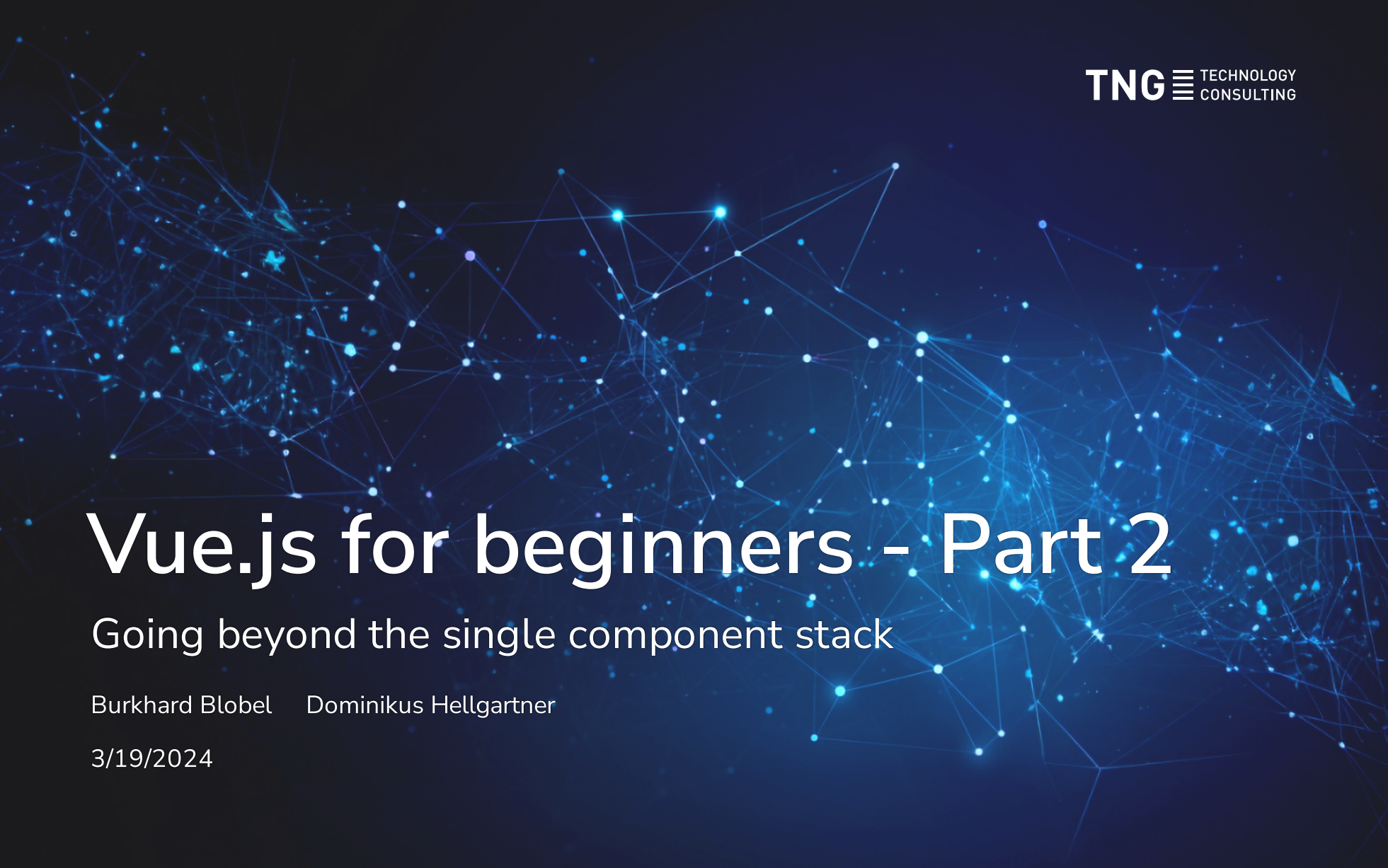


Vue.js for beginners - Part 2

Going beyond the single component stack

Burkhard Blobel Dominikus Hellgartner

3/19/2024

Development setup

- Clone the repository

```
git clone https://github.com/Hellgartner/js-days-munich-vue-advanced-workshop-2024.git
```

- Prepare the backend

```
cd backend
npm ci
npm run start:dev
```

  Now http://localhost:3000/api should provide a swagger representation of the backend api.

