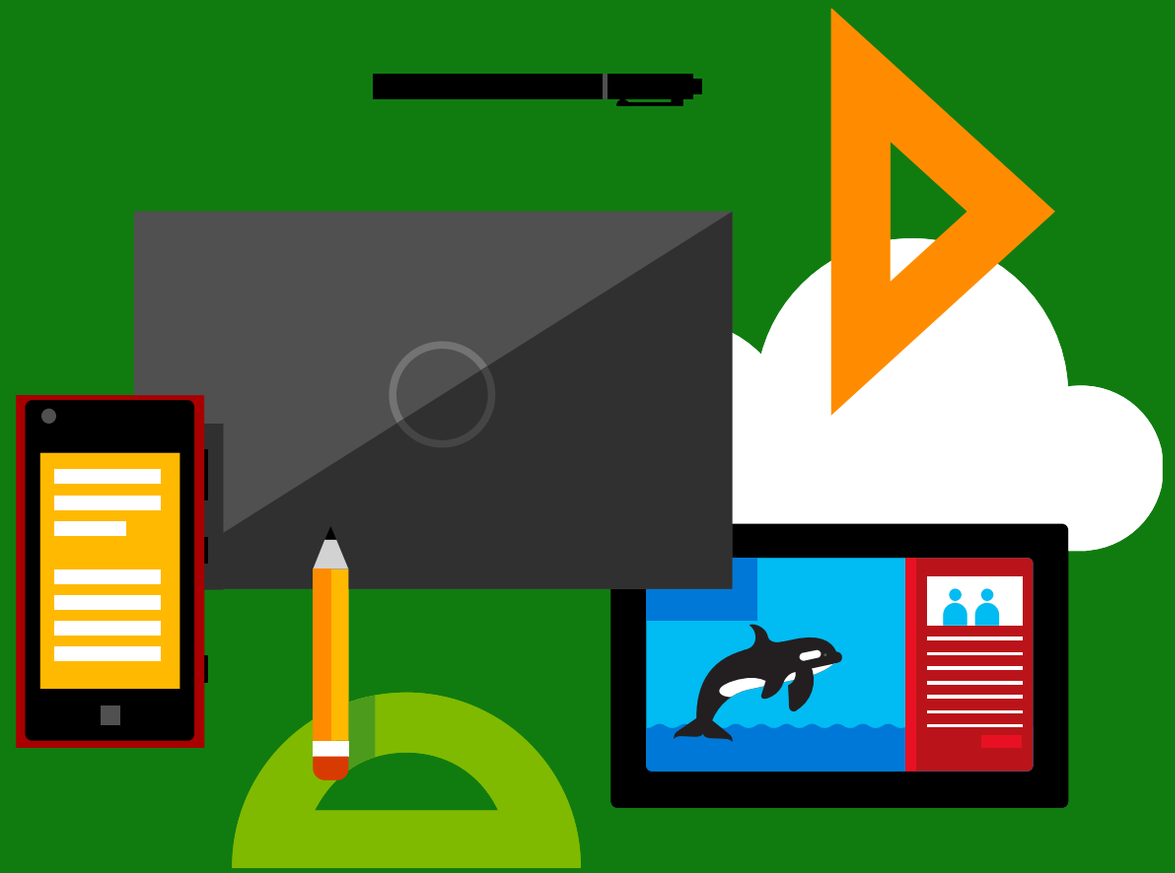
Desenvolvimento Web

HTML, CSS e Vue.js

Instrutor: Júlio Pereira Machado (julio.machado@pucrs.br)

Vue.js (continuação)



Props

- Props são atributos customizados utilizados para passar dados para um componente
- Para definir os atributos dentro de <script setup> utiliza-se a macro *defineProps()*
  - Não necessita ser importada
- CUIDADO:
  - Props são de vinculação *one-way-down*, ou seja, somente de leitura e são atualizadas pelo componente pai que passa os valores via atributos para o componente filho

Props

```
<script setup>
defineProps(['title'])
</script>

<template>
  <h4>{{ title }}</h4>
</template>
```

Arquivo “BlogPost.vue”

```
<script setup>
import { ref } from 'vue'
import BlogPost from './BlogPost.vue'
const posts = ref([
  { id: 1, title: 'My journey with Vue' },
  { id: 2, title: 'Blogging with Vue' },
  { id: 3, title: 'Why Vue is so fun' }
])
</script>

<template>
    <BlogPost
      v-for="post in posts"
      :key="post.id"
      :title="post.title"
    ></BlogPost>
</template>
```

