DOCUMENTAÇÃO VUEJS 3
DOCUMENTAÇÃO VUEJS 2
(Ambiente e extensões)

Seção: Introdução

VueJS é um framework progressivo para construir interfaces de usuário. Ele é projetado para ser adotado incrementalmente, o que significa que você pode usá-lo para partes específicas de seu projeto ou como um framework completo para uma aplicação complexa. VueJS é conhecido por sua simplicidade e flexibilidade, facilitando a integração com outros projetos e bibliotecas.

Seção: Usando VueJS para Interagir com a DOM

Importação do Vue

Para começar a usar o Vue, você pode incluir a biblioteca diretamente em seu HTML com um CDN ou instalá-la via npm para projetos mais complexos. Um exemplo de importação via CDN:

```
<script
src="https://cdn.jsdelivr.net/npm/vue@2"></script>
```

Entendendo VueJS Templates

Os templates no VueJS são usados para descrever a saída desejada da interface de usuário. Eles são escritos em HTML, mas com a adição de diretivas VueJS que estendem a funcionalidade do HTML padrão.

Sintaxe de Template e Instância VueJS Trabalhando Juntos

A instância Vue é a peça central da aplicação Vue. Ela é criada usando o construtor `Vue` e vinculada a um elemento DOM. Exemplo:

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

Neste exemplo, o conteúdo do elemento com id `app` será gerenciado pelo Vue, e o dado `message` será reativo.

Acessando Dados na Instância VueJS

Os dados definidos na instância Vue são acessíveis dentro dos templates usando interpolação (`{{ }}`). Exemplo:

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

Binding de Atributos

Você pode ligar atributos HTML a dados da instância Vue usando a diretiva `v-bind`. Exemplo:

```
<img v-bind:src="imageSrc">
```

Entendendo e Usando Diretivas

Diretivas são atributos especiais com o prefixo `v-` que indicam ao VueJS para fazer algo no DOM. Exemplo:

```
<p v-if="isVisible">Este parágrafo só aparece se isVisible for verdadeiro.</p>
```

Evitando Re-Renderização com v-once*

A diretiva `v-once` renderiza o elemento e seus filhos apenas uma vez, evitando re-renderizações subsequentes.

```
<p v-once>{{ message }}</p>
```

Como Imprimir HTML Puro

Para renderizar HTML puro de forma segura, utilize a diretiva `v-html`. Exemplo:

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

Escutando Eventos

Você pode escutar eventos DOM usando a diretiva `v-on`. Exemplo:

```
<button v-on:click="doSomething">Clique aqui</button>
```

Obtendo Dados do Evento

Quando você adiciona eventos a elementos do DOM em VueJS, como `click`, `input`, etc., o método associado ao evento recebe automaticamente o **objeto do evento** como argumento, que contém várias informações sobre o evento, como o alvo (`target`), tipo, coordenadas, entre outras propriedades.

Exemplo Prático:
Neste exemplo, o método `doSomething` recebe o evento como argumento e imprime no console o elemento HTML que disparou o evento (usando `event.target`).

```
<div id="app">
    <button @click="doSomething">Clique Aqui</button>
</div>

<script>
    new Vue({
        el: '#app',
        methods: {
            doSomething(evento) {
                console.log(evento.target); // Exibe o
botão clicado no console
            }
        }
    });
</script>
```

Explicação: Quando o botão é clicado, o método `doSomething` recebe o `evento`, e `evento.target` se refere ao elemento que disparou o evento (neste caso, o botão). Essa

informação pode ser usada para manipular dinamicamente o DOM, aplicar estilos ou interagir com o usuário de forma personalizada.

Passando nossos próprios Argumentos com Eventos

Para passar argumentos personalizados, use a sintaxe de método dentro da diretiva `v-on`.

```
<button v-on:click="doSomething('custom
argument')">Clique aqui</button>
```

Modificadores de Eventos

Os **modificadores de eventos** em VueJS são usados para indicar uma intenção específica ao lidar com eventos. Eles modificam o comportamento padrão do evento ou afetam como ele é tratado. Um exemplo comum é o `prevent`, que previne a ação padrão de um evento (como o envio de um formulário).

Exemplo Prático:
Aqui está um formulário simples com um botão de submit. Ao clicar no botão, o comportamento padrão de envio (recarregar a página) é **impedido** com o modificador `.prevent`.

```
<div id="app">
    <form v-on:submit.prevent="onSubmit">
        <input type="text" v-model="name"
placeholder="Digite seu nome">
        <button type="submit">Enviar</button>
    </form>
</div>

<script>
    new Vue({
        el: '#app',
        data: {
            name: ''
        },
        methods: {
            onSubmit() {
```

```
                console.log("Formulário submetido!",
this.name);
            }
        }
    });
</script>
```

Explicação: O modificador `.prevent` impede o comportamento padrão do submit, que normalmente recarregará a página. Dessa forma, o método `onSubmit` é executado sem que o navegador realize o comportamento padrão.

Eventos de Teclado

Você pode escutar eventos de teclado com modificadores específicos.

```
<input v-on:keyup.enter="submit"> <!-- Aciona o método
submit ao pressionar Enter -->
```

Código JavaScript no Template

Embora seja desencorajado, você pode incluir pequenas expressões JavaScript nos templates.

```
<p>{{ number + 1 }}</p>
```

Usando Two-Way-Binding

O `v-model` cria uma ligação bidirecional entre o dado e o input.

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

Propriedades Computadas

Propriedades computadas são declaradas como métodos, mas são acessadas como propriedades.

```
computed: {
    reversedMessage() {
```

```
        return
this.message.split('').reverse().join('');
    }
}
```

Monitorando as Mudanças

Você pode usar `watchers` para executar código em resposta a mudanças de dados.

```
watch: {
    message(newVal, oldVal) {
        console.log(`Message changed from ${oldVal} to
${newVal}`);
    }
}
```

Sintaxe Reduzida (Shorthands)

Vue oferece shorthands para `v-bind` (`:`) e `v-on` (`@`).

```
<img :src="imageSrc">
<button @click="doSomething">Clique aqui</button>
```

Estilo Dinâmico e Classe CSS

O VueJS permite que você aplique **classes CSS e estilos** de forma dinâmica usando o `v-bind:class` e `v-bind:style`, o que facilita a alteração do visual de um componente com base em variáveis ou estados reativos.

Exemplo Prático - v-bind:class:

Aqui, a classe CSS `active` é aplicada ao `div` quando a propriedade `isActive` for verdadeira.

```
<div id="app">
    <div :class="{ active: isActive }">Este é um
bloco</div>
    <button @click="toggleActive">Alternar
Ativo</button>
```

```
</div>

<script>
    new Vue({
        el: '#app',
        data: {
            isActive: false
        },
        methods: {
            toggleActive() {
                this.isActive = !this.isActive;
            }
        }
    });
</script>

<style>
    .active {
        background-color: yellow;
        color: black;
    }
</style>
```

Explicação: Quando o usuário clica no botão "Alternar Ativo", o valor de `isActive` é invertido, o que adiciona ou remove dinamicamente a classe `active` do `div`. Se `isActive` for `true`, o `div` recebe a classe `active`, aplicando os estilos definidos no CSS.

Exemplo Prático - v-bind:style:

Você pode vincular estilos diretamente a um `div` usando `v-bind:style`. No exemplo abaixo, a cor e o tamanho da fonte são definidos com base nos dados reativos.

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

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

<script>
    new Vue({
        el: '#app',
        data: {
            activeColor: 'blue',
            fontSize: 18
        }
    });
</script>
```

Explicação: Aqui, a cor do texto é `blue` e o tamanho da fonte é de 18 pixels, porque estamos vinculando as propriedades `activeColor` e `fontSize` diretamente ao estilo do elemento `div`.

4. Estilo Dinâmico Sem Classes CSS

Você também pode aplicar estilos diretamente via `v-bind:style` sem usar classes CSS, o que é útil para cenários onde os estilos são gerados dinamicamente.

Exemplo Prático - Computed Style:
Aqui, os estilos são aplicados dinamicamente com base em uma **propriedade computada**.

