

Questões Teóricas

1) Explique a estrutura básica de um arquivo `.vue`. Quais são as principais seções e suas finalidades?

Um arquivo `.vue` é um *Single File Component* (SFC) com três blocos principais: `<template>` para a marcação, `<script>` para a lógica e `<style>` para os estilos. No *Vue 3*, é comum usar `<script setup>` para código mais conciso. O bloco `<style>` pode ser global ou *scoped*, e cada bloco aceita atributos como `lang="ts"` ou `lang="scss"` para tipagem/pré-processadores.

2) Defina o que são `props` no *Vue.js* e como elas são usadas para comunicação entre componentes.

Props são entradas declaradas pelo componente filho para receber dados do pai. No *Vue 3* são definidas com `defineProps(...)` ou na opção `props`; são imutáveis dentro do filho e podem ter tipo, valor padrão e validação. O pai passa valores como atributos (por exemplo, `<UserCard :user="obj">`), garantindo fluxo unidirecional e componentes reutilizáveis.

3) Explique a diferença entre `props` e eventos personalizados na comunicação entre componentes no *Vue.js*. Quando usar cada um deles?

Props levam dados do pai para o filho; eventos personalizados fazem o caminho inverso, permitindo que o filho notifique o pai sobre interações ou mudanças (`$emit/defineEmits`). Use `props` para configurar/fornecer estado ao filho e eventos quando o filho precisa comunicar ações (por exemplo, "salvou", "fechou"). Em *Vue 3*, `v-model` é um atalho que combina `prop (modelValue)` + evento (`update:modelValue`).

4) O que são `slots` no *Vue.js* e qual a sua principal utilidade? Dê um exemplo de como eles podem ser usados para personalizar componentes.

Slots são espaços reservados dentro do `template` de um componente onde o pai injeta conteúdo, tornando o componente flexível sem alterar sua estrutura. Por exemplo, um componente `<Card>` pode definir um `slot` padrão para o corpo e um `slot` nomeado "footer" para ações; o pai então fornece os trechos desejados, podendo até receber dados via `slots` com escopo (`slot props`) para personalizações avançadas.

5) Descreva o conceito de *mixins* no *Vue.js*. Quais são os benefícios e os possíveis problemas de usá-los em projetos grandes?

Mixins reúnem e reaproveitam lógica (dados, métodos, *watchers* e *hooks*) que será mesclada em múltiplos componentes. Beneficiam a DRY e padronização, mas em projetos grandes podem causar colisões de nomes, acoplamento implícito e dificultar rastreamento da origem da lógica; por isso, no *Vue 3* costuma-se preferir *composables* com a *Composition API* (funções *useX*) por oferecerem composição explícita.

6) Explique a importância de utilizar o atributo *v-bind:key* ao trabalhar com listas dinâmicas em um componente.

Usar *v-bind:key* em listas dá ao *Vue* uma identidade estável para cada item, permitindo *diffs* eficientes e preservando estado local de componentes filhos e animações ao reordenar/atualizar. Chaves únicas e estáveis evitam renderizações incorretas; evite usar o índice do *array* como *key* quando a ordem pode mudar.

7) Quais são as vantagens de organizar um projeto *Vue.js* utilizando pastas como *components/*, *views/*, *router/* e *store/*?

Organizar o projeto em *components/*, *views/*, *router/* e *store/* separa responsabilidades: *components/* guarda peças reutilizáveis, *views/* reúne páginas ligadas a rotas, *router/* centraliza a configuração de navegação e *store/* concentra o estado global (*Vuex/Pinia*). Essa estrutura melhora escalabilidade, testes, *code-splitting* e a colaboração entre times.

8) O que acontece se dois componentes diferentes utilizarem o mesmo *mixin*? Como o *Vue* trata possíveis conflitos de métodos ou dados?

Quando dois componentes usam o mesmo *mixin*, o conteúdo do *mixin* é mesclado em cada componente de forma independente. Em conflitos, o que estiver declarado no componente tem precedência sobre o *mixin*; *hooks* de ciclo de vida são encadeados (todos são chamados) e dados são mesclados com as chaves do componente sobrescrevendo as do *mixin*. Ainda assim, é recomendável documentar e evitar colisões.

9) Qual a diferença entre usar um *slot simples* (*<slot></slot>*) e um *slot nomeado* (*<slot name="..."></slot>*) em um componente *Vue*?

