

# Advanced Vue

# Component Registration

# Local Registration

Каждый vue компонент должен быть зарегистрирован.

Есть два способа регистрации компонентов, одна из них локальная:

Script setup

```
<script setup>
import ComponentA from './ComponentA.vue'
</script>

<template>
  <ComponentA />
</template>
```

Non script setup

```
import ComponentA from './ComponentA.js'

export default {
  components: {
    ComponentA
  },
  setup() {
    // ...
  }
}
```

# Global Registration

Глобальная регистрация

*Особенности:*

- *Глобально зарегистрированный компонент можно не импортировать при использовании*

main.ts

```
import MyComponent from './App.vue'

app.component('MyComponent', MyComponent)
```

Причем регистрировать компоненты можно “цепочкой”

```
app
  .component('ComponentA', ComponentA)
  .component('ComponentB', ComponentB)
  .component('ComponentC', ComponentC)
```

# Global Registration

Глобально зарегистрированные компоненты можно использовать в шаблоне любого компонента в приложении:

```
<script setup lang="ts"></script>

<template>
  <ComponentA/>
  <ComponentB/>
  <ComponentC/>
</template>
```

- 1) Глобальная регистрация не позволяет системам сборки удалять неиспользуемые компоненты
- 2) Глобальная регистрация делает отношения зависимости менее явными в больших приложениях.

# Props

# Props

Vue предоставляет возможность не просто определить пропсы, но и указать определенные параметры для них (напр. тип или опциональность). Причем сделать это можно как на JS, так и на TS. Здесь и далее будем рассматривать только Composition Api

## Javascript

```
<script setup>
const props = defineProps({
  foo: {
    type: String,
    required: true,
  },
  bar: {
    type: Number,
    default: undefined
  }
})

console.log(props.foo)
</script>
```

## Typescript

```
<script setup>
const props = defineProps<{
  foo: string,
  bar?: number,
}>()

console.log(props.foo)
</script>
```

# JS vs TS props declaration

Помимо этого можно добавить кастомные правила валидации пропсов.  
Но JS очень многословен, поэтому в дальнейшем в примерах всегда будем использовать Typescript

## Javascript

```
const props = defineProps({  
  order: {  
    type: Number,  
    required: true,  
    validate: (value) => [1, -1].includes(value)  
  },  
})  
  
console.log(props.order)
```

## Typescript

```
const props = defineProps<{  
  order: 1 | -1  
  
console.log(props.order)
```



# Prop passing details

ArticleCard.vue

```
<script setup lang="ts">
const props = defineProps<{
  title: string
  text: string
  paymentRequired: boolean
}>()
</script>
```

ParentComponent.vue

```
<template>
  <ArticleCard
    title="Hello World!"
    :text="article.text"
    payment-required
  />
</template>
```

Можно передавать пропсы как статические значения или передавать их динамически с помощью v-bind

boolean переменные можно указывать в таком стиле, эквивалентно

```
:payment-required="true"
```

# Prop passing details

Причем пропсы можно передавать как объект с помощью v-bind, тогда все поля объекта будут переданы компоненту

ParentComponent.vue

```
<script setup lang="ts">
const card = {
  title: "Hello World!",
  text: article.text,
  paymentRequired: true,
}
</script>

<template>
  <ArticleCard v-bind="card"/>
</template>
```

Часто v-bind используется в связке с **\$props** - объектом содержащим все переданные пропсы (доступно только в темплейте)

```
<script setup lang="ts">
defineProps<{
  label: string;
  required: number;
}>();
</script>

<template>
  <MyInput v-bind="$props" />
</template>
```

# One-Way Data Flow

Важно, все пропсы образуют одностороннюю привязку между дочерним и родительским свойствами: когда родительское свойство обновляется, оно передается дочернему свойству, но не наоборот. Это предотвращает случайное изменение состояния родительского компонента дочерними компонентами, что может затруднить понимание потока данных вашего приложения.

Если сообщать родителю об изменении пропса не требуется, но подписка на изменение пропса нужна, то можно использовать утил функцию `toRef`.

Подробнее в [документации](#)

```
const props = defineProps(['foo'])
```

```
// ✗ warning, props are readonly!  
props.foo = 'bar'
```

```
const state = reactive({  
  foo: 1,  
  bar: 2  
})
```

```
// a two-way ref that syncs with the  
original property  
const fooRef = toRef(state, 'foo')
```

Emits

# Emits

Для передачи данных родительскому компоненту используются эмиты.