Props

```
<script setup lang="ts">
const props = defineProps<{
  foo: string
  bar?: number
}>()
</script>
```

```
<script setup lang="ts">
interface Props {
  foo: string
  bar?: number
}

const props = defineProps<Props>()
</script>
```

Props

Number

```template
<!-- Even though `42` is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.       -->
<BlogPost :likes="42" />

<!-- Dynamically assign to the value of a variable. -->
<BlogPost :likes="post.likes" />
```

Props

Boolean

```
<!-- Including the prop with no value will imply `true`. -->
<BlogPost is-published />

<!-- Even though `false` is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.          -->
<BlogPost :is-published="false" />

<!-- Dynamically assign to the value of a variable. -->
<BlogPost :is-published="post.isPublished" />
```

Props

Array

```template
<!-- Even though the array is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.            -->
<BlogPost :comment-ids="[234, 266, 273]" />

<!-- Dynamically assign to the value of a variable. -->
<BlogPost :comment-ids="post.commentIds" />
```

Props

Object

template

```
<!-- Even though the object is static, we need v-bind to tell Vue that -->
<!-- this is a JavaScript expression rather than a string.              -->
<BlogPost
  :author="{
    name: 'Veronica',
    company: 'Veridian Dynamics'
  }"
 />

<!-- Dynamically assign to the value of a variable. -->
<BlogPost :author="post.author" />
```

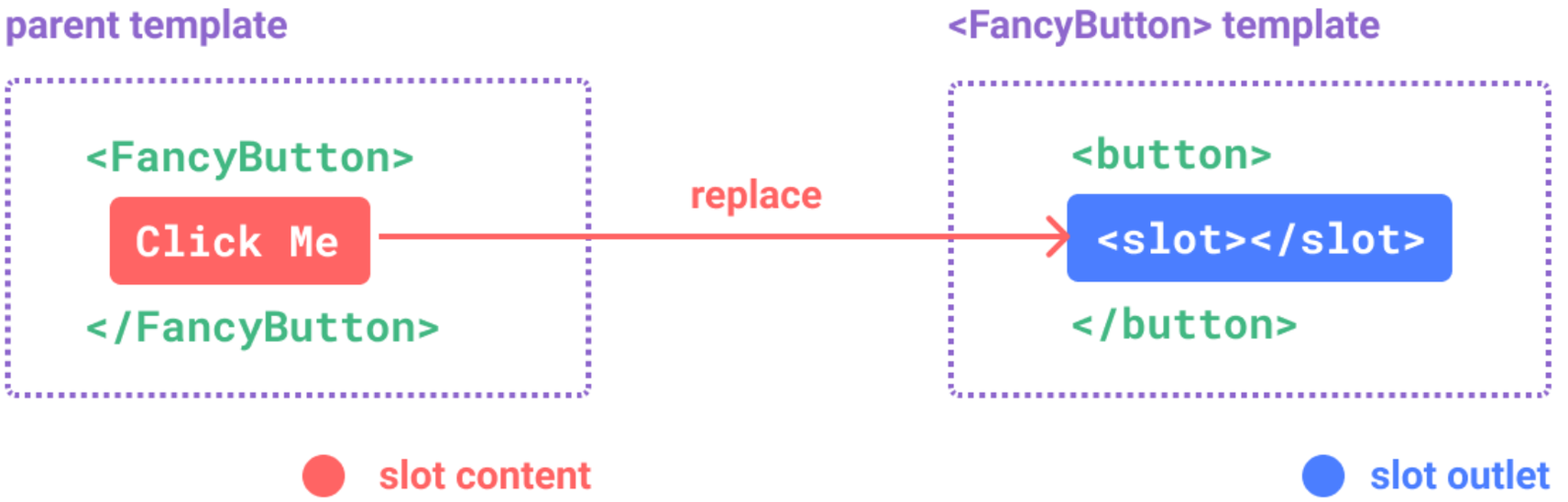
Props

- Para definir valores padrão para propriedades tipadas, utiliza-se a macro *withDefaultValues()*

```
export interface Props {
  msg?: string
  labels?: string[]
}

const props = withDefaults(defineProps<Props>(), {
  msg: 'hello',
  labels: () => ['one', 'two']
})
```

