# Thư viện Pinia : Pinia là một thư viện quản lý trạng thái (state *management) dựa trên Vue.js. Nó được tạo ra để giải quyết các vấn đề* liên quan đến quản lý trạng thái trong các ứng dụng Vue lớn và phức tạp.

## createPinia : Gắn Pinia vào ứng dụng

import { createPinia } from 'pinia'

import { createApp } from 'vue'

import App from './App.vue'

const app = createApp(App)

app.use(createPinia()) // add pinia

app.mount('#app')

* defineStore : Định nghĩa store trong vue sử dụng Pinia
* Tên store
* State (trạng thái) : Bạn cần định nghĩa trạng thái ban đầu của store.
* Actions (hành động): Bạn có thể định nghĩa các hành động để thực hiện các thay đổi trạng thái trong store

export const useTodoStore = defineStore('todo', {

  state: () => ({

    todos: []

  }),

  actions: {

    addTodo(todo) {

      this.todos.push(todo)

    },

    removeTodo(index) {

      this.todos.splice(index, 1)

    }

  }

})

* getters in defineStore : xác định các hàm truy vấn (getters) cho các trạng thái trong store.

state : () =>({}),

getters: {

    //c1

    completed: (state) => state.tasks.filter((task) => task.is\_completed),

    //c2

    uncompleted() {

      return this.tasks.filter((task) => !task.is\_completed)

    },

    uncompletedCount() {

      return this.uncompleted.length

    }

  }

* actions in defineStore : để xác định các hành động (actions) cho việc thay đổi trạng thái trong store..

state : () =>({}),

getters: {},

 actions: {

    async fetchAllTasks() {

      const { data } = await allTask()

      this.tasks = data.data

    }

* getters vs actions
  + getters
    - Mục tiêu: Getters là các hàm truy vấn được định nghĩa trong store để tính toán các giá trị dựa trên các trạng thái hiện có và trả về kết quả
    - Sử dụng: Khi bạn muốn tính toán và truy xuất dữ liệu từ trạng thái hiện có, bạn nên sử dụng getters
  + Actions
    - Mục tiêu: Actions là các hàm được định nghĩa trong store để thực hiện các thay đổi trạng thái hoặc thực hiện các tác vụ bất đồng bộ
    - Sử dụng: Khi bạn muốn thay đổi trạng thái của store hoặc thực hiện các tác vụ phức tạ

## Khai báo và sử dụng

* + - * Khai báo
* const store = useTaskStore()
  + - * Sử dụng
    - \*C1 : Truy xuất từ export
      * Sử dụng giá trị trả về như các hàm thông thường trong JavaScript

//truy xuất action trong store

const { fetchAllTasks } = store

* + - \*C2 : storeToRefs trong Pinia -> là một hàm tiện ích được cung cấp để giúp bạn trích xuất tất cả các trạng thái (state) từ một store Pinia và chuyển chúng thành các ref trong Vue.
      * Sử dụng giá trị trả về trong cú pháp của Vue như v-for, v-if, và các đối tượng reactive khác trong Vue

const { task } = storeToRefs(store)

* + - store.$patch : là một phương thức dành cho store để cập nhật một số thuộc tính (properties) trong trạng thái của store một cách mở rộng

store.$patch({

  task: {

    name: 'First task update use $patch',

    is\_completed: true

  }

})

# vue-router : yarn add vue-router

* + Khai báo các tuyến

const routes = [

  {

    path: '/tasks',

    component: TasksPage,

    name: 'tasks'

  },

  {

    path: '/login',

    component: LoginPage,

    name: 'login'

  },

  {

    path: '/register',

    component: RegisterPage,

    name: 'register'

  }

]

* + Tạo các tuyến

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

import routes from './routes'

const router = createRouter({

  routes,

  history: createWebHistory()

})

export default router

* + Áp dụng

import router from './router' // vue-route

const app = createApp(App)

app.use(router) // vue-route

<template>

  <Navbar />

  <router-view />

  <Footer />

</template>

* Navigation
  + C1

<router-link to="/'tasks' " > Tasks </router-link>

* + C2

<router-link :to="{ name: 'tasks' }" >Tasks</router-link>

name đã khai báo ở trên

* + C3

<a href="#" @click.prevent="($event) => $router.push('/tasks')" >Tasks</a

* activeLink with router-link
  + c1

const router = createRouter({

  routes,

  history: createWebHistory(),

  linkActiveClass: 'active'

})

