

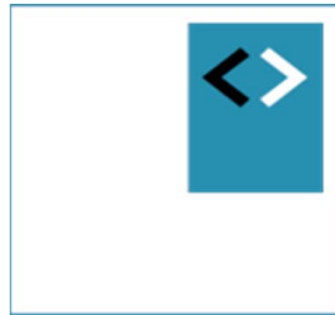


Vue Fundamentals - 04

Component communication



nationale
nederlanden

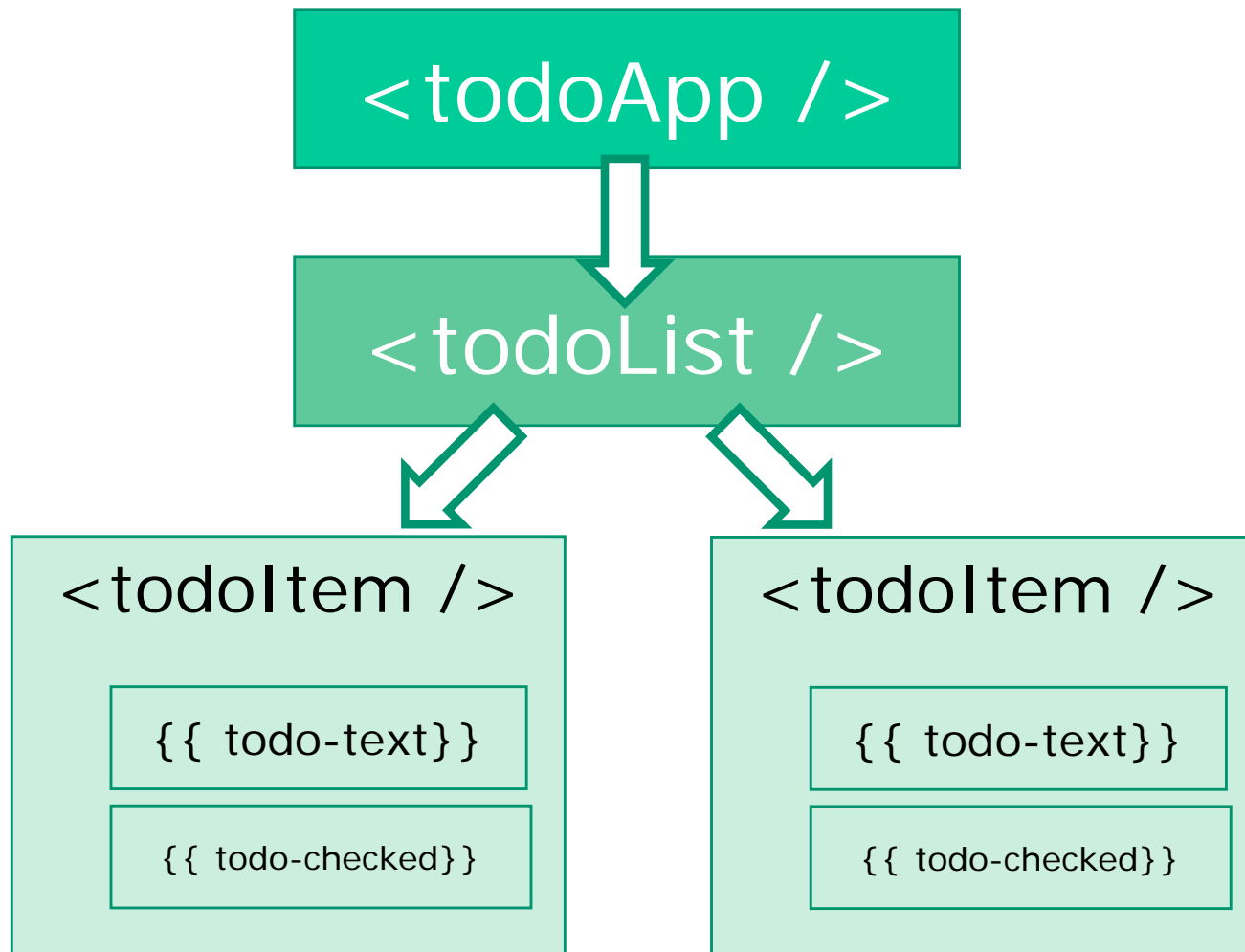


Peter Kassenaar –
info@kassenaar.com

Contents

- **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 slots

Vue app : Tree of components



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 Component and move the HTML from the parent Component