- Prepare the frontend

```
cd frontend
npm ci
npm run dev
```

  Now the example application should be available at http://localhost:5173/

Table of contents

1. Introduction
2. Testing
3. Fetching backend data
4. Composables
5. Routing
6. State management
7. Summary

1 Introduction

Moving out of the vue-core part now!

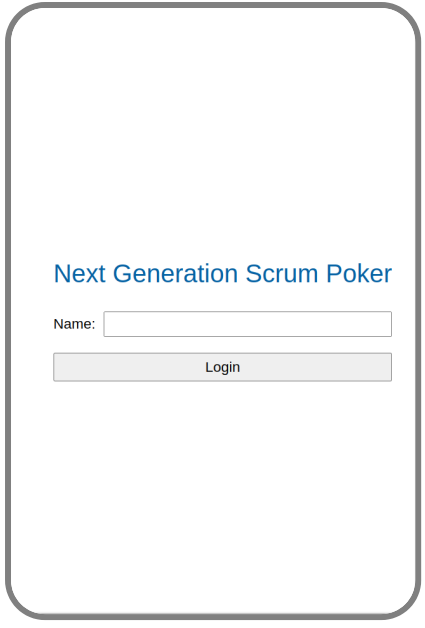
Additional libraries required

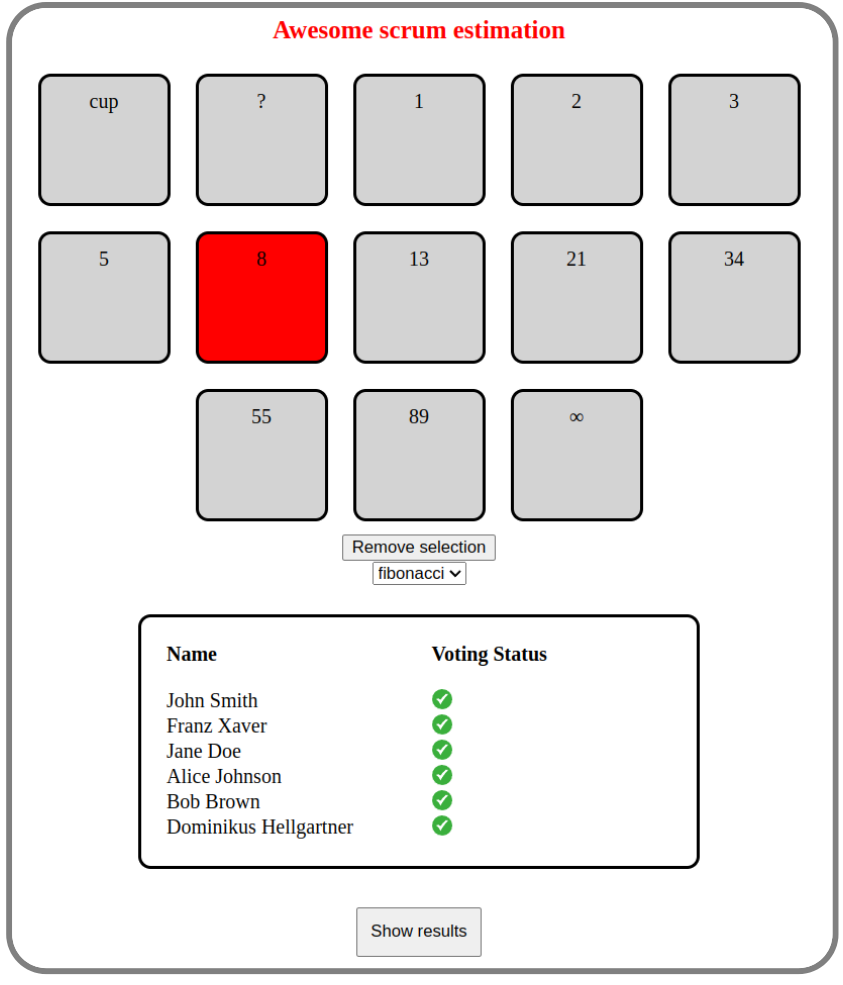
Set-up is not part of this workshop

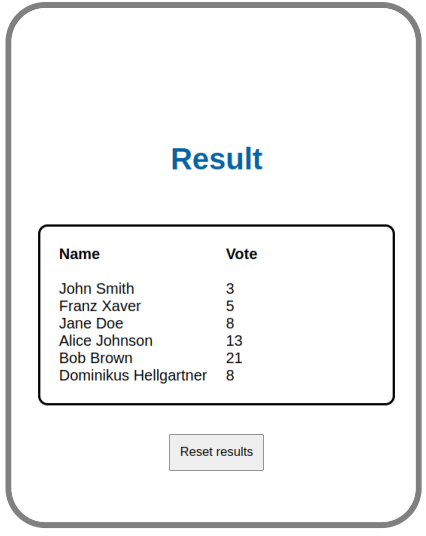
This workshop will be way more opinionated than the beginners part!

It needs to be otherwise the number of variants to discuss is too large.

What are we going to build today

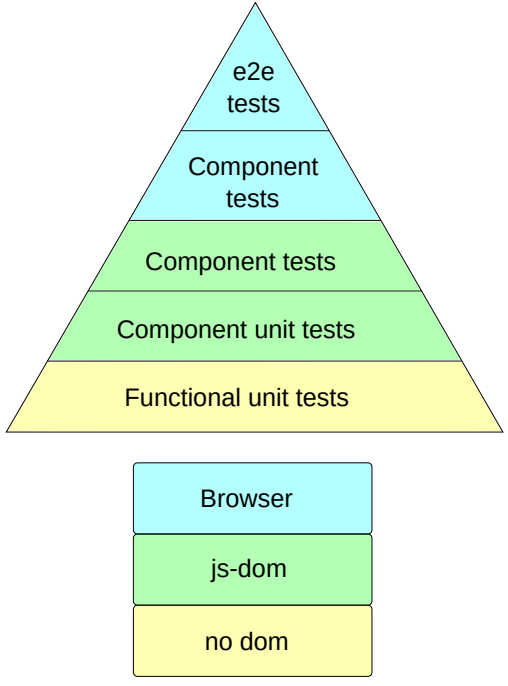






2 | Testing

Terminology



Browser based tests

TNG TECHNOLOGY CONSULTING

Choose your favorite testing framework

- Playwright
- Cypress
- Web Test Runner
- Selenium
- ...

(Nearly) Independent of the stack used

Functional unit tests

TNG TECHNOLOGY CONSULTING

Equivalent to testing any logic in a javascript file.

Use any combination of test runner and assertion library you want.

Tests based on a javascript dom

TNG TECHNOLOGY CONSULTING