```
<div id="app">
    <div :style="computedStyle">Texto com estilo
dinâmico</div>
    <button @click="toggleStyle">Mudar Estilo</button>
</div>

<script>
    new Vue({
        el: '#app',
        data: {
            isStyled: false
        },
```

```
        computed: {
            computedStyle() {
                return this.isStyled ? { color: 'red',
fontSize: '24px' } : { color: 'green', fontSize: '16px'
};
            }
        },
        methods: {
            toggleStyle() {
                this.isStyled = !this.isStyled;
            }
        }
    });
</script>
```

Explicação: Com base no valor da propriedade `isStyled`, o estilo do `div` muda entre `color: red` e `fontSize: 24px` ou `color: green` e `fontSize: 16px`. Isso é feito de forma reativa usando uma **propriedade computada**.

Seção: Usando Condicionais & Renderização de Listas

Renderização Condicional com v-if/v-else

```
<p v-if="seen">Agora você me vê</p>
<p v-else>Agora você não me vê</p>
```

Seleção Múltipla com v-else-if

```
<div v-if="type === 'A'">A</div>
<div v-else-if="type === 'B'">B</div>
<div v-else>C</div>
```

Usando v-if com Template

```
<template v-if="ok">
    <h1>OK</h1>
    <p>All good</p>
</template>
```

Esconda o Elemento com v-show

O `v-show` é similar ao `v-if`, mas apenas alterna a visibilidade do elemento.

```
<p v-show="seen">Você pode me ver</p>
```

Renderizando Lista com v-for

O `v-for` é usado para renderizar listas de itens.

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

Acessando o Índice Atual

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

Usando v-for com Template

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

Iterando em Objetos

Você pode iterar em propriedades de um objeto.

```
<div v-for="(value, key) in object">{{ key }}: {{ value }}</div>
```

Iterando em uma Lista de Números

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

Identificando os Elementos no v-for

Use a diretiva `key` para ajudar o Vue a identificar quais itens mudaram.

```
<li v-for="item in items" :key="item.id">{{ item.text }}</li>
```

Seção: Entendendo a Instância Vue

Noções básicas sobre a Instância Vue

A instância Vue é o coração de uma aplicação VueJS, gerenciando o estado e os comportamentos da interface do usuário.

Usando Múltiplas Instâncias Vue

Você pode ter várias instâncias Vue em uma única página.

```
var app1 = new Vue({ ... });
var app2 = new Vue({ ... });
```

Acessando a Instância Vue Externamente

A instância Vue pode ser acessada através da variável na qual foi instanciada.

```
console.log(app1.message);
```

Como o VueJS Gerencia os Dados e Métodos

O VueJS usa um sistema de reatividade para gerenciar mudanças de dados e atualizar a interface de usuário de forma eficiente.

Meu Vue Framework

A criação de um Vue Framework envolve a configuração e o gerenciamento das instâncias Vue de acordo com as necessidades específicas do projeto.

Uma Análise Mais Detalhada de $el e $data

Os objetos `$el` e `$data` referem-se ao elemento DOM gerenciado pelo Vue e aos dados reativos, respectivamente.

Colocando $refs e Usando nos Templates

Referências (`$refs`) são usadas para acessar elementos DOM diretamente.

```
<button ref="button">Clique</button>
```

```
this.$refs.button.textContent = 'Clicado';
```

Onde Aprender Mais sobre a API do Vue

A documentação oficial do VueJS é o melhor recurso para aprender sobre a API do Vue.

Montando um Template

A montagem de templates envolve a combinação de HTML, CSS e lógica VueJS para criar interfaces interativas e dinâmicas. Um template Vue típico combina HTML com diretivas VueJS para vinculação de dados, eventos e condicionais, criando uma estrutura de interface flexível e reativa.

Usando Componentes

Componentes são blocos reutilizáveis de código que encapsulam HTML, CSS e JavaScript. Eles são registrados global ou localmente e usados para dividir a aplicação em partes menores e mais gerenciáveis.

```
Vue.component('my-component', {
    template: '<div>A custom component!</div>'
});
```

```
<my-component></my-component>
```

Limitações dos Templates

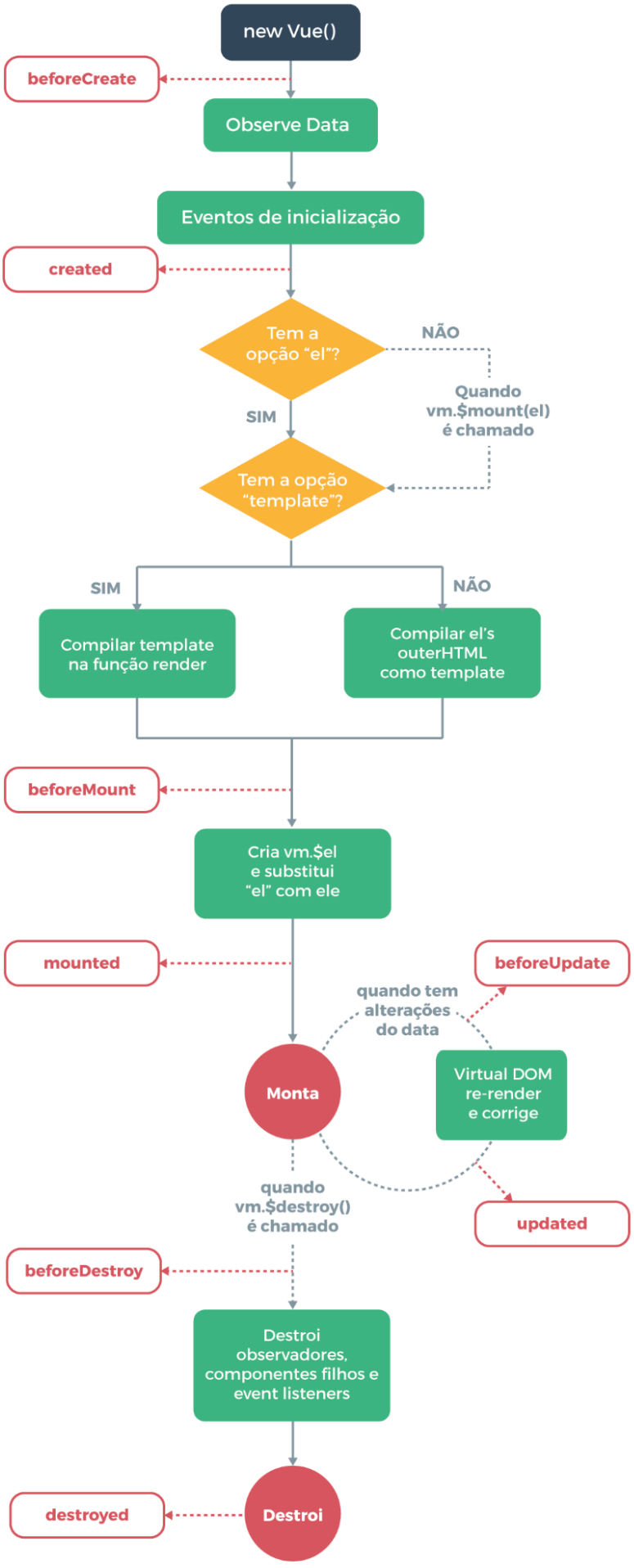
Os templates Vue são limitados pelo escopo e regras do HTML. Para lógica mais complexa, métodos e propriedades computadas devem ser usados em vez de incluir lógica pesada diretamente nos templates.

Como o VueJS Atualiza o DOM

O VueJS utiliza um DOM virtual (virtual DOM) para rastrear as mudanças e atualizar apenas as partes necessárias do DOM real, tornando o processo eficiente e rápido.

O Ciclo de Vida da Instância Vue

O ciclo de vida da instância Vue inclui vários estágios desde a criação até a destruição. Métodos de ciclo de vida (hooks) permitem que você execute código em diferentes momentos do ciclo de vida da instância.


new Vue()
beforeCreate
Observe Data
Eventos de inicialização
created
Tem a opção “el”?
NÃO
Quando vm.$mount(el) é chamado
SIM
Tem a opção “template”?
SIM
NÃO
Compilar template na função render
Compilar el’s outerHTML como template
beforeMount
Cria vm.$el e substitui “el” com ele
mounted
beforeUpdate
quando tem alterações do data
Monta
Virtual DOM re-render e corrige
quando vm.$destroy() é chamado
updated
beforeDestroy
Destroi observadores, componentes filhos e event listeners
destroyed
Destroi

Principais Etapas do Ciclo de Vida

1. **Criação (Creation)**
    - **beforeCreate:** Executado logo após a instância Vue ser inicializada. Neste ponto, as instâncias de dados observáveis (`data`, `props`) e as opções de observadores (`methods`, `computed`, `watch`) ainda não estão disponíveis.
    - **created:** Após a instância ser criada, as propriedades reativas são configuradas e a instância já pode acessar `data`, `props`, `computed`, e `methods`. No entanto, o DOM ainda não foi montado.

Exemplo:

