

Web Frameworks (Vue.js)

Interface Pessoa-Máquina - 25/26 - LEI / UM

Hugo Pacheco

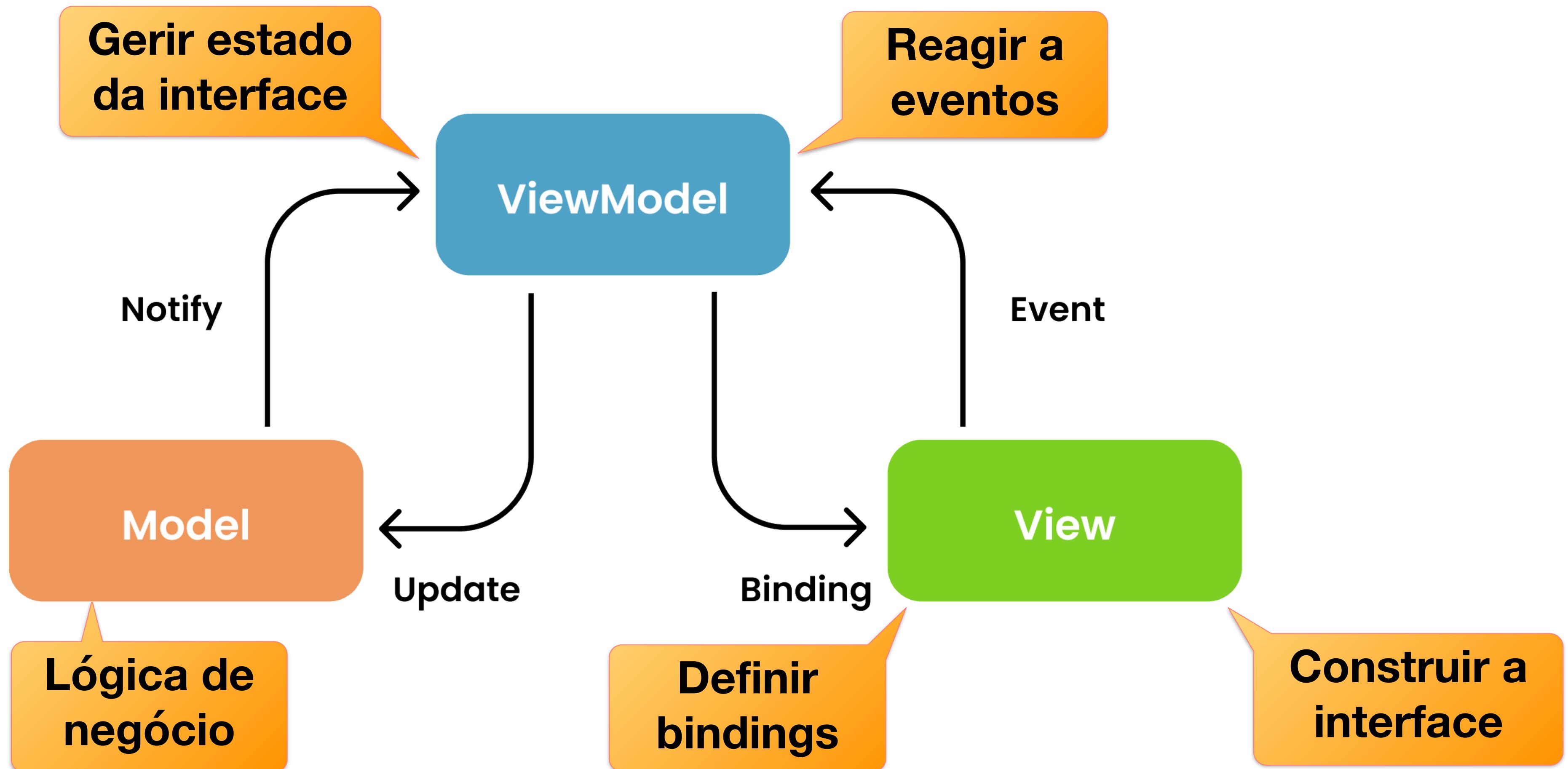
hpacheco@di.uminho.pt

Vue.js

- **Framework JavaScript** progressiva e versátil, com curva de aprendizagem baixa
- **Frontend framework**, visa criar páginas HTML com **interatividade e reatividade**
 - **Single-Page Applications (SPAs)** primordialmente
 - Server-side Rendering (SSR) via bibliotecas
 - Static Site Generator (SSG) via bibliotecas
 - Aplicações nativas para desktop e mobile via bibliotecas
- Oferece...
 - Modelo de programação **declarativo** e baseado em **componentes**
 - **API intuitiva** e **boa documentação**
 - Um **bom ecossistema** de bibliotecas, ferramentas e plugins

