

Vue Introduction



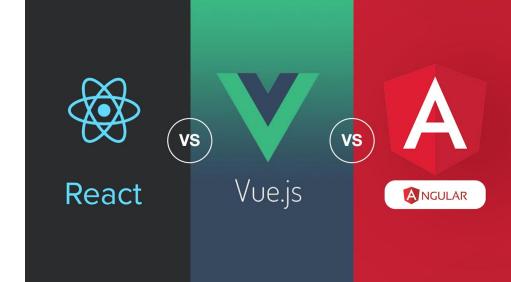
What is Vue.js

Vue.js (произносится как “вью”) - прогрессивный JavaScript фреймворк (<https://vuejs.org/>).

Vue.js (сокр. просто **Vue**) - это удобный, производительный и универсальный фреймворк для создания пользовательских веб-интерфейсов.

Vue был создан сотрудником Google Эваном Ю в 2014 году и сразу стал набирать популярность.

По результатам опроса [State of JavaScript 2022](#) Vue активно используют 46.2% фронтенд разработчиков.



Advantages of Vue

Чем Vue лучше React и Angular?

- Vue.js обычно считается более простым и интуитивным, чем Angular и React.
- Vue.js известен своей исчерпывающей и понятной документацией
- Vue.js обычно меньше по размеру, чем Angular и React, что может привести к быстрейшей загрузке и лучшему времени отклика.
- Vue.js легко интегрируется с существующими проектами и технологиями, что делает его удобным для добавления нового функционала к уже существующим приложениям.
- Vue идеально подходит для:
 - Небольших и средних веб-приложений
 - Проектов с ограниченным бюджетом и временем
 - Обучения и прототипирования

Connecting Vue to project

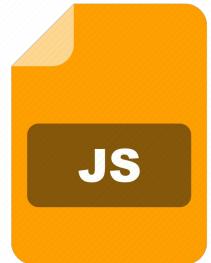
Ways to connect Vue to the project

Есть 2 основных способа подключения Vue к проекту:

1) Без использования средств сборки

Особенности:

- Простое подключение
- Возможность использовать Vue, как обычную библиотеку



2) С использованием средств сборки

Особенности:

- Возможность использования Single-File Components (.vue файлов компонентов)
- Основной способ создания Vue приложений.



Connecting Vue without build tools

Если средства сборки не используются, то можно подключить Vue на страницу просто с помощью CDN через тег `<script>`:

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script> ← Подключение Vue.js (Global build)
<script>
    // Vue object is available here
</script> ← Код вашего приложения
```

Либо можно использовать версию Vue в виде ES модуля:

```
<script type="module">
    import * as Vue from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js' ← Подключение Vue.js (ESM build)
    // Vue object is available here
</script> ← Код вашего приложения
```

(обратите внимание, что ESM модули не работают через `file://`)

Clicked 7 times

Increment counter!



A simple Vue application

Рассмотрим создание простого кликера с помощью Vue:

```
<div id="app">
  <div>Clicked {{ count }} times</div>
  <button v-on:click="onClick">Increment counter!</button>
</div>

<script type="module">
  import * as Vue from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'

  const app = Vue.createApp({
    setup() {
      const count = Vue.ref(0)           ← Стейт
      const onClick = () => {
        count.value++                  ← Метод
      }
      return {
        count,
        onClick
      }
    }
  })
  app.mount("#app")
</script>
```

Готовим разметку приложения
(незнакомые браузеру элементы разметки будут обработаны Vue при инициализации)

Импортируем Vue как ESM модуль

Настраиваем Vue приложение

Возвращаем стейт и метод, чтобы они стали доступны в разметке

Привязываем приложение к разметке
(div элементу с id="app")

Clicked 7 times

Increment counter!

A simple Vue application

С помощью деструктуризации можно импортировать методы объекта Vue напрямую:

```
<div id="app">
  <div>Clicked {{ count }} times</div>
  <button v-on:click="onClick">Increment counter!</button>
</div>

<script type="module">
  import { createApp, ref } from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'
    ↗
  const app = createApp({
    setup() {
      const count = ref(0)
      const onClick = () => {
        count.value++
      }
      return {
        count,
        onClick
      }
    }
  })
  app.mount( "#app" )
</script>
```

Импорт необходимых методов из Vue

importmaps

Недавнее новшество браузеров - **importmaps** позволяет сократить url для импорта, дав ему псевдоним:

```
<script type="importmap">
{
  "imports": {
    "vue": "https://unpkg.com/vue@3/dist/vue.esm-browser.js"
  }
}
</script>

<script type="module">
  import { createApp, ref } from 'vue'
  // ...
</script>
```

Говорим браузеру, по какому URL расположен модуль 'vue'

Вместо длинного URL можем писать короткое название модуля 'vue'

API Styles

У Vue есть два варианта API: **Options API** and **Composition API**.

На предыдущем слайде мы использовали Composition API. Но мы могли бы с легкостью написать наше приложение с использованием Options API. Вот сравнение между ними:

Options API:

```
const app = createApp({
  data() {
    return {
      count: 0
    }
  },
  methods: {
    onClick() {
      this.count++
    }
  }
})
```

Composition API:

```
const app = createApp({
  setup() {
    const count = ref(0)
    const onClick = () => {
      count.value++
    }
    return {
      count,
      onClick
    }
  }
})
```

Options API vs Composition API

Composition API сейчас является более приоритетным, поэтому в дальнейшем мы будем использовать именно его.

В [документации Vue](#) вы всегда можете переключать API стиль между **Composition API** и **Options API**.

Но вот что сама документация пишет о выборе между ними:

- Go with **Options API** if you are not using build tools, or plan to use Vue primarily in low-complexity scenarios, e.g. progressive enhancement.
- Go with **Composition API + Single-File Components** if you plan to build full applications with Vue.

Generating a Vue project with Vite

Стандартный способ создания Vue приложений - с помощью официального сборщика на основе Vite.

Проект инициализируется одной командой:

```
npm create vue@latest
```

После ее выполнения мы отвечаем на вопросы
сборщика:

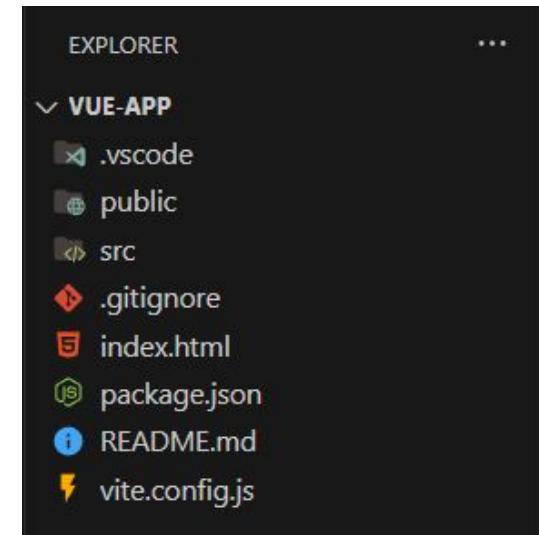
```
Vue.js - The Progressive JavaScript Framework

✓ Project name: ... vue-app
✓ Add TypeScript? ... No / Yes
✓ Add JSX Support? ... No / Yes
✓ Add Vue Router for Single Page Application development? ... No / Yes
✓ Add Pinia for state management? ... No / Yes
✓ Add Vitest for Unit Testing? ... No / Yes
✓ Add an End-to-End Testing Solution? » No
✓ Add ESLint for code quality? ... No / Yes

