САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Фронтэнд разработка

Отчет

Лабораторная работа №3 “Разработка одностраничного веб-приложения (SPA) с использованием фреймворка Vue.JS”

Выполнил:

Хисаметдинова Д.Н.

Группа
K3441

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Задание: Мигрировать ранее написанный сайт на фреймворк Vue.JS.

Минимальные требования:

- Должен быть подключён роутер
- Должна быть реализована работа с внешним API
- Разумное деление на компоненты
- Использование composable

Ход работы

1. **Архитектура системы**

   1.1 Файловая структура:

   src/

   ├── components/ # Переиспользуемые компоненты

   │ ├── SearchForm.vue # Форма поиска с геолокацией

   │ ├── PropertyCard.vue # Карточка недвижимости

   │ ├── PropertyList.vue # Список свойств

   │ ├── AppNavbar.vue # Навигация с темами

   │ ├── AppFooter.vue # Футер

   │ ├── TravelBenefits.vue # Преимущества путешествий

   │ └── icons/ # SVG иконки

   │ ├── IconCalendar.vue

   │ ├── IconCoins.vue

   │ ├── IconSale.vue

   │ ├── IconSnooze.vue

   │ └── UshankaSvg.vue

```
├── views/              # Страницы-роуты
│   ├── HomeView.vue    # Главная страница с поиском
│   ├── SearchView.vue  # Страница результатов поиска
│   ├── PropertyView.vue # Детальная страница свойства
│   ├── MessagesView.vue # Страница сообщений
│   ├── ProfileView.vue # Профиль пользователя
│   ├── LoginView.vue   # Страница входа
│   └── RegisterView.vue # Страница регистрации
├── composables/        # Композиционные функции
│   ├── useApiService.ts # API сервис с моками
│   └── useTheme.ts     # Управление темами
├── stores/             # Pinia store
│   └── auth.ts         # Состояние аутентификации
├── router/             # Vue Router
│   └── index.ts        # Маршруты и навигационные guards
└── types/              # TypeScript типы
    └── index.ts        # Общие интерфейсы
```

1.2 Используемые паттерны

— Composition API: Использование ref(), computed(), onMounted() для управления состоянием

— Composables: Вынесение логики в переиспользуемые функции (useApiService, useTheme)

— Component-based: Разделение на независимые компоненты с четкой ответственностью

2. API и работа с данными

2.1 Структура API слоя

Приложение использует композиционную функцию useApiService, которая создает axios инстанс с базовыми настройками:

```
const API_BASE_URL = 'http://localhost:8000'

const api = axios.create({

 baseURL: API_BASE_URL,

 headers: {

   'Content-Type': 'application/json',

 }

})
```

Настроены два хэндлера запросов:

- Request interceptor автоматически добавляет JWT токен из localStorage к каждому запросу,
- Response interceptor обрабатывает 401 ошибки и перенаправляет на страницу входа.

2.2 Типы данных и их обработка

Основные TypeScript интерфейсы:

```
interface Property {

 id: number

 title: string

 location: string

 type: string

 price: number

 rating: number

 reviews: number

 image: string

 amenities: string[]
```

```
 maxGuests: number
 bedrooms: number
 bathrooms: number
 description: string
}
interface LoginResponse {
  success: boolean
  user?: any
  token?: string
  message?: string
}
interface RegisterResponse {
  success: boolean
  user?: any
  token?: string
  message?: string
}


interface PropertiesResponse {
  success: boolean
  data?: Property[]
  message?: string
}
interface Message {
 id: number
 text: string
 isFromGuest: boolean
 timestamp: string
}
```

2.3 Архитектура данных

Присутствует первичный источник получения данных в виде HTTP-запросов к бэкенду, а также дополнительный – локальные mock данные при недоступности бэкенда. Из внешних API это OpenStreetMap Nominatim для геолокации и поиска отелей, а также https://www.bigdatacloud.com/reverse-geocoding для перевода из значений долготы и широты в человекочитаемую геолокацию. Предусмотрено кэширование в LocalStorage для внешних свойств.

2.4 API-методы

Методы аутентификации:

— login(email, password) - вход пользователя с JWT токеном,

— register(userData) - регистрация нового пользователя,

— logout(userId) - выход пользователя.

