Vue Components

Component Overview

- Components break up UI into reusable pieces

- At high level, components

  - accept input using props

  - generate DOM

  - respond to user interactions using event handling

  - produce data using events, callback functions, and Vuex state

- Rendered using a virtual DOM, just like in React

  - minimizes number of actual DOM updates performed

- Automatically updated when their data changes

  - includes changes to props, data, and
    state in a Vuex store (if used by component)

Ways to Define Components

- **Single File Component (SFC)**

  - most common

  - defined in a `.vue` file that holds valid HTML

  - exports an instance definition object described soon

  - supports scoped styles described in "Styles" section

  - using components can assign alternate name to avoid conflicts

  - Webpack `vue-loader` processes these files

    - Vue CLI configures this by default

- **Vue.component(*name, instanceDefinitionObject*)**

  - much less common

  - no support for scoped styles

  - names must be unique throughout app

  - defined in a `.js` or `.vue` file

  - "can be used in the template of any <u>root Vue instance</u> (new Vue) created after registration" very restrictive!

3

Vue.component

- Components defined this way can only be used in the template of a <u>root Vue instance</u>

```
Vue.component('Greet', {
  name: 'Greet',
  props: {
    name: {
      type: String,
      required: true
    }
  },
  //template: '<div>Hello, {{ name }}!</div>'
  render(createElement) {
    const msg = `Hello, ${this.name}!`;
    const children = [msg];
    return createElement('div', children);
  }
});
```

can use template property or **render** method

4

`createElement` Arguments

- Tag or component name - ex. `'div'` or `Greeting`

- Object describing attributes - optional

  - ex. `{attrs: {id: 'message'}, class: 'danger'}`

  - some attributes are treated specially such as `class`, `style`, and those for event handling

- String or array describing children - optional

  - ex. `['warning: the', otherElement, 'is too hot']`

```
Vue.component('Danger', {
  props: {
    message: {type: String, required: true}
  },
  render(createElement) {
    return createElement('div', {class: 'danger'}, this.message);
  }
});
```

Example usage:
`<Danger message="out of memory" />`

```
Vue.component('Danger', {
  props: {
    message: {type: String, required: true}
  },
  render() {
    return <div class="danger">{this.message}</div>;
  }
});
```

version using JSX instead of `createElement`

more on JSX at the end of this section

5

SFC Layout

- **`<template>`**

  - holds HTML that is not immediately rendered

  - later it can be cloned and added to DOM, zero or more times

  - Vue components do this for each instance

  - can use other components, interpolation, and directives

  - can include `<!-- comments -->`

- **`<script>`**

  - holds JavaScript that defines and exports an "instance definition"

  - can import things from other files

- **`<style>`**

  - holds CSS or another syntax such as Sass

  - can be scoped to the component so it doesn't affect HTML outside it

```
<template>
  ...
</template>


<script>
  ...
</script>


<style lang="scss" scoped>
  ...
</style>
```

`.vue` file containing valid HTML

Instance Definition Objects

- **Vue**.**component** takes one as its second argument
  - rarely used
- SFC **script** tags export one
- Contains same properties in either case
- Most are optional
- The following slides describe them

```
{
  el: 'some-selector',
  name: 'SomeName',
  components: { ... },
  props: { ... },
  computed: { ... },
  data() {
    return { ... };
  },
  watch: { ... },
  methods: { ... },
  life-cycle-methods,
  template: 'some-template'
}
```

`el` Property

- Typically only specified in top-level components

- Value is CSS selector string that specifies
  where this component should be rendered

- Example `el: '#app',`

- Alternate way to specify

  - create a new **Vue** object and call its **$mount** method

  - **main.js** in apps generated by Vue CLI does this

```
import Vue from 'vue';
import App from './App.vue';

new Vue({
  render: h => h(App)
}).$mount('#app');
```

h stands for "hyperscript" which is a name given to scripts that generate HTML

8

`name` Property

- Component name

- Only used in SFCs

  - with `Vue.component` the name is specified in first argument

- Typically matches source file name

- If kebab-case

  - same name must be used in elements

  - ex. `'foo-bar'` ➡ `<foo-bar>`

- If camel-case

  - element name can be kebab-case or camel-case

  - ex. `'FooBar'` ➡ `<foo-bar>` or `<FooBar>`

- Example `name: 'FooBar',`

> I prefer to make components
> names camel-case and
> refer to them with camel-case
> which matches React convention.

