Welcome to the vue.js cross...

.JS

Outline

DON'T SHOOT, I'M A NOOB
NO... PLEASE
pat pat

Before Introduction to Vue.js

Declarative Programming: "What" Should be Done
Imperative Programming: “How” Should be Done

Introduction to Vue.js

A. What is Vue.js?
B. Why use Vue.js?
C. Setting up a Vue.js project

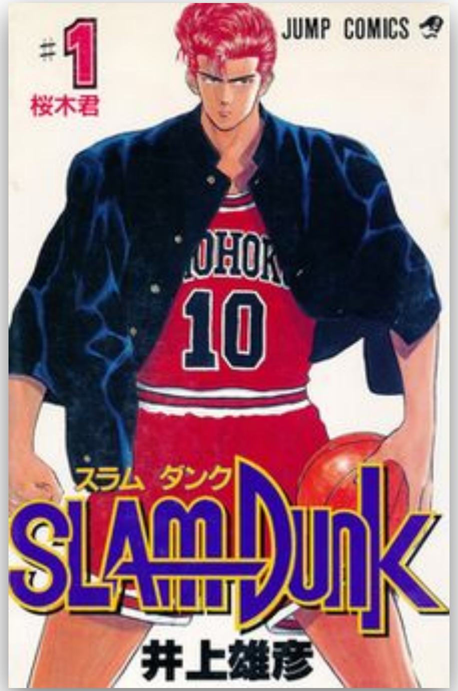
B. Why vue.js

1. Easy learning curve
2. Scalability
3. Component-based architecture
4. Reactive data binding
5. Community and ecosystem
6. Incremental adoption
7. Performance
8. Active development

Vue Fun Facts

- Vue released 10 years ago
- Version names are often derived from manga and anime.
- Vue only weight 33.9kb
- Vue was created by Evan You after working for Google using AngularJS in several projects

Vue 3.4



Setting up a Vue.js project

1. Using Vue CLI

```
npm install -g @vue/cli
vue create project-name
```

2. Using create vue command

```
npm create vue@latest
```

```
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 an End-to-End Testing Solution? › - Use arrow-keys. Return to submit.
    No
    Cypress
    Nightwatch
❯   Playwright
```

Components

- In Vue.js, components are reusable and self-contained units of code that encapsulate HTML, CSS, and JavaScript logic.
- Recommended ways to create theme are by defining them as Single File Components (.vue files).

Example

```
// Welcome.vue
<script setup lang="ts">
    defineProps<{
     msg: string
    }>()
</script>

<template>
 <div class="greetings">
   <h1 class="green">{{ msg }}</h1>
   <h3>
     You’ve successfully created a project with
    </h3>
 </div>
</template>

<style scoped>
</style>
```

Importing a component

```
// HomePage.vue
<script setup lang="ts">
  import Welcome from './Welcome.vue'
</script>

<template>
  <TheWelcome />
</template>
```

Rendering component dynamically

=> Since components are referenced as variables we should use the **:is** binding to render component dynamically

```
<script setup>
import Foo from './Foo.vue'
import Bar from './Bar.vue'
</script>

<template>
  <component :is="Foo" /> // Equivalent with <Foo/>
  <component :is="someCondition ? Foo : Bar" />
</template>
```

Component life cycles

- **Some commonly used lifecycle hooks include:**
    - **beforeCreate**: Called before the instance is created.
    - **created**: Called after the instance is created. Data observation and event initialization occur here.
    - **beforeMount**: Called right before the component is mounted to the DOM.
    - **mounted**: Called after the component is mounted to the DOM.
    - **beforeUpdate**: Called when data changes, before the DOM is re-rendered.
    - **updated**: Called after a data change causes the DOM to be re-rendered.
    - **beforeDestroy**: Called right before a component is destroyed.
    - **destroyed**: Called after a component is destroyed.

Are you crazy who is going to remember all of this?