```
new Vue({
    data() {
        return {
            message: 'Hello Vue!'
        };
    },
    beforeCreate() {
        console.log('beforeCreate:', this.message); //
undefined
    },
    created() {
        console.log('created:', this.message); //
'Hello Vue!'
    }
});
```

2. **Montagem (Mounting)**
    - **beforeMount:** Chamado antes de o DOM ser montado. Aqui, o template ou o `render function` ainda não foi transformado em um DOM renderizado.
    - **mounted:** Este hook é disparado após a instância ser montada no DOM. Neste ponto, você pode acessar o DOM renderizado através do `$el` e manipular ou interagir com elementos do DOM.

Exemplo:

```
new Vue({
    el: '#app',
```

```
    data() {
        return {
            message: 'Hello World!'
        };
    },
    beforeMount() {
        console.log('beforeMount:', this.$el); //
undefined
    },
    mounted() {
        console.log('mounted:', this.$el); // <div
id="app">...</div>
    }
});
```

3. **Atualização (Updating)**
    - **beforeUpdate:** Chamado sempre que os dados reativos mudam, antes que o DOM seja atualizado.
    - **updated:** Após a atualização do DOM, esse hook é chamado. Este é um bom momento para fazer manipulações adicionais no DOM que dependem da atualização dos dados.

Exemplo:

```
new Vue({
    el: '#app',
    data() {
        return {
            counter: 0
        };
    },
    beforeUpdate() {
        console.log('beforeUpdate:', this.counter);
    },
    updated() {
        console.log('updated:', this.counter);
```

```
    },
    methods: {
        incrementCounter() {
            this.counter++;
        }
    }
});
```

Uso:

```
<div id="app">
    <p>{{ counter }}</p>
    <button
@click="incrementCounter">Increment</button>
</div>
```

4. **Destruição (Destroying)**
    - **beforeDestroy:** Chamado antes de a instância Vue ser destruída. Aqui, você pode realizar limpeza, como remover ouvintes de eventos ou parar timers.
    - **destroyed:** Chamado após a instância Vue ser destruída. Todos os bindings e ouvintes de eventos foram removidos.

Exemplo:

```
new Vue({
    el: '#app',
    data() {
        return {
            message: 'Goodbye Vue!'
        };
    },
    beforeDestroy() {
        console.log('beforeDestroy:', this.message);
    },
    destroyed() {
        console.log('destroyed:', this.message);
    },
```

```
    methods: {
        destroyInstance() {
            this.$destroy();
        }
    }
});
```

Uso:

```
<div id="app">
    <p>{{ message }}</p>
    <button @click="destroyInstance">Destroy</button>
</div>
```

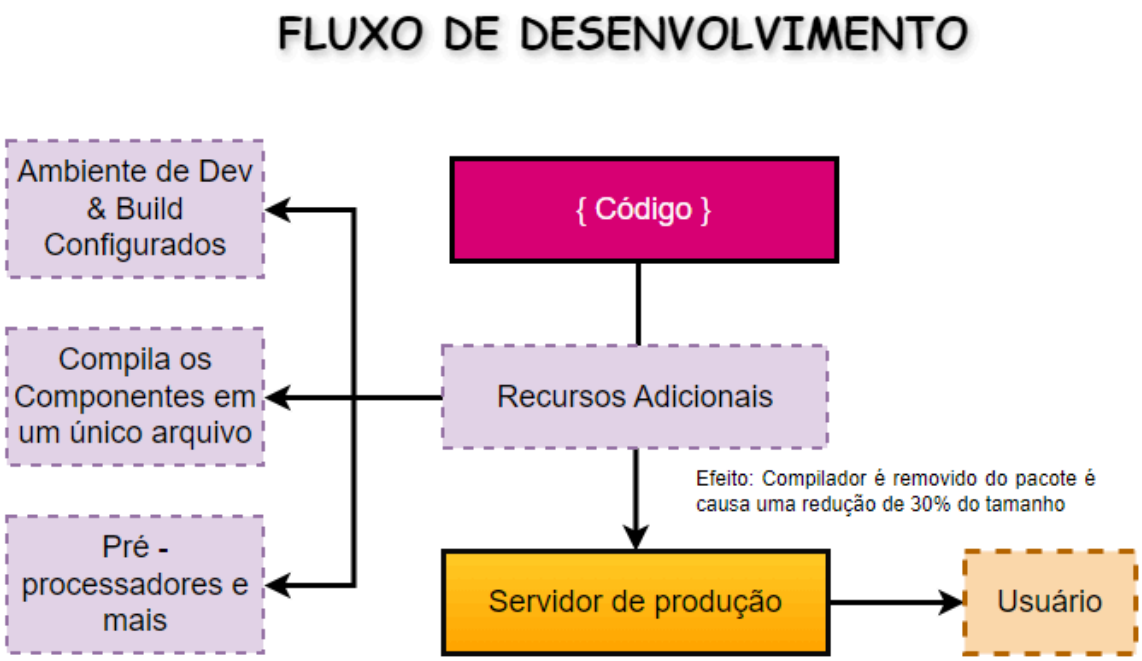
Seção: Fluxo de Desenvolvimento "Real" Usando Vue CLI

Por que Precisamos de um Servidor Web?

Um servidor web é necessário para servir a aplicação, gerenciar rotas, e fornecer uma experiência de desenvolvimento mais realista e robusta.

O que "Fluxo de Desenvolvimento" Significa?

Fluxo de desenvolvimento refere-se ao conjunto de ferramentas e processos que facilitam a criação, teste e implantação de aplicações web.



Usando o Vue CLI para Criar Projetos

O Vue CLI é uma ferramenta poderosa que simplifica a criação e o gerenciamento de projetos VueJS. Ela fornece uma estrutura padrão e ferramentas integradas para facilitar o desenvolvimento.

Instalando o Vue CLI e Criando um Novo Projeto

Instale o Vue CLI globalmente com npm:

```
npm install -g @vue/cli
```

ou

```
yarn global add @vue/cli
```

Crie um novo projeto:

```
vue create my - project
```

Uma Visão Geral sobre a Estrutura de Pastas

A estrutura de pastas de um projeto Vue CLI típico inclui diretórios como `src` para código-fonte, `public` para arquivos públicos, e `node_modules` para dependências.

Entendendo os Arquivos "vue"

Arquivos `.vue` são componentes de arquivo único que combinam template, script e estilo em um único arquivo, proporcionando uma maneira modular de desenvolver componentes.

Como Construir sua APP para Produção

Para preparar sua aplicação para produção, use o comando:

```
npm run build
```

Este comando gera uma versão otimizada da aplicação.

Criando um Projeto e Salvando Template

Você pode criar e salvar templates personalizados para reutilização em outros projetos Vue CLI.

Adicionando Plugins ao Projeto

Plugins adicionam funcionalidades adicionais ao projeto Vue. Eles podem ser adicionados durante a criação do projeto ou posteriormente com o Vue CLI.

Conclusão do Módulo

O módulo de fluxo de desenvolvimento com Vue CLI fornece uma base sólida para criar e gerenciar projetos VueJS de forma eficiente e profissional.

Mais sobre Arquivos "vue" e o CLI

Explore a documentação do Vue CLI para obter mais informações sobre como personalizar e otimizar seu fluxo de desenvolvimento.

Depurando Projetos VueJS

Use ferramentas como Vue Devtools para inspecionar e depurar aplicações VueJS. Elas permitem visualizar a árvore de componentes, estado, e eventos em tempo real.

Seção: Introdução aos Componentes

Revisão Sobre Componentes

Componentes são fundamentais no VueJS para criar interfaces de usuário modulares e reutilizáveis. Eles permitem a organização do código de forma estruturada e eficiente.

Registrar Componentes (Global e Local)

Componentes podem ser registrados globalmente com `Vue.component` ou localmente dentro de uma instância Vue ou outro componente.