Методы работы с недвижимостью:

— getProperties(filters) - получение списка с фильтрацией на бэкенде,

— getPropertyById(id) - получение конкретного свойства,

— getFallbackProperties(filters) - резервные данные при недоступности API.

Геолокационные методы:

— getUserLocation() - получение координат пользователя,

— searchHotelsNearLocation(lat, lng) - поиск отелей рядом с координатами,

— reverseGeocode(lat, lng) - получение информации о местоположении,

— getNearbyHotels(lat, lng) - поиск через OpenStreetMap API

3. Управление состоянием

3.1 Архитектура управления состоянием

— pinia Store - для глобального состояния (аутентификация),

— composition API - для локального состояния компонентов,

— localStorage - для персистентности данных,

— composables - для переиспользуемой логики состояния.

3.2 Глобальное состояние

3.2.1 Auth Store - Централизованная аутентификация

Сначала объявлена переменная для PiniaStore. Далее создаются реактивная переменная isAuthenticated (реактивность — это способность автоматически обновлять зависимые части приложения при изменении данных), зависящая от user.

```
export const useAuthStore = defineStore('auth', () => {
 const user = ref<User | null>(null)
 const token = ref<string | null>(null)
 const isLoading = ref(false)
 const error = ref<string | null>(null)
 const { apiService } = useApiService()
 const initializeAuth = () => {
   const savedToken = localStorage.getItem('authToken')
   const savedUser = localStorage.getItem('user')
   if (savedToken && savedUser) {
     token.value = savedToken
     user.value = JSON.parse(savedUser)
   }
 }
 const isAuthenticated = computed(() => {
   return !!token.value && !!user.value
 })
```

Для объяснения реактивности приведу пример, как это происходит в контексте логина:

1. Исходное состояние

user: null

isLoading: false

isAuthenticated: false (вычисляется из user)

2. Начало логина

isLoading: true ⬅ изменение

isAuthenticated: false

3. Успешный логин

user: { id: 1, name: 'John' } ⬅ изменение

isLoading: false ⬅ изменение

isAuthenticated: true ⬅ АВТОМАТИЧЕСКОЕ изменение

4. Компонент перерисовывается

Вместо формы логина показывается профиль пользователя

Вот различие переменных с ref(), создающей реактивную ссылку и без:

Обычная переменная (НЕ реактивная) – Если count изменится, Vue об этом не узнает:

let count = 0

Реактивная переменная — когда count изменится, Vue обновит все зависимые компоненты

const count = ref(0)

Состояние Auth Store:

— user: User | null - данные авторизованного пользователя,

— token: string | null - JWT токен для API запросов,

— isLoading: boolean - индикатор загрузки операций аутентификации,

— error: string | null - ошибки при входе/регистрации.

Computed свойства:

— isAuthenticated - реактивная проверка авторизации пользователя

Actions (методы изменения состояния):

— initializeAuth() - восстановление сессии при загрузке приложения,

— login(email, password) - процесс входа с обновлением состояния,

— register(userData) - регистрация нового пользователя,

— logout() - выход с полной очисткой состояния и localStorage,

— clearError() - сброс ошибок аутентификации.

3.2.1 Синхронизация с localStorage

```
const login = async (email: string, password: string) => {
  isLoading.value = true
  error.value = null
  try {
    const response = await apiService.login(email, password)
    if (response.success) {
      user.value = response.user
      token.value = response.token || null
      return { success: true }
    } else {
      error.value = response.message || 'Login failed'
```

```
      return { success: false, message: error.value }
    }
  } catch (err) {
    error.value = 'Login failed'
    return { success: false, message: error.value }
  } finally {
    isLoading.value = false
  }
}
```

3.3 Локальное состояние компонентов

3.3.1 SearchView.vue - Состояние поиска и фильтрации

```
const properties = ref<Property[]>([])
const isLoading = ref(false)
const isLoadingLocation = ref(false)
const searchSource = ref<string>('')
const searchSourceText = ref<string>('')
const searchQuery = ref({
 location: (route.query.location as string) || '',
 checkIn: (route.query.checkIn as string) || '',
 checkOut: (route.query.checkOut as string) || '',
 guests: (route.query.guests as string) || '1'
})
const filters = ref({
 types: [] as string[],
 minPrice: null as number | null,
 maxPrice: null as number | null,
 minRating: null as number | null,
 amenities: [] as string[],
 adults: 1,
 children: 0,
 checkIn: '',
```

```
  checkOut: ''

})
```

4. Composables

4.1 Для чего они нужны

Composables - это функции, которые используют Composition API Vue для инкапсуляции и переиспользования реактивной логики состояния. По сути, это паттерн для извлечения логики из компонентов в отдельные переиспользуемые функции.