Don’t worry

We got you covered

Composition API

Composition API

```
<script>
import { ref } from 'vue'

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

    // expose to template and other options API hooks
    return {
      count
    }
  },

  mounted() {
    console.log(this.count) // 0
  }
}
</script>

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

Composition API

Wait a minute I am confused.

Somethings need to be clear before continuing…

If you were focusing 😀, which you don’t, we were seeing two type of script setups.

Composition API

First One

```
<script>
import { ref } from 'vue'

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

    // expose to template and other options API hooks
    return {
      count
    }
  },

  mounted() {
    console.log(this.count) // 0
  }
}
</script>
```

Composition API

<u>Second One</u>

```
<script setup>
import { ref } from 'vue'

// No need to export it, it will automatically be exposed to the template scope.
const count = ref(0)

</script>
```

=> This one is the recommended way of using script tag if you are using a SFC(Single File Component)(i.e. .vue files), which usually you will.

Template Syntax

Vue uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying component instance data.

Under the hood, Vue compiles

Templates □ **Highly-optimized JavaScript code** ➕ **Combined with the reactivity system**

E.g. `<span>count: {{ count }}</span>`

Attribute Binding

Before we move in to a separate section on directive, we have to cross by some of the vue.js magics…

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

// Shorthand
<div :id="dynamicId"></div>

//same-name shorthand
<!-- same as :id="id" -->
<div :id></div>
```

Binding Multiple Attributes

```
const attrs = {

  id: 'container',
  class: 'wrapper'
}
<div v-bind="attrs"></div>
```

Directives

A directive's job is to **reactively** apply **updates** to the **DOM** when the value of its **expression** changes.

Take `v-if` as an example:

```
<p v-if="seen">Now you see me</p>
```

Here, the `v-if` directive would remove / insert the `<p>` element based on the truthiness of the value of the expression `seen`.

Some directive can take an argument for example

v-bind, v-on

E.g.

```
<a v-bind:href="url"> ... </a>
<!-- shorthand →
<a :href="url"> ... </a>
```

```
<a v-on:click="doSomething"> ... </a>
<!-- shorthand -->
<a @click="doSomething"> ... </a>
```

Directives

Dynamic argument

```
<a :[attributeName]="url"> ... </a>
```

```
<a @[eventName]="doSomething"> ... </a> //
```
e.g. if eventName is focus will be equivalent to v-on:focus

Modifiers

Modifiers are special postfixes denoted by a dot, which indicate that a directive should be bound in some special way

```
<form @submit.prevent="onSubmit">...</form>
```

Built In Directives

v-text □ `<span v-text="msg">...</span>`

v-show □ `<span v-show="show">...</span>`

v-if & v-else-if & v-else □

```
<div v-if="see">
  Now you see me
</div>
<div v-else>
  Now you don't
</div>
```

Templates and Directives

Built In Directives

v-for

Expects: `Array | Object | number | string | Iterable`

```
<div v-for="item in items">
  {{ item.text }}
</div>

<div v-for="(value, key) in object"></div>
```

v-once

Render the element and component once only, and skip future updates.

On subsequent re-renders, the element/component and all its children will be treated as static content and skipped. This can be used to optimize update performance.

