VUEJS

LOGISTIC

- Hours and Scheduling
- Lunch & breaks

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- Store
- Testing
- Typescript
- Composition
- SSR

WHAT IS VUEJS

OVERVIEW

- *What is Vue.js*
- Instance
- Tooling
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- Store
- Testing
- Typescript
- Composition
- SSR

INTRODUCTION

Vue.js was created in February 2014 by Evan You. It has been maintained by himself & his team ever since (not by any members of the GAFAM family).

Its 3.0 version was announced in Fall 2018 & released two years after in September 2020.

Version 3 is usable and officially released. However, some libs are still in progress: Vuetify for example.

INTRODUCTION

Vue.js is the most starred Javascript Framework project on *GitHub*. (And the 8th most starred project)

GitHub is a web-based hosting service for version control using git & commonly used to host open-source software projects.

https://github.com

INSTALLATION

Vue.js can be installed on your computer via *npm*. npm is a package manager for Node.js. All packages can be found on npmjs.org

You may also add the library to your project via a *CDN* link by adding it in the script tag.

INSPIRATION

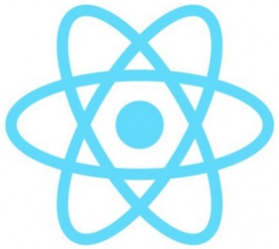
Vue.js is used to write sophisticated Single-Page Applications (SPA). It is also designed to be incrementally adoptable with other Javascript Libraries.

In fact, a lot of concepts used by Vue.js are directly inspired by other great libraries & frameworks such as *AngularJS*, *React*, *Svelte* or even *jQuery*.

ANGULARJS SHARED CONCEPTS

- Syntax (v-if vs ng-if)
- Two-way binding
- Directives
- Watchers

REACT SHARED CONCEPTS



- Virtual DOM
- Render Functions & JSX Support
- Focus on the View Layer
- Hooks (Composition API)
- Fragments, Portals (Teleport), Suspense

SVELTE SHARED CONCEPTS

- Single File Components syntax improvements

REFERENCE DOCUMENTATION & LEARNING MATERIALS

- VueJS official doc
- API VueJS
- Tutorial VueJS
- Vue School
- Vue Mastery

Lab 1

THE APPLICATION INSTANCE

OVERVIEW

- What is Vue.js
- *Instance*
- Tooling
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- Store
- Testing
- Typescript
- Composition
- SSR

OUR FIRST INSTANCE

A Vue application consists of a root instance optionally organized into a tree of nested, reusable components.

The application instance is declared using the Vue createApp function.

```
const vm = Vue.createApp({
    // options
})
```

Each instance is declared with an options object which can contain data, methods, elements to mount on or lifecycle callbacks.

MOUNT A DOM ELEMENT

To bind the Application Instance to a DOM element, we use the app.mount method and pass a selector as a param.

```
<div id="app">
    <h1>Hello World!</h1>
</div>
```

```
const app = Vue.createApp({})
app.mount('#app')
```

As a convention, we often use the variable `vm` (short for ViewModel) or `app` to refer to our instance.

DATA PROPERTIES

- Each property found in the data object is *proxified*
- Vue attaches setters & getters to each one of these properties
- All properties are accessible on the client side

```
<div id="app">
  <h1>{{ message }}</h1>
</div>
```

```
const app = Vue.createApp({
  data: function () {
    return {
      message: 'Hello World!'
    }
  }
}).mount('#app')
```

Lab 2

TOOLING

OVERVIEW

- What is Vue.js
- Instance
- *Tooling*
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- Store
- Testing
- Typescript
- Composition
- SSR

VUE-CLI

Vue CLI was the official webpack-based toolchain for Vue and is now in maintenance mode. This build tool lets you easily create, scale, and configure a Vue project.

Some features provided:

- Interactive project scaffolding (`vue create`).
- A dev server & module bundler (based on `Webpack`).
- Rich plugin ecosystem
- A full graphical user interface to manage your project (`vue ui`).

VITE

Build tool that provide a faster and leaner development experience for modern web projects.

It consists of two major parts:

- A dev server that provides rich feature enhancements over native ES modules (based on `esbuild`).
- A build command that bundles your code (based on `Rollup`).

VITE - CREATE A PROJECT

create-vue is the official Vue project scaffolding tool.

Create a project :

```
$ npm init vue
```

VITE - CREATE A PROJECT

Running the `npm init vue` command allows you to select the features you want to integrate into your application.

```
~/Apps $ npm init vue

Vue.js - The Progressive JavaScript Framework

✔ Project name: … vue-project
✔ Add TypeScript? … No / Yes
✔ Add JSX Support? … No / Yes
✔ Add Vue Router for Single Page Application development? … No / Yes
✔ Add Pinia for state management? … No / Yes
✔ Add Vitest for Unit Testing? … No / Yes
✔ Add Cypress for End-to-End testing? … No / Yes
✔ Add ESLint for code quality? … No / Yes
✔ Add Prettier for code formatting? … No / Yes

Scaffolding project in /home/jeremy/Apps/vue-project...

Done. Now run:

  cd vue-project
  npm install
  npm run lint
  npm run dev
```

EXTEND DEFAULT CONFIGURATION

Vite configuration can be modified from the `vite.config.js` file.

Below is the default content of this file.

```
// vite.config.js
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    }
  }
})
```

IMPORTANT COMMANDS TO KNOW

- Install dependencies:

```
npm install
```

- Compile and Hot-Reload for Development:

```
npm run dev
```

- Type-Check, Compile and Minify for Production:

```
npm run build
```

ENV VARIABLES

Vite supports env variables out of the box. They can be accessed on the special `import.meta.env` object. During production, these env variables are statically replaced.

Only variables prefixed with *VITE_* are accessible from your Vite-processed code.

```
# .env
VITE_APP_NAME=My TV shows
APP_NAME=My TV shows
```

```
// src/main.js

console.log(import.meta.VITE_APP_NAME) // log "My TV shows" from VITE_APP_NAME

console.log(import.meta.APP_NAME) // log undefined because the env variable APP_NAME is not
exposed in your Vite app
```

SINGLE FILE COMPONENTS

- File ending with `.vue` (e.g: `CardShow.vue`)
- Groups up all elements of a Vue component
  - template: `html` or `pug`
  - script: `javascript` or `typescript`
  - style: `css`, `sass`, `scss`, `stylus`
- Better readability
- Better developer experience

SINGLE FILE COMPONENTS - EXAMPLE


SingleFileComponent.vue

```vue
<template>
  <h1>{{ title }}</h1>
</template>

<script>
export default {
  name: 'App',
  data() {
    return {
      title: 'My app'
    }
  }
}
</script>

<style>
  h1 {
    font-size: 2em;
    margin-bottom: 20px;
  }
</style>
```

SINGLE FILE COMPONENTS - EXAMPLE


SingleFileComponent.advanced.vue

```pug
<template lang="pug">
  div
    h1 {{ title }}
</template>

<script lang="ts">
import { defineComponent } from 'vue'

export default defineComponent({
  name: 'App',
  data() {
    return {
      title: 'My App'
    }
  }
})
</script>

<style lang="stylus" scoped>
  h1
    font-size 2em
    margin-bottom 20px
</style>
```

VUE DEVTOOLS

A browser extension very useful for debugging Vue applications (Components, Events, Store, Routes).

Available on Chrome, Firefox or standalone application.

- Chrome extension
- Firefox extension

VETUR / VOLAR

VS Code has many extensions to enrich the development experience.

Extensions exist for `Vue` that will help the IDE handle the syntax of `*.vue` component files.
Some features provided by these extensions :

- Syntax & Semantic highlighting
- Snippet
- Linting / Error Checking
- Formatting ...

Vue 2 project extension : Vetur
Vue 3 project extension : Volar

`Volar` has been created specifically for Vue 3 and has better performance than `Vetur`.

Lab 3

SYNTAX

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- *Syntax*
- Components
- Plugins
- Routing
- Data fetching
- Store
- Testing
- Typescript
- Composition
- SSR

INTERPOLATIONS - TEXT

The most basic form of data binding is text interpolation using the "Mustache" syntax (double curly braces):

```
<span>Message: {{ msg }}</span>
```

The mustache tag will be replaced with the value of the msg property from the corresponding component instance. It will also be updated whenever the msg property changes.

INTERPOLATIONS - HTML

The double mustaches interprets the data as plain text, not HTML. In order to output real HTML, you will need to use the v-html directive:

```
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

Dynamically rendering arbitrary HTML on your website can be very dangerous because it can easily lead to XSS vulnerabilities.

ATTRIBUTES

Mustaches cannot be used inside HTML attributes. Instead, use a v-bind directive:

```
<div id="{{ dynamicId }}"></div>
```

👎 Wrong way to interpolate attribute

```
<div v-bind:id="dynamicId"></div>
```

👍 Right way to interpolate attribute

If the bound value is null or undefined then the attribute will not be included on the rendered element.

ATTRIBUTES - BOOLEAN

In the case of boolean attributes, where their mere existence implies true, v-bind works a little differently. For example:

```
<button v-bind:disabled="isButtonDisabled">Button</button>
```

The disabled attribute will be included if isButtonDisabled has a truthy value. It will also be included if the value is an empty string, maintaining consistency with `<button disabled="">`. For other falsy values the attribute will be omitted.

JAVASCRIPT EXPRESSIONS

So far we've only bound simple property keys in our templates. But Vue.js actually supports the full power of JavaScript expressions in all data bindings:

```
<span>{{ number + 1 }}</span>
<p>{{ ok ? 'YES' : 'NO' }}</p>
<p>{{ message.split('').reverse().join('') }}</p>
<div v-bind:id="'list-' + id"></div>
```

These expressions will be evaluated as JavaScript in the data scope of the current active instance.

DIRECTIVES

BASICS

`v-directive="“js expression”"`

```
<span v-if="displayMessage">Now you see me</span>
```

A directive allows Vue to add a side effect to the DOM

- Starts with a v-
- The value is a Javascript expression

ARGUMENTS

`v-directive:arguments`

```
<img v-bind:src="logoUrl">
```

Pass arguments to the directive

- Denoted by a `:` after the directive's name

MODIFIERS

`v-directive.modifier`