Um *slot simples* (*<slot>*) insere um único conteúdo padrão num ponto do componente. *Slots* nomeados (*<slot name="..."></slot>*) permitem múltiplas áreas distintas (por

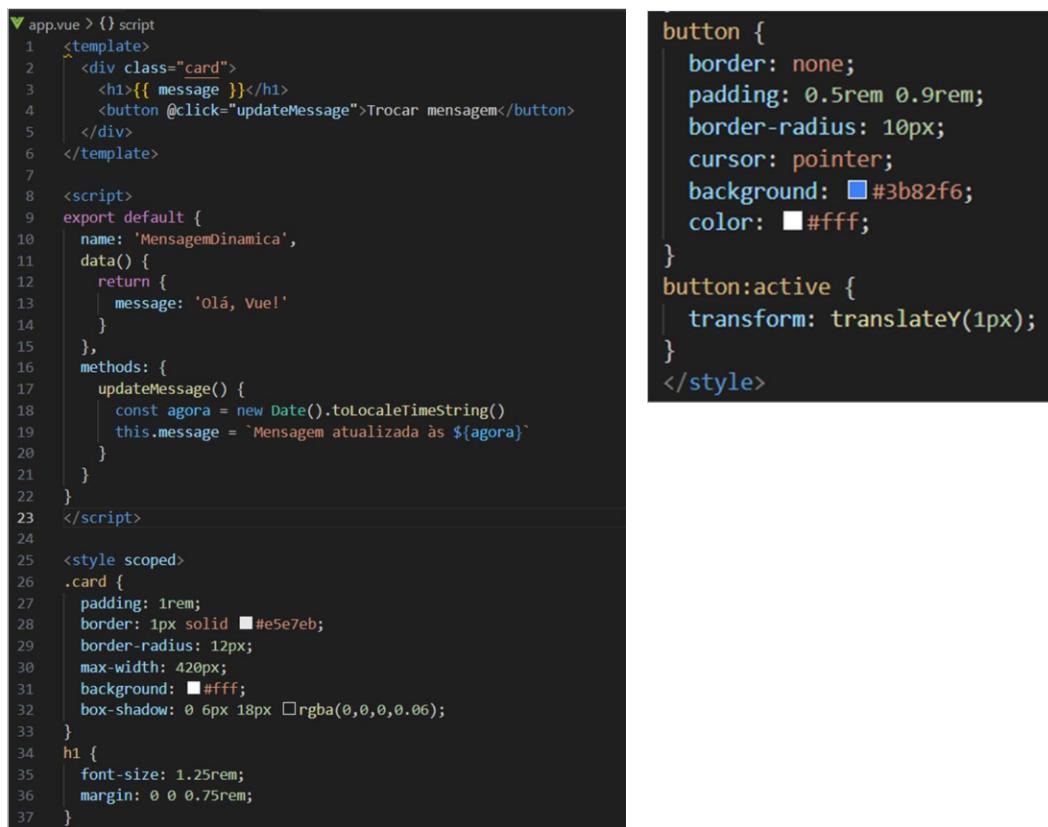
exemplo, "header", "footer"), que o pai preenche com `<template v-slot:header>` e assim por diante; isso aumenta a flexibilidade de *layout* e composição.

10) Descreva como o *Vue.js* facilita a comunicação de dados entre componentes pais e filhos. Por que isso é importante para aplicações modulares?

O *Vue* facilita o fluxo pai→filho com *props* e o fluxo filho→pai com eventos personalizados (ou *v-model*, que combina ambos); para comunicação mais profunda há *provide/inject* e, no escopo global, um *store* (*Pinia/Vuex*). Esse modelo explícito mantém módulos desacoplados, previsíveis e testáveis, essencial para aplicações grandes e evolutivas.

