



Global Knowledge®



# Vue Fundamentals

## Module – Style bindings & in depth components

Peter Kassenaar – [info@kassenaar.com](mailto:info@kassenaar.com)

### WORLDWIDE LOCATIONS

BELGIUM CANADA COLOMBIA DENMARK EGYPT FRANCE IRELAND JAPAN KOREA MALAYSIA MEXICO NETHERLANDS NORWAY QATAR  
SAUDI ARABIA SINGAPORE SPAIN SWEDEN UNITED ARAB EMIRATES UNITED KINGDOM UNITED STATES OF AMERICA



# Using v-model

Two-way databinding with Vue

Peter Kassenaar

INCLUSIEF  
GRATIS  
WEBVERSIE  
VAN HET  
BOEK

# Vue.js

Web Development Library



VANDUUREN MEDIA

P. 122 e.v.

## Using v-model to select changes

*"You can use the `v-model` directive to create **two-way data bindings** on form input, textarea, and select elements. It automatically picks the correct way to update the element based on the input type."*

## Using `v-model`

### *Two-way data binding*

Reflect changes in the UI back to the component  
and the other way around

```
<input type="text" v-model="...">
```


# Push items to (new) array

**New Countries**

Add Country

Canada

France



```
<input type="text" class="form-control-lg"  
      v-model="newCountry"  
      @keyup.enter="addCountry()"  
>  
<button @click="addCountry()" class="btn btn-info">  
  Add Country  
</button>
```

```
methods: {  
  selectCountry(index) {  
    this.selectedCountryIndex = index;  
  },  
  addCountry(){  
    this.newCountries.push(this.newCountry);  
    this.newCountry='';  
  },  
},  
computed: {
```

## Using `v-model` on check boxes

```
<input type="checkbox" v-model="...">
```

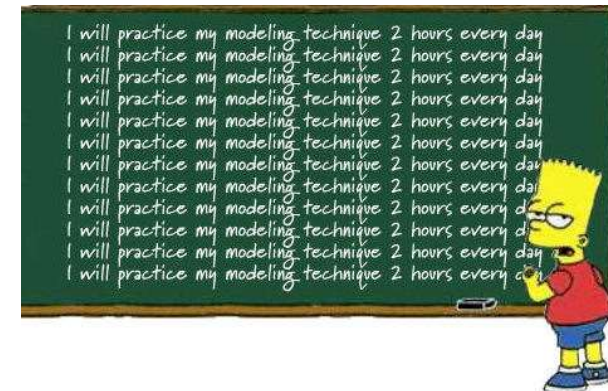


## Using `v-model` on radio buttons

```
<input type="radio" v-model="...">
```

# Workshop v-model

- Create a component with 2 input fields. The values you type in one field, are copied to the other field and vice versa
- Add checkboxes to your own data list. If a field is checked, it is added to an array and shown in the user interface
- **Optional:** create a textfield on one component.
  - Text that is typed in, is passed on as a `prop` to another component
  - See default `HelloWorld` component as an example for `props`
- Examples: [.../125-v-model](#), [126-...](#), [127-...](#), [128-...](#)



# Optional - modifiers for v-model

- Modifying the input, received from a `v-model` textbox
  - `.lazy`
  - `.number`
  - `.trim`
- <https://vuejs.org/v2/guide/forms.html#Modifiers>

## Modifiers

### # `.lazy`

By default, `v-model` syncs the input with the data after each `input` event (with the exception of IME composition as [stated above](#)). You can add the `lazy` modifier to instead sync after `change` events:

```
<!-- synced after "change" instead of "input" -->  
<input v-model.lazy="msg" >
```

HTML

### # `.number`

If you want user input to be automatically typecast as a number, you can add the `number`







