ДИСЦИПЛИНА	Фронтенд и бэкенд разработка
ИНСТИТУТ	ИПТИП
КАФЕДРА	Индустриального программирования
ВИД УЧЕБНОГО МАТЕРИАЛА	Методические указания к практическим занятиям
ПРЕПОДАВАТЕЛЬ	Астафьев Рустам Уралович
СЕМЕСТР	1 семестр, 2025/2026 уч. год

Ссылка на материал:
https://github.com/astafiev-rustam/frontend-and-backend-development/tree/practice-1-27

Практическое занятие 27: Vue 3 + Vite: основы реактивности, шаблоны и директивы

Введение

Сегодня мы начинаем изучение Vue.js - современного фреймворка для создания пользовательских интерфейсов. Vue сочетает в себе простоту изучения и мощные возможности. Мы познакомимся с основными концепциями: реактивностью, директивами и компонентами. Основная информация по фреймворку содержится в материалах лекции, а также в официальной документации:

https://ru.vuejs.org/

Подготовка проекта

Первым делом создадим новый проект Vue. Откройте терминал и выполните команду:

```
npm create vue@latest vue-practice-27
```

Когда система спросит о дополнительных возможностях, просто нажимайте Enter для выбора значений по умолчанию. После создания проекта перейдите в его папку и установите зависимости:

```
cd vue-practice-27
npm install
```

Теперь можно запустить сервер разработки:

```
npm run dev
```

Проект будет доступен по адресу http://localhost:5173

Подключение компонентов в Vue

Сначала обновим основной файл `src/App.vue`:

```
<template>
  <div id="app">
    <header class="app-header">
      <h1>🎯 Vue 3 Практика - Основы</h1>
      <p>Изучаем реактивность, директивы и компоненты</p>
    </header>

    <!-- Навигация между примерами -->
    <nav class="navigation">
      <button
        @click="currentDemo = 'reactive'"
        :class="{ active: currentDemo === 'reactive' }"
        class="nav-button"
      >
        Пример 1: Реактивность
      </button>
      <button
        @click="currentDemo = 'conditional'"
        :class="{ active: currentDemo === 'conditional' }"
        class="nav-button"
      >
        Пример 2: Списки и условия
      </button>
      <button
        @click="currentDemo = 'events'"
        :class="{ active: currentDemo === 'events' }"
        class="nav-button"
      >
        Пример 3: События
      </button>
    </nav>

    <!-- Отображаем выбранный компонент -->
    <main class="main-content">
      <!-- Компонент ReactiveDemo -->
      <ReactiveDemo v-if="currentDemo === 'reactive'" />

      <!-- Компонент ConditionalListDemo -->
      <ConditionalListDemo v-else-if="currentDemo === 'conditional'" />

      <!-- Компонент EventComputedDemo -->
      <EventComputedDemo v-else-if="currentDemo === 'events'" />

      <!-- Сообщение если ничего не выбрано -->
      <div v-else class="welcome-message">
        <h2>Добро пожаловать!</h2>
```

```
        <p>Выберите пример для изучения из навигации выше.</p>
      </div>
    </main>

    <footer class="app-footer">
      <p>Vue 3 + Vite • Практика 27</p>
    </footer>
  </div>
</template>

<script>
// Импортируем наши компоненты, пока файлов нет - должно быть закомментировано
/*import ReactiveDemo from './components/ReactiveDemo.vue'
import ConditionalListDemo from './components/ConditionalListDemo.vue'
import EventComputedDemo from './components/EventComputedDemo.vue'
*/

import { ref } from 'vue'

export default {
  name: 'App',

  // Регистрируем компоненты чтобы использовать их в шаблоне
  components: {
//    ReactiveDemo,
//    ConditionalListDemo,
//    EventComputedDemo
  },

  setup() {
    // Текущий активный демо-компонент
    const currentDemo = ref('reactive')

    return {
      currentDemo
    }
  }
}
</script>

<style>
/* Глобальные стили */
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  background-color: #f5f5f5;
  color: #333;
  line-height: 1.6;
}
```

```
#app {
  min-height: 100vh;
  display: flex;
  flex-direction: column;
}

/* Стили шапки */
.app-header {
  background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
  color: white;
  padding: 2rem;
  text-align: center;
}

.app-header h1 {
  margin-bottom: 0.5rem;
  font-size: 2.5rem;
}

.app-header p {
  opacity: 0.9;
  font-size: 1.1rem;
}

/* Навигация */
.navigation {
  display: flex;
  justify-content: center;
  gap: 1rem;
  padding: 1.5rem;
  background-color: white;
  box-shadow: 0 2px 10px rgba(0,0,0,0.1);
  flex-wrap: wrap;
}

.nav-button {
  padding: 0.75rem 1.5rem;
  border: 2px solid #667eea;
  background-color: transparent;
  color: #667eea;
  border-radius: 25px;
  cursor: pointer;
  font-size: 1rem;
  transition: all 0.3s ease;
}

.nav-button:hover {
  background-color: #667eea;
  color: white;
  transform: translateY(-2px);
}

.nav-button.active {
```

```
    background-color: #667eea;
    color: white;
  }

  /* Основное содержимое */
  .main-content {
    flex: 1;
    padding: 2rem;
    max-width: 1200px;
    margin: 0 auto;
    width: 100%;
  }

  .welcome-message {
    text-align: center;
    padding: 4rem 2rem;
    color: #666;
  }

  .welcome-message h2 {
    margin-bottom: 1rem;
    color: #333;
  }

  /* Подвал */
  .app-footer {
    background-color: #333;
    color: white;
    text-align: center;
    padding: 1rem;
    margin-top: auto;
  }
  </style>
```

Пока код с подключением компонент закомментирован, но по мере добавления открываем комментарии.

Введение в Vue.js: архитектура, синтаксис и сравнение с React

Философия и архитектура Vue.js

Vue.js представляет собой прогрессивный JavaScript-фреймворк для построения пользовательских интерфейсов. Термин "прогрессивный" означает, что Vue может использоваться как для создания отдельных интерактивных компонентов на странице, так и для построения полноценных Single Page Applications (SPA). В отличие от монолитных фреймворков, Vue позволяет разработчику постепенно интегрировать функциональность по мере необходимости.

Сравнение Vue и React: архитектурные различия

Подход к компонентам

React использует JSX (JavaScript XML) - синтаксическое расширение JavaScript, позволяющее писать HTML-подобный код внутри JavaScript:

```
// React компонент
function Greeting({ name }) {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h1>Привет, {name}!</h1>
      <button onClick={() => setCount(count + 1)}>
        Clicked {count} times
      </button>
    </div>
  );
}
```

Vue использует однофайловые компоненты (Single File Components, SFC) с разделением на три секции: `<template>`, `<script>` и `<style>`. Это обеспечивает более строгое разделение ответственности:

```
<!-- Vue компонент -->
<template>
  <div>
    <h1>Привет, {{ name }}!</h1>
    <button @click="count++">
      Clicked {{ count }} times
    </button>
  </div>
</template>

<script>
import { ref } from 'vue'

export default {
  props: ['name'],
  setup() {
    const count = ref(0)
    return { count }
  }
}
</script>

<style scoped>
/* стили компонента */
</style>
```

Система реактивности

React использует явное управление состоянием через хуки (`useState`, `useReducer`). При изменении состояния компонент перерендеривается полностью:

```
const [user, setUser] = useState({ name: '', age: 0 });

// изменение состояния требует создания нового объекта
setUser({ ...user, name: 'Иван' });
```