```
<button v-on:click.prevent.stop="clickButtonCallback()">
```

Binds the directive a certain way

- Denoted by a `.` after the directive's name

CONDITIONAL RENDERING

V-IF / V-ELSE / V-ELSE-IF / V-SHOW

Built-in directives used to add or remove DOM element

```
<div v-if="test">Test</div>
```

```
<template v-if="test">
    <h1>Title</h1>
    <h2>Subtitle</h2>
</template>
```

```
<div v-if="done">done</div>
<div v-else-if="loading">loading...</div>
<div v-else>Nothing to display</div>
```

```
<dialog v-show="toast"></dialog>
```

V-IF / V-ELSE / V-ELSE-IF / V-SHOW

Data object

```
{
    display: false,
    userStatus: 'loggedOut'
}
```

Template

```
<div class="alert" v-show="display">Alert displayed</div>
<template v-if="true">
    <h1>Title</h1>
    <h2>Subtitle</h2>
</template>
<button v-if="userStatus === 'loggedIn'">Logout</button>
<button v-else>Login</button>
```

V-IF / V-ELSE / V-ELSE-IF / V-SHOW

What will it render in the DOM ?

Rendered DOM:

```
<div class="alert" style="display: none;">Alert displayed</div>
<h1>Title</h1>
<h2>Subtitle</h2>
<button>Login</button>
```

LIST RENDERING

V-FOR

Built in directives used to render a list

```
v-for="item in array"
```

```
v-for="(item, index) in array"
```

```
v-for="(value, key, index) in object"
```

V-FOR

Example:

```
<ul>
    <li v-for="route in routes">{{ route.name }}</li>
</ul>
```

To help Vue render each element correctly and faster, use the `v-bind:key` attribute with a unique identifier:

```
<card v-for="element in menu" v-bind:key="element.id"></card>
```

V-FOR

Data object

```
{
  users: [
    { id: 1, name: 'Evan You', username: '@yyx990803' },
    { id: 2, name: 'Edd Yerburgh', username: '@eddyerburgh' }
  ],
  auth: { type: 'jwt', exp: 1501585988266 }
}
```

Template

```
<span v-for="n in 5">{{ n }}</span>
<template v-for="user in users">
    <span v-bind:key="user.id">{{ user.name }}</span>
    <span v-bind:key="user.id">{{ user.username }}</span>
</template>
<div v-for="(value, key, index) in auth">{{index}}# {{key}}: {{value}}</div>
```

V-FOR

What will it render in the DOM ?

Rendered DOM:

```
<span>1</span>
<span>2</span>
<span>3</span>
<span>4</span>
<span>5</span>

<span>Evan You</span>
<span>@yyx990803</span>
<span>Edd Yerburgh</span>
<span>@eddyerburgh</span>

<div>0# type: jwt</div>
<div>1# exp: 1501585988266</div>
```

CLASS AND STYLE BINDINGS

V-BIND

Built-in directive used to dynamically bind an HTML attribute to a JS expression

```
data() {
    return {
        dynamicId: 28
    }
}
```

```
<div v-bind:id="dynamicId"></div>
```

Render:

```
<div id="28"></div>
```

V-BIND:CLASS

Conditionally bind a CSS class to an element using `v-bind:class`

Using an object:

```
<div class="link" v-bind:class="{ 'active-link': isActive }"></div>
```

V-BIND:CLASS

What will it render?

Render:

- `isActive === true`

```
<div class="link active-link"></div>
```

- `isActive === false`

```
<div class="link"></div>
```

V-BIND:CLASS

Using an array:

```
<div class="alert" v-bind:class="[alertLevel, position]"></div>
```

```
data() {
    return {
        alertLevel: 'info',
        position: 'top'
    }
}
```

V-BIND:CLASS

What will it render?

Render:

```
<div class="alert info top"></div>
```

V-BIND:CLASS

Using a ternary operator to return `string`:

```
<div class="tag" v-bind:class="isFinished ? 'is-danger' : 'is-success'"></div>
```

```
computed: {
  isFinished() {
    return this.props.status === 'FINISH'
  }
}
```

V-BIND:CLASS

What will it render?

Render:

- `isFinished === true`

```
<div class="tag is-danger"></div>
```

- `isFinished === false`

```
<div class="tag is-success"></div>
```

V-BIND:STYLE

Update CSS properties dynamically using `v-bind:style`

```
<div class="alert" v-bind:style="{'background-color': alertLevel}"></div>
```

```
data () {
    return {
      alertLevel: '#00FF00'
    }
}
```

V-BIND:STYLE

What will it render?

```
<div class="alert" style="background-color: #00FF00"></div>
```

EVENT HANDLING

V-ON

Built-in directive used to listen to DOM or component events

```
<element v-on:event-name="codeToRunOnEvent"></element>
```

Example:

```
<button v-on:click="myAlert('login')">Login</button>
```

V-ON - EVENT MODIFIERS

- `.stop`: event.stopPropagation()
- `.prevent`: event.preventDefault()
- `.capture`: uses capture mode when adding the event listener
- `.self`: only from the first component (ignored for nested component)
- `.once`: the event is captured only once

```
<a href="" v-on:click.stop.prevent.capture.self.once="alert('link')">Link</a>
```

V-ON - KEY MODIFIER

- `.enter`: captures "Enter" key
- `.tab`: captures "Tab" key
- `.delete`: captures both “Delete” and “Backspace” keys
- `.esc`: captures "Escape" key
- `.space`: captures "Space" key
- `.up`: captures "Arrow up" key
- `.down`: captures "Arrow down" key
- `.left`: captures "Arrow left" key
- `.right`: captures "Arrow right" key

```
<input type="text" v-on:keyup.enter="login()">Login</input>
```

SHORTHANDS

V-BIND

`v-bind:attr` can be shortened into `:attr`

The following

```
<a v-bind:id="user.id" v-bind:href="user.loginUrl"
   v-bind:class="{ 'active-link': isActive }"
>
  Link
</a>
```

Is the same as

```
<a :id="user.id" :href="user.loginUrl"
   :class="{ 'active-link': isActive }"
>
  Link
</a>
```

V-ON

`v-on:event` can be shortened into `@event`

The following

```
<input v-on:keyup.enter="login()" v-on:click="focus()" />
```

Is the same as

```
<input @keyup.enter="login()" @click="focus()" />
```

CUSTOM EVENTS

CUSTOM EVENTS

It is possible to emit and listen custom events

In the child:

```
{
  ...
  emits: ['my-event'],
  methods: {
    validate() {
      this.$emit('my-event')
    }
  }
}
```

CUSTOM EVENTS

In the parent:

```
<child v-on:my-event="myFunction"></child>
```

or:

```
<child @my-event="myFunction"></child>
```

CUSTOM EVENTS - WITH PAYLOAD

In the child:

```
{
  ...
  emits: ['my-event']
  methods: {
    validate() {
      this.$emit('my-event', { data: 'some data' })
    }
  }
}
```

CUSTOM EVENTS - WITH PAYLOAD

In the parent:

```
<child @my-event="myFunction"></child>
```

```
{
  ...
  methods: {
    myFunction(payload) {
      // payload === { data: 'some data' }
    }
  }
}
```

NAMING CUSTOM EVENTS

Unlike components and props, event names will never be used as variable or property names in JavaScript, so there's no reason to use `camelCase` or `PascalCase`.

Additionally, `v-on` event listeners inside DOM templates will be automatically transformed to lowercase (due to HTML's case-insensitivity), so `v-on:myEvent` would become `v-on:myevent` – making `myEvent` impossible to listen to.

For these reasons, we recommend to **always use kebab-case for event names**.

FORM INPUT BINDINGS

NAIVE IMPLEMENTATION

Using `v-bind` and `v-on`

```
<input type="text" :value="text" @input="text = $event.target.value">
```

👍 Works well with input text or custom component

👎 Limited when using radio, select or multi-checkbox

V-MODEL

Built-in directive used to provide "two-way" data bindings

Text

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

Checkboxes

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

Multi-line text

```
<textarea v-model="multiLine"></textarea>
```

V-MODEL

Multi checkbox

```
data () {
    return {
        selection: ['red', 'blue']
    }
}
```

```
<input type="checkbox" v-model="selection" value="red">
<input type="checkbox" v-model="selection" value="blue">
<input type="checkbox" v-model="selection" value="green">
```

Radio

```
<input type="radio" v-model="gender" value="M">
<input type="radio" v-model="gender" value="F">
```

V-MODEL

Select

```
<select v-model="phoneType">
  <option disabled value="">Pick one</option>
  <option>android</option>
  <option>iOS</option>
  <option>windows phone</option>
</select>
```

V-MODEL - MODIFIERS

- `.lazy`: update model on `change` event instead of `input`
- `.number`: typecast to `Number`
- `.trim`: auto trim the value

```
<input type="text" v-model.trim.number.lazy="firstName">
```

V-MODEL - CUSTOM COMPONENT

Can also be used over your own components, these are default names

```
app.component('custom-input', {
  props: [ 'modelValue' ],
  emits: [ 'update:modelValue' ],
  template: `
    <input
      :value="modelValue"
      @input="$emit('update:modelValue', $event.target.value)"
    >
  `
})
```

Now v-model should work perfectly with this component:

```
<custom-input v-model:modelValue="searchText"></custom-input>
```

CUSTOM DIRECTIVE

APP.DIRECTIVE