Выходит, вместо дублирования логики в компонентах:

```
const ComponentA = {
  data() {
    return { count: 0 }
  },
  methods: {
    increment() { this.count++ }
  }
}
```

выносится в composable и переиспользуеся везде.

```
function useCounter() {
  const count = ref(0)
  const increment = () => count.value++
  return { count, increment }
}
const { count, increment } = useCounter()
```

Преимущества над mixins и другими паттернами:

— Явные зависимости - видно откуда берутся данные,

— Type Safety - полная поддержка TypeScript,

— Нет конфликтов имен - можно переименовывать при импорте,

— Лучшая композиция - легко комбинировать логику.

4.2 useApiService.ts - Централизованный API слой

Главный composable приложения - useApiService представляет собой централизованный слой для всех взаимодействий с API. Этот подход решает несколько архитектурных задач:

Инкапсуляция HTTP логики

```
const api = axios.create({
 baseURL: API_BASE_URL,
 headers: {
   'Content-Type': 'application/json',
 }
})
```

Все HTTP конфигурации собраны в одном месте. Axios instance настраивается один раз и используется повсеместно, что обеспечивает консистентность запросов.

Автоматическая обработка аутентификации

```
api.interceptors.request.use(
 (config) => {
   const token = localStorage.getItem('authToken')
   if (token) {
     config.headers.Authorization = `Bearer ${token}`
   }
   return config
 },
```

```
 (error) => {
   return Promise.reject(error)
 }
)
```

Interceptors автоматически добавляют JWT токен к каждому запросу. Компонентам не нужно заботиться об аутентификации - это происходит прозрачно на уровне HTTP слоя.

Централизованная обработка ошибок

При получении 401 ошибки происходит автоматический логаут и редирект. Это избавляет от дублирования логики обработки ошибок в каждом компоненте.

```
api.interceptors.response.use(
 (response) => response,
 (error) => {
   if (error.response?.status === 401) {
     localStorage.removeItem('authToken')
     localStorage.removeItem('user')
     window.location.href = '/login'
   }
   return Promise.reject(error)
 }
)
```

Типизированные API методы

```
export const useApiService = () => {
  const apiService = {
    async login(email: string, password: string): Promise<LoginResponse> {
      // логика авторизации
    },
    async getProperties(filters?: any): Promise<PropertiesResponse> {
      // получение недвижимости с фильтрацией
```

```
  },

  async searchHotelsNearLocation(lat: number, lng: number):
Promise<HotelsSearchResponse> {

    // поиск отелей по геолокации

  }

 }

 return { apiService }

}
```

Интеграция с внешними API

```
  async searchHotelsNearLocation(latitude: number, longitude: number) {
    try {
      console.log('Searching hotels near:', latitude, longitude)
      const placeholderImages = [
        '/images/cozy-belgrade-apartment.webp',
        '/images/budapest-hotel.webp',
        '/images/cherno-more-hotel.webp',
        '/images/haludovo-palace.webp',
        '/images/modern-krakow-loft.webp',
        '/images/riga-luxury-suite.webp'
      ]
      const mockHotels: Property[] = Array.from({ length: 6 }, (_, index) => ({
        id: 1001 + index,
        title: `Hotel ${index + 1} near your location`,
        location: 'Near your location',
        type: 'hotel',
        price: Math.floor(Math.random() * 100) + 80,
        rating: parseFloat((Math.random() * 1.5 + 3.5).toFixed(1)),
        reviews: Math.floor(Math.random() * 100) + 10,
        image: placeholderImages[index],
        amenities: ['wifi', 'pool', 'restaurant', 'parking'],
```

```
        maxGuests: 4,

        bedrooms: 2,

        bathrooms: 1,

        description: `A nice hotel near your current location
(${latitude.toFixed(4)}, ${longitude.toFixed(4)})`

      }))

      return { success: true, data: mockHotels }

    } catch (error) {

      console.error('Error searching hotels:', error)

      return { success: false, message: 'Failed to search hotels' }

    }

  }

}

return { apiService }
```