```
Vue.component('my-component', { ... }); // Global
```

```
components: {
    'my-component': MyComponent // Local
}
```

Criar Projeto Usando Vue CLI

Use o Vue CLI para iniciar rapidamente um novo projeto Vue com suporte a componentes.

Criar um Componente

Crie componentes com arquivos `.vue` ou com a sintaxe JavaScript padrão.

```
Vue.component('my-component', {
    template: '<div>My Component</div>'
});
```

Usando Componentes

Inclua componentes nos templates usando a tag do componente.

```
<my-component></my-component>
```

Usando CSS com Escopo de Componente

O Vue permite definir estilos CSS escopados a um componente usando a opção `scoped`.

```
<style scoped>
    .my-style {
        color: red;
    }
</style>
```

Organizando os Componentes em Pastas

Organize componentes em pastas de acordo com sua funcionalidade ou relação hierárquica.

Regras de Nomes de Componentes

Nomeie componentes de forma clara e consistente, geralmente em `PascalCase` ou `kebab-case`.

Seção: Comunicação Entre Componentes

Comunicação Direta com Props

Props permitem a passagem de dados de um componente pai para um componente filho.

```
Vue.component('child', {
    props: ['message']
});
```

```
<child message="Hello"></child>
```

Nome das Propriedades são Case-Sensitive

Os nomes das props são case-sensitive, então `myProp` e `myprop` seriam considerados diferentes.

Usando Props no Componente Filho

Props recebidas em um componente filho são tratadas como dados locais.

```
<template>
  <div>{{ message }}</div>
</template>
<script>
export default {
  props: ['message']
};
</script>
```

Validando Props

Props podem ser validadas usando a opção `type` e outras opções de validação.

```
props: {
    message: {
        type: String,
```

```
            required: true
    }
}
```

Comunicação Indireta com Eventos Personalizados

A **comunicação indireta com eventos personalizados** é uma maneira eficaz de permitir que um componente filho envie informações de volta para o componente pai. No VueJS, os componentes podem emitir eventos personalizados com `this.$emit()`, que podem ser ouvidos pelo componente pai.

Exemplo Prático:

Neste exemplo, o componente filho (`ChildComponent`) emite um evento personalizado chamado `myEvent` com uma mensagem. O componente pai (`ParentComponent`) escuta esse evento e responde a ele.

```
<div id="app">
  <parent-component></parent-component>
</div>

<script>
Vue.component("child-component", {
  template: '<button @click="sendMessage">Enviar
Mensagem</button>',
  methods: {
    sendMessage() {
      this.$emit("myEvent", "Olá, Pai!"); // Emite o
evento 'myEvent'
    },
  },
});

Vue.component("parent-component", {
  template: `
    <div>
```

```
      <child-component
@myEvent="handleEvent"></child-component>
      <p>{{ message }}</p>
    </div>
  `,
  data() {
    return {
      message: "",
    };
  },
  methods: {
    handleEvent(data) {
      this.message = data; // Captura a mensagem do
evento
    },
  },
});

new Vue({
  el: "#app",
});
</script>
```

Explicação: Quando o botão no componente filho é clicado, ele emite o evento `myEvent`, enviando a mensagem "Olá, Pai!". O componente pai escuta esse evento e atualiza sua própria mensagem com o valor recebido.

Comunicação Indireta com Callback

Outra abordagem para comunicação entre componentes é passar **funções de callback** como props. Isso permite que o componente filho execute uma função no componente pai, fornecendo mais controle.

Exemplo Prático:

No exemplo abaixo, o componente pai passa uma função como prop para o filho. O filho chama essa função para notificar o pai de uma ação.

```
<div id="app">
  <parent-component></parent-component>
</div>

<script>
Vue.component('child-component', {
  props: ['notify'],
  template: '<button @click="notifyParent">Notificar
Pai</button>',
  methods: {
    notifyParent() {
      this.notify('Mensagem do Filho'); // Chama a
função passada pelo pai
    }
  }
});

Vue.component('parent-component', {
  template: `
    <div>
      <child-component
:notify="updateMessage"></child-component>
      <p>{{ message }}</p>
    </div>
  `,
  data() {
    return {
      message: ''
    };
  },
  methods: {
```

```
    updateMessage(data) {
      this.message = data; // Atualiza a mensagem
    }
  }
});

new Vue({
  el: '#app'
});
</script>
```

Explicação: A função `updateMessage` é passada para o componente filho via prop. Quando o filho chama a função, o pai atualiza sua mensagem com o valor recebido.

Problema da Comunicação entre Componentes Irmãos

A comunicação entre **componentes irmãos** não é direta no VueJS, porque irmãos não têm acesso imediato ao estado ou métodos uns dos outros. Normalmente, essa comunicação precisa ser mediada por um componente pai ou outra solução.

Exemplo do Problema:

```
<div id="app">
  <sibling-one></sibling-one>
  <sibling-two></sibling-two>
</div>

<script>
Vue.component('sibling-one', {
  template: '<button @click="sendMessage">Enviar para o
Irmão</button>',
  methods: {
    sendMessage() {
      // Como enviar uma mensagem para o irmão?
    }
  }
```

```
});

Vue.component('sibling-two', {
  template: '<p>Aguardando Mensagem</p>'
});

new Vue({
  el: '#app'
});
</script>
```

Explicação: Aqui, `sibling-one` não pode se comunicar diretamente com `sibling-two`. Para resolver isso, podemos usar um **Event Bus** ou **Vuex**.

Usando Event Bus para Comunicação entre Componentes Irmãos

Uma solução comum para o problema da comunicação entre componentes irmãos é usar um **Event Bus**. O Event Bus é um **instância de Vue** que atua como um hub central para eventos. Componentes irmãos podem emitir eventos para o Event Bus e outros componentes podem escutar esses eventos.

Exemplo Prático:

```
<div id="app">
  <sibling-one></sibling-one>
  <sibling-two></sibling-two>
</div>

<script>
// Criando um Event Bus
const EventBus = new Vue();

Vue.component('sibling-one', {
  template: '<button @click="sendMessage">Enviar para o
Irmão</button>',
  methods: {
```

```
    sendMessage() {
      EventBus.$emit('message', 'Olá, Irmão!'); //
Emite um evento no Event Bus
    }
  }
});

Vue.component('sibling-two', {
  data() {
    return {
      message: ''
    };
  },
  created() {
    EventBus.$on('message', (data) => {
      this.message = data; // Ouve o evento e atualiza
a mensagem
    });
  },
  template: '<p>{{ message }}</p>'
});

new Vue({
  el: '#app'
});
</script>
```

Explicação: Quando `sibling-one` emite o evento `message` no Event Bus, `sibling-two` escuta o evento e atualiza sua mensagem com o valor recebido.

Adicionando Métodos no Event Bus

Além de emitir e ouvir eventos, você também pode adicionar **métodos** ao Event Bus para centralizar mais comportamentos que diferentes componentes possam utilizar.

Exemplo Prático:

```
<script>
// Event Bus com Métodos
const EventBus = new Vue({
  methods: {
    sendMessage(message) {
      this.$emit('message', message);
    }
  }
});

Vue.component('sibling-one', {
  template: '<button @click="sendMessage">Enviar para o Irmão</button>',
  methods: {
    sendMessage() {
      EventBus.sendMessage('Olá, do Irmão 1'); // Usa o método do Event Bus
    }
  }
});