What you can do in a javascript dom

- Set up a vue component (including children)
- Click on parts of a vue component
- Watch the dom change (asynchronously)
- Use web apis (like fetch)
- Check all attributes of the generated DOM (including classes)

What you cannot do in a javascript dom

A javascript dom does not really render the component, it "only" handles the DOM. This means the CSS is not evaluated. Thus, you cannot

- Check the styling of an element (color, size, ...)
- Do screenshot testing
- Some parts of a11y testing (contrast!)
- Navigation

Testing framework

TNG TECHNOLOGY CONSULTING

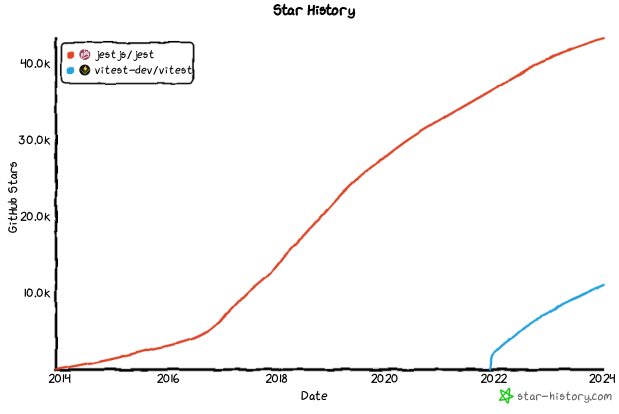
Jest

- Most well known framework
- Wide range of tools available
- Build does not use vite → Separate test build required

Vitest

- Relatively new (2022)
- Designed to be compatible with jest
- Uses vite for building
  - Fast
  - Test build corresponds to prod build

Github stars



What else is required

- Vue Test Utils: To enable testing vue components

  > Vue Test Utils (VTU) is a set of utility functions aimed to simplify testing Vue.js components. It provides some methods to mount and interact with Vue components in an isolated manner.

