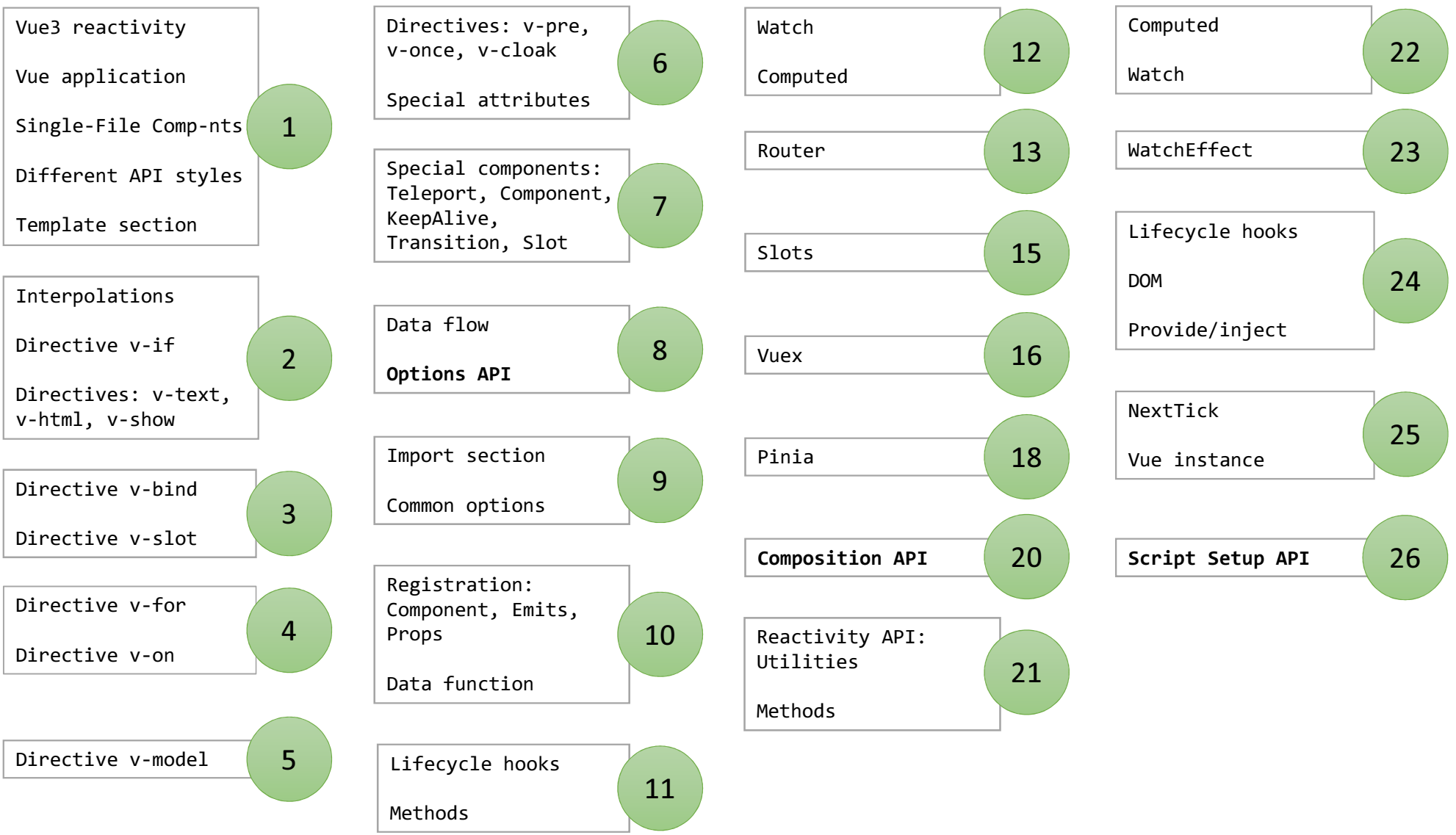

Vue3 reactivity
Vue application
Single-File Comp-nts
Different API styles
Template section
1
Interpolations
Directive v-if
Directives: v-text, v-html, v-show
2
Directive v-bind
Directive v-slot
3
Directive v-for
Directive v-on
4
Directive v-model
5
Directives: v-pre, v-once, v-cloak
Special attributes
6
Special components: Teleport, Component, KeepAlive, Transition, Slot
7
Data flow
Options API
8
Import section
Common options
9
Registration: Component, Emits, Props
Data function
10
Lifecycle hooks
Methods
11
Watch
Computed
12
Router
13
Slots
15
Vuex
16
Pinia
18
Composition API
20
Reactivity API: Utilities
Methods
21
Computed
Watch
22
WatchEffect
23
Lifecycle hooks
DOM
Provide/inject
24
NextTick
Vue instance
25
Script Setup API
26