Vue предоставляет встроенную реактивную систему, основанную на прокси-объектах. Изменения отслеживаются автоматически:

```
const user = ref({ name: '', age: 0 });

// прямое изменение свойства автоматически триггерит обновление
user.value.name = 'Иван';
```

Двустороннее связывание данных

В **React** двустороннее связывание реализуется вручную через комбинацию `value` и `onChange`:

```
<input
  value={inputValue}
  onChange={(e) => setInputValue(e.target.value)}
/>
```

В **Vue** для этого существует директива `v-model`, которая автоматически синхронизирует данные:

```
<input v-model="inputValue" />
```

Синтаксис шаблонов Vue

Интерполяция данных

Vue использует синтаксис "усов" (mustache syntax) для вывода данных в шаблоне:

```
<template>
  <p>Сообщение: {{ message }}</p>
  <p>Вычисление: {{ count * 2 + 1 }}</p>
  <p>Метод: {{ formatDate(date) }}</p>
</template>
```

Директивы

Директивы - это специальные атрибуты с префиксом `v-`, которые применяют реактивное поведение к DOM:

- `v-bind` (сокращение `:`) - связывает атрибут с данными

```
<img v-bind:src="imagePath" />
<img :src="imagePath" />  <!-- сокращенная форма -->
```

- `v-on` (сокращение `@`) - обрабатывает события

```
<button v-on:click="handleClick">Click</button>
<button @click="handleClick">Click</button>  <!-- сокращенная форма -->
```

- `v-model` - создает двустороннее связывание

```
<input v-model="text" />
```

- `v-if`/`v-else-if`/`v-else` - условный рендеринг

```
<div v-if="isVisible">Виден</div>
<div v-else>Скрыт</div>
```

- `v-for` - рендеринг списков

```
<li v-for="item in items" :key="item.id">{{ item.name }}</li>
```

Composition API vs Options API

Vue 3 предлагает два API для написания компонентов:

Options API (классический подход)

```
export default {
  name: 'CounterComponent',  // имя компонента (опционально, для отладки)

  props: {                    // входящие параметры от родительского компонента
    initialValue: {
      type: Number,
      default: 0
```

```
    }
  },

  data() {                    // реактивное состояние компонента
    return {
      count: 0
    }
  },

  computed: {                 // вычисляемые свойства
    doubleCount() {
      return this.count * 2
    }
  },

  methods: {                  // методы компонента
    increment() {
      this.count++
    }
  },

  watch: {                    // наблюдатели за изменениями данных
    count(newValue, oldValue) {
      console.log(`Count changed from ${oldValue} to ${newValue}`)
    }
  },

  created() {                 // lifecycle hook: компонент создан
    console.log('Component created')
  },

  mounted() {                 // lifecycle hook: компонент смонтирован в DOM
    console.log('Component mounted')
  }
}
```

Структура Options API:

- `name` - имя компонента. Используется в DevTools для отладки и при рекурсивных компонентах. Необязательное свойство.
- `props` - объект описания входящих параметров компонента с валидацией типов и значениями по умолчанию.
- `data()` - функция, возвращающая объект с реактивными данными. Должна быть функцией, чтобы каждый экземпляр компонента имел свою копию данных.
- `computed` - объект вычисляемых свойств, которые кэшируются и обновляются только при изменении зависимостей.
- `methods` - объект методов компонента. Доступ к данным через `this.propertyName`.
- `watch` - объект наблюдателей за изменениями конкретных свойств.
- **Lifecycle hooks** (`created`, `mounted`, `updated`, `unmounted` и др.) - специальные методы, вызываемые на разных этапах жизни компонента.

Важно: в Options API контекст `this` автоматически привязан к экземпляру компонента, поэтому все свойства доступны через `this.count`, `this.increment()` и т.д.

Composition API (современный подход, аналог React Hooks)

```
import { ref, computed, watch, onMounted, onUnmounted } from 'vue'

export default {
  name: 'CounterComponent',  // имя компонента (опционально)

  props: {                    // props определяются так же, как в Options API
    initialValue: {
      type: Number,
      default: 0
    }
  },

  setup(props, context) {     // главная функция, где определяется вся логика
    // реактивное состояние
    const count = ref(0)

    // вычисляемые свойства
    const doubleCount = computed(() => count.value * 2)

    // методы
    const increment = () => {
      count.value++
    }

    // наблюдатели
    watch(count, (newValue, oldValue) => {
      console.log(`Count changed from ${oldValue} to ${newValue}`)
    })

    // lifecycle hooks
    onMounted(() => {
      console.log('Component mounted')
    })

    onUnmounted(() => {
      console.log('Component unmounted')
    })

    // возвращаем всё, что должно быть доступно в шаблоне
    return {
      count,
      doubleCount,
      increment
    }
  }
}
```

Структура Composition API:

- `name` - имя компонента. Определяется на том же уровне, что и в Options API.
- `props` - определяются идентично Options API, вне функции `setup()`.
- `setup(props, context)` - центральная функция, где происходит вся логика компонента:
  - **Параметр** `props` - объект с входящими параметрами (реактивный).
  - **Параметр** `context` - объект с `attrs`, `slots`, `emit` для работы с контекстом компонента.
  - Внутри `setup()` определяются все реактивные переменные, computed, методы, watchers и lifecycle hooks.
  - `return` - объект со всеми переменными и функциями, которые должны быть доступны в шаблоне.

Важно: в Composition API нет контекста `this`. Все переменные и функции определяются как обычные JavaScript константы и функции.

Сравнение организации логики

Options API группирует код по типу опций:

```
data()     → все состояние в одном месте
methods    → все методы в одном месте
computed   → все вычисляемые свойства в одном месте
```

Composition API группирует код по фичам:

```
setup() {
  // Фича 1: Счетчик
  const count = ref(0)
  const increment = () => count.value++
  const doubleCount = computed(() => count.value * 2)

  // Фича 2: Таймер
  const time = ref(0)
  const startTimer = () => { /* ... */ }

  return { count, increment, doubleCount, time, startTimer }
}
```

Это позволяет легко выносить логику в отдельные composable функции для переиспользования.

Где указывать имя компонента

В обоих API имя компонента указывается в свойстве `name` объекта, экспортируемого из `<script>`:

```
<script>
export default {
```

```
  name: 'MyComponent',  // ← здесь указывается имя компонента
  // остальная конфигурация...
}
</script>
```

Для чего нужно свойство `name`:

1. **Отладка** - отображается в Vue DevTools
2. **Рекурсивные компоненты** - компонент может ссылаться сам на себя по имени
3. **Keep-alive** - для include/exclude списков
4. **Предупреждения** - Vue использует имя в сообщениях об ошибках

Примечание: если не указать `name`, Vue автоматически выведет имя из имени файла компонента.

В данной практике мы используем **Composition API**, так как он обеспечивает лучшую организацию кода и повторное использование логики через композируемые функции (composables).