- Vue Testing Library: For more comfort

  > Simple and complete Vue.js testing utilities that encourage good testing practices.

  > Vue Testing Library is a lightweight adapter built on top of DOM Testing Library and @vue/test-utils.

- jest-dom: For better assertions

  > The @testing-library/jest-dom library provides a set of custom jest matchers that you can use to extend jest. These will make your tests more declarative, clear to read and to maintain.

  (Also works with vitest)

A basic test

TNG TECHNOLOGY CONSULTING

Use `render` to render a vue component

```
1  import { describe, expect, it } from 'vitest'
2  import {fireEvent, render} from '@testing-library/vue'
3  import MyComponent from "../MyComponent.vue";
4
5  describe('MyComponent', () => {
6      it('renders', () => {
7          const {container} = render(MyComponent)
8
9          expect(container).not.toBeNull()
10     })
11 })
```

Use props

Props can be passed as additional arguments to `render`

```
1
2   describe('MyComponent', () => {
3       it('has the correct title', () => {
4           const {container} = render(MyComponent, {
5               props: {
6                   value: "some test value",
7                   someOtherValue: "some test value"
8               }
9           })
10
11          const title =  getByTestId<HTMLElement>(container as HTMLElement, 'title')
12          expect(title).toHaveTextContent("some test value")
13      })
14  })
```

Check for emitted events

TNG TECHNOLOGY CONSULTING

Check for emitted events using the `emitted` function

```
1
2   describe('MyComponent', () => {
3       it('Clicking the title emits tile-clicked event', async () => {
4           const {container, emitted} = render(MyComponent)
5
6           const title =  getByTestId<HTMLElement>(container as HTMLElement, 'title')
7           await fireEvent.click(title);
8
9           const expectedEvent = emitted<string[]>('tile-clicked');
10          expect(expectedEvent).toBeTruthy();
11      })
12  })
```

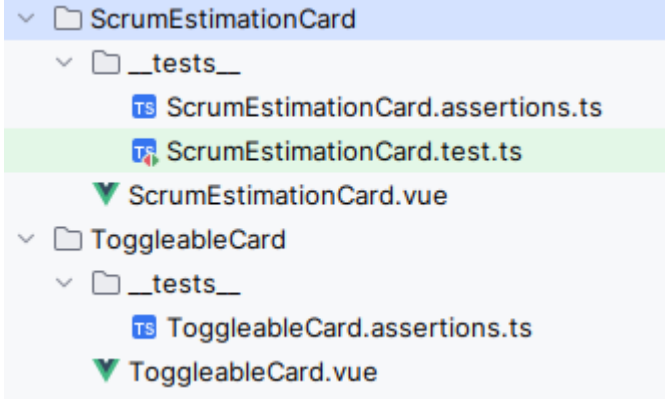
Rendering a component

Calling render creates the DOM for the component including *all child components*!

- Advantage: Extracting sub-components does not require test changes
- Disadvantage: Tests might depend on details of child components:

```
const title =  getByTestId<HTMLElement>(container as HTMLElement, 'title-of-the-10th-child')
```

- Recommendation: Colocate all asserts for a component with the component's tests
  - If the component changes just update the associated assertions file
  - Side effect: Test get way more readable



If you want to use mocked child components, use shallow mount of vue testing library directly