Slots

- Componentes podem receber um fragmento de template para ser renderizado através do elemento *<slot>*



Slots

- Slots podem definir um valor padrão caso não recebam o conteúdo

```
<button type="submit">
  <slot>
    Submit
  </slot>
</button>
```

Arquivo “SubmitButton.vue”

```
<SubmitButton />
```

```
<SubmitButton>Save</SubmitButton>
```

Slots

- CUIDADO:
  - As expressões no template pai só têm acesso ao escopo pai
  - As expressões no template filho só têm acesso ao escopo filho

```
<template>
  <slot>Fallback content</slot>
</template>
```

Arquivo “ChildComp.vue”

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

const msg = ref('from parent')
</script>

<template>
  <ChildComp>Message: {{ msg }}</ChildComp>
</template>
```

Slots

- Componente pode definir vários slots nomeados com o atributo *name*

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot> <!-- nome é “default” -->
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

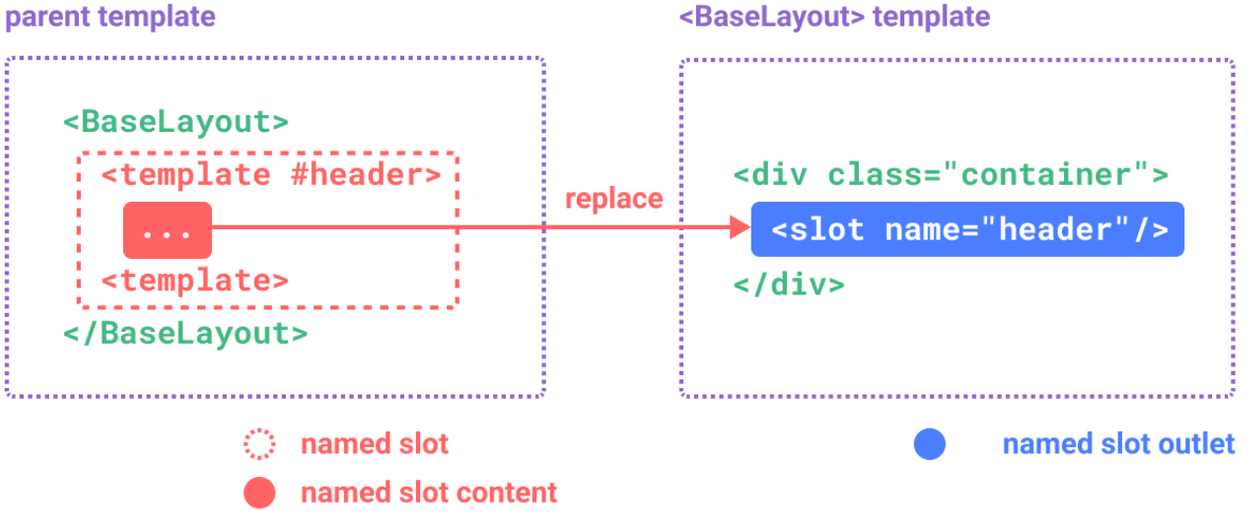
Arquivo “BaseLayout.vue”

Slots

- Para passar fragmentos para um slot nomeado utiliza-se a diretiva *v-slot* dentro do elemento *<template>*

```
<BaseLayout>
  <template v-slot:header>
    <!-- conteúdo -->
  </template>
</BaseLayout>
```