Отличия от JSX

Vue **не использует JSX** в стандартной разработке. Вместо этого применяется декларативный шаблонный синтаксис, основанный на HTML. Это дает несколько преимуществ:

1. **Более низкий порог входа** - шаблоны Vue ближе к стандартному HTML
2. **Лучшая оптимизация** - Vue компилятор может провести статический анализ шаблона
3. **Интеграция с визуальными редакторами** - легче создавать GUI-инструменты

Однако при необходимости Vue поддерживает render-функции и JSX через плагин:

```
// возможно, но не рекомендуется в стандартной разработке
export default {
  render() {
    return <div>Hello {this.name}</div>
  }
}
```

Жизненный цикл компонента

Как и в React, компоненты Vue имеют жизненный цикл. В Composition API используются lifecycle hooks:

```
import { onMounted, onUpdated, onUnmounted } from 'vue'

export default {
  setup() {
    onMounted(() => {
      console.log('компонент смонтирован')
    })

    onUpdated(() => {
      console.log('компонент обновлен')
```

```
    })

    onUnmounted(() => {
      console.log('компонент размонтирован')
    })
  }
}
```

Сравнение с React:

- `onMounted` ≈ `useEffect(() => {}, [])`
- `onUpdated` ≈ `useEffect(() => {})`
- `onUnmounted` ≈ `useEffect(() => { return () => {} }, [])`

Пример 1: Реактивность и двустороннее связывание

Давайте создадим простой компонент, который демонстрирует основы реактивности Vue. Реактивность - это способность Vue автоматически обновлять интерфейс при изменении данных.

Создайте файл `src/components/ReactiveDemo.vue`:

```
<template>
  <div class="demo-container">
    <h2>Пример 1: Реактивность и v-model</h2>

    <p>Директива v-model создает двустороннее связывание между элементом формы и данными компонента.</p>

    <!-- простое текстовое поле с v-model -->
    <div class="input-group">
      <label>Введите ваше имя:</label>
      <input
        v-model="userName"
        placeholder="Например: Иван"
        class="text-input"
      >
      <!-- интерполяция реактивной переменной в шаблоне -->
      <p>Привет, <strong>{{ userName }}</strong>!</p>
    </div>

    <!-- Textarea с v-model -->
    <div class="input-group">
      <label>Опишите ваши интересы:</label>
      <textarea
        v-model="userBio"
        placeholder="Расскажите о себе..."
        class="text-area"
        rows="3"
      ></textarea>
      <p>Длина текста: {{ userBio.length }} символов</p>
```

```
    </div>

    <!-- Select с v-model -->
    <div class="input-group">
      <label>Выберите технологию:</label>
      <select v-model="selectedTech" class="select-input">
        <option value="vue">Vue.js</option>
        <option value="react">React</option>
        <option value="angular">Angular</option>
      </select>
      <p>Вы выбрали: {{ getTechName(selectedTech) }}</p>
    </div>

    <!-- Чекбокс с v-model -->
    <div class="input-group">
      <label>
        <input type="checkbox" v-model="isSubscribed">
        Получать уведомления
      </label>
      <p v-if="isSubscribed">Вы подписаны на уведомления ✓</p>
      <p v-else>Вы не подписаны на уведомления</p>
    </div>

    <!-- Группа радио-кнопок -->
    <div class="input-group">
      <label>Уровень опыта:</label>
      <div>
        <label>
          <input type="radio" v-model="experienceLevel" value="beginner">
          Начинающий
        </label>
        <label>
          <input type="radio" v-model="experienceLevel" value="intermediate">
          Средний
        </label>
        <label>
          <input type="radio" v-model="experienceLevel" value="advanced">
          Продвинутый
        </label>
      </div>
      <p>Ваш уровень: {{ getExperienceText(experienceLevel) }}</p>
    </div>

    <!-- Отладочная информация -->
    <div class="debug-info">
      <h3>Текущее состояние данных:</h3>
      <pre>{{ JSON.stringify({
        userName,
        userBio,
        selectedTech,
        isSubscribed,
        experienceLevel
      }, null, 2) }}</pre>
    </div>
```

```
  </div>
</template>

<script>
import { ref } from 'vue'

export default {
  name: 'ReactiveDemo',

  setup() {
    // ref создает реактивную переменную
    // Когда значение меняется, Vue автоматически обновляет шаблон
    const userName = ref('')
    const userBio = ref('')
    const selectedTech = ref('vue')
    const isSubscribed = ref(false)
    const experienceLevel = ref('beginner')

    // Методы для форматирования данных
    const getTechName = (tech) => {
      const techMap = {
        'vue': 'Vue.js',
        'react': 'React',
        'angular': 'Angular'
      }
      return techMap[tech] || tech
    }

    const getExperienceText = (level) => {
      const levelMap = {
        'beginner': 'Начинающий разработчик',
        'intermediate': 'Разработчик со средним опытом',
        'advanced': 'Опытный разработчик'
      }
      return levelMap[level] || level
    }

    // возвращаем все переменные и методы, чтобы они были доступны в шаблоне
    return {
      userName,
      userBio,
      selectedTech,
      isSubscribed,
      experienceLevel,
      getTechName,
      getExperienceText
    }
  }
}
</script>

<style scoped>
.demo-container {
  max-width: 600px;
```

```
  margin: 20px auto;
  padding: 20px;
  border: 1px solid #ddd;
  border-radius: 8px;
  background-color: #f9f9f9;
}

.input-group {
  margin-bottom: 20px;
}

label {
  display: block;
  margin-bottom: 5px;
  font-weight: bold;
}

.text-input, .text-area, .select-input {
  width: 100%;
  padding: 8px;
  border: 1px solid #ccc;
  border-radius: 4px;
  font-size: 14px;
}

.debug-info {
  margin-top: 30px;
  padding: 15px;
  background-color: #e9ecef;
  border-radius: 4px;
}

pre {
  background-color: white;
  padding: 10px;
  border-radius: 4px;
  overflow-x: auto;
}
</style>
```

Разбор кода Примера 1: реактивность и директива v-model

Структура однофайлового компонента

Компонент Vue разделен на три логические секции:

1. `<template>` - содержит разметку интерфейса
2. `<script>` - содержит логику компонента
3. `<style scoped>` - содержит стили, изолированные для данного компонента

Реактивная система на основе ref

В функции `setup()` мы создаем реактивные переменные с помощью функции `ref()`:

```
const userName = ref('')
```

Принцип работы ref:

- `ref()` оборачивает примитивное значение в реактивный объект
- Внутри `setup()` для доступа к значению используется `.value`: `userName.value`
- В шаблоне Vue автоматически разворачивает ref, поэтому используется просто `userName`

Сравнение с React:

React:

```
const [userName, setUserName] = useState('');
// изменение: setUserName('Новое имя')
```

Vue:

```
const userName = ref('');
// изменение: userName.value = 'Новое имя'
```

Ключевое различие: в React для изменения состояния необходим setter, в Vue можно изменять значение напрямую благодаря реактивным прокси.

Директива v-model: двустороннее связывание

Директива `v-model` создает двустороннее связывание данных между формой и состоянием:

```
<input v-model="userName" />
```

Под капотом `v-model` является синтаксическим сахаром для:

```
<input
  :value="userName"
  @input="userName = $event.target.value"
/>
```

Особенности работы v-model с разными элементами:

1. **Text input / textarea** - связывается со свойством `value` и событием `input`
2. **Checkbox** - связывается со свойством `checked` и событием `change`

3. **Radio** - связывается со свойством `checked`, все radio с одним `v-model` образуют группу
4. **Select** - связывается со свойством `value` и событием `change`

Интерполяция в шаблоне

Синтаксис `{{ }}` используется для вывода данных в текстовое содержимое элемента:

```
<p>Привет, <strong>{{ userName }}</strong>!</p>
```

Внутри интерполяции можно использовать:

- Переменные: `{{ userName }}`
- Выражения: `{{ userBio.length }}`
- Вызовы методов: `{{ getTechName(selectedTech) }}`

Важно: интерполяция работает только для текстового содержимого. Для атрибутов используется `v-bind` или `:`.

