



Vue.js e ES6

Jean
Raphael
Vinicius



Sobre o VueJS

- ❑ Um framework progressivo
 - Não monolítico, adoção incremental
- ❑ Comparado aos outros frameworks:
 - vs React: tudo em JS (JSX e CSS)
 - vs Angular: Typescript e estrutura rígida
- ❑ Curva de aprendizado: templates e CSS



A Instância Vue

- Criando uma instância Vue
 - el: elemento raiz
 - data: objeto com as propriedades para o sistema de reatividade do Vue

```
<div id="app">
  {{ message }}
</div>
```

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

» Sintaxe de Template

- Interpolação
 - Texto: sintaxe “Mustache”
 - Raw HTML: diretiva v-html

```
<span>Message: {{ msg }}</span>  
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

» Sintaxe de Template

□ Interpolação

- Atributos: diretiva v-bind
- Usando expressões JavaScript

```
<div v-bind:id="dynamicId"></div>

{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}
```

» Sintaxe de Template

□ Shorthands

- v-bind Shorthand
- v-on Shorthand

```
<!-- full syntax -->  
<a v-bind:href="url"> ... </a>  
  
<!-- shorthand -->  
<a :href="url"> ... </a>
```

```
<!-- full syntax -->  
<a v-on:click="doSomething"> ... </a>  
  
<!-- shorthand -->  
<a @click="doSomething"> ... </a>
```

» Sintaxe de Template

- Criando uma instância Vue
 - el: elemento raiz
 - data: objeto com as propriedades para o sistema de reatividade do Vue

```
<div id="app">
  {{ message }}
</div>
```

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

»» Condicionais e iterações

- ❑ `v-if` e `v-show`
- ❑ `v-if`: elemento só é renderizado de acordo com a condição
- ❑ `v-show`: elemento é renderizado sempre, mostrando ou não conforme a condição (CSS toggling)

» Condicionais e iterações

□ v-for

```
<div id="app-4">
  <ol>
    <li v-for="todo in todos">
      {{ todo.text }}
    </li>
  </ol>
</div>
```

```
var app4 = new Vue({
  el: '#app-4',
  data: {
    todos: [
      { text: 'Learn JavaScript' },
      { text: 'Learn Vue' },
      { text: 'Build something awesome' }
    ]
  }
})
```

» Condicionais e iterações

- v-if, v-else, v-else-if

```
<div id="app-3">  
  <span v-if="seen">Now you see me</span>  
</div>
```

```
var app3 = new Vue({  
  el: '#app-3',  
  data: {  
    seen: true  
  }  
})
```

v-for

- ❑ Executa um loop em um Iterável (Array, Set)
- ❑ Quebra o iterável em dois valores: Valor atual e Índice
- ❑ Podem ser usados em templates

```
//[1,2,3,4,5] pode ser um variável do data()  
<p v-for="(valorAtual, indice) in [1,2,3,4,5]">  
  {{valorAtual}}  
</p>
```

» Métodos

- ❑ Funções do javascript
- ❑ Executam somente ao serem chamadas

```
<script>
  new Vue({
    methods: {
      adicionaProduto(parametro1, parametro2) {
        this.propriedade = parametro1 + parametro2
      }
    }
  })
</script>
```

» Eventos

- ❑ Usado para comunicação entre componentes
- ❑ Um componente pode escutar:
@nomeDoEvento="metodo(\$event)"
- ❑ E pode emitir: *\$emit("nomeDoEvento", dados)*

```
// método no componente filho
methods: {
  enviarDados() {
    //emite um evento nomeDoEvento com o array dados
    this.$emit("nomeDoEvento", {
      dados: [1,2,3]
    });
  }
}
```

```
//componente pai
<componente-filho @nomeDoEvento="recebeDados($event)" />
// a função recebeDados pertence ao componente pai
// e recebe os dados através da variável especial $event
```

V-model

□ Aplica “two-way binding”

- Faz com que o componente ou elemento receba um valor (:value)
- Adiciona um evento que emite informação quando o valor mudar (@input)



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

//é o mesmo que

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

» Computed e watchers

□ Computed

- usado para “computar” instantaneamente certas variáveis
- deixa tudo mais reativo
- valores

```
<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
```

```
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // a computed getter
    reversedMessage: function () {
      // `this` points to the vm instance
      return this.message.split('').reverse().join('')
    }
  }
})
```

» Computed e watchers

□ Watchers

- Observa alterações de uma determinada propriedade (em data)