Scaffolding project in C:\Users\aramt\Desktop\temp\vue-app...
Done. Now run:

cd vue-app
npm install
npm run dev
```

и проект создан:



Generating a Vue project with Vite

Можно зайти в файл `src/App.vue` и, отредактировав его, создать в нем наш кликер.

`App.vue`

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

const count = ref(0)

const onClick = () => {
  count.value++
}

</script>

<template>
  <div id="app">
    <div>Clicked {{ count }} times</div>
    <button v-on:click="onClick">Increment counter!</button>
  </div>
</template>

<style scoped>
</style>
```

Каждый файл `.vue` представляет собой vue компонент. Он состоит из 3-х частей:

script setup - инициализация объявление состояний и методов компонента

template - определение разметки компонента

style scoped - изолированные стили компонента

Generating a Vue project with Vite

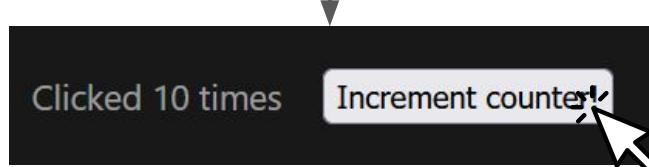
Сборщик стандартного проекта Vue поддерживает сборку и dev сервер:

Запуск **dev** сервера

```
npm run dev
```

build проекта

```
npm run build
```



Generating a Vue project with Vite

При отсутствии сборщика мы изначально пишем код в `.html`, `.css`, `.js` файлах и их же деплоим. Так действовали разработчики в прошлом. Этот подход не позволял использовать что-то кроме `html`, `css`, `js`, так как браузер понимает только эти технологии. Но все изменилось с приходом сборщиков.

Использование сборщика предполагает, что мы пишем исходный код в папке `src/` и будем пользоваться технологиями, которые непонятны обычному браузеру (например файлами `.vue`, `.ts`, `.scss`).

Потом, когда мы захотим собрать наш проект для деплоя, то сборщик соберет все что мы накодили в папке `src/`, преобразует в обычные файлы `.html`, `.css`, `.js` понятные браузеру и положит в отдельную папку `dist/` (или `build/` в зависимости от сборщика).

Кроме того большинство сборщиков поддерживают так называемый режим **dev сервера**, который позволяет быстро поднять локальный сервер, который хостит наше приложение. Он избавляет от необходимости при каждом изменении в папке `src/` делать билд проекта в `dist/` и выгружать его на реальный сервер.

Creating a Vue Application

createApp

Каждое приложение Vue начинается с создания нового экземпляра приложения с помощью функции `createApp`:

```
import { createApp } from 'vue'

const app = createApp({
  /* root component options */
})
```

На вход `createApp` принимает корневой компонент
(в виде Options API либо Composition API)

```
const app = createApp({
  data() {
    return {
      count: 0
    }
  },
  methods: {
    onClick() {
      this.count++
    }
  }
})
```

```
const app = createApp({
  setup() {
    const count = ref(0)
    const onClick = () => {
      count.value++
    }
    return {
      count,
      onClick
    }
  }
})
```

Mounting the App

После создания Vue приложения с помощью `createApp` его нужно привязать привязать к реальному DOM элементу с помощью метода `mount`:

В html задается контейнер:

```
<div id="app"></div>
```

У приложения вызывается метод `mount`, куда передается селектор контейнера:

```
app.mount('#app')
```

После этого Vue приложение будет рендерится внутри данного DOM элемента.

Creating & Mounting

В сгенерированном с помощью `npm create vue@latest` проекте в файле `main.js` можно увидеть вызов метода `createApp` с последующей привязкой к DOM с помощью `mount`:

```
import { createApp } from 'vue'  
import App from './App.vue'  
  
createApp(App).mount('#app')
```

В данном случае корневой компонент импортируется из `.vue` файла Single-File Component (SFC).

Создаем приложение, передав корневой компонент и привязываем его к DOM контейнеру

Можно создать одновременно несколько инстансов приложений, привязанных к разным DOM контейнерам:

```
import { createApp } from 'vue'  
import App1 from './App1.vue'  
import App2 from './App2.vue'  
  
createApp(App1).mount('#container-1')  
createApp(App2).mount('#container-2')
```

App Configurations

Каждый экземпляр приложения app открывает объект config, который содержит настройки конфигурации для этого приложения. Мы можем изменить его свойства (описанные [в документации](#)) перед монтированием приложения.

Одна из самых полезных настроек - это глобальный обработчик ошибок errorHandler, отлавливающий необработанные ошибки из всех компонентов приложения.

```
app.config.errorHandler = (err) => {
  alert(`Unhandled error: ${err.message}`)
}
```

Вывод текста ошибки
пользователю

```
<button @click="() => { throw new Error('Error msg') }">Click!</button>
```

localhost:5173 says
Unhandled error: Error msg

OK

Using templates

Templates

В Vue используется синтаксис шаблонов на основе HTML, позволяющий декларативно связывать визуализируемый DOM с данными экземпляра компонента. Эти шаблоны, являющиеся валидным HTML, компилируются Vue в высокооптимизированный JavaScript-код.

Простейшая подстановка свойства компонента в шаблон происходит с помощью "Mustache" синтаксиса: `{{ property }}`

Template в HTML:

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>


{{ message }}


<script>
  const { createApp, ref } = Vue
  createApp({
    setup() {
      const message = ref('Hello vue!')
      return {
        message
      }
    }
  }).mount('#app')
</script>
```

Template в SFC (App.vue файле):

```
<script setup>
import { ref } from "vue"
const message = ref("Hello vue!")
</script>

<template>
  <div class="app">{{ message }}</div>
</template>
```

Output raw HTML (v-html directive)

Простейшая подстановка с помощью "Mustache" синтаксиса: `{{ property }}` экранирует HTML.

Таким образом Vue обеспечивает защиту от [XSS](#).

```
const message = ref("<h1>Hello vue!</h1>")
```

```
<h1>Hello vue!</h1>
```

```
<div>{{ message }}</div>
```

Чтобы в элемент шаблона подставить HTML код без экранирования используется специальный атрибут-директива `v-html`:

```
<div v-html="message"></div>
```

Hello vue!

Attribute Bindings (v-bind directive)

Использовать “Mustache” {{ }} подстановку **нельзя**, если требуется подставить значение в атрибут:

```
<div class="{{ dynamicClass }}"></div>
```

Так не сработает!

Для подстановки в атрибут используется директива v-bind:<название атрибута>

```
<div v-bind:class="dynamicClass"></div>
```

Или сокращенный вариант синтаксиса :<название атрибута>

```
<div :class="dynamicClass"></div>
```

Можно биндить сразу
несколько атрибутов, обернув
их в объект и передав в v-bind:

```
const objectOfAttrs = {  
  id: 'container',  
  class: 'wrapper'  
}
```

```
<div v-bind="objectOfAttrs"></div>
```

(Если происходит бинд булевого атрибута, то он будет включен в случае true значения состояния и не включен в случае false значения.)

JS Expressions in templates

Vue фактически поддерживает всю мощь выражений JavaScript внутри всех биндов данных:

Внутри текстовых
интерполяций "Mustache":

```
{{ speed / 1000 * 3600 }}  
{{ ok ? 'YES' : 'NO' }}
```

В атрибутах директив Vue (атрибуты, начинающиеся с v-):

```
<input :placeholder="message.split('').reverse().join('')" type="text">  
<div :id=`list-${id}`></div>
```

Каждая привязка может содержать только одно expression (при этом statement использовать нельзя):
(так работать **не** будет!)

