Vue Templates

Overview

- This section describes Vue template syntax
- Templates can appear in two places
  - **`<template>`** tag of single-file components (SFCs)
  - **`template`** property of instance definition objects

Interpolation

- Inserts result of a JavaScript expression into HTML
- Uses Mustache syntax
  `{{ expression }}`
- Refer to **`prop`**, **`computed`**, **`data`**, and **`method`** names here without **`this.`**

Directives

- Appear as HTML attributes
- Many are provided by Vue
- Custom directives can be defined
  - may want to avoid because other developers won't be familiar with them
- Most commonly used
  - **`v-bind`**, **`v-if`**, **`v-else-if`**, **`v-else`**, **`v-for`**, **`v-model`**, **`v-on`**, **`v-show`**
- Less commonly used
  - **`v-cloak`**, **`v-html`**, **`v-once`**, **`v-pre`**, **`v-text`**

v-bind or : ...

- Binds an expression evaluated in current component instance to a prop of a child component or HTML element
  - typically references props or data in current component instance
- Example

```
<input type="checkbox" v-bind:checked="isChecked">
or
<input type="checkbox" :checked="isChecked">
```

- Also used to pass non-string literal values as props
  - includes boolean, number, object, and array literal values
  - example: `size="true"` passes a string, but `:size="true"` passes a boolean
  - similarly, `size="12"` passes a string, but `:size="12"` passes a number

... v-bind or :

- Several attributes commonly bind a value
- To compute CSS class names
  - use **`:class`** which takes a string value or an object
    where keys are class names and values are boolean expressions
    that determine whether each class name should be used
  - can be combined with use of a **`class`** attribute
    not preceded by a colon for class names that are always present

```
<div
  class="foo"
  :class="{bar: isBar, baz: isBaz}"
>
```

- To conditionally disable a form element
  - use **`:disabled`** which takes a boolean value

```
<input
  type="text"
  :disabled="age < 21"
  v-model="drink"
>
```

- To dynamically generate CSS styles
  - use **`:style`** which takes an object where the keys are camel-case CSS property names

```
<div :style="{backgroundColor: bgColor, color: fontColor}">
```

v-if

- Provides conditional rendering
- Can use any JavaScript expression

```
<div v-if="color.length">{{ color }}</div>

<div v-else-if="3 < size && size <= 7">medium</div>

<div v-else>unknown color and not medium</div>
```

- Elements that uses **v-else-if** must have a preceding sibling that uses **v-if**
- Element that uses **v-else** must have a preceding sibling that uses **v-if** or **v-else-if**

v-for

- Provides iteration over array elements and object properties

```
<div v-for="element in array" :key="element">
  {{ element }}
</div>

<div v-for="(element, index) in array" :key="index">
  {{ index + 1 }}) {{ element }}
</div>

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

keys can be any unique value

- Binding a value to **`key`** prop is required to allow Vue to minimize number of DOM updates performed when data changes

v-model

- Creates two-way binding between
  a form element value and a component **`data`** value
  - current value comes from **`data`** value
  - **`data`** value is updated when user changes form element value
- Can be used in **`<input>`**, **`<textarea>`**, and **`<select>`** elements
- Recall that **`<input>`** elements can be used for
  text, checkboxes, and radio buttons
- Example

```
<input type="text" v-model="firstName" />
```

`v-model` Modifiers

- By default, **`data`** values associated with a **`v-model`** are updated on each keystroke
- To wait until focus leaves input, use **`.lazy`** modifier

```
<input v-model.lazy="firstName" />
```

- To automatically trim string values, use **`.trim`** modifier.

```
<input v-model.trim="firstName" />
```

- To automatically convert value to a number, use **`.number`** modifier

```
<input v-model.number="age" />
```

  - works even without **`type="number"`**
  - needed even when **`type="number"`**

- Can use more than one modifier

```
<input v-model.lazy.trim="firstName" />
```

`v-model` for Checkboxes

- When only one checkbox uses a given **`v-model`** the value is a boolean indicating if it checked
- When multiple use the same **`v-model`** whose value is an array, it is populated with the values of the selected checkboxes
  - for example, when red and blue are checked, the array will be **`['red', 'blue']`**
  - values are in the order in which they are selected