```
<BaseLayout>
  <template #header>
    <!-- conteúdo -->
  </template>
</BaseLayout>
```



Slots

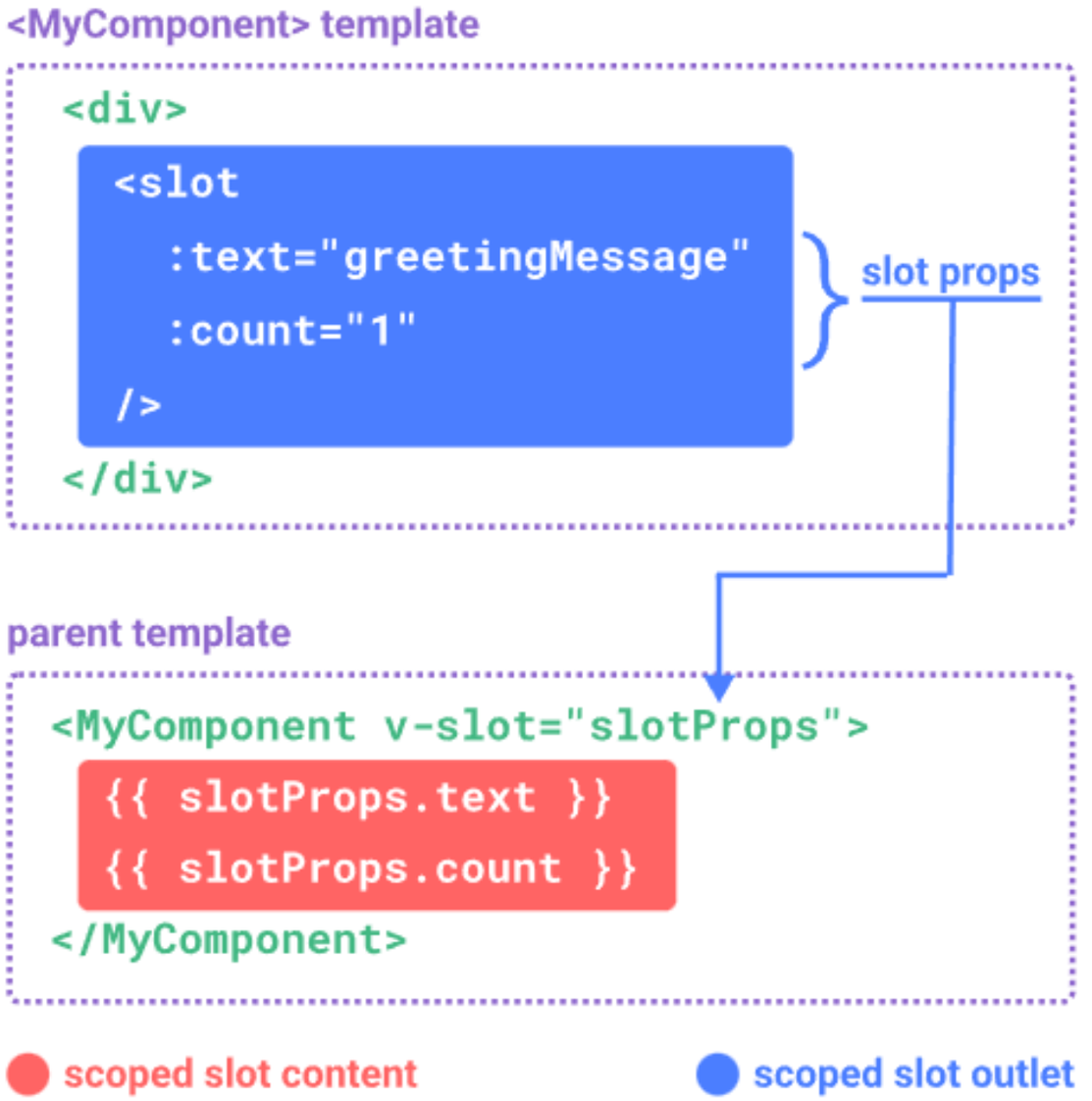
```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>  <!-- nome é “default” -->
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