9

`components` Property

- List of other components used by this one

- Omit if no other components are used

- Value is object where
  keys are component names and
  values are components

- Example

  - suppose components `Foo` and `Bar` are used

  ```
  components: {Foo: Foo, Bar: Bar},
  ```

  - or using ES6 object shorthand

  ```
  components: {Foo, Bar},
  ```

- Ability to choose names by which components will be referenced

  - important when there are name conflicts because it allows
    use of multiple components that happen to have the same name

props Property ...

- Object or array describing props this component accepts

- Allows parent components to pass data to child components

- When value is an array

  - just a list of prop names; bypasses type checking

```
props: ['age', 'name'],
```

- When value is an object

  - keys are prop names

  - values are either a type
    or another object

```
props: {age: Number, name: String},
```

  - object properties are

    - `type` - JavaScript type name (ex. `String`)
      or custom class name (ex. `Person`)

    - `default` - optional default value that matches type

    - `required` - optional boolean

```
props: {
  name: {
    type: String,
    required: true
  },
  age: {
    type: Number,
    default: 0
  }
},
```

supported JS classes include
`Boolean`, `Number`,
`String`, `Symbol`,
`Date`, `Function`,
`Object`, and `Array`

... `props` Property ...

- For more fine-grained validation of a prop value
  add **validator** method

- Example

```
const isNumber = value => typeof value === 'number';
const isString = value => typeof value === 'string';
...

  props: {
    person: {
      type: Object,
      required: true,
      validator(person) {
        const {name, age} = person;
        return isString(name) && isNumber(age);
      }
    }
  }
```

`... props` Property

- Prop values are passed in from parent components using attributes

- Can be any kind of value

  - including functions defined in parent component that child component can call

- When prop values change

  - component is updated rather than creating a new instance

  - `beforeUpdate` and `updated` lifecycle methods are invoked

- Camel-cased prop names must be written in kebab-case in HTML

  - example: `fooBar` prop in HTML would be
    `<SomeComponent foo-bar="some-value" />`

  - of course using single-word prop names avoids this issue

13

computed Property

- Object describing props that are
  computed based on other props and data

- Defines methods whose names are prop names and return prop value

- Results are cached and only recomputed when data they depend on changes

- Makes them more efficient that implementing
  a method that returns the same computed value

  - defining instance methods is described later in this section

- Example

```
computed: {
  fullName() {
    return this.firstName + ' ' + this.lastName;
  }

  // Works, but verbose.
  //fullName: function () {
  //  return this.firstName + ' ' + this.lastName;
  //}

  // Does not work because it uses wrong "this" value.
  //fullName: () => this.firstName + ' ' + this.lastName
}
```

14

`data` Property ...

- Value is a function that returns an object
  containing data specific to each component instance
  - allows each instance to maintain its own data
  - similar to "state" in React
- In returned object
  - keys are data names
  - values are initial values which can be changed later
- **`v-model`** directives refer to data property names
- Example

```
data() {
  return {
    email: '@gmail.com',
    rating: 10,
    car: {
      make: '',
      model: '',
      year: new Date().getFullYear()
    }
  };
},
```

> **`v-model`** directive, described in "Templates" section, provides two-way data binding between a form input and a **`data`** property

15

... `data` Property ...

- Outputs error if set to an object instead of a function
  - "error: data property in component must be a function"
- Changes to data are watched by Vue
- Some JavaScript approaches to modifying data aren't seen by Vue, so other techniques must be used
  - details on next slide

... `data` Property ...

- Primitive values
  - set with **`this.`*`property`*` = `*`value`*`;`**
  - delete with **`this.`*`property`*` = `**`null;`**
- Object values
  - can assign a new object
  - can set a property
    - if initially present use **`this.`*`someObj.property`*` = `*`value`*`;`**
    - if not use **`this.``$set``(this.`*`someObj`*`, `*`property`*`, `*`value`*`);`**
  - delete a property with **`this.``$delete``(this.`*`someObj`*`, `*`property`*`);`**
- Array values
  - can assign a new array
  - set an element with **`this.``$set``(this.`*`someArray`*`, `*`index`*`, `*`value`*`);`**
  - delete an element in two ways
    - 1) assign result of **`Array`** **`slice`** or **`splice`** method to **`this.`*`someArray`*`**
    - 2) **`this.``$delete``(this.`*`someArray`*`, `*`index`*`);`**

17

- Sometimes desirable to set data based on a prop value and update it whenever a new prop value is passed in

- One way to achieve this

  - prop name is `foo` and data name is `fooData`

```
props: {
  foo: {
    type: String,
    required: true
  }
},
data() {
  return {
    fooData: this.foo // captures initial value
  };
},
watch: {
  foo(newValue) {
    this.fooData = newValue; // captures updates
  }
},
```

watch is described in two more slides

When to use `this.`

- Instance definition object **props**, **data**, and **computed** all define properties on the component instance