Exercise 1

3

Fetching backend data

The component lifecycle

Data

TNG TECHNOLOGY CONSULTING

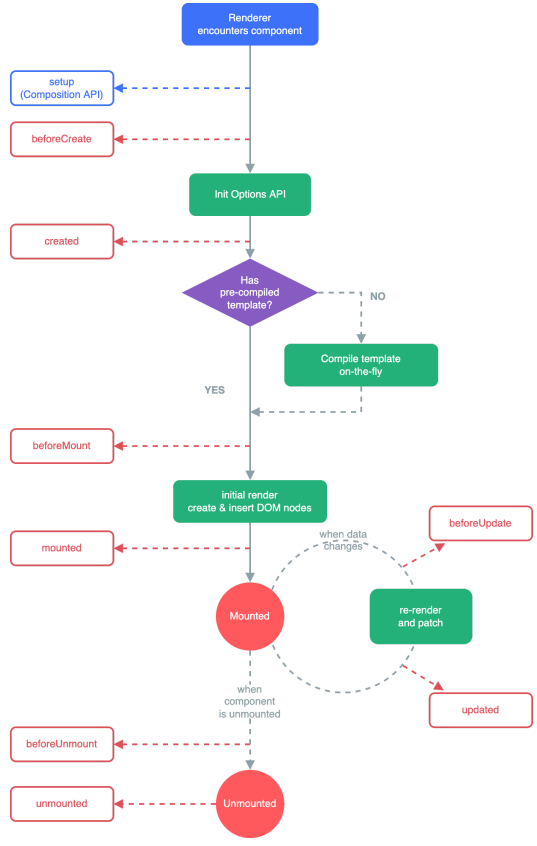
When to fetch data

- When the user clicks a button

```
const clickHandler = async () => {
   const result = await fetch('https://someHost:somePort/some/path')
}
```

- Much more common: On page/component load
- Variant: Polling

The component lifecycle



source: Vue documentation

How to trigger actions on lifecycle events

TNG TECHNOLOGY CONSULTING

- Just implement the corresponding hook `script` part

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

onMounted(() => {
  console.log(`Let's go`)
})
</script>
```

Testing components with backend calls

TNG TECHNOLOGY CONSULTING

Mock the fetch calls

- msw (used for the exercises)
- nock
- Mirage
- fetch-mock
- ...

In more real world scenarios, typically some middleware is used. Often, there are dedicated mocking libraries available:

- axios --> axios-mock-adapter
- vue apollo --> use local resolver

Depending on your testing scenario, creating a wrapper which then can be mocked is also an option.

Testing Pitfall: Async calls

TNG TECHNOLOGY CONSULTING

```
1   it('does a get request on mounting', async () => {
2     let hasBeenCalled = false;
3     server.use(http.get('http://localhost:3000/foo/bar', () => {
4         hasBeenCalled = true;
5         return new HttpResponse()
6     }))
7     render(MyComponent)
8
9     expect(hasBeenCalled).toBeTruthy()
10  })
```

This fails because the async server response is only executed after the test terminates. *flushPromises()* (vue test utils) to the rescue:

```
1   it('does a get request on mounting', async () => {
2     let hasBeenCalled = false;
3     server.use(http.get('http://localhost:3000/foo/bar', () => {
4         hasBeenCalled = true;
5         return new HttpResponse()
6     }))
7     render(MyComponent)
8
9     await flushPromises()
10
11    expect(hasBeenCalled).toBeTruthy()
12  })
```

Exercise 2

4 | Composables

In the context of Vue applications, a "composable" is a function that leverages Vue's Composition API to encapsulate and reuse **stateful logic**.

Vue docs

Example: Smart meter display

TNG TECHNOLOGY CONSULTING

```vue
<template>
  <div>Current power consumption: {{currentPower}} W</div>
</template>

<script setup>
  //includes
  const currentPower = ref(0.0)
  async function fetchPower() {
      const fetchResult = await fetch("http://someIp:somePort/smartHome/devices/1234567/status")
      currentPower.value = (await fetchResult.json()).currentPower
  }

  let intervalId: number|undefined = undefined;
  onMounted(async () =>  intervalId = setInterval(fetchPower, 1000) as unknown as number)

  onUnmounted(() => {
    if(intervalId) {
      clearInterval(intervalId)
    }
  })
</script>
```

Example: Shortcomings

TNG TECHNOLOGY CONSULTING

- Not feature complete:
  - Error handling
  - Loading status
  - Props for setting the device id
- Logic already now not trivial
- Hard to read: This only gets the current value of `currentPower`
- We will need the same logic on other places
  - Visual display of the power
  - Overview pages
  - ...

How to reuse this logic?

Composable to the rescue: Definition

TNG TECHNOLOGY CONSULTING