# Component lifecycle hooks

Tapping into the lifecycle of created components

Peter Kassenaar

INCLUSIEF  
GRATIS  
WEBVERSIE  
VAN HET  
BOEK

Vue.js

Web Development Library



VANDUUREN MEDIA

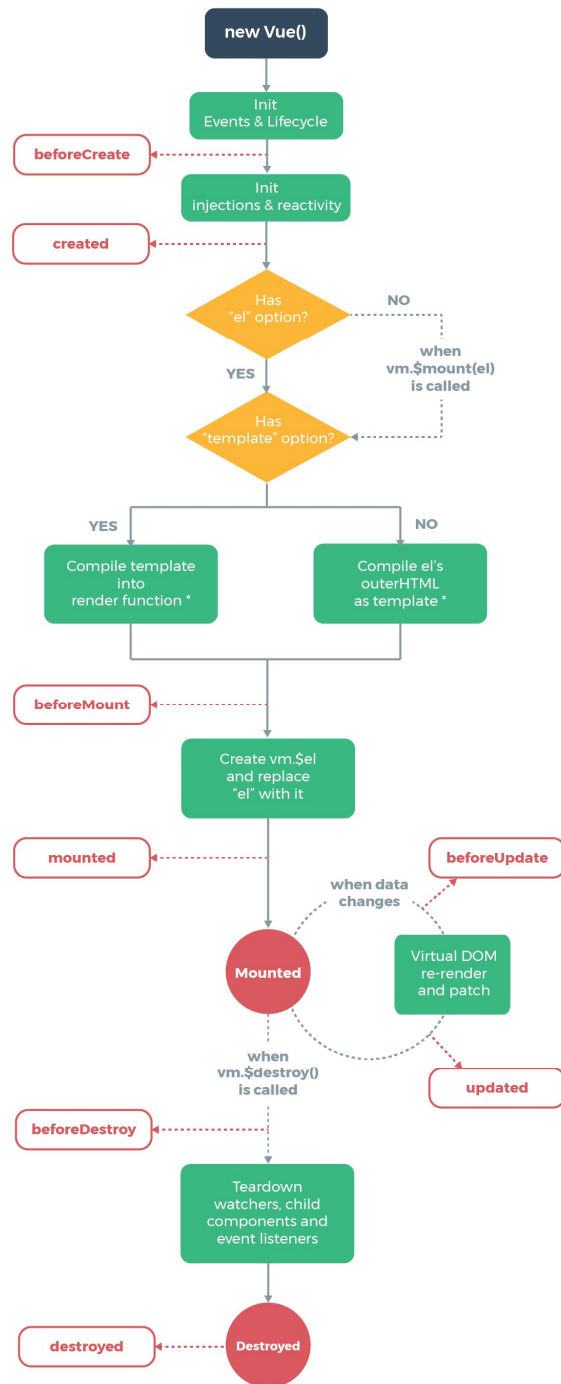
P. 118 e.v.

# Lifecycle hooks

- Perform an action automatically when a specific lifecycle event occurs

*“Each component instance goes through a series of initialization steps when it’s created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes.”*





## Official lifecycle diagram

The Red squares are the lifecycle hook methods.

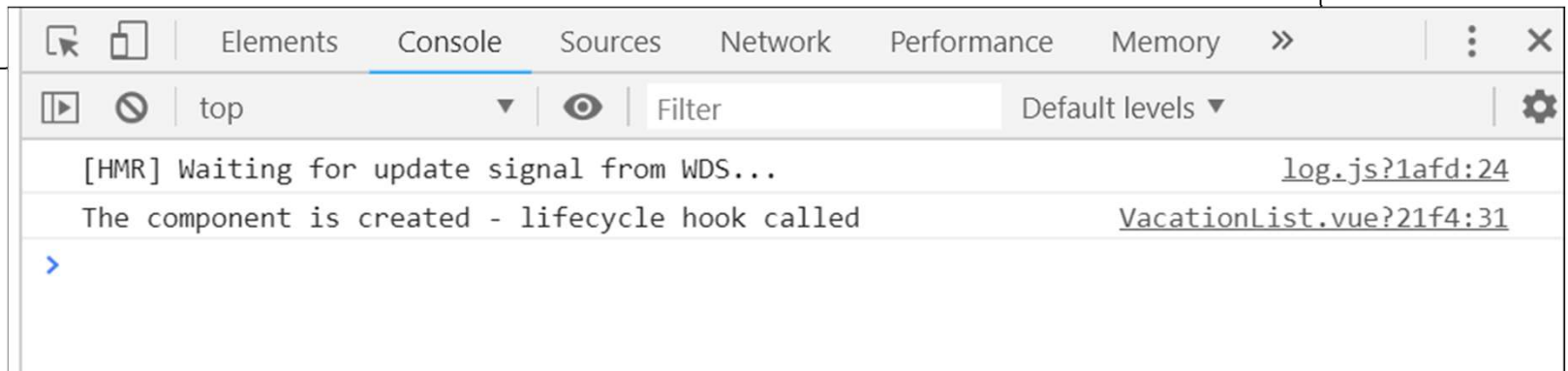
Most used:

- created
- updated
- destroyed

<https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks>

# Using the created hook

```
export default {
  name: "VacationList",
  data() {
    return {
      header: 'List of destinations',
    }
  },
  // Using the 'created' lifecycle hook.
  created(){
    console.log('The component is created - lifecycle hook called');
    // update the header
    this.header = 'The component is created';
  },
  ...
}
```

