Components

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://about.softuni.bg

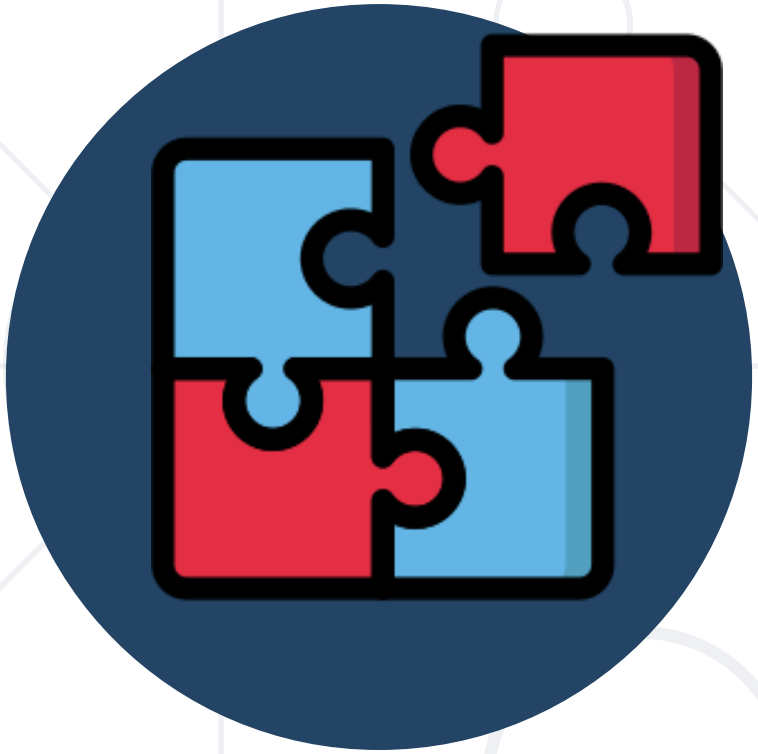
Table of Contents

Have a Question?

Software University

sli.do

#vue-js



Components Overview

Declaration, Reusability, Data

Why create components?

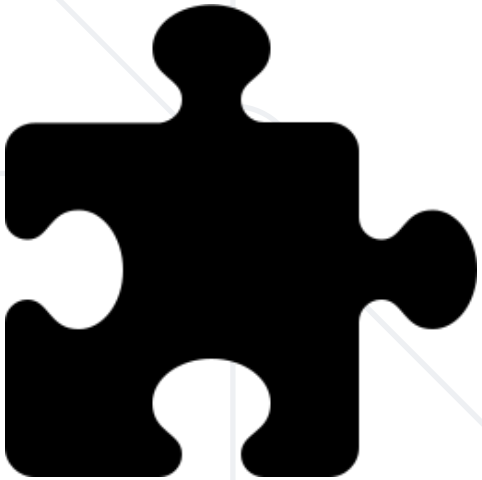
- **Reusability**
  - encapsulate UI and logic, making it easy to reuse them across your application
- **Modularity**
  - modular and organized code structure, simplifying development and maintenance
- **Isolation**
  - components have their own scoped styles and state, reducing conflicts and improving code predictability
- **Scalability**
  - facilitate the scaling of your application as it grows, by breaking it into smaller, manageable pieces

What is a Component?

- Usually, a dedicated file using the **.vue** extension
- Components are reusable Vue instances with a name that corresponds to it's tag (e.g., **<MyCustomButton></MyCustomButton>)**
- They have data, computed, methods, watch properties
- Each component is designed to be as simple as possible
- Split the application into multiple components
- Each component must be first registered
- The data must be a function which returns an object literal

Basic component - example

```
// File named ButtonCounter.vue

<script>
export default {
  data() {
    return {
      count: 0
    }
  }
}
</script>

<template>
  <button @click="count++">You clicked me {{ count }}
times.</button>
</template>
```



Using a Component - Local Registration

- To use a component, we need to import it first
- Local registration is done using the **components:{}** option
- This component will be accessible only where we import and register it!

```
<script>
import ButtonCounter from './ButtonCounter.vue'

export default {
  components: {
    ButtonCounter
  }
}
</script>

<template>
  <h1>Here is a child component!</h1>
  <ButtonCounter />
</template>
```