```
1   const usePowerBackend = (deviceId:String) => {
2       const currentPower = ref(0.0)
3       async function fetchPower() {
4               const fetchResult = await fetch(`http://someIp:somePort/smartHome/devices/${deviceId}/status`)
5               currentPower.value = (await fetchResult.json()).currentPower
6       }
7
8       let intervalId: number|undefined = undefined;
9
10      onMounted( async () => intervalId = setInterval(fetchPower, 1000) as unknown as number)
11
12      onUnmounted(() => {
13          if(intervalId) {
14              clearInterval(intervalId)
15          }
16      })
17      return { currentPower }
18  }
19
20  export default usePowerBackend;
```

Composable to the rescue: Usage

TNG TECHNOLOGY CONSULTING

```vue
<template>
  <div>Current power consumption: {{currentPower}} W</div>
</template>

<script setup>
  //includes
  const {currentPower} = usePowerBackend("1233567")
</script>
```

Achievements:

- Easier to read ✓
- Reusable ✓
- Better information hiding ✓
- Can be tested in isolation ✓

This is the primary mechanism used by today’s vue ecosystem libraries

Conventions

- Name:
  - camelCase
  - starts with "use"
- Can have arbitrary arguments (⚠ Take care about reactivity)
- Returns an object whose values are Refs

Exercise 3

(Optional)

5 | Routing

Routing

```
https://www.myhomepage.com/next/page?param1=A&param2=B
\___/   \_______________/\_______/ \_____________/
  |             |           |             |
Schema         Host        Path         Query
```

In a Multi-Page Application (MPA):

- server-side routing:
  - server will make sense of path and query parameters
  - server will answer with HTML content

In a Single-Page Application (SPA)

- client-side routing:
  - client application will make sense of path and query parameters
  - fetches relevant data from backend (if needed)
  - re-renders parts of the current view or renders a different one

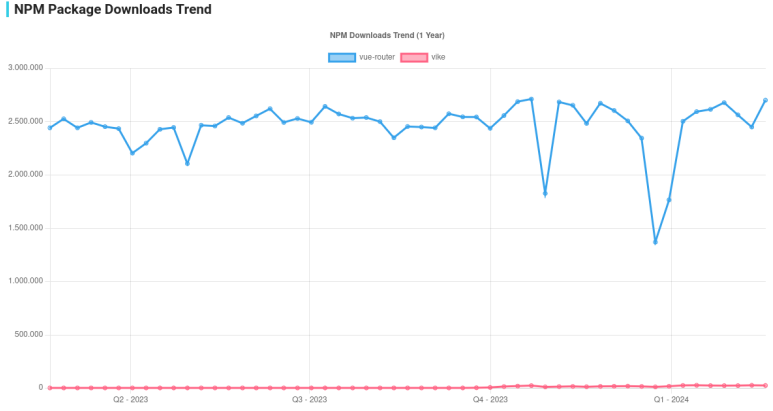
Routing

Vue Router

- the officially-supported and recommended router for Vue (https://router.vuejs.org)
- under the hood used by Nuxt
- works well with Vite

Vike (former vite-plugin-ssr)

- for more complicated interplay between server-side and client-side rendering



Vue Router

TNG TECHNOLOGY CONSULTING

- Setting up Vue Router

```
const routes = [
    {path: '/', component: HomeView},            // route definitions:
    {path: '/about', component: AboutView},      // → which path corresponds to which view
]

const router = createRouter({                    // configure router
    routes: routes,
    history: createWebHistory(),                 // configure history object
})

const app = createApp({})
app.use(router)                                  // use router in Vue app

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

- Displaying the view:

```
<template>
  <Layout>
    <RouterView/>                                <!— specify where the view should be shown —>
  </Layout>
</template>
```

Vue Router

Navigation between views

TNG TECHNOLOGY CONSULTING

Server-Side Routing:

- Template:

```
<template>
  <a href="/about">About</a>
</template>
```

- Script block:

```
<script setup>
  const goToAbout =
      () => window.location.assign("/about")
  const goBack = () => window.history.back()
</script>
```