```
<!-- this is a statement, not an expression: -->  
{{ var a = 1 }}  
  
<!-- flow control won't work either, use ternary expressions -->  
{{ if (ok) { return message } }}
```

Внутри бинда можно вызвать метод, определенный в
компоненте (он будет вызываться при каждой
перерисовке):

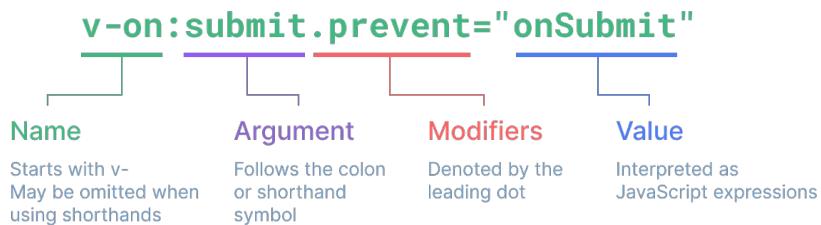
```
{{ calculateValue(data) }}
```

Directives

Мы рассмотрели пока только две директивы `v-html` и `v-bind`.

Но Vue поддерживает и другие директивы: `v-if`, `v-for`, `v-on`, `v-slot` и т д.

О них мы будем говорить позднее, но все директивы имеют общий синтаксис:



Атрибут в директиве может быть **динамическим** (указывается в квадратных скобках)

```
<a v-bind:[attributeName]="url"> ... </a>
```

Поэтому когда мы пишем `v-bind:class="message"` для привязки свойства `message` к атрибуту `class`, то “`class`” идущий после двоеточия “`:`” называется **аргументом** директивы `v-bind`.

Reactivity

Declaring Reactive State

Для создания реактивных состояний во Vue в большинстве случаев используется функция `ref()`.

`ref()` принимает аргумент и возвращает его в виде ref-объекта со свойством `.value`:

```
const count = ref(0)
```

```
count.value // returns 0
```

Для доступа к стейту из JavaScript
используется свойство `.value`

```
setTimeout(() => {
  count.value++ // changing the state after one second
}, 1000)
```

В шаблоне для подстановки используется имя стейта без `.value`

```
<div>{{ count }}</div>
```

Шаблон перерисуется

Как только `count.value` поменяется.

Обычные переменные Vue не отслеживаются!

Если попытаться использовать в шаблоне обычные переменные вместо ref-состояний, то компонент **не будет обновляться** при изменении значения переменной

```
let count = 0
```

```
setTimeout(() => {
  count++ // changing the variable after one second
}, 1000)
```

```
<div>{{ count }}</div>
```

Шаблон не перерисуется!

В нем останется начальное значение (в нашем случае 0)

Using Reactive State in Template

При использовании Composition API без SFC (`script setup`) в методе `setup()` необходимо возвращать состояния и методы, которые должны быть доступны в разметке (т.е шаблоне):

```
import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)

    // expose the ref to the template
    return {
      count
    }
  }
}

<div>{{ count }}</div>
```

Импорты верхнего уровня, переменные и функции, объявленные в `<script setup>`, автоматически можно использовать в шаблоне того же компонента.

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

const count = ref(0)
</script>

<template>
  <div>{{ count }}</div>
</template>
```

В дальнейшем будем придерживаться синтаксиса Composition API с SFC (т.е. со `script setup`). Т.к. он используется наиболее часто на практике.

Deep Reactivity

`ref()` поддерживает реактивность объектов любой вложенности.

Реактивность объектов можно обеспечить другим способом (без обертки в объект со свойством `value`): с помощью функции `reactive()`. Она делает объект реактивным

```
const state = ref({  
  nested: { count: 0 },  
  arr: ['str1', 'str2']  
})
```

```
state.value.nested.count++ // Vue tracks that
```

```
const state = reactive({  
  nested: { count: 0 },  
  arr: ['str1', 'str2']  
})
```

```
state.nested.count++ // Vue tracks that
```

DOM Update Timing

При изменении реактивного состояния, DOM обновляется автоматически. Но эти обновления не применяются синхронно. Вместо этого Vue буферизирует их до "следующего тика" в цикле обновления, чтобы гарантировать, что каждый компонент обновляется только один раз, независимо от того, сколько изменений состояния вы сделали.

Чтобы дождаться завершения обновления DOM после изменения состояния, можно использовать функцию `nextTick()`:

```
import { nextTick } from 'vue'

async function increment() {
  count.value++
  await nextTick()
  // Now the DOM is updated
}
```

Caveat when Unwrapping in Templates

unwrapping `ref.value` в шаблонах применяется только в том случае, если `ref` является свойством верхнего уровня в контексте рендеринга шаблона:

```
const count = ref(0)
const object = { id: ref(1) }
```

```
{{ count + 1 }}
```

работает корректно

```
{{ object.id + 1 }}
```

не работает корректно

```
{{ object.id.value + 1 }}
```

в данном случае нужно явно обращаться к свойству `.value`

Больше подробностей о тонкостях работы с `ref` и `reactive` можно найти [в документации](#).

Class and Style Bindings

Binding HTML Classes

Помимо строк `v-bind:class` может принимать объект или массив для облегчения задания классов.

```
const isActive = ref(true)
const hasError = ref(false)
```

```
<div
  class="static"
  :class="{ active: isActive, 'text-danger': hasError }"
></div>
```

В результате:

```
<div class="static active"></div>
```

```
const activeClass = ref('active')
const errorClass = ref('text-danger')
```

```
<div :class="[activeClass, errorClass]"></div>
```

В результате:

```
<div class="active text-danger"></div>
```

Binding Inline Styles

Помимо строк `v-bind:style` может принимать объект или массив для облегчения задания inline-стилей.

Через ref:

```
const activeColor = ref('red')
const fontSize = ref(30)
```

```
<div :style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
```

Через reactive:

```
const styleObject = reactive({
  color: 'red',
  fontSize: '13px'
})
```

```
<div :style="styleObject"></div>
```

Если передать массив объектов стилей, то Vue объединит эти объекты и применит их к одному и тому же элементу.

```
<div :style="[baseStyles, overridingStyles]"></div>
```

Computed Properties

computed

Можно делать вычисления на основе реактивных свойств прямо в шаблоне, но они будут повторяться при каждой перерисовке компонента.

```
<p>Has published books:</p>
<span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
```

Чтобы это оптимизировать можно вынести вычисления в отдельный метод и обернуть его в функцию `compute`:

```
import { computed } from 'vue'
```

```
// a computed ref
const publishedBooksMessage = computed(() => {
  return author.books.length > 0 ? 'Yes' : 'No'
})
```

```
const author = reactive({
  name: 'John Doe',
  books: [
    'book1',
    'book2',
    'book3'
  ]
})
```

И в шаблоне можно подставлять этот метод.
Пересчет значения будет происходить только при изменении реактивных зависимостей.

```
<p>Has published books:</p>
<span>{{ publishedBooksMessage }}</span>
```

Writable computed

Computed свойства по умолчанию предназначены только для чтения.

Чтобы сделать возможных запись в них нужно явно задать геттер и сеттер:

Тогда при присваивании computed свойству определенного значения будет вызван соответствующий сеттер

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

const firstName = ref('Bob')
const lastName = ref('Roe')

const fullName = computed({
    // getter
    get() {
        return firstName.value + ' ' + lastName.value
    },
    // setter
    set(newValue) {
        // we are using destructuring assignment syntax here
        [firstName.value, lastName.value] = newValue.split(' ')
    }
})
```

```
fullName.value = 'John Doe'
```

Conditional Rendering

Conditional Rendering

Для условного рендеринга используются директивы: `v-if`, `v-else`, `v-else-if`

Они определяют, какой элемент будет рендерится в зависимости от значения выражения (которое обычно зависит от реактивного свойства).