```
<div v-my-directive></div>
```

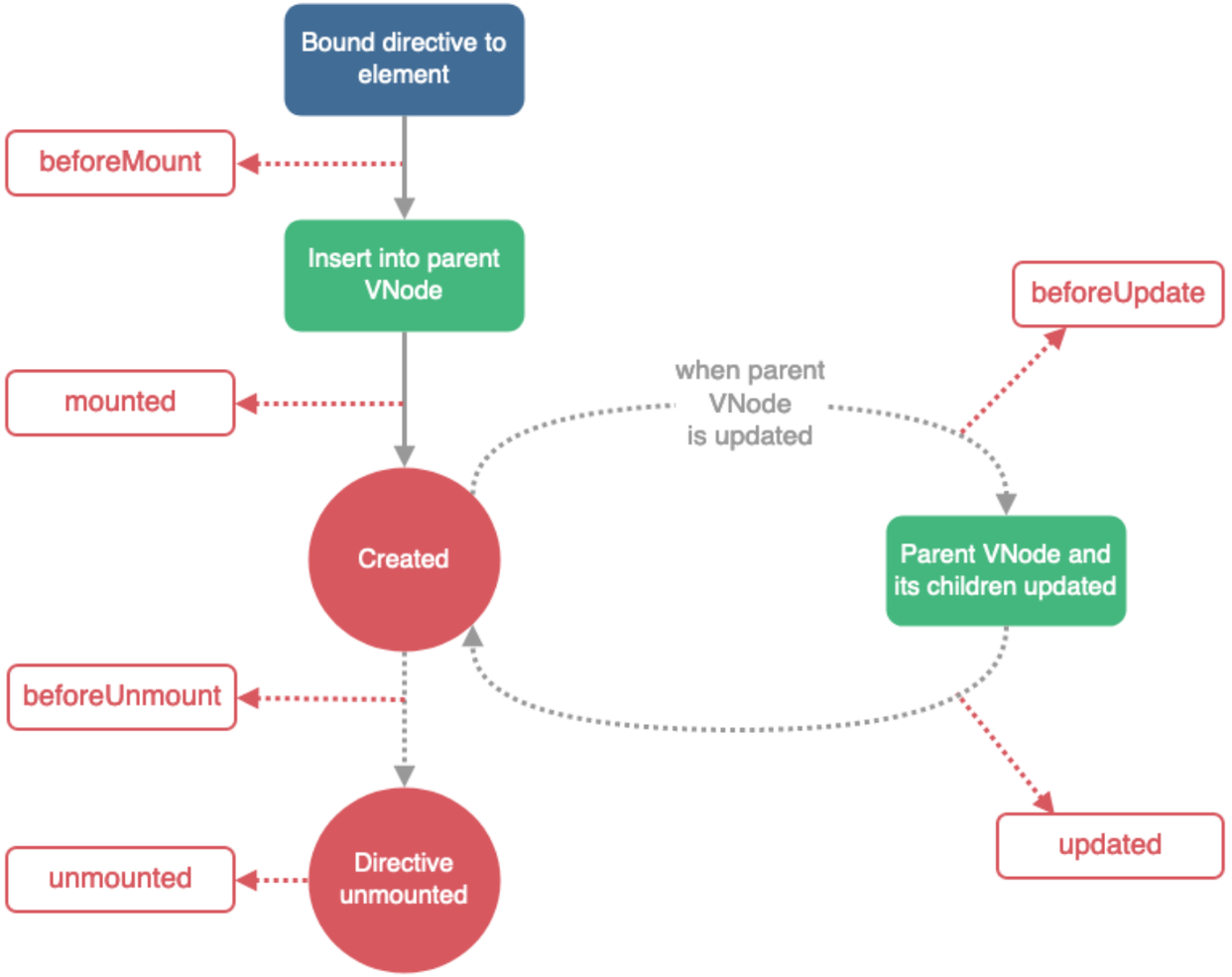
Register it globally

```
app.directive('my-directive', {
    ...
})
```

Register it locally

```
export default {
  name: 'myComponent',
  directives: {
    myDirective: {
      // ...
    }
  }
}
```

DIRECTIVE HOOKS



DIRECTIVE HOOKS ARGUMENTS

- `el`: The element the directive is bound to. This can be used to directly manipulate the DOM.
- `binding`: An object containing the following properties:
  - `instance`: The instance of the component where directive is used.
  - `value`: The value passed to the directive. For example in `v-my-directive="1 + 1"`, the value would be `2`.
  - `oldValue`: The previous value, only available in `beforeUpdate` and `updated`. It is available whether the value has changed or not.
  - `arg`: The argument passed to the directive, if any. For example in `v-my-directive:foo`, the arg would be `"foo"`.

DIRECTIVE HOOKS ARGUMENTS

- `binding`: An object containing the following properties:
  - `modifiers`: An object containing modifiers, if any. For example in `v-my-directive.foo.bar`, the modifiers object would be `{ foo: true, bar: true }`.
  - `dir`: an object, passed as a parameter when directive is registered.
  - `vnode`: A blueprint of the real DOM element received as el argument above.
  - `prevNode`: The previous virtual node, only available in the beforeUpdate and updated hooks.

Apart from `el`, you should treat these arguments as read-only and never modify them.

FUNCTION SHORTHAND

Most of the time you'll want the same behavior on `mounted` and `updated`, but won't care for other hooks.

Vue provides a shorthand notation for this:

```
app.directive('my-directive', (el, binding) => {
  /*...*/
})
```

Lab 4

COMPONENTS

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- *Components*
- Plugins
- Routing
- Data fetching
- Store
- Testing
- Typescript
- Composition
- SSR

OVERVIEW

- Components are custom HTML elements.
- They allow encapsulating reusable code.
- Each component should have its own behavior.

GLOBAL REGISTRATION

We can use `app.component(tagName, options)` to register a component at the application level. Components declared this way are *globally registered*.

```
import { createApp } from 'vue'

import BaseButton from './components/BaseButton.vue'

const app = createApp({})

app.component(BaseButton)

app.mount('#app')
```

We can then use it as a custom element in an instance's template.

```
<div id="app">
    <BaseButton />
</div>
```

LOCAL REGISTRATION

In a Vue application, generated by Vite or Vue CLI, we usually import and register components locally in other `.vue` components.

```
import ComponentA from './ComponentA.vue'

export default {
  components: {
    ComponentA
  }
}
```

```
<ComponentA />
```

`ComponentA` is a shorthand syntax for `ComponentA: ComponentA` where the key and the symbol have the same name.

SINGLE FILE COMPONENTS

Vue's big strength is the SFC pattern, as we saw in the Tooling chapter.

Single File Component syntax

See SFC Playground

```
<template>
  <h1>{{ message }}</h1>
</template>
```

```
export default {
  data() {
    return {message: 'Hello World!'}
  }
}
```

```
<style scoped>
h1 {
  font-size: 1.5rem;
}
</style>
```

DEFINE A COMPONENT

In `<script>` tags you can directly export an object to define a component:

```
export default {
  // component options. For example a prop:
  props: ['message']
}
```

However, the recommendation is rather to use the defineComponent helper method which provides better type inference (even for a component written in Javascript).

```
import { defineComponent } from 'vue'

export default defineComponent({
  // component options. For example a prop:
  props: ['message']
})
```

DATA

Like the Application Instance, we can put a data method in our component. This way, each component will have its *own instance* of its data and won't share it with other components.

Data values can be used in a component's template:

```
export default {
  data() {
    return {
      message: 'Hello Vue'
    }
  }
}
```

```
<template>
  <h1>{{ message }}</h1>
</template>
```

Which will render:

```
<h1>Hello Vue</h1>
```

See SFC Playground

PROPS

- Data can be passed down to child components using props.
- Props must be explicitly declared with the props option element.
- Props can be used inside templates.

```
// src/components/StrongMessage.vue
export default {
  props: ['message']
}
```

```
<template>
  <strong>{{ message }}</strong>
</template>
```

```
import StrongMessage from
'@/components/StrongMessage.vue'

export default {
  components: { StrongMessage }
}
```

```
<StrongMessage message="Message from my
parent" />
```

See SFC Playground

PROPS

To specify a prop type, you can list them as an object, where the properties' names and values contain the prop names and types:

```
import { defineComponent } from 'vue'

export default defineComponent({
    props: {
        message: String
    },
})
```

PROPS

And even further:

```
import { defineComponent } from 'vue'

export default defineComponent({
    props: {
        message: {
            type: String,
            required: false,
            default: 'Default message',
            validator: value => value.length > 0
        }
    },
})
```

PASSING PROPS WITH V-BIND

We can give expression to component props with `v-bind`.

```
import { defineComponent } from 'vue'
export default defineComponent({
  data() {
    return {
      description: 'Short show description',
      isFavorite: true,
    }
  }
})
```

```
<custom-component
  :is-favorite="isFavorite"
  :message="description"
></custom-component>
```

METHODS

- Methods are functions declared in our component instance.
- They can be declared using the methods option element.
- They can only be used in the component's scope.

```
// src/components/LikesButton.vue
export default {
  data() {
    return { likes: 0 }
  },
  methods: {
    addLike() { this.likes++ }
  }
}
```

```
<template>
  <button @click="addLike">{{ likes }}
</button>
</template>
```

```
import LikesButton from
'@/components/LikesButton.vue'
export default {
  components: {
    LikesButton
  }
}
```

```
<template>
  <LikesButton />
</template>
```

See SFC Playground

COMPUTED PROPERTIES

- Computed properties are used to avoid complex expressions in templates.
- We can use them with the `computed` option element.
- A computed property will only re-evaluate when some of its dependencies have changed.

```
export default {
  data () {
    return {
      firstName: 'Evan',
      lastName: 'You'
    }
  },
  computed: {
    fullName () {
      return `${this.firstName}
${this.lastName}`
    }
  }
}
```

```
<template>
  <div>{{ fullName }}</div>
</template>
```

Renders:

```
<div>Evan You</div>
```

See SFC Playground

WATCHERS

- With computed properties we can handle most cases.
- But for some specific cases we can use watchers.
- Useful for modifying data based on asynchronous or expensive operations.

WATCHERS

Example :

```
export default {
  data() {
      return { firstName: '', writingIssues: false }
  },
  watch: {
    async firstName(value) {
      // expensive asynchronous operation
      this.writingIssues = await checkWriting();
    }
  }
}
```

```
<div>
  <input type="text" v-model="firstName"/>
  <span>{{ writingIssues }}</span>
</div>
```

WATCHERS

Only use a `watcher` when it's relevant. A `computed` is often a better idea.

Below is a bad example as it should be a `computed`.

```
export default {
  data() {
    return { firstName: '', reversedFirstName: '' }
  },
  watch: {
    firstName(value) {
      this.reversedFirstName = value.split('').reverse().join('')
    }
  }
}
```

```
<div>
  <input type="text" v-model="firstName"/>
  <span>{{ reversedFirstName }}</span>
</div>
```

SLOTS

With slots, we can pass a template fragment to a child component, and let the child component render the fragment within its own template.