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

<script>
  export default {
    name: "CountryDetail",
  }
</script>
```

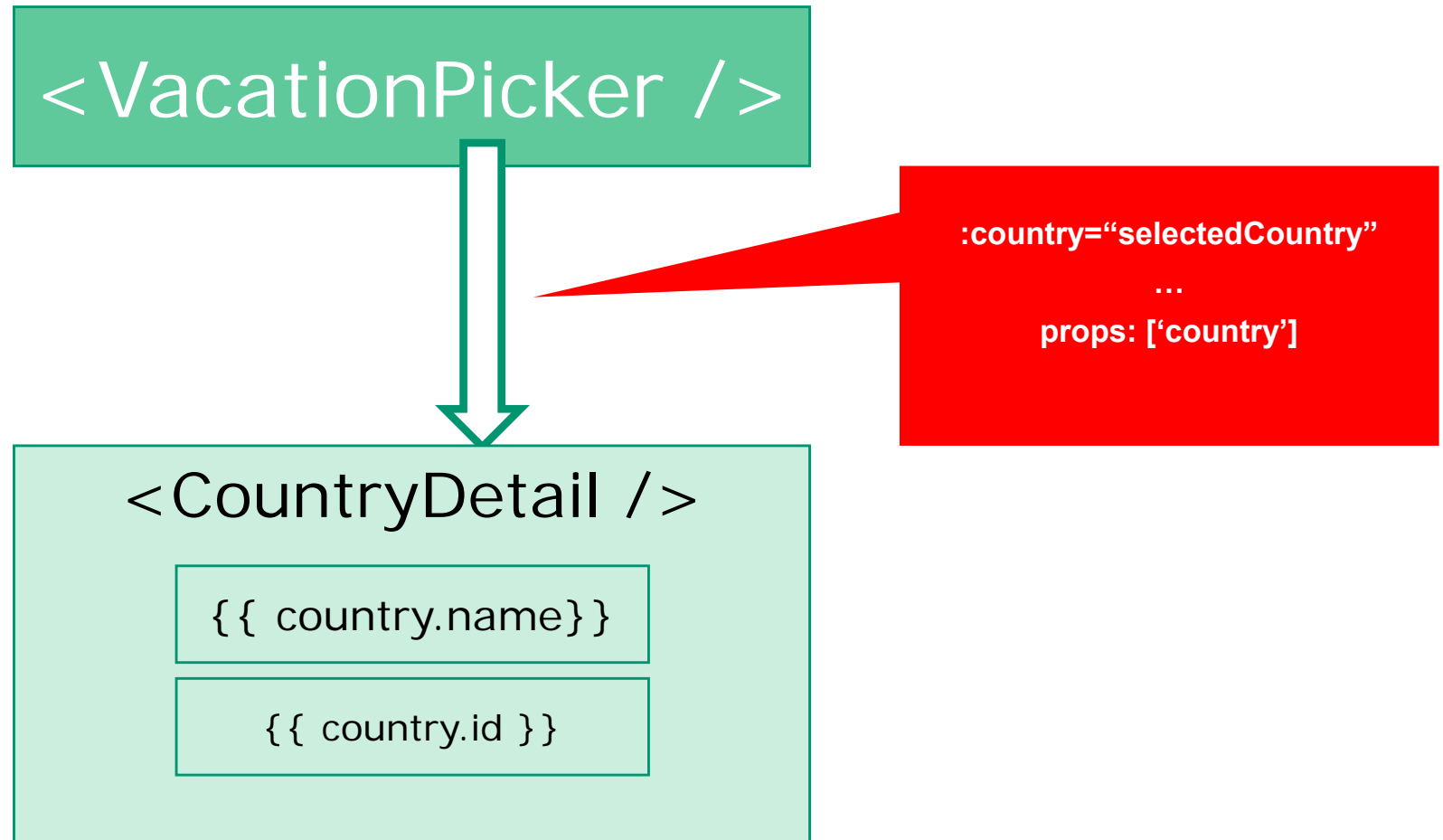


Data flow between components

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

*Data flows out of a component via
@event events"*

Parent-Child flow: de 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 is an array on the component, like:

```
export default {  
  name: "CountryDetail",  
  props: ['country'],  
}
```

We can then bind to the properties of

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

2. Update Parent component to send data down



```
<div class="col-6">
  <CountryDetail v-if="showDetails" :country="country" />
</div>
...
<script>
  // import the country data
  import data from '../data/data';
  import CountryDetail from "../CountryDetail";

  export default {
    name: 'VacationPicker',
    components: {CountryDetail},
    ...
  },
</script>
```



Move methods and computed properties

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

