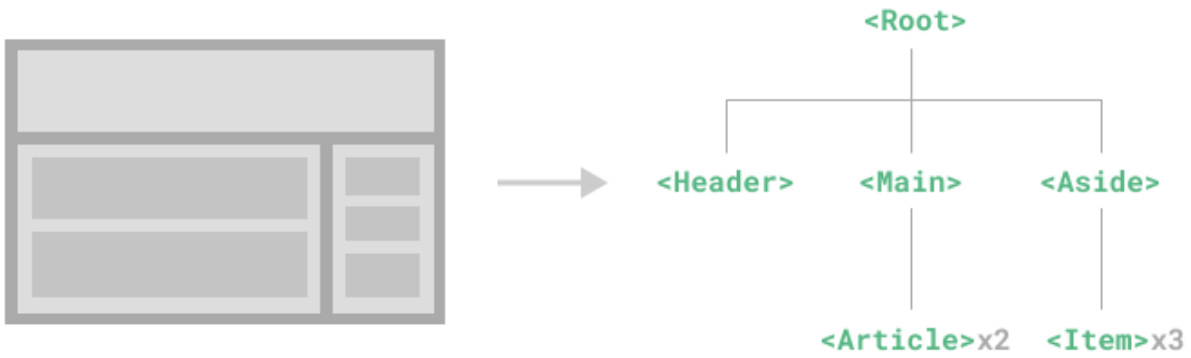
Chapter 6 – Basic Components

Asst.Prof. Dr. Umaporn Supasitthimethee
ผศ.ดร.อุมาพร สุภสิทธิเมธี

https://vuejs.org/guide/essentials/component-basics.html

Components Basics

- Components allow us to split the UI into independent and reusable pieces and think about each piece in isolation.
- It's common for an app to be organized into a tree of nested components:
- This is very similar to how we nest native HTML elements, but Vue implements its own component model that allow us to encapsulate custom content and logic in each component.



Using a Component

- To use a child component, we need to import it in the parent component.
- Assuming we placed our counter component inside a file called `ButtonCounter.vue`, the component will be exposed as the file's default export:

```
<script setup>
import ButtonCounter from './ButtonCounter.vue'
</script>
<template>
    <h1>Here is a child component!</h1>
    <ButtonCounter />
 </template>
```

- With `<script setup>`, imported components are automatically made available to the template.

Using a Component

- Components can be reused as many times as you want:

```
<h1>Here are many child components!</h1>
<ButtonCounter />
<ButtonCounter />
<ButtonCounter />
```

- In SFCs, it's recommended to use ***PascalCase*** tag names for child components to differentiate from native HTML elements.

Style Guide: Component Name Casing

- Using PascalCase names when registering components.
- PascalCase names are valid JavaScript identifiers. This makes it easier to import and register components in JavaScript. It also helps IDEs with auto-completion.
- <PascalCase /> makes it more obvious that this is a Vue component instead of a native HTML element in templates. It also differentiates Vue components from custom elements (web components).
- Vue supports resolving kebab-case tags to components registered using PascalCase. This means a component registered as MyComponent can be referenced in the template via both <MyComponent> and <my-component>.

https://vuejs.org/guide/components/registration.html#component-name-casing

Style Guide: Component

- Base components that apply app-specific styling and conventions should all begin with a specific prefix, such as `Base`, `App`, or `V`.

Good

```
components/
|- BaseButton.vue
|- BaseTable.vue
|- BaseIcon.vue
```

```
components/
|- AppButton.vue
|- AppTable.vue
|- AppIcon.vue
```

```
components/
|- VButton.vue
|- VTable.vue
|- VIcon.vue
```

Component Registration

https://vuejs.org/guide/components/registration.html#component-registration

Component Registration

- A Vue component needs to be "registered" so that Vue knows where to locate its implementation when it is encountered in a template.
- There are two ways to register components: **global** and **local**.

Global Registration

- We can make components available globally in the current Vue application using the `app.component()` method:

//main.js

```
import { createApp } from 'vue'
import App from './App.vue'
import MyComponent from './MyComponent.vue'

const app = createApp(App)

app.component(
  // the registered name (tag name),
imported component implementation
  'MyComp', MyComponent
 )
app.mount('#app')
```

//app.vue

```
<template>
  <MyComp msg="This is my world!"></MyComp>
</template>
```

Global Registration

- the `app.component()` method can be chained:

```
// the app.component() method can be chained:
app
  .component('ComponentA', ComponentA)
  .component('ComponentB', ComponentB)
  .component('ComponentC', ComponentC)
```

- Globally registered components can be used in the template of any component within this application:

```
<!-- this will work in any component inside the app -->
<ComponentA/>
<ComponentB/>
<ComponentC/>
```

Global Registration Drawbacks

- While convenient, global registration has **a few drawbacks**:
- Global registration prevents build systems from removing **unused components** (a.k.a "tree-shaking"). If you globally register a component but end up not using it anywhere in your app, it will still be included in the final bundle.
- Global registration makes dependency relationships less explicit in large applications. It makes it **difficult to locate** a child component's implementation from a parent component using it. This can affect long-term maintainability similar to using **too many global** variables.

Local Registration

- Local registration scopes the availability of the registered components to the current component only.
- It makes the dependency relationship more explicit and is more tree-shaking friendly.

Local Registration

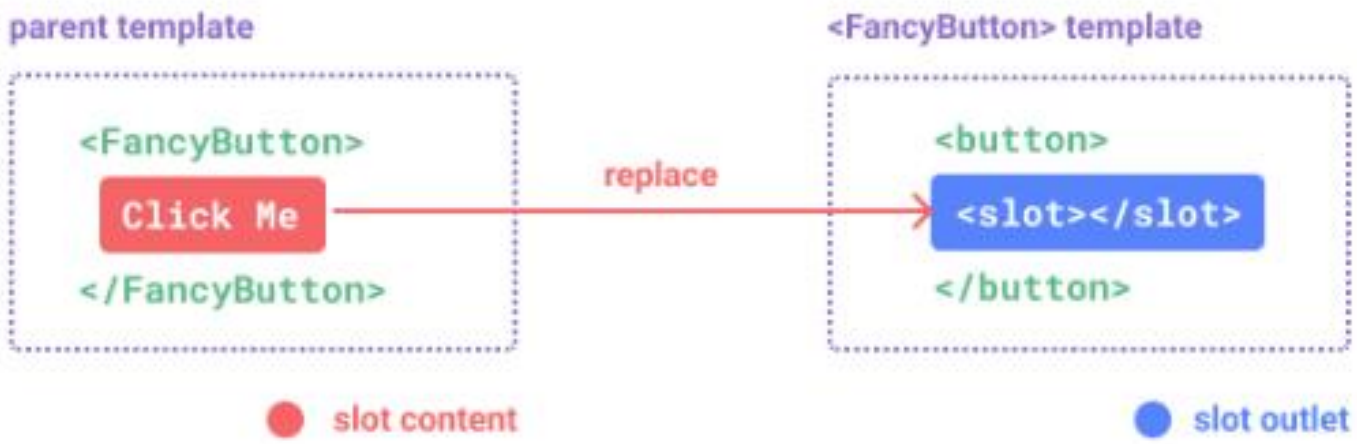
```
<script setup>
import ComponentA from './components/ComponentA.vue'
</script>

<template>
 <ComponentA />

</template>

<style></style>
```

Note that locally registered components are not also available in descendent components. In this case, ComponentA will be made available to the current component only, not any of its child or descendent components.