Questões Práticas

1. Crie um componente básico *Vue (.vue)* que exiba uma mensagem dinâmica vinda de um dado *no data()*. Adicione estilos locais na seção `<style>`.



```
▼ app.vue > {} script
1  <template>
2    <div class="card">
3      <h1>{{ message }}</h1>
4      <button @click="updateMessage">Trocar mensagem</button>
5    </div>
6  </template>
7
8  <script>
9  export default {
10    name: 'MensagemDinamica',
11    data() {
12      return {
13        message: 'Olá, Vue!'
14      }
15    },
16    methods: {
17      updateMessage() {
18        const agora = new Date().toLocaleTimeString()
19        this.message = `Mensagem atualizada às ${agora}`
20      }
21    }
22  }
23 </script>
24
25 <style scoped>
26 .card {
27   padding: 1rem;
28   border: 1px solid #e5e7eb;
29   border-radius: 12px;
30   max-width: 420px;
31   background: #fff;
32   box-shadow: 0 6px 18px rgba(0,0,0,0.06);
33 }
34 h1 {
35   font-size: 1.25rem;
36   margin: 0 0 0.75rem;
37 }
```

```
button {
  border: none;
  padding: 0.5rem 0.9rem;
  border-radius: 10px;
  cursor: pointer;
  background: #3b82f6;
  color: #fff;
}
button:active {
  transform: translateY(1px);
}
</style>
```

2. Crie um componente pai e um componente filho. O pai deve passar uma mensagem como *prop* para o filho, e o filho deve exibi-la na tela.

```
<template>
  <div class="filho">
    <h3>Componente Filho</h3>
    <input v-model="inputValor" placeholder="Digite algo..." />
    <button @click="enviarParaPai">Enviar para o Pai</button>
  </div>
</template>

<script>
export default {
  name: "Filho",
  data() {
    return {
      inputValor: ""
    }
  },
  methods: {
    enviarParaPai() {
      this.$emit("enviar-mensagem", this.inputValor)
      this.inputValor = ""
    }
  }
}
</script>
```

```
<template>
  <div class="pai">
    <h2>Componente Pai</h2>
    <Filho :mensagem="mensagemDoPai" />
  </div>
</template>

<script>
import Filho from "./Filho.vue"

export default {
  name: "Pai",
  components: { Filho },
  data() {
    return {
      mensagemDoPai: "Olá do componente Pai!"
    }
  }
}
</script>

<style scoped>
.pai {
  padding: 1rem;
  border: 2px solid #3b82f6;
  border-radius: 12px;
  background: #eff6ff;
}
```

3. Implemente um sistema onde o componente filho envie um dado para o componente pai por meio de um evento personalizado.

```
<template>
  <div class="filho">
    <p>Mensagem recebida do pai: <strong>{{ mensagem }}</strong></p>
  </div>
</template>

<script>
export default {
  name: "Filho",
  props: {
    mensagem: {
      type: String,
      required: true
    }
  }
}
</script>

<style scoped>
.filho {
  padding: 0.8rem;
  border: 1px solid #ddd;
  border-radius: 8px;
  background: #f9fafb;
}
```

```
<style scoped>
.filho {
  padding: 1rem;
  border: 1px solid #d1d5db;
  border-radius: 8px;
  background: #fefce8;
}
input {
  padding: 0.4rem;
  border: 1px solid #ccc;
  border-radius: 6px;
  margin-right: 0.5rem;
}
button {
  background: #16a34a;
  color: #fff;
  border: none;
  padding: 0.4rem 0.8rem;
  border-radius: 6px;
  cursor: pointer;
}
button:hover {
  background: #15803d;
}
</style>
```

```

<template>
  <div class="pai">
    <h2>Componente Pai</h2>
    <Filho @enviar-mensagem="receberMensagem" />
    <p v-if="mensagemRecebida">
      Mensagem recebida do filho: <strong>{{ mensagemRecebida }}</strong>
    </p>
  </div>
</template>

<script>
import Filho from "./Filho.vue"

export default {
  name: "Pai",
  components: { Filho },
  data() {
    return {
      mensagemRecebida: ""
    }
  },
  methods: {
    receberMensagem(valor) {
      this.mensagemRecebida = valor
    }
  }
}
</script>

<style scoped>
.pai {
  padding: 1.5rem;
  border: 2px solid #2563eb;
  border-radius: 12px;
  background: #eff6ff;
}
</style>

```

4. Crie um componente que utilize um slot simples para renderizar conteúdo personalizado vindo do componente pai.

Componente com slot

```

<template>
  <div class="card">
    <slot></slot>
  </div>
</template>

<script>
export default {
  name: "Card"
}
</script>

<style scoped>
.card {
  padding: 1rem;
  border: 1px solid #d1d5db;
  border-radius: 12px;
  background: #f9fafb;
  box-shadow: 0 4px 10px rgba(0,0,0,0.06);
}
</style>