Условный рендеринг с v-if и v-else

```
<p v-if="isSubscribed">Вы подписаны на уведомления ✓</p>
<p v-else>Вы не подписаны на уведомления</p>
```

`v-if` полностью удаляет или добавляет элемент в DOM в зависимости от условия. Это отличается от CSS-свойства `display: none`, которое просто скрывает элемент.

Сравнение с React:

React:

```
{isSubscribed ? (
  <p>Вы подписаны</p>
) : (
  <p>Вы не подписаны</p>
)}
```

Vue:

```
<p v-if="isSubscribed">Вы подписаны</p>
<p v-else>Вы не подписаны</p>
```

Vue предоставляет более декларативный синтаксис, приближенный к естественному языку.

Функция setup() и возврат значений

Функция `setup()` является точкой входа для Composition API:

```
setup() {
  const userName = ref('')

  const getTechName = (tech) => {
    // логика
  }

  return {
    userName,      // делаем доступным в шаблоне
    getTechName    // делаем доступным в шаблоне
  }
}
```

Все, что возвращается из `setup()`, становится доступным в шаблоне. Переменные и методы, не включенные в возвращаемый объект, остаются приватными для компонента.

Реактивность примитивов vs объектов

Для примитивных значений (string, number, boolean) используется `ref()`:

```
const count = ref(0)
count.value++  // изменение через .value
```

Для объектов можно использовать `reactive()`:

```
const user = reactive({ name: '', age: 0 })
user.name = 'Иван'  // изменение напрямую, без .value
```

В данном примере мы используем `ref()` для всех типов данных, что обеспечивает единообразие кода.

Отладочный блок с JSON.stringify

```
<pre>{{ JSON.stringify({
  userName,
  userBio,
  selectedTech,
  isSubscribed,
  experienceLevel
}, null, 2) }}</pre>
```

Этот блок демонстрирует текущее состояние всех реактивных переменных в JSON-формате. Параметры `JSON.stringify`:

- Первый параметр - объект для сериализации
- Второй параметр - функция-заменитель (null = не используется)
- Третий параметр - количество пробелов для форматирования (2 = читаемый формат)

Пример 2: Условный рендеринг и списки

Теперь рассмотрим директивы v-if, v-for и v-show для управления отображением элементов.

Создайте файл `src/components/ConditionalListDemo.vue`:

```
<template>
  <div class="demo-container">
    <h2>Пример 2: Условный рендеринг и списки</h2>

    <!-- Форма для добавления технологий -->
    <div class="add-form">
      <input
        v-model="newTechName"
        placeholder="Введите название технологии"
        class="text-input"
        @keyup.enter="addTechnology"
      >
      <button @click="addTechnology" class="add-button">
        Добавить
      </button>
    </div>

    <!-- Переключение видимости с v-show -->
    <div class="controls">
      <button @click="showCompleted = !showCompleted" class="toggle-button">
        {{ showCompleted ? 'Скрыть' : 'Показать' }} завершенные
      </button>
      <button @click="sortBy = sortBy === 'name' ? 'date' : 'name'" class="toggle-
button">
        Сортировать по: {{ sortBy === 'name' ? 'названию' : 'дате' }}
      </button>
    </div>

    <!-- Условный рендеринг с v-if -->
    <div v-if="technologies.length === 0" class="empty-state">
      <p>Список технологий пуст. Добавьте первую технологию!</p>
    </div>

    <!-- Отображение списка с v-for -->
    <div v-else class="tech-list">
      <div
        v-for="tech in sortedTechnologies"
        :key="tech.id"
        class="tech-item"
        :class="{ completed: tech.completed }"
      >
```

```
        <div class="tech-info">
          <h3>{{ tech.name }}</h3>
          <span class="tech-date">Добавлено: {{ formatDate(tech.createdAt) }}
</span>
        </div>

        <div class="tech-actions">
          <!-- v-show для переключения видимости -->
          <button
            v-show="!tech.completed"
            @click="completeTechnology(tech.id)"
            class="complete-button"
          >
            Завершить
          </button>

          <button
            @click="removeTechnology(tech.id)"
            class="remove-button"
          >
            Удалить
          </button>
        </div>
      </div>
    </div>

    <!-- Статистика -->
    <div class="stats">
      <h3>Статистика:</h3>
      <p>Всего технологий: {{ technologies.length }}</p>
      <p>Активных: {{ activeCount }}</p>
      <p>Завершенных: {{ completedCount }}</p>

      <!-- Условный рендеринг с v-if/v-else -->
      <div v-if="completedCount > 0" class="progress-section">
        <p>Прогресс: {{ progressPercentage }}%</p>
        <div class="progress-bar">
          <div
            class="progress-fill"
            :style="{ width: progressPercentage + '%' }"
          ></div>
        </div>
      </div>
      <div v-else>
        <p>Начните изучать технологии чтобы увидеть прогресс!</p>
      </div>
    </div>
  </div>
</template>

<script>
import { ref, computed } from 'vue'

export default {
```

```
  name: 'ConditionalListDemo',

  setup() {
    // реактивные данные
    const newTechName = ref('')
    const technologies = ref([])  // массив объектов технологий
    const showCompleted = ref(true)
    const sortBy = ref('date')

    // вычисляемые свойства автоматически пересчитываются при изменении
зависимостей
    const activeCount = computed(() =>
      technologies.value.filter(tech => !tech.completed).length
    )

    const completedCount = computed(() =>
      technologies.value.filter(tech => tech.completed).length
    )

    const progressPercentage = computed(() => {
      if (technologies.value.length === 0) return 0
      return Math.round((completedCount.value / technologies.value.length) * 100)
    })

    const sortedTechnologies = computed(() => {
      const techsToShow = showCompleted.value
        ? technologies.value
        : technologies.value.filter(tech => !tech.completed)

      return [...techsToShow].sort((a, b) => {
        if (sortBy.value === 'name') {
          return a.name.localeCompare(b.name)
        } else {
          return new Date(b.createdAt) - new Date(a.createdAt)
        }
      })
    })

    // методы для работы с массивом
    const addTechnology = () => {
      if (!newTechName.value.trim()) return

      // добавляем новый объект в массив
      technologies.value.push({
        id: Date.now(),  // уникальный идентификатор
        name: newTechName.value.trim(),
        completed: false,
        createdAt: new Date().toISOString()
      })

      // очищаем поле ввода
      newTechName.value = ''
    }
```

```
    const removeTechnology = (id) => {
      // фильтруем массив, удаляя элемент с указанным id
      technologies.value = technologies.value.filter(tech => tech.id !== id)
    }

    const completeTechnology = (id) => {
      // находим технологию по id и меняем её состояние
      const tech = technologies.value.find(tech => tech.id === id)
      if (tech) {
        tech.completed = true
      }
    }

    const formatDate = (dateString) => {
      return new Date(dateString).toLocaleDateString('ru-RU')
    }

    return {
      newTechName,
      technologies,
      showCompleted,
      sortBy,
      activeCount,
      completedCount,
      progressPercentage,
      sortedTechnologies,
      addTechnology,
      removeTechnology,
      completeTechnology,
      formatDate
    }
  }
}
</script>

<style scoped>
.demo-container {
  max-width: 600px;
  margin: 20px auto;
  padding: 20px;
}

.add-form {
  display: flex;
  gap: 10px;
  margin-bottom: 20px;
}

.text-input {
  flex: 1;
  padding: 8px;
  border: 1px solid #ccc;
  border-radius: 4px;
}
```

```
.add-button, .toggle-button {
  padding: 8px 16px;
  background-color: #007bff;
  color: white;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

.controls {
  display: flex;
  gap: 10px;
  margin-bottom: 20px;
}

.tech-list {
  display: flex;
  flex-direction: column;
  gap: 10px;
}

.tech-item {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 15px;
  border: 1px solid #ddd;
  border-radius: 8px;
  background-color: white;
}

.tech-item.completed {
  background-color: #d4edda;
  border-color: #c3e6cb;
}

.tech-info h3 {
  margin: 0 0 5px 0;
}

.tech-date {
  font-size: 12px;
  color: #666;
}

.tech-actions {
  display: flex;
  gap: 5px;
}

.complete-button {
  padding: 5px 10px;
  background-color: #28a745;
```

```
    color: white;
    border: none;
    border-radius: 4px;
    cursor: pointer;
  }

  .remove-button {
    padding: 5px 10px;
    background-color: #dc3545;
    color: white;
    border: none;
    border-radius: 4px;
    cursor: pointer;
  }

  .empty-state {
    text-align: center;
    padding: 40px;
    color: #666;
  }

  .stats {
    margin-top: 30px;
    padding: 20px;
    background-color: #f8f9fa;
    border-radius: 8px;
  }

  .progress-bar {
    width: 100%;
    height: 20px;
    background-color: #e9ecef;
    border-radius: 10px;
    overflow: hidden;
  }

  .progress-fill {
    height: 100%;
    background-color: #28a745;
    transition: width 0.3s ease;
  }
  </style>
```