Vue3 reactivity

```
   /*In Vue 3, data is made reactive by
leveraging JavaScript Proxies.
   The Proxy object allows you to create
an object that can be used in place of
the original object, but which may
redefine fundamental Object operations
like getting, setting, and defining
properties.*/

const target = {
  message1: "hello",
  message2: "everyone"
};

const handler = {
  get(target, prop, receiver) {
    return "reading";
  }
  set(target, prop, receiver) {
    Render();
  }
};

const proxy = new Proxy(target,
handler);

console.log(proxy.message1); // expects
"hello", but it is "reading“

proxy.message1 = 2; // call render
```

Vue application

```
import { createApp } from 'vue'
import plugin from './pugin'

const app = createApp({


  /* root component options */


});

app.use(plugin);

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

Different API styles

```
//1.Options API                          (8)

//2.Composition API                      (20)

//3.Script setup (composition API
option)                                  (26)
```

Single-File Components

```
//There are 3 sections

<template>
  ....
</template>

<script>
  ....
</script>

<style>
  ....
</style>
```

Template section

```
/*Use variables from Data, Computed,
Props, …*/

<template>
  //HTML

  //Interpolations                 (2)

  //Directives                     (2-6)

  //Custom components

  //Special components             (7)

  //Special attributes             (6)
</template>
```

Interpolations

```
<!-- use reactive variables, computed variables, props, methods,
functions to provide data -->
<template>
  <span>Message: {{ msg }}<span>
  <span>{{ formatDate(date) }}<span>

  <!-- JavaScript expressions, only single expressions -->
  <div>
    {{ number + 1 }}
    {{ ok ? 'YES' : 'NO' }}
    {{ message.split('').reverse().join('') }}
  </div>

  <!-- attribute value -->
  <div :id="`list-${id}`"></div>
</template>
```

Directives: v-text, v-html, v-show

```
<template>
  <!-- v-text updates the element's text content -->
  <span v-text="msg"></span>
  <!-- same as -->
  <span>{{msg}}</span>

  <!-- v-text updates the element's innerHTML -->
  <div v-html="html"></div>

  <!-- v-show toggles the element's visibility, does not delete
element from DOM -->
  <div v-show="false">Hidden text</div>
</template>
```

Directive v-if

```
<template>
  <!-- v-if conditionally renders an element -->
  <div v-if="false">Does not render this element</div>

   <!-- v-else denotes the "else block" for v-if or
   a v-if / v-else-if chain -->
  <div v-if="Math.random() > 0.5">
    Now you see me
  </div>
  <div v-else>
    Now you don't
  </div>

  <!-- v-else-if denotes the "else if block" for v-if. Can be
  chained -->
  <div v-if="type === 'A'">
    A
  </div>
  <div v-else-if="type === 'B'">
    B
  </div>
  <div v-else-if="type === 'C'">
    C
  </div>
  <div v-else>
    Not A/B/C
  </div>
</template>
```

Directive v-bind p.1

```
<template>
<!-- v-bind dynamically binds one or more attributes, or a
component prop to an expression -->
  <!-- bind an attribute -->
  <img v-bind:src="imageSrc" />

  <!-- dynamic attribute name -->
  <button v-bind:[key]="value"></button>

  <!-- shorthand -->
  <img :src="imageSrc" />

  <!-- shorthand dynamic attribute name -->
  <button :[key]="value"></button>

  <!-- with inline string concatenation -->
  <img :src="'/path/to/images/' + fileName" />