- Templates access them without **this.** prefix

- Methods must use **this.** prefix

```
<template>
  <div>
    Hello, {{ name }} ({{ initials }})!<br />
    Today is {{ date }}.
  </div>
</template>
<script>
export default {
  name: 'Greet2',
  props: {
    name: {
      type: String,
      required: true
    }
  },
  computed: {
    initials() {
      return this.name.split(' ')
        .map(part => part[0].toUpperCase())
        .join('');
    }
  },
  data() {
    return {
      date: new Date().toDateString()
    }
  }                     example: 'Sat Mar 09 2019'
};
</script>
```

19

`watch` Property

- Object where
keys are names of data to be watched and
values are functions to execute when value changes

- These functions are passed new and old values

- To watch for <u>deep changes</u> in an object or array,
use an object for the value with
deep and **handler** properties

  - **deep** is a boolean that must be set to **true**

  - **handler** is a function that is invoked with new and old values
when anything in watched object changes

- If goal is only to compute new property values
based on changes to watched props,
use **computed** properties instead of **watch**

```
props: {
  user: {
    type: Object,
    required: true
  }
},
watch: {
  //user(newUser) {
  //   console.log(newUser);
  //}
  user: {
    deep: true,
    handler(newUser) {
      console.log(newUser);
    }
  }
},
```

(A)

(B)

Approach A works if new object
is assigned to **user** prop,
but not if properties inside
existing object are changed.
Approach B works when
a new object is assigned AND
when properties inside
existing object are changed.

20

`methods` Property

- Value is an object that defines component methods
- Primarily used for event handling
  - example `<button @click="handleClick">Do It</button>`
- Inside these methods, `this` refers to a component instance
- Lifecycle methods are defined at top of instance definition object, not here
  - more on these later
- Example

```
methods: {
  handleClick() {
    // handle the click
  },
  handleSubmit() {
    // handle the submit
  }
},
```

Reactive Properties

- Vue component properties are reactive

  - includes `props`, `data`, and `computed`

  - DOM updates are triggered when methods change their values

Lifecycle Methods

- Described in "Lifecycle" section

`template` Property

- String of HTML to be rendered

- Alternative to `<template>` element

- Fine for small amounts of HTML, but
  `<template>` is preferred for large amounts

- Example `template: '<div>Email: {{ email }}</div>',`

- To spread across multiple lines,
  surround with backticks →

```
template: `
  <div>
    Email: {{ email }}
  </div>
`,
```

- Does not support JSX

  - use **render** method on next slide for that

- Requires `runtimeCompiler` option

vue.config.js
```
module.exports = {
  runtimeCompiler: true
};
```

24

`render` Method ...

- Alternative to **template** property and **<template>** element

- Supports using JavaScript to determine content
  instead of Vue template directives

  - like in React

- Passed **createElement** function

- Return result of **createElement** call or JSX | more details on JSX later |

  - many find JSX more readable than calls to **createElement**

- Examples

  - **ColorList** component on next slide renders a list of color names in three ways

  - in this case, using a **<template>** element or **template** property are the best options

25

... `render` Method ...

```
export default {
  name: 'ColorList',
  props: {
    colors: {
      type: Array,
      required: true
    }
  },
  template: `
    <div>
      <div v-for="color of colors">{{ color }}</div>
    </div>
  `,
  render(h) {
    const children = this.colors.map(
      color => h('div', color));
    return h('div', children);
  },
  render() {
    return (
      <div>
        {this.colors.map(color => <div>{color}</div>)}
      </div>
    );
  }
}
```

using template
remove **render** methods

using render
with createElement
remove **template** and
other **render** method

using render with JSX
remove **template** and
other **render** method

can only use one of
these **render** methods

h is common alias for
createElement function
stands for **hyperscript** which is a
"script that generates HTML structures"

26

... `render` Method

- SFCs that use a **`render`** method instead of an HTML template can still include a `<style>` element

- If no CSS is needed

  - file extension can be changed from `.vue` to `.js`

  - `<script>` start and end tags can be removed

  - works because in this case Vue build tooling is not needed

27

JSX ...