```

Componente Pai

```
<template>
  <div>
    <h2>Exemplo de Slot Simples</h2>

    <Card>
      <h3>Conteúdo vindo do Pai</h3>
      <p>Esse texto foi injetado dentro do componente Card usando slot.</p>
      <button @click="acao">Clique aqui</button>
    </Card>
  </div>
</template>

<script>
import Card from "./Card.vue"

export default {
  name: "App",
  components: { Card },
  methods: {
    acao() {
      alert("Botão dentro do slot clicado!")
    }
  }
}</script>
```

5. Adapte o exercício anterior para utilizar slots nomeados, permitindo renderizar diferentes partes de um layout (ex.: cabeçalho e rodapé).

Componente com Slots Nomeados

```
<template>
  <div class="card">
    <header class="card-header">
      <slot name="header"></slot>
    </header>

    <main class="card-body">
      <slot></slot>
    </main>

    <footer class="card-footer">
      <slot name="footer"></slot>
    </footer>
  </div>
</template>

<script>
export default {
  name: "Card"
}
</script>
```

```
<style scoped>
.card {
  border: 1px solid #d1d5db;
  border-radius: 12px;
  background: #f9fafb;
  max-width: 400px;
  margin: 1rem auto;
  box-shadow: 0 4px 10px rgba(0,0,0,0.06);
}
.card-header {
  padding: 0.75rem;
  border-bottom: 1px solid #e5e7eb;
  background: #f3f4f6;
  font-weight: bold;
}
.card-body {
  padding: 1rem;
}
.card-footer {
  padding: 0.75rem;
  border-top: 1px solid #e5e7eb;
  background: #f3f4f6;
  text-align: right;
}
</style>
```

Componente Pai

```
<template>
  <div>
    <h2>Exemplo com Slots Nomeados</h2>

    <Card>
      <template v-slot:header>
        <h3>Meu Cabeçalho Personalizado</h3>
      </template>

      <p>Este é o corpo do card, preenchido pelo slot padrão.</p>

      <template v-slot:footer>
        <button @click="acao">Confirmar</button>
      </template>
    </Card>
  </div>
</template>

<script>
import Card from "./Card.vue"

export default {
  name: "App",
  components: { Card },
  methods: {
    acao() {
      alert("Ação confirmada!")
    }
  }
}
</script>
```

6. Desenvolva um mixin que contenha um método e um dado. Implemente esse mixin em dois componentes diferentes e demonstre o uso de suas funcionalidades.

Arquivo do mixin

```
export default {
  data() {
    return {
      contador: 0
    }
  },
  methods: {
    incrementar() {
      this.contador++
    }
  }
}
```

Componente 1

```
<template>
  <div class="componente-a">
    <h3>Componente A</h3>
    <p>Contador: {{ contador }}</p>
    <button @click="incrementar">Incrementar A</button>
  </div>
</template>

<script>
import meuMixin from "./meuMixin"

export default {
  name: "ComponenteA",
  mixins: [meuMixin]
}
</script>

<style scoped>
.componente-a {
  border: 2px solid #2563eb;
  padding: 1rem;
  border-radius: 10px;
  margin-bottom: 1rem;
}
</style>
```

Componente 2

```
<template>
  <div class="componente-b">
    <h3>Componente B</h3>
    <p>Contador: {{ contador }}</p>
    <button @click="incrementar">Incrementar B</button>
  </div>
</template>

<script>
import meuMixin from "./meuMixin"

export default {
  name: "ComponenteB",
  mixins: [meuMixin]
}
</script>

<style scoped>
.componente-b {
  border: 2px solid #16a34a;
  padding: 1rem;
  border-radius: 10px;
}
</style>
```