  <!-- class binding -->
  <div :class="{ red: isRed }"></div>
  <div :class="[classA, classB]"></div>
  <div :class="[classA, { classB: isB, classC: isC }]"></div>

  <!-- style binding -->
  <div :style="{ fontSize: size + 'px' }"></div>
  <div :style="[styleObjectA, styleObjectB]"></div>

  <!-- binding an object of attributes -->
  <div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>
</template>
```

Directive v-bind p.2

```
<template>
  <!-- prop binding. "prop" must be declared in the child
component -->
  <MyComponent :prop="someThing" />

  <!-- pass down parent props in common with a child component
-->
  <MyComponent v-bind="$props" />

  <div :someProperty.prop="someObject"></div>
  <!-- equivalent to -->
  <div .someProperty="someObject"></div>
</template>
```

Directive v-slot

```
<!-- v-slot denotes named slots or slots that expect to receive
props -->
  <template v-slot:header>
    Header content
  </template>

  <!-- named slot that receives props -->
  <InfiniteScroll>
    <template v-slot:item="slotProps">
      <div class="item">
        {{ slotProps.item.text }}
      </div>
    </template>
  </InfiniteScroll>
```

Directive v-for

```
<template>
  <!-- v-for renders the element or template block multiple
  times based on the source data -->
  <div v-for="item in items">
    {{ item.text }}
  </div>
  <div v-for="(item, index) in items" :key="item.id"></div>

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

  <div v-for="(value, name, index) in object"></div>

  <li v-for="item in items">
    <span v-for="childItem in item.children">
      {{ item.message }} {{ childItem }}
    </span>
  </li>

  <!-- v-if has a higher priority than v-for,  move v-for to a
  wrapping <template> -->
  <template v-for="todo in todos">
    <li v-if="!todo.isComplete">
      {{ todo.name }}
    </li>
  </template>
</template>
```

Directive v-on

```
<template>
<!-- v-on attaches an event listener to the element -->
  <!-- method handler -->
  <button v-on:click="doThis"></button>

  <!-- dynamic event -->
  <button v-on:[event]="doThis"></button>

  <!-- inline statement -->
  <button v-on:click="doThat('hello', $event)"></button>

  <!-- shorthand -->
  <button @click="doThis"></button>

  <!-- shorthand dynamic event -->
  <button @[event]="doThis"></button>

  <!-- stop propagation -->
  <button @click.stop="doThis"></button>

  <!-- prevent default -->
  <button @click.prevent="doThis"></button>

  <!-- prevent default without expression -->
  <form @submit.prevent></form>

  <!-- key modifier using keyAlias -->
  <input @keyup.enter="onEnter" />

  <!-- the click event will be triggered at most once -->
  <button v-on:click.once="doThis"></button>
</template>
```

Directive v-model

```
<template>
<!-- v-model creates a two-way binding on a form input element or a component -->
  <input :value="text" @input="event => text = event.target.value" />

  <!-- equivalent to -->
  <input v-model="text">
</template>



<!-- v-model for a custom component -->
<template>
  <CustomInput :modelValue="searchText“  @update:modelValue="newValue => searchText = newValue" />
</template>

  <!-- CustomInput.vue -->
  <script>
    export default {
      props: ['modelValue'],
      emits: ['update:modelValue']
    }
  </script>

  <template>
      <input :value="modelValue“  @input="$emit('update:modelValue', $event.target.value)" />
  </template>
  <!-- CustomInput.vue -->

<!-- equivalent to -->
<CustomInput v-model="searchText" />
```

Directives: v-pre, v-once, v-cloak

```
<template>
  <!-- v-pre skips compilation for this element and all its
children, the most common use case of this is displaying raw
mustache tags -->
  <span v-pre>{{ this will not be compiled }}</span>

  <!-- v-once renders the element and component once only, and
skip future updates -->
  <span v-once>This will never change: {{msg}}</span>

  <!--v-cloak uses to hide un-compiled template until it is
ready Combined with CSS rules such as [v-cloak] { display: none
}, it can be used to hide the raw templates until the component
is ready -->
  [v-cloak] {
    display: none;
  }

  <div v-cloak>
    {{ message }}
  </div>
</template>
```

Special attributes

```
<template>
  <!-- key uses as a hint for Vue's virtual DOM algorithm to
  identify vnodes -->
  <ul>
    <li v-for="item in items" :key="item.id">...</li>
  </ul>