- Stands for "JavaScript XML"

- XML syntax for generating DOM

- Can be returned by a **render** method

  - or by methods the **render** method calls

- Alternative to using a template

- Requires a Babel plugin

  - projects created by Vue CLI have this configured by default

- Most Vue developers prefer to not use JSX

```
<script>
export default {
  name: 'Greeting',
  props: {
    name: {
      type: String,
      required: true
    }
  },
  render() {
    return (
JSX →  <div class="greeting">
        Hello, {this.name}!
      </div>
    );
  }
};
</script>
```

28

… JSX …

- Vue template vs. JSX syntax differences

  - interpolation `{{ }}` ➝ `{ }`

  - Vue directives ➝ interpolation containing JavaScript expressions

  - `v-if` ➝ ternary inside `{ }`

  - `v-for` ➝ `map` method inside `{ }`

  - `v-model` ➝ custom event handling that updates a `data` property

29

... JSX

- Example of **render** method that returns JSX and uses a method call to get more JSX

```
<script>
export default {
  name: 'ColorList',
  data() {
    return {
      colors: ['red', 'green', 'blue']
    };
  },
  methods: {
    getItems() {
      return this.colors.map(color => <li>{color}</li>);
    }
  },
  render() {
    return <ul>{this.getItems()}</ul>;
  },

};
</script>
```

Vue Components

Exercise ...

- Build a **`Temperature`** component in **`my-project`** created earlier

-10	temperature: 70F	110
	comfortable	

  - renders a slider using `<input type="range">`

  - accepts many props

    - `min` and `max` are the minimum and maximum allowed temperatures

    - `cold` is the temperature where all values at or below are considered cold

    - `hot` is the temperature where all values at or above are considered hot

    - `temperature` is the current temperature

  - emits a **`change`** event when slider is moved

  - displays current value

  - displays evaluation of cold, comfortable, or hot

- All the code is provided so the focus is on gaining experience working with Vue

  - also, some topics haven't been covered yet

- After writing the code, verify that it works

... Exercise ...

- Create **src/components/Temperature.vue**

- **template** section of **Temperature.vue**

```
<template>
  <div class="temperature">
    <input
      :min="min"
      :max="max"
      :style="{backgroundColor: color}"
      :value="value"
      @input="handleInput"
      type="range"
    />
    <div class="labels">
      <div>{{ min }}</div>
      <div>temperature: {{ value }}F</div>
      <div>{{ max }}</div>
    </div>
    <div class="evaluation">{{ evaluation }}</div>
  </div>
</template>
```

min and **max** are props that are passed in

color is a computed prop

value is a data value

handleInput is a method

evaluation is a computed prop

… Exercise …

- **script** section of **Temperature.vue**

```
<script>
export default {
  name: 'Temperature',
  props: {
    cold: {
      type: Number,
      required: true
    },
    hot: {
      type: Number,
      required: true
    },
    max: {
      type: Number,
      default: 100
    },
    min: {
      type: Number,
      default: 0
    },
    temperature: {
      type: Number,
      default: 0
    }
  },
```

```
  data() {
    return {
      value: this.temperature
    };
  },
  computed: {
    color() {
      return this.evaluation === 'cold'
        ? 'blue'
        : this.evaluation === 'hot'
        ? 'red'
        : 'green';
    },
    evaluation() {
      return this.value <= this.cold
        ? 'cold'
        : this.value >= this.hot
        ? 'hot'
        : 'comfortable';
    }
  },
  methods: {
    handleInput(event) {
      this.value = Number(event.target.value);
      this.$emit('change', this.value);
    }
  }
};
</script>
```

> initial value is the temperature that is passed in via a prop

> `event.target.value` is a string

> parent components can listen for the event emitted here

33

... Exercise ...

- **style** section of **Temperature.vue**

```
<style scoped>     styles are scoped to Temperature component
.evaluation {
  display: flex;   using flexbox for layout
  font-weight: bold;
  justify-content: center;
}

input {
  appearance: none;   necessary to change background color
  background-color: gray;
  width: 100%;
}

.labels {
  display: flex;    using flexbox for layout
  justify-content: space-between;
}

.temperature {
  width: 100%;
}
</style>
```

34

... Exercise

- Modify **App.vue** to use **Temperature** component

```
<script>
import Temperature from './components/Temperature.vue';

export default {
  name: 'App',
  components: {
    Temperature
  },
  data() {
    return {
      temperature: 70
    };
  },
  methods: {
    temperatureChange(t) {
      this.temperature = t;
      console.log(
        'App.vue temperatureChange: this.temperature =',
        this.temperature
      );
    }
  }
};
</script>
```

```
<template>                          at top of file
  <div id="app">
    <Temperature
        :min="-10"
        :max="110"
        :cold="30"
        :hot="90"
        :temperature="temperature"
        @change="temperatureChange"
    />
  </div>
</template>
```