Arquivo “BaseLayout.vue”

```
<BaseLayout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>

  <template #default>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </template>

  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

Slots

- Lembrando...
  - Conteúdo do slot não tem acesso ao estado do componente filho
- Para permitir o compartilhamento de dados entre diferentes escopos, slots suportam passagem de atributos



Slots

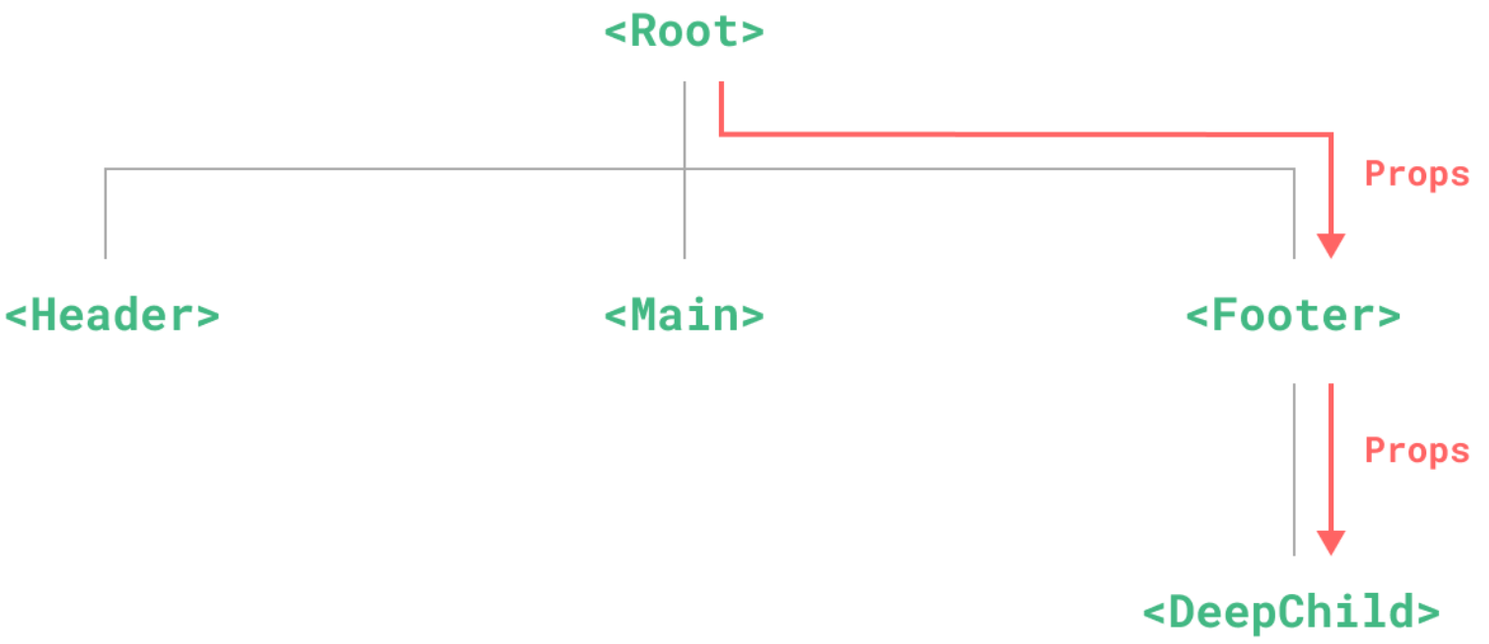
Arquivo “FancyList.vue”

```vue
<template>
  <ul>
    <li v-if="!items.length">
      Loading...
    </li>
    <li v-for="item in items">
      <slot name="item" v-bind="item"/>
    </li>
  </ul>
</template>
```