```
var vm = new Vue({  
  el: '#example',  
  data: {  
    message: 'Hello',  
    price: 100  
  },  
  watch: {  
    price: function (val) {  
      this.message = 'price changed to ' + val  
    }  
  }  
})
```




COMPONENTES

- ❑ São instâncias reutilizáveis de código
- ❑ Podem ser incorporadas na instância raiz
- ❑ Podem ser incorporadas em outros componentes





COMPONENTES

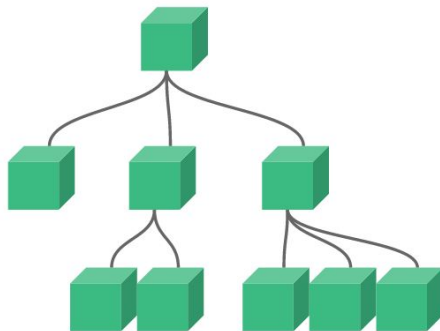
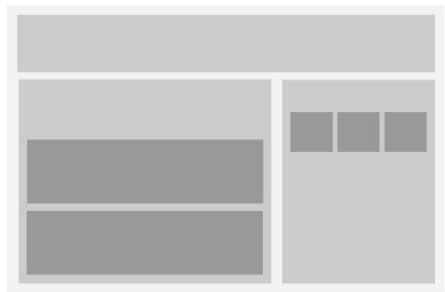
Os *componentes* possuem as mesmas opções que o componente raiz, como:

- **data**
- **computed**
- **watch**
- **methods**
- **gatilhos do ciclo de vida**

```
Vue.Component('nome-do-component', {  
  template: 'Aqui vai todo meu html',  
  
  data() {  
    //...  
  },  
  computed: {  
    //...  
  },  
  watch: {  
    //...  
  },  
  methods: {  
    //...  
  },  
  created() {  
  }  
})
```

COMPONENTES

Ao criar componentes você está facilitando a organização e manutenção do projeto, não apenas o reuso.





COMPONENTES

Um componente pode ser registrado de duas maneiras:

Em um projeto com **Webpack**

```
<template><!--HTML--></template>
<script>
  export default {
    /* methods, data, computed... */
  }
</script>
<style>/* CSS */</style>
```

Em uma página **HTML** comum

```
<script>
  Vue.component("NomeDoComponente", {
    /* methods, computed, data... */
  });
</script>
```



COMPONENTES



```
// instância raiz  
new Vue({  
  el: '#exemplo'  
})
```



```
<div id="exemplo">  
  <meu-componente-personalizado></meu-componente-personalizado>  
  <meu-componente-personalizado></meu-componente-personalizado>  
  <meu-componente-personalizado></meu-componente-personalizado>  
</div>
```

Resultado:

A custom component!

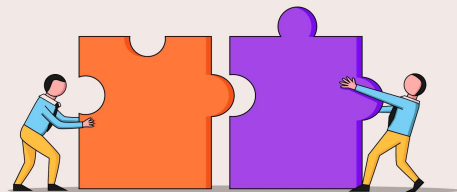
A custom component!

A custom component!



COMUNICAÇÃO ENTRE COMPONENTES

Quando trabalhamos com componentes é comum a necessidade de passar dados de entre eles.



» PROPRIEDADES (props)

PROPS

São atributos **vuejs** personalizados que são registrados em componentes e permitem passar dados de um componente pai para um componente filho.

Múltiplos props

```
props: ['nome', 'telefone', 'email']
```

PROPS

Tipos:

Strings,
Numbers,
Boolean,
Array,
Objects,
Function,
Promise

```
props: {  
  nome: String,  
  telefone: Number,  
  endereco: Object,  
  callback: Function,  
  ativo: Boolean  
}
```


» PROPS - Required

Nem todas props são criadas iguais. Além do tipo, algumas delas podem ser necessárias para o correto funcionamento do componente

```
props: {  
  nome: {  
    type: String,  
    required: true  
  }  
}
```



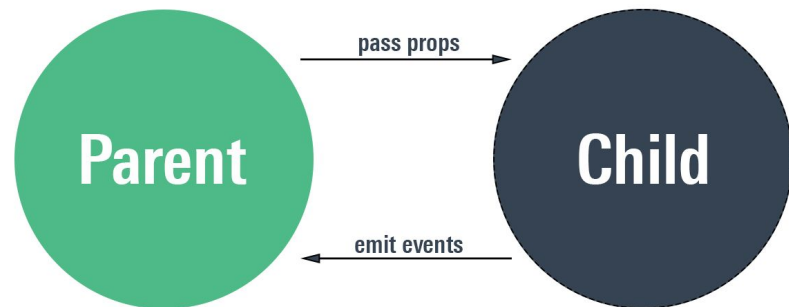
PROPS - Default

Muitas vezes precisamos definir um valor padrão para a propriedade