slots

Slot Content and Outlet

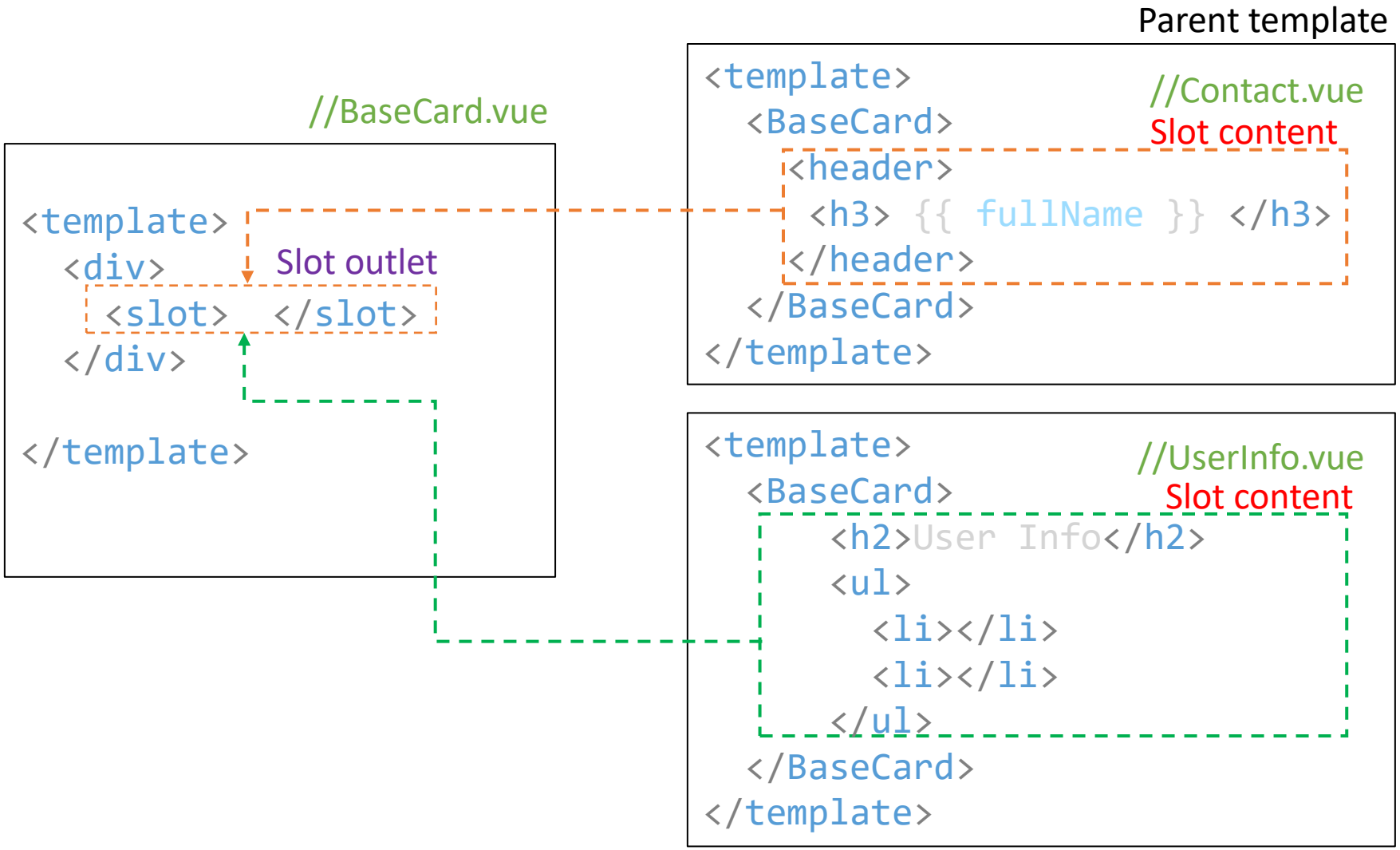
- In some cases, we may want to pass a template fragment markup to a child component and let the child component render the fragment within its own template.
- The `<slot>` element is a **slot outlet** that indicates where the parent-provided **slot content** should be rendered.