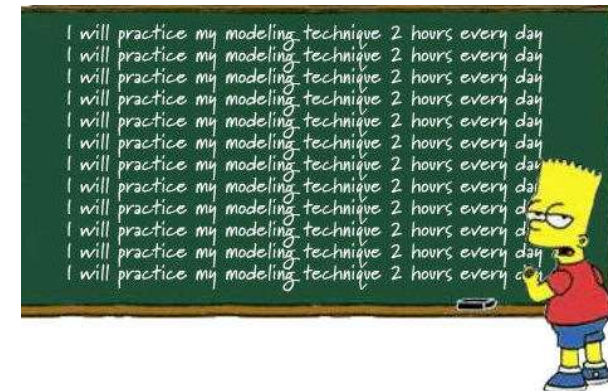


# Usage of lifecycle hooks

- Typical usage
  - `created` – initialisation of variables, call API's for fetching data etc.
  - `mounted` – if you want to access or modify the DOM.
  - `updated` – when the component receives new data from the outside (props)
  - `destroyed` – to destroy or garbage collect stuff that is not removed automatically
  - **Vue 3:** `unmounted()` (instead of `destroyed()`)

# Workshop

- Create a new component.
- Give it some data that you bind in the UI.
- Use a lifecycle hook `created` to log to the console that the component is created.
- Edit the data in the `created` lifecycle hook. Verify that it is shown correctly in the UI.
- Read the documentation on some other lifecycle hooks, for instance <https://www.digitalocean.com/community/tutorials/vuejs-component-lifecycle>
- Example: [.../150-lifecycle-hooks](#)