```
<template>
  ...
  <div v-for="color in colors" :key="color">
    <label>
      <input
        type="checkbox"
        v-model="selectedColors"
        :value="color"
      /> {{ color }}
    </label>
  </div>
  ...
</template>
<script>
  ...
  data: () => {
    return {
      colors:['red', 'green', 'blue'],
      selectedColors: []
    };
  },
  ...
</script>
```

Form Validation ...

- HTML5 added many features to support form validation
- Form elements that include **`required`** attribute
  and have no value or an invalid value entered
  will remind user to enter a valid value
- Requires use of a **`<form>`** element
  that listens for **`submit`** event
- Typically want to prevent default behavior of form submission
  by including **`.prevent`** modifier

```
<form @submit.prevent="handleSubmit">
  <label>Name</label>
  <input type="text" v-model="name" required />
  <button type="submit">Submit</button>
</form>
```

… Form Validation

- Can use **`pattern`** attribute to specify regular expression that must be matched
  - can be a list of valid values - ex. **`pattern="red|green|blue"`**
  - can be a string pattern - ex. **`pattern="\d{3}-\d{3}=\d{4}"`** for U.S. phone numbers
- Can specify **`minlength`** and/or **`maxlength`** of string values
- Can specify **`min`**, **`max`**, and/or **`step`** for number values
- Can use **`list`** attribute referring to a **`<datalist>`** **`id`**
  to turn an input into a dropdown
- Valid form elements match CSS pseudo-class **`:valid`**
  and invalid ones match **`:invalid`**
  - use to style form elements based on their validity
- Can use JavaScript to customize content and style of error messages
- For more form validation options, see
  https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation

v-on or @

- Registers event handling for a specific event
- Value can be a
  - method name
  - JavaScript statement such as a method call with arguments or assignment to a **`data`** property
- Example
  - assuming **`delete`**, **`add`**, and **`processKey`** are component methods and **`havePower`** is a data property

```
<!-- calls with no arguments -->
<button @click="delete">Delete</button>

<!-- calls with an argument -->
<button @click="add(5)">Add 5</button>

<!-- passes event object -->
<input @keypress="processKey($event)" />

<!-- assigns to data -->
<button @click="havePower = !havePower">
  Toggle Power
</button>
```

Event Modifiers

- These follow an event name
- They include
  - `.prevent` to trigger `event.preventDefault()`
  - `.stop` to trigger `event.stopPropagation()` related to event capture and bubbling phases
  - `.once` to only process first event
  - `.self` to only process event if it occurred on this element (at target), as opposed to on an ancestor element — rarely used
  - `.capture` to only process event if it did not occur at target and occurred during capture phase — rarely used
  - `.passive` can improve scrolling performance, especially on mobile devices — rarely used
- Example

```
<button @click.prevent.stop="handleClick">
  Press Me
</button>
```

Key Modifiers

- These check for common key codes
- They include
  `.enter .tab .delete .esc .space .up .down .left .right`
- When these are used, event handling code is only invoked when the specified key is pressed

```
<input @keypress.enter="handleEnterKey">
```

System Modifiers

- These check for keys that can be pressed in conjunction with another key to change the meaning
- They include
  `.ctrl .alt .shift .meta`
  - in macOS, **`.meta`** detects the command key
- For example, to call method **`interrupt`** if ctrl-c is pressed while focus is an **`<input>`**

```
<input
  type="password"
  @keyup.ctrl.c="interrupt"
  v-model="password"
/>
```

Mouse Modifiers

- These check for presses of specific mouse buttons
- They include
  `.left .right .middle`
- Example

```
<div @click.right="handleClick">
  Click Me
</div>
```

v-show

- Toggles value of CSS **`display`** property for an element between its normal value (such as **`block`**, **`inline`**, or **`inline-block`**) and **`none`**
- Example
  - assuming component has a **`data`** property named **`showGreeting`**

```
<div v-show="showGreeting">Hello!</div>
<button @click="showGreeting = !showGreeting">
  Toggle Greeting
</button>
```

- Differs from conditional rendering using **`v-if`**
  - **`v-if`** will remove elements from DOM when condition is false
  - **`v-show`** leaves element in DOM and just changes its CSS **`display`** property

Other Directives ...

- **`v-cloak`**
  - hides an element until all the data it uses has been loaded
  - avoids displaying unevaluated interpolations (double curly braces)
  - this rarely occurs, so **`v-cloak`** is **rarely used**
  - in order for this directive to work, a CSS rule must be defined