```
<!-- HTML of Parent component -->
<my-dialog>
  <div>This is the Dialog Content</div>
</my-dialog>
```

```
<!-- HTML of child component my-dialog-->
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="dialogContent">
    <slot></slot>
  </div>
</section>
```

SLOTS

It will produce:

```
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="dialogContent">
    <div>This is the Dialog Content</div>
  </div>
</section>
```

SLOTS - DEFAULT CONTENT

It's possible to specify a default content by simply putting it in the `slot` element:

```
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="dialogContent">
    <slot>Default content</slot>
  </div>
</section>
```

NAMED SLOTS

It's also possible to name slots :

```
<!-- HTML of Parent component -->
<my-dialog>
  <template v-slot:title>This is a dialog title</template>
  <template #body>This is the Dialog Content</template>
</my-dialog>
```

```
<!-- HTML of child component my-dialog-->
<section class="dialog">
  <h1><slot name="title"></slot></h1>
  <div class="dialogContent"><slot name="body"></slot></div>
</section>
```

NAMED SLOTS

It will produce:

```
<!-- HTML of child component my-dialog-->
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="dialogContent">
    <div>This is the Dialog Content</div>
  </div>
</section>
```

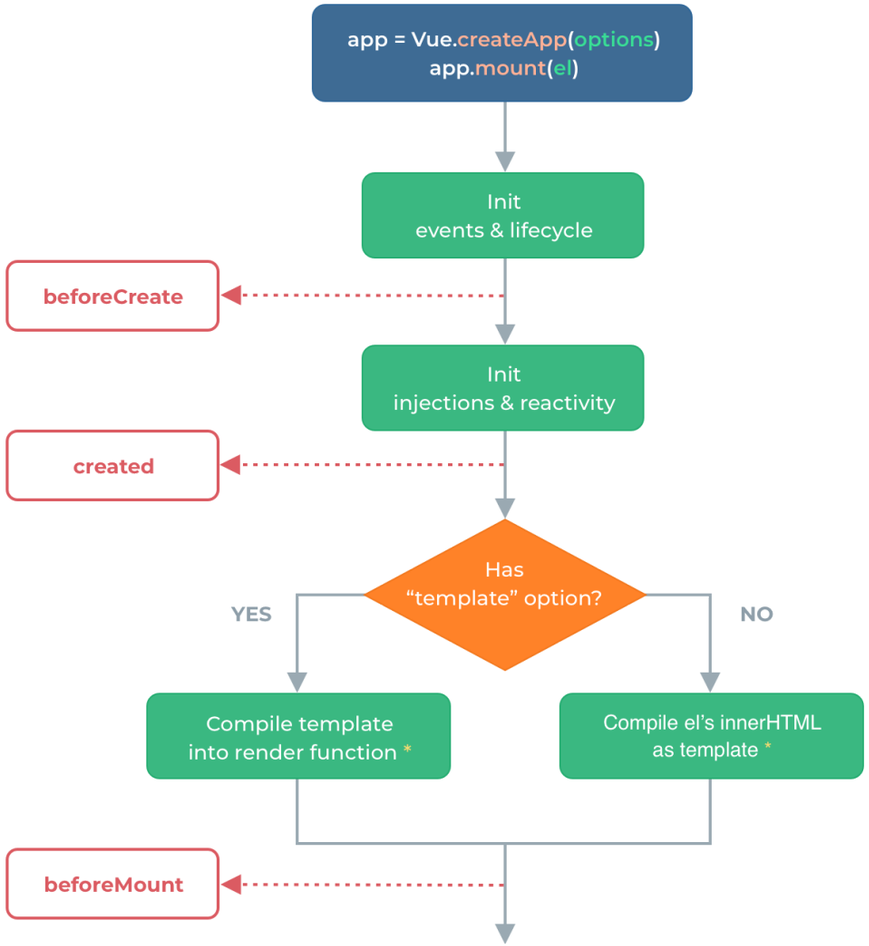
LIFECYCLE HOOKS

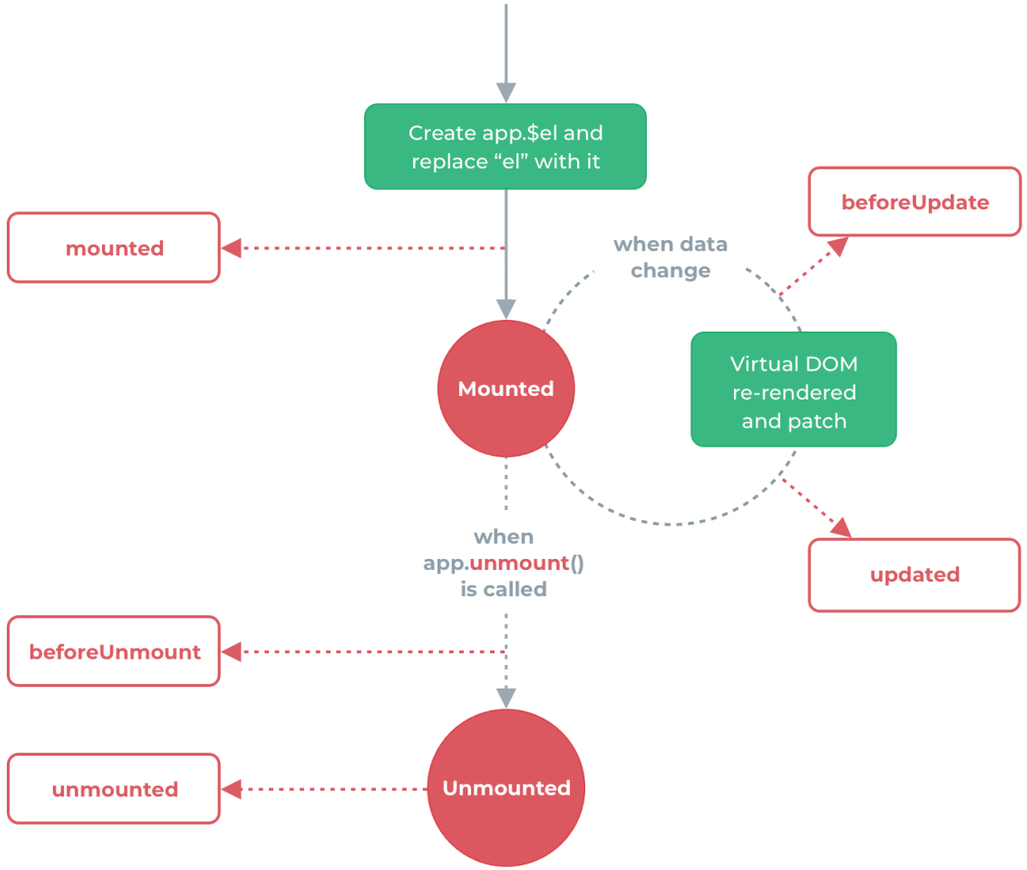
- Lifecycle hooks gives the opportunity to add our own code at specific stages
- Lifecycle hooks are non blocking for the render

```
export default {
    created() {
      // lifecycle hook appropriate for an API call
      console.log("I'm created");
    },
    mounted() {
      // interaction with the DOM possible
      console.log("I'm mounted");
    },
    beforeUnmount() {
      // can be used to clean up timers, listeners
      console.log("I'm beforeUnmount");
    },
}
```

See SFC Playground for more interactivity

LIFECYCLE DIAGRAM





* Template compilation is performed ahead-of-time if using a build step, e.g., with single-file components.

Lab 5

PLUGINS

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- *Plugins*
- Routing
- Data fetching
- Store
- Testing
- Typescript
- Composition
- SSR

WHY PLUGINS ?

- Register one or more global components or custom directives with app.component() and app.directive().
- Make a resource injectable throughout the app by calling app.provide().
- Add some global instance properties or methods by attaching them to app.config.globalProperties.
- A library that needs to perform some combination of the above (e.g. vue-router).

USING A PLUGIN

Just call `app.use()`

```
import App from './App.vue'
import myPlugin from './path/to/myPlugin'

const app = createApp(App)
app.use(myPlugin, { myPluginOption: true })
```

`Vue.use` automatically prevent multiple initialization

```
import myPlugin from './path/to/myPlugin'
// ...
app.use(myPlugin, { config: 1 })
app.use(myPlugin, { config: 2 })
app.use(myPlugin, { config: 3 })

// myPlugin is initialized with {config: 1}
```

USING A PLUGIN

awesome-vue, the list of Vue plugins

WRITING A PLUGIN

A plugin is defined as either an object that exposes an `install()` method, or simply a function that acts as the install function itself.

```
export default {
  install: (app, options) => {
    // Add anything you want via Vue application instance in params
  }
}
```

WRITING A PLUGIN

You can add VueJS components, directives

```
export default {
  install: (app, options) => {
    app.directive('my-directive', { /* ... */ })
    app.component('my-component', { /* ... */ })
  }
}
```

WRITING A PLUGIN

You can add an instance method. We usually prefix the method with a $

```
export default {
  install: (app, options) => {
    app.config.globalProperties.$myMethod = function (arg) { return arg * 2; }
  }
}
```

You can now use $myMethod in each component

```
export default {
  name: 'my-component',
  data() {
    return {
      toto: this.$myMethod(21)
    }
  }
}
```

Lab 6

ROUTING

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- Plugins
- *Routing*
- Data fetching
- Store
- Testing
- Typescript
- Composition
- SSR

ROUTING

Vue has an official router plugin called *vue-router*. We can add and use it by 2 ways:

- Import library via CDN
- Installing it via npm : `npm install vue-router`

BASIC ROUTING

Example :

```
import {createRouter, createWebHistory} from 'vue-router';

import Foo from '@/views/Foo.vue'
import Boo from '@/views/Bar.vue'

const router = createRouter({
  history: createWebHistory(),
  routes: [
    {path: '/foo', component: Foo},
    {path: '/bar', component: Bar}
  ]
})

app.use(router)
```

BASIC ROUTING

Example :

```
<div id="app">
  <h1>Hello App!</h1>

  <div>
    <router-link to="/foo">Go to Foo</router-link>
    <router-link to="/bar">Go to Bar</router-link>
  </div>

  <router-view></router-view>
</div>
```

BASIC ROUTING

In this example we can notice few things :

- We import `Foo` and `Bar` components
- We associate routes to each component.
- We create the router instance and pass the `routes` option.
- We inject the router in our root instance.
- We use *router-link* component, and we provide it a link as a prop.
- Then *router-view* renders the component that matches our current route.

DYNAMIC ROUTE MATCHING

We may want to share a component for many users or products. We need to have a core path that can fit with our different IDs that represent our users or products.

To do it we can use a dynamic segment in the path.

DYNAMIC ROUTE MATCHING

```
import { createRouter, createWebHistory } from 'vue-
router'
import UserDetails from '@/views/UserDetails.vue'

const router = createRouter({
  history: createWebHistory(),
  routes: [
    {
      path: '/user/:id',
      component: UserDetails
    }
  ]
})
```

```
<template>
  <div>User details</div>
</template>
```

By doing this, we will be able to navigate to *'user/1337'* or *'user/42'* and use the same User component.

DYNAMIC ROUTE MATCHING

The *:id* parameter of our route is exposed in every component. We can access it easily using *$route.params*.