```
const show = ref(true)
```

определяем реактивное свойство для обозначения видимости блока

```
<button @click="show = !show">Toggle</button>
```

по нажатию на кнопку можем менять значение свойства `show`, чтобы управлять видимостью блока

```
<h1 v-if="show">Content shown :)</h1>
```

если условие истинно (`show == true`), то будет показан этот блок

```
<h1 v-else>content hidden</h1>
```

иначе, показан этот блок

Обратите внимание, что блока с `v-else` может не быть. В этом случае при ложном условии не будет выведено никакого блока.

v-else-if

С помощью директивы `v-else-if` можно строить длинные цепочки альтернатив для условного рендеринга.

```
<div v-if="type === 'A'">A</div>
<div v-else-if="type === 'B'">B</div>
<div v-else-if="type === 'C'">C</div>
<div v-else>Not A/B/C</div>
```



Окончить цепочку можно блоком `v-else`, который будет отрендерен, если не выполнилось никакое из условий выше.

Эти цепочки имеют ту же самую логику, что и обычные конструкции `if-else` в программировании.

v-if on <template>

По-умолчанию директивы условного рендеринга (`v-if`, `v-else`, `v-else-if`) работают с одним элементом.

Чтобы работать с несколькими элементами, нужно обернуть их в один `div` или `template`. В последнем случае элементы будут рендерится без создания дополнительного блока-обертки (т.е. сам `<template>` не будет содержаться в результирующей разметке).

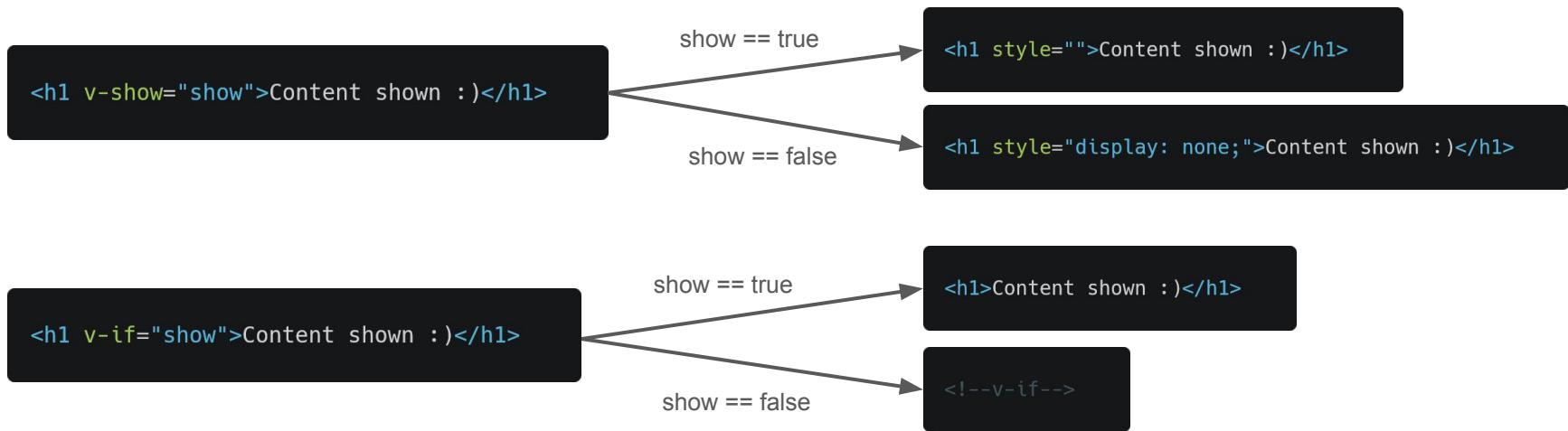
```
<div v-if="show">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</div>
<div v-else>
  <h1>Not allowed</h1>
  <p>Sorry, this block is unavailable</p>
</div>
```

```
<template v-if="show">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
<template v-else>
  <h1>Not allowed</h1>
  <p>Sorry, this block is unavailable</p>
</template>
```

v-show

Альтернативой v-if является директива v-show, которая управляет видимостью элемента с помощью CSS свойства display.

Поэтому когда условие ложно, то v-show устанавливает элементу display: none (в то время как v-if в этом случае удаляет элемент со страницы).



v-show не поддерживает элемент <template> и не работает с v-else.

List Rendering

List Rendering

Для вывода списков используется директива `v-for`

Как правило список базируется на каком-то реактивном массиве объектов:

```
const items = ref([{ message: 'msg1' }, { message: 'msg2' }, { message: 'msg3' }])
```

- msg1
- msg2
- msg3

`v-for = "<псевдоним текущего элемента внутри цикла v-for> in <имя массива элементов>"`

```
<ul>
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
```

результат

```
▼ <ul>
  ▼ <li>
    ::marker
    "msg1"
  </li>
  ▼ <li>
    ::marker
    "msg2"
  </li>
  ▼ <li>
    ::marker
    "msg3"
  </li>
</ul>
```

v-for with index and key

v-for также поддерживает необязательный второй псевдоним для индекса текущего элемента:

```
<div v-for="(item, index) in items">  
    {{ index }} - {{ item.message }}  
</div>
```

результат

```
<div>0 - msg1</div>  
<div>1 - msg2</div>  
<div>2 - msg3</div>
```

```
0 - msg1  
1 - msg2  
2 - msg3
```

Vue по-умолчанию использует для вывода списков стратегию ["in-place patch"](#), которая предполагает, что вывод списка не зависит от состояния дочерних компонентов или временного состояния DOM (например, значений ввода формы).

Чтобы дать Vue указание, позволяющее отслеживать идентичность каждого узла и, таким образом, повторно использовать и переупорядочивать существующие элементы, нам нужно предоставить уникальный ключевой атрибут `key` для каждого элемента:

```
<div v-for="item in items" :key="item.id">  
    <!-- content -->  
</div>
```

обычно id содержится в самих объектах, которые мы выводим (напр. комментарии или посты)
этот id мы можем использовать в качестве key

nested v-for

Vue поддерживает вложенные v-for.

```
const items = ref([
  { children: [{ msg: 'msg1' }, { msg: 'msg2' }, { msg: 'msg3' }] },
  { children: [{ msg: 'msg4' }, { msg: 'msg5' }, { msg: 'msg6' }] },
  { children: [{ msg: 'msg7' }, { msg: 'msg8' }, { msg: 'msg9' }] },
])
```

```
<div v-for="item in items">
  <div v-for="child in item.children">
    {{ child.msg }}
  </div>
</div>
```

```
▼ <div>
  <div>msg1</div>
  <div>msg2</div>
  <div>msg3</div>
</div>
▼ <div>
  <div>msg4</div>
  <div>msg5</div>
  <div>msg6</div>
</div>
▼ <div>
  <div>msg7</div>
  <div>msg8</div>
  <div>msg9</div>
</div>
```

v-for with objects and ranges

v-for позволяет итерироваться по свойствам объекта

```
const myObject = reactive({  
    title: 'Title',  
    author: 'Author',  
    publishedAt: '2017-09-08'  
})
```

```
<ul>  
    <li v-for="(value, key, index) in myObject">  
        {{ index }} - {{ key }}: {{ value }}  
    </li>  
</ul>
```