Client-Side Routing:

- Template:

```
<template>
  <RouterLink to="/about">About</RouterLink>
</template>
```

- Script block:

```
<script setup>
  import {useRouter} from "vue-router"

  const router = useRouter()
  const goToAbout = () => router.push("/about")
  const goBack = () => router.go(-1)
</script>
```

Dynamic Routing

TNG TECHNOLOGY CONSULTING

Static routing

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

```
<template>
  <RouterLink to="/about">About</RouterLink>
</template>
```

Dynamic routing

```
const routes = [{path: '/user/:name', component: UserView}]     // e.g. /user/adam, /user/bob, ...
```

```
<template>
  <div>User {{ route.params.name }}</div>                       <!-- read params from the route object -->
  <div>Age {{ route.query.age }}</div>                          <!-- read query params from the route object -->
</template>

<script setup lang="ts">
  import {useRoute} from "vue-router";
  const route = useRoute();                                     // get the route object
</script>
```

For advanced route matchers, see Vue Router Documentation

Named routes

Route can be named:

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

The following are equivalent:

```
<template>
  <RouterLink to="/user/bob?id=123">User</RouterLink>
  <RouterLink :to="{ path: '/user/bob', query: {id: '123'} }">User</RouterLink>
  <RouterLink :to="{ name: 'user', params: { username: 'bob' }, query: {id: '123'}}">User</RouterLink>
</template>
```

Advantages:

- No hardcoded URLs
- Automatic encoding/ decoding of params

Advanced topics

TNG TECHNOLOGY CONSULTING

- Navigation guards
- Meta fields
- Transitions
- Scroll on page load
- Lazy loading routes
- Navigation Failures
- Creating routes dynamically

Exercise 4

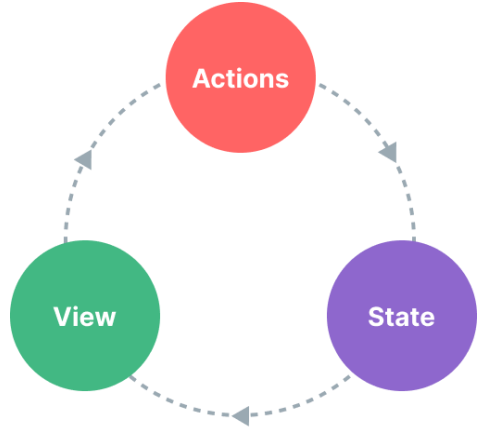
6

State management

State management of a single Vue component

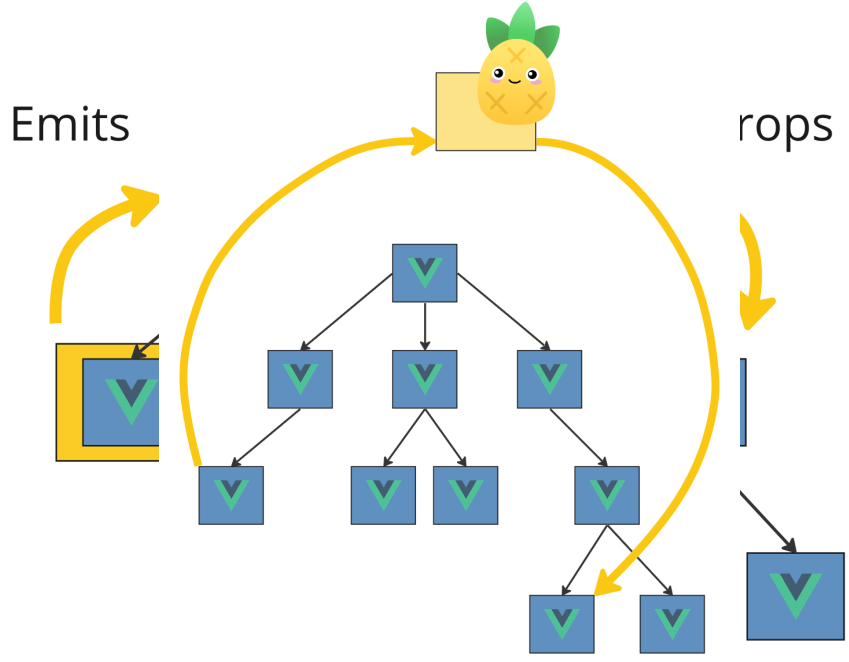
Source: Vue documentation

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

// state
const count = ref(0)