# Using mixins

Reuse functionality across components

Peter Kassenaar

INCLUSIEF  
GRATIS  
WEBVERSIE  
VAN HET  
BOEK

Vue.js

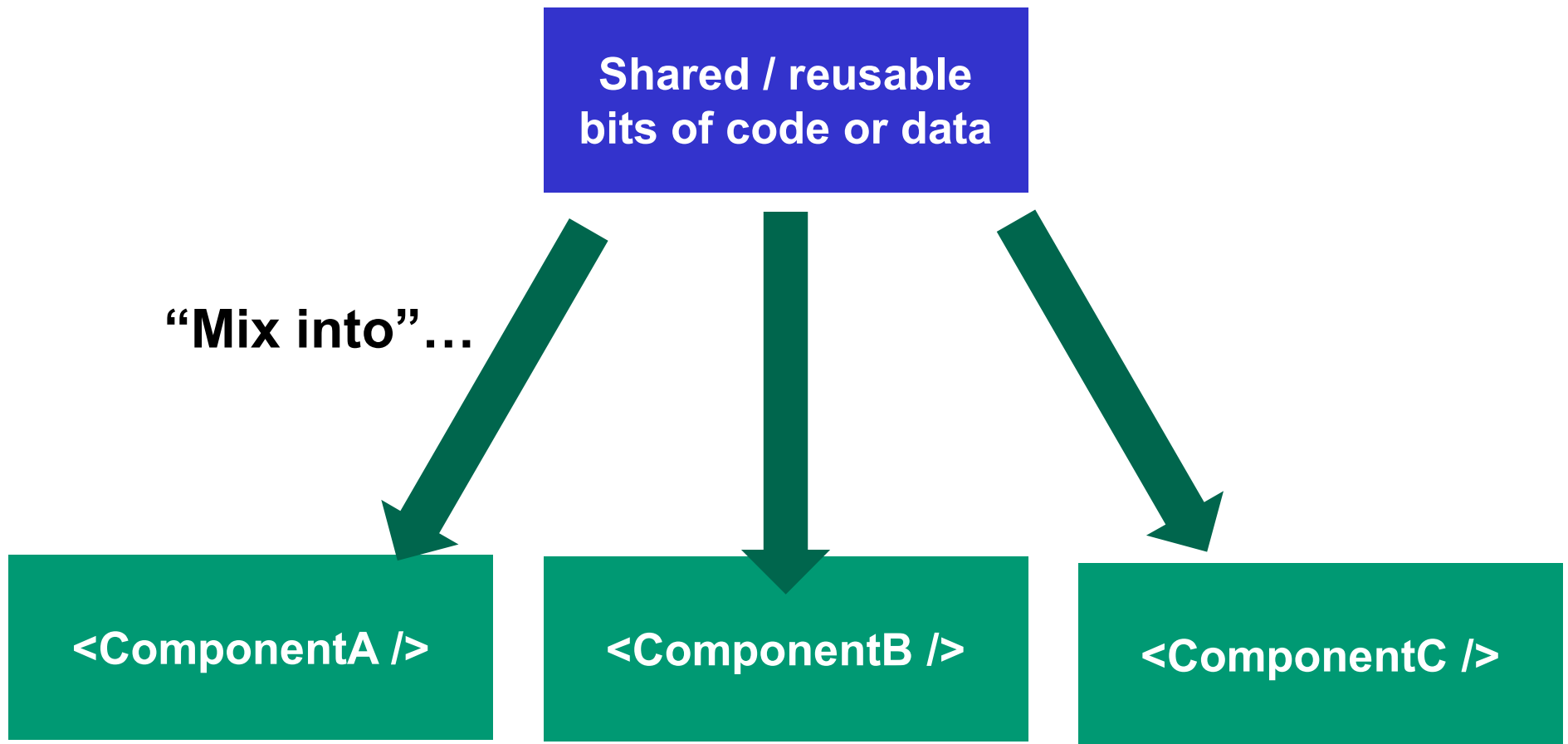
Web Development Library



VANDUUREN MEDIA

P. 113 e.v.

# Mixin architecture



*"Mixins are a flexible way to distribute reusable functionalities for Vue components. A mixin object can contain any component options. When a component uses a mixin, all options in the mixin will be "mixed" into the component's own options."*



# Mixins

- Mixins are a way to share functionality across components
- Useful if you find yourself duplicating code in multiple components
- Usually stored in a separate file

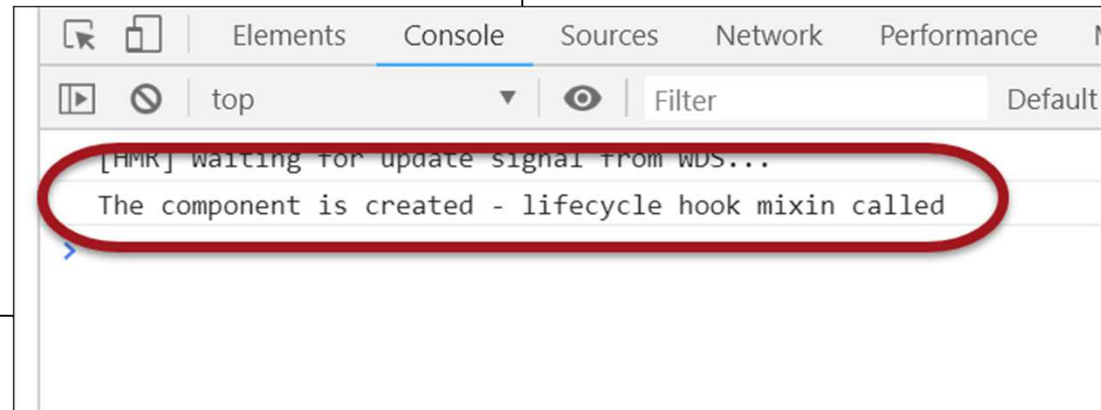
```
// mixins.js - export default mixins in this application.  
// They can be loaded into every component that needs them.  
export default {  
  // Using the 'created' lifecycle hook in a mixin  
  created(){  
    console.log('Component created - lifecycle hook mixin called');  
  },  
}
```