Vue.component('sibling-two', {
  data() {
    return {
      message: ''
    };
  },
  created() {
    EventBus.$on('message', (data) => {
      this.message = data;
    });
  },
  template: '<p>{{ message }}</p>'
```

```
});

new Vue({
  el: '#app'
});
</script>
```

Explicação: Aqui, adicionamos um método `sendMessage` ao Event Bus, permitindo que qualquer componente que tenha acesso ao Event Bus possa utilizar o método para enviar uma mensagem.

Props por Valor vs Props por Referência

Ao passar **props** para um componente em VueJS, é importante entender a diferença entre **valores primitivos** (passados por valor) e **objetos/arrays** (passados por referência).

- **Primitivos** (como números e strings) são passados por valor, ou seja, o componente filho recebe uma **cópia** do valor.
- **Objetos e Arrays** são passados por referência, o que significa que tanto o componente pai quanto o filho estão compartilhando o **mesmo objeto/array**.

Exemplo Prático:

```
<div id="app">
  <parent-component></parent-component>
</div>

<script>
Vue.component('child-component', {
  props: ['primitiveValue', 'referenceValue'],
  template: `
    <div>
      <p>Valor Primitivo: {{ primitiveValue }}</p>
      <p>Valor de Referência: {{ referenceValue.text
}}</p>
    </div>
  `
```

```
});

Vue.component('parent-component', {
  template: `
    <div>
      <child-component :primitiveValue="primitive"
:referenceValue="reference"></child-component>
    </div>
  `,
  data() {
    return {
      primitive: 'Texto Simples',
      reference: { text: 'Texto Complexo' }
    };
  }
});

new Vue({
  el: '#app'
});
</script>
```

Explicação:

- O valor **primitivo** (`primitive`) é uma string simples. Alterações no valor dentro do filho não afetarão o valor no pai.
- O valor **referenciado** (`reference`) é um objeto. Se o objeto for modificado no componente filho, essa mudança será refletida no pai, pois eles compartilham o mesmo objeto.

Seção: Uso Avançado de Componentes

Como Passar Conteúdo no Corpo do Componente?

Você pode passar conteúdo dinâmico para dentro de um componente usando slots.

Passando Conteúdo com Slots

Slots são utilizados para inserir conteúdo dinâmico dentro de componentes.

```
<div class="citacao">
        <slot></slot>
    </div>
```

Dentro do componente:

```
 <Citacao>
     <h4> {{citacoes[indice].autor}}</h4>
     <p> {{citacoes[indice].texto}}</p>
     <h6> {{citacoes[indice].fonte}}</h6>
 </Citacao>
```

```
<child>
    <p>Some content</p>
</child>
```

Como o Conteúdo do Slot é Estilizado

O conteúdo inserido em slots herda os estilos do componente pai, mas pode ser estilizado adicionalmente no componente filho.

Usando Múltiplos Slots (Slots Nomeados)

Você pode usar slots nomeados para passar diferentes partes do conteúdo.

```
<child>
    <template v-slot:header>
        <h1>Header Content</h1>
    </template>
    <template v-slot:footer>
        <p>Footer Content</p>
    </template>
</child>
```

Usando Slot Padrão

O slot padrão é utilizado quando não é fornecido um nome específico.

```
<child>
    <p>Default slot content</p>
</child>
```

Alternando entre Múltiplos Componentes com Componentes Dinâmicos

Use o componente `component` para alternar dinamicamente entre componentes.

```
<component :is="componentType"></component>
```

Entendendo o Comportamento do Componente Dinâmico

Componentes dinâmicos permitem alternar a renderização de diferentes componentes com base no estado da aplicação.

Mantendo o Componente Dinâmico Vivo

Use `keep-alive` para manter o estado dos componentes dinâmicos quando alternar entre eles.

```
<keep-alive>
    <component :is="componentType"></component>
</keep-alive>
```

Métodos de Ciclo de Vida de um Componente Dinâmico

Componentes dinâmicos possuem métodos de ciclo de vida como `activated` e `deactivated` para gerenciar seu estado.

Seção: Manipulando Entrada de Usuário com Formulários

Ligação básica em Formulário usando `<input>`

Crie um formulário básico com ligação bidirecional de dados usando `v-model`:

```
<input v-model="message" placeholder="Digite uma
mensagem">
<p>Mensagem: {{ message }}</p>
```

Agrupando Dados e Pré-populando Inputs

Agrupe dados no estado do componente e pré-popule os inputs:

```
<input v-model="user.name" placeholder="Nome">
<input v-model="user.email" placeholder="Email">
<p>Nome: {{ user.name }}</p>
<p>Email: {{ user.email }}</p>
```

```
data() {
    return {
        user: {
            name: 'John Doe',
            email: 'john.doe@example.com'
        }
    };
}
```

Modificar Entrada de Usuário com Modificadores de Input

Use modificadores para ajustar a entrada do usuário:

```
<input v-model.lazy="message" placeholder="Digite uma
mensagem">
<input v-model.trim="message" placeholder="Digite uma
mensagem">
<input v-model.number="age" placeholder="Idade">
```

Usando `<textarea>` e Salvando Quebras de Linha

Capture entradas de texto multi-linhas com `<textarea>`:

```
<textarea v-model="message" placeholder="Digite uma
mensagem" style="white-space: pre-line;"
```

```
></textarea>
<p>{{ message }}</p>
```

Usando Checkboxes e Salvando os Dados em um Array

Gerencie múltiplas seleções com checkboxes:

```
<input type="checkbox" v-model="checkedNames"
value="Jack"> Jack
<input type="checkbox" v-model="checkedNames"
value="John"> John
<input type="checkbox" v-model="checkedNames"
value="Mike"> Mike
<p>Nomes: {{ checkedNames }}</p>
```

```
data() {
    return {
        checkedNames: []
    };
}
```

Usando Botões Radio

Capture a seleção do usuário com botões radio:

```
<input type="radio" v-model="picked" value="One"> One
<input type="radio" v-model="picked" value="Two"> Two
<p>Selecionado: {{ picked }}</p>
```

```
data() {
    return {
        picked: ''
    };
```

```
}
```

Manipulando Combobox com `<select>` e `<option>`

Trabalhe com seleções de `<select>`:

```
<select v-model="selected">
    <option disabled value="">Por favor
selecione</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
</select>
<p>Selecionado: {{ selected }}</p>
```

```
data() {
    return {
        selected: ''
    };
}
```

O que o `v-model` faz e Como criar um Input Personalizado

Entenda o funcionamento do `v-model` e crie um componente de input personalizado:

```
Vue.component('custom-input', {
    props: ['value'],
    template: `<input :value="value"
@input="$emit('input', $event.target.value)">`
});
```

```
<custom-input v-model="customMessage"></custom-input>

<p>{{ customMessage }}</p>
```

```
data() {
    return {
        customMessage: ''
    };
}
```

Submetendo Formulário

Submeta formulários usando VueJS:

```
<form @submit.prevent="submitForm">
    <input v-model="form.name" placeholder="Nome">
    <input v-model="form.email" placeholder="Email">
    <button type="submit">Enviar</button>
</form>
```

```
methods: {
    submitForm() {
        console.log(this.form);
    }
}
```

Seção: Usando e Criando Diretivas

Entendendo Diretivas

As diretivas em VueJS são atributos especiais com o prefixo `v-`. Elas são usadas para adicionar comportamento reativo a um elemento DOM.

Como a Diretiva Funciona - Funções Gatilho (Hooks)

As diretivas possuem hooks que permitem interagir com o DOM em diferentes momentos do ciclo de vida do elemento:

- `bind`: Chamado uma vez, quando a diretiva é ligada ao elemento.
- `inserted`: Chamado quando o elemento é inserido no DOM pai.
- `update`: Chamado quando o elemento é atualizado.
- `componentUpdated`: Chamado depois que o componente filho foi atualizado.
- `unbind`: Chamado uma vez, quando a diretiva é desvinculada do elemento.

Criando uma Diretiva Simples

Para criar uma diretiva simples, use o método `Vue.directive`:

```
Vue.directive('focus', {
    inserted: function (el) {
        el.focus();
    }
});
```

Use essa diretiva em um elemento:

```
<input v-focus>
```

Passando Valor para Diretiva Personalizada

Você pode passar valores para diretivas personalizadas:

```
Vue.directive('color', {
    bind(el, binding) {
        el.style.color = binding.value;
    }
});
```

Use a diretiva no template:

```
<p v-color="'red'">Texto em vermelho</p>
```

Passando Argumento para Diretiva Personalizada

As diretivas podem receber argumentos usando `binding.arg`:

```
Vue.directive('color', {
```

```
    bind(el, binding) {
        if (binding.arg === 'bg') {
            el.style.backgroundColor = binding.value;
        } else {
            el.style.color = binding.value;
        }
    }
});
```

Use a diretiva com um argumento:

```
<p v-color:bg="'yellow'">Fundo amarelo</p>
<p v-color="'red'">Texto vermelho</p>
```

Modificando Diretivas Personalizadas com Modificadores

Os modificadores são sufixos especiais indicados por um ponto:

```
Vue.directive('color', {
    bind(el, binding) {
        let delay = 0;
        if (binding.modifiers.delayed) {
            delay = 2000;
        }
        setTimeout(() => {
            el.style.color = binding.value;
        }, delay);
    }
});
```

Use a diretiva com modificadores:

```
<p v-color.delayed="'blue'">Texto azul com atraso</p>
```

Registrando Diretivas Localmente

Diretivas também podem ser registradas localmente em componentes:

```
export default {
    directives: {
        focus: {
            inserted: function (el) {
                el.focus();
            }
        }
    }
};
```

Usando Múltiplos Modificadores

Você pode combinar vários modificadores:

```
<p v-color.delayed.slow="'green'">Texto verde com atraso e animação lenta</p>
```

Passando Valores mais Complexos para as Diretivas

Para passar valores complexos, como objetos, use a sintaxe de vinculação de dados:

```
Vue.directive('color', {
    bind(el, binding) {
        el.style.color = binding.value.color;
        el.style.fontSize = binding.value.size + 'px';
    }
});
```

Use a diretiva com um objeto:

```
<p v-color="{ color: 'purple', size: 18 }">Texto roxo</p>
```

Seção: Melhorando sua App com Filtros e Mixins

Criando um Filtro Local

Filtros são usados para formatar a saída de dados. Registre um filtro local em um componente:

```
filters: {
    capitalize(value) {
        if (!value) return '';
        value = value.toString();
                return  value.charAt(0).toUpperCase()  +
value.slice(1);
    }
}
```

Use o filtro em um template:

```
<p>{{ message | capitalize }}</p>
```

Filtro Global e Como Encadear Múltiplos Filtros

Registre um filtro globalmente:

```
Vue.filter('capitalize', function (value) {
    if (!value) return '';
    value = value.toString();
              return   value.charAt(0).toUpperCase()   +
value.slice(1);
});
```

Encadeie filtros:

```
<p>{{ message | capitalize | truncate(10) }}</p>
```

Filtro & v-bind

Você pode usar filtros com `v-bind`:

```
<p v-bind:class="message | capitalize"></p>
```

Duplicando Código para Usar os Mixins

Mixins permitem compartilhar funcionalidades entre componentes. Crie um mixin:

```
const myMixin = {
    created() {
        console.log('Mixin criado!');
    },
    methods: {
        greet() {
            console.log('Hello from mixin!');
        }
    }
};
```

Use o mixin em um componente:

```
export default {
    mixins: [myMixin]
};
```

Criando e Usando Mixins

Crie um mixin para compartilhar dados e métodos:

```
const myMixin = {
    data() {
        return {
            sharedData: 'Este é um dado compartilhado'
        };
    },
    methods: {
        sharedMethod() {
                        console.log('Este é um método
compartilhado');
```

```
        }
    }
};
```

Criando um Mixin Global (Caso Especial!)

Registre um mixin globalmente:

```
Vue.mixin({
    created() {
        console.log('Mixin global criado!');
    }
});
```

Seção: Adicionando Animações e Transições

Entendendo as Transições

Transições são usadas para animar a entrada e saída de elementos do DOM. O VueJS oferece um sistema de transição integrado para facilitar esse processo.

Preparando o Código para Usar Transições

Envolva o conteúdo que deseja animar com o componente `<transition>`:

```
<transition name="fade">
    <p v-if="show">Olá, VueJS!</p>
</transition>
```

Configurando Transição

Defina classes CSS para a transição:

```
.fade-enter-active,
.fade-leave-active {
    transition: opacity 0.5s;
}

.fade-enter,
```

```
.fade-leave-to {
    opacity: 0;
}
```

Definindo as Classes CSS para Transição

As classes padrão são:

- `v-enter`: Estado inicial da entrada.
- `v-enter-active`: Estado ativo durante a entrada.
- `v-enter-to`: Estado final da entrada.
- `v-leave`: Estado inicial da saída.
- `v-leave-active`: Estado ativo durante a saída.
- `v-leave-to`: Estado final da saída.

Criando Transição "Fade" com Propriedade CSS transition

```
.fade-enter-active,
.fade-leave-active {
    transition: opacity 1s;
}

.fade-enter,
.fade-leave-to {
    opacity: 0;
}
```

Criando Transição "Slide" com Propriedade CSS animation

```
.slide-enter-active {
  animation: slide-in 0.5s;
}
.slide-leave-active {
  animation: slide-out 0.5s;
}
@keyframes slide-in {
```

```
  from {
    transform: translateX(100%);
  }
  to {
    transform: translateX(0);
  }
}
@keyframes slide-out {
  from {
    transform: translateX(0);
  }
  to {
    transform: translateX(-100%);
  }
}
```

Misturando as Propriedades transition e animation

Combine transições e animações para efeitos complexos:

```
.custom-enter-active,
.custom-leave-active {
    transition: opacity 0.5s;
    animation: bounce 1s;
}

.custom-enter,
.custom-leave-to {
    opacity: 0;
}

@keyframes bounce {

    0%,
```

```
    100% {
        transform: translateY(0);
    }

    50% {
        transform: translateY(-20px);
    }
}
```

Usando v-show

O `v-show` pode ser usado para alternar a visibilidade de um elemento sem removê-lo do DOM:

```
<transition name="fade">
    <p v-show="visible">Olá, VueJS!</p>
</transition>
```

Configurando Animação no Carregamento do Componente

Use transições durante o carregamento de componentes:

```
<transition name="fade" mode="out-in">
    <component :is="currentComponent"></component>
</transition>
```

Usando Nomes Diferentes de Classes CSS

Você pode personalizar os nomes das classes CSS:

```
<transition        enter-active-class="animate__animated
animate__fadeIn"
                   leave-active-class="animate__animated
animate__fadeOut">
    <p v-if="show">Olá, VueJS!</p>
</transition>
```

Usando Nomes e Atributos Dinâmicos

```
<transition :name="transitionName">
    <p v-if="show">Olá, VueJS!</p>
</transition>
```

```
data() {
    return {
        transitionName: 'fade'
    };
}
```

Transicionar entre Múltiplos Elementos

```
<transition name="fade" mode="out-in">
    <p v-if="show">Olá, VueJS!</p>
    <h1 v-else>Bem-vindo!</h1>
</transition>
```

Escutando a Eventos de Transição (Hooks)

Adicione hooks para eventos de transição:

```
<transition @before-enter="beforeEnter" @enter="enter"
@after-enter="afterEnter">
    <p v-if="show">Olá, VueJS!</p>
</transition>
```

Entendendo Animação em JavaScript

Use JavaScript para manipular transições diretamente:

```
<transition @enter="enter">
    <p v-if="show">Olá, VueJS!</p>
</transition>
```

```
methods: {
    enter(el, done) {
        // Animação JavaScript
        done();
    }
}
```

Excluindo CSS da Animação

Desabilite CSS para controlar animações via JavaScript:

```
<transition :css="false" @enter="enter">
    <p v-if="show">Olá, VueJS!</p>
</transition>
```

Criando Animações em JavaScript

```
methods: {
    enter(el, done) {
        // Definir animação
        done();
    }
}
```

Animando Componentes Dinâmicos