https://vuejs.org/guide/components/slots.html

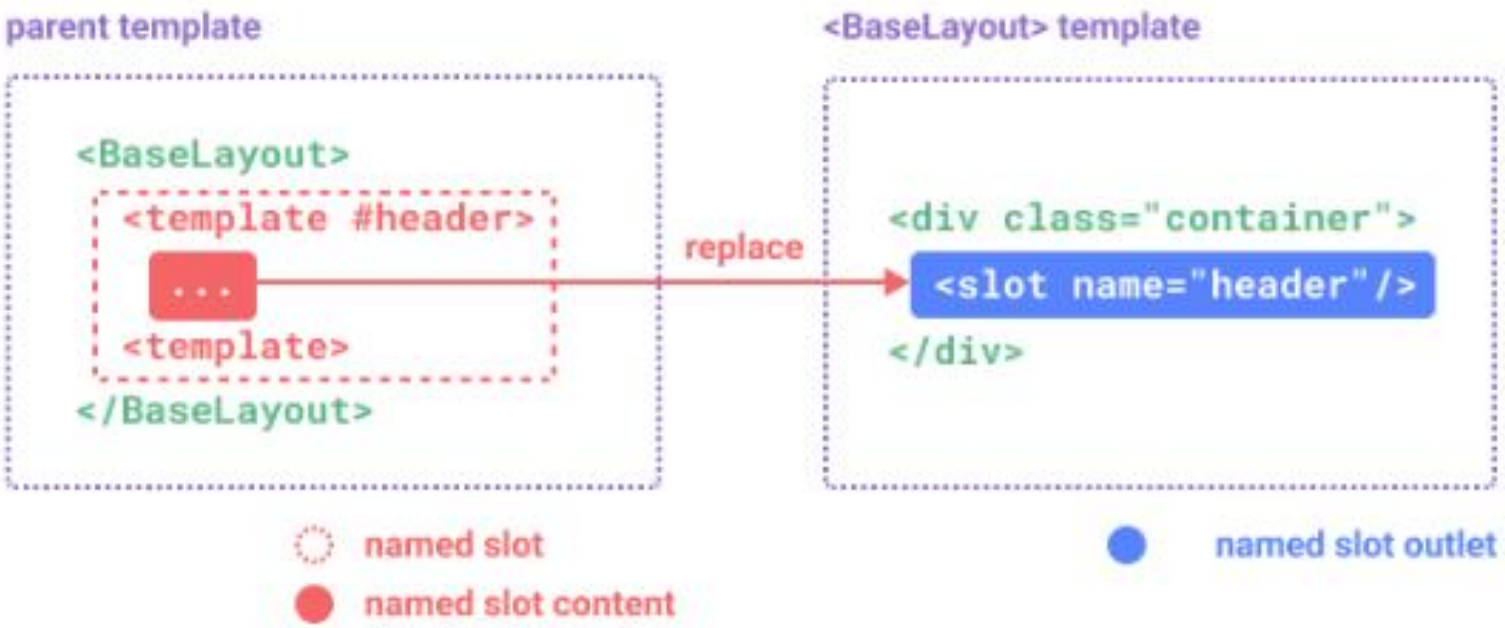
Slots

Slot allow us to receive dynamic HTML code content from outside components

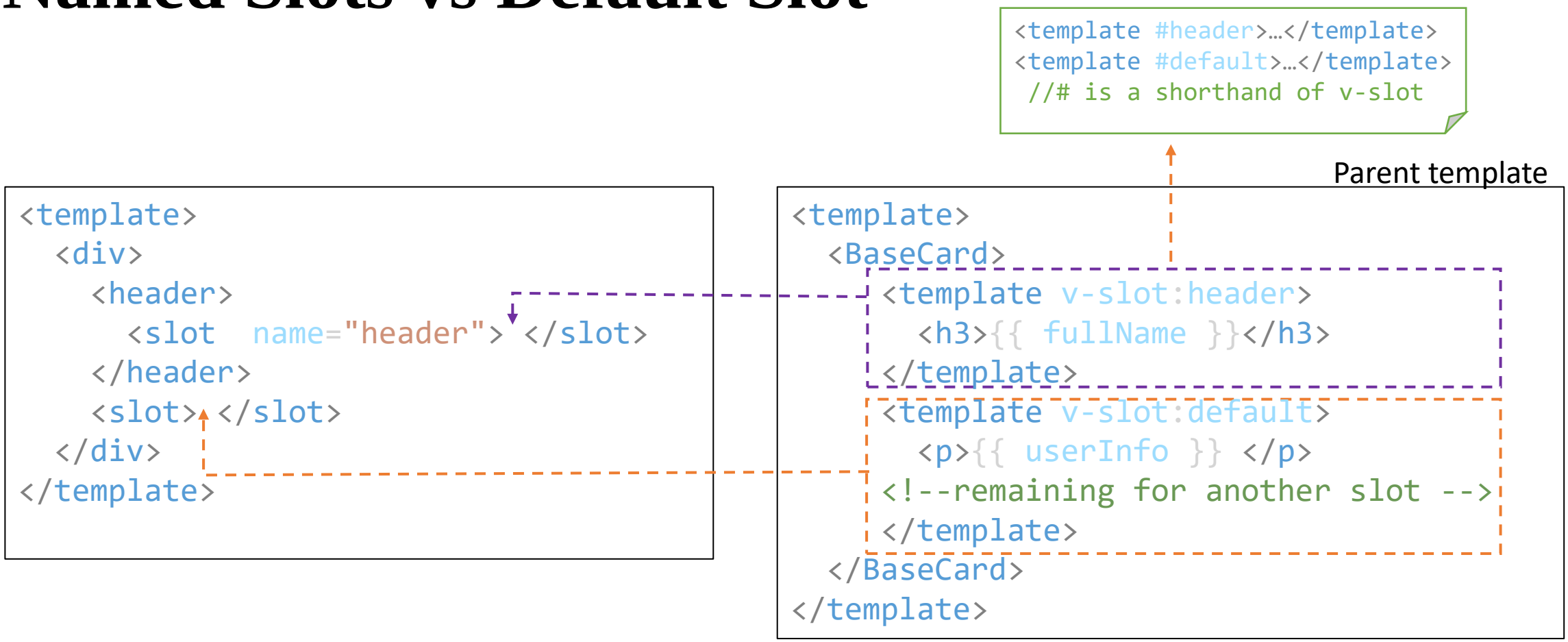


Named Slots

If you use more than slot, you can use named slot.
A <slot> outlet without name implicitly has the name "default".



Named Slots vs Default Slot



Name Slots and Default Slots

Parent template

```
<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>
```

<BaseLayout> template

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

Result

```
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

Default Fallback Content

There are cases when it's useful to specify fallback (i.e. default) content for a slot, to be rendered only when no content is provided.

Parent template

```
<template>
  <div>
    <header>
      <slot  name="header">
         <h2>The Default Header</h2>
      </slot>
    </header>
    <slot></slot>
  </div>
</template>
```

```
<template>
  <BaseCard>
    <template v-slot:default>
      <p>{{ userInfo }} </p>
    </template>
  </BaseCard>
</template>
```

Not Provide Content for header, *"The Default Header"* will receive and show here

Passing Props

Props

- **Configuration Props**: change how component behaves, for examples, disabled, compact view, outline button, number of columns
- **Data Props**: data using in component, for examples, items, products, url, source

Style Guide: `props`

- We declare long prop names **using camelCase** because this avoids having to use quotes when using them as property keys, and allows us to reference them directly in template expressions because they are valid JavaScript identifiers:
- props definitions **should be as detailed as** possible.

```
defineProps({
  greetingMessage: String
})

<span>{{ greetingMessage }}</span>
```