```
export default {
  name: "CountryDetail",
  props: ['country'],
  methods: {
    getImgUrl(img) {
      console.log(img);
      return require('../assets/countries/' + img);
    }
  },
  computed: {
    isExpensive() {
      return this.country.cost > 4000;
    },
    isOnSale() {
      return this.country.cost < 1000;
    }
  }
}
```

Casing of props

- HTML attributes are case-*insensitive*
- If you use camelCase on prop-names, use a hyphen in the html
 - i.e. props: ['countryDetail', 'countryName'] become
`<DetailComponent country-detail="..." country-name="..." />`
- If you are using string templates this limitation does not apply

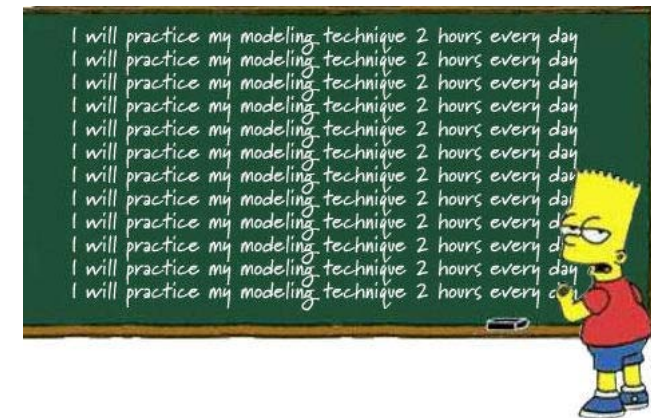
```
JS
Vue.component('blog-post', {
  // camelCase in JavaScript
  props: ['postTitle'],
  template: '<h3>{{ postTitle }}</h3>'
})
```

```
HTML
<!-- kebab-case in HTML -->
<blog-post post-title="hello!"></blog-post>
```

<https://vuejs.org/v2/guide/components-props.html>

Workshop

- Create an extra prop on the `CountryDetailComponent` and pass it.
- New: 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.
- Optional: implement the lifecycle hook `beforeupdate` on the child component, showing a counter that says how many times the component is updated.
- Generic Example on props: [../200-props](#)





Validating props

Making sure only specific kinds of data get passed

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)

Usually though, you'll want every prop to be a specific type of value. In these cases, you can list props as an object, where the properties' names and values contain the prop names and types, respectively:

```
props: {  
  title: String,  
  likes: Number,  
  isPublished: Boolean,  
  commentIds: Array,  
  author: Object  
}
```

JS

Simple validation of CountryDetail props

```
export default {  
  name: "CountryDetail",  
  props: {  
    country: {  
      type: Object,  
      required: true  
    },  
    name: {  
      type:  
        String, required: true  
    }  
  },  
  ...  
}
```



Errors if you give prop the wrong value


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



```
WDS...  
✖ [Vue warn]: Invalid prop: vue.runtime.esm.js?2b0e:601  
  type check failed for prop "country". Expected Object,  
  got String with value "test".  
  
found in  
  
---> <CountryDetail> at src/components/CountryDetail.vue  
      <VacationPicker> at  
src/components/VacationPicker.vue  
      <App> at src/App.vue  
      <Root>  
  
undefined CountryDetail.vue?0eaf:39
```


Errors if you do not pass a required prop

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



```
[HMR] Waiting for update signal from WDS... log.js?latd:24  
✖ [Vue warn]: Missing required prop: "name"  
found in  
---> <CountryDetail> at src/components/CountryDetail.vue  
      <VacationPicker> at  
src/components/VacationPicker.vue  
      <App> at src/App.vue  
      <Root>  
washington.jpg CountryDetail.vue?0eaf:39  
> |
```

The application *will* continue to run, but shows the error in the 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:function (value) {
    return [1, 2, 3, 4, 5, 6].includes(value)
  }
}
```