```
[v-cloak] {
  display: none;
}
```

- **`v-html`**
  - renders a string of HTML as HTML instead of plain text
  - example: **`<span v-html="myHtml" />`**
    where **`myHtml`** is a **`prop`** or **`data`** value that is a string of HTML
  - the HTML should not contain Vue directives
    because those will not be processed

… Other Directives

- **`v-once`**
  - causes content of an element to be evaluated only once
  - if data used in content changes after initial render,
    the element will continue to render same output as initial render
  - avoids taking time to determine if re-rendering is needed
- **`v-pre`**
  - skips all evaluation of element content which allows rendering
    double curly braces and component references
  - one use is for outputting example Vue code
- **`v-text`**
  - alternative to using double curly braces to output a data value
  - for example, these are equivalent

```
<span v-text="firstName"></span>
<span>{{ firstName }}</span>
```

Refs

- Elements in a template can have a **`ref`** attribute
- Allows component methods to get a reference to them
- Has many uses
- One is to get a reference to an **`input`** element so focus can be moved to it

```
<template>
  <div>
    <input type="text" v-model="name" ref="name" />
    <button @click="clearName">Clear</button>
  </div>
</template>
<script>
export default {
  name: 'SomeComponent',
  data() {
    return {name: ''};
  },
  mounted() {
    this.$refs.name.focus();
  },
  methods: {
    clearName() {
      this.name = '';
      this.$refs.name.focus();
    }
  }
}
</script>
```

a lifecycle method covered later

Exercise ...

- Create beginning of an app that manages dogs
  - **`vue create dogs`** - select default preset
  - **`cd dogs`**
  - **`npm run serve`**
  - configure ESLint and Prettier by following steps on last slide in "Tools" section
  - modify **`src/App.vue`** to render the **`Dog`** component instead **`HelloWorld`**
    - in **`template`**, remove the **`<img>`** and **`<HelloWorld />`** elements and add **`<Dogs />`**
    - in list of components used, replace **`HelloWorld`** with **`Dogs`**
  - create a **`Dog`** component in **`src/components/Dogs.vue`** that
    - allows entry of dog names
    - displays them in a table
    - see tips on next few slides
  - run the app and verify that it works by adding a few dogs
  - we'll see how to persist data later

... Exercise ...

- **`Dog.vue`** template

```
<template>
  /* All of these are TODOs.
  Wrap everything in a div with a class of dogs.
  Render a table only if there is at least one dog (use v-if).
  Render a tr with a th that says "Name".
  For each dog in the dogs array (use v-for and :key="dog.name"),
  render a tr with a td that contains
  the value of dog.name (use interpolation).
  Render a div that contains a label, input, and button.
  The label should contain "Name".
  The input should have a v-model on the "name" data prop.
  The input should call the addDog method
  if the enter key is pressed (use @keypress.enter).
  The button should contain "Add".
  The button should call the addDog method when clicked.
  */
</template>
```

… Exercise …

- **Dog.vue** script

```
<script>
function sortDogs(dogs) {
  dogs.sort((dogA, dogB) => dogA.name.localeCompare(dogB.name))
  return dogs;
}

export default {
  //TODO: Define the data properties dogs and name.
  //TODO: Initialize dogs to an empty array.
  //TODO: Initialize name to an empty string.
  methods: {
    addDog() {
      // If a dog with that name is already present, do nothing.
      const exists = this.dogs.some(dog => dog.name === this.name);
      if (!exists) this.dogs = sortDogs(this.dogs.concat({name: this.name}));
      //TODO: Set the name data prop to an empty string to clear the input.
    },
  }
};
</script>
```

… Exercise

- **`Dog.vue`** styles

```
<style scoped>
button, input {
  border: solid gray 1px;
  border-radius: 4px;
  padding: 4px;
}

input {
  margin: 0 10px;
}

label {
  font-weight: bold;
}

table {
  border-collapse: collapse;
  margin-bottom: 10px;
}

td, th {
  border: solid gray 1px;
  padding: 5px;
}
</style>
```