```
props: {  
  nome: {  
    type: String,  
    required: true,  
    default: 'Insira o nome'  
  }  
}
```


» Eventos

PROPS ENTRAM, EVENTOS SAEM



Eventos

O componente filho emite um evento para o componente pai com os valores

```
  
this.$emit('meu-evento', { id: 1, item: 'Banana', valor: 2 })
```

Eventos

O componente pai escuta o evento
do componente filho

```
  
<componente-pai>  
  <componente-filho @meu-evento="executaEssaFuncao"></componente-filho>  
</componente-pai>
```

Eventos

```
methods: {  
  executaEssaFuncao(value) {  
    //...  
  }  
}
```



INICIANDO UM PROJETO

- ☐ `npm install -g @vue/cli`
- ☐ `vue create nome-do-projeto -b`
- ☐ `npm run serve`



CRIANDO UM PROJETO

- ☐ Manually select features
- ☐ Deixar marcados:
 - Babel
 - Linter / Formatter
- ☐ ESLint with error prevention only
- ☐ Lint on save
- ☐ In dedicated json files



Módulos ES6

export, import

Aumenta reusabilidade

Diminui complexidade

```
export Calculator {  
  soma (a, b) {  
    return a + b  
  },  
  subtracao (a, b) {  
    return a - b  
  }  
}
```

```
import Calculator from Calculator  
  
soma(1,2)  
subtracao(10,1)
```

```
import { soma } from Calculator  
  
soma(1,2)
```

Ciclo de Vida

1. Creation: `beforeCreate`, `created`

- Podemos carregar dados necessários para inicialização

3. Updating: `beforeUpdate`, `updated`

- Útil para debugar cada atualização

2. Mounting: `beforeMount`, `mounted`

- Componentes estão inicializados, podemos modificá-los

4. Destruction: `beforeDestroy`, `destroyed`

- Podemos limpar resquícios e ativar gatilhos



Vue Router

- ❑ Componente oficial para gerenciamento de rotas
- ❑ Permite passar parâmetros através da URL
- ❑ Efeitos de transição
- ❑ Automaticamente coloca classes CSS para links ativos



Instalando o Vue Router

- ☐ npm install vue-router
- ☐ modificar o main.js
- ☐ Vue.use()
- ☐ new VueRouter()
- ☐ routes
- ☐ passar router no new Vue

```
import Vue from 'vue'
import App from './App.vue'
import VueRouter from "vue-router"
import HomePage from "./views/HomePage"

Vue.use(VueRouter);

const router = new VueRouter({
  routes: [
    {
      path: "/",
      component: HomePage
    }
  ]
});

new Vue({
  render: h => h(App),
  router
}).$mount('#app')
```



Criando rotas

- `params = :id`
`url.com/produto/1`
`this.$route.params.id`
- `query`
`url.com/produto?id=1`
`this.$route.query.id`
- `meta = qualquer info`
`this.$route.meta.title`

```
import Contato from "@views/Contato.vue"
import Produto from "@views/Produto.vue"

const router = new VueRouter({
  routes: [
    {
      path: "/contato",
      component: Contato,
      meta: {
        titulo: "Entre em contato"
      }
    },
    {
      path: "/produto/:id",
      component: Produto
    },
  ]
});
```



Acessando o router

- Pode ser acessado em qualquer parte do componente

```
<template>
  <div>
    {{$route.id}}
  </div>
</template>

<script>
export default {
  created() {
    console.log(this.$route)
  }
}
</script>
```



Componentes do Vue Router



```
<router-link :to="data_ou_computed">Home</router-link>  
<router-link to="/produtos">Produtos</router-link>
```



```
<router-view>  
  /* Componente será renderizado aqui! */  
</router-view>
```



Exercício

<https://github.com/vncscoelho/treinamento-vue>

- ❑ Criar um novo projeto com Vue Router
- ❑ Deve IMPORTAR o **assets/database.json** contendo os produtos
- ❑ Deverá possuir no mínimo dois componentes:
 - Lista de produtos
 - Produto
- ❑ Ao acessar a rota **"/produto/id_do_produto"** deverá ser possível visualizar as infos do produto e um botão **"Comprar"**
- ❑ Deverá possuir um contador no topo da página que incremente a cada produto comprado, como um carrinho em um e-commerce



Guias de Estilo

- Serve também como um banco de soluções para problemas rotineiros
- Acelera a participação de novos colaboradores