Errors in console on validation

```
countries: [  
  {  
    id: 100,  
    name: 'USA',  
    capital: 'Washington',  
    ...  
  },  
],
```



```
[HMR] waiting for update signal from WDS... log.js?1a1d:24  
✖ [Vue warn]: Invalid prop: vue.runtime.esm.js?2b0e:601  
custom validator check failed for prop "id".
```

found in

```
---> <CountryDetail> at src/components/CountryDetail.vue  
      <VacationPicker> at  
src/components/VacationPicker.vue  
      <App> at src/App.vue  
      <Root>
```

washington.jpg

CountryDetail.vue?0eaf:46

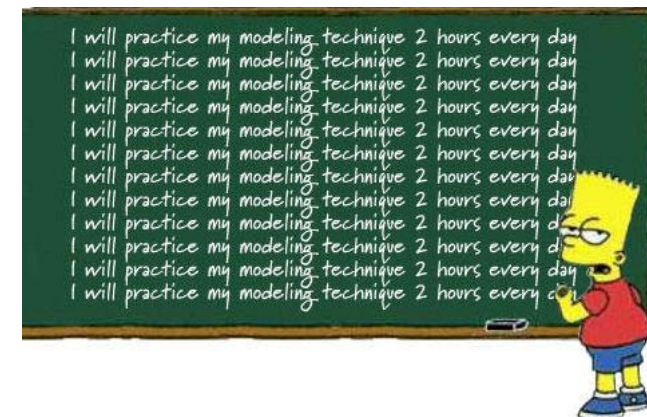
> |

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.”*

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://alligator.io/vuejs/property-validation/>
- Generic example: [../210-props-validation](#)