Сначала регистрируем событие

Потом “эмиттим” его

ChildComponent.vue

```
<script setup>
const emit = defineEmit<{
  (e: 'change', id: number): void
}>()

function buttonClick() {
  emit('change', someNumberValue)
}
</script>
```

ParentComponent.vue

```
<template>
  <ChildComponent @change="onChange">
</template>
```

В родительском компоненте с помощью директивы v-on можем навесить обработчик события onChange

# Emits

Эмиты используются и для передачи какого то события без данных, например событие закрытия модального окна

MyModal.vue

```
<script setup lang="ts">
defineProps<{
  open: boolean;
}>();
const emit = defineEmits<{
  (e: 'close'): void;
}>();
</script>

<template>
  <div v-if="open">
    <div @click="$emit('close')">Close</div>
    <div>
      <slot /> <!-- next in this lecture -->
    </div>
  </div>
</template>
```

ParentComponent.vue

```
<script setup lang="ts">
const isModalOpen = ref(false);
</script>

<template>
  <div>
    <div @click="isModalOpen = true"> Open Modal </div>
    <MyModal
      :open="isModalOpen"
      @close="isModalOpen = false"
    >
      Hi there!
    </MyModal>
  </div>
</template>
```

# V-model

# V-model

Мы можем сами реализовать двустороннюю v-model связку для кастомных компонентов. Для этого нужно определить:

- 1) Пропс с именем `modelValue`, с которым синхронизируется значение реактивной переменной;
- 2) Событие с именем `update:modelValue`, которое генерируется при изменении значения переменной.

MyInput.vue

```
<script setup lang="ts">
defineProps<{
  modelValue: string;
}>();
defineEmits<{
  (e: 'update:modelValue', value: string): void;
}>();
</script>

<template>
  <input
    :value="props.modelValue"
    @input="$emit('update:modelValue', $event.target.value)"
  />
</template>
```

ParentComponent.vue

```
<script setup lang="ts">
const str = ref('');
</script>

<template>
  <MyInput v-model="str" />
</template>
```



# Multiple v-model bindings

Мы можем накладывать двустороннюю связку не только на пропс `modelValue`, но и на любой другой. Так мы можем определить двустороннее связывание для пропса `text` или любого другого

MyInput.vue

```
<script setup lang="ts">
defineProps<{
  text: string;
}>();
defineEmits<{
  (e: 'update:text', value: string): void;
}>();
</script>

<template>
  <input
    :value="props.text"
    @input="$emit('update:text', $event.target.value)"
  />
</template>
```

ParentComponent.vue

```
<script setup lang="ts">
const str = ref('');
</script>

<template>
  <MyInput v-model:text="str" />
</template>
```

# Multiple v-model bindings

Причем v-model связываний может быть несколько.

Реализация по аналогии

```
<UserName  
  v-model:first-name="first"  
  v-model:last-name="last"  
>
```

Подробнее о v-model в [документации](#).

В обновлении Vue 3.4 был добавлен макрос `defineModel`, который упрощает работу с v-model

# Fallthrough Attributes

# Fallthrough Attributes

Fallthrough attribute — это атрибут или прослушиватель событий v-on, который передается компоненту, но не объявлен явно в пропсах или эмитах. Типичными примерами этого являются атрибуты class, style и id.

MyButton.vue

```
<template>
  <button class="btn">click me</button>
</template>
```

Мы уже рассматривали такой пример. Все переданные в родительском компоненте атрибуты будут добавлены корневому элементу компонента MyButton.vue

Итого зарендеренный DOM будет выглядеть так:

ParentComponent.vue

```
<template>
  <MyButton
    class="large"
    :style="{ width: '12px' }"
    @click="onClick"
  />
</template>
```



```
<button
  class="btn large"
  style="width: 12px"
  onclick="onClick"
>
  click me
</button>
```

Такое же поведение будет, если корневой элемент MyButton - другой компонент

# Fallthrough Attributes

Иногда нам может потребоваться обернуть фактический элемент `<button>` дополнительным `<div>`

Но мы хотим, чтобы все атрибуты, такие как прослушивающие классы и `v-on`, были применены к внутреннему `<button>`, а не к внешнему `<div>`.