Arquitectura

MVVM Architecture



Setup

- Instalação ([official guide](#))
 - Online: [Vue Playground](#), [CodePen](#), [Stackblitz](#), ...
 - Sem instalação:
 - Importar JS de CDN
 - Standard web development (HTML + CSS + JS) com extensões Vue
 - Localmente via npm
 - Opcionalmente utilizando o `vite` para criar e gerir projetos Vue
- Vamos utilizar Vue 3

Bootstrap (1)

- Várias formas de construir uma Single-Page Application com Vue
 - Equivalentes, apenas organização / sintaxe / APIs diferentes

1. SPA = Ficheiros HTML + CSS + JS separados

- CSS standard
- HTML + JS com extensões de sintaxe
 - 💡 Compilados pela framework para gerar código standard para a SPA
 - Pode ser deployed independentemente da framework
 - ❗ Muito do comportamento continua a ser definido em runtime

Bootstrap (1)

HTML
File

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Counter</title>
  <link rel="stylesheet" href="style.css" type="text/css">
  <script src="https://unpkg.com/vue" defer></script>
  <script src="app-counter.js" defer></script>
</head>
<body>
  <div id="reactive-app">
    <h1>Contador</h1>
    <p id="display">{{ counter }}</p>
    <button v-on:click="increment">Incrementar</button>
  </div>
</body>
</html>
```

Import Vue from CDN

Import CSS

Import JS

Identificar elemento controlado pelo Vue

JS
File

```
const app = Vue.createApp({
  data: function () {
    return {
      counter: 0
    }
  },
  methods: {
    increment() {
      this.counter++;
    }
  }
});
app.mount("#reactive-app");
```

Criar a aplicação Vue

Montar a aplicação Vue
no elemento

Bootstrap (2)

2. Single-File Component (SFC) = Um só ficheiro com extensão .vue

- Ajuda a estruturar a aplicação utilizando componentes
 - Extensível naturalmente a múltiplos componentes separados por ficheiros
 - Mais tarde vamos criar SPAs com vários componentes
 - Maior encapsulamento
 - CSS para o scope do componente, ao invés de CSS global
 - Mesma sintaxe Vue para HTML + JS

Bootstrap (2)

HTML template

JS

CSS

```
<template>
  <form>
    <label for="abutton">{{ nome }}: {{ counter }}</label>
    <button id="abutton" v-on:click.prevent="handleIncrement">Incrementar</button>
  </form>
</template>

<script>
  export default {
    props: ['nome'],
    emits: ['increment'],
    data() {
      return {
        counter: 0
      }
    },
    methods: {
      handleIncrement() {
        this.counter++;
        this.$emit('increment', this.nome);
      }
    }
  }
</script>

<style>
  form {
    display: flex;
    flex-direction: row;
    align-items: center;
    margin-top: 2rem;
    width: 100%;
  }
  label {
    margin-right: 2rem;
    width: 10rem;
    box-sizing: border-box;
  }
</style>
```

Bootstrap (3)

3. Duas formas de escrever **Vue JS**

- **Options API**
 - Código separado entre múltiplas secções
 - `data()`, `methods`, `computed`, `watch` (definem lógica do componente)
 - `props`, `emits` (controlam interação com mundo exterior)
- **Composition API**
 - Código unificado numa só secção `setup()`
 - Explicit `reactivity` using mutable references

Bootstrap (3)

Options API