  <!-- ref uses to register a reference to an element or a
  child component -->
  <p ref="p">hello</p>
  <!-- stored as this.$refs.p -->

  <!-- the ref attribute can also be bound to a function, which
  will be called on each component update -->
  <input :ref="(el) => {
      /* assign el to a property or ref */
  }">

  <!-- is uses for binding dynamic components, see also
  KeepAlive component -->
  <component :is="currentTab"></component>

</template>
```

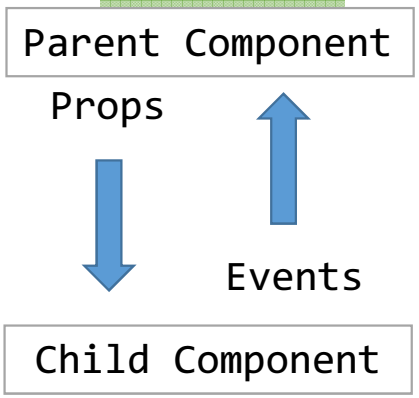
Special components: Teleport, Component

```
<template>
<!-- Teleport, renders its slot content to another part of the
DOM -->
  <Teleport to="body">
    <div v-if="open" class="modal">
      <p>Hello from the modal!</p>
      <button @click="open = false">Close</button>
    </div>
  </Teleport>
</template>


<!-- Component, a "meta component" for rendering dynamic
components or elements -->
<script>
  import Foo from './Foo.vue'
  import Bar from './Bar.vue'

  export default {
    components: { Foo, Bar },
    data() {
      return {
        view: 'Foo'
      }
    }
  }
</script>

<template>
  <component :is="view" />
</template>
```

Special components: KeepAlive, Transition, Slot

```
<template>
  <!-- KeepAlive, caches dynamically toggled components wrapped
  inside, often uses for Tabs -->
  <KeepAlive>
    <component :is="view"></component>
  </KeepAlive>

  <!-- Transition, provides animated transition effects to a
  single element or component -->
  <Transition name="fade" mode="out-in" appear>
    <component :is="view"></component>
  </Transition>

  <!-- TransitionGroup, provides transition effects for
multiple
  elements or components in a list -->
  <TransitionGroup tag="ul" name="slide">
    <li v-for="item in items" :key="item.id">
      {{ item.text }}
    </li>
  </TransitionGroup>

  <!-- Slot, denotes slot content outlets in templates -->
  <FancyButton>
    Click me! <!-- slot content -->
  </FancyButton>
  <!-- The <slot> element is a slot outlet that indicates where
  the parent-provided slot content -->
  <button class="fancy-btn">
    <slot></slot> <!-- slot outlet -->
  </button>
</template>
```

Data flow



Props

```vue
<template>
   <BlogPost title="My journey with Vue" />

   <!-- Dynamically assign the value of a variable -->
   <BlogPost :title="post.title" />

   <!- Static String, static Number, static Boolean -->
   <BlogPost title="User" :num="42" :isFull="true" />
</template>
```

Events

```vue
<template>
  <!--Inside MyComponent -->
  <button @click="$emit('someEvent',param)">click</button>

  <MyComponent @some-event="callback" />
</template>
```

Options API

```vue
<template>
 …
</template>

<script>
  //import                          (9)
  export default {
    //common options                (9)

    //component registration        (10)
    //props registration            (10)
    //emits registration            (10)

    //data function                 (10)

    //lifecycle hooks               (11)
    //methods                       (11)
    //watch                         (12)
    //computed                      (12)

    //component instance features
  }
</script>

<style scoped lang="SCSS">
   /*with scoped attribute  CSS will apply to elements of the current
   component only, declare pre-processor languages using the lang
   attribute*/
  .text {
    color: v-bind(color); /* v-bind directive */
  }
</style>
```

Script

Import section, Option API

```
<script>
  //import Components, @ is an alias to /src
  import MyComp from '@/components/MyComp.vue';
  import ParComponent from '@/components/ParentComponent.vue';

  //mixin objects can contain instance options like normal
  instance objects,
  //and they will be merged to the current component
  import mixin from '@/mixins';
</script>
```

Common options, Option API