```
import { createRouter, createWebHistory } from 'vue-
router'
import UserDetails from '@/views/UserDetails.vue'

const router = createRouter({
  history: createWebHistory(),
  routes: [
    {
      path: '/user/:id',
      component: UserDetails
    }
  ]
})
```

```
<template>
  <div>
    User {{ $route.params.id }}
  </div>
</template>
```

DYNAMIC ROUTE MATCHING

Many parameters can be passed to our routes. They will all be accessible in *$route.params*, defined by their name.

```
import UserDetails from '@/views/UserDetails.vue'

const routes = [{
  path: '/user/:firstname/:lastname',
  component: UserDetails
}];

const router = createRouter({history: createWebHistory(), routes})
```

```
<template>
  <div>
    My name is {{$route.params.firstname}} {{$route.params.lastname}}
  </div>
</template>
```

PARAMS CHANGES

Note that when a route renders the same component, the same component instance will be reused, so lifecycle hooks of the component will not be called again.

To trigger params changes, we can use the `beforeRouteUpdate` navigation guard:

```
import {defineComponent} from 'vue';

const User = defineComponent({
  beforeRouteUpdate(to, from) {
    // react to route changes...
    // return the route you want to redirect the user to
  }
});
```

BEST PRACTICES FOR PROPS

We saw that our props are accessible in *$route.params* but this syntax can be hard to understand when many props are passed. A better way is to the use *props* option.

```
import UserDetails from '@/views/UserDetails.vue'
const routes = [{
    path: '/user/:id',
    component: UserDetails,
    props: true
}]
```

```
<template>
  <div>User {{ id }}</div>
</template>

<script>
import { defineComponent } from 'vue'
export default defineComponent({
  props: ['id']
})
```

NAMED ROUTES

Each route can have a `name` in addition to a path. This can be specified in the route declaration.

```
const routes = [
  {
    path: '/user',
    name: 'user',
    component: User
  }
]
```

```
<router-link :to="{ name: 'user' }">
  User
</router-link>
```

NAMED ROUTES

Named routes can be very useful when we want to pass props directly from our HTML.

```
const routes = [
  {
    path: '/user/:userId',
    name: 'user',
    component: User
  }
]
```

```
<router-link :to="{ name: 'user', params: { userId: 42 }}">
  User
</router-link>
```

NAMED VIEWS

Named views can be used when we want to render many views at the same time. For that we just need to pass a parameter to router-view called *name*.

```
<router-view></router-view>
<router-view name="sidebar"></router-view>
<router-view name="navbar"></router-view>
```

```
const routes = [
  {
    path: '/',
    components: {
      default: Main,
      sidebar: Sidebar,
      navbar: Navbar,
    }
  }
]
```

A router-view without a name will be given *default* as its name.

PROGRAMMATIC NAVIGATION

We can use the `router` we constructed with `createRouter` to navigate:

```
this.$router.push({ name: 'user', params: { userId: 42 }})
```

same as

```
<router-link :to="{ name: 'user', params: { userId: 42 }}">
```

PROGRAMMATIC NAVIGATION

To replace our current history entry:

```
this.$router.replace({ name: 'home' })
```

same as

```
<router-link :to="{ name: 'home' }" replace>
```

Or to go forwards or go backwards in the history stack:

```
this.$router.go(-1)
```

REDIRECTION

Vue router allows us to redirect path, for example redirect from `/foo` to `/bar`

```
const routes = [
  {
    path: '/foo',
    redirect: '/bar'
  }
]
```

We can also use a function instead of passing a hard coded path :

```
const routes = [
  {
    path: '/foo',
    redirect: (to) => {
      // the function receives the target route as the argument
      // we return a redirect path/location here.
    }
  }
]
```

ALIAS

An alias can be used when we want to apply the same logic to some routes.

For example if we want to share the component Main to both paths `/` and `/home` we will use an alias.

Example :

```
const routes = [
  {
    path: '/',
    component: Home,
    alias: '/home'
  }
]
```

LAZY-LOADING ROUTES

When an application starts to grow, its bundle does too. A good solution to gain performance is to load on demand when the route is accessed:

```
import Homepage from './routes/Homepage.vue'

const routes = [
  {
    path: '/',
    component: Homepage
  },
  {
    path: '/other',
    component: () => import('./routes/LazyLoadedPage.vue')
  }
]
```

Thanks to `() => import('...')` our file LazyLoadedPage will only load when visited.

ROUTE MIDDLEWARES

Some middlewares can be applied to a route, for example to block access to users who are logged in or logged out.

```
const routes = [
  { path: '/', component: Home, meta: { loggedIn: true } },
  { path: '/login', component: LogIn, meta: { loggedOut: true } },
  {
    // Fake route, to confirm an email address from a link sent by email
    path: '/confirm',
    beforeEnter(to, from, next) {
      axios.post('/confirm', { key: to.query.key, email: to.query.email })
        .then(response => next('/'), err => next('404'));
    }
  }
];
const router = VueRouter.createRouter({routes});
```

ROUTE MIDDLEWARES

You can also add guard on each route

```
const router = VueRouter.createRouter({routes});
router.beforeEach((to, from, next) => {
  // Force redirect to login page if route requires it
  if (to.meta.loggedIn && !isLoggedIn()) {
    next({ name: 'login'});
  }

  // Force redirect out of login page if already logged in
  if (to.meta.loggedOut && isLoggedIn()) {
    next({ name: 'home'});
  }

  next();
});
```

CATCH ALL / 404 NOT FOUND ROUTE

We can catch all route and link it to a 404 route by adding a regexp inside parentheses right after the param.

In the example below, if the user goes to a route different than the homepage, then he will be redirected to the `NotFound` component.

```
const routes = [
  {path: '/', name: 'homepage', component: Homepage},
  // will match everything and put it under `$route.params.pathMatch`
  {path: '/:pathMatch(.*)*', name: 'NotFound', component: NotFound},
]
```

Lab 7

DATA FETCHING

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- Plugins
- Routing
- *Data fetching*
- Store
- Testing
- Typescript
- Composition
- SSR

DATA FETCHING

Most of the time we want to fetch data from a remote source or consume an API in our applications.

Initially, `vue-resource` was used to fulfill this need but since the creator of Vuejs (Evan You) announced that the Vue.js team is discontinuing `vue-resource`, Axios is recommended.

AXIOS

Axios is one of the most popular HTTP client libraries. It uses promises by default and runs on both the client and the server (so we can combine with async/await). Moreover, it is universal, supports cancellation and has TypeScript definitions.

Installation:

```
npm install axios --save
```

FETCHING DATA

Here is a basic example of how we can get some data from a URL :

```
import { defineComponent } from 'vue'
import axios from "axios";

export default defineComponent({
  data: () => {
    return {
      todos: []
    }
  },
  async created() {
    const response = await axios.get(
      "https://jsonplaceholder.typicode.com/todos"
    );

    this.todos = response.data;
  }
})
```

POST DATA

Example :

```
import axios from "axios";

export default {
  methods: {
    async onSubmit() {
      await axios.post("https://jsonplaceholder.typicode.com/todos",
        {
          title: "Become a ninja with Vue 3",
          completed: false
        }
      );
    }
  }
};
```

CREATE A BASE INSTANCE

Axios allows us to create a base instance to make it easier to configure. We could provide parameters such as base URL or headers.

```
// src/api.js

export const httpClient = axios.create({
  baseURL: 'http://www.example.com',
  headers: { Accept: 'application/json' }
})
```

```
import { httpClient } from '@/api'

export default {
  async created () {
    try {
      const response = await httpClient.get('data')
      // etc...
    } catch (error) {
      console.log(error)
    }
  }
}
```

Lab 8

STORE

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- *Store*
- Testing
- Typescript
- Composition
- SSR

STATE MANAGEMENT IN VUEJS

When your application grows, it is often necessary to be able to share data between several components.

Several options are possible to manage the state of your application: the Reactivity API (which we will discuss in the advanced part of this training) and the use of a dedicated library.

Vuex has long been the official library but since version 3 is the default version of Vue, Pinia has become the recommended library for state management.

PINIA

Pinia is a store library. It allows you to share a state across components/pages.

It is similar in philosophy to Vuex, except that it doesn't differentiate between mutations and actions. There are only actions and they can be both synchronous and asynchronous and mutate the state directly without the need to call a mutation, making the use of the library simpler.

The Pinia library integrates seamlessly with VueJs:

- Devtools support
- Hot module replacement
- Plugins to extend Pinia features and fit our needs
- TypeScript support and autocompletion for JS users
- Server Side Rendering support

INSTALL PINIA

Install the dependency

```
npm install pinia
```

Create a Pinia instance by invoking the createPinia method and pass it to the application instance:

```
// src/main.js

import { createPinia } from 'pinia'

import { createApp } from 'vue'
import App from '@/App.vue'

const app = createApp(App)

app.use(createPinia())
```

DEFINE A STORE

A store is an entity holding state and business logic. It has three concepts: the *state*, *getters* and *actions*. Basically they are the equivalent of *data*, *computed* and *methods* in components.

Use the defineStore method from the Pinia library:

```
// src/stores/index.js

import { defineStore } from 'pinia'

// the first argument is a unique id of the store across your application
export const useStore = defineStore('main', {
  state: () => {
    return {
    }
  },
  getters: {
  },
  actions: {
  },
})
```

DEFINE A STATE

The state is defined as a function that returns the initial state.