Using a Component - Global Registration

Software University

- Register the component in the application's root instance via **.component()**
- Globally registered components can be used in the template of any component within this application
- No manual import is needed later

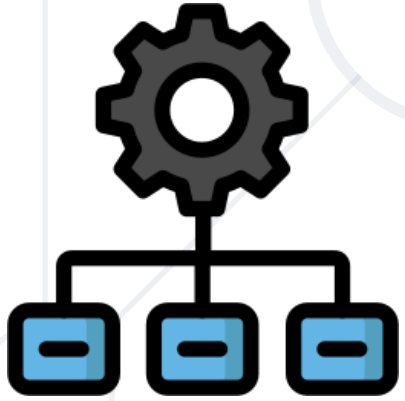
```
import { createApp } from 'vue';
import MyComponent from './ MyComponent.vue';
import ComponentA from './ ComponentA.vue';
import 'ComponentB from './ ComponentB.vue';
import 'ComponentC from './ ComponentC.vue';

const app = createApp({})

app.component('MyComponent', MyComponent)

// Or chain multiple components app
  .component('ComponentA', ComponentA)
  .component('ComponentB', ComponentB)
  .component('ComponentC', ComponentC)
```

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

Reusing Components

- Components can be reused as many times as you want
- Each component maintains its own state

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



Component's Data

- The data option must be a function
- Each instance maintains an independent copy of the data

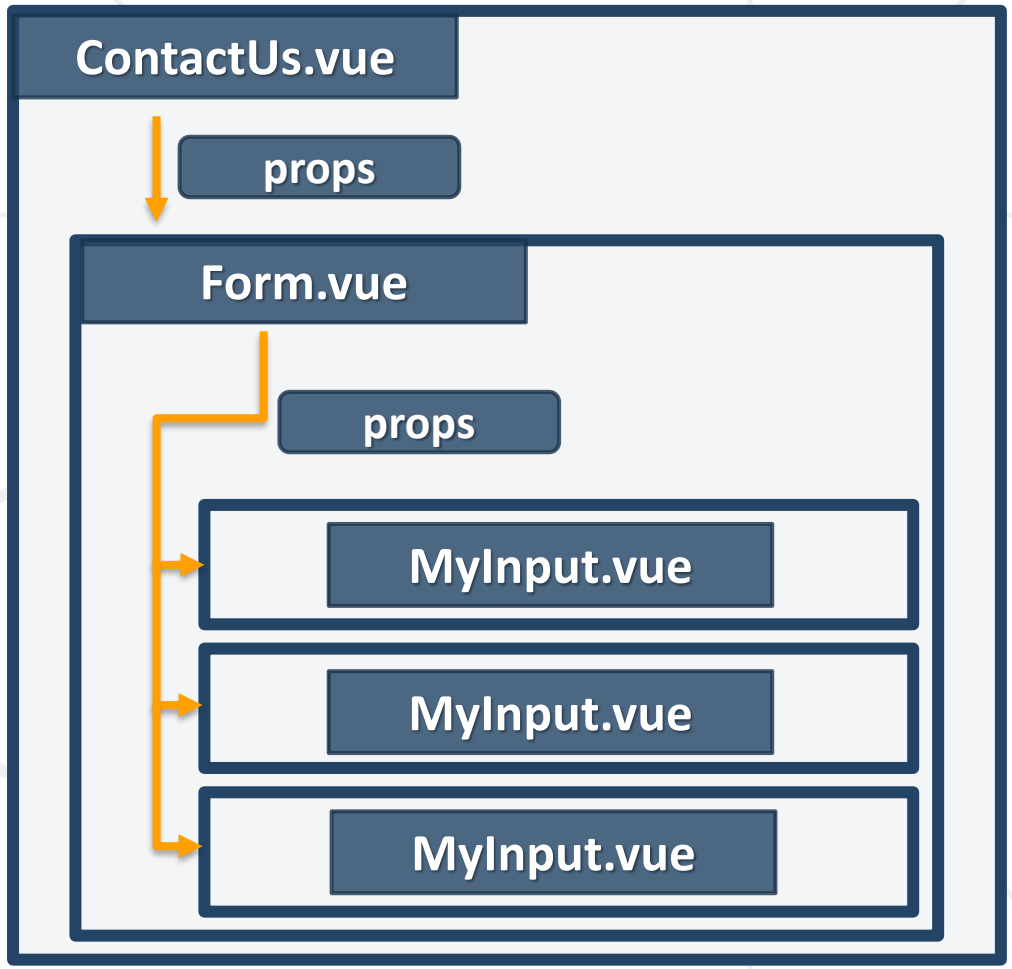
```
data() {
  return {
    count: 0,
    items: [ 'First', 'Second', 'Third' ]
  }
}
```

Props

Passing data down - Parent to Child

What is a prop?