```
▼<ul>  
  ▼<li>  
    ::marker  
    "0 - title: Title"  
  </li>  
  ▼<li>  
    ::marker  
    "1 - author: Author"  
  </li>  
  ▼<li>  
    ::marker  
    "2 - publishedAt: 2017-09-08"  
  </li>  
</ul>
```

v-for позволяет также итерироваться по диапазонам 1..n

```
<div v-for="n in 10">{{ n }}</div>
```

```
<div>1</div>  
<div>2</div>  
<div>3</div>  
<div>4</div>  
<div>5</div>  
<div>6</div>  
<div>7</div>  
<div>8</div>  
<div>9</div>  
<div>10</div>
```

Event handlers

Event handlers

Функции, определенные в компоненте могут использоваться внутри шаблона. Они передаются в директиву **v-on**

Синтаксис директивы: **v-on:<название события>**

```
<script setup>
import { ref } from "vue"
const count = ref(0)

const incrementCount = () => {
  count.value++
}

</script>

<template>
  <div>{{ count }}</div>
  <button v-on:click="incrementCount">increment!</button>
</template>
```

сокращенный вариант директивы:
@<название события>

```
<button @click="incrementCount">increment!</button>
```

Также возможны Inline обработчики:

```
<button @click="count++">Add 1</button>
```

Event object

Каждый обработчик принимает по-умолчанию один аргумент - нативный DOM объект события

```
const clickHandle = (event) => {
  // `event` is the native DOM event
  console.log(`click x: ${event.clientX}, y:${event.clientY}`)
}
```

```
<button @click="clickHandle">increment!</button>
```

Но с помощью Inline обработчиков можно делать вызов метода с произвольными параметрами:

```
function say(message) {
  alert(message)
}
```

```
<button @click="say('hello')">Say hello</button>
<button @click="say('bye')">Say bye</button>
```

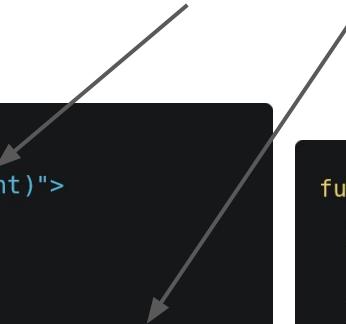
То есть метод вызывается не напрямую, а через Inline обработчик с передачей произвольных аргументов.

Accessing Event Argument in Inline Handlers

Чтобы не терять объект события при кастомном вызове метода из inline обработчика, его можно передать в метод с помощью встроенной переменной `$event`, либо обернув вызов метода в стрелочную функцию.

```
<!-- using $event special variable -->
<button @click="warn('Form cannot be submitted yet.', $event)">
  Submit
</button>

<!-- using inline arrow function -->
<button @click="(event) => warn('Form cannot be submitted yet.', event)">
  Submit
</button>
```



```
function warn(message, event) {
  // now we have access to the native event
  if (event) {
    event.preventDefault()
  }
  alert(message)
}
```

Event Modifiers

Если объект события нужен в методе только чтобы вызвать `event.preventDefault()` и `event.stopPropagation()`, то можно переложить эту задачу на Vue с помощью модификаторы события, которые пишутся через точку после имени события:

```
<!-- the click event's propagation will be stopped -->
<a @click.stop="doThis"></a>

<!-- the submit event will no longer reload the page -->
<form @submit.prevent="onSubmit"></form>
```

Vue предоставляет и другие дополнительные модификаторы:

- `.self`
- `.capture`
- `.once`
- `.passive`

```
<!-- modifiers can be chained -->
<a @click.stop.prevent="doThat"></a>

<!-- just the modifier -->
<form @submit.prevent></form>
```

```
<!-- only trigger handler if event.target is the element itself -->
<!-- i.e. not from a child element -->
<div @click.self="doThat">...</div>

<!-- use capture mode when adding the event listener -->
<!-- i.e. an event targeting an inner element is handled here before
being handled by that element -->
<div @click.capture="doThis">...</div>

<!-- the click event will be triggered at most once -->
<a @click.once="doThis"></a>

<!-- the scroll event's default behavior (scrolling) will happen -->
<!-- immediately, instead of waiting for `onScroll` to complete -->
<!-- in case it contains `event.preventDefault()` -->
<div @scroll.passive="onScroll">...</div>
```

Key Modifiers & Mouse Button Modifiers

Vue предоставляет и другие модификаторы для упрощения работы, например, для событий нажатия клавиш клавиатуры или мыши:

```
<!-- only call `submit` when the `key` is `Enter` -->
<input @keyup.enter="submit" />

<!-- we can directly use any valid key names exposed via
KeyboardEvent.key as modifiers by converting them to kebab-case -->
<input @keyup.page-down="onPageDown" />
```

```
<!-- this will fire if you click via right mouse button -->
<button @click.right="onRightClick">A</button>
```

Более подробно о различных модификаторах v-on можно почитать [в документации](#).

Form Input Bindings

Form Input Bindings

Очень часто при работе с формами требуется синхронизировать значение в поле ввода с нужным реактивным состоянием во Vue. Это называется *Two-way binding* (Двустороннее связывание). Двусторонним оно называется потому что нужно добиться связи в две стороны:

1) поле ввода -> состояние

Т.е. когда пользователь вводит что-то в поле ввода, реактивное состояние должно меняться.

2) состояние -> поле ввода

Т.е. когда реактивное состояние меняется (напр. JS кодом), то и текст в поле ввода тоже должен измениться.

Two-way binding можно реализовать следующим образом:

```
const text = ref('') ← состояния, которое будет привязано к полю ввода
```

```
<input  
  @input="event => text = event.target.value" ← привязка поле ввода -> состояние  
  :value="text"> ← привязка состояние -> поле ввода
```

v-model

Для *Two-way binding* существует отдельная директива `v-model`, которая обеспечивает короткий синтаксис для двустороннего связывания:

```
const text = ref('')
```

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

Это аналогично заданию связей вручную через `v-on` и `v-bind` (как в предыдущем примере)

`v-model` работает и для других элементов форм:

```
<textarea v-model="message"></textarea>
```

(в `message` будет находиться текст)

```
<input type="checkbox" id="checkbox" v-model="checked" />
```

(в `checked` будет находиться `true` или `false`)

```
<select v-model="selected">
  <option value="a">A</option>
  <option value="b">B</option>
  <option value="c">C</option>
</select>
```

(в `selected` будет находиться 'a' или 'b' или 'c')

более подробно об этом можно почитать [в документации](#)

v-model modifiers

Директива v-model поддерживает следующие модификаторы:

.lazy

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

.number

```
<input v-model.number="age" />
```

.trim

```
<input v-model.trim="message" />
```

.lazy дает указание Vue обновлять состояние только при возникновении события "change" (а не "input", как это происходит по-умолчанию)

.number делает автоматическое приведение к числу (parseFloat)

.trim делает автоматическую обрезку пробельных символов в начале и в конце строки

Lifecycle Hooks

Lifecycle Hooks

При создании каждый экземпляр компонента Vue проходит ряд этапов инициализации - например, ему необходимо настроить наблюдение за данными, скомпилировать шаблон (template), подключить экземпляр к DOM и обновить DOM при изменении данных.

Попутно выполняются функции, называемые хуками жизненного цикла, которые дают нам возможность добавлять свой собственный код на определенных этапах.

Например, часто возникает необходимость сделать что-то при загрузке компонента. Для этого мы можем воспользоваться хуком жизненного цикла `onMounted`:

```
import { onMounted } from 'vue';

onMounted(() => {
  // will be executed on component mount
})
```

В `onMounted` передаем функцию, которая будет вызвана после загрузки компонента.

Loading data on mounting a component

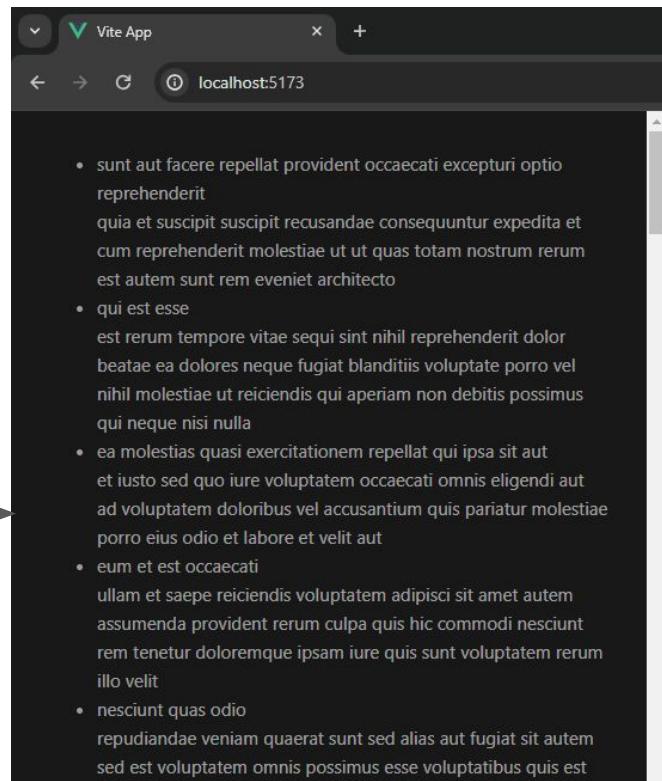
Часто при загрузке компонента необходимо подгрузить данные из апи. А потом вывести загруженные данные в шаблоне. Это также можно реализовать с помощью хука жизненного цикла `onMounted`

`App.vue`:

```
<script setup>
import { onMounted, ref } from 'vue';

const posts = ref([])
onMounted(async () => {
  const data = await fetch('https://jsonplaceholder.typicode.com/posts')
  posts.value = await data.json()
})
</script>

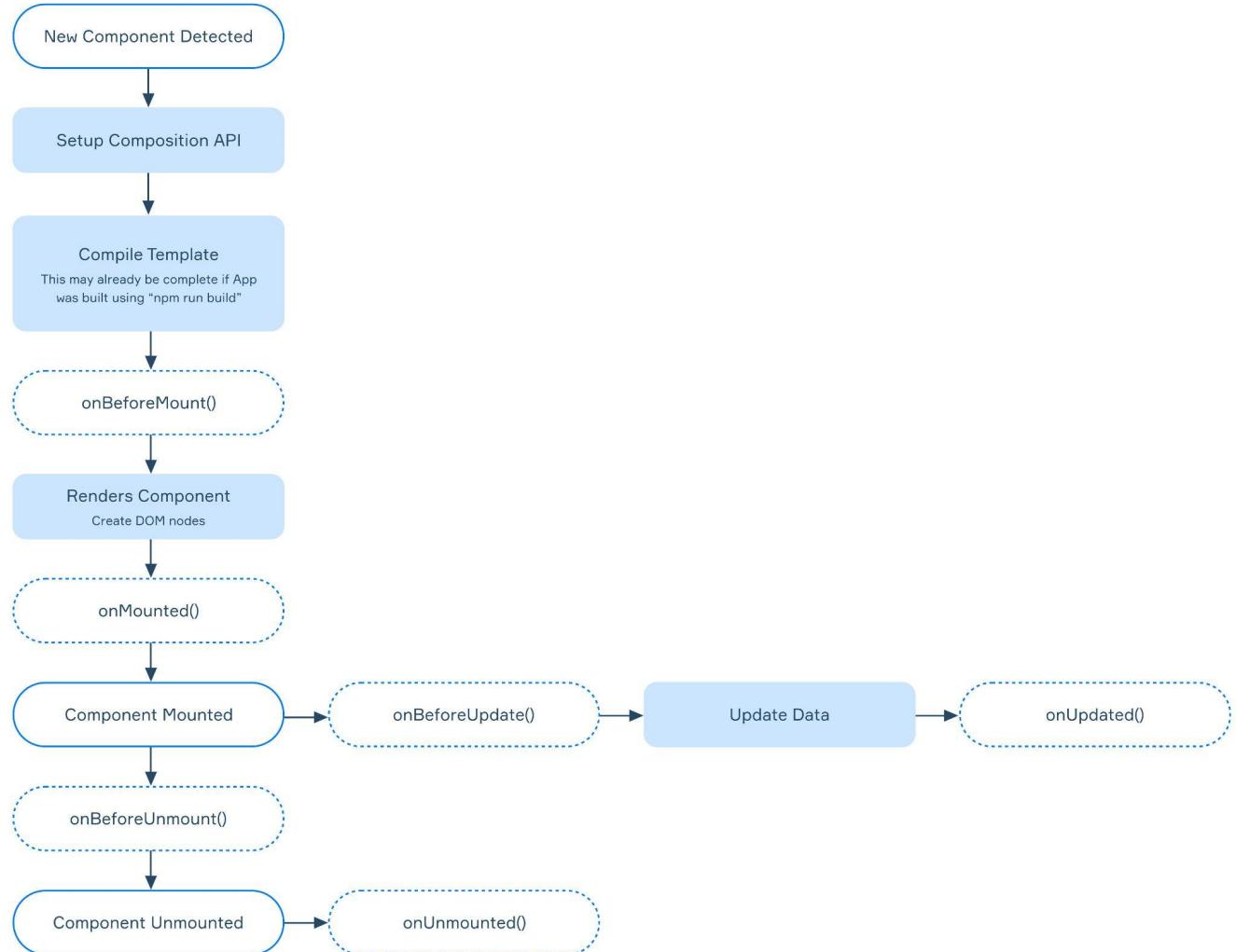
<template>
  <ul class="posts">
    <li v-for="post in posts">
      <div>{{ post.title }}</div>
      <div>{{ post.body }}</div>
    </li>
  </ul>
</template>
```



Lifecycle

Помимо `onMounted` существуют и другие хуки. Они также принимают на вход функцию и вызывают ее в определенный момент жизненного цикла компонента.

На диаграмме жизненного цикла вы можете видеть эти хуки:



onUpdated and onUnmounted

Также помимо `onMounted` существует еще 2 очень распространенных хука:

`onUpdated` - выполняется при обновлении любого реактивного свойства, вызывающего перерисовку шаблона компонента

`onUnmounted` - выполняется при удалении компонента со страницы (например, из-за условного рендеринга)

Например, у нас есть компонент `Clicker.vue`, где заданы эти хуки:

```
<script setup>
import { onUnmounted, onUpdated, ref } from 'vue';

const count = ref(0);

onUpdated(() => {
  console.log(`Updated: ${count.value}`)
});
onUnmounted(() => {
  console.log('Unmounted')
})
</script>

<template>
  <button @click="count++">{{ count }}</button>
</template>
```

Тогда при любом изменении количества очков будет вызван `onUpdated` в `App.vue`:

Чтобы увидеть вызов `onUnmounted` достаточно в родительском для `Clicker` компоненте (напр. `App.vue`) реализовать условный рендеринг, при котором компонент `Clicker` может быть удален со страницы:

```
<script setup>
import { ref } from 'vue';
import Clicker from './components/Clicker.vue';

const showClicker = ref(true)
</script>

<template>
  <div>
    <Clicker v-if="showClicker"/>
    <button @click="showClicker = !showClicker">show/hide clicker</button>
  </div>
</template>
```

Watchers

Watchers

Бывают случаи, когда нам нужно выполнить "side эффекты" в ответ на изменение состояния - например, мутировать DOM или изменить другое состояние на основе результата асинхронной операции.

В Composition API мы можем использовать функцию `watch` для запуска `callback` при каждом изменении реактивного состояния:

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

const state = ref('')

// watch works directly on a ref
watch(state, (newValue, oldValue) => {
  // handle side effect
})
```

передаем реактивное состояние, для которого хотим установить callback

callback вызывается при каждом изменении состояния
(ему передается новое и старое значение в качестве параметров)

Watching the properties

Мы не можем просто так передать свойство реактивного объекта в `watch`:

```
const obj = reactive({ count: 0 })

// this won't work because we are passing a number to watch()
watch(obj.count, (count) => {
  console.log(`count is: ${count}`)
})
```

Для наблюдения за свойством нужно обернуть свойство в метод. Таким образом, Vue будет следить за тем, когда значение, возвращаемое данным методом изменится и это будет означать изменение нужного нам свойства.

```
// instead, use a getter:
watch(
  () => obj.count, ←
  (count) => {
    console.log(`count is: ${count}`)
  }
)
```

передаем метод для отслеживания
свойства реактивного объекта

Deep Watchers

Если мы передаем реактивное свойство напрямую в `watch`, то наблюдение будет по-умолчанию глубоким (отслеживающим изменения вложенных свойств). Но при передаче метода (геттера) `watch` будет отслеживать только изменение возвращаемого методом значения, то есть только неглубокие изменения (переприсваивание свойства):

```
watch(  
  () => state.someObject, ←  
  () => {  
    // fires only when state.someObject is replaced  
  }  
)
```

`watch` будет отслеживать только
переприсваивание свойства `someObject`

```
watch(  
  () => state.someObject,  
  (newValue, oldValue) => {  
    // Note: `newValue` will be equal to `oldValue` here  
    // *unless* state.someObject has been replaced  
  },  
  { deep: true } ←  
)
```

если мы укажем третьим параметром значение
`deep: true`, то Vue будет отслеживать все изменения
внутри объекта-свойства `someObject`, а не только его
переприсваивание

Eager Watchers

Иногда мы можем захотеть, чтобы callback вызвался в самом начале при загрузке компонента, а не только при изменении реактивного состояния.

В таком случае мы можем передать третьим аргументом параметр

`immediate: true`

```
watch(  
  source,  
  (newValue, oldValue) => {  
    // executed immediately, then again when `source` changes  
  },  
  { immediate: true } ←  
)
```

передача параметра `immediate: true` говорит Vue о необходимости вызвать callback дополнительно сразу при загрузке компонента

watchEffect

У `watch` есть альтернатива `watchEffect`, которая не требует передачи отслеживаемого объекта. Она принимает только `callback` и автоматически отслеживает изменение реактивных состояний, присутствующих в `callback`-е.

Также `watchEffect` выполняется первый раз в самом начале (аналогично `watch` с `immediate: true`)

Следующие два фрагмента кода аналогичны:

```
const todoId = ref(1)
const data = ref(null)

watch(
  todoId,
  async () => {
    const response = await fetch(
      `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
    )
    data.value = await response.json()
  },
  { immediate: true }
)
```

```
watchEffect(async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
})
```

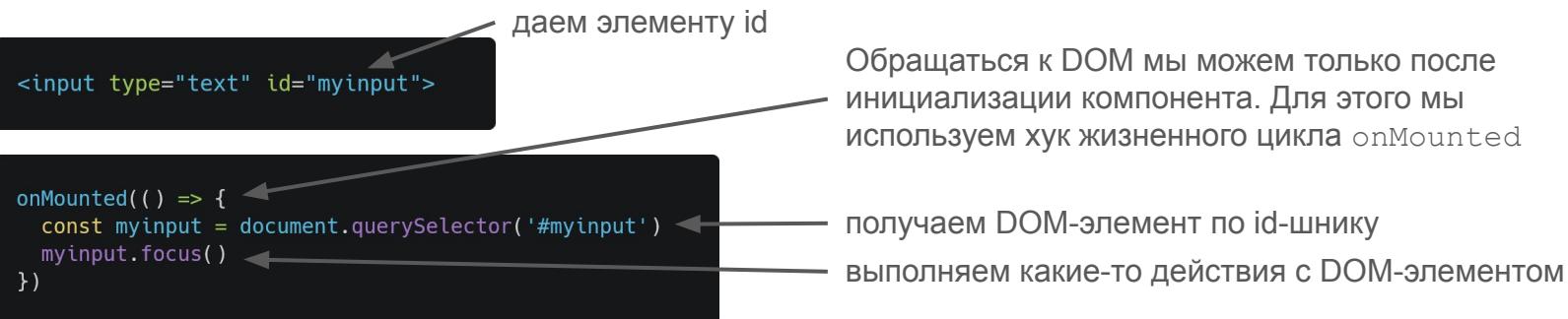
Мы отслеживаем свойство `todoId` и подгружаем через API нужные данные, когда это свойство поменяется.

Template Refs

Getting DOM element

При работе иногда возникает необходимость получения доступа к определенному DOM элементу (напр. чтобы подключить внешнюю библиотеку или вызвать метод DOM-api).

Для этого можно, конечно, просто дать элементу определенный id и получить его с помощью `document.querySelector` или `document.getElementById`



Template Refs

Но Vue предоставляет более удобный способ получения DOM объектов - **Template Refs**.

Мы можем привязать обычное реактивное `ref` свойство к нужному элементу в шаблоне с помощью свойства `ref`:

```
<script setup>
import { ref, onMounted } from 'vue'

// declare a ref to hold the element reference
// the name must match template ref value
const myinput = ref(null)

onMounted(() => {
  myinput.value.focus()
})
</script>

<template>
  <input ref="myinput" />
</template>
```

создаем реактивное свойство с начальным значением null

в шаблоне нужному элементу задаем параметр `ref`, куда передаем имя созданной `ref`-переменной

после инициализации компонента Vue сделает так, что в `myinput.value` будет находиться ссылка на нужный нам DOM элемент

Function Refs

Вместо строкового ключа в атрибут `ref` можно передать функцию, которая будет вызываться при каждом обновлении компонента и обеспечит полную гибкость в выборе места хранения ссылки на элемент. Функция получает ссылку на элемент в качестве первого аргумента:

внутри такой функции мы можем сами решать,
куда сохранять ссылку на DOM-элемент



```
<input :ref="(el) => { /* assign el to a property or ref */ }">
```

Например:

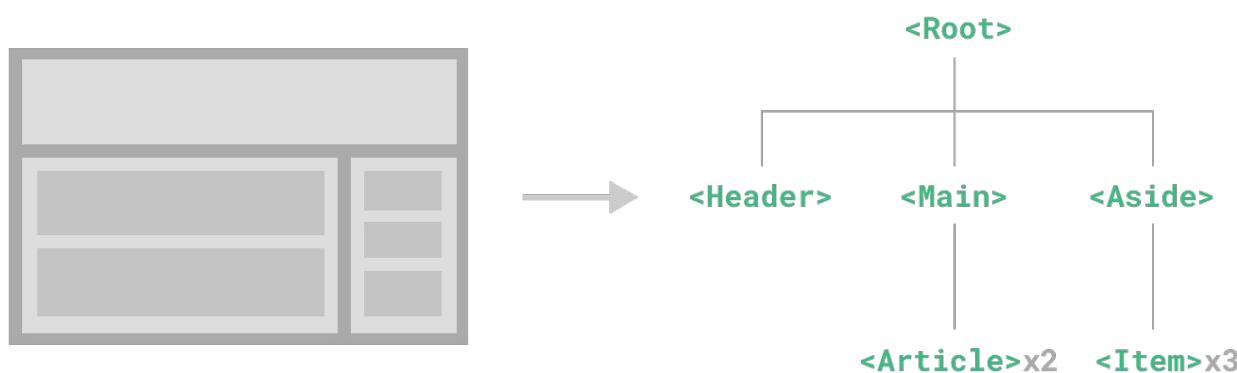
```
<input :ref="(el) => { myinput = el }">
```