# Using a mixin

- Import the `mixin.js` file in the component
- Add it to the `mixins` property of the component.
- It is used as before
- NOT just lifecycle hooks! All kinds of functionality & data you want to share

```
<script>
  ...
  // import the mixin
  import createdHookMixin from '../mixins/mixins.js'

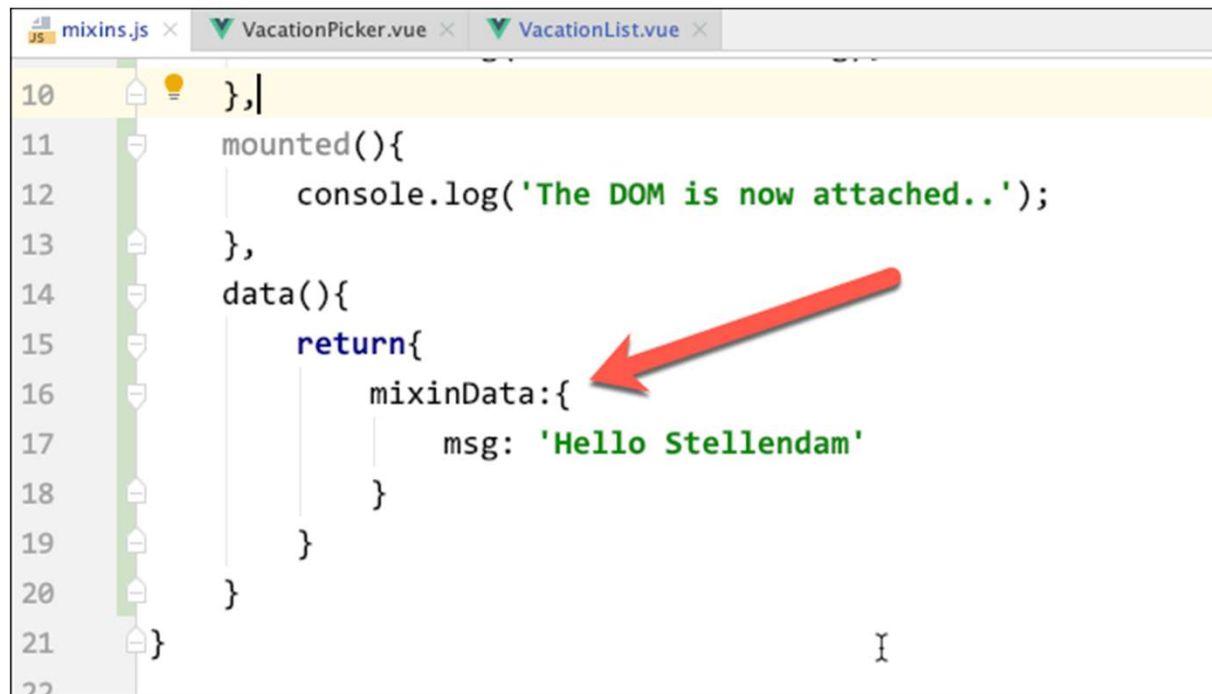
  export default {
    name: "VacationList",
    ...,
    mixins:[createdHookMixin]
  }
</script>
```



# Setting a 'namespace'

Tip: if you have to reuse data, it is an option to put in a nested object, so the mixin data can be distinguished from the component data:

You have now created some kind of custom 'namespace'



```
10  },|
11  mounted(){
12    console.log('The DOM is now attached..');
13  },
14  data(){
15    return{
16      mixinData:{
17        msg: 'Hello Stellendam'
18      }
19    }
20  }
21 }
```

A red arrow points to the `mixinData` object in the `data()` function, highlighting it as a custom namespace for the mixin data.

# Global mixins

- You can create a mixin for the complete `Vue` instance
- Use the `Vue.mixin()` option in `main.js`
- Candidates for global mixin data:
  - Global URL endpoints
  - Error messages
  - Translations
  - General data

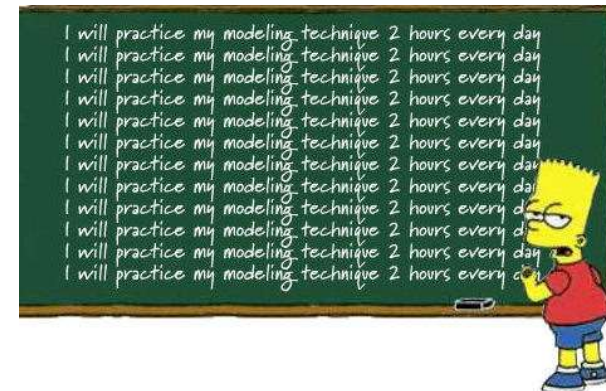
```
5
6 // Global mixins - available in ALL components
7 import mixins from "../mixins/mixins"
8 Vue.mixin(mixins);
9
10 import ...
12
```

# Rules

- Component methods **override** mixin methods
- Component computed properties **override** mixin computed properties
  - “last one wins”
- Component lifecycle hooks are **added** to mixin lifecycle hooks
- Component data is **added** to mixin data
  - UNLESS you have a naming conflict