- Components communicate with each other all the time (pass data between themselves)
- To pass data from a parent to a child component we use **props**
- Props can be thought of as custom attributes
- The value of a prop is received from the parent component
- **Props should NOT be mutated; they are meant to be read-only**
- With props **data flows one-way** - from parent to child



Declaring a prop

- Each component instance has a **props:{}** option
- After it has been passed it becomes a property of that instance
- Accessible through **this** same as **data(){}** properties

```
export default {
  props: ['foo'],
  created() {
    // props are exposed on `this`
    console.log(this.foo)
  }
}
```

```
export default {
  props: {
    title: String,
    likes: Number
  }
}
```

Using the prop

- declared using camelCase in the **props:{}** option
- used with kebab-case on the component

```
<MyComponent
  greeting-message="hello"
  :team-count="9“
/>
```

```
<template>
   <span>{{ greetingMessage }}
from {{ teamCount }}
        team members
   </span>
</template>
<script>
  export default {
    props: {
      greetingMessage: String,
      teamCount: Number
    }
  }
</script>
```

One-Way Data Flow

```
<template>
  <span>{{ greetingMessage }} from {{ teamCount }} team
members</span>
</template>

<script>
  export default {
    props: {
      greetingMessage: String,
      teamCount: Number
    },
  methods: {
    changeTitle() {
       // ❌ warning, props are readonly!
       this.greetingMessage = 'Changed greeting message!’;
    }
  }
}
</script>
```

Prop types

- The type of a prop can be one of the following native constructors
  - **Object**
  - **Date**
  - **Function**
  - **Symbol**
  - **String**
  - **Number**
  - **Boolean**
  - **Array**

Prop Validation

- All props are optional by default, unless **required: true** is specified
- An absent optional prop other than **Boolean** will have **undefined** value
- The **Boolean** absent props will be cast to **false**
- If a **default** value is specified, it will be used if the resolved prop value is **undefined** - this includes both when the prop is absent, or an explicit **undefined** value is passed
- When prop validation fails, Vue will produce a console warning (if using the development build)

```
props: {
    // Basic type check
    propA: Number,

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

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

Prop Validation – validator function

Software University

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

    // Custom validator function
    propE: {
      validator(value) {
        // The value must match one of these strings
        // Should always return true or false
        return ['success', 'warning',
'danger'].includes(value)
      }
    },
  }
```

Boolean Casting

```
export default {
  props: {
    disabled: Boolean
  }
}
```

```
<!-- equivalent of passing
:disabled="false" -->
<MyComponent />
```