(аналогично стандартному функционалу, когда
мы задаем `ref` строкой “`myinput`”)

Components Basics

Components

Компоненты позволяют нам разделить пользовательский интерфейс на независимые и многократно используемые части и думать о каждой части отдельно. Обычно приложение организовано в виде дерева вложенных друг в друга компонентов:



Defining a Component

При использовании сборщика мы обычно определяем каждый компонент Vue в отдельном файле с расширением `.vue` - так называемый **Single-File Component** (сокращенно **SFC**):

`Clicker.vue:`

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

const count = ref(0)
</script>

<template>
  <button @click="count++">You clicked me {{ count }} times.</button>
</template>
```

Определим компонент кликера с собственной логикой и шаблоном. Логика определяется в разделе `script`, а шаблон - в `template`.

`Clicker.js:`

```
import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)
    return { count }
  },
  template: `
    <button @click="count++">
      You clicked me {{ count }} times.
    </button>
  `
}
```

Без сборщика компоненты также возможно определить, но обычно этой возможностью пользуются редко (т.к. всегда используется сборщик).

Вот пример такого же компонента кликера без использования сборщика:
(В таком случае `vue` файлы уже не используются, так как браузер не знает, как с ними работать.
Компонент определяется в обычном js файле.)

Using a Component

Чтобы использовать дочерний компонент, нам нужно импортировать его в родительский компонент.

Если мы поместим наш компонент кликера Clicker.vue в папку components (созданную по-умолчанию генератором Vue проектов), то его подключение и использование в корневом компоненте App.vue будет выглядеть следующим образом:

App.vue:

```
<script setup>
import Clicker from './components/Clicker.vue'; ← Импортируем созданный компонент Clicker
</script>

<template>
  <h1>Here is a child component!</h1>
  <Clicker /> ← Далее можно использовать
</template>                                         компонент Clicker в
                                                    шаблоне как <Clicker />
```

В результате кликер появится на странице:

Here is a child component!

You clicked me 5 times.

Using a Component

```
<script setup>
import Clicker from './components/Clicker.vue';
</script>

<template>
  <h1>Here is a child component!</h1>
  <Clicker />
  <Clicker />
  <Clicker />
</template>
```

Компоненты могут быть переиспользованы много раз

Here is a child component!

You clicked me 6 times. You clicked me 5 times. You clicked me 0 times.

Props

При создании компонентов мы можем столкнуться с необходимостью передать в них какую-то информацию. Для этого используются **props**-ы.

Они задаются с помощью макроса `defineProps`. В него передается массив названий пропсов. В ответ `defineProps` возвращает объект, который содержит переданные в компонент пропсы от родителя.

Добавим в `Clicker` пропс, отвечающий за количество очков, прибавляемое за клик:

`Clicker.vue`:

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

const count = ref(0)

const props = defineProps(['amount'])
console.log('clicker amount: ${props.amount}')
</script>

<template>
  <button @click="count += amount">You clicked me {{ count }} times.</button>
</template>
```

`App.vue`:

```
<script setup>
import Clicker from './components/Clicker.vue';
</script>

<template>
  <h1>Here is a child component!</h1>
  <Clicker :amount="3" />
</template>
```

передаем значение пропса из родительского компонента

Props

Объявив пропс, мы можем пользоваться компонентом из родительского компонента, передавая различные значения пропса.

```
<script setup>
import Clicker from './components/Clicker.vue';
</script>

<template>
  <h1>Here is a child component!</h1>
  <Clicker :amount="3" />
  <Clicker :amount="6" />
  <Clicker :amount="10" />
</template>
```

В родительском компоненте можно создавать экземпляры дочернего компонента с различными значениями пропсов.

Зачастую при использовании сторонних библиотек мы просто можем использовать чужие компоненты и передавать в них нужные значения пропсов (которые предусмотрел разработчик библиотеки), не вникая в детали реализации этих компонентов.

Listening to Events

Фактически пропсы позволяют нам задать для своего компонента кастомные атрибуты, в которые родитель передает нужные значения.

Но что если мы хотим добавить в свой компонент кастомные события, на которые родительский компонент сможет подписаться аналогично тому, как можно сделать, например, `onClick` со встроенными html элементами?

Сделать это можно с помощью макроса `defineEmits`. Он принимает список названий событий и возвращает функцию `emit`, которую можно использовать для генерации события для родительского компонента.

Добавим в компонент `Clicker` кастомное событие “`userClicked`”, которое возникает при клике, и на которое родитель может подписаться, получив в качестве аргумента текущее количество очков:

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

const count = ref(0)

const props = defineProps(['amount'])
console.log('clicker amount: ${props.amount}')

const emit = defineEmits(['userClicked'])
const onClick = () => {
  count.value += props.amount
  emit('userClicked', count.value)
}
</script>

<template>
  <button @click="onClick">You clicked me {{ count }} times.</button>
</template>
```

Объявляем кастомное событие “`userClicked`” и вызываем его в нужное время с помощью функции `emit`, передав название события и определенное значение, которое мы хотим передать в родительский компонент (в нашем случае это количество очков)

В родительском компоненте мы можем подписаться на событие с помощью `v-on (@)` и получить значение очков от дочернего компонента:

```
<script setup>
import Clicker from './components/Clicker.vue';
</script>

<template>
  <h1>Here is a child component!</h1>
  <Clicker :amount="3" @userClicked="(score) => console.log(`${score} clicked`)" />
</template>
```

Content Distribution with Slots

Помимо каких-то простых значений может возникнуть необходимость передать внутрь компонента целые куски шаблона.

Делается это с помощью `<slot />`. Контент передается родительским компонентом между открывающим и закрывающим тегом дочернего компонента. То есть наш компонент как бы перестает быть самозакрывающимся и требует передачи в него данных между открывающим и закрывающим тегом.

Добавим в наш `Clicker` возможность кастомизировать кнопку, по которой осуществляется клик (так чтобы она передавалась из родительского компонента между открывающим и закрывающим тегом `<Clicker>`):

`Clicker.vue:`

```
...
<template>
  <div>You clicked me {{ count }} times.</div>
  <div @click="onClick">
    <slot />
  </div>
</template>
```

`App.vue:`

```
...
<template>
  <h1>Here is a child component!</h1>
  <Clicker :amount="3" @userClicked="(score) => console.log(` ${score} clicked`)">
    <button>Custom button!</button>
  </Clicker>
</template>
```

контент (между открывающим и закрывающим тегом в шаблоне родителя) будет вставляться тут



Теперь из родителя мы передаем кнопку для клика между открывающим и закрывающим тегом `<Clicker>`

Темы для докладов

- 1) vue reactivity, reactivity utils

<https://vuejs.org/api/reactivity-advanced.html>

<https://vuejs.org/api/reactivity-utilities.html>

Спасибо за внимание!