Componente Pai

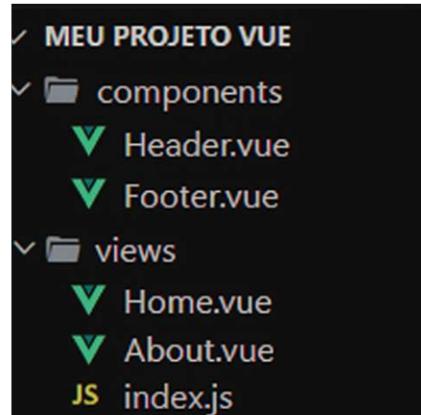
```
<template>
  <div>
    <h2>Exemplo com Mixins</h2>
    <ComponenteA />
    <ComponenteB />
  </div>
</template>

<script>
import ComponenteA from "./ComponenteA.vue"
import ComponenteB from "./ComponenteB.vue"

export default {
  name: "App",
  components: { ComponenteA, ComponenteB }
}
</script>
```

7. Organize um projeto Vue.js criando a seguinte estrutura:

- Uma pasta **components**/ contendo dois componentes reutilizáveis (ex.: **Header.vue** e **Footer.vue**).
- Uma pasta **views**/ contendo duas páginas (**Home.vue** e **About.vue**).
- Um arquivo **router/index.js** para configurar rotas entre as páginas.



8. Crie um componente que renderize uma lista de itens usando v-for. Garanta que cada item tenha uma key única com v-bind:key.

```

<template>
  <div>
    <h2>Lista de Itens</h2>
    <ul>
      <li v-for="item in items" :key="item.id">
        {{ item.name }}
      </li>
    </ul>
  </div>
</template>

<script>
export default {
  name: "ItemList",
  data() {
    return {
      items: [
        { id: 1, name: "Item 1" },
        { id: 2, name: "Item 2" },
        { id: 3, name: "Item 3" },
      ],
    };
  },
}
</script>

```

9. Desenvolva um componente pai que contenha um botão. Ao clicar no botão, o pai deve enviar um dado para o componente filho usando props.

Componente filho

```

<template>
  <div>
    <h3>Componente Filho</h3>
    <p>Dado recebido do pai: {{ message }}</p>
  </div>
</template>

<script>
export default {
  name: "ChildComponent",
  props: {
    message: {
      type: String,
      required: true
    }
  }
}
</script>

```

Componente Pai

```

<template>
  <div>
    <h2>Componente Pai</h2>
    <button @click="sendMessage">Enviar Mensagem</button>
    <ChildComponent :message="parentMessage" />
  </div>
</template>

<script>
import ChildComponent from "./ChildComponent.vue";
export default {
  name: "ParentComponent",
  components: { ChildComponent },
  data() {
    return {
      parentMessage: "Mensagem inicial"
    };
  },
  methods: {
    sendMessage() {
      this.parentMessage = "Mensagem atualizada do Pai às "
      + new Date().toLocaleTimeString();
    }
  }
}
</script>

```

10. Crie um formulário que utilize slots para personalizar seus campos de entrada. O formulário deve conter um cabeçalho, campos de texto e um botão, todos configurados dinamicamente por meio de slots.

Componente base

```
<template>
  <form @submit.prevent="handleSubmit" class="form-wrapper">
    <header class="form-header">
      <slot name="header">Cabeçalho padrão</slot>
    </header>
    <section class="form-fields">
      <slot name="fields">
        <p>Nenhum campo definido.</p>
      </slot>
    </section>
    <footer class="form-footer">
      <slot name="button">
        <button type="submit">Enviar</button>
      </slot>
    </footer>
  </form>
</template>
<script>
export default {
  name: "FormWrapper",
  methods: {
    handleSubmit() {
      alert("Formulário enviado!");
    }
  }
}
</script>
<style scoped>
.form-wrapper {
  border: 1px solid #ccc;
  padding: 1rem;
  border-radius: 8px;
  max-width: 400px;
}
.form-header {
  font-weight: bold;
  margin-bottom: 1rem;
}
.form-fields {
  margin-bottom: 1rem;
}
</style>
```

App.vue

```
<template>
  <div>
    <FormWrapper>
      <template #header>
        <h2>Cadastro de Usuário</h2>
      </template>
      <template #fields>
        <label>
          Nome:
          <input type="text" v-model="name" />
        </label>
        <br />
        <label>
          Email:
          <input type="email" v-model="email" />
        </label>
      </template>
      <template #button>
        <button type="submit">Registrar</button>
      </template>
    </FormWrapper>
  </div>
</template>
<script>
import FormWrapper from "./FormWrapper.vue";

export default {
  name: "App",
  components: { FormWrapper },
  data() {
    return {
      name: "",
      email: ""
    };
  }
}
</script>
```