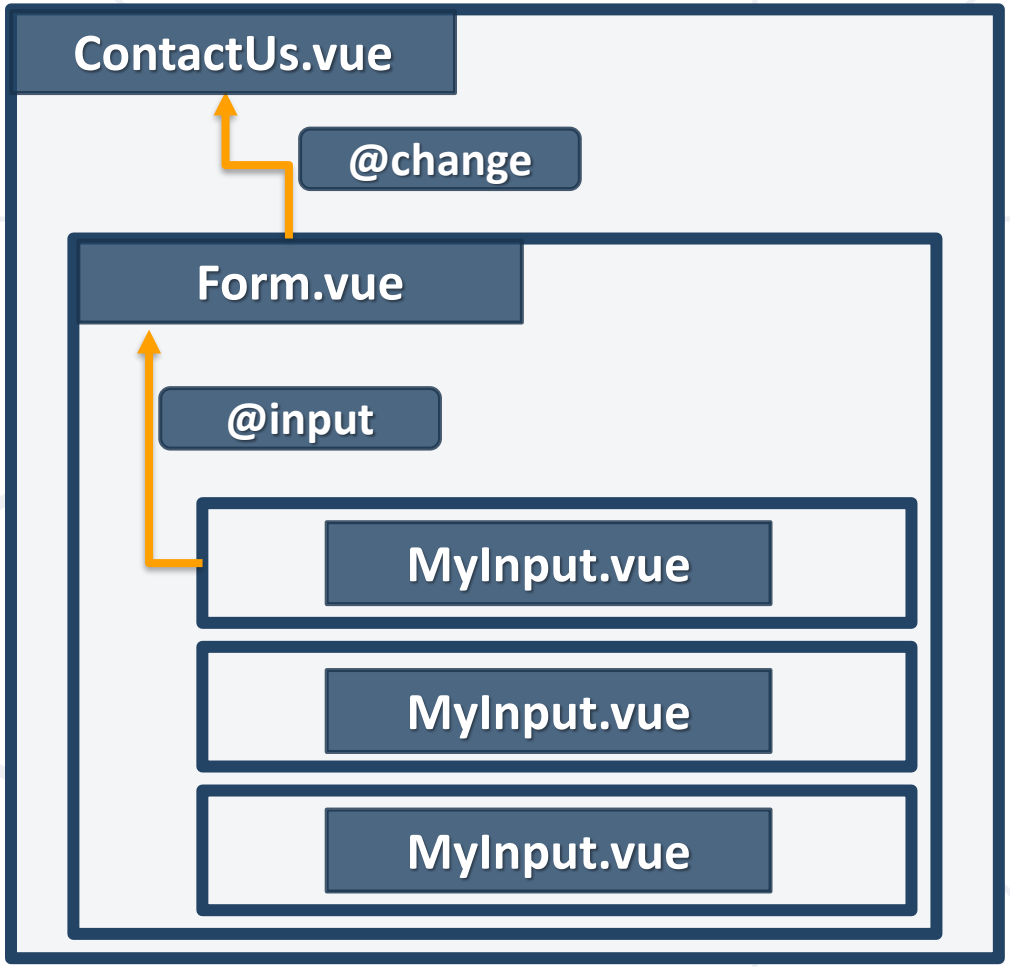
```
<!-- equivalent of passing
:disabled="true" -->
<MyComponent disabled />
```

Emitting Events

Passing data up – Child to Parent

What is a custom event?

- Sometimes, communicating back up to the parent is required
- Vue provides a custom event system to achieve this
- You can call the **$emit** function inside the child component to create a custom event
- The parent component can handle the custom event using **v-on** / **@** like a normal event



Emitting Events

- Component can explicitly declare the events it will emit using the **emits:{}** option

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

<script>
  export default {
    emits: ['someEvent'],
    methods: {
      submit() {
        this.$emit('someEvent')
      }
    }
  }
</script>
```

```
<MyComponent @some-
event="handleIt()" />
```

Emitting data

- **$emit** accepts also arguments
  - First argument is always a string and the name of the event we send
  - Any additional arguments will be passed into the listener's callback function