Мы можем добиться этого с помощью `v-bind="$attrs"`:

MyButton.vue

```
<div class="btn-wrapper">  
  <button class="btn">click me</button>  
</div>
```

MyButton.vue

```
<div class="btn-wrapper">  
  <button  
    class="btn"  
    v-bind="$attrs"  
  >  
    click me  
  </button>  
</div>
```

# Attribute Inheritance on Multiple Root Nodes

Но для элементов с несколькими корневыми элементами будет runtime warning, поэтому мы должны явно привязать атрибуты на выбранный элемент с помощью `v-bind="$attrs"`

CustomLayout.vue

```
<header>...</header>  
<main>...</main>  
<footer>...</footer>
```

ParentComponent.v

```
<CustomLayout id="custom-layout" @click="changeValue" />
```

```
⚠ [Vue warn]: Extraneous non-props attributes (class) were passed to runtime-core.esm-bundler.js:41  
component but could not be automatically inherited because component renders fragment or text root nodes.  
  at <TestComp class="foo" >  
  at <App>
```

Slots

# Slot Content

В некоторых случаях мы можем захотеть передать фрагмент шаблона дочернему компоненту и позволить дочернему компоненту отображать этот фрагмент в своем собственном шаблоне.

MyButton.vue

```
<template>
  <button class="my-btn">
    <slot />
  </button>
</template>
```

Тогда зарендеренный DOM будет выглядеть так:

```
<button class="fancy-btn">Click me!</button>
```

ParentComponent.vue


```
<MyButton>
  Click me! <!-- slot content -->
</MyButton>
```



# Render Scope

Содержимое слота имеет доступ к области данных родительского компонента, поскольку оно определено в родительском компоненте. Например:

ParentComponent.vue



```
<script setup lang="ts">
const message = ref('Hello world!');
</script>

<template>
  <div>
    <span>{{ message }}</span>
    <MyButton>{{ message }}</MyButton>
  </div>
</template>
```

Содержимое слота не имеет доступа к данным дочернего компонента. Выражения в шаблонах Vue имеют доступ только к той области, в которой они определены, в соответствии с лексической областью действия JavaScript.

# Fallback Content

Мы можем указать дефолтное значение для слота (т.е. если слот не был передан)

MyButton.vue

```
<button type="submit">
  <slot>
    Submit <!-- fallback content -->
  </slot>
</button>
```

ParentComponent.vue

```
<SubmitButton />
```

Rendered DOM

```
<button type="submit">Submit</button>
```

ParentComponent.vue

```
<SubmitButton>Save</SubmitButton>
```

Rendered DOM

```
<button type="submit">Save</button>
```

# Named Slots

Бывают случаи, когда полезно иметь несколько слотов в одном компоненте. Это удобно например при проектировании лэйаутов. В этих случаях элемент `<slot>` имеет специальный атрибут `name`, который можно использовать для присвоения уникальных идентификаторов различным слотам, чтобы вы могли определить, где контент должен отображаться. Слот без имени неявно имеет имя `default`.

Чтобы передать именованный слот, нам нужно использовать элемент `<template>` с директивой `v-slot`, а затем передать имя слота в качестве аргумента `v-slot`:

MyLayout.vue

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

ParentComponent.vue

```
<MyLayout>
  <template v-slot:header>
    <h1>Here might be a page title</h1>
  </template>

  <template #default>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </template>

  <template #footer>
    <p>Here's some contact info</p>
  </template>
</MyLayout>
```

# Named Slots

Тогда зарендеренный DOM будет выглядеть так:

```
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

# Dynamic Slot Names

Мы также можем указывать в родительском компоненте имя слота динамическим значением, а не литералом

ParentComponent.vue

```
<MyLayout>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>

  <!-- with shorthand -->
  <template #[dynamicSlotName]>
    ...
  </template>
</MyLayout>
```

# Scoped Slots

Как уже обсуждалось содержимое слота не имеет доступа к состоянию дочернего компонента. Однако мы можем передать данные из дочерней области в слот. Для этого используются **Scoped slots**. Фактически, мы можем передавать атрибуты в слот точно так же, как передаем пропсы компоненту:

MyComponent.vue

```
<div>
  <slot :text="greetingMessage" :count="1"></slot>
</div>
```

ParentComponent.vue