Разбор кода Примера 2: списки, условный рендеринг и вычисляемые свойства

Директива v-for: рендеринг списков

Директива `v-for` используется для отображения массива элементов:

```
  <div
    v-for="tech in sortedTechnologies"
```

```
    :key="tech.id"
    class="tech-item"
  >
    <h3>{{ tech.name }}</h3>
  </div>
```

Ключевые аспекты v-for:

1. **Синтаксис итерации**: `item in items` или `item of items`
2. **Атрибут :key**: обязателен для оптимизации рендеринга, должен быть уникальным
3. **Доступ к индексу**: `v-for="(tech, index) in technologies"`
4. **Итерация объектов**: `v-for="(value, key, index) in object"`

Сравнение с React:

React:

```
{technologies.map((tech) => (
  <div key={tech.id}>
    <h3>{tech.name}</h3>
  </div>
))}
```

Vue:

```
<div v-for="tech in technologies" :key="tech.id">
  <h3>{{ tech.name }}</h3>
</div>
```

React использует метод массива `.map()`, в то время как Vue предоставляет специальную директиву `v-for`, что делает код более декларативным.

Важность атрибута :key

Атрибут `:key` (сокращение от `v-bind:key`) необходим для эффективного отслеживания изменений в списке:

```
<div v-for="tech in technologies" :key="tech.id">
```

Vue использует алгоритм виртуального DOM для минимизации манипуляций с реальным DOM. Без уникального `key` Vue не сможет правильно определить, какие элементы были добавлены, удалены или изменены.

Правила выбора key:

- Должен быть уникальным среди siblings
- Должен быть стабильным (не меняться между ре-рендерами)
- Не использовать индекс массива, если порядок элементов может меняться

Вычисляемые свойства (computed)

Вычисляемые свойства - это реактивные значения, которые автоматически пересчитываются при изменении зависимостей:

```
const activeCount = computed(() =>
  technologies.value.filter(tech => !tech.completed).length
)
```

Преимущества computed над методами:

1. **Кэширование**: результат кэшируется и пересчитывается только при изменении зависимостей
2. **Реактивность**: автоматически обновляется при изменении используемых данных
3. **Производительность**: избегает лишних вычислений

Сравнение computed и методов:

Метод (вызывается при каждом рендере):

```
const getActiveCount = () => {
  return technologies.value.filter(tech => !tech.completed).length
}
```

Computed (вычисляется только при изменении technologies):

```
const activeCount = computed(() =>
  technologies.value.filter(tech => !tech.completed).length
)
```

Сравнение с React:

React (useMemo):

```
const activeCount = useMemo(() =>
  technologies.filter(tech => !tech.completed).length,
  [technologies]
);
```

Vue (computed):

```
const activeCount = computed(() =>
  technologies.value.filter(tech => !tech.completed).length
)
```

В Vue не нужно явно указывать массив зависимостей - они определяются автоматически.

Разница между v-if и v-show

v-if - условное отображение с удалением из DOM:

```
<div v-if="technologies.length === 0">
  Список пуст
</div>
```

v-show - условное отображение через CSS display:

```
<button v-show="!tech.completed">
  Завершить
</button>
```

Когда использовать:

- **v-if**: когда условие редко меняется, элемент сложный, или есть v-else
- **v-show**: когда элемент часто переключается между видимым и скрытым состоянием

Производительность:

- `v-if` имеет более высокую стоимость переключения (создание/удаление DOM-элементов)
- `v-show` имеет более высокую начальную стоимость рендеринга (элемент всегда в DOM)

Динамические классы с :class

```
<div
  class="tech-item"
  :class="{ completed: tech.completed }"
>
```

Синтаксис объекта для `:class` позволяет условно применять классы:

- Ключ - имя класса
- Значение - булево условие

Варианты синтаксиса:

Объект:

```
:class="{ active: isActive, 'text-danger': hasError }"
```

Массив:

```
:class="[activeClass, errorClass]"
```

Комбинированный:

```
:class="['base-class', { active: isActive }]"
```

Сравнение с React:

React (classnames библиотека):

```
<div className={classNames('tech-item', { completed: tech.completed })}>
```

React (условный оператор):

```
<div className={`tech-item ${tech.completed ? 'completed' : ''}`}>
```

Vue:

```
<div class="tech-item" :class="{ completed: tech.completed }">
```

Модификаторы событий

```
<input @keyup.enter="addTechnology">
```

Модификатор `.enter` автоматически проверяет код клавиши, что упрощает обработку событий клавиатуры.

Популярные модификаторы клавиш:

- `.enter`, `.tab`, `.delete`, `.esc`, `.space`
- `.up`, `.down`, `.left`, `.right`
- `.ctrl`, `.alt`, `.shift`, `.meta`

Модификаторы событий мыши:

- `.stop` - вызывает `event.stopPropagation()`
- `.prevent` - вызывает `event.preventDefault()`
- `.capture` - использует режим capture
- `.self` - срабатывает только если target === currentTarget
- `.once` - срабатывает только один раз

Работа с массивами в реактивной системе

При работе с массивами в Vue 3 все методы мутации отслеживаются автоматически:

```
// эти операции реактивны
technologies.value.push(newTech)
technologies.value.pop()
technologies.value.shift()
technologies.value.unshift(newTech)
technologies.value.splice(index, 1)

// для не-мутирующих методов нужно заменить весь массив
technologies.value = technologies.value.filter(tech => tech.id !== id)
technologies.value = technologies.value.map(tech => ({ ...tech, updated: true }))
```

В Vue 2 некоторые операции с массивами не отслеживались, но в Vue 3 (благодаря Proxy) все операции реактивны.

Динамические стили с :style

```
<div :style="{ width: progressPercentage + '%' }">
```

Директива `:style` позволяет динамически применять inline-стили. Синтаксис аналогичен объектам JavaScript, но имена свойств можно писать в camelCase или kebab-case:

```
:style="{ fontSize: '14px', backgroundColor: 'red' }"
:style="{ 'font-size': '14px', 'background-color': 'red' }"
```

Пример 3: Работа с событиями и вычисляемыми свойствами

В третьем примере рассмотрим обработку событий и мощные вычисляемые свойства.