```
<script>
export default {
  props: {
    contadores: {
      type: Array,
      required: true
    }
  },
  emits: ['delete'],
  data() {
    return {
      selectedCounters: []
    };
  },
  methods: {
    deleteCounters() {
      this.$emit('delete', this.selectedCounters);
      this.selectedCounters = [];
    }
  }
};
</script>
```

Composition API

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

const props = defineProps({
  contadores: {
    type: Array,
    required: true
  }
});

const emit = defineEmits(['delete']);

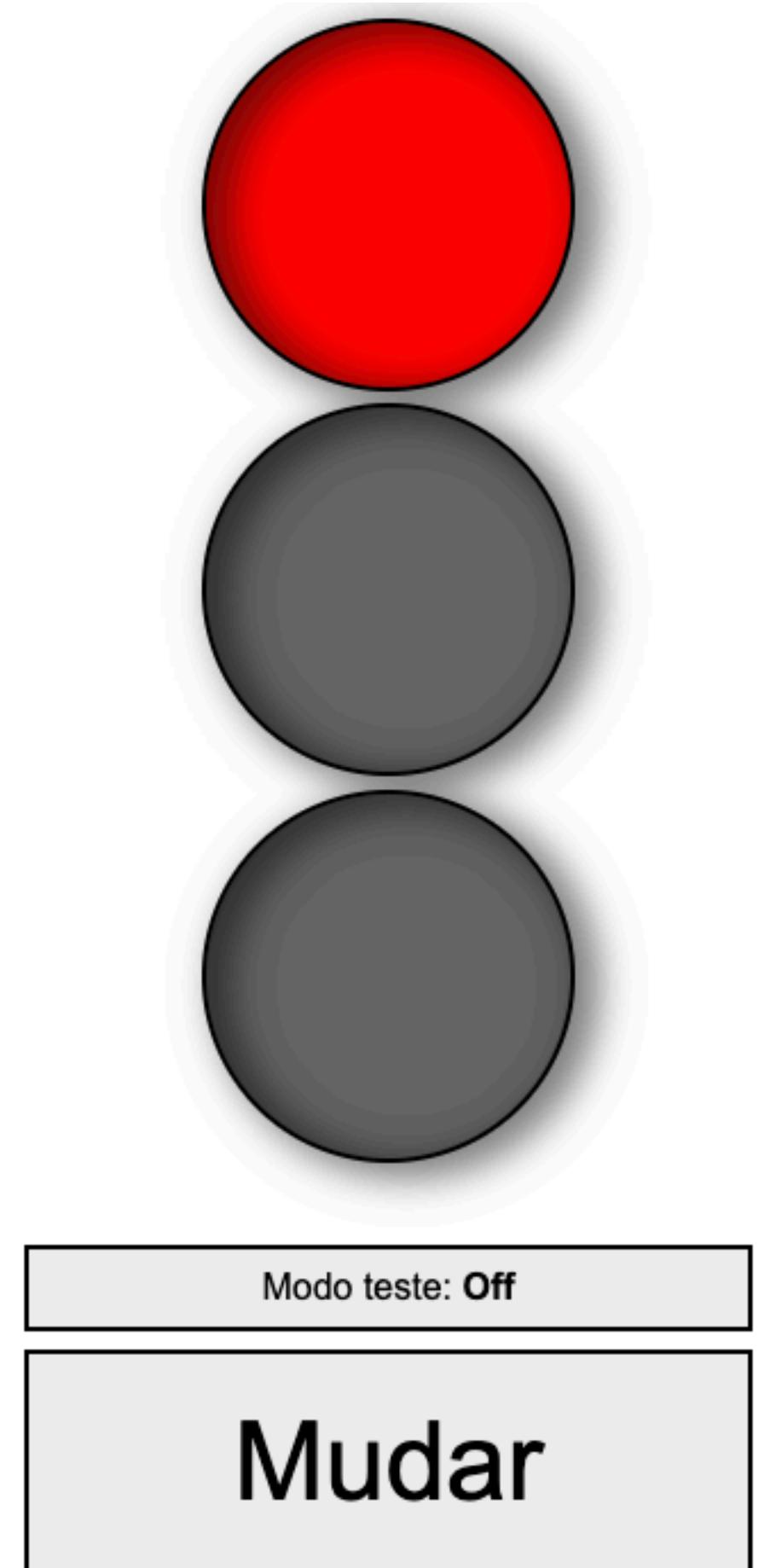
const selectedCounters = ref([]);

function deleteCounters() {
  emit('delete', selectedCounters.value);
  selectedCounters.value = [];
}

</script>
```

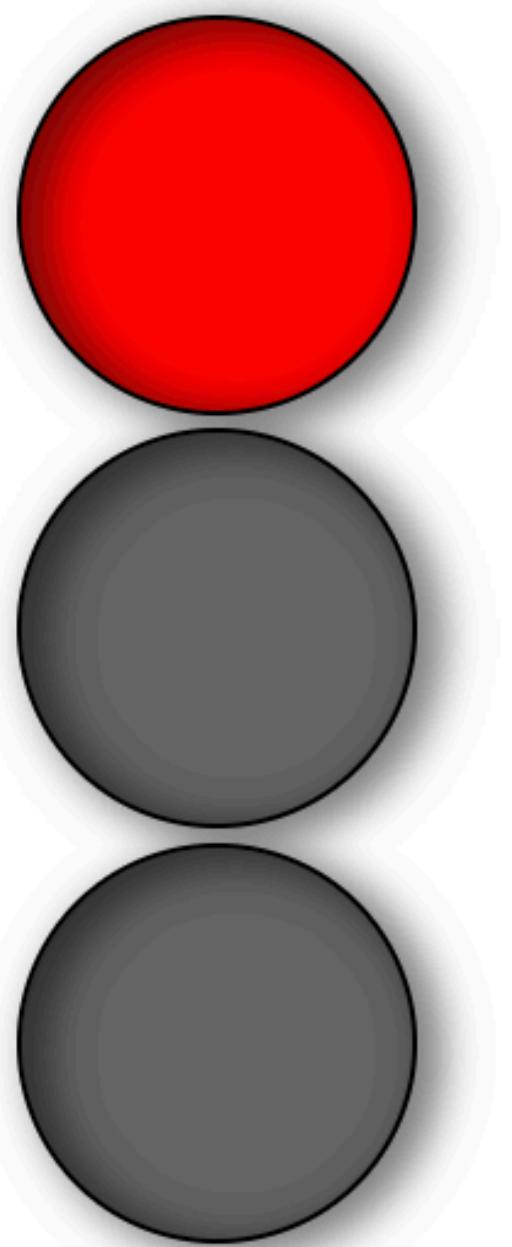
Exemplo (Semaforo JS)