4.2 Геолокация через Geolocation API

Помимо следованию документации по использованию api есть детальная обработка ошибок и все работает асинхронно.

```
  async getUserLocation() {

    return new Promise((resolve, reject) => {

      if (!navigator.geolocation) {

        reject(new Error('Geolocation is not supported by this browser'))

        return

      }

      navigator.geolocation.getCurrentPosition(

        (position) => {

          resolve({

            latitude: position.coords.latitude,

            longitude: position.coords.longitude,

            accuracy: position.coords.accuracy

          })

        },

        (error) => {
```

```
          let message = 'Location access denied'
          switch (error.code) {
            case error.PERMISSION_DENIED:
              message = 'Location access denied by user'
              break
            case error.POSITION_UNAVAILABLE:
              message = 'Location information unavailable'
              break
            case error.TIMEOUT:
              message = 'Location request timeout'
              break
          }
          reject(new Error(message))
        },
        {
          enableHighAccuracy: true,
          timeout: 10000,
          maximumAge: 600000
        }
      )
    })
  },
```

Использование в компонентах

В SearchView.vue через loadProperties происходит загрузка 18 объектов недвижимости и их характеристик из локального API

```
const loadProperties = async () => {
 isLoading.value = true
 try {
   const response = await apiService.getProperties()
   if (response.success && response.data) {
     properties.value = response.data
```

```
    }
  } catch (error) {
    console.error('Failed to load properties:', error)
  } finally {
    isLoading.value = false
  }
}
```

В SearchView.vue, const searchNearby происходит получение геопозиции по API, затем преобразования latitude и longitude в названия страны и региона тоже благодаря другой API.

```
const searchNearby = async () => {
  console.log('searchNearby function called')
  if (!navigator.geolocation) {
    alert('Geolocation is not supported by your browser')
    return
  }
  isLoadingLocation.value = true
  try {
    console.log('Requesting geolocation permission...')
    const position = await new Promise<GeolocationPosition>((resolve, reject) => {
      navigator.geolocation.getCurrentPosition(
        resolve,
        reject,
        {
          enableHighAccuracy: true,
          timeout: 15000,
          maximumAge: 0
        }
      )
    })
    console.log('Geolocation success:', position)
```

```
    const location = {
      latitude: position.coords.latitude,
      longitude: position.coords.longitude,
      accuracy: position.coords.accuracy
    }
    console.log('Location received:', location)
    const hotelsResponse = await apiService.searchHotelsNearLocation(
      location.latitude,
      location.longitude
    )
    console.log('Hotels response:', hotelsResponse)
    if (hotelsResponse.success && hotelsResponse.data) {
      properties.value = hotelsResponse.data
      searchQuery.value.location = 'Near your location'
      alert(`Found ${hotelsResponse.data.length} properties near your location!`)
    } else {
      alert('No properties found near your location')
    }
  } catch (error: any) {
    console.error('Geolocation error:', error)
    let message = 'Unable to get your location. '
    if (error.code) {
      switch (error.code) {
        case 1:
          message += 'Location access was denied. Please enable location services in
your browser.'
          break
        case 2:
          message += 'Location services are currently unavailable.'
          break
        case 3:
          message += 'Location request timed out. Please try again.'
```

```
          break
        default:
          message += 'An unknown error occurred.'
      }
    } else {
      message += error.message || 'Please try again or enter location manually.'
    }
    alert(message)
  } finally {
    isLoadingLocation.value = false
  }
}
```

4.3 useTheme.ts - Управление темой приложения

Этот composable демонстрирует более сложную архитектуру темизации с поддержкой автоматического режима и системных предпочтений пользователя.

— auto - автоматически следует системной теме

— light - принудительно светлая тема

— dark - принудительно темная тема

```
type Theme = 'auto' | 'light' | 'dark'

const currentTheme = ref<Theme>('auto')

export function useTheme() {
 const themes: Theme[] = ['auto', 'light', 'dark']
 const getStoredTheme = (): Theme | null => {
   const stored = localStorage.getItem('theme') as Theme
   return themes.includes(stored) ? stored : null
 }
```

Computed свойства для реактивности

```
 const getEffectiveTheme = computed(() => {
   if (currentTheme.value === 'auto') {
```

```
    return window.matchMedia('(prefers-color-scheme: dark)').matches ? 'dark' :
'light'

  }

  return currentTheme.value

})
```

Computed свойство автоматически пересчитывает активную тему при изменении системных настроек или выбора пользователя.