```
<MyComponent v-slot="slotProps">
  {{ slotProps.text }} {{ slotProps.count }}
</MyComponent>
```

# Scoped Slots

Причем данные можно передавать и в именованные слоты. Реализация по аналогии

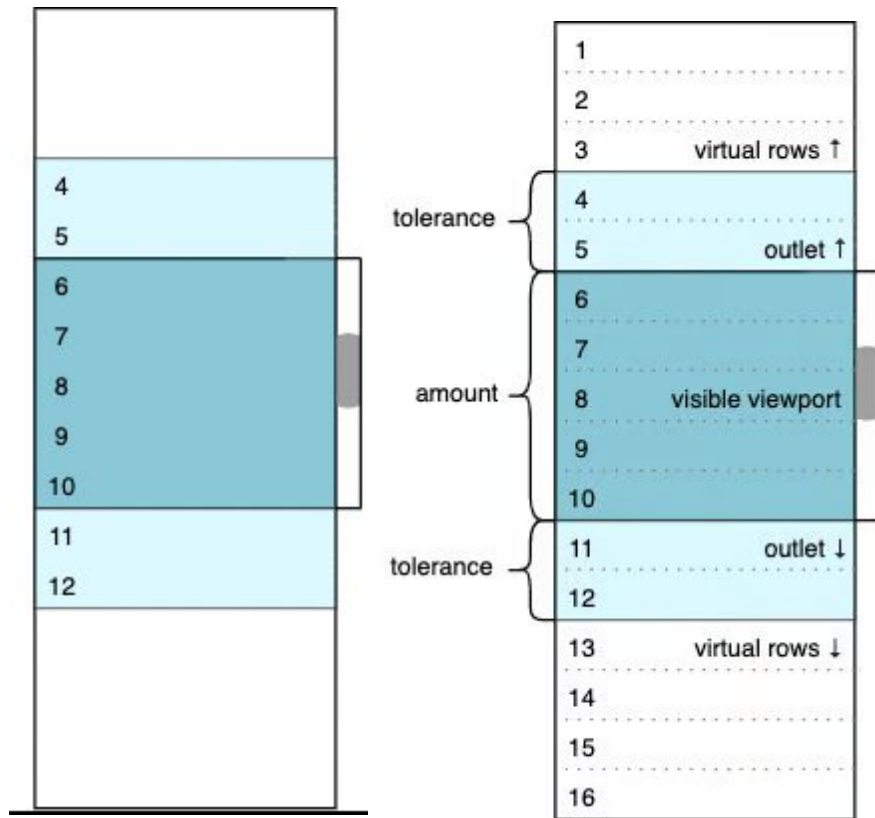
```
<MyComponent>
  <template v-slot:header="headerProps">
    {{ headerProps }}
  </template>

  <template #default="defaultProps">
    {{ defaultProps }}
  </template>
</MyComponent>
```

# Virtual scroll

Хороший пример использования  
Scoped slots - виртуальный скролл.

