deliverables без значень паролів і токенів.

10. Виробнича збірка та завершення

Зупиніть dev-сервер через Ctrl+C. Виконайте:

```
npm run build
npm exec vite preview -- --port 5177 --strictPort
```

Через http://localhost:5177 перевірте пряму адресу редактора, вхід, reload, збереження та вихід. Vite preview перевіряє локальну збірку; публікація сайту належить до окремої теми deployment.

Зупиніть preview через Ctrl+C. У README зафіксуйте відповідальність модулів, маршрути, життєвий цикл і межі навчальної моделі. Збережіть lock-файл та переконайтеся, що node_modules і dist не потрапляють до Git. Із кореня власного репозиторію виконайте:

```
git add lab7
git commit -m "Лабораторна 7: Pinia та приватний редактор"
git push
```

Що здається

Здаються артефакти з поля deliverables у lab.yaml. Застосовуються загальні правила звіту й репозиторію курсу. Каталог роботи — lab7.

Вимоги до змісту та оформлення звіту

- Файл звіту завантажується до СЕЗН Moodle у форматі PDF.
- Звіт містить титульний аркуш і деталі виконання роботи.
- Звіт містить посилання на репозиторій GitHub, Bitbucket або GitLab із роботою в каталозі `lab1`, `lab2`, … за номером роботи.
- Звіт містить команди, які виконуються під час роботи.
- Звіт містить знімки екрана, з яких зрозумілий контекст роботи та видно персональні дані автора: адресний рядок, дані студента й відомості про середовище.

Крім переліченого, за цією роботою здається:

- Вихідний код lab7 із Setup Store, HTTP-модулем, гардом, приватними представленнями та package-lock.json

- README з відповідальністю модулів, таблицею маршрутів, життєвим циклом сеансу й поясненням меж DummyJSON та sessionStorage
- Таблиця сценаріїв з очікуваним і фактичним результатом, адресою, HTTP-статусом і кількістю операцій входу або PATCH
- Знімки форми входу, приватного списку, редактора, помилок валідації, успішного збереження та перевірочних розмірів; знімки Network без значень паролів і токенів
- Результат npm run build і перевірки прямих приватних адрес через vite preview

Контрольні питання

1. Який стан приватного редактора має бути спільним, а який має залишатися локальним?
2. Як ref, computed і функції Setup Store відповідають поняттям state, getters та actions?
3. Навіщо використовувати storeToRefs замість звичайної деструктуризації store?
4. Чим відрізняються автентифікація та авторизація?
5. Чому збережений access token перевіряється через профіль після reload?
6. Чому beforeEach із requiresAuth не замінює перевірку захищеного API?
7. Коли чернетка форми має змінювати спільні підтверджені дані?
8. Чому новий GET DummyJSON після успішного PATCH повертає початковий товар?

Література до роботи

1. Vue. State Management — https://vuejs.org/guide/scaling-up/state-management.html
2. Vue. Provide / Inject — https://vuejs.org/guide/components/provide-inject.html
3. Vue. Composables — https://vuejs.org/guide/reusability/composables.html
4. Pinia. Introduction — https://pinia.vuejs.org/introduction.html
5. Pinia. Defining a Store — https://pinia.vuejs.org/core-concepts/
6. Pinia. State — https://pinia.vuejs.org/core-concepts/state.html
7. Pinia. Getters — https://pinia.vuejs.org/core-concepts/getters.html
8. Pinia. Actions — https://pinia.vuejs.org/core-concepts/actions.html
9. Pinia. Composing Stores — https://pinia.vuejs.org/cookbook/composing-stores.html
10. Pinia. Stores outside of components — https://pinia.vuejs.org/core-concepts/outside-component-usage.html
11. Vue Router. Navigation Guards — https://router.vuejs.org/guide/advanced/navigation-guards.html
12. Vue Router. Composition API — https://router.vuejs.org/guide/advanced/composition-api.html
13. Vue Router. Route Meta Fields — https://router.vuejs.org/guide/advanced/meta.html
14. Vue. Form Input Bindings — https://vuejs.org/guide/essentials/forms.html
15. DummyJSON. Auth — https://dummyjson.com/docs/auth
16. DummyJSON. Authorizing Resources — https://dummyjson.com/docs
17. DummyJSON. Products — https://dummyjson.com/docs/products
18. MDN. Using Fetch — https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
19. MDN. sessionStorage — https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage
20. OWASP. HTML5 Security — https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html