- Technically, you can also use camelCase **when passing props** to a child component. However, the convention is **using kebab-case** in all cases to align with HTML attributes:

```
<MyComponent greeting-message="hello" />
```

https://vuejs.org/guide/components/props.html#prop-passing-details

Props Declaration Syntax

1. Array of string

```
const props = defineProps(['errorMsg', 'randNumbers'])
```

2. Object syntax

```
const props = defineProps({
  errorMsg: String,
  randNumbers: Array
})
```

3. Object syntax with Prop validation

```
const props = defineProps({
  errorMsg: {
    type: String,
    require: true
  },
  randNumbers: {
    type: Array,
    require: true
  }
})
```

Prop Declaration

- Vue components require explicit `props` declaration so that Vue knows what external props passed to the component should be treated as fall through attributes

//App.vue

```
<script setup>
import { ref } from 'vue'
const user= ref('Umaporn')
</script>

<template>
<user-data :userName="user">Passing Data</user-data>
</template>
```

//UserData.vue

```
<script setup>
const props = defineProps({
  userName: {
    type: String,
    require: true
  }
 })
</script>
```

Static vs. Dynamic Props

props passed as static values, like in:

```
<BlogPost title="My journey with Vue" />
```

You've also seen props assigned dynamically with v-bind or its : shortcut, such as in:

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

<!-- Dynamically assign the value of a complex expression -->
<BlogPost :title="post.title + ' by ' + post.author.name" />
```

https://vuejs.org/guide/components/props.html#static-vs-dynamic-props

Passing Different Value Types

Number

```
<!-- 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" />
```

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" />
```

https://vuejs.org/guide/components/props.html#passing-different-value-types

Passing Different Value Types

Array