- Relembrando, construímos um semáforo que:
 - Tem 3 cores (**verde**, **amarelo**, **vermelho**), apenas uma ativa
 - Botão “Mudar” avança o semáforo
 - **verde** → **amarelo** → **vermelho** → **verde**
 - Botão “Modo teste” ativa/desativa animação automática
 - Semáforo avança a cada 1 segundo
 - Botão “Mudar” inativo se “Modo teste” ativo



Exemplo (Semaforo Vue)

- Reimplementar como uma aplicação Vue:
 - SFC + Composition API
 - Bastante mais declarativo
 - HTML template + Reactive Vue data
 - Data binding: filling HTML template with Vue data
 - Data changes trigger automatic re-rendering (VDOM)
 - DOM events trigger Vue methods on data
 - Mantém a expressividade de poder integrar código JS



Modo teste:
Off

Mudar

Exemplo (Semaforo Vue)

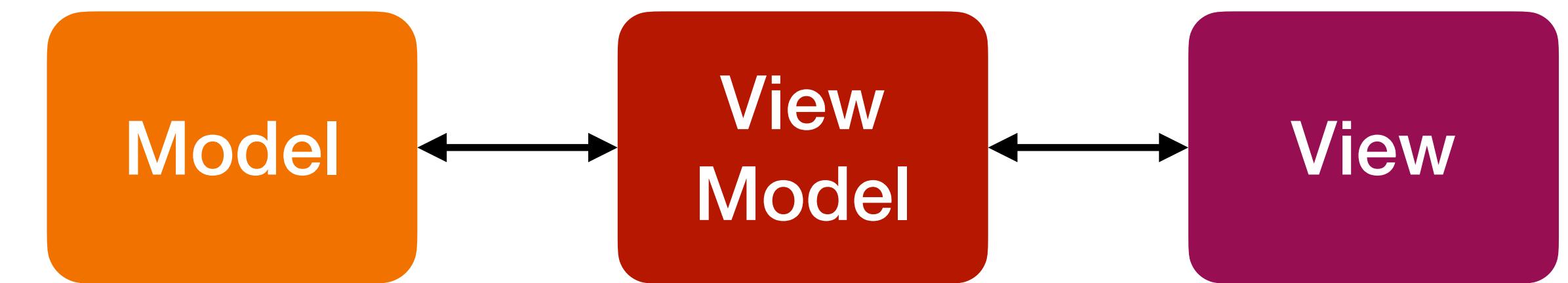
```
<template>
  <div id="app">
    <div id="red" class="lampada
vermelho" :class="{'lampada-on' : lightOn ==
2}"></div>
    <div id="yellow" class="lampada
amarelo" :class="{'lampada-on' : lightOn ==
1}"></div>
    <div id="green" class="lampada
verde" :class="{'lampada-on' : lightOn ==
0}"></div>
    <div id="botoes">
      <button @click="doAuto" :class="{'auto-
on' : auto}">Modo teste: <b
id="teste">{{strAuto}}</b></button>
      <button
@click="doMudar" :disabled="auto">Mudar</
button>
    </div>
  </div>
</template>

<style>
...
</style>
```

```
<script setup>
import { ref, watch, computed } from "vue";
const lightOn = ref(2);
const auto = ref(null);
const strAuto = computed(() => auto.value ?
"On" : "Off");

function stepLight() { lightOn.value =
(lightOn.value + 1) % 3; }
function doMudar() { stepLight(); }
function doAuto() { ... }

</script>
```



Reactive data (**Model**)

