Introducción a Vue.js: Props, Componentes, Eventos, Enrutamientos, Estado Global y Ciclo de Vida

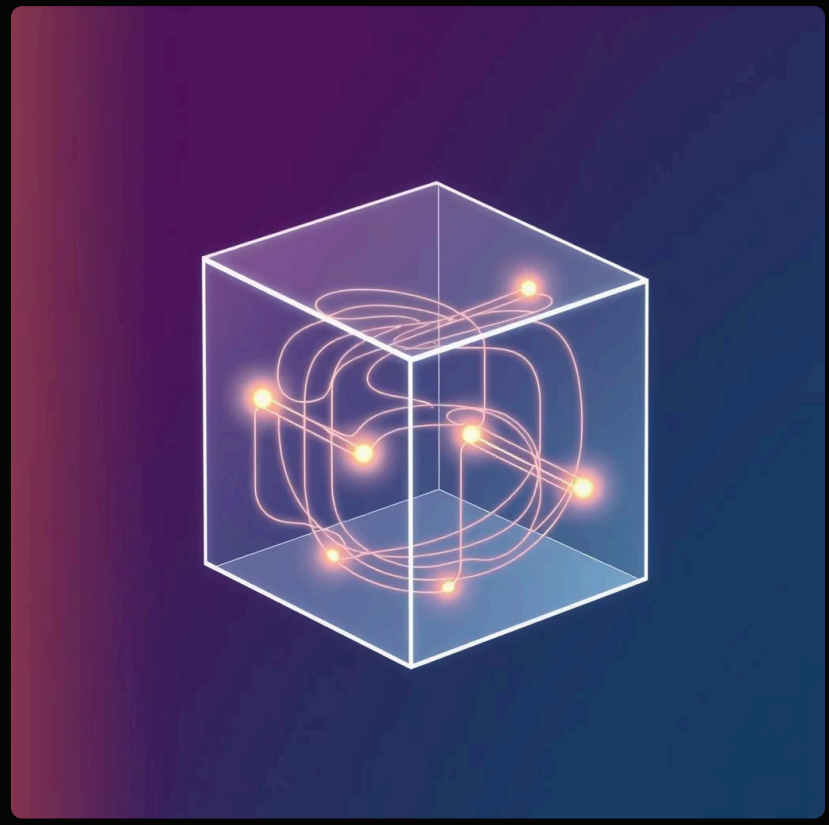
Desbloqueando el Poder de las Aplicaciones Reactivas

Presentado por Jhan Carlos Zamora, David Ruiz, Jhon Neiver Valencia y Gabriel Sánchez

El Corazón de Vue.js: Los Componentes

En Vue.js, los componentes son los bloques fundamentales de construcción para cualquier aplicación. Imagínelos como piezas de LEGO: cada componente encapsula su propia estructura (HTML), lógica (JavaScript) y estilo (CSS), haciéndolos completamente autónomos y reutilizables.

Esta modularidad permite dividir interfaces de usuario complejas en unidades pequeñas y manejables, lo que facilita el desarrollo, la depuración y el mantenimiento. Un componente puede ser tan simple como un botón o tan complejo como una sección entera de un sitio web.



Ejemplo práctico:

Un componente de "Botón Contador" que mantiene su propio estado de clics, sin afectar otras partes de la aplicación.

Props: Flujo de Datos Descendente

Definición Esencial

Las "props" (abreviatura de propiedades) son la forma principal en que los datos se pasan de un componente padre a un componente hijo en Vue.js. Piensa en ellas como atributos que le das a un elemento HTML, pero para tus componentes personalizados.

Declaración y Validación

- Puedes declarar props de forma sencilla con un array: `props: ['titulo', 'likes']`.
- Para mayor robustez, se recomienda usar un objeto para definir el tipo, si es requerido y un valor por defecto: `props: { titulo: String, likes: { type: Number, default: 0 }}`.
- La validación ayuda a prevenir errores y documenta claramente las expectativas del componente.

Uso Práctico

Para pasar una prop, simplemente la enlazas usando `v-bind` (o su atajo `:`) en la plantilla del componente padre:

```
<ChildComponent
:titulo="mensajeDesdePadre" :likes="10"
/>
```

Esto asegura que el componente hijo reciba los datos correctos y reactivos del padre.

Las props son fundamentales para crear componentes reutilizables y gestionar el flujo de datos de manera predecible en aplicaciones Vue.js.

Eventos: Comunicación Ascendente del Hijo al Padre

Mientras que las props permiten que los padres pasen datos a los hijos, los eventos permiten que los componentes hijos comuniquen acciones o cambios de estado de vuelta a sus padres. Este es el mecanismo de "flujo de datos ascendente" en Vue.js.

Emisión de Eventos

Un componente hijo puede "emitir" un evento usando el método `$emit()`. Esto notifica al componente padre que algo ha ocurrido.

```
this.$emit('nombreEventoPersonalizado', datosOpcionales)
```

Puedes incluir datos adicionales para enviar al padre, como el nuevo valor de un campo o un identificador.

Escucha de Eventos

El componente padre escucha estos eventos personalizados usando la sintaxis `v-on` (o su atajo `@`) en la plantilla, igual que escucharía eventos nativos del DOM.

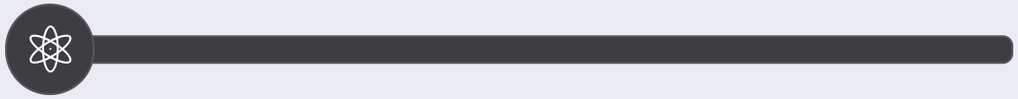
```
<ChildComponent
@nombreEventoPersonalizado="manejadorEnPadre" />
```

El "manejadorEnPadre" es una función definida en el script del padre que se ejecutará cuando el evento sea emitido, recibiendo los datos opcionales si los hay.

Ejemplo: Un componente de "Botón" en el hijo emite un evento `@clic` cada vez que se presiona. El componente padre puede escuchar este evento para incrementar un contador global o realizar otra acción.

El Ciclo de Vida de un Componente Vue.js

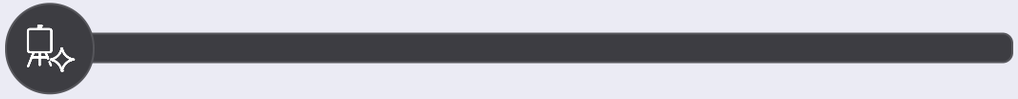
Cada componente en Vue.js atraviesa una serie de etapas desde su creación hasta su destrucción. Estas etapas se conocen como el "ciclo de vida" de un componente, y Vue proporciona "hooks" (ganchos) que te permiten ejecutar código en momentos específicos de este ciclo.



Creación

beforeCreate: El componente se inicializa, pero los datos y eventos reactivos aún no están disponibles.

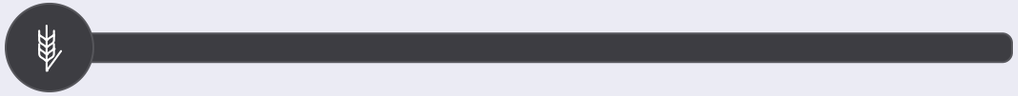
created: El componente ha sido creado. Los datos reactivos y los eventos ya están configurados. Es ideal para llamadas a API.



Actualización

beforeUpdate: Los datos reactivos han cambiado y el componente está a punto de re-renderizarse. Puedes acceder al DOM antes de la actualización.

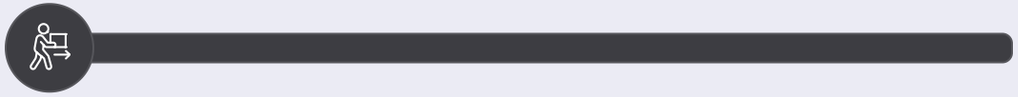
updated: El componente se ha re-renderizado en el DOM debido a cambios en los datos. Ten cuidado al manipular el DOM aquí para evitar bucles infinitos.



Montaje

beforeMount: La plantilla del componente ha sido compilada, pero aún no se ha montado en el DOM.

mounted: El componente ha sido montado en el DOM real. Puedes acceder a los elementos del DOM y realizar manipulaciones directas aquí.



Desmontaje

beforeUnmount: El componente está a punto de ser destruido. Es el lugar perfecto para limpiar recursos.

unmounted: El componente ha sido completamente destruido y removido del DOM. Ya no hay elementos asociados a él.

Comprender estos hooks te permite controlar el comportamiento de tu componente en cada fase de su existencia.

Ejemplo Práctico: Implementando Hooks del Ciclo de Vida

Los hooks del ciclo de vida son cruciales para ejecutar lógica en momentos específicos de la vida de un componente. Aquí te mostramos cómo se usarían algunos de los más comunes y sus propósitos.

Carga de Datos Inicial

```
// Enviar peticiones HTTP a APIs
```

```
created() {
  console.log('Componente creado: obteniendo datos...');
  // axios.get('/api/data').then(response => {
  //   this.datos = response.data;
  // });
}
```

Cuándo usar: `created()` es ideal para inicializar el estado de los datos, realizar llamadas a la API y configurar escuchadores no relacionados con el DOM, ya que el componente ha sido creado y sus datos reactivos están disponibles.

Manipulación del DOM

```
// Acceder a elementos HTML
```

```
mounted() {
  console.log('Componente montado: DOM disponible.');
  // new Chart(this.$refs.myCanvas, config);
}
```

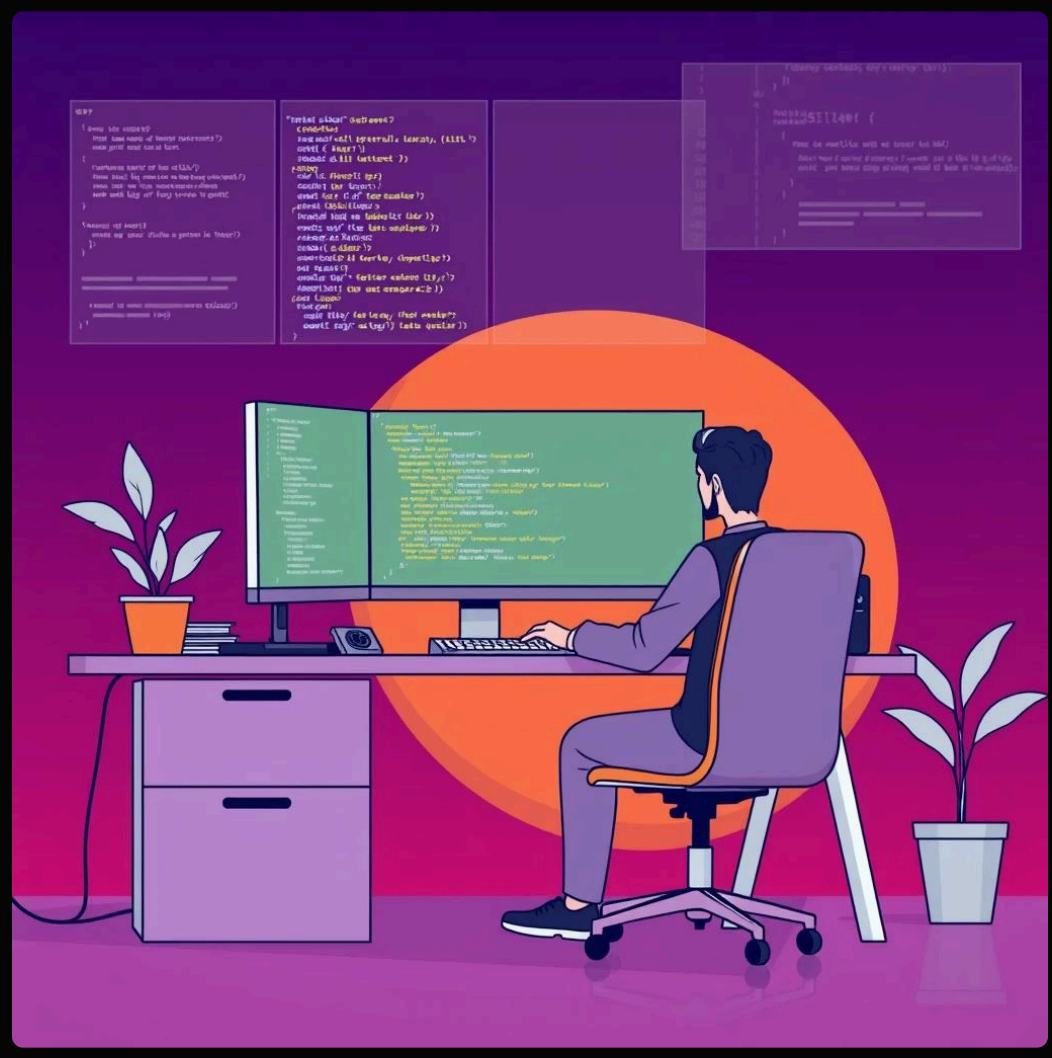
Cuándo usar: `mounted()` es el mejor lugar para interactuar con el DOM del componente (por ejemplo, acceder a elementos con `this.$refs`), integrar librerías de terceros que requieran un elemento DOM, o realizar operaciones que necesiten que el componente ya esté visible en la página.

Limpieza de Recursos

```
// Eliminar suscripciones, timers
```

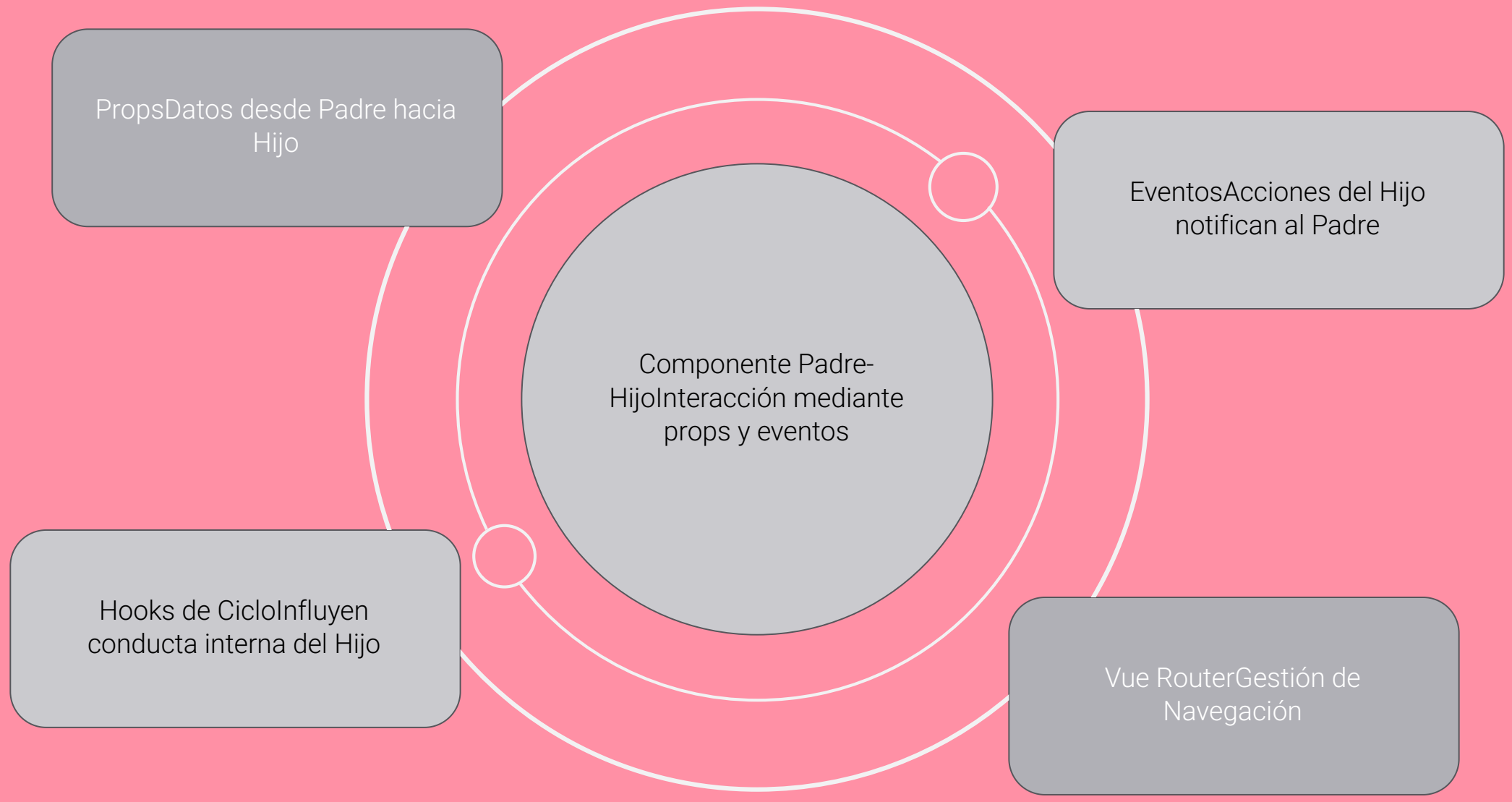
```
beforeUnmount() {
  console.log('Componente a punto de ser desmontado:
limpiando...');
  // clearInterval(this.timerId);
  // eventBus.off('some-event', this.handler);
}
```

Cuándo usar: `beforeUnmount()` es crucial para la "limpieza" de la aplicación. Aquí debes cancelar temporizadores, eliminar escuchadores de eventos globales (como los de `window` o un `EventBus`), o cancelar suscripciones para evitar fugas de memoria y asegurar que tu aplicación se comporte correctamente al destruir componentes.



La Sinergia: Props + Eventos + Ciclo de Vida + Vue Router + Pinia

La verdadera potencia de Vue.js reside en la forma en que todos estos conceptos —props, eventos, el ciclo de vida, Vue Router y Pinia— trabajan juntos para crear aplicaciones interactivas, robustas y completas, conformando el ecosistema integral de Vue.js.



- **Props:** Permiten que el Padre pase información relevante al Hijo para que este se configure y muestre los datos apropiadamente.
- **Eventos:** Habilitan que el Hijo notifique al Padre sobre acciones del usuario o cambios internos, permitiendo que el padre reaccione y actualice su propio estado.
- **Ciclo de Vida:** Asegura que estas comunicaciones y otras lógicas (como la carga inicial de datos o la limpieza de recursos) se ejecuten en el momento preciso durante la vida del componente.
- **Vue Router:** Facilita la navegación entre diferentes vistas o páginas de la aplicación, integrándose profundamente con el ciclo de vida de los componentes.
- **Pinia:** Proporciona una solución de gestión de estado centralizada y reactiva, permitiendo compartir datos entre componentes de manera eficiente y escalable.

Este modelo, complementado con herramientas como Vue Router y Pinia, permite una arquitectura de componentes clara, fácil de entender y potente, donde cada parte tiene responsabilidades bien definidas.

Vue Router: Enrutamiento en Vue.js

Vue Router es el sistema oficial de enrutamiento para Vue.js, esencial para construir **Aplicaciones de Una Sola Página (SPA)**. Permite que tu aplicación navegue entre diferentes "vistas" o componentes sin recargar la página completa, ofreciendo una experiencia de usuario fluida y dinámica.

Instalación y Configuración Básica

Primero, instala Vue Router en tu proyecto. Luego, configura tu enrutador principal, definiendo el modo de historial y el array de rutas.

```
npm install vue-router@4
```

```
// src/router/index.js
import { createRouter, createWebHistory } from 'vue-router';

const routes = [
  // Tus rutas irán aquí
];

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

export default router;
```

Asegúrate de importar y usar este enrutador en tu archivo `main.js`.

Definición de Rutas y Renderización

Cada ruta es un objeto que mapea una URL a un componente específico. El componente `<router-view />` actúa como un marcador de posición donde se renderizará el componente de la ruta actual.

```
// src/router/index.js
import HomeView from '../views/HomeView.vue';
import AboutView from '../views/AboutView.vue';

const routes = [
  { path: '/', name: 'home', component: HomeView },
  { path: '/about', name: 'about', component: AboutView }
];
```

```
<!-- src/App.vue -->
<template>
  <nav>
    <router-link to="/">Inicio</router-link>
    <router-link to="/about">Acerca de</router-link>
  </nav>
  <router-view /> <!-- Aquí se renderizan los componentes -->
</template>
```

Utiliza `<router-link>` para la navegación declarativa.

Navegación Programática y Parámetros

Puedes navegar entre rutas mediante programación usando `this.$router.push()`. Los parámetros de ruta te permiten pasar datos dinámicos a tus componentes, como el ID de un usuario.

```
// Navegación programática
this.$router.push('/about');
this.$router.push({ name: 'home', params: { userId: '123' } });
```

```
// Definición de ruta con parámetro
const routes = [
  { path: '/users/:id', name: 'user-profile', component: UserProfile }
];

// Acceso al parámetro dentro de UserProfile.vue
<template>
  <h1>Perfil de Usuario: {{ $route.params.id }}</h1>
</template>
```

Los parámetros son reactivos y accesibles a través de `this.$route.params`.

Guards de Navegación

Los Guards de Navegación son funciones que se ejecutan antes de que se complete una navegación. Son ideales para autenticación, autorización o para confirmar acciones del usuario antes de salir de una página.

```
// src/router/index.js
router.beforeEach((to, from, next) => {
  const isAuthenticated = false; // Simula el estado de autenticación
  if (to.meta.requiresAuth && !isAuthenticated) {
    next('/login'); // Redirige a la página de login
  } else {
    next(); // Continúa con la navegación
  }
});

// Definición de ruta con meta field
const routes = [
  { path: '/dashboard', component: Dashboard, meta: { requiresAuth: true } }
];
```

Esto permite un control fino sobre el flujo de navegación de tu aplicación.

Pinia: Gestión de Estado Global en Vue.js

En aplicaciones complejas, el estado (los datos que cambian a lo largo del tiempo) puede volverse difícil de manejar cuando se comparte entre múltiples componentes. La **gestión de estado global** proporciona un patrón centralizado para organizar, acceder y mutar este estado de manera predecible y sencilla. Pinia es la solución de gestión de estado recomendada para Vue.js, ofreciendo una experiencia de desarrollo intuitiva y de alto rendimiento.

Ligero y Rápido

Pinia es increíblemente ligero, con un tamaño mínimo y sin dependencias pesadas, lo que contribuye a un rendimiento óptimo de tu aplicación.

Modular y Organizado

Permite organizar tu estado en "stores" modulares, facilitando la escalabilidad y el mantenimiento de aplicaciones grandes.

Tipado Seguro

Diseñado con TypeScript en mente, ofrece inferencia de tipos completa para el estado, getters y acciones, mejorando la seguridad y la experiencia del desarrollador.

Devtools Mejoradas

Se integra perfectamente con las Vue Devtools, proporcionando una experiencia de depuración excepcional, incluyendo seguimiento de acciones, mutaciones y líneas de tiempo.

---

1. Instalación y Configuración Básica

Para empezar a usar Pinia, primero debes instalarlo en tu proyecto y luego configurarlo en tu aplicación Vue.

```
npm install pinia
# o
yarn add pinia
```

```
// src/main.js
import { createApp } from 'vue';
import { createPinia } from 'pinia';
import App from './App.vue';

const app = createApp(App);
const pinia = createPinia();

app.use(pinia);
app.mount('#app');
```

Esto inicializa Pinia y lo hace disponible para toda tu aplicación Vue.

2. Creando un Store (Estado, Getters y Acciones)

Un "store" es un contenedor de estado. Lo defines usando `defineStore()`, donde puedes especificar el estado reactivo, los getters (propiedades computadas del store) y las acciones (métodos para cambiar el estado).

```
// src/stores/counter.js
import { defineStore } from 'pinia';

export const useCounterStore = defineStore('counter', {
  state: () => ({
    count: 0,
    name: 'Eduardo'
  }),
  getters: {
    doubleCount: (state) => state.count * 2,
    doubleCountPlusOne() {
      return this.doubleCount + 1;
    }
  },
  actions: {
    increment() {
      this.count++;
    },
    async fetchNewCount() {
      // Simula una llamada a API
      const newCount = await Promise.resolve(100);
      this.count = newCount;
    }
  }
});
```

Cada store tiene un ID único (ej. `'counter'`) y es una función que exportas para usar en tus componentes.

---

3. Usando el Store en un Componente

Para acceder al estado, getters y acciones de un store en tus componentes de Vue, simplemente importas y llamas a la función del store.

```
<!-- src/components/CounterComponent.vue -->
<template>
  <div>
    <h3>Contador: {{ counter.count }}</h3>
    <p>Doble Contador: {{ counter.doubleCount }}</p>
    <button @click="counter.increment()">Incrementar</button>
    <button @click="counter.fetchNewCount()">Cargar Nuevo</button>
  </div>
</template>

<script setup>
import { useCounterStore } from '../stores/counter';

const counter = useCounterStore();

// También puedes desestructurar para usar propiedades directamente
// import { storeToRefs } from 'pinia';
// const { count, doubleCount } = storeToRefs(counter);
// const { increment, fetchNewCount } = counter;
</script>
```

Pinia se integra perfectamente con la Composition API y Options API, haciendo que el manejo del estado sea una parte natural del desarrollo de componentes.

Integración de Vue Router y Pinia: Gestión de Estado en la Navegación

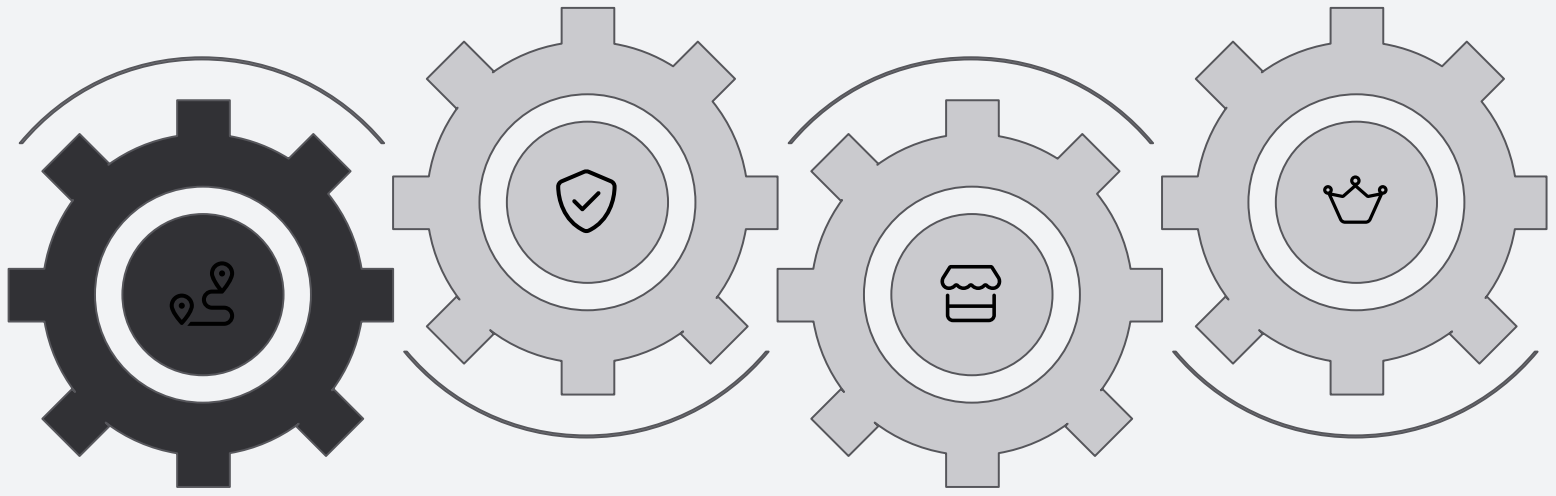
La combinación de Vue Router y Pinia es fundamental para construir aplicaciones Vue.js robustas y escalables. Permite gestionar el estado global de tu aplicación de forma centralizada y reaccionar a los cambios de ruta para controlar el acceso, cargar datos o simplemente mantener la consistencia de la interfaz de usuario.

Navigation Guards

Check auth via store before enter

Vue Components

React to route and state changes



Vue Router

Manages route changes and guards

Pinia Auth Store

Holds user token and auth state

1. Usando Stores en Guards de Navegación

Los Guards de Navegación de Vue Router son el lugar perfecto para integrar la lógica de autenticación o autorización de tu Pinia Store. Antes de permitir el acceso a una ruta protegida, puedes verificar el estado de autenticación o los roles del usuario directamente desde tu store.

```
// src/router/index.js
import { createRouter, createWebHistory } from 'vue-router';
import { useAuthStore } from '../stores/auth'; // Importa tu store

const router = createRouter({
  history: createWebHistory(),
  routes: [
    { path: '/', component: Home },
    { path: '/dashboard', component: Dashboard, meta: {
requiresAuth: true } },
    { path: '/login', component: Login }
  ]
});

router.beforeEach((to, from, next) => {
  const authStore = useAuthStore(); // Accede al store
  if (to.meta.requiresAuth && !authStore.isAuthenticated) {
    next('/login'); // Redirige si no está autenticado
  } else {
    next(); // Continúa la navegación
  }
});

export default router;
```

De esta forma, el control de acceso está desacoplado de los componentes de vista y se gestiona globalmente a través del router y el store.

2. Persistencia de Estado entre Rutas

Una de las grandes ventajas de Pinia es que su estado persiste mientras la aplicación está viva. Al navegar entre rutas, el estado de tus stores se mantiene, lo que es crucial para datos como la información del usuario autenticado, configuraciones o datos de un carrito de compras. Esto evita tener que volver a cargar o recalcular datos cada vez que el usuario cambia de vista, mejorando la experiencia y el rendimiento.

Para asegurar la persistencia incluso al recargar la página, considera usar un plugin de persistencia para Pinia que guarde el estado en el almacenamiento local.

3. Mejores Prácticas

- **Modularización:** Organiza tus stores por módulos (ej. `auth.js`, `products.js`) para mantener la claridad.
- **Reactividad:** Utiliza `storeToRefs` cuando desestructuras el store en los componentes para mantener la reactividad de las propiedades.
- **Guards específicos:** Crea guards de navegación específicos para lógicas complejas (ej. `isAuthenticatedGuard.js`) y los importas en el router.
- **Carga de datos:** Puedes usar las acciones de Pinia en los guards para cargar datos necesarios para una ruta antes de que se renderice.

Ejemplo Práctico: Sistema de Autenticación Completo

A continuación, detallaremos la implementación de un sistema de autenticación robusto, integrando el manejo de estado de Pinia con la gestión de rutas de Vue Router. Este ejemplo cubre la definición de un store de autenticación, la protección de rutas mediante guards y la interacción del usuario a través de componentes de login y logout.

1

1. Store de Autenticación (`stores/auth.js`)

Este store gestiona el estado de autenticación del usuario (`isAuthenticated`) y los datos del usuario (`user`). Incluye acciones para `login` y `logout`, y utiliza `localStorage` para una persistencia básica entre sesiones.

```
// src/stores/auth.js
import { defineStore } from 'pinia';
import router from '../router'; // Importar el router para redirección

export const useAuthStore = defineStore('auth', {
  state: () => ({
    isAuthenticated: localStorage.getItem('isAuthenticated') === 'true',
    user: JSON.parse(localStorage.getItem('user')) || null,
  }),
  actions: {
    async login(username, password) {
      // Simular una llamada a API para autenticación
      if (username === 'user' && password === 'password') {
        this.isAuthenticated = true;
        this.user = { name: username, email: 'user@example.com' };
        localStorage.setItem('isAuthenticated', 'true');
        localStorage.setItem('user', JSON.stringify(this.user));
        router.push('/dashboard'); // Redirigir al dashboard tras login
        return true;
      }
      return false; // Autenticación fallida
    },
    logout() {
      this.isAuthenticated = false;
      this.user = null;
      localStorage.removeItem('isAuthenticated');
      localStorage.removeItem('user');
      router.push('/login'); // Redirigir a la página de login
    },
  },
});
```

2

2. Configuración de Vue Router (`router/index.js`)

Definimos las rutas de la aplicación, marcando las rutas protegidas con `meta: { requiresAuth: true }`. Un guard de navegación global `beforeEach` utiliza el store de autenticación para verificar el acceso y redirigir si el usuario no está autenticado.

```
// src/router/index.js
import { createRouter, createWebHistory } from 'vue-router';
import { useAuthStore } from '../stores/auth';
import HomeView from '../views/HomeView.vue';
import DashboardView from '../views/DashboardView.vue';
import LoginView from '../views/LoginView.vue';

const router = createRouter({
  history: createWebHistory(),
  routes: [
    { path: '/', name: 'Home', component: HomeView },
    { path: '/login', name: 'Login', component: LoginView },
    { path: '/dashboard', name: 'Dashboard', component: DashboardView, meta: { requiresAuth: true } },
    // Redirección para rutas no encontradas
    { path: '/:catchAll(.*)', redirect: '/' },
  ],
});

router.beforeEach((to, from, next) => {
  const authStore = useAuthStore();
  if (to.meta.requiresAuth && !authStore.isAuthenticated) {
    next('/login'); // Redirige a login si la ruta requiere autenticación y no está logeado
  } else if (to.name === 'Login' && authStore.isAuthenticated) {
    next('/dashboard'); // Redirige a dashboard si intenta ir a login estando ya autenticado
  } else {
    next(); // Continúa la navegación
  }
});

export default router;
```

3

3. Componente de Login (`views/LoginView.vue`)

Este componente provee una interfaz para que el usuario ingrese sus credenciales. Utiliza la acción `login` del store de autenticación para procesar el intento de inicio de sesión.

```
<template>
  <div class="login-container">
    <h2>Iniciar Sesión</h2>
    <form @submit.prevent="handleLogin">
      <div class="form-group">
        <label for="username">Usuario:</label>
        <input type="text" id="username" v-model="username" required />
      </div>
      <div class="form-group">
        <label for="password">Contraseña:</label>
        <input type="password" id="password" v-model="password" required />
      </div>
      <button type="submit">Entrar</button>
      <p v-if="error" class="error-message">{{ error }}</p>
    </form>
  </div>
</template>

<script setup>
import { ref } from 'vue';
import { useAuthStore } from '../stores/auth';

const authStore = useAuthStore();
const username = ref('user');
const password = ref('password');
const error = ref('');

async function handleLogin() {
  error.value = '';
  const success = await authStore.login(username.value, password.value);
  if (!success) {
    error.value = 'Credenciales incorrectas.';
  }
}
</script>
<style scoped>
/* Estilos básicos para el formulario */
.login-container { max-width: 400px; margin: 50px auto; padding: 20px; border: 1px solid #ccc; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); }
.form-group { margin-bottom: 15px; }
label { display: block; margin-bottom: 5px; font-weight: bold; }
input[type="text"], input[type="password"] { width: calc(100% - 10px); padding: 8px; border: 1px solid #ddd; border-radius: 4px; }
button { background-color: #4CAF50; color: white; padding: 10px 15px; border: none; border-radius: 4px; cursor: pointer; font-size: 16px; }
button:hover { background-color: #45a049; }
.error-message { color: red; margin-top: 10px; }
</style>
```

4

4. Componente Protegido (`views/DashboardView.vue`)

Este componente solo es accesible para usuarios autenticados. Muestra información del usuario obtenida del store y permite cerrar la sesión a través de la acción `logout`.

```
<template>
  <div class="dashboard-container">
    <h2>Bienvenido al Dashboard</h2>
    <p v-if="authStore.user">
      Hola, <b>{{ authStore.user.name }}</b> ({{ authStore.user.email }}).
    </p>
    <button @click="authStore.logout()">Cerrar Sesión</button>
  </div>
</template>

<script setup>
import { useAuthStore } from '../stores/auth';

const authStore = useAuthStore();
</script>
<style scoped>
.dashboard-container { text-align: center; margin-top: 50px; }
</style>
```

5

5. Componente Principal de la Aplicación (`App.vue` y `HomeView.vue`)

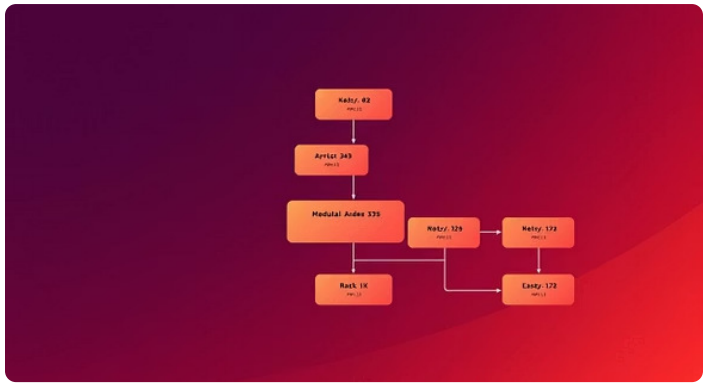
El componente raíz de la aplicación (`App.vue`) utiliza el store de autenticación para mostrar enlaces condicionalmente. `HomeView.vue` es una ruta pública de ejemplo.

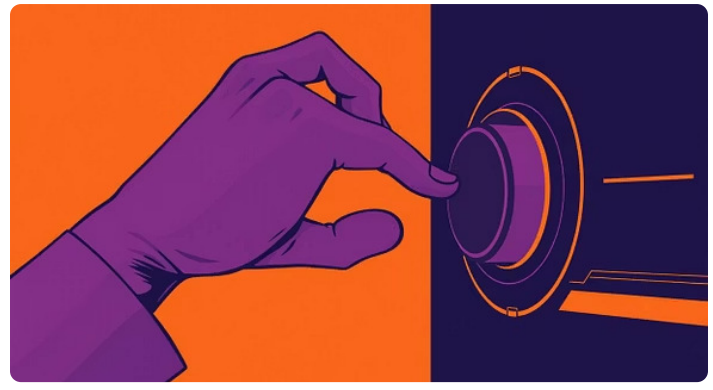
```
<!-- src/App.vue -->
<template>
  <div id="app">
    <nav>
      <router-link to="/">Inicio</router-link> |
      <span v-if="authStore.isAuthenticated">
        <router-link to="/dashboard">Dashboard</router-link> |
        <a @click="authStore.logout()" href="#">Cerrar Sesión</a>
      </span>
      <span v-else>
        <router-link to="/login">Iniciar Sesión</router-link>
      </span>
    </nav>
    <router-view />
  </div>
</template>

<script setup>
import { useAuthStore } from './stores/auth';
const authStore = useAuthStore();
</script>
<style>
#app { font-family: Avenir, Helvetica, Arial, sans-serif; -webkit-font-smoothing: antialiased; -moz-osx-font-smoothing: grayscale; text-align: center; color: #2c3e50; }
nav { padding: 30px; }
nav a { font-weight: bold; color: #2c3e50; margin: 0 10px; }
nav a.router-link-exact-active { color: #42b983; }
</style>
```

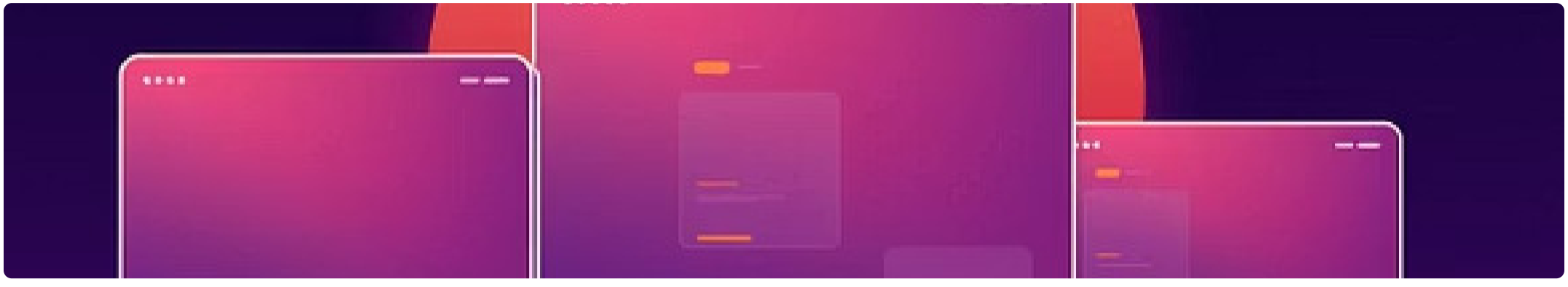
```
<!-- src/views/HomeView.vue -->
<template>
  <div class="home-container">
    <h2>Página de Inicio</h2>
    <p>Contenido accesible para todos los usuarios.</p>
  </div>
</template>
<script setup></script>
<style scoped>
.home-container { text-align: center; margin-top: 50px; }
</style>
```

Con esta implementación, tienes un sistema de autenticación básico pero funcional, demostrando cómo Pinia y Vue Router trabajan en conjunto para gestionar el estado de usuario y la navegación.

Beneficios de Aplicar Estos Conceptos en Vue.js







Código Modular y Mantenible

Al dividir la UI en componentes, tu código se vuelve más organizado, fácil de entender y de mantener. Cada componente es una unidad autónoma.

Comunicación Clara

El flujo de datos unidireccional (props) y la comunicación ascendente (eventos) establecen patrones claros y predecibles para el intercambio de información entre componentes.

Control Preciso

Los hooks del ciclo de vida te dan un control granular sobre cuándo y cómo se ejecuta la lógica en tu aplicación, desde la carga inicial hasta la limpieza final.

Experiencia de Usuario Superior

La reactividad inherente de Vue.js, potenciada por estos conceptos, permite que las interfaces de usuario respondan instantáneamente a los cambios, creando una experiencia fluida y dinámica para el usuario.

Navegación Fluida

Vue Router permite construir Single Page Applications (SPAs) donde las transiciones entre vistas son instantáneas, sin recargas completas de página, mejorando la velocidad y la UX.

Estado Centralizado

Herramientas como Pinia facilitan la gestión de un estado global compartido, simplificando el manejo de datos que necesitan ser accesibles por múltiples componentes en la aplicación.

Conclusión y Próximos Pasos

> **"Dominar los props, eventos, el ciclo de vida, Vue Router y Pinia es la piedra angular para construir aplicaciones Vue.js eficientes, escalables y con un mantenimiento sencillo."**

Esperamos que esta presentación les haya proporcionado una base sólida para comprender estos cinco pilares que forman la base completa para aplicaciones Vue.js modernas.

¡Gracias por su atención!

¿Preguntas y Discusión?

Gabriel Sánchez, Jhan Carlos Zamora, David Ruiz y Jhon Neiver Valencia