```
<v-virtual-scroll
  :height="300"
  :items="Array.from({length: 1000}).map((it, i) => `${i}`)"
>
  <template v-slot:default="{ item }">
    Item {{ item }}
  </template>
</v-virtual-scroll>
```

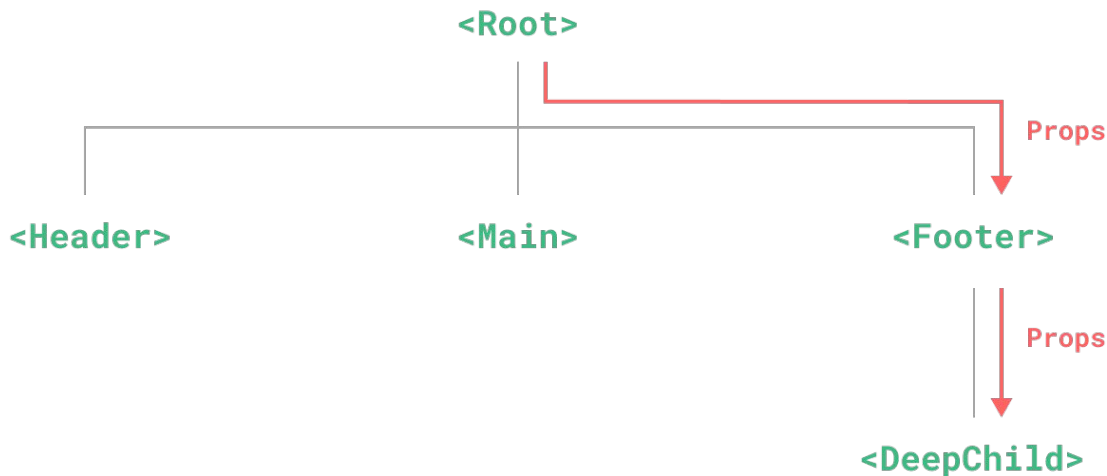




Provide / Inject

# Prop Drilling

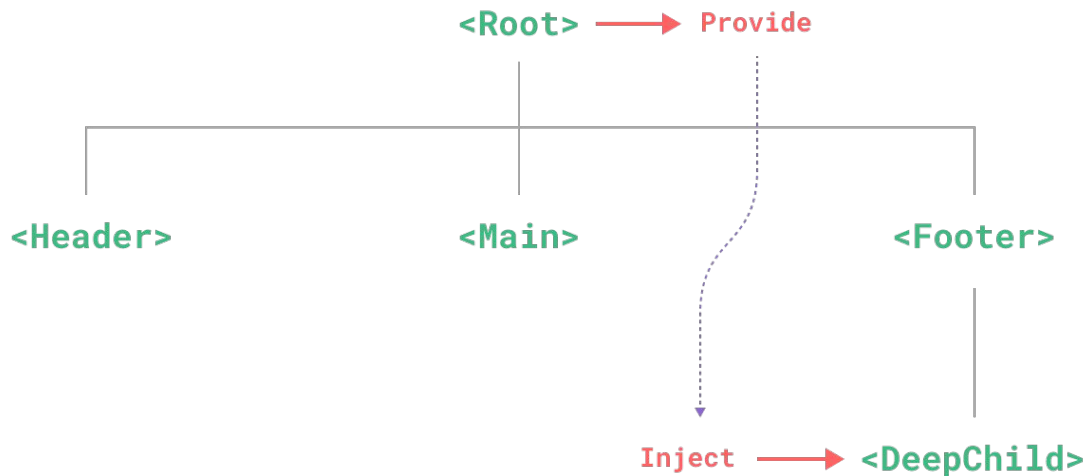
Обычно, когда нам нужно передать данные от родительского компонента к дочернему, мы используем пропсы. Однако, когда у нас большое дерево компонентов, а глубоко вложенному компоненту нужно что-то от компонента-предка, то используя только пропсы, нам пришлось бы передавать один и тот же пропс по всей родительской цепочке:



# Provide / Inject

Ситуация с Props Drilling ухудшается, если компоненты между Root и DeepChild вообще не используют “drilling”-пропсы.

На помощь приходит **Provide / Inject**



# Provide / Inject

- 1) Один компонент может вызывать метод `provide()` несколько раз с разными ключами внедрения для предоставления разных значений.
- 2) Второй аргумент — это предоставленное значение. Значение может быть любого типа, включая реактивное состояние, например ссылки

Root.vue

```
import { ref, provide } from 'vue'

const message = ref('Hello world!')
provide('message', message)
```

DeepChild.vue

```
<script setup>
import { inject } from 'vue'

const message: Ref<string> | undefined = inject('message')
</script>
```

Важно: инжектировать данные можно **только** во вложенных компонентах компонента-поставщика

# Provide / Inject

Причем если мы провайдем реактивное значение, то DeepChild может изменить ее значение, что может быть не безопасно. Для того чтобы предотвратить такую возможность можно использовать readonly.

Подробнее в [документации](#)

Root.vue

```
<script setup lang="ts">
import { ref, provide, readonly } from 'vue'

const count = ref(0)
provide('read-only-count', readonly(count))
</script>
```

# Composables

# Composables

Composable функция это Composition API функция которая использует логику отслеживанием состояния.

Они немного отличаются от стандартных util функций, так как если стандартные утильсы вызываются и сразу же возвращают результат, composables же возвращают состояние, которое меняется с течением времени. Простой пример - отслеживание позиции курсора мыши

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

export function useMouse() {
  const x = ref(0)
  const y = ref(0)

  function update(event) {
    x.value = event.pageX
    y.value = event.pageY
  }

  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))

  return { x, y }
}
```

По соглашению имена составных функций начинаются с «use».

Состояние, инкапсулированное и управляемое composable

Составной объект может со временем обновлять свое управляемое состояние.