```
// src/stores/index.js

import { defineStore } from 'pinia'

// the first argument is a unique id of the store across your application
export const useStore = defineStore('main', {
  state: () => {
    return {
      counter: 1
    }
  },
})
```

USE A STATE

Use the mapState helper method provided by Pinia to retrieve the value of the state.

```
import { defineComponent } from 'vue'
import { mapState } from 'pinia'
import { useStore } from '@/stores'

export default defineComponent({
  computed: {
      ...mapState(useStore, ['counter']), // gives access to this.counter inside the component

      ...mapState(useStore, {
        myOwnName: 'counter', // same as above but registers it as this.myOwnName
        // you can also write a function that gets access to the store
        double: store => store.counter * 2,
      }),
  },
  mounted() {
    console.log(this.counter, this.myOwnName, this.double)
  }
})
```

USE A STATE

Use the mapWritableState helper method provided by Pinia if you want to update the state

```
import { defineComponent } from 'vue'
import { mapWritableState } from 'pinia'
import { useStore } from '@/stores'

export default defineComponent({
  computed: {

      // gives access to this.counter inside the component and allows updating this.counter
      ...mapWritableState(useStore, ['counter'])

    }),
  },
  mounted() {
    this.counter = 3
  }
})
```

DEFINE SOME GETTERS

Getters are the equivalent of computed values for the state of a Store.

```
// src/stores/index.js

import { defineStore } from 'pinia'

export const useStore = defineStore('main', {
  state: () => ({
    counter: 0,
  }),
  getters: {
    // compute a value from the state
    doubleCount: (state) => state.counter * 2,
    // access other getter
    doubleCountPlusOne() {
      return this.doubleCount + 1
    },
  },
})
```

RETRIEVE THE GETTERS

Use the mapState helper method provided by Pinia to retrieve the value of the getters.

```
import { defineComponent } from 'vue'
import { mapState } from 'pinia'
import { useStore } from '@/stores'

export default defineComponent({
  computed: {
    ...mapState(useStore, ['doubleCount',])
    ...mapState(useStore, {
      myOwnName: 'doubleCount',
      double: store => store.doubleCount * 2,
    }),
  },
  mounted() {
    console.log(this.doubleCount, this.myOwnName, this.double)
  }
})
```

DEFINE SOME ACTIONS

Actions are the equivalent of methods in components. They can be synchronous and asynchronous. Actions get access to the whole store instance through `this`.

```
import { defineStore } from 'pinia'
import axios from 'axios'

export const useStore = defineStore('main', {
  state: () => ({
    counter: 0,
  }),
  actions: {
    increment() {
      this.counter++
    },
    async randomizeCounter() {
      const { data } = await axios.get('http://www.randomnumberapi.com/api/v1.0/random')
      if (data.count && data.count.length > 0) {
        this.counter = data.count[0]
      }
    },
  },
})
```

USE THE ACTIONS

Use the mapActions helper provided by Pinia.

```
import { defineComponent } from 'vue'
import { mapActions } from 'pinia'
import { useStore } from '@/stores'

export default {
  methods: {
    ...mapActions(useStore, ['increment', 'randomizeCounter']),

    ...mapActions(useStore, {
      myOwnName: 'increment'
    }),
  },
  async mounted() {
    this.increment()
    await this.randomizeCounter()
  }
}
```

ACCESS THE WHOLE STORE

You can get access to the whole store with mapStores() helper.

The variable made available is the concatenation of the store name (`main` in this case) and the suffix `Store` => `this.mainStore` in the example below

```
import { mapStores } from 'pinia'
import { useStore } from '@/stores'

export default {
  computed: {
    ...mapStores(useStore)
  },
  mounted() {
    console.log(this.mainStore.counter)
    this.mainStore.increment()
  },
}
```

Lab 9

TESTING

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- Store
- *Testing*
- Typescript
- Composition
- SSR

WHY TEST

Writing tests on an application allows you to:

- prevent regression
- increase our confidence in what we deliver in production
- encourages developers to better organize code into easy-to-test modules

TESTING TYPES

- *unit*: verifies that inputs to a given function, class, or composable produce the expected output or side effects
- *component*: verifies that your component mounts, displays, can interact with, and behaves as expected.
- *e2e*: verifies functionality of the application as a whole, possibly involving multiple pages and interaction with an API

UNIT TESTS

RECOMMENDATIONS

Jest if your project was scaffolded with Vue CLI.

Vitest if it's a Vite project

EXAMPLE

```
export function toggleFavorite(show) {
  show?.user?.favorited = !show?.user?.favorited
}
```

```
import { toggleFavorite } from './toggle-favorite.js'

describe('toggleFavorite', () => {
  test('toggles favorite status', () => {
    const show = { user: { favorited: true } }

    toggleFavorite(show)
    expect(show.user.favorited).toBe(false)

    toggleFavorite(show)
    expect(show.user.favorited).toBe(true)
  })

  test('does not throw error if show is null', () => {
    expect(() => toggleFavorite(null)).not.toThrow()
    expect(() => toggleFavorite(undefined)).not.toThrow()
  })
})
```

COMPONENT TESTING

RECOMMENDATIONS

Like unit tests, use Jest (for Vue CLI) or Vitest (for Vite)

Component testing often involves mounting the component being tested in isolation. You will therefore need a mounting library. There are two very popular mounting libraries:

VUE TESTING LIBRARY

Simple and complete testing utilities that encourage good testing practices

VUE TEST UTILS (LOW LEVEL, USED BY VUE-TESTING-LIBRARY)

As this library exposes some low level API, we can write fragile tests by testing implementation details. So be careful ! When testing, remember to test what a component does, not how it does it.

EXAMPLE

We will show how to implement a test with both libraries by using a simple example.

```
<div class="container">
  <p>{{ title }} : {{ counter }}</p>
  <button id="decrement" @click="decrement" class="button">–</button>
  <button id="increment" @click="increment" class="button">+</button>
</div>
```

```
export default {
  props: ['title'],
  data: () => ({
      counter: 0
  }),
  methods: {
    increment() {
      this.counter++
    },
    decrement() {
      this.counter--
    }
  }
}
```

WITH VUE-TESTING-LIBRARY

Check the docs here

```
import { render, fireEvent } from '@testing-library/vue'
import Counter from '@/components/Counter.vue'

describe('Counter', () => {
  it('should increment the counter', async () => {
    // The render method returns a collection of utilities to query your component.
    const { getByText } = render(Counter, {
      props: { title: 'My counter', }
    })

    // getByText returns the first matching node for the provided text, and
    // throws an error if no elements match or if more than one match is found.
    getByText('My counter : 0')

    const incrementBtn = getByText('+')
    await fireEvent.click(incrementBtn)

    getByText('My counter : 1')
  })
})
```

WITH VUE-TEST-UTILS

Check the Documentation

```
import { mount } from '@vue/test-utils'
import Counter from '@/components/Counter.vue'

describe('Counter', () => {
  it('should increment the counter', () => {
    // Create an instance of our component
    const wrapper = mount(Counter, {
      props: { title: 'My counter', }
    })

    expect(wrapper.text()).toContain('My counter : 0')

    wrapper.find('#increment').trigger('click')
    wrapper.vm.increment() // => bad way to test your component

    expect(wrapper.text()).toContain('My counter : 2')
    expect(wrapper.vm.counter).toBe(2) // => bad way to test your component
  })
})
```

E2E

RECOMMENDATIONS

There are two very popular libraries:

CYPRESS

All-in-one testing framework, assertion library, with mocking and stubbing features.

PLAYWRIGHT

Like Cypress, it is also a valuable E2E testing solution with a wider range of browser support (mainly WebKit).

EXAMPLE WITH CYPRESS

Check the Documentation

```
describe('ShowList', () => {
  it('renders the list of shows', () => {
    cy.visit('/')
    cy.get('h1').contains('My TV shows')
    cy.get('[data-cy=card-show]').should('have.length', 9)
  })
})
```

Cypress runs as fast as your browser can render content. You can watch tests run in real time as you develop your applications.

Lab 10

TYPESCRIPT

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- Store
- Testing
- *Typescript*
- Composition
- SSR

INTRODUCTION

- Language created by Anders Hejlsberg in 2012
- Open source project maintained by Microsoft
- Inspired by JavaScript, Java and C#

FUNCTIONALITIES

TypeScript is a language for application-scale JavaScript development. It's a typed superset of JavaScript that compiles to plain JavaScript. It allows writing components as classes using properties, decorators and using data types.

- ES2015+
- Typings
- Generics
- Classes / Interfaces / Inheritance
- Modular development
- Mixins
- Decorators

INSTALLATION

When you run the `npm init vue`, select Typescript to configure it properly.

You can also pass the `--typescript` flag to the command:

```
npm init vue -- --typescript my_project
```

VUE3 AND TYPESCRIPT

One of the goal of the version 3 of Vue is to be more type(script) friendly. Add simply `lang="ts"` and your component becomes TS compatible.

```
<script lang="ts">
import { defineComponent } from 'vue'

const Component = defineComponent({
  data() {
      return {
          title: 'Hello world' // Type will be string
          data: ['Paul', 0] // Type will be (string|number)[]
      }
  },
  mounted() {
      this.title = 0 // This will throw a type error
  }
})
</ script>
```

See Typescript Support in the doc for more detail.

CAST DATA TYPES

If you have a complex type or interface, you can cast it using type assertion:

```
interface User {
  name: string
  age: number
  favoriteFoods: string[]
}

const Component = defineComponent({
  data() {
    return {
      user: {
        name: 'Paul Arthur',
        age: 20,
        favoriteFoods: [] // If not cast, type inference will type any[]
      } as User
    }
  }
})
```

ANNOTATING PROPS

Vue does a runtime validation on props with a type defined.