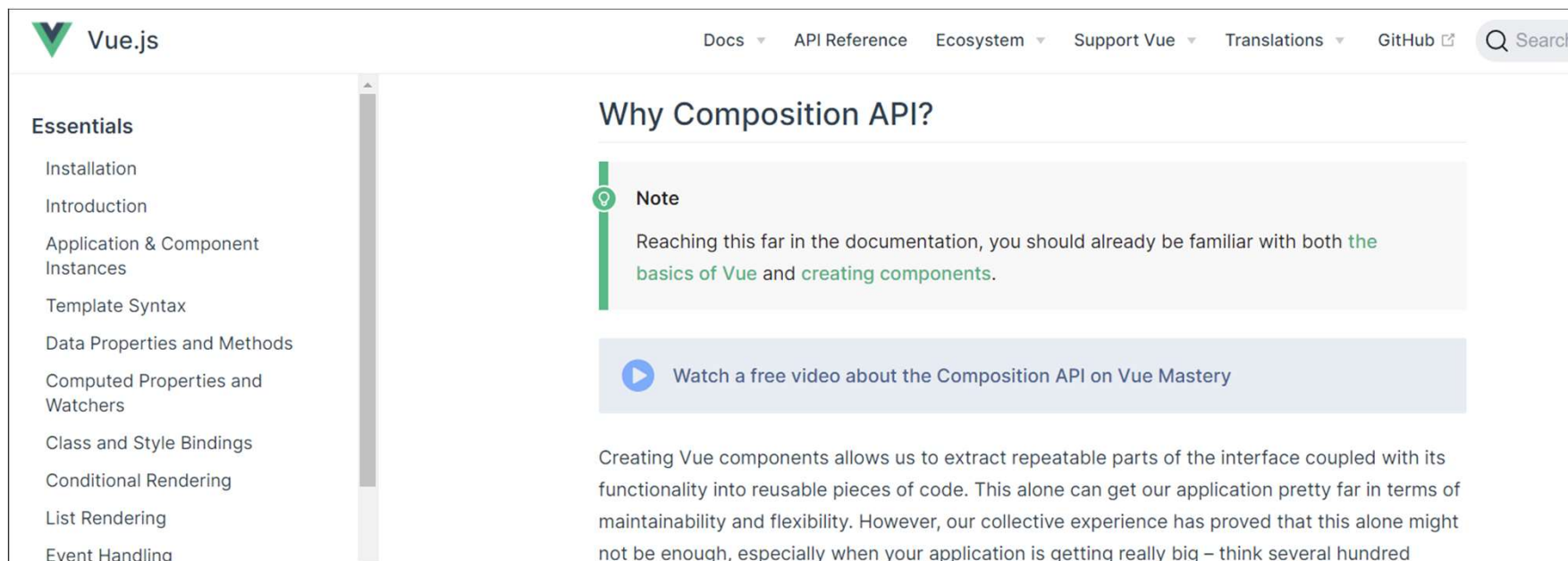
# Workshop - mixins

- Create a mixin for the `data.js` – file
- Use it in the component.
  - See if the component can still access the data
  - This way you can share data over multiple components
- Optional: create a global mixin and see if you can access it from all components
- General example: `../160-mixins`



# Vue 3 – mixins less important

- New concept – **Composition API**
- Setup() – for composing components
- <https://v3.vuejs.org/guide/composition-api-introduction.html#why-composition-api>





# Style Bindings

On using global styles and scoped styles



# Global styles and scoped styles

With default styles, CSS is globally available.

For instance, see `App.vue`:

```
<style>
  #app {
    font-family: 'Avenir', Helvetica, Arial, sans-serif;
    color: #2c3e50;
  }
</style>
```

This is also true for components!

# Using scoped styles

- To avoid naming collisions, it is best to add the `scoped` attribute to a style block inside a component
- Different components now can reuse the same classname without clashes.

```
<template>
  <div>
    <h2 class="heading">Component 1</h2>
    ...
  </div>
</template>

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

<style scoped>
  .heading {
    font-size: 36px;
    color: cornflowerblue;
  }
</style>
```

```
<h2 class="heading">Component 2</h2>
<style scoped>
  .heading {
    font-size: 36px;
    color: crimson;
  }
</style>
```

```
<h2 class="heading">Component 3</h2>
<style scoped>
  .heading {
    font-size: 48px;
    color: rebeccapurple;
  }
</style>
```

# Three components. Same class name, different styling.

## Component 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit. At illum molestiae quae tempore ut. Expedita nostrum omnis perspiciatis porro praesentium repellat similique voluptate voluptatum. Dolorum eaque ex praesentium quibusdam voluptates?