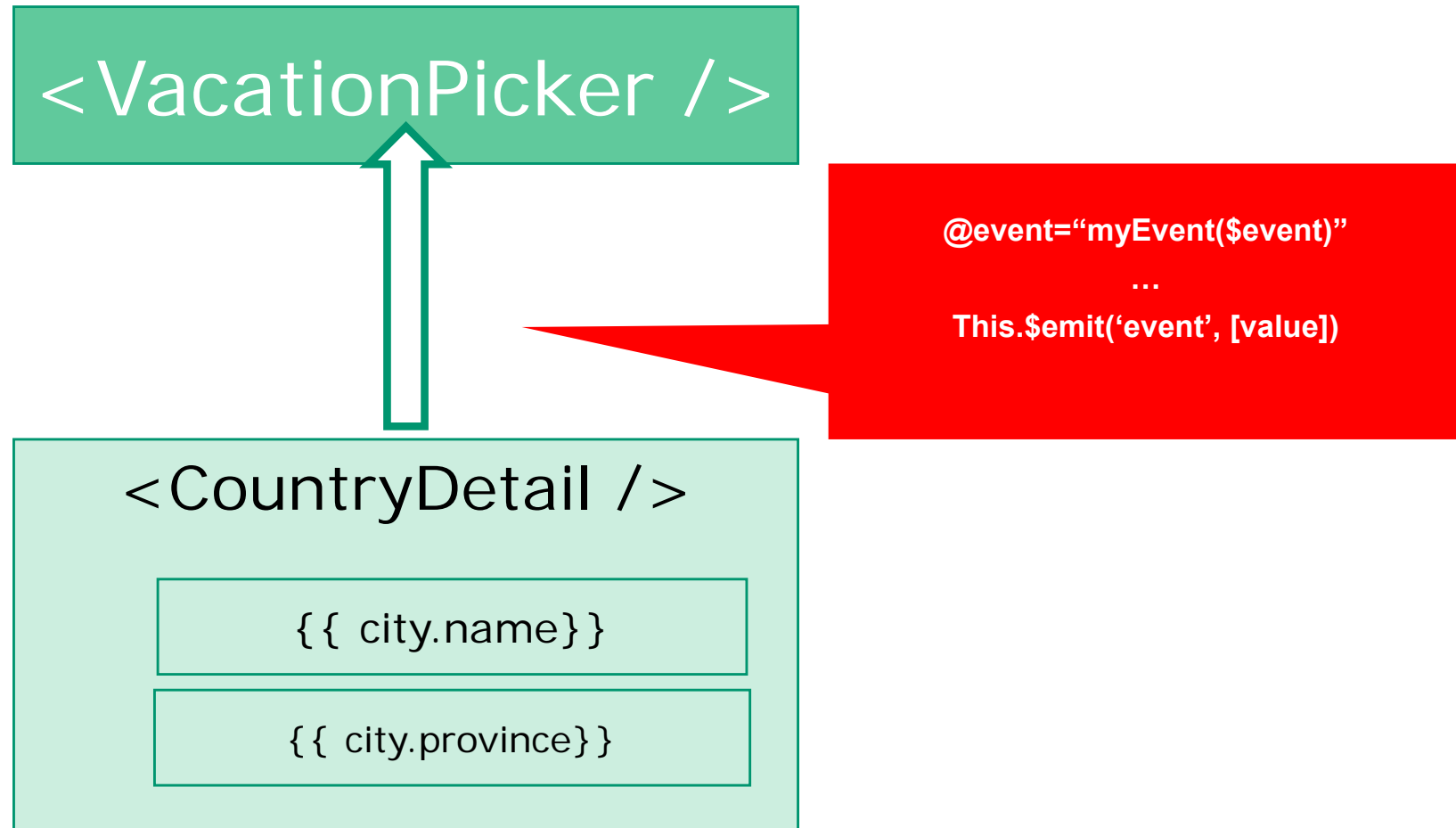




Passing data back

Communicating from child to parent component by sending events

Child-Parent flow: custom events



Binding to custom events

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

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>
```



```
methods: {  
  ...  
  setRating(value){  
    this.$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


```
onRating(rating){
  console.log('rating received for ' + this.country.name);
  this.data.countries[this.currentCountryIndex].rating += rating;
}
```