```
import { defineComponent, PropType } from 'vue'

interface User [
  firstName: string
]

const Component = defineComponent({
  props: {
    id: [Number, String],
    user: {
      type: Object as PropType<User>,
      required: true
    },
    metadata: {
      type: null // metadata is typed as any,
      required: false // Can be null
    }
  }
})
```

ANNOTATING EMITS

We can annotate a payload for the emitted event. Also, all non-declared emitted events will throw a type error when called:

```
const Component = defineComponent({
  emits: {
    addBook(payload: { bookName: string }) {
      // perform runtime validation
      return payload.bookName.length > 0
    }
  },
  methods: {
    onSubmit() {
      this.$emit('addBook', {
        bookName: 123 // Type error!
      })

      this.$emit('non-declared-event') // Type error!
    }
  }
})
```

ANNOTATING RETURN TYPES

By default, Typescript can infer the types of your component's methods. But it may have difficulty inferring the `computed` types.

For this reason, we recommend to annotate the return types of the methods, computed properties and watchers.

```
import { defineComponent } from 'vue'

const Component = defineComponent({
  data() {
    return {
      message: 'Hello!'
    }
  },
  computed: {
    // an annotation is needed
    greeting(): string {
      return this.message + '!'
    },
  }
})
```

Lab 11

COMPOSITION API

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- Store
- Testing
- Typescript
- *Composition*
- SSR

COMPOSITION API

Until now, we used what is called the Options API. Coming with Vue 3, the Composition API and it's `setup` method is a new way to write our components.

```
export default {
  props: {
    user: {type: String}
  },
  setup(props) {
    console.log(props) // { user: '' }

    return {} // anything returned here will be available for the rest of the component
  },
  ...
}
```

COMPOSITION API

The `setup` method is executed before the component is created, once the `props` are resolved but before getting access to `this`.

It is the only place in a Vue component where we can use the reactivity API.

Except for props, you won't be able to access any properties declared in the component – local state, computed properties or methods.

Everything that we return from `setup` will be exposed to the rest of our component (computed properties, methods, lifecycle hooks and so on) as well as to the component's template.

JAVASCRIPT DEFAULT REACTIVITY

JavaScript is executed line by line and changes aren't necessarly reflected in previous declarations.

```
let val1 = 2
let val2 = 3
let sum = val1 + val2

console.log(sum)
// 5

val1 = 3

console.log(sum)
// sum is still 5...
```

With Vue3 and the Composition API comes the Reactivity API.

VUE REACTIVITY API

Both the Options API and the Composition API use the Reactivity API under the hood.

Both can be used, even in the same component. The object returned from `setup()` can be used inside Options API, but not the other way around.

COMPOSITION AND OPTIONS API

For example:

```
export default {
  props: ['firstName', 'lastName'],
  setup(props) { // composition
    const fullName = computed(() => {
      return `${props.firstName} ${props.lastName}`
    })
    return {fullName}
  },
  // data option
  data() {
    return {
      cohabitation: 'isPossible'
    }
  }
}
```

COMPOSITION API: DATA

To create a reactive state from a primitive value, we can use the `ref` method:

```
import {ref} from 'vue'

export default {
  setup() {
    const counter = ref(0)

    console.log(counter) // { value: 0 }
    console.log(counter.value) // 0

    counter.value++ // if it is a component props / data, the component will re-render
    console.log(counter.value) // 1
  },
  /*...*/
}
```

`ref` creates a wrapper around our object, which means we have to use `.value` to access or modify its value. `ref` values are auto-unwrapped in template which means we don't have to do `count.value++`

COMPOSITION API: DATA

To create a reactive state - like we did with the `data` object - from a JavaScript object, we can use the `reactive` method:

```
import {reactive} from 'vue'

export default {
  setup() {
    const state = reactive({
      display: false,
      userStatus: 'loggedOut'
    })

    state.display = true;
    console.log(state) // { display: true, userStatus: 'loggedOut' }

    state.userStatus = 'loggedIn';
    console.log(state) // { display: true, userStatus: 'loggedIn' }
  }
}
```

Usually we will prefer `reactive` over `ref` as it's easier to use.

COMPOSITION API: DATA

For `reactive` values, we can combine the spread operator `...` and the `toRefs` method:

```
<span v-if="display">{{ userStatus }}</span>
```

```
import {reactive, toRefs} from 'vue'

export default {
  setup() {
    const data = reactive({
      display: false,
      userStatus: 'loggedOut'
    })

    return {...toRefs(data)}
  }
}
```

Using the spread operator without `toRefs` breaks the reactivity as it creates a new **classic** JS object.

COMPOSITION API: METHODS

Just replace it by a simple vanilla function. Don't forget that `this` doesn't exist yet:

```
<span v-if="display">{{ userStatus }}</span>
<button @click="toggleUserDisplay">Toggle display</button>
```

```
import {reactive, toRefs} from 'vue'

export default {
  setup() {
    const data = reactive({display: false, userStatus: 'loggedOut'})

    const toggleUserDisplay = () => {
      data.display = !data.display
    }

    return {...toRefs(data), toggleUserDisplay}
  }
}
```

COMPOSITION API: COMPUTED

Use the `computed` method:

```
<span>{{ fullName }}</span>
```

```
import {computed} from 'vue'

export default {
  props: ['firstName', 'lastName'],
  setup(props) {
    const fullName = computed(() => {
      return `${props.firstName} ${props.lastName}`
    })

    return {fullName}
  }
}
```

Note that we are using the `props` parameter of the setup function. `props` are already exposed in the template and don't need to be returned.

COMPOSITION API: WATCHEFFECT

The `watchEffect` method will execute a callback function automatically when one of the reactive dependency is updated:

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

```
import { watchEffect, reactive } from 'vue';
import {sendEvent} from './my-tracking-service.js'

export default {
  setup() {
    const data = reactive({
      userConsent: false,
    })
    watchEffect(() => {
      sendEvent('user-consent-changed', { enabled: data.userConsent });
    })

    return {data}
  }
}
```

COMPOSITION API: EVENTS

Use the second parameter of the `setup` method:

```
<button @click="emitEvent">Emit an event</button>
```

```
export default {
  emits: ['my-event'],
  setup(props, {emit}) {
    const emitEvent = () => {
      emit('my-event', {data: 'some data'})
    }

    return {emitEvent}
  }
}
```

The second parameter is called `context` and contains other non-reactive objects which means we can destructure it safely.

COMPOSITION API: COMPONENT METADATA

Some component properties can't be moved to the `setup` method:

```
export default {
  name: 'ComponentName',
  components: { /* ... */},
  props: [/* ... */],
  emits: [/* ... */],
  setup() {
    // ...
  }
}
```

This is our component metadata and should remain as is.

COMPOSITION API: LIFECYCLE HOOKS

Lifecycle hooks in the Composition API have the same name as their counterpart in the Options API but are prefixed with `on`: i.e. `mounted` would look like `onMounted`.

These functions accept a callback that will be executed when the hook is called by the component.

COMPOSITION API: LIFECYCLE HOOKS

Example:

```
export default {
  props: ['user'],
  setup(props) {
    const repositories = ref([])
    const getUserRepositories = async () => {
      repositories.value = await fetchUserRepositories(props.user)
    }

    onMounted(getUserRepositories) // on `mounted` call `getUserRepositories`

    return {
      repositories,
      getUserRepositories
    }
  }
}
```

COMPOSITION API: COMPOSABLES

As its name suggests, an advantage of using the Composition API is that we can break down our components features into shared methods and get a better Separation of Concerns.

```
// src/composables/useUserRepositories.js
import {fetchUserRepositories} from '@/api/repositories'
import {ref, onMounted, watch} from 'vue'

export default function useUserRepositories(user) {
  const repositories = ref([])
  const getUserRepositories = async () => {
    repositories.value = await fetchUserRepositories(user.value)
  }

  onMounted(getUserRepositories)
  watch(user, getUserRepositories)

  return {
    repositories,
    getUserRepositories
  }
}
```

COMPOSITION API: COMPOSABLES

After outsourcing a feature to a separate file, we can start using it in our component:

```
// src/components/UserRepositories.vue
import {toRefs} from 'vue'
import useUserRepositories from '@/composables/useUserRepositories'

export default {
  props: {
    user: {type: String}
  },
  setup(props) {
    const {user} = toRefs(props)
    const {repositories, getUserRepositories} = useUserRepositories(user)

    return {
      repositories,
      getUserRepositories,
    }
  }
}
```

COMPOSITION API: SCRIPT SETUP

`<script setup>` is a compile-time syntactic sugar close to Svelte for using Composition API inside Single File Components (SFCs). It is the recommended syntax if you are using both SFCs and Composition API.

It provides a number of advantages over the normal `<script>` syntax:

- More succinct code with less boilerplate
- Ability to declare props and emitted events using pure TypeScript
- Better runtime performance (the template is compiled into a render function in the same scope, without an intermediate proxy)
- Better IDE type-inference performance (less work for the language server to extract types from code)

COMPOSITION API: SCRIPT SETUP

We can use all we saw in the setup method directly in the script section.