* + c2 : css class : router-link-active

<style scoped>

.nav-link.router-link-active {

  color: red;

}

</style>

* router.beforeEach() : đăng kí middleware cho router sẽ thực thi trước khi chuyển đến route mới

router.beforeEach((to, from) => {

  console.log('global befor each', to, from)

  if (to.path === '/tasks') {

    return { name: 'login' }

  }

})

* Hay dùng để checklogin hay chưa
* NotFound

{

    path: '/:notFound(.\*)',

    component: NotFoundPage,

    name: 'error.404'

  }

* + {{ $route.params.notFound }} : get params
* useRouter()

const router = useRouter()

* router.push('/tasks')
* trainsition : hiệu ứng chuyển route

 <router-link v-slot="{ Component }">

    <transition name="fade" mode="out-in">

      <Component :is="Component" />

    </transition>

  </router-link>

* + v-slot : nhận dữ liệu từ component
  + Thẻ transition : thẻ để tạo hiệu ứng
  + :is : sử dụng để chọn component sẽ được hiển thị dựa vào giá trị của biến Component

# Định nghĩa

* Reactivity: là khả năng tự động cập nhật và theo dõi sự thay đổi của các dữ liệu trong một ứng dụng, giúp đồng bộ hóa trạng thái hiển thị với trạng thái dữ liệu thay đổi
* \*ref: Là một hàm util dùng để tạo một biến reactivity. Biến reactivity có khả năng tự động cập nhật và theo dõi sự thay đổi của giá trị nó chứa

const myVar = ref(initialValue)

// Thay đổi giá trị của biến reactivity

count.value = 1;

* \*reactive: là một hàm util dùng để tạo một đối tượng (object)

const person = reactive({

  name: 'John',

  age: 30

});

* Single-File Components (SFC) : cho phép bạn viết mã HTML, CSS và JavaScript liên quan đến một thành phần duy nhất trong một tệp duy nhất
* Bindings : Kết nối dữ liệu với giao diện cú pháp
  + :class, :style

:class="{ active: isActive, 'text-danger': hasError }"

<div :style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>

* Conditional Rendering
  + v-if , v-else, v-else-if
  + v-show
  + v-show : ẩn hiện vẫn chiếm không gian
  + v-if : ẩn hiện k chiếm không gian
* List Rendering
  + v-for (arr)

<li v-for="(item, index) of items"></li>

* + v-for (obj)

<li v-for="value in items"></li>

* + v-for (Component)

<MyComponent

   v-for="(item, index) in items"

   :item="item"

   :index="index"

   :key="item.id"

/>

* Array Change Detection
  + push() : thêm 1 or nhìu p.tử vào cúi mảng trả về length
  + unshift() : thêm 1 or nhìu p.tử vào đầu …
  + pop() : Loại bỏ phần tử cuối mảng và trả về p.tử đó
  + shift() : Loại bỏ phần tử đầu ….
  + splice() : loại bỏ , thêm , thay thế

array.splice(start, deleteCount, item1, item2, ...)

* + sort() : sắp xếp
  + reverse() : đảo
* Event Modifiers
  + .stop : ngăn chặn lan truyền sự kiện

<div @click="outerClick">

  <div @click.stop="innerClick">Inner Div</div>

</div>

* Kích vào “Inner Div” ko lan truyền tới thằng cha . Nếu k có thì hàm “outerClick” vẫn đc gọi.
  + .prevent : ngăn chặn tải lại trang khi nhấn vào liên kết
  + .capture : xử lý theo cơ chế capture thay vì bubbling ( gọi thằng cha trước )

<div @click.capture="parentClick">

  <div @click="childClick">Child Div</div>

</div>

* Kích vào “childClick” thì sẽ xử lý sự kiện “parentClick” trước xong mới xuống sk “childClick”
  + . self : chỉ xử lý khi click đúng phần tử đó
  + .once : Chỉ kích hoạt 1 lần duy nhất , xong sẽ bị gỡ bỏ
  + .passive : Không cần gọi sự kiện “event.preventDefault()” trong hàm xử lý ( ngăn chặn các hành vi mặc định của trang web . vd : tải lại trang khi dùng thẻ a ) . Thường dùng trong các sự kiện @scroll, @touchmove, @wheel
* Key Modifiers
* .enter: Xử lý sự kiện khi nhấn enter
* .delete hoặc .backspace
* . esc
* .space
* .up, .down, .left, .right