```vue
<template>
  <FancyList>
    <template #item="{ body, username, likes }">
      <div class="item">
        <p>{{ body }}</p>
        <p class="meta">by {{ username }} | {{ likes }} likes</p>
      </div>
    </template>
  </FancyList>
</template>
```

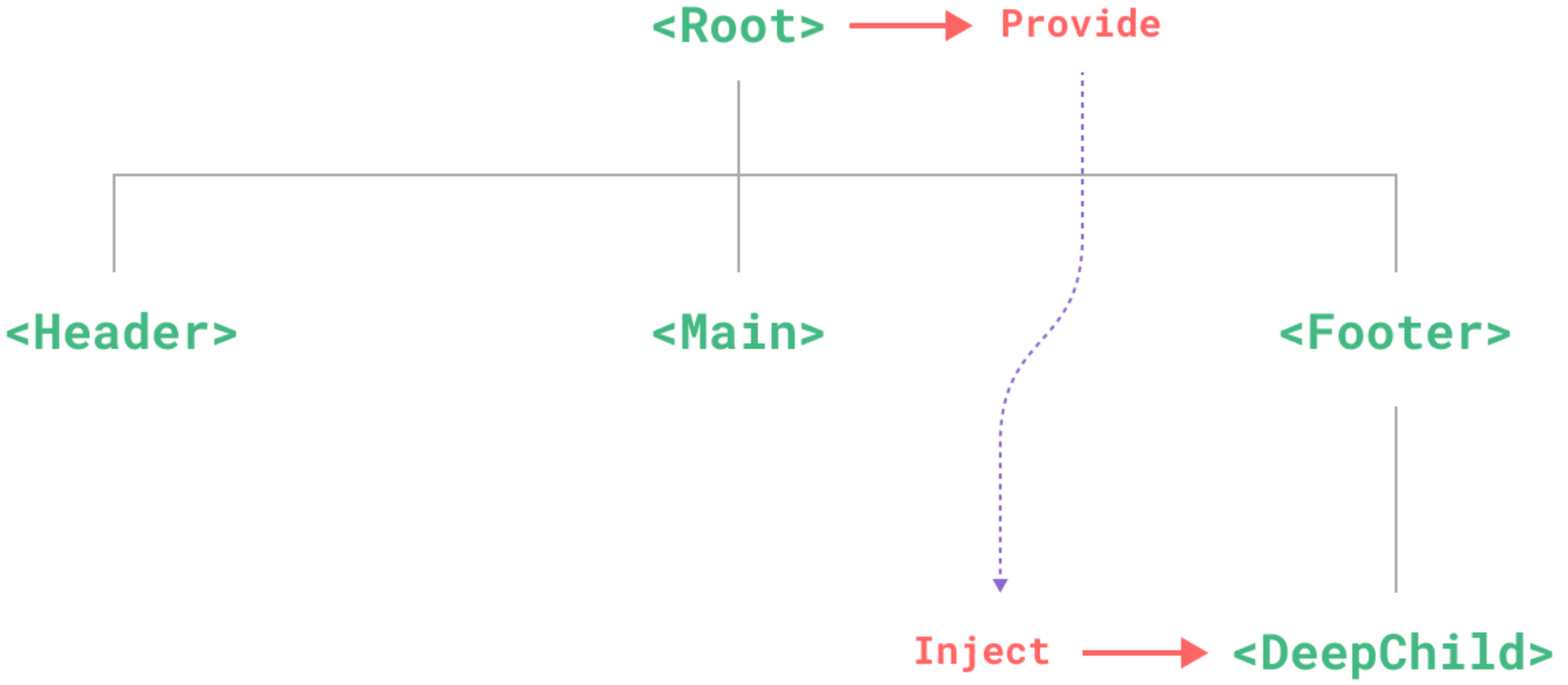
Provide e Inject

- Props são utilizados para passar dados de um componente pai para um componente filho direto
- Como fazer para passar dados para um filho indireto?



Provide e Inject

- Utilizar as funções:
  - *provide()*
  - *inject()*



Provide

- Provê dados para componentes descendentes

```
<script setup>
import { provide } from 'vue'
provide(/* chave */ 'message', /* valor */ 'hello!')
</script>
```

```
import { ref, provide } from 'vue'
const count = ref(0)
provide('key', count)
```

```
import { createApp } from 'vue'
const app = createApp({})
app.provide(/* key */ 'message', /* value */ 'hello!')
```

Inject

- Fornece o ponto de injeção do valor provido por um componente ancestral

```
<script setup>
import { inject } from 'vue'
const message = inject('message')
</script>
```

```
// `value` será o "valor padrão"
// se "message" não for provido
const value = inject('message', 'default value’)
const value = inject('message', () => new ExpensiveClass(), true)
```

Provide/Inject e Reatividade

- Mutação deve preferencialmente ser realizada no provedor dos valores
- Se torna útil prover juntamente com o valor uma função de mutação

```
<!-- component -->
<script setup>
import { provide, ref } from 'vue'

const location = ref('North Pole')

function updateLocation() {
  location.value = 'South Pole'
}

provide('location', {
  location,
  updateLocation
})
</script>inside provider
```

```
<!-- in injector component -->
<script setup>
import { inject } from 'vue'

const { location, updateLocation } = inject('location')
</script>

<template>
  <button @click="updateLocation">{{ location }}</button>
</template>
```

Eventos

- Componentes Vue.js podem gerar notificações através de eventos
- Expressões dentro do template utilizam o método *$emit()*
- CUIDADO:
  - Eventos emitidos por componentes Vue.js não sofrem *bubbling*
  - Um componente somente pode definir tratadores de eventos para componentes filhos diretos

```
<!-- MyComponent -->
<button @click="$emit('someEvent')">click me</button>
```

```
<MyComponent @some-event="callback" />
```

```
<!-- MyComponent -->
<button @click="$emit('someEvent',1)">click me</button>
```