```
<script setup>
import { ref } from 'vue'
import OtherComponent from '@/components/OtherComponent.vue'

const count = ref(0)
const increment = () => { count.value++ }
</script >

<template>
  <button @click="increment">{{ count }}</button>
  <OtherComponent />
</template>
```

COMPOSITION API: SCRIPT SETUP

Defining props and emits

```
<script setup>
  import { computed } from 'vue'
  const props = defineProps({
    firstName: String,
    lastName: String,
  });
  const emit = defineEmits(['delete'])

  const fullName = computed(() => {
    return `${props.firstName} ${props.lastName}`
  })
  const deleteUser = () => {
    emit('delete')
  }
</script >

<template>
  <button @click="deleteUser">{{ fullName }}</button>
</template>
```

COMPOSITION API: SCRIPT SETUP - TYPESCRIPT

```
<script setup lang="ts">
  import { computed } from 'vue'
  type Props = {
    firstName: string,
    lastName: string,
  }
  const props = defineProps<Props>();
  const emit = defineEmits<{
    (e: 'delete', username: string): void
  }>()

  const fullName = computed(() => {
    return `${props.firstName} ${props.lastName}`
  })
  const deleteUser = () => {
    emit('delete', fullName.value)
  }
</script >

<template>
  <button @click="deleteUser">{{ fullName }}</button>
</template>
```

Lab 12 - Bonus

SSR

OVERVIEW

- What is Vue.js
- Instance
- Tooling
- Syntax
- Components
- Plugins
- Routing
- Data fetching
- Store
- Testing
- Typescript
- Composition
- *SSR*

SERVER SIDE VS CLIENT SIDE RENDERING

Initially, web frameworks had views rendered on the server. Whenever you want to see a new web page, your browser makes a request to the server, and it returns the HTML content.

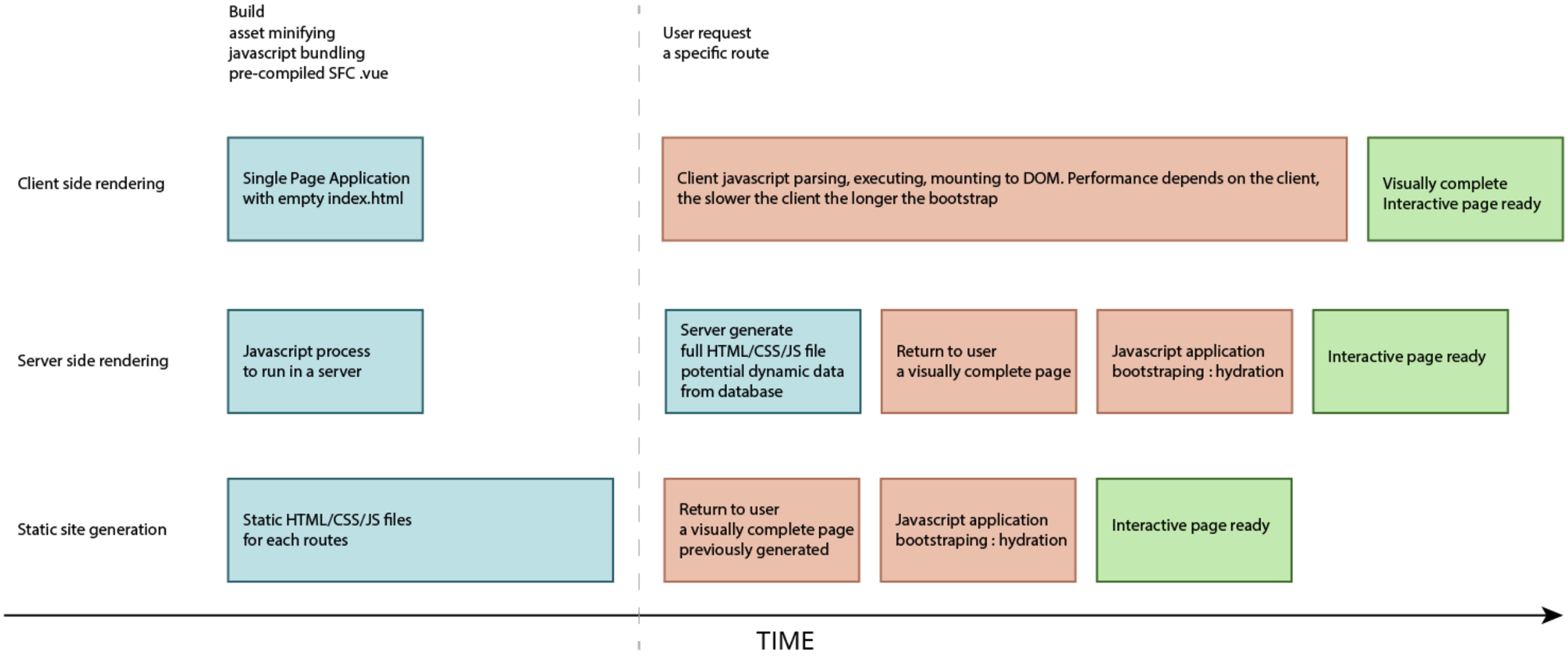
But today, front-end frameworks have introduced that content is rendered directly in the browser using JavaScript.

The main benefit is that the browser will not make another request to the server when you want to load more content. It will reload only the part we want to.

The downside to using client-side rendering is that it can be bad for SEO. Many search engines are unable to properly crawl the content.

Another point is that the initial load can be slow. The browser will first ask the server for the HTML content, then the content of our javascript application which contains our entire application.

SSR / CSR / SSG



NUXT

Nuxt is a great alternative to avoid client side rendering. Nuxt is a framework built on top of the Vue.js. It offers server side rendering and/or static file rendering to boost the performance of your web application. It can be the best of both worlds with it's hybrid approach to render some routes with serverless function and other routes with full statically generated HTML.

⚠️ Nuxt is still in Vue 2. Nuxt 3 with Vue 3 is in beta phase.

NUXT - DIRECTORY STRUCTURE

- `assets`: un-compiled assets such as Less, Sass or JavaScript
- `components`: Vue.js Components. Nuxt.js doesn't supercharge the data method on these components
- `layout`: Application Layouts
- `middleware`: custom functions that can be run before rendering either a page or a group of pages (layouts)
- `pages`: Application Views and Routes.
- `plugins`: Javascript plugins that you want to run before instantiating the root Vue.js Application.
- `static`: static files served on /
- `store`: each file in this folder create a namespaced vuex module.

NUXT - ROUTING

One of the biggest advantage of Nuxt is it's dynamic and file-based routing system.

For example this directory structure:

```
pages/
--| user/
-----| index.vue
-----| _id.vue
--| index.vue
```

NUXT - ROUTING

Will generate the following routing structure:

```
router: {
  routes: [{
    name: 'index',
    path: '/',
    component: 'pages/index.vue'
  }, {
    name: 'user',
    path: '/user',
    component: 'pages/user/index.vue'
  }, {
    name: 'user',
    path: '/user/:id',
    component: 'pages/user/id.vue'
  }]
}
```

Lab 13 - Bonus

END OF VUEJS TRAINING

Thank you

We hope that you enjoyed this training about *VueJS* and will be happy to see you again in other trainings like *TypeScript*, *Vue Advanced*, *GraphQL*...

Let's go evaluate this training at zeval.zenika.com!

QCM DAY 2

QUESTION 1

- Component - How do you define the *data* of a component ?
  - `data: { field: value }`
  - `data: () => ({ field: value })`
  - `data: function() { return { field: value } }`

✅ `data: function() { return { field: value } }` ✅ `data: () => ({ field: value })`

QUESTION 2

- Directive - How many *modifiers* can be chained on a directive ?
  - 1️⃣
  - 3️⃣
  - ∞
  - 0️⃣

✅ ∞

QUESTION 3

- Directive - What will be rendered in the DOM : `<div v-show="false">Test</div>` ?
  - Nothing
  - `<div style="display: none">Test</div>`
  - `<div></div>`

✅ `<div style="display: none">Test</div>`

QUESTION 4

- Component - Which one of these sentences is wrong ?
  - The *computed* option can't use data, props or other computed
  - The *computed* option is available inside the *template*
  - The *computed* option is recomputed only when one of its dependency change
  - The *computed* option is a function whose result is cached until the next function call

✅ The *computed* option can't use data, props or other computed

QUESTION 5

- Component - Which *lifecycle hook* doesn't exist in Vue ?
  - `mounted`
  - `beforeUnmount`
  - `shouldComponentUpdated`
  - `beforeCreated`
  - `updated`

✅ `shouldComponentUpdated`

QUESTION 5

- Directive - How many syntaxes are available for `v-bind` with the `class` ?
  - 1️⃣
  - 2️⃣
  - 3️⃣
  - 4️⃣
  - 5️⃣
  - ∞

✅ 3️⃣, `:class="{'is-favorite': myData}"`, `:class="['is-favorite', 'fa', 'fa-star']"` and `:class="'is-favorite fa fa-star'"`

QUESTION 6

- Component - True/False: A component can be used like a HTML tag, with attributes and children ?
  - True
  - False

✅ True

```
<my-component id="3" class="my-class">
  <p>Some children</p>
</my-component>
```

QUESTION 7

- Directive - What shorthand is available for `v-bind` ?
  - `<div v-bind:id="42" />`
  - `<div :id="42" />`
  - `<div @id="42" />`
  - `<div #id="42" />`
  - `<div $id="42" />`
  - `<div &id="42" />`

✅ `<div :id="42" />`

QCM DAY 3

QUESTION 1

- Plugins - Which sentence is wrong ?
  - A *plugin* can record one or many components globally.
  - A *plugin* can create a new lifecycle hook.
  - A *plugin* can't create a new method in each instance.
  - A *plugin* can't be initialized several times, only the first time matter.

✅ * A *plugin* can't create a new method in each instance.

QUESTION 2

- Store - Which one of the following is the official store manager for Vue3 ?
  - 🍍 Pinia
  - Vuex
  - Redux
  - NgRx
  - Recoil
  - XState

✅ * 🍍 Pinia

QUESTION 3

- Store - How could you update a *store* ?
  - Only inside actions within the same store.
  - Anywhere, as long as you have a reference to the state.
  - Only inside components methods.
  - We can't update a state, we should replace it entirely.

✅ Anywhere, as long as you have a reference to the state.

QUESTION 4

- Store - True/False - Is *Pinia* supported by Vue devtools ?
  - True
  - False

✅ True

QUESTION 5

- Composant - How many dependencies a *computed* can track ?
  - 1️⃣
  - 2️⃣
  - 3️⃣
  - 4️⃣
  - ∞

✅ ∞

QUESTION 6

- Composant - True/False - My *mounted* hook is defined as `async`, is it blocking for the render ?
  - True
  - False

✅ False

QUESTION 7

- Composant - Which sentence is wrong ?
  - You can access to *slots* in the option with `this.$slots`.
  - You defined a *props* as a required `string`, but at runtime if you give a `number` value, Vue will raise an error.
  - You defined a *computed* with the same name as a *props*, Vue will crash when compiling.
  - You defined a *data* intialized with a *props* value, the *data* and the *props* will be equals only at the component initialisation.

✅ You defined a *props* as a required `string`, but at runtime if you give a `number` value, Vue will raise an error.

It will only raise a warning, in dev mode only.

QUESTION 8

- Router - When do you use the `<router-link />` component ?
  - Never, the HTML tag `<a/>` are enough.
  - Only, for internal link of my SPA.
  - For every link, the HTML tag `<a/>` is really not handle properly in Vue.

✅ Only, for internal link of my SPA.