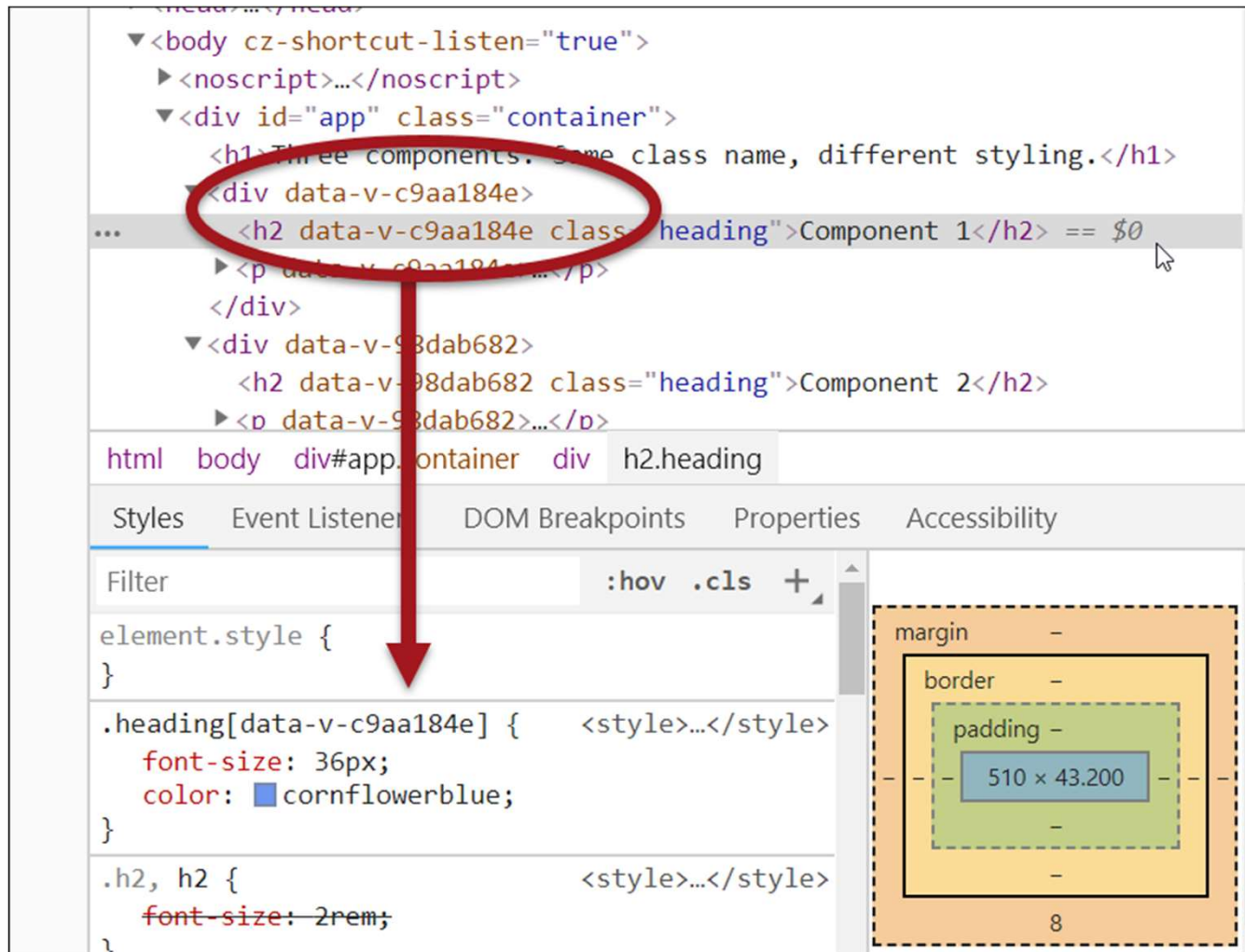
## Component 2

Lorem ipsum dolor sit amet, consectetur adipisicing elit. At illum molestiae quae tempore ut. Expedita nostrum omnis perspiciatis porro praesentium repellat similique voluptate voluptatum. Dolorum eaque ex praesentium quibusdam voluptates?

## Component 3

Lorem ipsum dolor sit amet, consectetur adipisicing elit. At illum molestiae quae tempore ut. Expedita nostrum omnis perspiciatis porro praesentium repellat similique voluptate voluptatum. Dolorum eaque ex praesentium quibusdam voluptates?

# Vue adds (semi random) hashes to elements



# General rules on styling

- Do not create global styles in components
- Only the top level component (`App.vue`) should have global styles
- You *can* use a generic CSS-framework like Bootstrap, Foundation, Vuetify, etc.