Интеграция с системными предпочтениями

```
const initTheme = () => {

  const stored = getStoredTheme()

  currentTheme.value = stored || 'auto'

  applyTheme(currentTheme.value)

  const mediaQuery = window.matchMedia('(prefers-color-scheme: dark)')

  mediaQuery.addEventListener('change', () => {

    if (currentTheme.value === 'auto') {

      applyTheme('auto')

    }

  })

}
```

MediaQuery API отслеживает изменения системной темы и автоматически обновляет интерфейс в режиме "auto".

Управление meta-тегами

```
const updateMetaThemeColor = (theme: Theme) => {

  const effectiveTheme = theme === 'auto' ? getEffectiveTheme.value : theme

  let metaThemeColor = document.querySelector('meta[name="theme-color"]')

  if (!metaThemeColor) {

    metaThemeColor = document.createElement('meta')

    metaThemeColor.setAttribute('name', 'theme-color')

    document.head.appendChild(metaThemeColor)

  }

  metaThemeColor.setAttribute('content', effectiveTheme === 'dark' ? '#1a1a1a' :
'#ffffff')
```

```
}
```

Автоматическое обновление meta[name="theme-color"] для правильного отображения в браузерах и PWA.

Циклическое переключение тем

Пользователь может переключаться между темами одной кнопкой: auto → light → dark → auto. Иконка показывает следующий режим.

```
const cycleTheme = () => {
  const currentIndex = themes.indexOf(currentTheme.value)
  const nextIndex = (currentIndex + 1) % themes.length
  setTheme(themes[nextIndex])
}
```

Кастомные события

Генерация кастомных событий позволяет другим частям приложения реагировать на смену темы.

```
const setTheme = (theme: Theme) => {
  if (!themes.includes(theme)) return
  currentTheme.value = theme
  storeTheme(theme)
  applyTheme(theme)
  window.dispatchEvent(new CustomEvent('themeChanged', {
    detail: { theme: theme, effectiveTheme: getEffectiveTheme.value }
  }))
}
```

Валидация и хранение данных

Сохранение выбора пользователя с валидацией - если в localStorage записано невалидное значение, используется default.

```
const getStoredTheme = (): Theme | null => {
  const stored = localStorage.getItem('theme') as Theme
  return themes.includes(stored) ? stored : null
}
```

Применение темы к DOM

Прямое манипулирование DOM для установки data-атрибутов, на которых основаны CSS переменные тем.

```
const applyTheme = (theme: Theme) => {

  const root = document.documentElement

  root.removeAttribute('data-theme')

  if (theme !== 'auto') {

    root.setAttribute('data-theme', theme)

  }

  updateMetaThemeColor(theme)

}
```

5. Фильтрация

Реализована на бэкенде через json-сервер

Преимущества серверной фильтрации:

— Масштабируемость: Обработка больших датасетов на сервере

— Производительность сети: Передача только отфильтрованных результатов

— Кеширование: Возможность кеширования результатов на уровне сервера

— Консистентность: Единые правила фильтрации для всех клиентов

```
      if (filters.location) {

        filteredData = filteredData.filter(property =>

          property.location.toLowerCase().includes(filters.location.toLowerCase())

        )

      }

      if (filters.type) {

        filteredData = filteredData.filter(property => property.type ===
filters.type)

      }

      if (filters.types && filters.types.length > 0) {
```

```
        filteredData = filteredData.filter(property =>
          filters.types.includes(property.type)
        )
      }
      if (filters.minPrice !== null && filters.minPrice !== undefined) {
        filteredData = filteredData.filter(property => property.price >=
filters.minPrice)
      }
      if (filters.maxPrice !== null && filters.maxPrice !== undefined) {
        filteredData = filteredData.filter(property => property.price <=
filters.maxPrice)
      }
      if (filters.minRating !== null && filters.minRating !== undefined) {
        filteredData = filteredData.filter(property => property.rating >=
filters.minRating)
      }
      if (filters.amenities && filters.amenities.length > 0) {
        filteredData = filteredData.filter(property =>
          filters.amenities.some((amenity: string) =>
property.amenities.includes(amenity))
        )
      }
      if (filters.adults && filters.adults > 0) {
        filteredData = filteredData.filter(property => property.maxGuests >=
filters.adults)
      }
      if (filters.children !== null && filters.children !== undefined) {
        const totalGuests = (filters.adults || 1) + filters.children
        filteredData = filteredData.filter(property => property.maxGuests >=
totalGuests)
      }
      return { success: true, data: filteredData }
    } catch (error) {
      console.error('Error fetching properties:', error)
      return { success: false, message: 'Failed to fetch properties' }
```