```
<!-- preferir a abreviação -->  
<a @click="salvarEmpresa"> ... </a>  
  
<!-- do que a sintaxe completa -->  
<a v-on:click="salvarEmpresa"> ... </a>
```



```
<div v-cloak>  
  Empresa: {{ empresa }}  
</div>  
  
<style>  
  [v-cloak] {  
    display: none;  
  }  
</style>
```



Guias de Estilo

FORTEMENTE RECOMENDADOS

Nomes de componente devem ser multi-palavras:

Para evitar conflitos com tags HTML existentes ou futuras



```
Vue.component('todo-item', {  
  // ...  
})
```



Guias de Estilo

FORTEMENTE RECOMENDADOS

“v-key” deve acompanhar um elemento no loop “v-for”

Para manter consistência do código e permitir animações (como transições)



```
<div  
  v-for="item in lista" :key="item.id"  
>  
  {{ item.conteudo }}  
</div>
```



Guias de Estilo

FORTEMENTE RECOMENDADOS

“v-if” não deve ser usado no mesmo elemento com “v-for”

Para eficiência do código e otimizar a renderização

```
<ul v-if="mostrar">
  <li
    v-for="item in lista" :key="item.id"
  >
    {{ item.conteudo }}
  </li>
</ul>
```



Guias de Estilo

FORTEMENTE RECOMENDADOS

“v-if” não deve ser usado no mesmo elemento com “v-for”

Para eficiência do código e renderização

```
<ul v-if="mostrar">
  <li
    v-for="item in lista" :key="item.id"
  >
    {{ item.conteudo }}
  </li>
</ul>
```



Guias de Estilo

FORTEMENTE RECOMENDADOS

Detalhar o máximo as “props”

Para melhorar a legibilidade do código e permitir que no futuro sejam escritos testes automatizados

```
props: {  
  status: {  
    type: String,  
    required: true,  
  }  
}
```



ECMAScript 6

- ❑ A versão mais atual do Javascript a ser amplamente suportada
- ❑ Facilita a construção de aplicações complexas
- ❑ Resolve problemas antigos do JS
- ❑ Necessita ser transpilado (BABEL) para garantir a compatibilidade com navegadores antigos (IE 11 e abaixo)



ES6 na Prática

let, const

Ao contrário de var, possuem escopo de bloco.

Ganho em legibilidade e evita bugs e problemas com variáveis de mesmo nome



```
// let é mutável  
let nome = "Aqui irá um nome"  
nome = "João" // ok  
  
// const é imutável  
const nome = "João"  
nome = "Pedro" // erro
```




ES6 na Prática

A mutação permite trabalhar com o mesmo formato do objeto, e alterar os valores dentro do mesmo.

```
const pessoa = { nome: "João" }  
pessoa.nome = "Maria"  
  
console.log(pessoa) //Maria
```



ES6 na Prática

Operações com vetores: MAP

O método `map()` cria um novo array preenchido com os resultados da chamada de uma função fornecida em todos os elementos do array de chamada.



```
//MAP
const arr = [1,2,3,4,5,6]

const newArr = arr.map(function(item) {
  return item * 2;
})

console.log(newArr) //[2,4,6,8,10,12]
```



ES6 na Prática

Operações com vetores: REDUCE

O método `reduce()`, basicamente é uma forma de consumir o vetor e transformá-lo em uma única variável

```
//REDUCE
const arr = [1,2,3,4,5,6]

const result = arr.reduce(function(total, next) {
  return item + next
})

console.log(result) //21
```



ES6 na Prática

Operações com vetores: FILTER

O método `filter()`, cria um novo vetor com os elementos que passam na condição implementada dentro da função



```
//FILTER
const arr = [1,2,3,4,5,6]

const filter = arr.filter(function(item) {
  return item % 2 === 0;
})

console.log(filter) //[2,4,6]
```



ES6 na Prática

Operações com vetores: FIND

O método `find()`, é basicamente utilizado para encontrar ou verificar informações dentro do array

```
//FIND
const arr = [1,2,3,4,5,6]

const find = arr.find(function(item) {
  return item === 4;
})

console.log(find) //4
```



ES6 na Prática

Arrow Functions

Código menos verboso

O "this" é léxico, guarda o valor referente ao seu bloco

```
salvar: function () {  
    var self = this;  
    swal().then(function () {  
        this.nome = "Joao"  
    })  
}  
  
