# Vue.js

1. Introduction to Vue.js
2. Vue.js Basic Syntax and Instance
3. Data Binding
4. Directives
5. Event Handling
6. Computed Properties
7. Watchers
8. Components
9. Props (Passing Data to Components)
10. Lifecycle Hooks
11. Vue Router (Navigation)
12. Vuex (State Management)
13. Slots (Content Projection)
14. Dynamic & Async Components
15. Filters (Formatting Data)
16. Mixins
17. Custom Directives
18. Transitions & Animations
19. Form Handling & Validation
20. Vue with Axios (HTTP Requests)
21. Vue CLI & Project Structure
22. Vue 3 Composition API
23. Scoped CSS in Vue Components
24. CSS Modules in Vue
25. Vue with TypeScript
26. Code Splitting & Lazy Loading Routes
27. Error Handling in Vue
28. Global Event Bus
29. Provide / Inject API
30. Vue 3 Teleport
31. Summary

## 1. Introduction to Vue.js

Vue.js is an **open-source JavaScript framework** used to build **user interfaces (UIs)** and **single-page applications (SPAs)**.
It focuses on being **progressive**, meaning you can start small (just like adding a script tag to an HTML file) and grow into a full-scale application with routing, state management, and build tools.

- **Developer:** Evan You
- **Initial Release:** February 2014
- **Reason for creation:** To combine the best features of AngularJS (data binding, directives) and React (virtual DOM, component-based) but with a simpler learning curve.

## 2. Vue.js Basic Syntax and Instance

Vue's syntax revolves around **reactive data binding**, **directives**, and **event handling** inside HTML templates.

**Interpolation ({{ }})**

Used to display variables inside HTML.

{{ message }}

**Directives (v- attributes)**

Special HTML attributes that give elements dynamic behavior.
Common directives:

- v-bind → Bind an HTML attribute to a Vue data property
- v-model → Two-way data binding between form input and data
- v-if, v-else-if, v-else → Conditional rendering
- v-for → Loop through data
- v-on (or @) → Listen to DOM events

Example:

<input v-model="name">

Hello {{ name }}

## 3. Data Binding

Data binding is the process of **connecting your JavaScript data with your HTML template**, so when data changes, the UI updates automatically — and vice versa (in two-way binding).

Vue supports **two types** of data binding:

1. **One-way binding** → Data flows from the Vue instance **to the view**.
2. **Two-way binding** → Data flows between **the view and the instance** in both directions.

**One-Way Data Binding**

- Done using {{ }} (interpolation) or v-bind (or : shorthand).
- Updates the view when data changes, but **not** the other way around.

**Example**

```
<!DOCTYPE html>
<html>
<head>
  <title>One-Way Binding</title>
  <script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
</head>
<body>
  <div id="app">
    <p>{{ message }}</p>
    <img :src="imageUrl" alt="Vue Logo" width="100">
  </div>
  <script>
    new Vue({
    el: '#app',
    data: {
       message: 'Hello Vue!',
       imageUrl: 'https://vuejs.org/images/logo.png'
     }
    });
```

```
  </script>
</body>
</html>
```

**Two-Way Data Binding**

- Achieved using the v-model directive.
- Commonly used with form inputs, textareas, and select elements.
- Updates the view when data changes **and** updates the data when the user changes the input.

**Example**

```
<!DOCTYPE html>
<html>
<head>
  <title>Two-Way Binding</title>
  <script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
</head>
<body>
  <div id="app">
    <input v-model="name" placeholder="Enter your name">
    <p>Hello, {{ name }}</p>
  </div>
  <script>
    new Vue({
    el: '#app',
    data: {
        name: "
      }
    });
  </script>
</body>
</html>
```

## 4. Directives

Directives are **special attributes** in Vue.js templates that are prefixed with v-.
They are used to **apply special reactive behavior** to the DOM, such as showing/hiding elements, looping through data, binding attributes, and handling events.