JSON Server Query API:

GET /properties?location_like=Belgrade&type=apartment&price_gte=30&price_lte=200&rating_gte=4.0

JSON Server автоматически поддерживает различные операторы:

_like - частичное совпадение

_gte / _lte - больше/меньше или равно

_ne - не равно

q - полнотекстовый поиск

5.2 Реактивная смена фильтров

Deep watching отслеживает изменения во вложенных объектах и массивах, автоматически запуская перефильтрацию.

```
const filters = ref({
 types: [] as string[],
 minPrice: null as number | null,
 maxPrice: null as number | null,
 minRating: null as number | null,
 amenities: [] as string[],
 adults: 1,
 children: 0,
 checkIn: '',
 checkOut: ''
})
```

Изменение фильтров происходит автоматически при применении

```
watch(
 () => route.query,
 (newQuery) => {
   searchQuery.value.location = (newQuery.location as string) || ''
   searchQuery.value.checkIn = (newQuery.checkIn as string) || ''
   searchQuery.value.checkOut = (newQuery.checkOut as string) || ''
```

```
    searchQuery.value.guests = (newQuery.guests as string) || '1'
    filterProperties()
  }
)
watch(
  [filters, () => searchQuery.value.location],
  () => {
    filterProperties()
  },
  { deep: true }
)
```

6. Компонентная архитектура

6.1 Иерархия компонентов

App.vue

├── AppNavbar.vue

├── SearchView.vue

│ ├── PropertyList.vue

│ │ └── PropertyCard.vue

├── PropertyView.vue

├── LoginView.vue

└── ProfileView.vue

6.2 PropertyCard.vue - Карточка недвижимости

Особенности компонента:

— Props типизация: Строгие TypeScript интерфейсы

— Event модификаторы: @click.stop предотвращает всплытие события

— Computed логика: Динамическое определение текста рейтинга

— Lazy loading: Оптимизация загрузки изображений

— Accessibility: Alt-теги и ARIA атрибуты

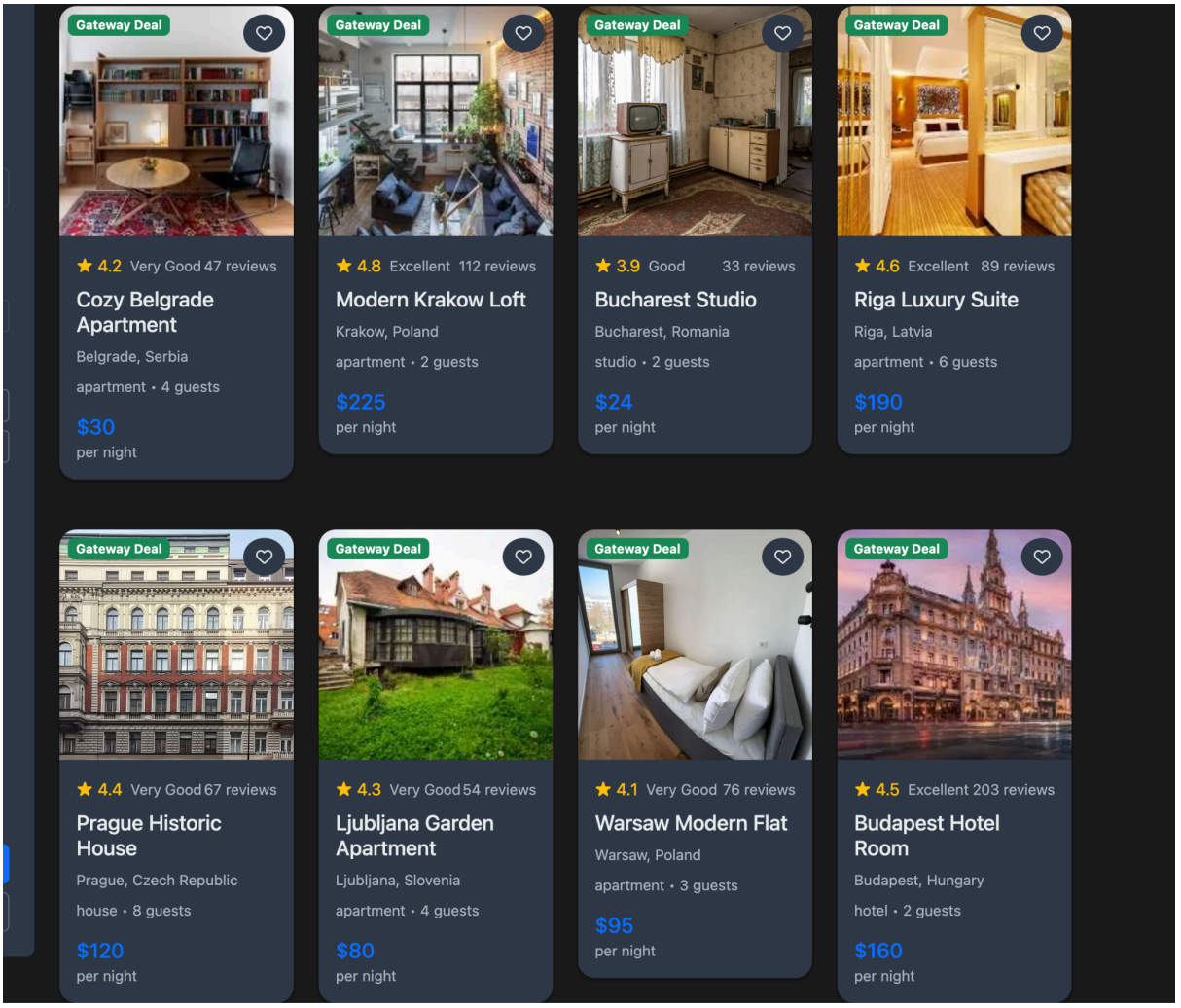

Рисунок 1 — Скриншот карточек отелей

6.3 PropertyList.vue - Список недвижимости

Принципы компонентной архитектуры:

— Single Responsibility: Каждый компонент отвечает за одну задачу

— Prop Drilling: Данные передаются через props

— Event Bubbling: События всплывают вверх по иерархии

— Composition: Сложные UI собираются из простых компонентов.

6.4 AppNavbar.vue - Навигационная панель

На примере него можно демонстрировать условную отрисовку, так как реализованы разные:

— Бизнес-логика: Разные пункты меню для авторизованных/неавторизованных пользователей

— UX: Пользователь видит только релевантные ему элементы

— Безопасность: Скрываются защищенные роуты (messages, profile) для неавторизованных

Основная условная отрисовка – авторизация:

```
<template v-if="isAuthenticated">
  <li class="nav-item">
    <RouterLink class="nav-link" to="/messages">Messages</RouterLink>
  </li>
  <li class="nav-item">
    <RouterLink class="nav-link" to="/profile">Profile</RouterLink>
  </li>
  <li class="nav-item">
    <button class="nav-link btn btn-link" @click="logout">Logout</button>
  </li>
</template>
<template v-else>
  <li class="nav-item">
    <RouterLink class="nav-link" to="/login">Login</RouterLink>
  </li>
  <li class="nav-item">
    <RouterLink class="nav-link" to="/register">Register</RouterLink>
  </li>
</template>
```

Динамический класс – условное применение стилей

```
div class="collapse navbar-collapse" :class="{ show: isNavbarOpen }"
id="navbarNav">
```

```
<script setup lang="ts">
import { useAuthStore } from '@/stores/auth'
import { useTheme } from '@/composables/useTheme'
const authStore = useAuthStore()
const { cycleTheme, nextThemeIcon, themeTitle } = useTheme()
const { isAuthenticated, user } = storeToRefs(authStore)
const logout = () => {
  authStore.logout()
}
</script>
```

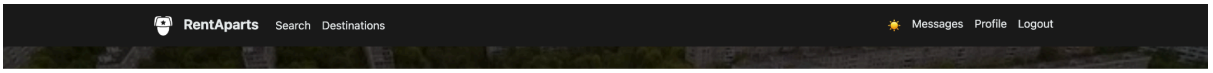

Рисунок 2 – Скриншот навигационной панели