Best if you are using content management system

`<span v-once>This will never change: {{msg}}</span>`

`Can be use both in` **`Element`** `and` **`Component`**

Built In Directives

v-memo

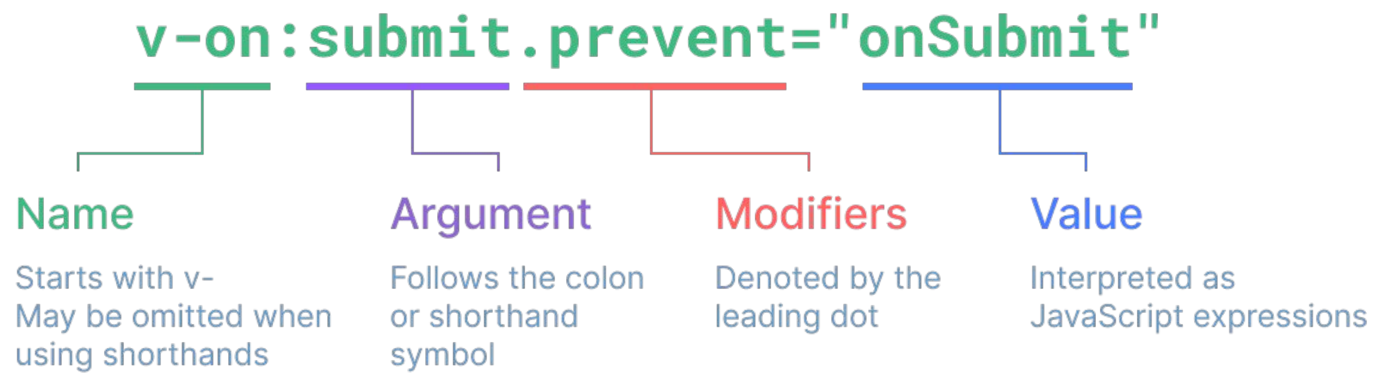
Expects: any[ ]

1. Memoize a sub-tree of the template.
2. `v-memo` is provided solely for micro optimizations in performance-critical scenarios and should be rarely needed.

```
<div v-for="item in list" :key="item.id" v-memo="[item.id === selected]">
  <p>ID: {{ item.id }} - selected: {{ item.id === selected }}</p>
  <p>...more child nodes</p>
</div>
```

The `v-memo` usage here is essentially saying "only update this item if it went from non-selected to selected, or the other way around". This allows every unaffected item to reuse its previous VNode and skip diffing entirely.

Directives


v-on:submit.prevent="onSubmit"
Name
Starts with v-
May be omitted when using shorthands
Argument
Follows the colon or shorthand symbol
Modifiers
Denoted by the leading dot
Value
Interpreted as JavaScript expressions

Built In Directive Reference

Routing in vue

Vue has its own official routing package.

`createWebHistory`

- allow you to have clean and SEO-friendly URLs

You take a look at the about route, this is how you use code spliting in vue.js…

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

const router = createRouter({
 history: createWebHistory(import.meta.env.BASE_URL),
 routes: [
   {
     path: '/',
     name: 'home',
     component: HomeView
   },
   {
     path: '/about',
     name: 'about',
     // route level code-splitting
     // this generates a separate chunk
(About.[hash].js) for this route
     // which is lazy-loaded when the route is
visited.
     component: () =>
import('../views/AboutView.vue')
   }
 ]
})
export default router
```

State Management with Reactivity API

```
// store.js
import { reactive } from 'vue'

export const store = reactive({
  count: 0
})
```

```
> With action
// store.js
import { reactive } from 'vue'

export const store = reactive({
  count: 0,
  increment() {
    this.count++
  }
})
```

State Management with Reactivity API

Usage

```
<script setup>
import { store } from './store.js'
</script>

<template>
  <button @click="store.increment()">
    {{ store.count }}
  </button>
</template>
```

State Management With Pinia Store

Pinia

The intuitive store for Vue.js

- Type Safe
- Extensible, and
- Modular by design.
- Stronger conventions for team collaboration
- Integrating with the **Vue DevTools**, including **timeline**, **in-component inspection**, and **time-travel debugging**
- Hot Module Replacement
- Server-Side Rendering support

The recommended way to manage your states,
...**Vue core team...**

Advance vue concepts

Performance Optimization

Code Splitting and Lazy Loading:

- In Vue.js, you can use dynamic imports (import()) like we see in the router section.

Virtual Scrolling:

- Virtual scrolling is a technique that only renders the visible elements in a list instead of rendering all.
- Vue.js has a built-in `<VirtualScroller>` component that you can use to implement virtual scrolling in your application.

Directives:

- Vue's directives, like **v-once**, **v-memo**, and **v-lazy**, can help you optimize the performance of your application.
- The **v-once** directive, for example, can be used to render an element only once, which can be useful for static content.
- The **v-memo** directive can be used to memoize the rendering of a component, which can be useful for expensive computations.
- The **v-lazy** used to lazy load components or resources,

I Thank You All For Steaking With Me This Long

...If you have a quetion please don’t ask...