Example syntax:

This will be shown if isVisible is true

**Common Directives in Vue.js**

**A) v-bind — Bind Attributes Dynamically**

- Used to bind HTML attributes to data.
- Shorthand: :

**B) v-model — Two-Way Data Binding**

- Synchronizes form inputs with data.

**C) v-if, v-else-if, v-else — Conditional Rendering**

- Displays elements only if a condition is true.

**D) v-show — Toggle Visibility**

- Similar to v-if, but keeps the element in the DOM and toggles CSS display.

**E) v-for — Looping Through Data**

- Loops through arrays or objects.

**F) v-on — Event Handling**

- Attaches event listeners to elements.
- Shorthand: @

**G) v-text and v-html**

- v-text → Sets text content.
- v-html → Renders raw HTML.

**H) v-pre — Skip Compilation**

- Renders content as-is, without parsing mustache syntax.

**I) v-once — Render Only Once**

- Renders element only the first time and skips updates.

## 5. Event Handling

Event handling in Vue.js allows you to **listen to user actions** (like clicks, key presses, mouse movements) and run **methods** in response.
We use the **v-on** directive (or its shorthand @) to attach event listeners.

Syntax:

@eventName="methodName"

Example:

```
<div id="app">
  <button v-on:click="sayHello">Click Me</button>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
new Vue({
el: '#app',
methods: {
    sayHello() {
      alert('Hello from Vue!');
    }
  }
});
</script>
```

## 6. Computed Properties

- **Computed properties** are properties in Vue that are **derived from other data properties**.
- They are **cached** based on their dependencies and only re-run when those dependencies change.
- Useful when you need to perform calculations or transformations in templates without repeating logic.

**Key Difference from Methods:**

- **Computed** is *reactive and cached.*

- **Methods** are executed every time they are called, even if data hasn't changed.

Syntax

```
computed: {
  propertyName() {
    // return computed value
  }
}
```

## 7. Watchers

- **Watchers** are special Vue features that **observe data changes** and allow you to execute custom logic when a property changes.
- They are useful when you want to **perform side effects** in response to data changes — such as making API calls, validating inputs, or updating other properties.

**Main Difference from Computed Properties:**

- **Computed properties** are for deriving new values from existing data.
- **Watchers** are for running code *in reaction* to changes in data.

Syntax:

```
watch: {
  propertyName(newValue, oldValue) {
    // Code to run when propertyName changes
  }
}
```

Example:

```
<div id="app">
  <p>Message: {{ message }}</p>
  <input v-model="message">
</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
new Vue({
```

```
  el: '#app',
  data: {
    message: 'Hello Vue!'
  },
  watch: {
    message(newVal, oldVal) {
      console.log(`Message changed from "${oldVal}" to "${newVal}"`);
    }
  }
});
</script>
```

## 8. Components

- **Components** are **reusable building blocks** in Vue applications.
- Each component has its **own template, logic, and styles**.
- They help you **break a large application into smaller, manageable pieces**.

**Example:** A blog page may have a **Header component**, **Post component**, and **Footer component** — each defined separately and reused wherever needed.

**Benefits of Using Components**

- **Reusability** → Write once, use multiple times.
- **Maintainability** → Easier to manage and update.
- **Separation of Concerns** → Keep HTML, CSS, and JS related to one piece together.
- **Scalability** → Makes large projects easier to build.

**Types of Components in Vue.js**

1. **Global Components** → Available everywhere in the app.
2. **Local Components** → Available only in the component where they are registered.

**Creating a Global Component**

```
<div id="app">
  <my-component></my-component>
  <my-component></my-component>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
Vue.component('my-component', {
  template: `<p>This is a global component!</p>`
});
new Vue({
  el: '#app'
});
</script>
```

## 9. Props (Passing Data to Components)

- **Props** (short for *properties*) are **custom attributes** you can register on a component.
- They allow **parent components to pass data down** to child components.
- Props are **read-only** inside the child — they should not be modified directly.

**2. Declaring Props**

You declare props in the child component using:

- **Array syntax** → simple list of names.
- **Object syntax** → with types, defaults, and validation.

**Example:**

```
<div id="app">
  <greeting-message name="John"></greeting-message>
</div>
```

```
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
Vue.component('greeting-message', {
  props: ['name'],
  template: `<p>Hello, {{ name }}!</p>`
});
new Vue({
  el: '#app'
});
</script>
```

## 10. Lifecycle Hooks

- Vue components go through a **creation → rendering → updating → destruction** cycle.
- **Lifecycle hooks** are special functions that run at specific points in this cycle.
- They allow you to **inject custom logic** (e.g., fetching data, cleaning up resources).

**A. Creation Phase (Before Render)**

1. **beforeCreate()**
    - Called **after instance initialization**, before reactive data and events are set up.
    - Data and props are **not** available yet.
2. **created()**

- Called **after data observation** and event setup.
- You can access data, computed, methods.
- DOM not mounted yet.

**B. Mounting Phase (Render to DOM)**

3. **beforeMount()**
    - Called right **before the initial render**.
    - Template is compiled but not added to DOM yet.

4. **mounted()**

- Called **after the component is inserted** into the DOM.
- Good for **API calls** or DOM-dependent logic.

**C. Updating Phase (Re-render)**

5. **beforeUpdate()**
    - Called **when reactive data changes**, before re-render.
    - DOM is still showing old data.
6. **updated()**

- Called **after the DOM re-renders**.
- Use with caution (avoid modifying reactive data here to prevent loops).

**D. Destruction Phase (Cleanup)**

7. **beforeDestroy()** (Vue 2) / **beforeUnmount()** (Vue 3)
    - Called **right before component is destroyed**.
    - Good for cleanup tasks (e.g., clearing intervals).
8. **destroyed()** (Vue 2) / **unmounted()** (Vue 3)

- Called **after component is removed** from DOM.

## 11. Vue Router (Navigation)

- **Vue Router** is the official routing library for Vue.js.
- It enables **navigation between pages** without reloading the browser (Single Page Application - SPA).
- It maps **URLs** to **components**.

**Key Concepts in Vue Router**

**A. <router-link>**

- Used for navigation without reloading.
- Automatically updates the browser's URL.

Example:

<router-link to="/about">About</router-link>

**B. <router-view>**

- Placeholder where the matched component is rendered.

Example:

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

**C. Dynamic Routes**

- Pass parameters via :id or :name.

Example:

```
const User = {
  template: '<h2>User ID: {{ $route.params.id }}</h2>'
};
const routes = [
  { path: '/user/:id', component: User }
];
```

## 12. Vuex (State Management)

- **State** = the reactive data your Vue components depend on.
- Defined in a component's **data()** function or in a **store**.
- When state changes, Vue's reactivity system automatically updates the DOM.

**Types of State in Vue**

**(a) Local State**

- Belongs to a single component.
- Stored in the data() function.

**(b) Shared State (via Props & Events)**

- Pass state **down** from parent → child using props.
- Send updates **up** from child → parent using $emit.

**(c) Global State**

- Managed by **Vuex** (Vue 2/3) or **Pinia** (Vue 3 recommended).
- Used when multiple components need access to the same state.

**Vuex State Management Flow**

**Core Concepts**:

1. **State** → Central reactive data store.
2. **Getters** → Compute derived state.
3. **Mutations** → Synchronous state changes.
4. **Actions** → Async logic, then commit mutations.
5. **Modules** → Split store into smaller sections.

**Flow:**

Component → dispatch(action) → commit(mutation) → update(state) → UI updates

## 13. Slots (Content Projection)

- **Slots** allow you to pass **custom content** from a parent component **into a child component's template**.
- Similar to **content projection** in Angular or **props.children** in React.
- They make components more **flexible** and **reusable**.

**Basic Slot**

- Child Component
- Parent Component
- Named Slots
- Scoped Slots

## 14. Dynamic & Async Components

**Dynamic Components in Vue.js**

Dynamic components allow you to **switch between components at runtime** without hardcoding them into the template.

**A. Basic Example**

```
<div id="app">
  <button @click="currentView = 'home'">Home</button>
  <button @click="currentView = 'about'">About</button>
  <!-- Dynamic Component -->
  <component :is="currentView"></component>
```

```
</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
Vue.component('home', {
  template: '<h2>Home Component</h2>'
});
Vue.component('about', {
  template: '<h2>About Component</h2>'
});
new Vue({
  el: '#app',
  data: {
    currentView: 'home'
  }
});
</script>
```

**B. With <keep-alive>**

- Prevents a component from being destroyed when switched out (keeps state in memory).

**C. Dynamic Components with Props**

**Async Components in Vue.js**

Async components are **loaded only when needed** — great for **code-splitting** and **performance**.

**A. Basic Async Component**

```
Vue.component('async-example', function (resolve, reject) {
  setTimeout(() => {
    resolve({
      template: '<h2>I was loaded asynchronously!</h2>'
    });
  }, 2000);
```

});

**B. Using ES6 Dynamic Import**

Modern and preferred approach:

Vue.component('async-view', () => import('./AsyncView.vue'));

**C. Async Components with Webpack's import()**

Example with route-based lazy loading:

const UserProfile = () => import('./UserProfile.vue');

**D. Async Components with Loading & Error States**

```
const AsyncComp = () => ({
  component: import('./MyComponent.vue'),
  loading: { template: '<p>Loading...</p>' },
  error: { template: '<p>Failed to load component.</p>' },
  delay: 200, // ms before showing loading
  timeout: 3000 // ms before showing error
});
new Vue({
  el: '#app',
  components: { AsyncComp }
});
```

## 15. Filters (Formatting Data)

Filters are used to **format or transform data before displaying it** in the template. They don't change the underlying data — only how it's **presented** in the view.

**Note:** In Vue 3, filters have been removed from the core and replaced by **methods** or **computed properties**, but they are still common in Vue 2.

Basic Syntax

{{ value | filterName }}

- value → The data you want to format.
- filterName → The filter function name.

## 16. Mixins

A **mixin** is a way to **re-use reusable functionality** across multiple Vue components.
They let you define:

- **Data properties**
- **Methods**
- **Lifecycle hooks**
- **Computed properties**
  …and merge them into components.

Think of them as **shared blueprints** for components.

**Why Use Mixins?**

- Avoid **code duplication**
- Keep code **modular** and **organized**
- Share logic between components without repeating code

Creating a Mixin:

```
const myMixin = {
  data() {
    return {
      message: "Hello from Mixin"
    }
  },
  methods: {
    greet() {
      console.log(this.message);
    }
  }
};
```

**Using a Mixin in a Component:**

```
new Vue({
  el: '#app',
  mixins: [myMixin],
  mounted() {
```

```
    this.greet(); // Output: Hello from Mixin
  }
});
```

## 17. Custom Directives

Vue comes with built-in directives like:

- v-model
- v-if
- v-for
- v-show

Sometimes, you need **your own directive** to add reusable DOM behavior.
Custom directives let you **directly manipulate the DOM** when elements are bound or updated.

Using in a Component

<input v-focus>

## 18. Transitions & Animations

- **Transitions** → Handle entering & leaving animations (e.g., fade in/out, slide in/out).
- **Animations** → Can be CSS animations or JavaScript-driven effects.
- Vue provides the <transition> and <transition-group> components to make this easy.

**Basic Transition Example**

- **Fade In/Out with CSS**

```
<template>
  <div>
    <button @click="show = !show">Toggle</button>
    <transition name="fade">
      <p v-if="show">Hello Vue Transitions!</p>
    </transition>
  </div>
</template>

<script>
```

```
export default {
  data() {
    return { show: true };
  }
};
</script>
<style>
.fade-enter-active, .fade-leave-active {
  transition: opacity 0.5s;
}
.fade-enter, .fade-leave-to {
  opacity: 0;
}
</style>
```

## 19. Form Handling & Validation

### Basic Form Handling

In Vue, handling form input is straightforward with **two-way data binding** using the v-model directive.

```
<template>
  <form @submit.prevent="submitForm">
    <input v-model="name" placeholder="Enter your name" />
    <button type="submit">Submit</button>
  </form>
  <p>Your name is: {{ name }}</p>
</template>
<script>
export default {
  data() {
    return {
      name: ''
    };
  },
  methods: {
    submitForm() {
      alert(`Form submitted with name: ${this.name}`);
```

```
    }
  }
};
</script>
```

**Handling Different Input Types**

- Text inputs: v-model="text"
- Checkboxes: v-model="checked"
- Radio buttons: v-model="picked"
- Select dropdowns: v-model="selected"

**Basic Validation**

- You can manually validate data inside your methods or watchers.

```
methods: {
  submitForm() {
    if (!this.name) {
      alert("Name is required");
      return;
    }
    // Proceed with submission
  }
}
```

## 20. Vue with Axios (HTTP Requests)

- Axios is a **promise-based HTTP client** for browsers and Node.js.
- Supports GET, POST, PUT, DELETE, and more.
- Easy to use for sending requests and handling responses.
- Works well with Vue for communicating with APIs.

**Installing Axios**

- You can install Axios using npm or yarn:

```
npm install axios
# or
yarn add axios
```

**Basic Usage in Vue**

Example: Fetch data from an API

```
<template>
  <div>
    <h2>Users</h2>
    <ul>
      <li v-for="user in users" :key="user.id">{{ user.name }}</li>
    </ul>
    <button @click="fetchUsers">Refresh</button>
  </div>
</template>

<script>
import axios from 'axios';

export default {
  data() {
    return {
      users: [],
      error: null,
    };
  },
  methods: {
    fetchUsers() {
      axios.get('https://jsonplaceholder.typicode.com/users')
        .then(response => {
          this.users = response.data;
          this.error = null;
        })
        .catch(error => {
          this.error = 'Failed to fetch users.';
          console.error(error);
        });
    }
  },
  mounted() {
    this.fetchUsers();
  }
};
</script>
```

**Making Different Types of Requests**

- **GET request:** axios.get(url)
- **POST request:** axios.post(url, data)
- **PUT request:** axios.put(url, data)
- **DELETE request:** axios.delete(url)

Example POST:

```
axios.post('/api/posts', {
  title: 'New Post',
  body: 'Post content'
})
.then(response => console.log(response))
.catch(error => console.error(error));
```

## 21. Vue CLI & Project Structure

- Vue CLI (Command Line Interface) is the official tool to **scaffold Vue.js projects quickly**.
- It helps set up a Vue project with build tools, hot-reloading, linting, testing, and more — all configured and ready to go.
- Supports plugins and presets, allowing customization.

### Typical Project Structure Generated

```
my-vue-app/
├── node_modules/          # Installed dependencies
├── public/
│   ├── favicon.ico        # Icon displayed in browser tab
│   └── index.html         # HTML entry point
├── src/
│   ├── assets/            # Images, fonts, styles
│   ├── components/        # Vue components
│   ├── views/             # View-level components (if using router)
│   ├── App.vue            # Root Vue component
│   ├── main.js            # Application entry point (initializes Vue app)
│   └── router/            # Router configuration (if Vue Router installed)
│       └── index.js
│   └── store/             # Vuex store files (if Vuex installed)
│       └── index.js
├── tests/                 # Unit or e2e test files (optional)
├── .gitignore             # Git ignore rules
├── babel.config.js        # Babel config
├── package.json           # Project metadata & dependencies
├── README.md              # Project documentation
└── vue.config.js          # Optional Vue CLI configuration file
```

## 22. Vue 3 Composition API

- Introduced in Vue 3 as a new way to write Vue components.
- Allows organizing component logic by feature or concern, rather than by option (data, methods, computed, etc.).
- Uses functions (like setup()) and reactive primitives (ref, reactive, computed) for more flexible and reusable code.
- Why Use the Composition API?

| Traditional Options API | Composition API |
| --- | --- |
| Logic spread across `data`, `methods`, `computed` etc. | Logic grouped by feature inside `setup()` |
| Harder to reuse and share logic | Easier to extract and reuse logic with composables |
| Can get messy in large components | Cleaner, more maintainable codebase |
| Relies heavily on `this` context | Uses explicit imports and variables |

**Watchers and WatchEffect**

- **watch()** to watch reactive variables or computed properties.
- **watchEffect()** to run reactive effects immediately and on dependencies change.

**Example:**

```
import { ref, watch, watchEffect } from 'vue';

setup() {
  const count = ref(0);
  watch(count, (newVal, oldVal) => {
    console.log(`count changed from ${oldVal} to ${newVal}`);
  });
  watchEffect(() => {
    console.log(`count is now: ${count.value}`);
  });
  return { count };
}
```

## 23. Scoped CSS in Vue Components

- Scoped CSS ensures **styles defined in a Vue component only apply to that component**, not globally.
- Prevents **style leakage** or conflicts between components.
- Useful for **modular, maintainable styles**.

**How to Use Scoped CSS in Vue**

- Add the `scoped` attribute to the `<style>` tag inside a `.vue` file:

```
<template>
  <div class="box">
```

```
    Scoped Styled Box
  </div>
</template>

<style scoped>
.box {
  background-color: lightblue;
  padding: 10px;
}
</style>
```

**What Happens Under the Hood?**

- Vue's build process **adds a unique attribute** to the component's root HTML elements (e.g., data-v-123abc).
- It also **appends this attribute to CSS selectors** in the scoped styles.
- This makes styles **apply only to elements inside this component**.

Example:

```
<div class="box" data-v-123abc>Scoped Styled Box</div>
```

## 24. CSS Modules in Vue

- CSS Modules provide **locally scoped CSS by default**.
- Each class name is **hashed and unique** to avoid naming collisions.
- Unlike scoped CSS (which adds unique attributes), CSS Modules **generate unique class names** and export them as JavaScript objects.
- Great for **component-based styles with predictable, collision-free class names**.

**.How CSS Modules Work in Vue**

- When you add module to your <style> tag, Vue treats it as a CSS Module.
- Your classes are accessible in the component script via a special $style object.
- You apply styles by binding :class="$style.className" or in templates with :class="$style['class-name']".

**Basic Usage**

```
<template>
  <div :class="$style.box">
    This box is styled with CSS Modules!
  </div>
</template>
```

```
<style module>
.box {
  background-color: coral;
  padding: 20px;
  color: white;
}
</style>
```

## 25. Vue with TypeScript

**Why Use TypeScript with Vue?**

- Adds **static typing** for better developer experience.
- Helps catch errors at compile time, improving code quality.
- Enables **autocomplete, type checking, and refactoring** in IDEs.
- Supports modern JS features with added type safety.
- Especially useful in **large scale apps**.

## Setting Up a Vue 3 + TypeScript Project

- The easiest way:
  vue create my-vue-ts-app

Choose **Manually select features**

- Select **TypeScript**
- Optionally choose **Class-style component syntax** or **Composition API** (recommended)
- Vue CLI sets up everything including tsconfig.json

## 26. Code Splitting & Lazy Loading Routes

## What is Code Splitting?

- Code splitting is the practice of **dividing your app's JavaScript into smaller chunks** that can be loaded on demand.

- Instead of loading the entire app bundle upfront, only the needed code is loaded initially.
- Improves **performance**, reduces initial load time.

**What is Lazy Loading?**

- Lazy loading is a technique to **load resources only when they are needed**.
- In Vue Router, lazy loading means loading a route's component **only when the route is visited**.
- Combines well with code splitting.

**Why Use It?**

| Benefit | Description |
|---|---|
| Faster initial load time | Smaller initial bundle size |
| Better user experience | Load only what user needs |
| Reduces bandwidth usage | Saves data by loading on demand |
| Scales well for large apps | Improves maintainability & performance |

## 27. Error Handling in Vue

**Types of Errors in Vue**

| Error Type | Description |
|---|---|
| Runtime Errors | Errors during component rendering or lifecycle hooks |
| Async Errors | Errors from asynchronous code (e.g., API calls) |
| Global Errors | Errors that happen anywhere in the app |
| Custom Application Errors | Business logic errors you want to catch and handle |

**Vue 2 vs Vue 3 Error Handling**

- Both versions support error capturing hooks.
- Vue 3 adds errorCaptured and global error handler via app.config.errorHandler.

**Component-Level Error Capturing**

- Use the `errorCaptured` lifecycle hook to catch errors from child components:

```
export default {
  errorCaptured(err, vm, info) {
    console.error('Error captured:', err);
    console.log('Info:', info);
    // Return false to stop propagation
    return false;
  }
}
```

## 28. Global Event Bus

- An event bus is a **central Vue instance used solely for emitting and listening to events**.
- Acts as a **messaging hub** to enable communication between components that do **not** have a direct parent-child relationship.
- Useful for loosely coupled components to communicate.

**How to Create a Global Event Bus**

- **Vue 2 Example:**

```
// event-bus.js
import Vue from 'vue';
export const EventBus = new Vue();
```

**Vue 3 and the Event Bus**

- Vue 3 **removed the global event API** ($on, $off, $emit on Vue instances).
- To create an event bus in Vue 3, use an **external event emitter** like Node.js's mitt.

Example with mitt:

npm install mitt

## 29. Provide / Inject API

- A way for ancestor components to provide data or dependencies to descendant components, no matter how deep, without passing props through every intermediate component.
- Useful for dependency injection, sharing global services, or passing data across deeply nested components.

**When to Use Provide / Inject?**

- When multiple nested components need access to the same data or service.
- To avoid **prop drilling** (passing props through components that don't need them).
- For plugin or library authors to inject dependencies.
- Example use cases: themes, localization, shared state, API clients.

**Basic Syntax**

- **Provider Component:**

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

export default {
  setup() {
    const theme = 'dark';
    provide('themeKey', theme);

    // ...rest of setup
  }
}
```

## 30. Vue 3 Teleport

- Teleport is a built-in Vue 3 feature that lets you render a component's template (or part of it) into a different place in the DOM outside the component's own hierarchy.
- Useful for UI elements that need to visually appear outside their parent container but remain logically connected — like modals, tooltips, dropdowns, or notifications.

**Why Use Teleport?**

| Problem Without Teleport | How Teleport Helps |
|---|---|
| Modals may get clipped by overflow:hidden or stacking context issues | Teleport renders modals directly in the root or target container |
| Tooltips or dropdowns nested deeply can cause positioning or style issues | Teleport moves them to a better DOM location |
| Complex CSS workarounds needed to fix z-index or overflow | Teleport simplifies DOM structure for such UI elements |

**Basic Usage**

Wrap the content you want to teleport inside <teleport> and provide a CSS selector for the target:

```
<template>
  <div>
    <h1>Main Content</h1>
    <teleport to="body">
      <div class="modal">
        <p>This modal is rendered inside &lt;body&gt;, not here.</p>
      </div>
    </teleport>
  </div>
</template>
```

## 31. Summary:

- Components have template, script, and style sections.
- Reactive data defined in data() or via Composition API (ref, reactive).
- Communication via props, events, provide/inject, or centralized state.

**Why Use Vue.js?**

- Easy to learn with clear documentation.
- Lightweight and performant.
- Flexible: Use as a library or a full framework.
- Strong community and ecosystem.
- Great tooling support.