Создайте файл `src/components/EventComputedDemo.vue`:

```
<template>
  <div class="demo-container">
    <h2>Пример 3: События и вычисляемые свойства</h2>
```

```
    <!-- Таймер изучения -->
    <div class="timer-section">
      <h3>Таймер изучения</h3>
      <div class="timer-display">
        {{ formatTime(elapsedTime) }}
      </div>
      <div class="timer-controls">
        <button @click="startTimer" :disabled="isRunning" class="timer-button
start">
          Старт
        </button>
        <button @click="pauseTimer" :disabled="!isRunning" class="timer-button
pause">
          Пауза
        </button>
        <button @click="resetTimer" class="timer-button reset">
          Сброс
        </button>
      </div>
    </div>

    <!-- Список сессий изучения -->
    <div class="sessions-section">
      <h3>Сессии изучения</h3>

      <div v-if="studySessions.length === 0" class="no-sessions">
        <p>Сессий пока нет. Запустите таймер!</p>
      </div>

      <div v-else class="sessions-list">
        <div
          v-for="session in sortedSessions"
          :key="session.id"
          class="session-item"
        >
          <div class="session-info">
            <span class="session-date">{{ formatDate(session.date) }}</span>
            <span class="session-duration">{{ formatTime(session.duration) }}
</span>
          </div>
          <button @click="removeSession(session.id)" class="delete-session">
            ×
          </button>
        </div>
      </div>

      <!-- Статистика сессий -->
      <div class="sessions-stats">
        <p>Всего сессий: {{ totalSessions }}</p>
        <p>Общее время: {{ formatTime(totalStudyTime) }}</p>
        <p>Средняя продолжительность: {{ formatTime(averageSessionTime) }}</p>
      </div>
    </div>
```

```
    <!-- Быстрые действия с модификаторами событий -->
    <div class="quick-actions">
      <h3>Быстрые действия</h3>
      <div class="action-buttons">
        <!-- Модификатор .prevent предотвращает действие по умолчанию -->
        <button @click.prevent="addQuickSession(30)" class="action-button">
          +30 мин
        </button>
        <button @click.prevent="addQuickSession(60)" class="action-button">
          +1 час
        </button>
        <!-- Модификатор .once срабатывает только один раз -->
        <button @click.once="addOneTimeSession" class="action-button special">
          Одноразовая сессия
        </button>
      </div>
    </div>
  </div>
</template>

<script>
import { ref, computed, onUnmounted } from 'vue'

export default {
  name: 'EventComputedDemo',

  setup() {
    // реактивные переменные для таймера
    const elapsedTime = ref(0)
    const isRunning = ref(false)
    let timerInterval = null  // не реактивная переменная для хранения интервала

    // массив сессий изучения
    const studySessions = ref([])

    // вычисляемые свойства для статистики
    const totalSessions = computed(() => studySessions.value.length)

    const totalStudyTime = computed(() =>
      studySessions.value.reduce((total, session) => total + session.duration, 0)
    )

    const averageSessionTime = computed(() => {
      if (totalSessions.value === 0) return 0
      return Math.round(totalStudyTime.value / totalSessions.value)
    })

    const sortedSessions = computed(() =>
      [...studySessions.value].sort((a, b) => new Date(b.date) - new Date(a.date))
    )

    // методы управления таймером
    const startTimer = () => {
```

```
      isRunning.value = true
      // setInterval вызывает функцию каждую секунду
      timerInterval = setInterval(() => {
        elapsedTime.value += 1
      }, 1000)
    }

    const pauseTimer = () => {
      isRunning.value = false
      if (timerInterval) {
        clearInterval(timerInterval)  // останавливаем интервал
        timerInterval = null
      }
    }

    const resetTimer = () => {
      pauseTimer()

      // сохраняем сессию только если прошло больше 30 секунд
      if (elapsedTime.value >= 30) {
        studySessions.value.push({
          id: Date.now(),
          date: new Date().toISOString(),
          duration: elapsedTime.value
        })
      }

      elapsedTime.value = 0
    }

    // методы для работы с сессиями
    const removeSession = (sessionId) => {
      studySessions.value = studySessions.value.filter(session => session.id !==
sessionId)
    }

    const addQuickSession = (minutes) => {
      const duration = minutes * 60  // переводим минуты в секунды
      studySessions.value.push({
        id: Date.now(),
        date: new Date().toISOString(),
        duration: duration
      })
    }

    const addOneTimeSession = () => {
      addQuickSession(45)
      alert('Одноразовая сессия добавлена! Эта кнопка больше не сработает.')
    }

    // вспомогательные методы форматирования
    const formatTime = (seconds) => {
      const mins = Math.floor(seconds / 60)
      const secs = seconds % 60
```

```
      return `${mins.toString().padStart(2, '0')}:${secs.toString().padStart(2,
'0')}`
    }

    const formatDate = (dateString) => {
      return new Date(dateString).toLocaleString('ru-RU')
    }

    // lifecycle hook: выполняется при размонтировании компонента
    onUnmounted(() => {
      if (timerInterval) {
        clearInterval(timerInterval)  // очищаем интервал чтобы избежать утечек
памяти
      }
    })

    return {
      elapsedTime,
      isRunning,
      studySessions,
      totalSessions,
      totalStudyTime,
      averageSessionTime,
      sortedSessions,
      startTimer,
      pauseTimer,
      resetTimer,
      removeSession,
      addQuickSession,
      addOneTimeSession,
      formatTime,
      formatDate
    }
  }
}
</script>

<style scoped>
.demo-container {
  max-width: 600px;
  margin: 20px auto;
  padding: 20px;
}

.timer-section {
  text-align: center;
  margin-bottom: 30px;
  padding: 20px;
  background-color: #f8f9fa;
  border-radius: 10px;
}

.timer-display {
  font-size: 3rem;
```

```
  font-weight: bold;
  margin: 20px 0;
  color: #007bff;
}

.timer-controls {
  display: flex;
  gap: 10px;
  justify-content: center;
}

.timer-button {
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  font-weight: bold;
}

.timer-button:disabled {
  opacity: 0.5;
  cursor: not-allowed;
}

.timer-button.start {
  background-color: #28a745;
  color: white;
}

.timer-button.pause {
  background-color: #ffc107;
  color: black;
}

.timer-button.reset {
  background-color: #dc3545;
  color: white;
}

.sessions-section {
  margin-bottom: 30px;
}

.sessions-list {
  display: flex;
  flex-direction: column;
  gap: 10px;
  margin-bottom: 20px;
}

.session-item {
  display: flex;
  justify-content: space-between;
  align-items: center;
```

```
  padding: 10px 15px;
  background-color: white;
  border: 1px solid #ddd;
  border-radius: 5px;
}

.session-info {
  display: flex;
  flex-direction: column;
}

.session-date {
  font-size: 14px;
  color: #666;
}

.session-duration {
  font-weight: bold;
  color: #007bff;
}

.delete-session {
  background: none;
  border: none;
  font-size: 20px;
  cursor: pointer;
  color: #dc3545;
}

.no-sessions {
  text-align: center;
  padding: 20px;
  color: #666;
  font-style: italic;
}

.sessions-stats {
  padding: 15px;
  background-color: #e9ecef;
  border-radius: 5px;
}

.quick-actions {
  text-align: center;
}

.action-buttons {
  display: flex;
  gap: 10px;
  justify-content: center;
  flex-wrap: wrap;
}

.action-button {
```

```
  padding: 10px 20px;
  background-color: #6c757d;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

.action-button.special {
  background-color: #17a2b8;
}
</style>
```

Разбор кода Примера 3: события, lifecycle hooks и продвинутые вычисляемые свойства

Модификаторы событий: .prevent и .once

В Vue модификаторы событий предоставляют декларативный способ изменения поведения обработчиков:

Модификатор .prevent:

```
<button @click.prevent="addQuickSession(30)">
```

Эквивалентен вызову `event.preventDefault()` в обработчике события. Предотвращает стандартное действие браузера (например, отправку формы).