```
<script>
  //allows this component to extend another, inheriting its
  component options
  extends: ParentComponent,

  //by Default Component name is equal to the current filename
  name: 'Home',

  // by Default true, when it false - the attributes can be
  explicitly bound to a non-root element using v-bind="$attrs"
  inheritAttrs: false,

  //custom directives
  directives: {
   …
  },
</script>
```

Common options, Option API

```
<script>
  //this object contains the properties that are available for
  injection into its descendants, should be either an object or
  a function that returns an object
  provide: {
    msg: 'Hello!',
  },
  //or
  provide() {
    return {
      msg: this.msg
    }
  },

  //declare properties to inject into the current component by
  locating them from ancestor providers, should be either an
  array of strings, or an object where the keys are the local
  binding name
  inject: ['msg'],
  //or
  inject: {
    localMessage: {from: 'msg'}
  },

  //an array of option objects to be mixed into the current
  component, mixin hooks are called in the order they are
  provided, and called before the component's own hooks
  mixins: [mixin],
</script>
```

Component registration, Option API

```
<script>
//imported components needs to be "registered", use PascalCase
names
  components: {
    MyComp, // shorthand
    RenamedMyComp: MyComp // register under a different name
  },
</script>
```

Props registration, Option API

```
<script>
  //all component props need to be explicitly declared
  //props can be declared in two forms: Simple form using an
  array of strings
  props: ['size', 'myMessage'],
  //or full form using an object
  props: {
    height: Number,  // type check
    age: {           // type check plus other validations
      type: Number,
      default: 0,
      required: true,
      validator: (value) => {
        return value >= 0
      }
    }
  },
</script>
```

Emits registration, Option API

```
<script>
  //declare the custom events emitted by the component
  //simple form using an array of strings
  emits: ['check'],
  //or object with validation
  emits: {
    // no validation
    click: null,
    // with validation
    submit: (payload) => {
      if (payload.email && payload.password) {
        return true
      } else {
        console.warn(`Invalid submit event payload!`)
        return false
      }
    }
  },
</script>
```

Data function, Option API

```
<script>
  //a function that returns the initial reactive state for the
  component instance
  data() {
    return {
      msg: 'Hi', a: 1, b: 2, c: [], d: {e:3}
    }
  },
</script>
```

Lifecycle hooks, Option API

```
<script>
  //called when the instance is initialized
  beforeCreate() {/*...*/},

  //called after the instance has finished processing all state-
  related options
  created() {/*...*/},

  //called right before the component is to be mounted
  beforeMount() {/*...*/},

  //called after the component has been mounted
  mounted() {/*...*/},

  //called right before the component is about to update its DOM
  tree due to a reactive state change
  beforeUpdate() {/*...*/},

  //called after the component has updated its DOM tree due to a
  reactive state change
  updated() {/*...*/},

  //called right before a component instance is to be unmounted
  beforeUnmount() {/*...*/},

  //called after the component has been unmounted
  unmounted() {/*...*/},

  //others: errorCaptured, activated ...
</script>
```

Methods, Option API

```
<script>
  //declare methods, avoid using arrow functions
  methods: {
     plus() {
       this.a++
     },

     anotherMethod() {
        //emit events
        this.$emit('foo')

        //with additional arguments
        this.$emit('bar', 1, 2, 3)
     },

     //if you ever want to execute a function after the DOM has
     been updated
     loadHTMLFromServer() {
        this.$nextTick(() => {/*...*/})
     }

   },
</script>
```

Watch, Option API

```
<script>
  //declare watch callbacks to be invoked on data change
  watch: {
    a(val, oldVal) {
      console.log(`new: ${val}, old: ${oldVal}`)
    },

    b: 'plus', //string method name, reference to the method

    d: {
      handler(val, oldVal) {
        console.log('d changed')
      },
      //use this parameter if it is an object or an array, so
      that the callback fires on deep mutations
      deep: true
    },

    //watching a single nested property
    'd.e': function (val, oldVal) {  /*do something*/ },

    c: {
      handler(val, oldVal) { /*do something*/ },
      //the callback will be called immediately after the start
      of the observation
      immediate: true,
      // default: 'pre', specify callback timing relative to Vue
      component update
      flush: 'pre' | 'post' | 'sync'     }
  },
</script>
```

Computed, Option API