```
<button @click="$emit('increaseBy', 1)">
  Increase by 1
</button>
```

```
// Everything after the event name will be passed
to the handler
<MyButton @increase-by=“num => { console.log(num)
}" />
```

Dynamic Components

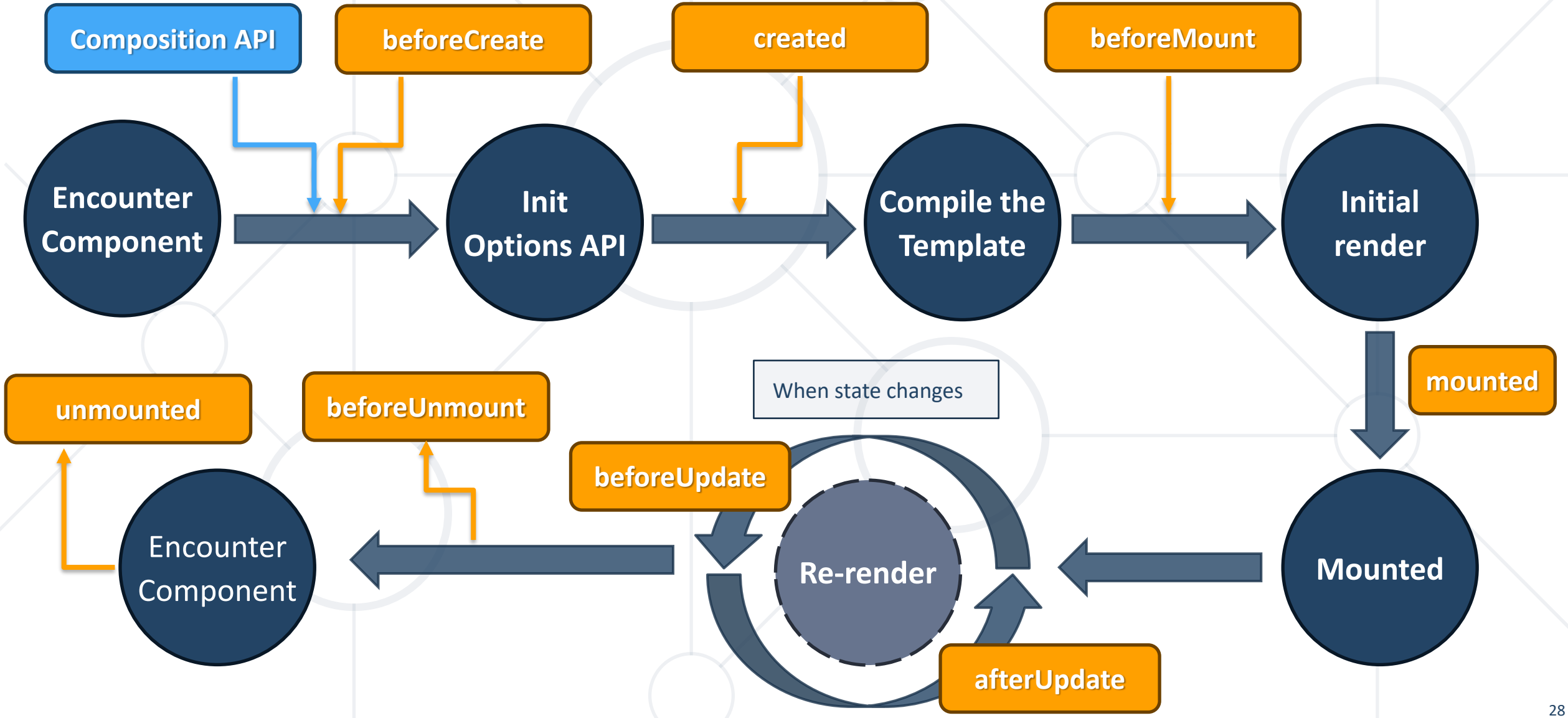
Dynamic Components

- Special **<component>** tag with **is** attribute, which accepts
- The name of a registered component as a string
- Or the actual imported component object

```
<template>
  <div>
    <button @click="toggleComponent">Toggle
Component</button>

    <!-- Dynamic component -->
    <component
:is="currentComponent"></component>
  </div>
</template>
```

```
data() {
    return {
      currentComponent:
'FirstComponent'
    };
  },
  components: {
    FirstComponent,
    SecondComponent
  },
  methods: {
    toggleComponent() {
      this.currentComponent =
        this.currentComponent ===
'FirstComponent'
          ? 'SecondComponent'
          : 'FirstComponent';
    }
  }
```

Lifecycles


Composition API
beforeCreate
created
beforeMount
Encounter Component
Init Options API
Compile the Template
Initial render
mounted
unmounted
beforeUnmount
When state changes
beforeUpdate
Encounter Component
Re-render
Mounted
afterUpdate

Slots

Passing content and components

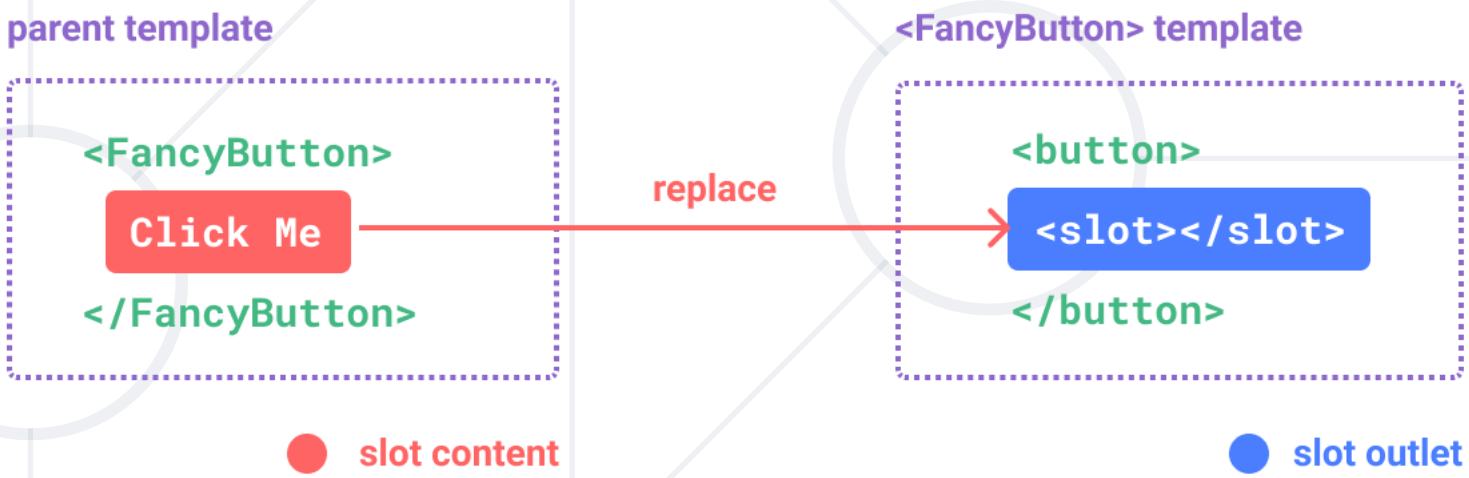
Why we need slots?

- Passing data with props can be harsh
  - The properties are a lot
  - You need to pass HTML between components
  - You want to be able to dynamically change internal components
- Slots serve as distribution outlets for content
  - Inspired by Web Components
  - Similar to React's children prop and Angular's <ng-content>

Slot example