Keydown : nhấn

Keyup : nhả

* System Modifier Keys
  + Nhấn cùng với sự kiện khác
    - .ctrl
    - .alt
    - .shift

<input @keyup.alt.enter="clear" />

* Lyfecycel Hook
* onBeforeCreate(() => {

console.log('beforeCreate: Component sắp được tạo.');

});

* onCreated(() => {

console.log('created: Component đã được tạo.');

});

* onBeforeMount(() => {

console.log('beforeMount: Component sắp được gắn kết.');

});

* \*onMounted(() => {

console.log('mounted: Component đã được gắn kết.');

});

* onBeforeUpdate(() => {

console.log('beforeUpdate: Component sắp được cập nhật.');

});

* onUpdated(() => {

console.log('updated: Component đã được cập nhật.');

});

* onBeforeUnmount(() => {

console.log('beforeUnmount: Component sắp bị huỷ.');

});

* \*onUnmounted(() => {

console.log('unmounted: Component đã bị huỷ.');

});

# Cân thiết

* computed: Tính toán lại khi dữ liệu reactivity thay đổi

const compoletedTasks = computed(() => (props.item.is\_completed ? 'completed' : ''))

* defineProps : Định nghĩa props khi truyền thì cha sang

const props = defineProps({

  item: Object,

  value: String

})

* watchEffect: Chạy khi mỗi lần dữ liệu reactivity bên trong thay đổi

watchEffect(() => {

  console.log('Item:', props.item)

})

* Watchers: theo dõi sự thay đổi sự thay đổi của biến dữ liệu

<script setup>

import { ref, watch } from 'vue';

const message = ref('Hello Vue!');

watch(message, (newVal, oldVal) => {

  console.log(`New: ${newVal}, Old: ${oldVal}`);

});

</script>

* onMounted: Chạy sau khi component mouted

onMounted(async () => {

  const { data } = await allTask() // gọi API

  tasks.value = data.data

})

* Provide / Inject: Chia sẽ dữ liệu giữa component cha/con ở xa
  + Component cha

<script setup>

import { provide } from 'vue';

// Tạo một biến dữ liệu và cung cấp cho component con

const sharedData = 'This is shared data from the parent';

provide('sharedData', sharedData);

</script>

* Provide(“key”,value)
  + Component con

<script setup>

import { inject } from 'vue';

// Inject dữ liệu từ component cha

const sharedData = inject('sharedData');

</script>

* ịnect(“key”)
* :key: là một thuộc tính đặc biệt được sử dụng để giúp Vue xác định các phần tử trong một danh sách hoặc vòng lặp khi chúng được cập nhật hoặc tái render.
* \*defineEmits: khai báo sự kiện của thành phần con và thành phần cha lắng nghe
  + component con

const emit = defineEmits(['added']) // khai báo

const addNewTask = (event) => {

  if (event.target.value.trim()) {

    newTask.name = event.target.value

    event.target.value = ''

    emit('added', newTask) // phát ra sự kiện added payload NewTask

  }

}

* + component Cha

 <NewTask @added="handleAddTask" />

* directive
  + vFocus : thực thi sau khi phần tử đó đã được gắn kết vào cây DOM

const vFocus = {

mounted: (el) => el.focus()

}

* KeepAlive : dữ liệu k thay đổi mỗi khi component cha render
  + :is -> định nghĩa component

  <!-- Inactive components will be cached! -->

<KeepAlive>

  <component :is="activeComponent" />

</KeepAlive>

* Giúp tối ưu performance
* Teleport: Chuyển phần tử (element) từ một vị trí DOM hiện tại đến một vị trí DOM khác mà không làm mất kết cấu của ứng dụng.

  <teleport to="body">

        <h2>Modal</h2>

   </teleport>

* Suspense: xử lý các tình huống liên quan đến việc tải dữ liệu không đồng bộ
* Để sử dụng <suspense>, bạn cung cấp hai template:
* default: Template này chứa component hoặc phần nội dung mà bạn muốn hiển thị khi dữ liệu đã được tải.
* fallback: Template này chứa nội dung hoặc component thay thế sẽ được hiển thị trong quá trình dữ liệu đang được tải.

   <suspense>

      <template #default>

        <AsyncComponent />

      </template>

      <template #fallback>

        <LoadingIndicator />

      </template>

    </suspense>

# NuxtJS

+ npx nuxi@latest init <project-name> -> yarn install

+ yarn add sass

+ eslint vs prettien

+ yarn add --save-dev uid sass @iconify/vue