```
<MyComponent @some-event="(n)=>{...}" />
```

Eventos

- Para definir os eventos dentro de <script setup> utiliza-se a macro *defineEmits()*
- É a forma recomendada de definir e gerar eventos

```
<script setup>
const emit = defineEmits(['inFocus', 'submit'])

function buttonClick() {
  emit('submit')
}
</script>
```

Eventos

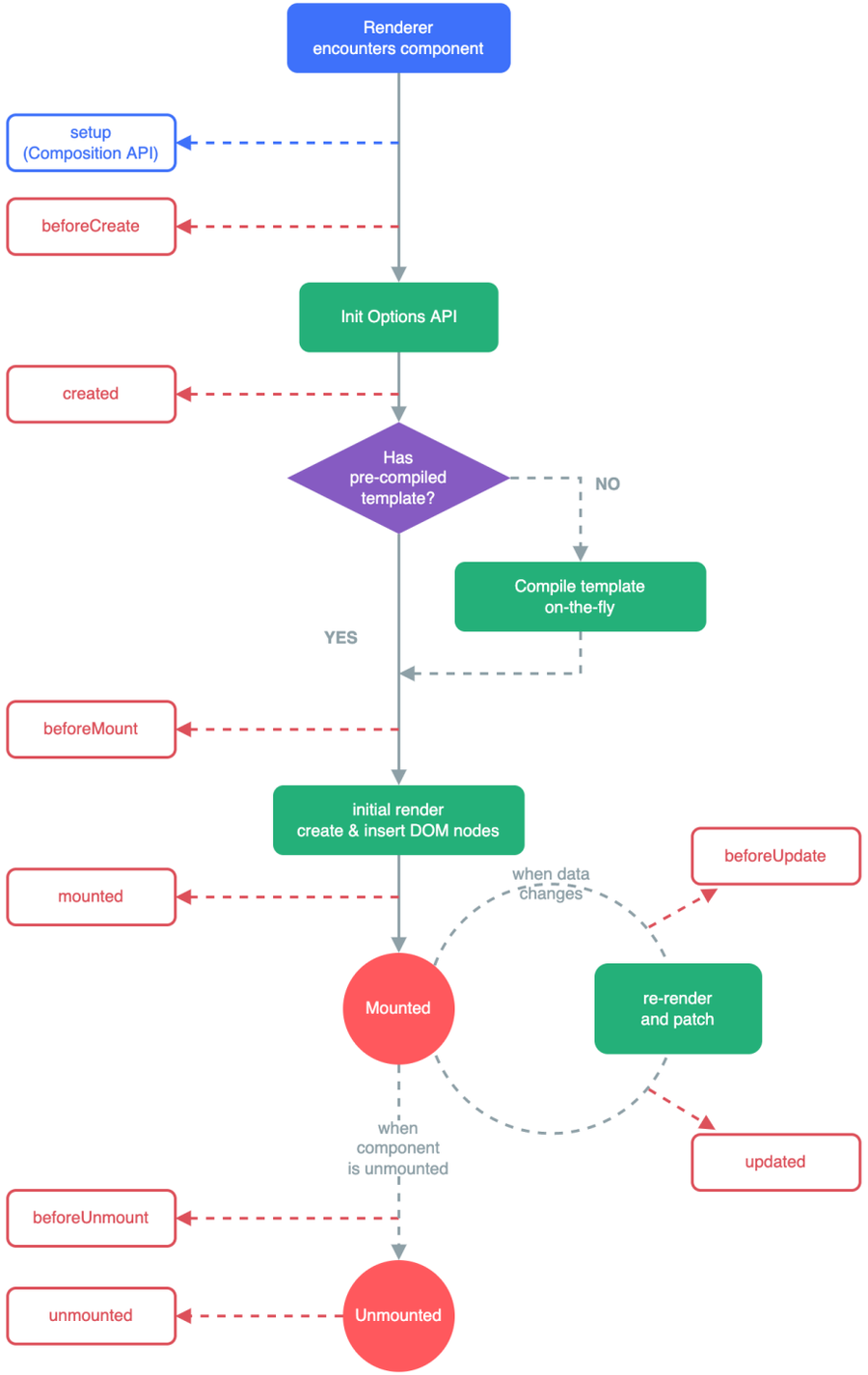
```
<script setup lang="ts">

const emit = defineEmits<{
  change: [id: number]
  update: [value: string]
}>()