```
<transition name="fade" mode="out-in">
    <component :is="currentComponent"></component>
</transition>
```

Animando Listas com `transition-group`

Use `transition-group` para animar listas:

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

```
.list-enter-active,
.list-leave-active {
    transition: all 0.5s;
}

.list-enter,
.list-leave-to {
    opacity: 0;
    transform: translateY(30px);
}
```

Usando `transition-group` - Preparações

Certifique-se de que cada item da lista tenha uma chave (`key`) única para animação adequada.

Usando `transition-group` para Animar Listas

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

Seção: Conectando com Servidor via HTTP

Configurações do Firebase

Configure o Firebase para usar como backend:

```
import firebase from 'firebase/app';
import 'firebase/database';

const firebaseConfig = {
```

```
    apiKey: "API_KEY",
    authDomain: "PROJECT_ID.firebaseapp.com",
    databaseURL: "https://PROJECT_ID.firebaseio.com",
    projectId: "PROJECT_ID",
    storageBucket: "PROJECT_ID.appspot.com",
    messagingSenderId: "SENDER_ID",
    appId: "APP_ID"
};

firebase.initializeApp(firebaseConfig);
```

Configuração Global do Axios

Configure Axios globalmente:

```
import axios from 'axios';

axios.defaults.baseURL =
'https://your-api-base-url.com';
```

Criando instância do Axios

```
const instance = axios.create({
    baseURL: 'https://your-api-base-url.com'
});
```

Criando formulário

Crie um formulário em seu componente para enviar dados:

```
<form @submit.prevent="submitForm">
          <input   v-model="formData.name"   type="text"
placeholder="Nome">
         <input   v-model="formData.email"   type="email"
placeholder="Email">
    <button type="submit">Enviar</button>
```

```
</form>
```

Enviando POST

Envie dados para o servidor:

```
methods: {
    submitForm() {
        axios.post('/form', this.formData)
            .then(response => {
                console.log(response);
            })
            .catch(error => {
                console.error(error);
            });
    }
}
```

Enviando GET

Recupere dados do servidor:

```
methods: {
    fetchData() {
        axios.get('/data')
            .then(response => {
                this.data = response.data;
            })
            .catch(error => {
                console.error(error);
            });
    }
}
```

Usando Axios Localmente

Use instâncias locais de Axios para diferentes endpoints:

```
const localInstance = axios.create({
    baseURL: 'https://local-api-endpoint.com'
});
```

Interceptando Requisições

Adicione interceptores para requisições:

```
axios.interceptors.request.use(config => {
    console.log('Request Intercepted:', config);
    return config;
});
```

Interceptando Respostas

Adicione interceptores para respostas:

```
axios.interceptors.response.use(response => {
    console.log('Response Intercepted:', response);
    return response;
});
```

Adicionando Headers Globais

Configure headers globais:

```
axios.defaults.headers.common['Authorization'] =
'Bearer token';
```

Correção de problema com o método DELETE do CRUD

Garanta que o método DELETE seja corretamente configurado:

```
methods: {
```

```
    deleteItem(id) {
        axios.delete(`/items/${id}`)
            .then(response => {
                console.log(response);
            })
            .catch(error => {
                console.error(error);
            });
    }
}
```

Implementando CRUD

Implemente todas as operações CRUD:

```
methods: {
    createItem() {
        axios.post('/items', this.newItem)
            .then(response => {
                this.items.push(response.data);
            })
            .catch(error => {
                console.error(error);
            });
    },
    readItems() {
        axios.get('/items')
            .then(response => {
                this.items = response.data;
            })
            .catch(error => {
                console.error(error);
            });
    },
    updateItem(item) {
```

```
        axios.put(`/items/${item.id}`, item)
            .then(response => {
                console.log(response);
            })
            .catch(error => {
                console.error(error);
            });
    },
    deleteItem(id) {
        axios.delete(`/items/${id}`)
            .then(response => {
                 this.items = this.items.filter(item =>
item.id !== id);
            })
            .catch(error => {
                console.error(error);
            });
    }
}
```

Exibindo Mensagens

Use mensagens de sucesso e erro para informar o usuário:

```
methods: {
    submitForm() {
        axios.post('/form', this.formData)
            .then(response => {
                    this.$emit('success', 'Formulário
enviado com sucesso!');
            })
            .catch(error => {
                    this.$emit('error', 'Erro ao enviar
formulário.');
            });
```

```
    }
}
```

Seção: Rotas em uma Aplicação VueJS

Introdução ao Vue Router

Vue Router é a biblioteca oficial de roteamento para VueJS, permitindo a criação de rotas e navegação entre diferentes vistas em uma aplicação Vue.

Instalação do Vue Router

Instale o Vue Router:

```
yarn add vue-router@3
```

Configuração do Vue Router

Crie a pasta padrão do Vue Route, à **views** os componentes Home e About:

```
<template lang="pt-br">
    <div>
        <h1>About</h1>
        <p>Hello pasta About</p>
    </div>
</template>
<script>
export default {

}
</script>
<style lang="">

</style>
```

Seguindo com o componente Home:

```
<template lang="">
    <div>
```

```
        <h1>Home</h1>

    </div>
</template>
<script>
export default {
}
</script>
<style lang="">

</style>
```

Configure o router em main.js:

```
import Vue from 'vue'
import App from './App.vue'
import Home from './views/HomePage.vue';
import About from './views/AboutPage.vue';
import Router from 'vue-router';

Vue.use(Router);

const router = new Router({
  mode: 'history',
  routes: [
    { path: '/', component: Home },
    { path: '/about', component: About }
  ]
});

Vue.config.productionTip = false
new Vue({
  router,
  render: h => h(App),
```

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

Entendendo Modos de Rotas (Hash vs History)

O Vue Router suporta modos de navegação `hash` e `history`. O modo `hash` usa um `#` na URL, enquanto o `history` utiliza a API History do HTML5 para URLs limpas.

Navegando com Router Links

Use `<router-link>` para navegar entre rotas:

```
<router-link to="/">Home</router-link>
<router-link to="/about">About</router-link>
<router-view></router-view>
```

Estilizando o Link Ativo

O Vue Router aplica uma classe CSS `router-link-active` aos links ativos, que pode ser personalizada:

```
.router-link-active {
    font-weight: bold;
}
```

Navegação via Código (Navegação Imperativa)

Navegue programaticamente usando o método `$router.push`:

```
methods: {
    goToAbout() {
        this.$router.push('/about');
    }
}
```

Configurando Parâmetros de Rotas

Defina parâmetros de rota usando `:`

```
const router = new Router({
    routes: [
        { path: '/user/:id', component: User }
    ]
});
```

Lendo e Usando Parâmetros de Rota

Acesse parâmetros de rota via `$route.params`:

```
created() {
    console.log(this.$route.params.id);
}
```

Reagindo a Mudanças em Parâmetros de Rotas

Reaja às mudanças de parâmetros de rota:

```
watch: {
    '$route'(to, from) {
        console.log(to.params.id);
    }
}
```

Parâmetros de Rotas via ‘props’

Passe parâmetros de rota como props para o componente:

```
const router = new Router({
    routes: [
        { path: '/user/:id', component: User, props:
true }
    ]
});
```

Configurando Rotas Filhas (Rotas Aninhadas)

Defina rotas aninhadas:

```
const router = new Router({
    routes: [
        {
            path: '/user/:id',
            component: User,
            children: [
                  { path: 'profile', component: Profile
},
                { path: 'posts', component: Posts }
            ]
        }
    ]
});
```

Navegando para Rotas Aninhadas

Use `router-link` para rotas aninhadas:

```
<router-link :to="{ name: 'profile', params: { id:
userId }}">Perfil</router-link>
```

Tornando Router Links mais Dinâmico

Construa links dinamicamente:

```
<router-link :to="'/user/' +
userId">Usuário</router-link>
```

Criando Links com Rotas Nomeadas

Nomeie rotas para facilitar a navegação:

```
const router = new Router({
    routes: [
```

```
        { path: '/user/:id', name: 'user', component:
User }
    ]
});
```

Usando Parâmetros da Query

Passe parâmetros de query:

```
<router-link :to="{ path: '/search', query: { q:
searchQuery }}">Buscar</router-link>
```

Múltiplos Router Views (Router Views Nomeados)

Use múltiplos router views:

```
const router = new Router({
    routes: [
        {
            path: '/user/:id',
            components: {
                default: User,
                sidebar: Sidebar
            }
        }
    ]
});
```

Redirecionamento

Configure redirecionamentos:

```
const router = new Router({
    routes: [
        { path: '/home', redirect: '/' }
    ]
```

```
});
```

Configurando Rota "Pega Tudo"

Capture todas as rotas não definidas:

```
const router = new Router({
    routes: [
        { path: '*', component: NotFound }
    ]
});
```

Animando Transições de Rotas

Adicione transições entre rotas:

```
<transition name="fade" mode="out-in">
    <router-view></router-view>
</transition>
```

Passando Fragmento Hash

Navegue para um fragmento hash:

```
this.$router.push({ path: '/home', hash: '#section1'
});
```

Controlando o Comportamento de Rolagem (Scroll)

Controle o comportamento de rolagem:

```
const router = new Router({
    routes: [...],
    scrollBehavior(to, from, savedPosition) {
        if (savedPosition) {
            return savedPosition;
        } else {
            return { x: 0, y: 0 };
        }
```

```
    }
});
```

Protegendo Rotas

Proteja rotas com guardas de navegação:

```
router.beforeEach((to, from, next) => {
                    if     (to.matched.some(record     =>
record.meta.requiresAuth)) {
        if (!isLoggedIn()) {
            next({ path: '/login' });
        } else {
            next();
        }
    } else {
        next();
    }
});
```

Usando o Evento "beforeEnter”

Use guardas de rota:

```
const router = new Router({
    routes: [
        {
            path: '/user/:id', component: User,
beforeEnter: (to, from, next) => {
                if (isValidUser(to.params.id)) {
                    next();
                } else {
                    next(false);
                }
            }
        }
```

```
    ]
});
```

Usando o Evento "beforeLeave”

Proteja a navegação fora do componente:

```
export default {
    beforeRouteLeave(to, from, next) {
          if (window.confirm('Deseja sair sem salvar?'))
{
             next();
         } else {
             next(false);
         }
    }
};
```

Carregando Rotas Tardiamente

Carregue componentes de rota sob demanda:

```
const User = () => import('./views/User.vue');

const router = new Router({
    routes: [
         { path: '/user/:id', component: User }
    ]
});
```

Essas seções cobrem a criação de diretivas personalizadas, utilização de filtros e mixins, implementação de animações e transições, conexão com servidores via HTTP, e configuração de rotas em uma aplicação VueJS. Boa sorte com seus estudos e implementação!

Seção: Gerenciamento de Estado com Vuex

Introdução ao Vuex

Vuex é uma biblioteca de gerenciamento de estado para aplicações VueJS. Ele permite gerenciar o estado de forma centralizada, facilitando a comunicação entre componentes e a manutenção do estado.

Por que usar um Gerenciador de Estado

Gerencie estados compartilhados entre componentes com Vuex, especialmente em aplicações de grande escala.

Entendendo "Estado Centralizado"

Centralize o estado da aplicação em um único local, facilitando o gerenciamento e a rastreabilidade.

Como Funciona o Vuex?

Vuex utiliza um estado centralizado (store) que pode ser acessado e manipulado por qualquer componente da aplicação. Ele segue um padrão de arquitetura unidirecional, onde as ações disparam mudanças no estado através de mutações.

Instalando o Vuex

Instale o Vuex em um projeto Vue CLI:

```
npm install vuex
```

Estrutura de um Store Vuex

Um store Vuex é composto por estado, getters, mutations e actions. Exemplo:

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

const store = new Vuex.Store({
    state: {
        count: 0
    },
    getters: {
        doubleCount: state => state.count * 2
    },
```

```
    mutations: {
        increment(state) {
            state.count++;
        }
    },
    actions: {
        increment({ commit }) {
            commit('increment');
        }
    }
});
```

Integrando Vuex com Vue

Para integrar Vuex com Vue, importe o store no arquivo principal da aplicação e forneça-o à instância Vue.

```
new Vue({
    el: '#app',
    store,
    render: h => h(App)
});
```

Usando Estado do Vuex em Componentes

Acesse o estado do Vuex diretamente nos componentes usando `this.$store.state`.

```
computed: {
    count() {
        return this.$store.state.count;
    }
}
```

Por que Estado Centralizado Sozinho Não Resolve

Entenda que apenas centralizar o estado não resolve todos os problemas, e é necessário uma arquitetura de gerenciador de estado robusta.

Entendendo Getters

Getters são usados para derivar dados do estado e são acessíveis como propriedades computadas. Use getters para derivar estados a partir do estado centralizado:

```
const store = new Vuex.Store({
    state: {
        todos: [
            { id: 1, text: 'Aprender Vue', done: true
},
            { id: 2, text: 'Aprender Vuex', done: false
}
        ]
    },
    getters: {
        doneTodos: state => {
            return state.todos.filter(todo =>
todo.done);
        }
    }
});
```

Usando Getters

Acesse getters nos componentes:

```
computed: {
    doneTodos() {
        return this.$store.getters.doneTodos;
    }
}
```

Mapeando Getters para Propriedades

Use `mapGetters` para mapear getters para propriedades do componente:

```
import { mapGetters } from 'vuex';

export default {
    computed: {
        ...mapGetters(['doneTodos'])
    }
}
```

Observação sobre o funcionamento da aplicação

Entenda como os getters se comportam reativamente, atualizando automaticamente quando o estado muda.

Modificando Estado com Mutations

Mutations são responsáveis por alterar o estado de maneira síncrona:

```
const store = new Vuex.Store({
    state: {
        count: 0
    },
    mutations: {
        increment(state) {
            state.count++;
        }
    }
});
```

Usando Mutations

Dispare mutations a partir dos componentes:

```
methods: {
    increment() {
        this.$store.commit('increment');
    }
}
```

Por que existem Mutations e Actions?

Mutations são síncronas e simples, enquanto actions permitem operações assíncronas e complexas.

Como Actions Complementam as Mutations

Actions disparam mutations e podem conter lógica assíncrona:

```
const store = new Vuex.Store({
    state: {
        count: 0
    },
    mutations: {
        increment(state) {
            state.count++;
        }
    },
    actions: {
        incrementAsync({ commit }) {
            setTimeout(() => {
                commit('increment');
            }, 1000);
        }
    }
});
```

Usando Actions

Dispare actions a partir dos componentes:

```
methods: {
    incrementAsync() {
        this.$store.dispatch('incrementAsync');
    }
}
```

Mapeando Actions para Métodos

Use `mapActions` para mapear actions para métodos do componente:

```
import { mapActions } from 'vuex';

export default {
    methods: {
        ...mapActions(['incrementAsync'])
    }
}
```

Vuex e Two-Way-Binding (`v-model`)

Integre Vuex com `v-model` para ligação bidirecional:

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

```
computed: {
    count: {
        get() {
            return this.$store.state.count;
        },
        set(value) {
            this.$store.commit('setCount', value);
        }
    }
}
```

Resumo do Vuex

Revise os conceitos e práticas essenciais do Vuex, incluindo estado, getters, mutations e actions.

Melhorando a Estrutura de Pastas

Organize melhor seu projeto Vuex, separando o estado em módulos.

Modularizando o Gerenciador de Estado

Divida o estado centralizado em módulos para facilitar o gerenciamento:

```
const moduleA = {
    state: () => ({ ... }),
    mutations: { ... },
    actions: { ... },
    getters: { ... }
};

const store = new Vuex.Store({
    modules: {
        a: moduleA
    }
});
```

Usando Arquivos Separados

Separe a lógica Vuex em diferentes arquivos para melhorar a manutenção e organização.

Usando Namespaces para Evitar Conflitos de Nomes

Utilize namespaces em módulos Vuex para evitar conflitos de nomes:

```
const moduleA = {
    namespaced: true,
    state: { ... },
    getters: { ... },
    actions: { ... },
    mutations: { ... }
};
```