Модификатор .once:

```
<button @click.once="addOneTimeSession">
```

Обработчик сработает только один раз, после чего автоматически удаляется. Полезно для действий, которые должны выполниться однократно.

Сравнение с React:

React (ручная обработка):

```
<button onClick={(e) => {
  e.preventDefault();
  addQuickSession(30);
}}>
```

Vue (декларативный подход):

```
<button @click.prevent="addQuickSession(30)">
```

Цепочки модификаторов:

Модификаторы можно комбинировать:

```
<button @click.stop.prevent="handler">
<input @keyup.enter.exact="submit">  <!-- только Enter без других клавиш -->
<button @click.ctrl="handler">       <!-- только с нажатым Ctrl -->
```

Динамическое отключение кнопок с :disabled

```
<button :disabled="isRunning">Старт</button>
<button :disabled="!isRunning">Пауза</button>
```

Директива `:disabled` (сокращение от `v-bind:disabled`) динамически управляет состоянием кнопки. Когда значение `true`, кнопка становится неактивной.

CSS-класс `:disabled` автоматически применяется браузером, и мы можем его стилизовать:

```
.timer-button:disabled {
  opacity: 0.5;
  cursor: not-allowed;
}
```

Lifecycle hook: onUnmounted

```
onUnmounted(() => {
  if (timerInterval) {
    clearInterval(timerInterval)
  }
})
```

`onUnmounted` - это lifecycle hook, который выполняется перед удалением компонента из DOM. Критически важен для очистки ресурсов:

- Отмены таймеров и интервалов
- Отписки от событий
- Закрытия WebSocket соединений
- Отмены асинхронных операций

Сравнение с React:

React:

```
useEffect(() => {
  const interval = setInterval(() => {
    // логика
  }, 1000);

  // cleanup функция
  return () => clearInterval(interval);
}, []);
```

Vue:

```
let timerInterval = null;

onMounted(() => {
  timerInterval = setInterval(() => {
    // логика
  }, 1000);
});

onUnmounted(() => {
  clearInterval(timerInterval);
});
```

В Vue lifecycle hooks разделены на отдельные функции, что делает код более читаемым.

Реактивные vs нереактивные переменные

```
const elapsedTime = ref(0)         // реактивная переменная
let timerInterval = null           // обычная JavaScript переменная
```

Когда использовать ref:

- Данные отображаются в шаблоне
- Данные должны триггерить ре-рендер при изменении
- Данные используются в computed или watch

Когда использовать обычные переменные:

- Технические переменные (таймеры, флаги)
- Ссылки на DOM-элементы
- Кэш, не влияющий на UI

В данном примере `timerInterval` не нужна реактивность, так как она не отображается в UI и служит только для хранения ссылки на интервал.

Вычисляемое свойство с зависимостью от другого computed

```
const totalSessions = computed(() => studySessions.value.length)

const averageSessionTime = computed(() => {
  if (totalSessions.value === 0) return 0
  return Math.round(totalStudyTime.value / totalSessions.value)
})
```

Вычисляемое свойство `averageSessionTime` зависит от другого computed - `totalSessions`. Vue автоматически отслеживает всю цепочку зависимостей и пересчитывает значения в правильном порядке.

Граф зависимостей:

```
studySessions (ref)
    ├─> totalSessions (computed)
    │       └─> averageSessionTime (computed)
    └─> totalStudyTime (computed)
            └─> averageSessionTime (computed)
```

При изменении `studySessions` Vue пересчитает `totalSessions` и `totalStudyTime`, затем `averageSessionTime`.

Методы массивов в computed

```
const sortedSessions = computed(() =>
  [...studySessions.value].sort((a, b) => new Date(b.date) - new Date(a.date))
)
```

Важная деталь: используется spread-оператор `[...studySessions.value]` для создания копии массива перед сортировкой. Это необходимо, потому что метод `.sort()` мутирует исходный массив.

Почему это важно:

- Прямая мутация реактивного массива может вызвать непредсказуемое поведение
- Computed должны быть чистыми функциями без побочных эффектов
- Создание копии гарантирует неизменяемость исходных данных

Использование reduce для агрегации

```
const totalStudyTime = computed(() =>
  studySessions.value.reduce((total, session) => total + session.duration, 0)
)
```

Метод `.reduce()` - функциональный подход к агрегации данных:

- Первый параметр - reducer функция `(accumulator, currentValue) => newAccumulator`
- Второй параметр - начальное значение accumulator (0)

Пошаговое выполнение:

```
// Если массив: [{duration: 100}, {duration: 200}, {duration: 50}]
// Итерация 1: total = 0 + 100 = 100
// Итерация 2: total = 100 + 200 = 300
// Итерация 3: total = 300 + 50 = 350
// Результат: 350
```

Работа с Date и форматирование времени

```
const formatTime = (seconds) => {
  const mins = Math.floor(seconds / 60)
  const secs = seconds % 60
  return `${mins.toString().padStart(2, '0')}:${secs.toString().padStart(2, '0')}`
}
```

Разбор логики:

- `Math.floor(seconds / 60)` - целое количество минут
- `seconds % 60` - остаток секунд
- `.padStart(2, '0')` - добавляет ведущий ноль (5 → "05")
- Результат: "05:30" вместо "5:30"

Форматирование даты:

```
const formatDate = (dateString) => {
  return new Date(dateString).toLocaleString('ru-RU')
}
```

Метод `.toLocaleString('ru-RU')` форматирует дату согласно русской локали: "03.12.2025, 00:48:15"

Обработка событий с передачей параметров

```
<button @click="addQuickSession(30)">+30 мин</button>
<button @click="addQuickSession(60)">+1 час</button>
```

В Vue можно напрямую передавать аргументы в обработчики событий. Vue автоматически оборачивает вызов в функцию.

Эквивалентный код:

```
<button @click="() => addQuickSession(30)">+30 мин</button>
```

Если нужен доступ к объекту события вместе с параметрами:

```
<button @click="(event) => handler(event, param)">
<!-- или использовать специальную переменную $event -->
<button @click="handler($event, param)">
```

Принципы управления побочными эффектами

В данном примере демонстрируется правильное управление побочными эффектами (side effects):

1. **Создание эффекта** - в `startTimer()` создается setInterval
2. **Хранение ссылки** - ссылка сохраняется в `timerInterval`
3. **Очистка эффекта** - в `pauseTimer()` и `onUnmounted()` вызывается clearInterval

Этот паттерн предотвращает утечки памяти и обеспечивает корректное поведение при переключении между компонентами.

Композиция логики через функции

Вся логика таймера инкапсулирована в функции `setup()`. Для переиспользования этой логики в других компонентах можно создать composable:

```
// useTimer.js
export function useTimer() {
  const elapsedTime = ref(0)
  const isRunning = ref(false)
  let timerInterval = null

  const startTimer = () => { /* ... */ }
  const pauseTimer = () => { /* ... */ }
  const resetTimer = () => { /* ... */ }

  onUnmounted(() => {
    if (timerInterval) clearInterval(timerInterval)
  })

  return { elapsedTime, isRunning, startTimer, pauseTimer, resetTimer }
}

// использование в компоненте
import { useTimer } from './useTimer'

export default {
```

```
  setup() {
    const { elapsedTime, isRunning, startTimer, pauseTimer, resetTimer } =
useTimer()
    // дополнительная логика компонента
    return { elapsedTime, isRunning, startTimer, pauseTimer, resetTimer }
  }
}
```

Это аналог custom hooks в React, но с более явной композицией.