```
<FancyButton>
  Click me! <!-- slot content -->
</FancyButton>
```

```
<button class="fancy-btn">
  <slot></slot> <!-- slot outlet -->
</button>
```



Render Scope

- Slot content has access to the data scope of the parent component, because it is defined in the parent
- Slot content does **not** have access to the child component's data

```
<template>
  <div>
    <!-- Parent component -->
    <span>{{capitalizeMessage() }}</span>
    <FancyButton>{{ capitalizeMessage()
}}</FancyButton>
  </div>
</template>
```

```
<script>
export default {
  data() {
    return {
      message: 'hello from parent
component!'
    };
  },
  methods: {
    capitalizeMessage() {
      return
this.message.charAt(0).toUpperCase() +
        this.message.slice(1);
    }
  }
};
</script>
```

Fallback Content

```
<button class="fancy-btn">
  <slot>
    Fancy! <!-- fallback content -->
  </slot>
</button>
```

```
<FancyButton></FancyButton>
// Will render → <button class="fancy-btn">Fancy!</button>


<FancyButton> Is this magic? </FancyButton>
// Will render → <button class="fancy-btn"> Is this magic?
</button>
```

Named Slots

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

  <main>
    <slot></slot>
  </main>

  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

```
<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>
```

CSS Deep Selector

Accessing children in a scoped CSS

CSS Deep selector

- When using **scoped**, styles in a parent component won't spill over into child components
- For layout purposes the root element of a child component will be influenced by both the parent's scoped CSS and the child's scoped CSS
- If you want a selector in scoped styles to be "deep", i.e. affecting child components, you can use the **:deep()** pseudo-class

```
<style scoped>
.a :deep(.b) {
  /* ... */
}
</style>
```

```
.a[data-v-f3f3eg9] .b {
  /* ... */
}
```

Exercise – Flip game

- Create a flip card memory game using components
  - Cards stay flipped only if they're the same pair
  - Each card should be a reused component
  - The controls of the game should be also in their own encapsulated component
  - The game should track time – after 1min you need to stop the game. The UI and notification should be updated.
  - Handle the win state when all cards are guessed correctly

Practice

Live Exercise in Class (Lab)

Summary

- Components help us create reusable or encapsulated logic
- Props and Events give us opportunity to communicate between the different component
- Lifecycle functions give us even more control and help us tap into the workflow of Vue
- Slots are great feature to create customizable and flexible components

Questions?

SoftUni

Software University | SoftUni Creative | SoftUni Digital | SoftUni Foundation | SoftUni Kids | Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

Flutter™ International

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

BOSCH

Postbank
Решения за твоето утре

PHAR VISION

SmartIT

DXC TECHNOLOGY

createX

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg