Vue Fundamentals

Component Communication

Peter Kassenaar –
info@kassenaar.com

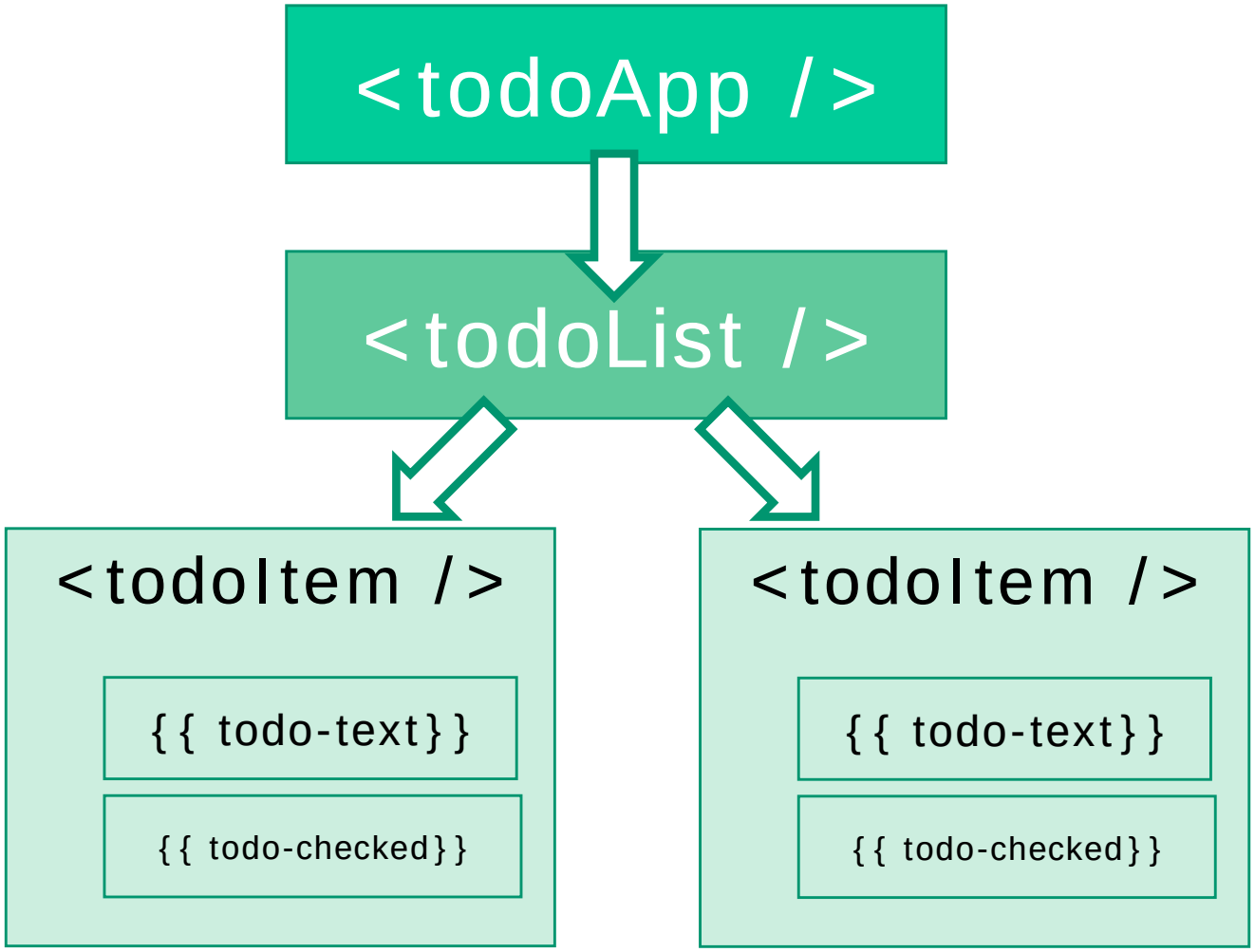

Peter Kassenaar
INCLUSIEF GRATIS WEBVERSIE VAN HET BOEK
Vue.js
Web Development Library
VANDUUREN MEDIA

P. 136 ev

Types of communication

- Parent–child communication:
  - Using `props` to share data with child components
  - Validating component properties
- Child–parent communication
  - Passing data back to parent components
- Injecting content into child components using `<slot>`'s

Vue app : Tree of components



Recap – Multiple components?

1. Create new `.vue` components
2. Import them in the parent component using `import …`
3. Reference them in the HTML, using `<ComponentName />`
4. Repeat for every component

Creating a CountryDetail Component

- We're creating a separate `CountryDetail.vue` Component and move the HTML from the parent Component
- We tell the component to receive a `country` property

```
<template>
  <h2>{{ country.name }}</h2>
  <ul class="list-group">
    <li class="list-group-item">{{ country.id }}</li>
    …
  </ul>
</template>


<script setup>
  const props = defineProps(['country’])
  const country = props.country
</script>
```

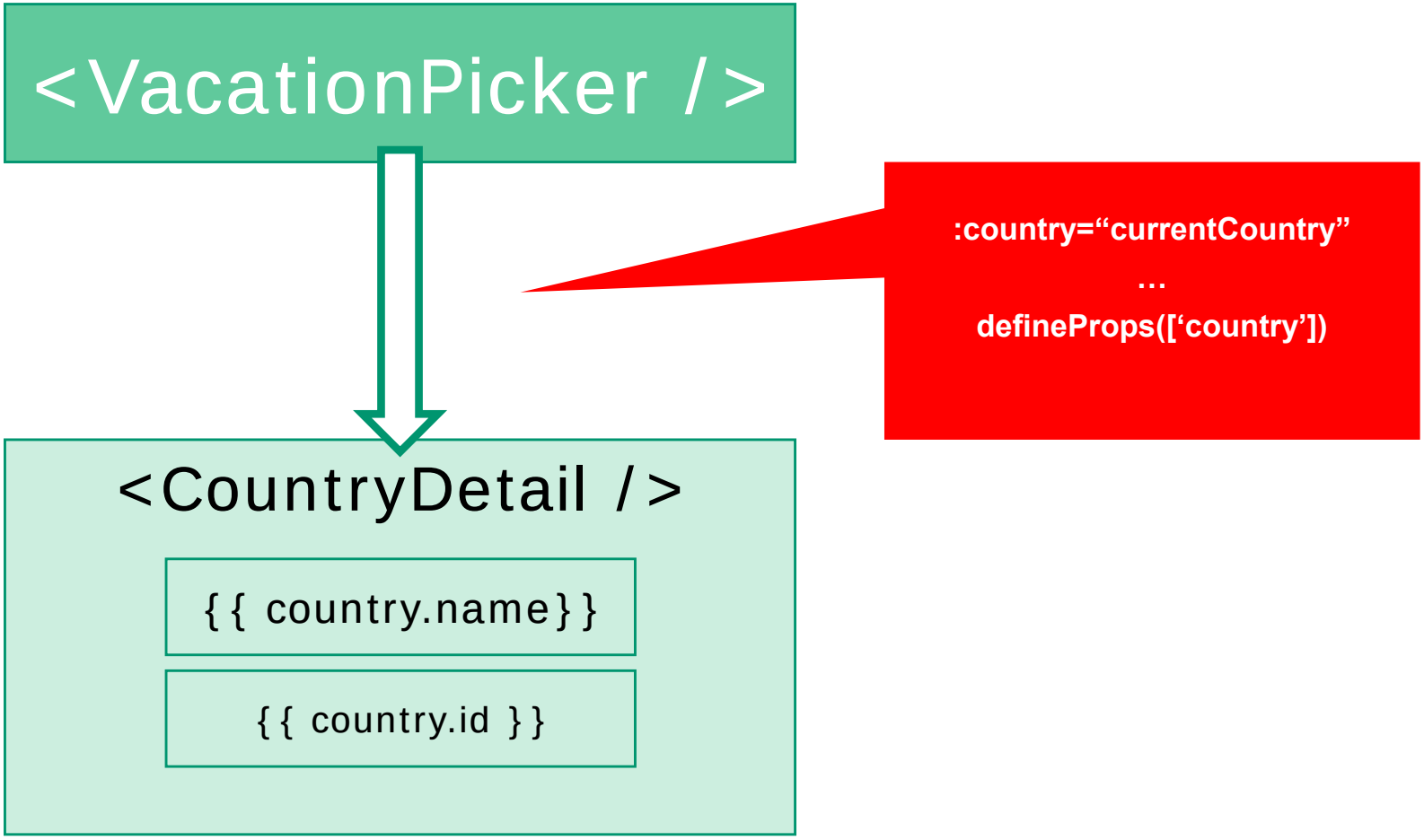
```
<script setup>
// OR, using destructuring:
const {country } = defineProps(['country'])
</script>
```

Data flow between components

"Data flows in to a component via `v-bind:` bindings"

Data flows out of a component via `v-on:` or `@event` events"

Parent-Child flow: v-bind: or :



1. Prepare Detail component to receive data

- The data you pass to a component are called *`props`*.
- Props can be strings, numbers, arrays, objects and so on.
- Props are defined using the built-in `defineProps()` macro

```
const {country} = defineProps(['country'])
```

We can then bind to the properties of

the passed in country with `country.id`, `country.name`, etc.

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

Or – using object notation in `defineProps()`

Or, using object notation and passing in the type of properties (you have more options if you use TypeScript for that)

```
const {country, name} = defineProps({
  country : Object,
  name: String
  // more props...

})
```

2. Update Parent component to send data down

```
<template>
   …
  <div class="col-6">
    <CountryDetail  v-if="showDetails" :country="currentCountry"/>
  </div>
</template>

<script setup>
// import the child component receiving the country details
import CountryDetail from "@/components/CountryDetail.vue";
…

</script>
```

Move methods and computed properties

- In this example: move or copy the necessary methods from the parent component to child component,
- In this example:
  - `imgUrl`
  - `isExpensive()`
  - `isOnsale()`

```
// Automatically calculate if a destination is expensive
const isExpensive = computed(() => {
  return country.cost > 4000;
});

// Automatically calculate if a destination is on Sale
const isOnSale = computed(() => {
  return country.cost < 1000;
});

// A computed property that returns the URL to the image for currentCountry.
const imgUrl = computed(() => {
  return new URL(`/src/assets/countries/${country.img}`, import.meta.url).href;
})
```

Casing of props

- HTML attributes are case-*insensitive*
- If you use camelCase on prop-names, use a hyphen in the html
  - i.e. `defineProps(['countryDetail', 'countryName'])` becomes `<DetailComponent country-detail="…" country-name="…" />`

```js
defineProps({
  greetingMessage: String
})
```

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

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

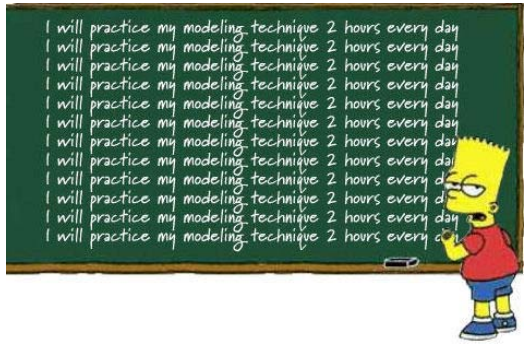
```template
<MyComponent greeting-message="hello" />
```

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

Workshop

1. Create a DetailComponent on your own application and pass data. OR:
   - Create an extra prop on the `CountryDetailComponent` and pass it.
2. Create a new component with a textbox and a button.
   - When the button is clicked, the text in the box is passed as a prop to a child component.
   - Tip: Use `v-model` on the textbox.
3. Optional: implement the lifecycle hook `onUnmounted` on the child component, showing a `console.log()` that the component was unmounted/hidden.

- Generic Example on props: `../200-props`



Validating props

Making sure only specific kinds of data get passed


Peter Kassenaar
INCLUSIEF GRATIS WEBVERSIE VAN HET BOEK
Vue.js
Web Development Library
VANDUUREN MEDIA

P. 144 ev

Validating props

- Prevent bad data being passed in.
- Use a keyed object instead of a simple array of props
- Optional: add extra attributes, like `required` or a `validator()` function.
- (With TypeScript the type checking of props is much easier)

Warning in advance:

'Validating' props in Vue doesn't actually do that much…

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

```js
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
  },
  // Required but nullable string
  propD: {
    type: [String, null],
    required: true
```

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

Simple validation of `CountryDetail` props

```
const {country, name } = defineProps({
  country: {
    type: Object,
    required: true
  },
  name: {
    type: String,
    required: true
  }
})
```

Use objects-inside-objects. The `key` is the name of the prop, the `value` is an object with the validation requirements.

Console errors if prop has wrong value

```
CountryDetail  v-if="showDetails"
                :country="'test'"
                :name="currentCountry.name"/>
```

```
Attempting                                content_script_bundle.js:1
initialization F  Dec 13 2024 14:11:52 GMT+0100
(Midden-Europese standaardtijd)

▶ [Vue warn]: Invalid prop:  VacationPicker.vue:11
type check failed for prop "country". Expected
Object, got String with value "test".
  at <CountryDetail key=0 country="test"
name="USA" >
  at <VacationPicker >
  at <App>

>
```

Errors if you do not pass a required prop

```
<CountryDetail v-if="showDetails"
               :country="country" />
```

```
initialization Fr Dec 13 2024 14:13:04 GMT+0100
(Midden-Europese standaardtijd)

▶ [Vue warn]: Missing                VacationPicker.vue:11
required prop: "name"
  at <CountryDetail key=0 country=
  {id: 1, name: 'USA', capital: 'Washington', cos
▶ t: 1250, details: 'United States are among the m
  ost visited country in the world.', …}
>
  at <VacationPicker >
  at <App>
```

The application *will* continue to run, but shows the error in te console to help you further.

This kind of ‘validation’ does *not* stop you from assigning bad values.

Validator functions for props

- You can pass in a `validator` function to validate the input. For example:
  - we want to pass in an `id`,
  - It has to fall inside a specific range
  - (in real life apps of course you wouldn't hardcode this).
  - Validator has to return `true` or `false`

```
id: {
  type: Number,
  required: true,
  validator(value) {
    return [1, 2, 3, 4, 5, 6, 7, 8, 9].includes(value);
  }
}
```

Errors in console on validation

```
countries: [
   {
      id: 100,
      name: 'USA',
      capital: 'Washington’,
      …
   },
```

```
initialization Fri Dec 13 2024 14:18:55 GMT+0100
(Midden-Europese standaardtijd)

▶ [Vue warn]: Invalid prop: VacationPicker.vue:11
custom validator check failed for prop "id".
  at <CountryDetail key="0" name="USA" id=100  ... >
  at <VacationPicker >
  at <App>

>
```

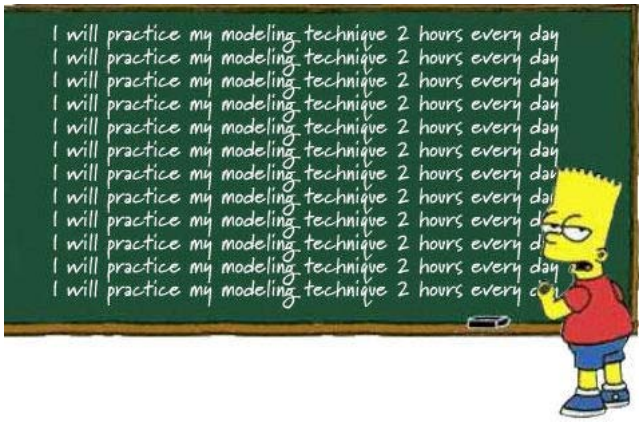
One-way data binding

*"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."*

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

Workshop

- Use your own component, add validation to the props it receives.
- Check different types: `String`, `Number`, `Boolean`, and so on
- Write a validation function on a string.
  - Use the `.includes()` (array), or `indexOf()` (string) methods to check if a requested value is available.
- Optional: use a `default` value for props!
  - We haven't covered this, look this up for yourself
  - https://vuejs.org/guide/components/props.html#prop-validation
- Generic example: `../210-props-validation`



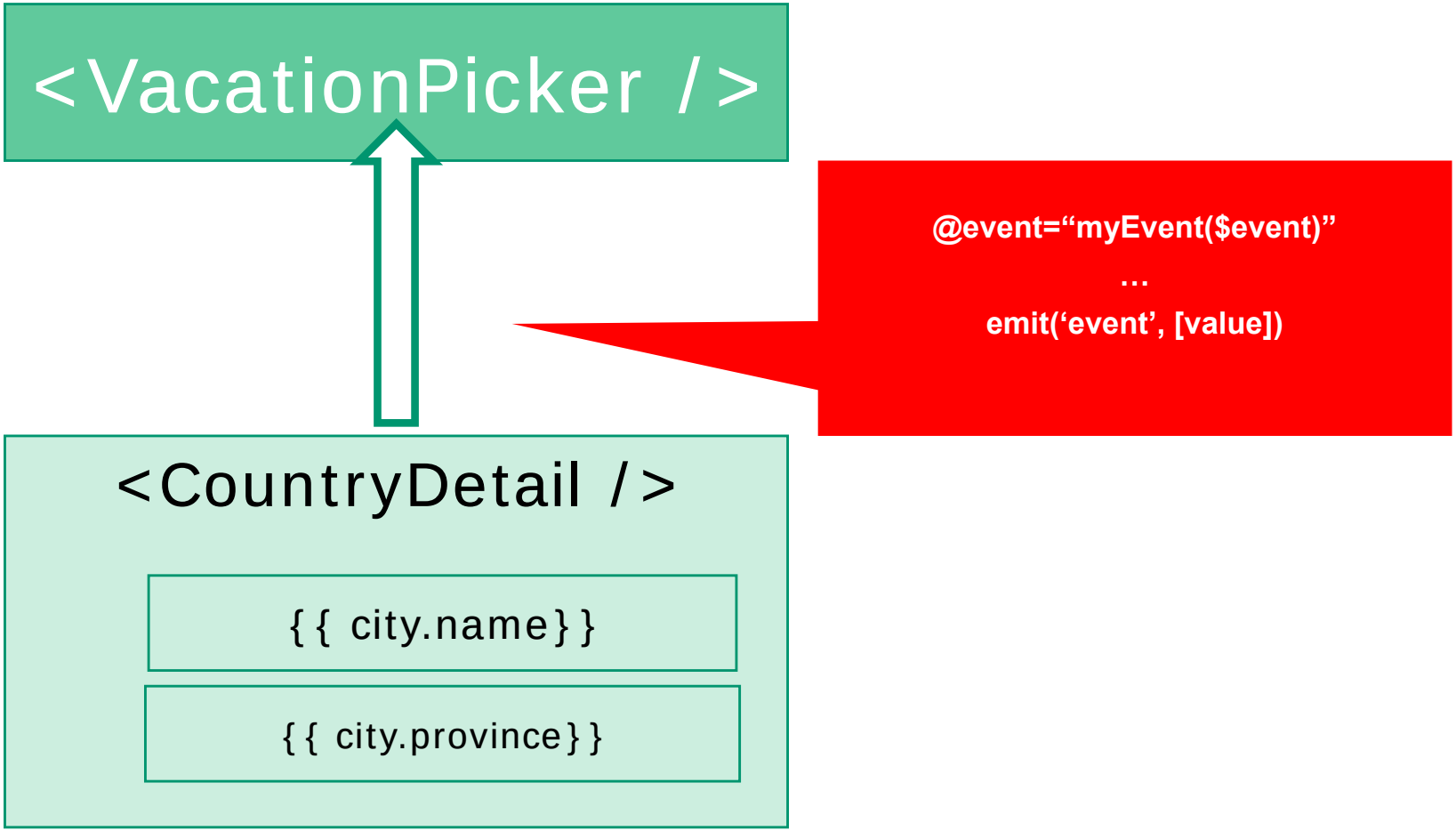
Passing data back

Communicating from child to parent component by sending events


Peter Kassenaar
INCLUSIEF GRATIS WEBVERSIE VAN HET BOEK
Vue.js
Web Development Library
VANDUUREN MEDIA

P. 148 e.v.

Child-Parent flow: custom events



Binding to custom events

- Child component can throw custom events, by using the `emit('eventName')` method in `<template>`
  - It is automatically available on every component
  - You can define the name of the event yourself
  - Optional: pass data in the event
- In the parent component, use the well-known `@eventName="handler($event)"` notation
  - Call a local event handler to handle the event
  - `$event` is a magic variable, containing the value, send from the child

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

Vue 3 – register the event to emit

- You can emit *directly* from the template, with something like
  - `<button @click="$emit('myEvent')>Click me</button>`
- Often times however, you don't want to much logic in the template
  - Emit from a function
- When emitting from a function in the script, use the built-in `defineEmits()` method.
- Otherwise, you'll get an error/warning in the browser console

```
const emit = defineEmits(['rating', 'favorite'])
…
const setRating = value => {
  emit('rating', value);
}
```

Example custom events - Child

Prepare the child component to emit its custom event(s)

```
<span class="float-right">
    <button @click="setRating(1)">+1</button>
    <button @click="setRating(-1)">-1</button>
</span>
```

```
const emit = defineEmits(['rating', 'favorite’])
…
const setRating = value => {
  // optional: do other stuff
  // finally: throw your event + value
  emit('rating', value);
}
```

Example custom events – parent

Prepare the parent component to receive custom event(s)

```
<CountryDetail v-if="showDetails"
               @rating="onRating($event)"
               :country="country" />
```

1. Catch event

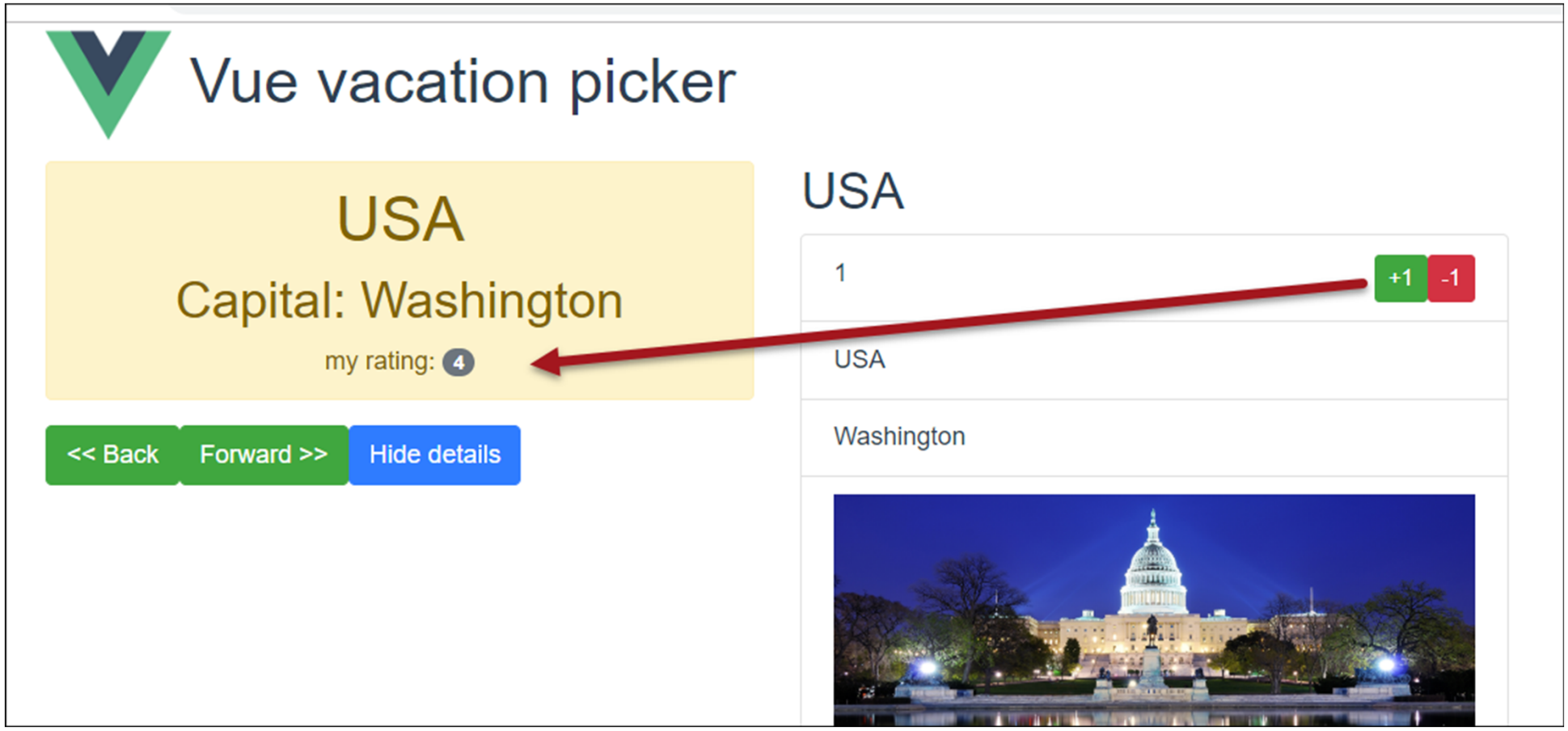
```
const currentRating = ref(0)
const onRating = value => {
  data.countries[currentCountryId.value].rating += value;
  currentRating.value += value;
}
```

2. Handle event

```
<div v-if="country.rating !== 0">
    my rating:
    <span class="badge bg-secondary badge-pill">{{currentRating}}</span>
</div>
```

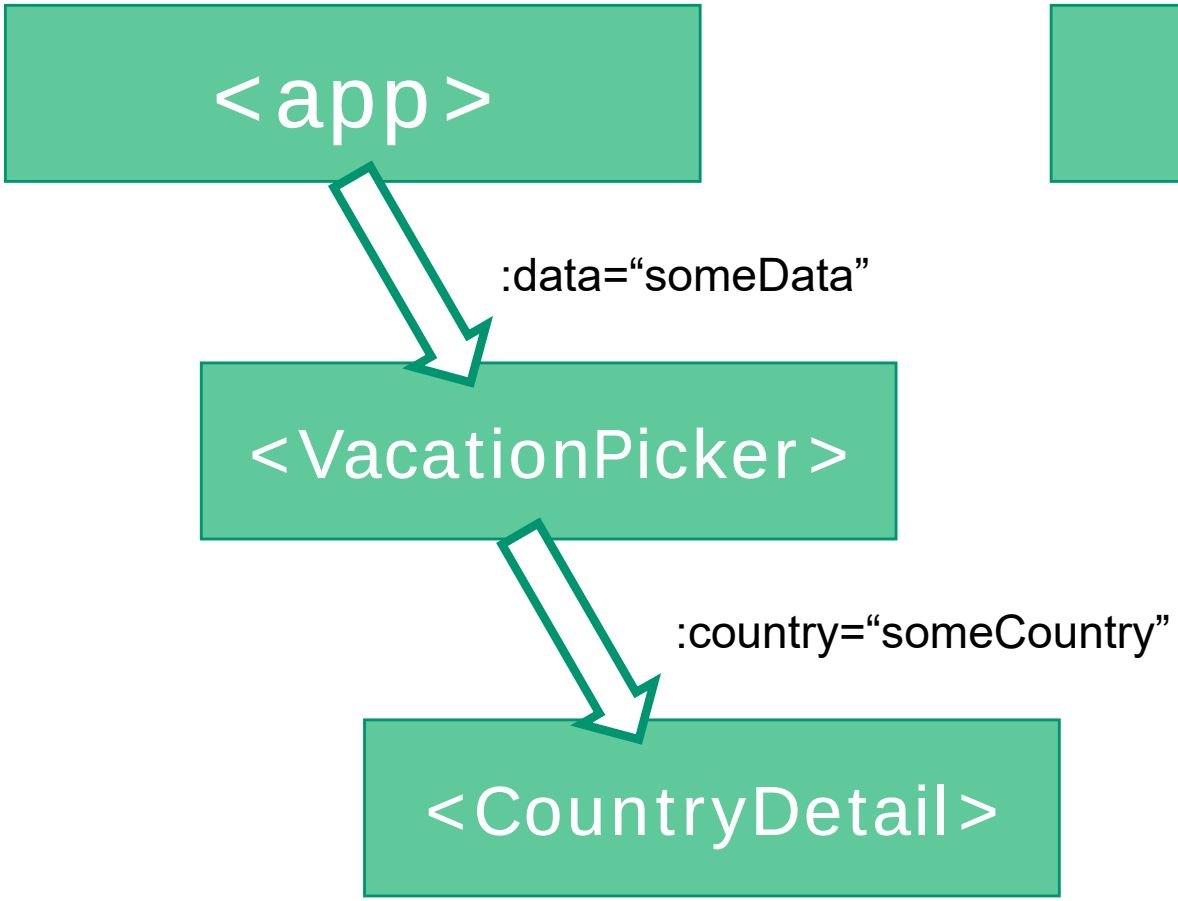
3. Show result in UI

Result

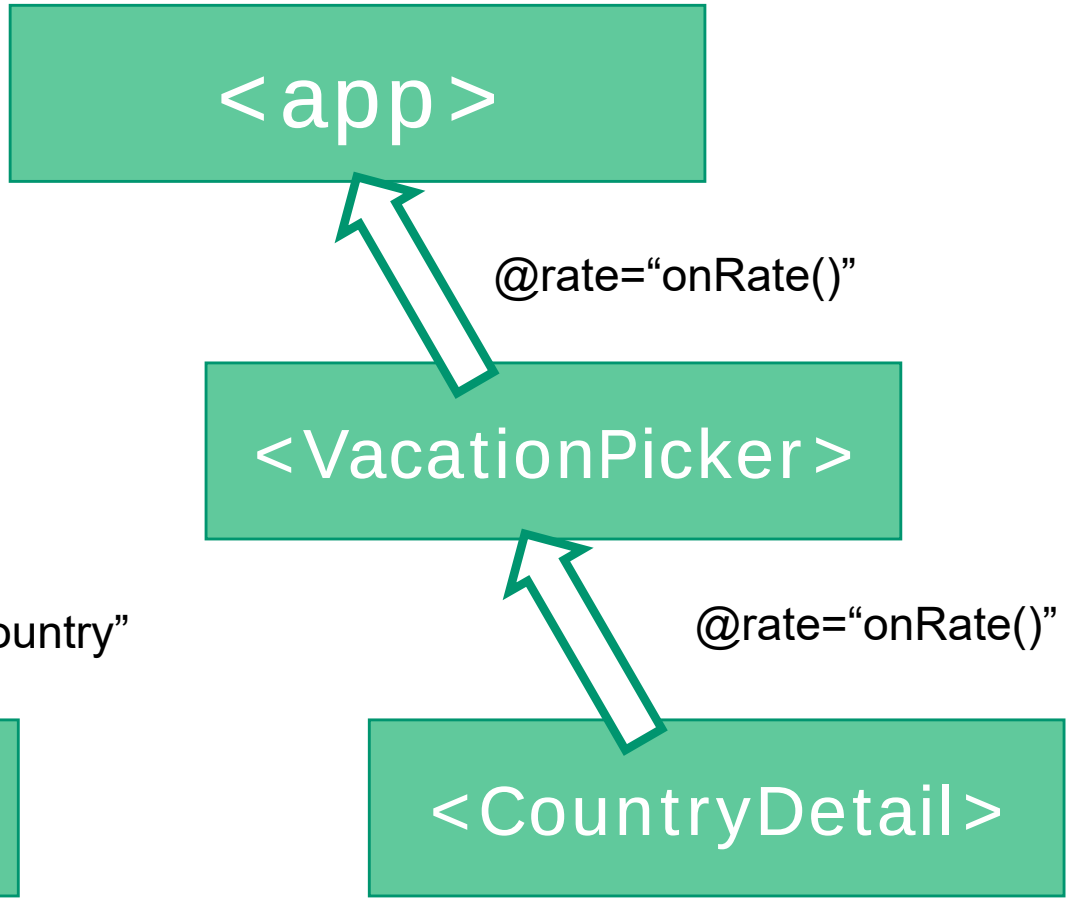

Vue vacation picker
USA
Capital: Washington
my rating: 4
<< Back
Forward >>
Hide details
USA
1
+1
-1
USA
Washington

Summary

Parent → Child

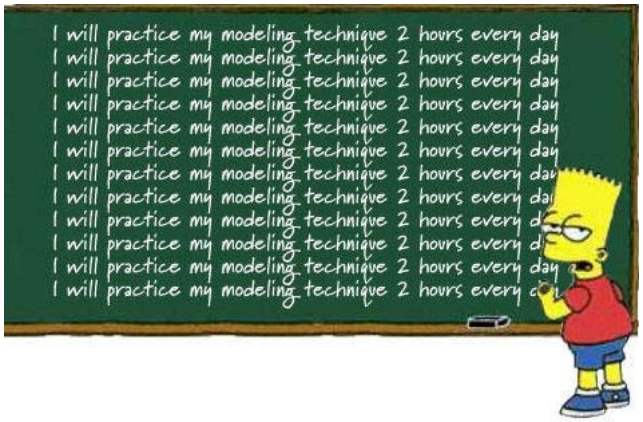


Child → Parent



Workshop

- Use `../220-emit-events` as a source, or use your own project
- Add a `favorite` event to the CountryDetail component, so a user can mark a country as favorite.
  - Update the data model with a `favorite` property.
  - Update the child component to `emit` or `$emit` the event.
  - Update the parent component to receive and handle the event
- Generic example: `../220-emit-events`



Injecting content

Using slots on the child component


Peter Kassenaar
INCLUSIEF GRATIS WEBVERSIE VAN HET BOEK
Vue.js
Web Development Library
VANDUUREN MEDIA

P. 154 e.v.

Inject UI into a component

- Why? With `props`, you inject *JavaScript variables*.
- But sometimes you want to pass *template content*.
- We then use a `<slot>`.
- For instance, we want `<CountryDetail />` to be in a collapsible `div`.
  - The show/hide content is on the header of this `div`, instead of somewhere else
- The structure then becomes like:

```
<CollapsibleSection>
    <CountryDetail @rating="onRating($event)"
                   :country="country" />
</CollapsibleSection>
```

https://vuejs.org/v2/guide/components-slots.html

Reusing components

- We create a reusable component `<CollapsibleSection />`, that takes all kinds of content
- Convention: put reuseables in a `\shared` folder
  - This is NOT required. Simply a convention.
- We can then also simplify our parent component
  - No more button needed (as the `CollapsibleSection` is responsible for showing/hiding content)
  - No more variable and `v-if` needed on the `CountryDetail` component (idem).

Structure of `CollapsibleSection`

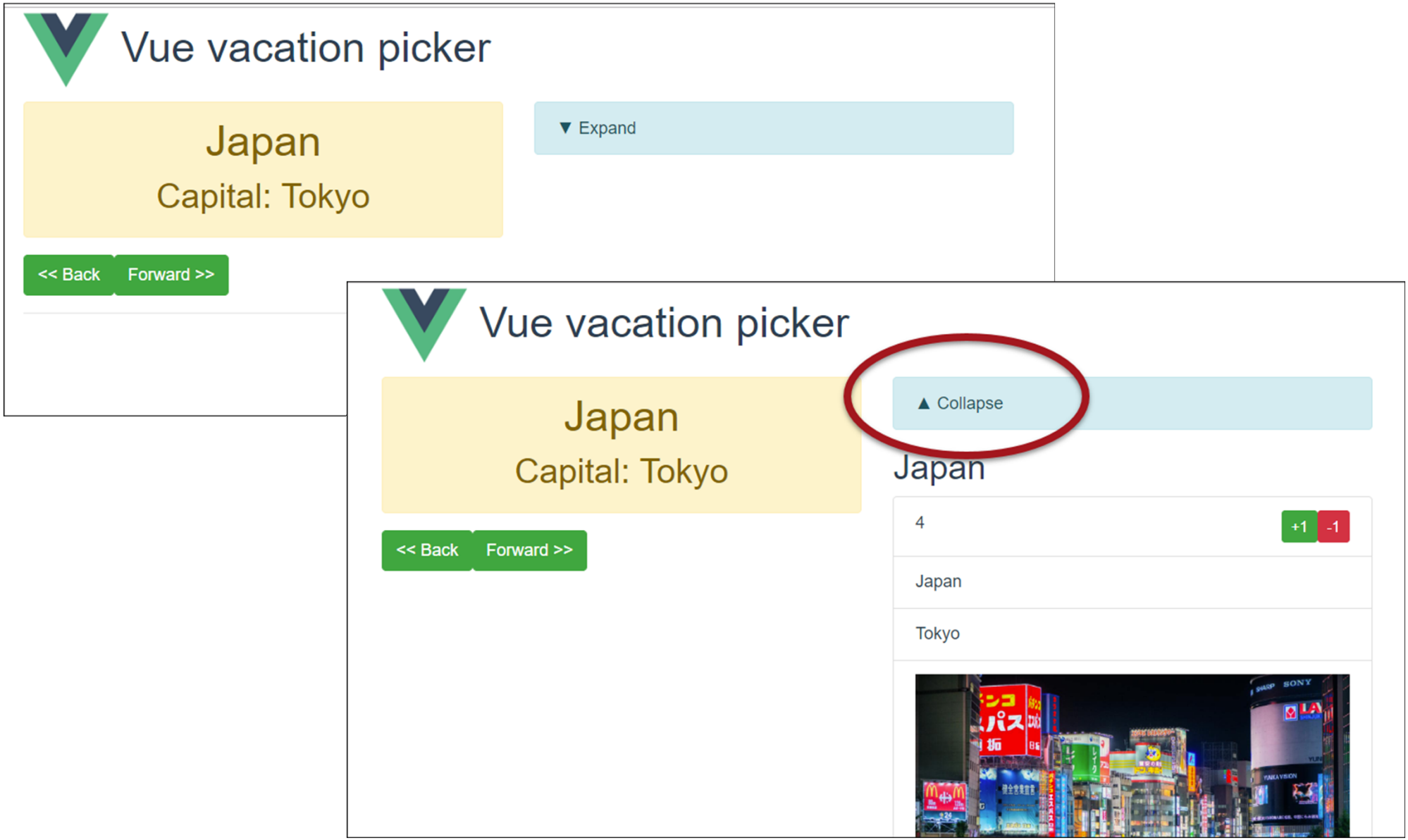
Just a template and a toggle/flag open:

```
<template>
  <div>
    <div class="alert alert-info" style="cursor: pointer">
      <span v-if="open" @click="open = !open">&#x25B2; Collapse</span>
      <span v-if="!open" @click="open = !open">&#x25BC; Expand</span>
    </div>
    <!--Injected content here-->
      <slot v-if="open"></slot>
  </div>
</template>

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

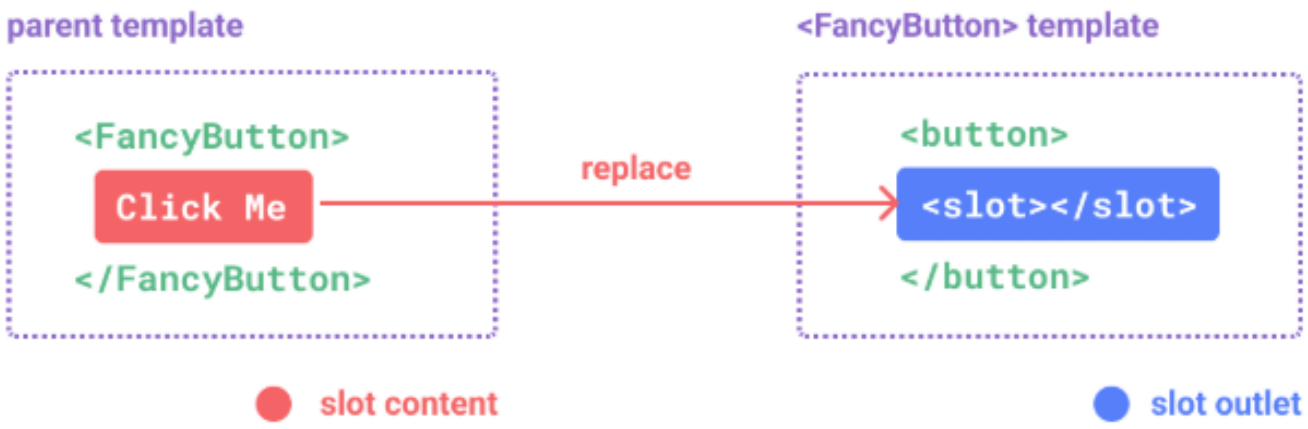
- The `&#x25B2` is just the HTML code for up/down arrow
- We use a simple bootstrap `alert` class here
- We give the header a `style` so a cursor is shown
- The `<slot>` is where the magic happens
- It is only visible if the collapsible is open

Result


Vue vacation picker
Japan
Capital: Tokyo
▼ Expand
<< Back
Forward >>
Vue vacation picker
Japan
Capital: Tokyo
▲ Collapse
Japan
4
+1
-1
Japan
Tokyo
<< Back
Forward >>

From documentation

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

Extra info on <slot> ‘s

- You can add default content inside a slot, like so:
  - `<slot> <div>…my default content…</div> </slot>`
- We can pass data into shared/reusable/slot component with props like normal.
- As you saw, slots can contain, HTML, or other components.
- We can have multiple slots on a component (see next slide)
  - Every slot gets its own name
  - You can target a slot by using its `name` in the parent component
  - Unnamed slots act as a ‘catch all’ slot for unnamed content
  - Their name is implicitly `<slot name="default">`

Multiple slots in a component

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

Option 1: using `template` tag

```
<template v-slot:header>
    <h1>This is the page title</h1>
</template>
<p>No name - so a paragraph for the main content.</p>
<p>And another one.</p>
<template v-slot:footer>
    <p>Footer contains contact info, disclaimer, etc</p>
</template>
```

Option 2: using the `v-slot` shorthand

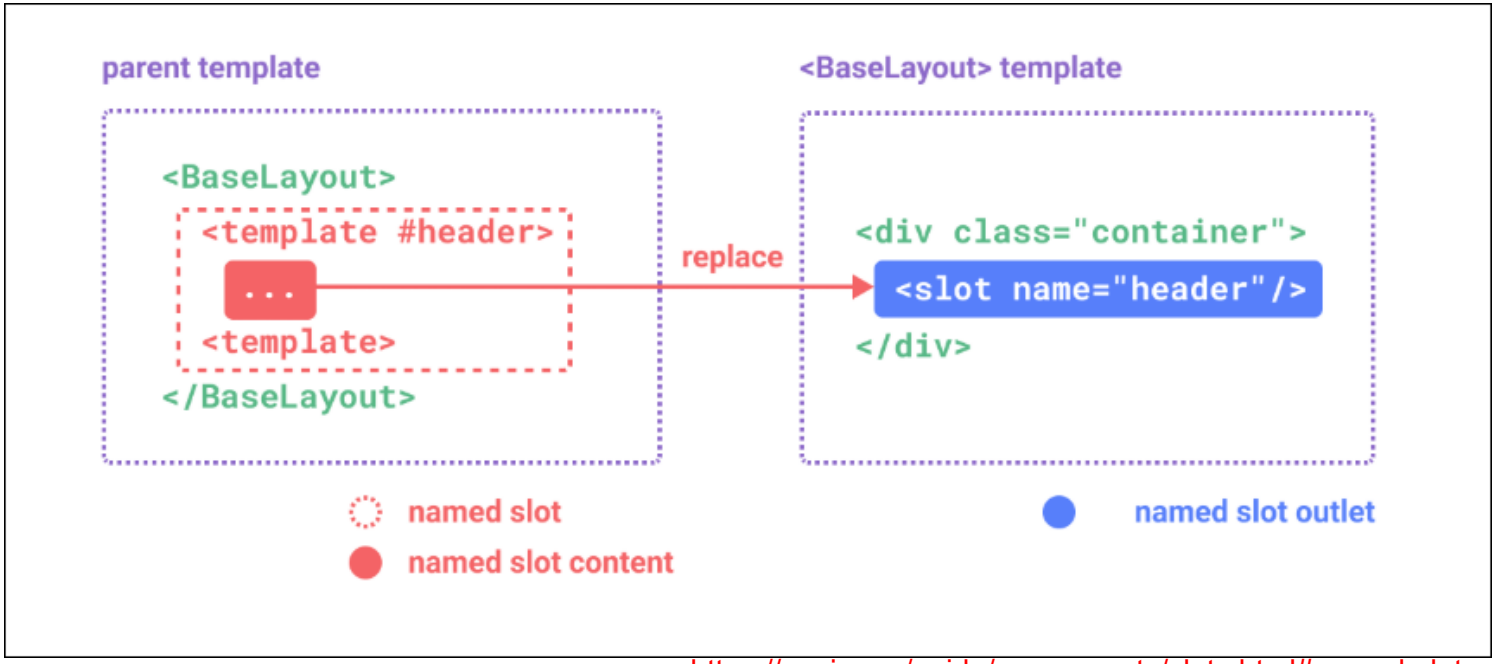
```
<template #header>This is the page title</template>

<p>No name - so a paragraph for the main content.</p>
<p>And another one.</p>

<template #footer">Footer contains contact info, disclaimer, etc</template>
```

https://vuejs.org/guide/components/slots.html#named-slots

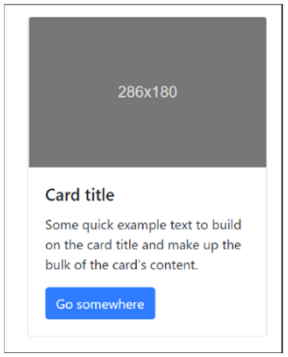
From the Vue documentation:

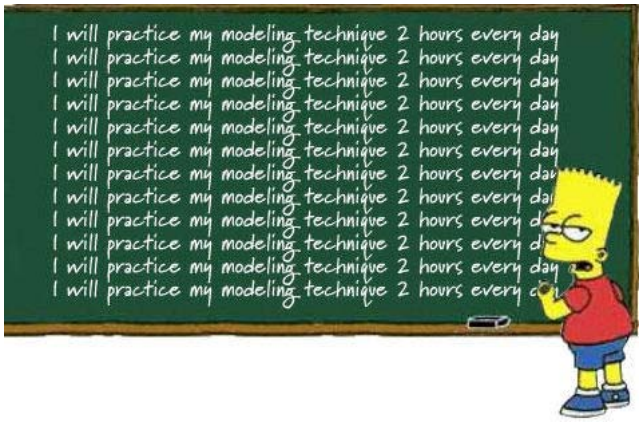


https://vuejs.org/guide/components/slots.html#named-slots

Workshop

- Use `../230-slots` as a source, or use your own project
  - In your own project: create a generic component using slots
- In example project: Create a new component, designed as a Bootstrap `Card` component
  - Create a `.vue` component and use slots to inject content
  - Documentation: https://getbootstrap.com/docs/5.0/components/card/
  - Call this component inside the `CountryDetail` component and pass data to the correct slots.
  - I.e. We want your `CountryDetail` to look like a Bootstrap `Card`.
- Generic example: `../230-slots`





Animation

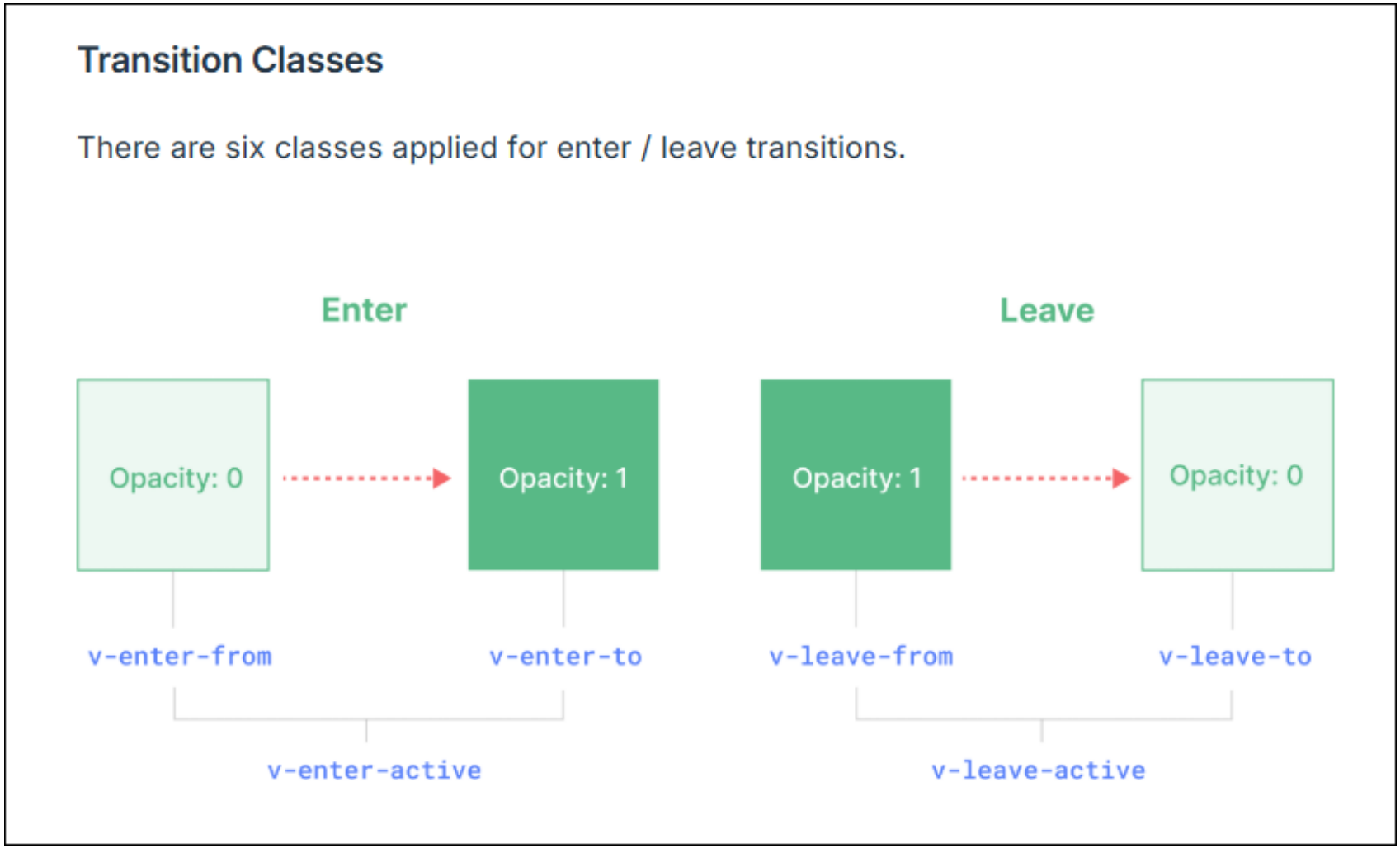
- You can animate content if you want to
  - Use the `<Transition name="someName">…</Transition>` element as a wrapper
  - Write CSS-classes providing the transformation / animation
- For instance:

```
<!--Injected content here-->
<Transition name="fade">
    <slot v-if="open"></slot>
</Transition>
```

```
<style scoped>
    .fade-enter-from, .fade-leave-from {
        transition: opacity .5s;
    }
    .fade-enter-to, .fade-leave-to {
        opacity: 0;
    }
</style>
```

https://vuejs.org/v2/guide/transitions.html

Use predefined class names on <Transition>



https://vuejs.org/guide/built-ins/transition.html#css-based-transitions

Checkpoint

- You know how to pass data down the component chain by creating and using props.
- You know about extending props with types and validating props.
- You can pass data back up the component chain by creating and capturing custom events.
- You know about working with [multiple] slots in your project to project content from parent components.