Публикация приложения на GitHub Pages

После завершения работы над приложением вы можете опубликовать его в интернете, используя GitHub Pages.

Использование пакета gh-pages

Публикация Vue приложения аналогична публикации React приложений.

Шаг 1: Установка пакета

```
npm install --save-dev gh-pages
```

Шаг 2: Настройка vite.config.js

Откройте файл `vite.config.js` и добавьте параметр `base`:

```
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

export default defineConfig({
  plugins: [vue()],
  base: '/vue-practice-27/'  // замените на имя вашего репозитория
})
```

Важно: значение `base` должно соответствовать имени вашего репозитория на GitHub. Если ваш репозиторий называется `my-vue-app`, то укажите `base: '/my-vue-app/'`.

Шаг 3: Добавление скриптов в package.json

Добавьте следующие скрипты в секцию `"scripts"` файла `package.json`:

```
{
  "scripts": {
    "dev": "vite",
```

```
    "build": "vite build",
    "preview": "vite preview",
    "predeploy": "npm run build",
    "deploy": "gh-pages -d dist"
  }
}
```

Пояснение:

- `"predeploy"` - автоматически выполняется перед `deploy`, собирает проект
- `"deploy"` - публикует содержимое папки `dist` в ветку `gh-pages`

Шаг 4: Инициализация git и первый коммит

Если вы еще не инициализировали git репозиторий:

```
git init
git add .
git commit -m "Initial commit"
```

Шаг 5: Подключение к GitHub

Создайте репозиторий на GitHub и подключите его:

```
git remote add origin https://github.com/<USERNAME>/<REPO>.git
git branch -M main
git push -u origin main
```

Шаг 6: Деплой одной командой!

Теперь для деплоя достаточно выполнить:

```
npm run deploy
```

Эта команда автоматически:

1. Соберет проект (`npm run build`)
2. Создаст ветку `gh-pages` (если её нет)
3. Опубликует содержимое `dist` в эту ветку
4. Отправит изменения на GitHub

Шаг 7: Настройка GitHub Pages

1. Откройте репозиторий на GitHub

2. Перейдите в **Settings** → **Pages**
3. В разделе **Source** выберите ветку `gh-pages` и папку `/ (root)`
4. Нажмите **Save**

Готово! Ваше приложение будет доступно по адресу: `https://<USERNAME>.github.io/<REPO>/`

Обновление приложения

После любых изменений в коде просто выполните:

```
git add .
git commit -m "Update app"
git push
npm run deploy
```

Устранение частых проблем

Проблема: Пустая страница или ошибка 404

Решение: Убедитесь, что в `vite.config.js` правильно указан параметр `base`:

```
base: '/имя-вашего-репозитория/'
```

Проблема: Стили и скрипты не загружаются

Решение: Проверьте, что пути к ресурсам используют относительные URL. Vite автоматически обрабатывает это при сборке, если `base` настроен правильно.

Проблема: GitHub Actions не запускается

Решение:

1. Проверьте права доступа в **Settings** → **Actions** → **General**
2. Убедитесь, что включена опция **Read and write permissions** для `GITHUB_TOKEN`

Обновление приложения

После внесения изменений в код:

Автоматический деплой (GitHub Actions):

```
git add .
git commit -m "Update application"
git push
```

Ручной деплой (скрипт):

```
npm run build
sh deploy.sh  # или deploy.bat для Windows
```

Контрольная работа №5

Задание для контрольной работы №5 разбито на несколько составляющих: 4 балла ставится за выполнения задания по практикам 27-28 4 балла ставится дополнительно за выполнение каждой из 4 контрольных работ (по 1 баллу за каждую работу).

Задание для практик 27-28: "Генератор цветовых палитр"

Общая концепция

Разработайте интерактивное веб-приложение на Vue.js для создания, управления и экспорта цветовых палитр. Приложение должно стать полезным инструментом для дизайнеров, разработчиков и всех, кто работает с визуальным контентом.

Общее описание проекта

Создайте генератор цветовых палитр, который позволяет пользователям:

- Генерировать гармоничные цветовые схемы
- Настраивать и редактировать палитры
- Анализировать цвета на соответствие стандартам доступности
- Сохранять и организовывать коллекции палитр
- Экспортировать цвета в различные форматы для использования в проектах

Проект реализуется в течение двух практических занятий с постепенным наращиванием функциональности.

Практика 27: Базовый функционал

Цель: Создать работающий прототип генератора с основными возможностями.

Обязательные функции:

1. Система генерации палитр

- Кнопка "Случайная палитра" - генерирует 5 гармоничных цветов
- Отображение палитры в виде горизонтальной полосы цветовых карточек
- Для каждого цвета показывать HEX-значение (например, #FF6B6B)

2. Управление отдельными цветами

- Клик по цветовой карточке копирует HEX-значение в буфер обмена
- Индикация успешного копирования (всплывающее уведомление)
- Возможность "закрепить" понравившийся цвет при генерации новой палитры

3. Базовые инструменты настройки

- Выбор количества цветов в палитре (3, 5, 7)
- Переключение между форматами отображения (HEX, RGB)
- Локальное сохранение текущей палитры в localStorage

4. Простой просмотрщик

- Превью палитры в mockup интерфейса (кнопка, карточка, заголовок)
- Переключение светлого/тёмного фона для тестирования контраста

Технические требования:

- Использование Vue 3 Composition API
- Реактивное управление состоянием цветов
- Применение директив v-for, v-if, v-model
- Вычисляемые свойства для преобразования цветов
- Обработка событий кликов и ввода

Критерии успеха практики 27:

1. ☑ Приложение генерирует случайные палитры
2. ☑ Цвета отображаются с HEX-значениями
3. ☑ Работает копирование в буфер обмена
4. ☑ Палитра сохраняется между перезагрузками
5. ☑ Интерфейс интуитивно понятен

Практика 28: Продвинутый функционал (Advanced Features)

Цель: Превратить прототип в полнофункциональный инструмент.

Дополнительные функции:

1. Продвинутая генерация

- Генерация на основе базового цвета (пользователь выбирает основной цвет)
- Различные типы палитр: аналогичная, монохромная, триада, комплементарная
- Генерация по "настроению": спокойные, энергичные, профессиональные палитры

2. Инструменты анализа и доступности

- Проверка контрастности между цветами по стандарту WCAG
- Показ уровня доступности (AA, AAA, недостаточно)
- Подбор акцентных цветов для выбранной палитры

- Графическое представление цветового круга

3. Управление библиотекой палитр

- Сохранение палитр в именованные коллекции
- Поиск и фильтрация по названиям и тегам
- Возможность редактирования сохранённых палитр
- Создание избранных коллекций

4. Экспорт и интеграция

- Экспорт в форматы: CSS variables, SCSS variables, Tailwind config
- Генерация готового CSS кода с цветами
- Превью палитры в различных UI-компонентах
- Создание шаринговых ссылок на палитры

Технические требования:

- Использование Vue Router для навигации
- Компонентный подход с передачей props
- Работа с v-model в кастомных компонентах
- Использование watchers для реактивных изменений
- Интеграция с внешними API для цветовых преобразований