/*  
    Com arrow function,  
    o "this" será sempre referente  
    ao contexto original  
*/  
  
salvar: function () {  
    swal().then(() => {  
        this.nome = "Joao"  
    })  
}
```



ES6 na Prática

Tipos de arrow function

Conciso: **return** é **implícito**

Bloco: **return** é **explícito**,
encapsulado por { }



```
var vezesDez = parametro => parametro * 10;  
// conciso
```

```
var soma = (parametro_1, parametro_2) => parametro_1 + parametro_2;  
// bloco
```

» ES6 na Prática

Desestruturação

A desestruturação é uma expressão JS que facilita extração de valores de um determinado objeto

```
//DESESTRUTURAÇÃO
const usuario = {
  nome: 'Diego',
  idade: 23,
  endereco: {
    cidade: 'Santa Maria',
    estado: 'RS'
  },
}

const { nome, idade, endereco:{cidade} } = usuario

console.log(nome) //Diego
console.log(idade) //23
console.log(cidade) //Santa Maria
```




ES6 na Prática

Spread

Permite que um objeto iterável seja expandido em outros locais



```
let array = [1,2,3,4,5]

let novoArray = [...array, 6, 7] // [1,2,3,4,5,6,7]

let objeto = { z: 1, x: 2, y: 3}

let novoObjeto = { a: 4, b: 5, ...objeto} // { a: 4, b: 5, z: 1, x: 2, y: 3}

const funcao = (a, b) => a

funcao({...objeto}) // retorna 4
```



ES6 na Prática

Spread

Spread permite criar clones de outros objetos ou arrays rapidamente

```
let objeto = {
  id: 1,
  nome: "Abajur",
  material: "Plástico"
};

// cópia da variável na memória
let copia_objeto = objeto;

// cria um novo objeto com
// as mesmas propriedades
let copia_objeto_spread = {
  ...objeto
};

// copia_objeto_spread não vai mudar
objeto.nome = "Luminária"
```

» ES6 na Prática

Definições de método

```
var obj = {  
  foo() {},  
  bar() {}  
};
```

```
var obj = {  
  foo: function() {},  
  bar: function() {}  
};
```



ES6 na Prática

Promises

- Um objeto especial do Javascript que pode ou não produzir um resultado no futuro
- Executa imediatamente a lógica interna
- Por padrão, é assíncrona
- Sintaxe: `new Promise(resolved, reject)`

```
new Promise(function (resolve, reject) {  
  //executa imediatamente este código  
  if (1 + 1 === 2) {  
    resolve("SUCESSO!")  
  }  
  
  reject("FALHOU!")  
})
```

» ES6 na Prática

Promises

- Pode executar lógica adicional ao encadear com as funções **then()**, **catch()** e **finally()**

```
new Promise(function (resolve, reject) {  
  //executa imediatamente este código  
  if (1 + 1 === 2) {  
    resolve("SUCESSO!")  
  }  
  reject("FALHOU!")  
}).then(resultado => {  
  // cai aqui se resolvida  
  console.log(resultado)  
}).catch(erro => {  
  // cai aqui se rejeitada  
  console.error(erro)  
}).finally(() => {  
  // o que estiver aqui será executado  
  // independente se obteve sucesso ou não  
})
```



ES6 na Prática

Fetch()

- Método global para fazer requisições HTTP (GET, POST...)
- Retorna uma Promise
- Pode receber configurações de:
 - Método HTTP
 - Headers HTTP

```
fetch("http://url")  
  // fetch retorna uma Promise  
  .then(resposta => resposta.json())  
  //Essa Promise precisa ser completada para  
  //obter o corpo/payload da resposta  
  .then(payload => console.log(payload))
```

» ES6 na Prática

async, await

Trabalhar com promises mais facilmente

async determina que uma função sempre retorne uma promise

await determina que a execução aguardará que a promise seja resolvida

```
let pessoas_na_empresa = [];  
const buscaPessoasDaEmpresa = (nome_empresa) => {  
  fetch('/empresas/' + nome_empresa)  
    .then(resposta => resposta.json())  
    .then(empresa => {  
      fetch('/pessoas/' + empresa.id)  
        .then(resposta => resposta.json())  
        .then(pessoas => pessoas_na_empresa = pessoas)  
    })  
}
```

» ES6 na Prática

async, await

- Facilita a leitura do código
- Permite a execução síncrona

```
let pessoas_na_empresa = [];  
const buscaPessoasDaEmpresa = async (nome_empresa) => {  
  const empresa = await fetch('/empresas/' + nome_empresa).then(resposta => resposta.json())  
  pessoas_na_empresa = await fetch('/pessoas/' + empresa.id).then(resposta => resposta.json())  
}
```