</script>
```

Ciclo de Vida

- Todo componente Vue.js está sob o controle de um ciclo de vida
- Funções de hook permitem implementar ações a cada fase do ciclo de vida
- Ver: https://vuejs.org/guide/essentials/lifecycle.html

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

onMounted(() => {
  console.log(`the component is now mounted.`)
})
</script>
```

Ciclo de Vida



Efeitos Colaterais

- Vue.js suporta a execução de efeitos colaterais como resposta à mudança de estado
- São exemplos de efeitos colaterais:
  - Mutação do DOM
  - Alterar o estado com base em uma operação assíncrona (tal como consulta a web service)
  - etc
- Função *watch()* introduz a manipulação de efeitos colaterais
  - Primeiro argumento é a fonte de estado reativo (ref e computed ref, reactive, funções getter, ou array de várias fontes)
  - Segundo argumento é uma função de call-back em resposta a uma alteração de estado reativa
    - Possui três argumentos: novo valor, valor antigo, função de call-back para limpeza (chamada antes da próxima execução do efeito)

Efeitos Colaterais

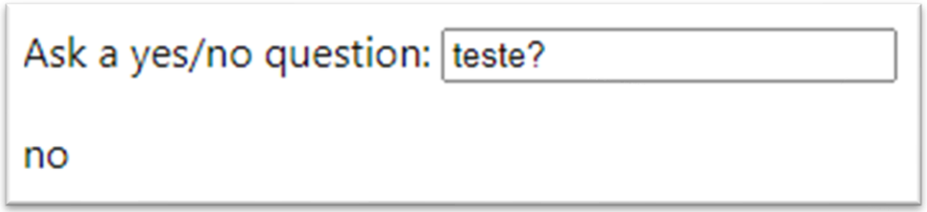
```
const x = ref(0)
const y = ref(0)

// ref
watch(x, (newX) => {
  console.log(`x is ${newX}`)
})

// getter
watch(
  () => x.value + y.value,
  (sum) => {
    console.log(`sum of x + y is: ${sum}`)
  }
)

// array
watch([x, () => y.value], ([newX, newY]) => {
  console.log(`x is ${newX} and y is ${newY}`)
})
```

```
// watching single source
function watch<T>(
  source: WatchSource<T>,
  callback: WatchCallback<T>,
  options?: WatchOptions
): StopHandle
```

```
// watching multiple sources
function watch<T>(
  sources: WatchSource<T>[],
  callback: WatchCallback<T[]>,
  options?: WatchOptions
): StopHandle
```

```
type WatchCallback<T> = (
  value: T,
  oldValue: T,
  onCleanup: (cleanupFn: () => void) => void
) => void
```

Efeitos Colaterais




```
<script setup>
import { ref, watch } from 'vue'
const question = ref('')
const answer = ref('Questions usually contain a question mark. ;-)')
watch(question, async (newQuestion) => {
  if (newQuestion.indexOf('?') > -1) {
    answer.value = 'Thinking...'
    try {
      const res = await fetch('https://yesno.wtf/api')
      answer.value = (await res.json()).answer
    } catch (error) {
      answer.value = 'Error! Could not reach the API. ' + error
    }
  }
})
</script>
```

```
<template>
  <p>
    Ask a yes/no question:
    <input v-model="question" />
  </p>
  <p>{{ answer }}</p>
</template>
```

Efeitos Colaterais

- Função *watch()* é lazy por padrão
  - O call-back não será chamado até que a fonte observada tenha alterado
- Função *watch()* pode ser configurada para ser eager
  - O call-back é executado imediatamente
  - Útil para uma busca inicial de dados, por exemplo

```
watch(source, (newValue, oldValue) => {
  // executa imediatamente, e novamente quando `source` é alterado
}, { immediate: true })
```

Efeitos Colaterais

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

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

Efeitos Colaterais

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

A função *watchEffect()* simplifica o uso de efeitos colaterais, pois realiza o gerenciamento automático das dependências reativas.

Template Refs

- Vue.js permite referências diretas a um elemento DOM através do atributo *ref*
  - Permite manipular o elemento DOM após sua adição à árvore
  - CUIDADO: o nome da referência precisar ser o mesmo utilizado no atributo

```
<script setup>
import { ref, onMounted } from 'vue'
const input = ref(null)
onMounted(() => {
  input.value.focus()
})
</script>

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