Composable также может подключаться к жизненному циклу своего компонента-владельца для определения и устранения побочных эффектов.

# Composables

Использовать можем так:

```
<script setup lang="ts">
import { useMouse } from './mouse.ts'

const { x, y } = useMouse()
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>
```



# useFetch

## Еще один пример composable-функции

Определяем стейт

Следим за изменением  
реактивной переменной и  
вызываем рефетч

Вызываем toValue и получаем  
значение реактивной  
переменной или геттера

```
import { MaybeRefOrGetter, ref, toValue } from 'vue';

export const useFetch = <TData, TError = unknown>(url: MaybeRefOrGetter<string> | ComputedRef<string>) => {
  const data = ref<TData | null>(null);
  const error = ref<TError | null>(null);

  const fetchData = async () => {
    data.value = null;
    error.value = null;

    try {
      const res = await fetch(toValue(url));
      const json = await res.json();
      data.value = json;
    } catch (e) {
      error.value = e;
    }
  };

  watchEffect(fetchData);

  return { data, error };
};
```

# useFetch

Используем так:

```
<script setup lang="ts">
import { useFetch } from './fetch';
const props = defineProps<{
  url: string;
}>();
const { data, error } = useFetch(() => props.url);
// const { data, error } = useFetch(toRef(props, 'url'));
</script>

<template>
  <div>
    <div v-if="data === null && error === null">Loading...</div>

    <template v-else-if="data">
      <p>data:</p>
      <pre>{{ data }}</pre>
    </template>

    <template v-else-if="error">
      <p>error:</p>
      <pre>{{ error }}</pre>
    </template>
  </div>
</template>
```

Теперь при изменении пропса будет вызываться рефетч по передаваемому пропсу

Причем мы можем использовать как геттер `() => props.url` так и `toRef(props, 'url')`

# Custom Directives

# Custom Directives

В дополнение к набору директив по умолчанию, поставляемому в ядре (например, v-model или v-show), Vue также позволяет вам регистрировать ваши собственные директивы.

```
<script setup>
const vFocus = {
  mounted: (el) => el.focus()
}
</script>

<template>
  <input v-focus />
</template>
```

Как только Input будет вмонтирован в страницу, сработает фокус. Причем это работает не только при загрузке страницы но и когда элемент динамически вставляется в страницу

# Custom Directives

В дополнение к набору директив по умолчанию (v-model, v-show), Vue также позволяет регистрировать собственные директивы.

```
const myDirective = {
  created(el, binding, vnode, prevVnode) {},
  beforeMount(el, binding, vnode, prevVnode) {},
  mounted(el, binding, vnode, prevVnode) {},
  beforeUpdate(el, binding, vnode, prevVnode) {},
  updated(el, binding, vnode, prevVnode) {},
  beforeUnmount(el, binding, vnode, prevVnode) {},
  unmounted(el, binding, vnode, prevVnode) {}
}
```

```
<div v-example:foo.bar="baz"/>
```

**el**: элемент, к которому привязана директива. Это можно использовать для прямого манипулирования DOM.

**binding**: объект, содержащий информацию о значении, аргументах и модификаторах переданных в директиву.

**vnode**: базовый VNode, представляющий связанный элемент.

**prevNode**: VNode, представляющий связанный элемент из предыдущего рендеринга. Доступно только в хуках beforeUpdate и Updated.