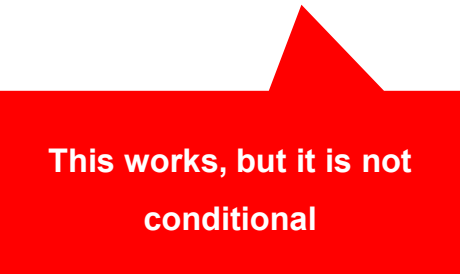
# Conditionally applying styles

- Bind to the style attribute like so:
  - `v-bind:style="{ ...some-style...}"` or just
  - `:style="{...some-style...}"`
  - For instance `:style="{ border: '2px solid black' }"`
  - These are actually just CSS styles and notation!
- If your CSS-style has a hyphen in them, a special notation is needed:
  - `:style="{['background-color']: 'lightBlue'}"`
  - or use camelCase notation:
  - `:style="{backgroundColor: 'lightBlue'}"`

# Making the style conditional

- For instance: we only want the style to be applied if the cost of a trip is less than 1000
- We can just bind to the HTML `:style` property
- For the value: use a computed property, or method.
- Let the computed property or method return a valid CSS style object

`:style="{backgroundColor: 'lightBlue'}"`



This works, but it is not conditional



This example: using a method

```
<li class="list-group-item"  
  :style="highlightBackground(index)"  
  v-for="(country, index) in data.countries" :key="country.id">  
    {{ country.id }} - {{country.name}}  
</li>
```

```
methods:{  
  highlightBackground(index){  
    return {  
      backgroundColor:  
        this.data.countries[index].cost < 1000 ?  
          'lightBlue' :  
          'transparent'  
    }  
  }  
}
```

# Conditionally applying styles

## List of destinations

1 - USA

2 - Netherlands

3 - Belgium


4 - Japan

5 - Brazil



6 - Australia

# Using v-model on a selection list

```
<h2>Destinations cheaper than:  
  <select class="form-control-lg" v-model="selectedCost">  
    <option value="1000">1000</option>  
    <option value="2000">2000</option>  
    <option value="3000">3000</option>  
    <option value="4000">4000</option>  
    <option value="5000">5000</option>  
    <option value="6000">6000</option>  
  </select>  
</h2>
```



```
data() {  
  return {  
    ...  
    selectedCost: 1000  
  }  
},  
methods: {  
  highlightBackground(index) {  
    return {  
      backgroundColor:  
        this.data.countries[index].cost < this.selectedCost ?  
          'lightBlue' :  
          'transparent'  
    }  
  }  
}
```



# Conditionally applying styles

## List of destinations

1 - USA
2 - Netherlands
3 - Belgium
4 - Japan
5 - Brazil
6 - Australia

Destinations cheaper than:

2000 ▾

1000

2000

3000

4000

5000

6000

# Conditionally applying classes

- Most of the times it is better to use CSS classes instead of inline styles
- Class binding is an object where the **keys** are the name of the CSS-class you want to toggle.
- You set the **value** to a boolean expression that should evaluate to `true` or `false`
  - If `true`, the class is applied
  - If `false`, the class is removed from the element
  - Of course this is all dynamic

# Same functionality – with class binding

Create a CSS class:

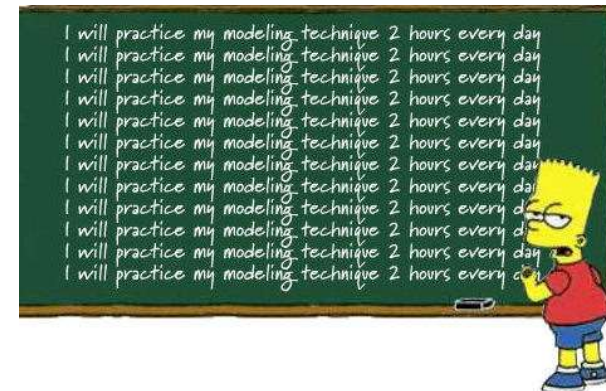
```
<style scoped>
  .lightblueBackground {
    background-color: lightblue;
  }
</style>
```

Apply the class conditionally in HTML:

```
:class="{ 'lightblueBackground': country.cost < selectedCost }"
```

# Workshop

- Create a component with a `<button>` and a `<div>`
- if the button is clicked, the class of the div is toggled
  - First – use conditional styles
  - Second – use conditional classes
- Add a `<div>`. If you hover the mouse over the div, toggle a class to highlight it
- Example: `140.../.../ConditionalClass.vue`



# Vue 3.2: Dynamic CSS Variables

- <https://www.jsnow.io/javascript/vue/vue-dynamic-css>





# Checkpoint

- You can use **v-model** in various situations
- You know about **lifecycle hooks**
- You can write and (re)use **mixins**
- You know the difference between **global styles** and **scoped styles**
- You know how to apply styles and classes **conditionally**