2. Handle event

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

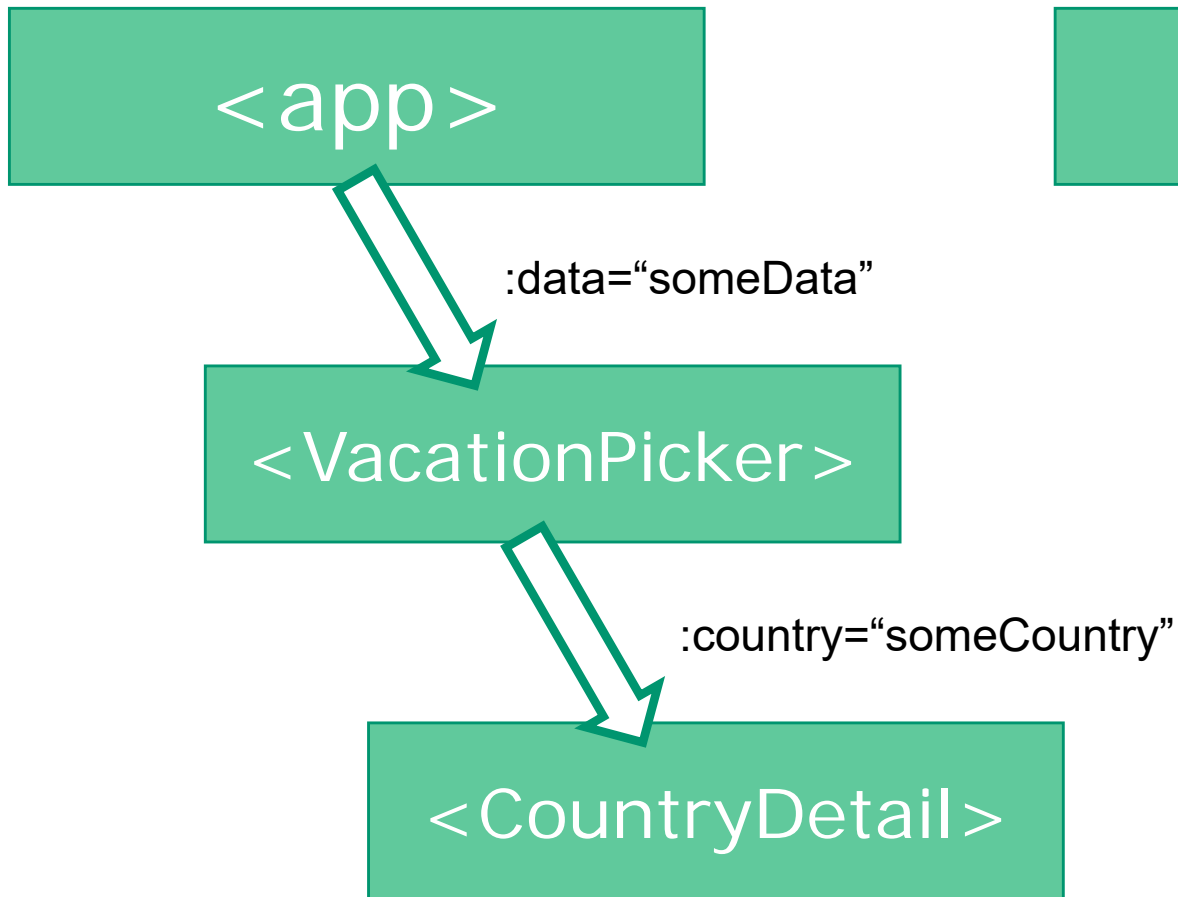
USA

Washington

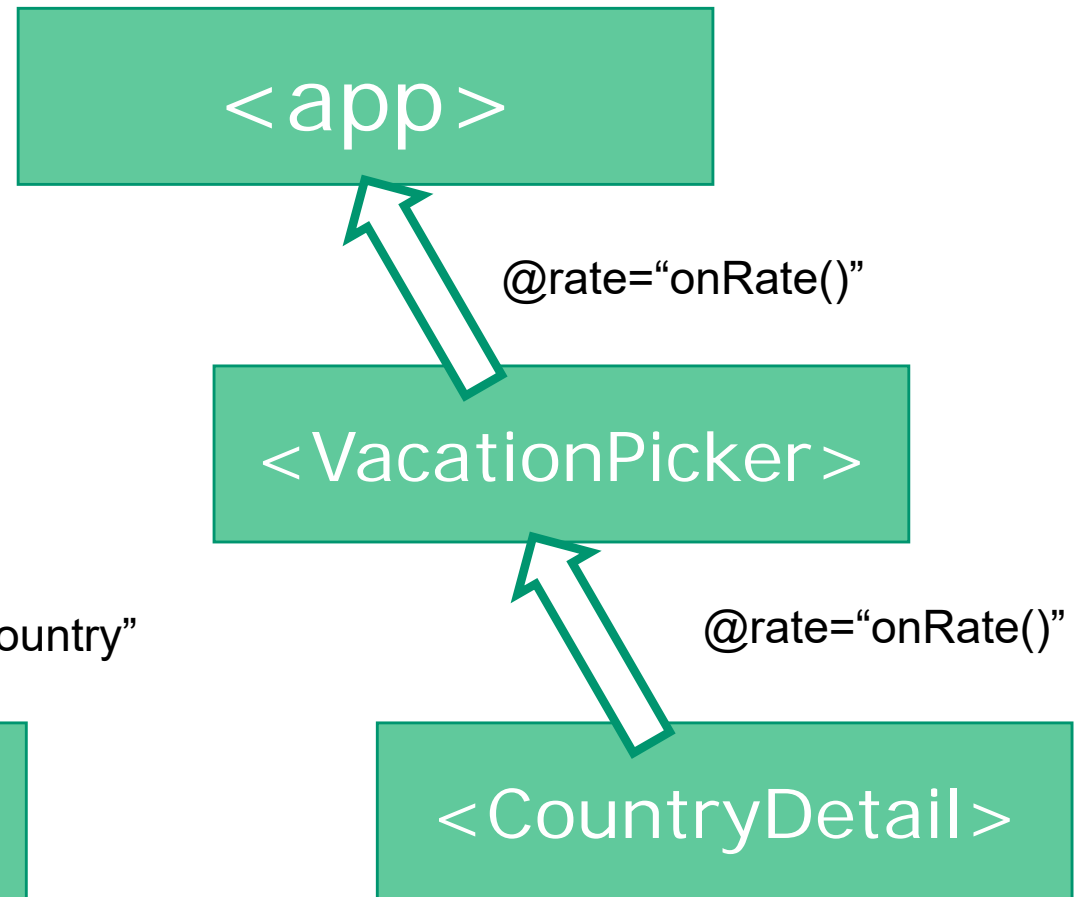


Summary

Parent → Child

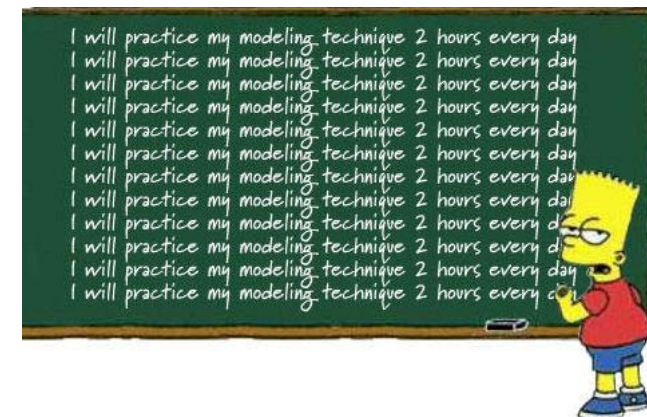


Child → Parent



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 the event.
 - Update the parent component to receive and handle the event
- Generic example: `../220-emit-events`





Independent event bus

Passing data without a direct parent/child relationship

Event bus in Vue

- <https://medium.com/easyread/vue-as-event-bus-life-is-happier-7a04fe5231e1>



Injecting content

Using slots on the child component

Inject data into a component

- Sometimes you want to create a component that you can pass content into.
- This component is responsible for showing the data in the right place.
- 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>
```

Reusing components

- We create a reusable component `<CollapsibleSection />`, that takes all kinds of content
- Best practice: put reuseables in a `\shared` folder
- 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

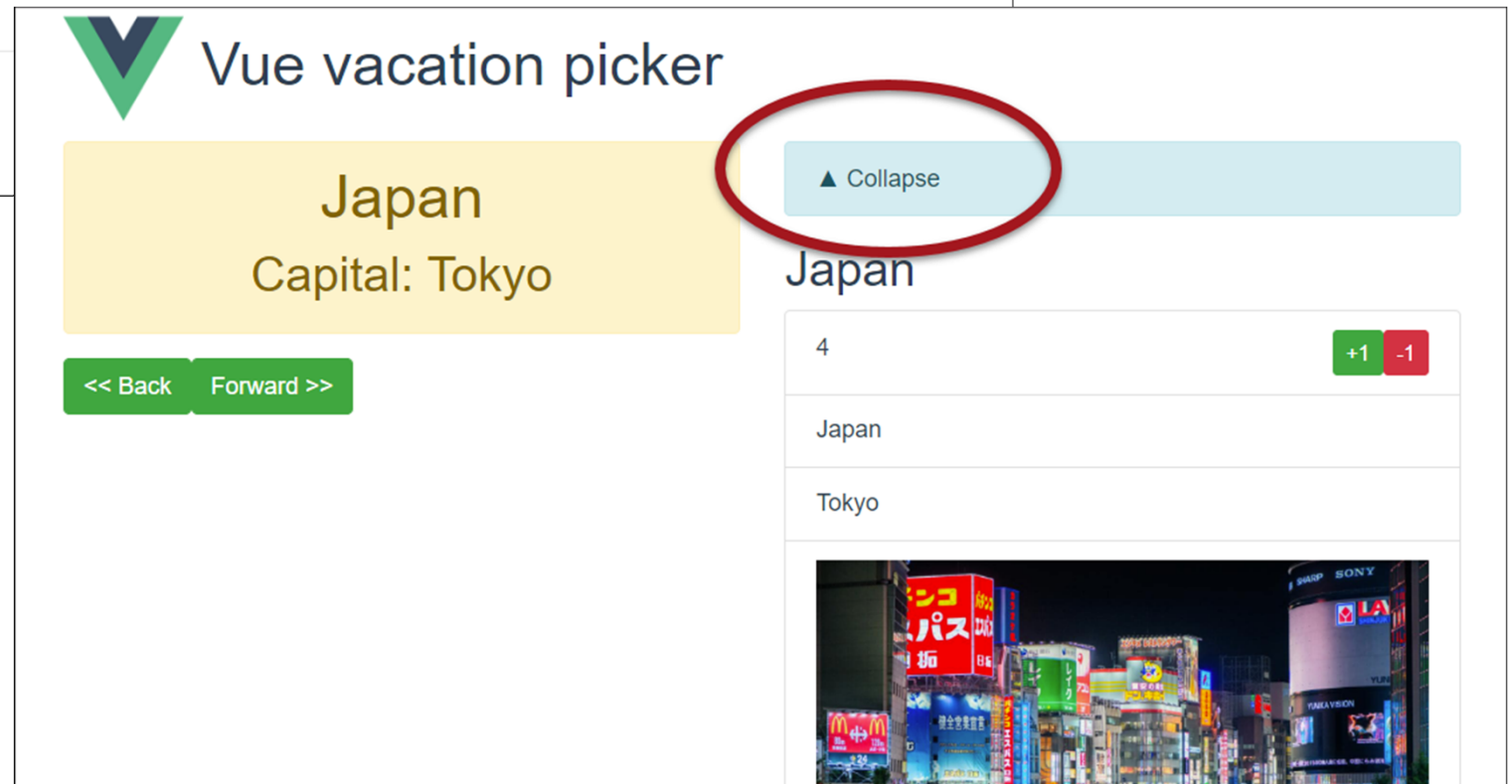
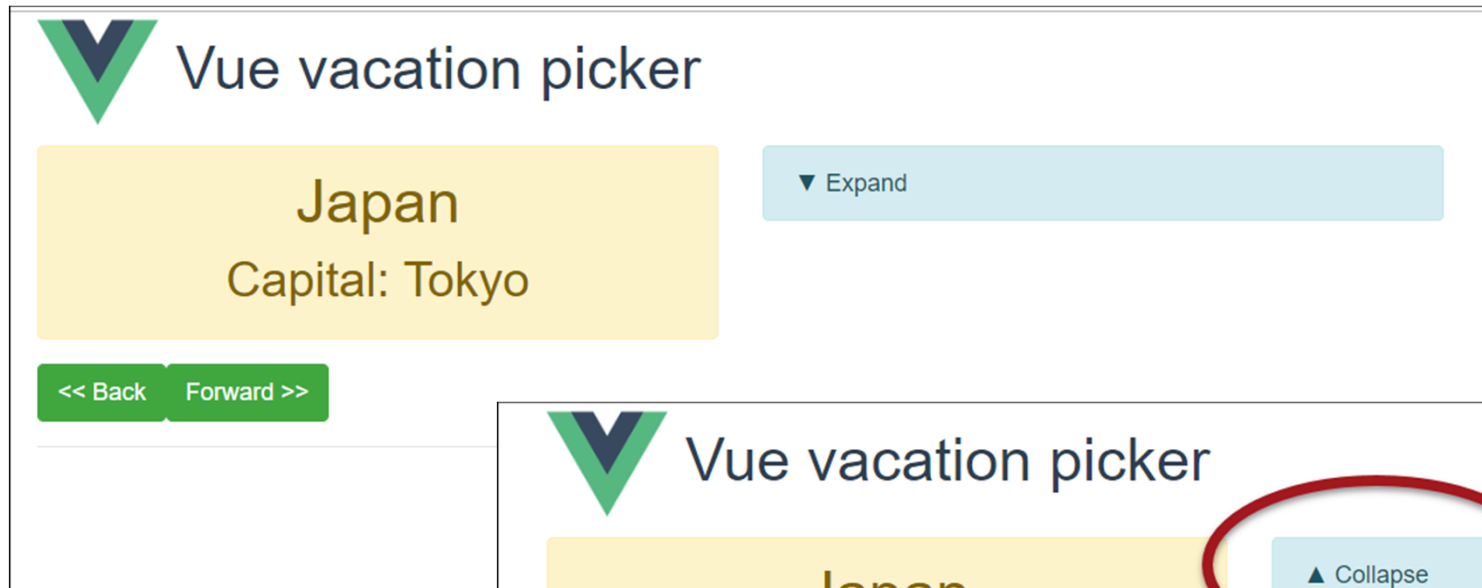
Just a template and a toggle/flag open:

```
<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>
  export default {
    name: "CollapsibleSection",
    data(){
      return {
        open: false
      }
    }
  }
</script>
```

- The ▲ 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



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

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 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 slot="footer">
  <p>Footer contains contact info, disclaimer, etc</p>
</template>
```

Option 2: simple HTML,
using the slot attribute

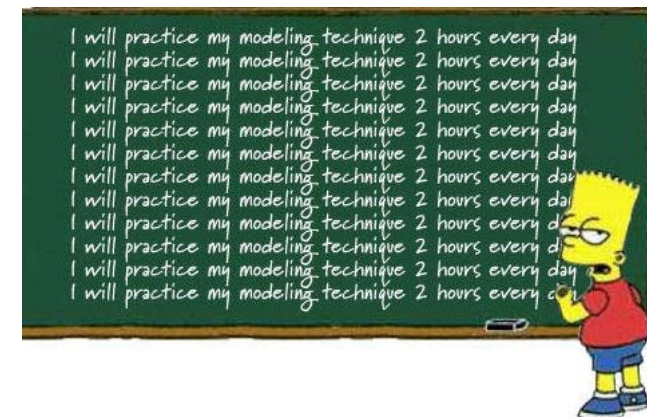
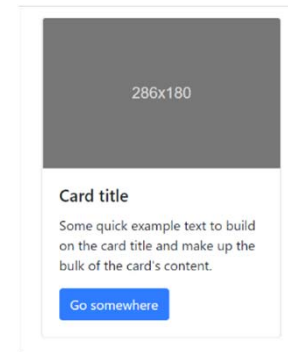
```
<h1 slot="header">This is the page title</h1>

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

<p slot="footer">Footer contains contact info, disclaimer, etc</p>
```

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/4.0/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-active, .fade-leave-active {  
    transition: opacity .5s;  
  }  
  .fade-enter, .fade-leave-to {  
    opacity: 0;  
  }  
</style>
```


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.