Transition  
TransitionGroup

# Transition

`<Transition>` — встроенный компонент, который можно использовать для применения анимации к элементам, переданным ему через слот.

```
<template>
  <button @click="show = !show">Toggle</button>
  <Transition>
    <p v-if="show">hello</p>
  </Transition>
</template>

<style scoped lang="scss">
.v-enter-active,
.v-leave-active {
  transition: opacity 0.5s ease;
}

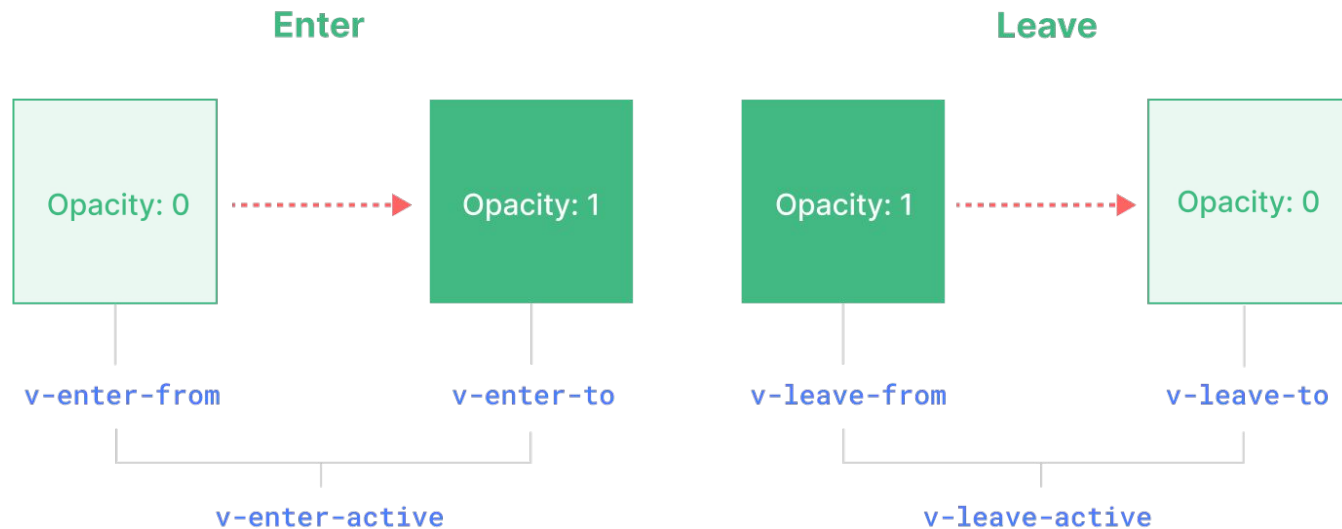
.v-enter-from,
.v-leave-to {
  opacity: 0;
}
</style>
```

Причем анимация будет срабатывать при следующих триггерах:

- 1) Условный рендеринг через **v-if**
- 2) Условное отображение через **v-show**
- 3) Динамическое переключение компонентов с помощью **<component>**
- 4) Изменение атрибута **key**

# Transition

Для переходов enter/leave применяется эти шесть классов



Подробнее в [документации](#)



# Transition Group

Встроенный компонент, предназначенный для анимации вставки, удаления и изменения порядка элементов или компонентов, отображаемых в списке.

Отличия от `<Transition>`:

- 1) Классы перехода CSS будут применяться к **отдельным элементам** в списке, а не к самой группе/контейнеру.
- 2) Элементы внутри всегда должны иметь **key**.
- 3) По умолчанию не отображает элемент-контейнер. Но можно указать элемент, который будет отображаться, с помощью пропса **tag**

Больше примеров в [документации](#)

```
<TransitionGroup name="list" tag="ul">
  <li v-for="item in items" :key="item">
    {{ item }}
  </li>
</TransitionGroup>
```

```
.list-enter-active,
.list-leave-active {
  transition: all 0.5s ease;
}
.list-enter-from,
.list-leave-to {
  opacity: 0;
  transform: translateX(30px);
}
```

KeepAlive

# Keep Alive

По умолчанию экземпляр активного компонента будет размонтирован при его отключении. Когда этот компонент отобразится снова, будет создан новый экземпляр с исходным состоянием. Это означает что состояние компонента будет утеряно. Используя **<KeepAlive>** мы можем закешировать состояние компонента

```
<KeepAlive>  
  <component :is="activeComponent" />  
</KeepAlive>
```

☒ A   ☐ B

Current component: A

count: 0

+

Teleport

# Teleport

Иногда мы можем столкнуться со следующей ситуацией: часть шаблона компонента логически принадлежит ему, но с визуальной точки зрения она должна отображаться где-то еще в DOM, вне приложения Vue.

Самый распространенный пример — создание полноэкранного модального окна.

```
<button @click="open = true">  
  Open Modal  
</button>  
<MyModal />
```

```
<Teleport to="body">  
  <div v-if="open" class="modal">  
    <p>Hello from the modal!</p>  
    <button @click="open = false">Close</button>  
  </div>  
</Teleport>
```

В таком случае в фактическом DOM контент нашей модальки будет находиться в конце body

# Темы для докладов

1) SSR

<https://vuejs.org/guide/scaling-up/ssr.html>

2) Render functions

<https://vuejs.org/guide/extras/render-function>

Спасибо за внимание!