```
<!-- 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" />
```

Object

```
<!-- 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" />
```

Prop validation

```
defineProps({
  // Basic type check
  //  (`null` and `undefined` values will allow any type)

  propA: Number,
  // Multiple possible types
  propB: [String, Number],

  // Required string
  propC: {
    type: String,
    required: true
  },

  // Number with a default value
  propD: {
    type: Number,
    default: 100
  })
```

To specify prop validations, you can provide an object with validation requirements to the defineProps(), instead of an array of strings.

All props are optional by default, unless required: true is specified.

https://vuejs.org/guide/components/props.html#prop-validation

One-Way Data Flow

- All props form a one-way-down binding between the child property and the parent one: when the parent property updates, it will flow down to the child, but not the other way around.
- This **prevents child components from accidentally mutating the parent's state**, which can make your app's data flow harder to understand.
- In addition, every time the parent component is updated, all props in the child component will be refreshed with the latest value.
- This means you **should not attempt to mutate a prop inside a child component**. If you do, **Vue will warn you** in the console:

```
const props = defineProps(['foo'])
// ❌ warning, props are readonly!
props.foo = 'bar'
```

https://vuejs.org/guide/components/props.html#one-way-data-flow

One-Way Data Flow

```
const props = defineProps({
  errorMsg: {
    type: String,
    require: true
  }
})
```

1. The prop is used **to pass in an initial value**; the child component wants to **use it as a local data property** afterwards. In this case, it's best to define a local data property that uses the prop as its initial value

```
const displayErrorText = ref(props.errorMsg.trim().toUpperCase())
```

2. The prop is passed in **as a raw value that needs to be transformed**. In this case, it's best to define a computed property using the prop's value:

```
const displayErrorText = computed(() => props.errorMsg.trim().toUpperCase())
```

Listening to Events

Component Events: Declaring Emitted Events

- A component can explicitly declare the events it will emit using the `defineEmits()` macro:

```
<script setup>
defineEmits(['chooseMail', 'submit'])
</script>
```

https://vuejs.org/guide/components/events.html#declaring-emitted-events

Component Events: Declaring Emitted Events

- A component can emit custom events directly in template expressions (e.g. in a v-on handler) using the built-in `$emit` method:
- All extra arguments passed to `$emit()` after the event name will be forwarded to the listener.
- For example, with `$emit('foo', 1, 2, 3)` the listener function will receive three arguments.

```
<input
    type="checkbox"
    :value="mail.id"
    @click="$emit('chooseMail', mail.id, 'unread')"
  />
```

https://vuejs.org/guide/components/events.html#event-arguments

Component Events: Declaring Emitted Events

- The `$emit` method that we used in the `<template>` isn't accessible within the `<script setup>` section of a component, but `defineEmits()` returns an equivalent function that we can use instead:

```
<script setup>
  const emits = defineEmits(['submit'])
  function buttonClick() {
     emits('submit')
  }
</script>
```

https://vuejs.org/guide/components/events.html#event-arguments

Component Events: Event Arguments

- You can pass into a method using the special `$event` variable

```
<input type="text" @change="$emit('changeMe', $event.target.value)" />
```

https://vuejs.org/guide/components/events.html#event-arguments

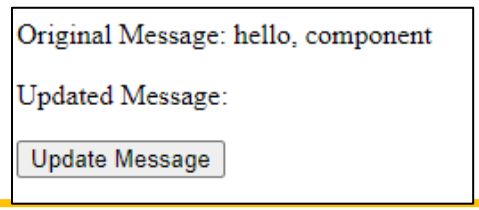
Component Events Example

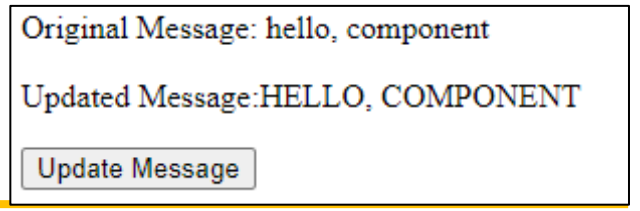
//App.vue

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

console.clear()

const msg = 'hello, component'
const newMsg = ref('')
const showUpdateMsg = ( updateMsg ) => {
  newMsg.value = updateMsg
}
</script>

<template>
  <p>Original Message: {{ msg }}</p>
  <p>Updated Message:{{ newMsg }}</p>
  <HelloWorld :msg="msg" @update="showUpdateMsg" />
</template>
```