// actions
function increment() {
  count.value++
}
</script>

<!-- view -->
<template>{{ count }}</template>
```



Global state

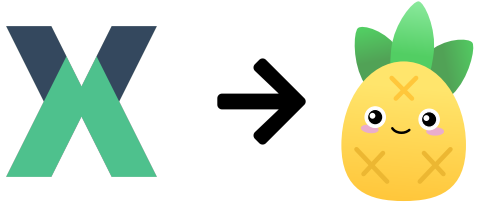
- Multiple components can share a common state
  - Multiple views may depend on the same state
  - Actions from different views may need to change the same state



Official state management library

TNG TECHNOLOGY CONSULTING

- Previous state library was *vuex*
- Official state library since February 2022 is *Pinia*



- Advantages of *Pinia*
  - 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
  - Solid type inference support when used with TypeScript

Some definitions

- Store:
  - Object which carries global state.
  - Exists independently of vue components.
  - Generally contains multiple reactive variables.
- Getter:
  - Function which allows to access a part of the state in the store.
  - Is reactive.
- Action:
  - Function which allows to modify the state associated to the store.
  - Can be asynchronous.

Defining the store

- Defining a store: `defineStore()` from *Pinia*
  - Requires a unique name as first argument.
  - Second Argument is a Setup function (or an Options object).
  - Return an object with the properties and methods.
- Naming convention:
  - Similar to defining composables: returning function in camelCase and starting with "use"

```
1   // in UniqueNameStore.ts
2   import { defineStore } from 'pinia';
3
4   export const useUniqueNameStore = defineStore('uniqueName', () => {
5     // The data of the store is the state: ref(),
6     // the computed properties of the store are the getters: computed(),
7     // and the methods are the actions: function().
8
9     return { }
10  });
```

Using the store

- The store will be created once `use...Store()` is called within a component.
- You can access any state, getters or actions exposed by the return object.
- `store` is an object wrapped with `reactive`
  - no need to write `.value`
  - cannot be destructured, use `storeToRefs()` instead to extract properties from the store while keeping the reactivity
  - actions can directly be destructured

```vue
<script setup>
import { useUniqueNameStore } from '@/stores/UniqueNameStore'

// access the `store` variable anywhere in the component
const store = useUniqueNameStore()

</script>
```

Example: CounterStore

TNG TECHNOLOGY CONSULTING

Defining the store

```ts
// in CounterStore.ts
import { defineStore } from 'pinia';

export const useCounterStore =
        defineStore('counter', () => {
  // The data of the store is the state: ref(),
  const counter = ref(0);
  // the computed properties of the store are the
  // getters: computed(),
  const doubleCounter = computed(
      () => counter.value * 2
  );
  // and the methods are the actions: function().
  const resetCounter = () => {
      counter.value = 0
  }

  return { counter, doubleCounter, resetCounter };
});
```

Using the store

```vue
<!-- in any vue component or composable -->
<script setup>
import { useCounterStore } from '@/stores/CounterStore'
import { storeToRefs } from 'pinia';

const store = useCounterStore();
// access the store
store.doubleCounter;

if (store.counter > 10) {
    store.resetCounter();
}

// destructuring of refs
const { counter, doubleCounter } = storeToRefs(store);
// destructuring of actions
const { resetCounter } = store;

</script>
```

Exercise 5

7

Summary

Summary

- You can do the majority of the testing without ramping up a server
- Use component lifecycle methods to query data on component load and to clean up afterward
- Use composables to structure and reuse your business logic
- Avoid hard page reloads using routing:
  - Define your routes via url and name
  - Handle request and/or query parameter
- Share State between components using Pinia
  - Actions trigger state changes
  - Organize your state into multiple independent stores

TNG TECHNOLOGY CONSULTING

Thank you!

Questions?

Burkhard Blobel

Dominikus Hellgartner