- Vue permite declarar **mutable references** em JS

```
const var = ref(v)
```

- Código JS pode ler o valor da referência

```
var.value
```

- Código JS pode modificar o valor da referência

```
variable.value = ...
```

- 💡 **Reactive programming**: a framework atualizará automaticamente, quando necessário, código que dependa do valor das referências

Computed properties (Model)

- Vue permite declarar **computed properties** em JS

```
const refC = computed(() => refA.value + refB.value)
```

- Criar novas referências que dependem de outros dados reativos

- Podem ser lidas

ref_C.value

- Não podem ser modificadas diretamente
-  Valor de uma computed property atualizado quando uma das variáveis reativas de que depende for modificada

Watchers (ViewModel)

- Vue permite declarar **watchers** em JS
- Ficam à escuta de uma referência e executam uma função JS
 - Tipicamente para executar side-effects (logging, fazer pedidos HTTP, etc)
 - Podem receber o valor novo e/ou antigo da referência

```
watch(refc, (newv, oldv) => {  
  console.log(`changed refc from ${oldv} to ${newv}`);  
})
```

- 💡 Watcher é executado sempre que o valor da referência mude

Data binding (**ViewModel**)

- Em Vue, a **View** é definida por
 - CSS standard
 - HTML templates
- **Model** é ligado à **View** através de data binding dentro dos HTML templates
 - `{{ expr }}` : binding do conteúdo de um elemento HTML a uma expressão JS

```
<span>{{ expr }}</span>
```

- 💡 **View** é atualizada sempre que **Model** mudar

Data binding (ViewModel)

- **Model** é ligado à **View** através de data binding dentro dos HTML templates
 - `v-bind:attr="expr"` : binding do valor de um atributo HTML a uma expressão JS
``
 - 💡 Atalho: `v-bind:attr` ≡ `:attr`
 - 💡 Valores de atributos HTML são sempre strings
`:type="expr ? 'text' : 'password'"`
 - 💡 Vue suporta sintaxe especial de “dicionários” para atributos `style` e `class`
`:style="{'background': expr ? 'red' : 'green'}"`
`:class="{'lampada-on': lightOn}"`

Event binding (ViewModel)

- No sentido inverso, quando a **View** muda podemos alterar o **Model**
 - Como em JS clássico, utilizando event handlers
 - 💡 Eventos suportados são os mesmos que em JS
 - Vue permite definir handlers no HTML (invocam **methods** = funções Vue)

```
<button v-on:click="handler">  
<button v-on:click="handler(expr)">  
<button v-on:click="handler(expr,$event)">
```

```
function handler(event)  
{ ... }  
function handler(v)  
{ ... }  
function handler(v,event)  
{ ... }
```

- 💡 Atalho: **v-on:event** \equiv **@event**
- 💡 Sintaxe especial **\$event** permite passar o event object nativo do JS explicitamente

Two-way binding (**ViewModel**)

- Algumas frameworks suportam bidirectional data binding entre **Model** e **View**
- Em Vue, mesmo comportamento juntando **v-bind** e **v-on**

```
<input @input="handler" :value="a"/>
```

```
function handler(event) {  
  a.value = event.target.value;  
}
```

- 💡 Atalho mais declarativo: **v-model**

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

Exemplo (Form Validation HTML + JS)

- Relembrando, podemos validar forms

1. Adicionando anotações ao HTML
(para casos comuns)

2. Utilizando a Constraint Validation API
a partir de JS (para qualquer
comportamento)

! Vue não suporta binding de pseudo-classes como `:invalid` ⇒ estado interno do DOM é controlado pelo browser

? Como podemos migrar este exemplo?

The image shows two side-by-side examples of a login form illustrating form validation.

HTML Example:

- Email Input:** A text input containing "your@email.com". To its right, an error message in red text reads "Email must have pattern string@string.string".
- Password Input:** A text input. To its right, an error message in red text reads "Password must be at least 6 characters".
- Sign in Button:** A grey button labeled "Sign in".

JS Example:

- Email Input:** A text input containing "your@email.com". To its right, an error message in red text reads "Email must have pattern string@string.string".
- Password Input:** A text input. To its right, an error message in red text reads "Password must contain an upper case letter and a number".
- Sign in Button:** A grey button labeled "Sign in".

Exemplo (Form Validation Vue)

- `v-model` para cada input field
- Validação como computed properties
- Watcher sincroniza validade com o DOM
 - Invocar `setCustomValidity` para cada input field
- 💡 `watchEffect` como `watch`, mas faz tracking de dependências automaticamente
- 💡 `onMounted` para esperar que componente esteja disponível no DOM

Email must have pattern string@string.string

Password must contain an upper case letter and a number