//HelloWorld.vue

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

const emits = defineEmits(['update'])
const props = defineProps({
  msg: {
    type: String,
    require: true
  }
})
</script>
<template>
  <div>
    <button @click="$emit( 'update' , msg.toUpperCase() )">
    Update Message</button>
  </div>
</template>
```





Component Events Example

//App.vue

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

const printMessage = (msg, target, num) => {
  console.log(msg)
  console.log(target)
  console.log(num)
}
</script>
<template>
    <simple-emit @changeMe="printMessage" />
</template>
```

//SimpleEmit.vue

```vue
<script setup>
const emit = defineEmits(['changeMe'])
</script>
<template>
    <input
    type="text"
    @change="$emit( 'changeMe',
    $event.target.value, $event.target, 999
)"
  />
</template>
```

Component Events Example:
wrapping child parameters into Object and add up with parent parameters

//App.vue

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

const printMessage = ( e , status ) => {
  console.log(e.targetEValue)
  console.log(e.targetE)
  console.log(e.num)
  console.log(status)

}
</script>
<template>
    <simple-emit @changeMe="printMessage($event, "Done" />
</template>
```

//SimpleEmit.vue

```vue
<script setup>
const emit = defineEmits(['changeMe'])
</script>
<template>
    <input
    type="text"
    @change="$emit('changeMe',
         {targetEValue: $event.target.value,
         targetE:$event.target, num: 999}
    )"
  />
</template>
```

Teleport

Teleporting: built-in Vue component

- `<Teleport>` is a built-in component that allows us to "teleport" a part of a component's template into a DOM node that exists outside the DOM hierarchy of that component.
- `<Teleport>` provides a clean way to break out of the nested DOM structure.
- The `to` target of `<Teleport>` expects a CSS selector string or an actual DOM node.

```
<Teleport to="CSSSelector(ID/Class)/TagName">…</Teleport>
```

- For examples,

```
<Teleport to="#modals">
   <div>A</div>
</Teleport>
```

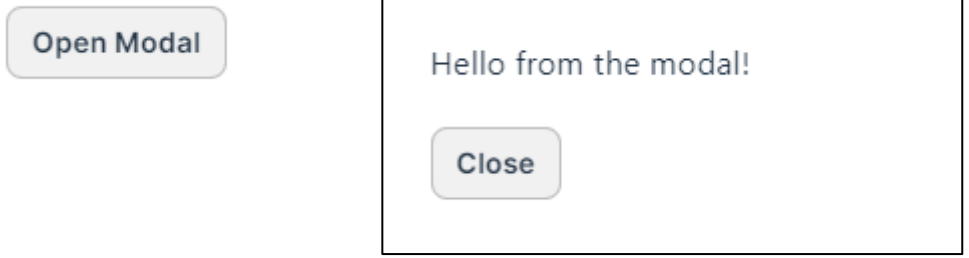
```
<Teleport to="body">
   <div>B</div>
</Teleport>
```

https://vuejs.org/guide/built-ins/teleport.html#teleport

Teleporting

```
<div class="outer">
    <h3>Vue Teleport Example</h3>
    <div>
      <MyModal />
    </div>
</div>
```

```
<html lang="en">
 ▶ <head> ⋯ </head>
 ▼ <body>
 ⋯  ▶ <div id="app" data-v-app> ⋯ </div> == $0
     <script type="module" src="/src/main.js?t=1678104127934">
     </script>
    ▶ <div data-v-381af681 class="modal"> ⋯ </div>
   </body>
</html>
```




```
<script setup>
import { ref } from 'vue'
const open = ref(false)
</script>

<template>
  <button @click="open = true">Open Modal</button>

  <Teleport to="body">
    <div v-if="open" class="modal">
      <p>Hello from the modal!</p>
      <button @click="open = false">Close</button>
    </div>
  </Teleport>

</template>
```