```
<script>
/* for complex logic that includes reactive data (in this
example "a" and "b"). An object where the key is the name of
the computed property, and the value is either a computed
getter, or an object with get and set methods (for writable
computed properties) */
  computed: {
    //readonly
    summ() {
      return this.a + this.b
    },

    //writable
    square: {
      get() {
        return this.a * this.a
      },
      set(newValue) {
        this.a = Math.sqrt(newValue)
      }
    }
  },
</script>
```

Router

```
<script>
  // import components Home, About, User

  // Define some routes
  // Each route should map to a component.
  // We'll talk about nested routes later.
  const routes = [
    { path: '/', component: Home },
    { path: '/about', component: About },
  ]

  //dynamic routes
  const routes = [
    // dynamic segments start with a colon
    { path: '/users/:id', component: User },
  ]

  //global before guards, uses for the authentication
  const router = createRouter({ ... })
  router.beforeEach((to, from) => {
    // ...
    // explicitly return false to cancel the navigation
    return false
  })
</script>
```

Router, nested routes

```
<script>
  //nested routes
  const routes = [
    {
      path: '/user/:id',
      component: User,
      children: [
        {
 //UserProfile will be rendered inside User's <router-view>
 //when /user/:id/profile is matched
          path: 'profile',
          component: UserProfile,
        },
        {
 //UserPosts will be rendered inside User's <router-view>
 //when /user/:id/posts is matched
          path: 'posts',
          component: UserPosts,
        },
      ],
    },
  ]

</script>
```

Router, template syntaxes

```
<template>
   <!-- use the router-link component for navigation. -->
   <router-link to="/about">Go to About</router-link>
</template>
```

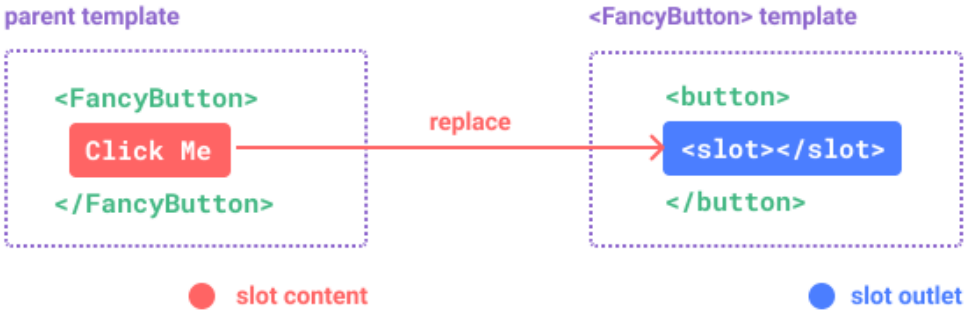
Router, Option API

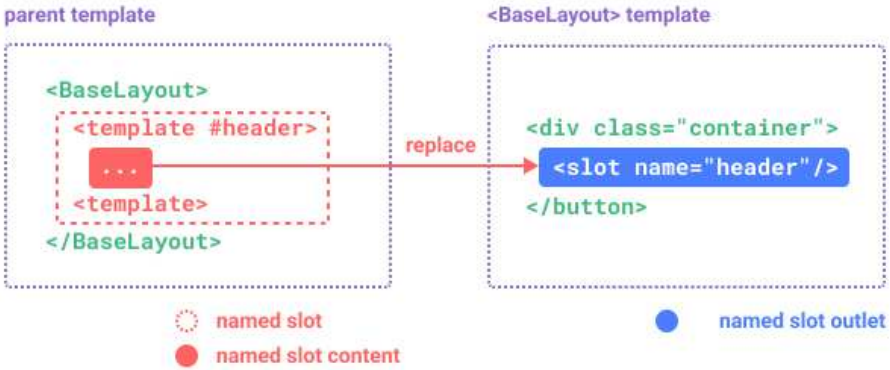
```
//inside Components:
<script>
  //import

  export default {
   …..
   this.$route.params //get parameter
   this.$router.push('/login') //got to /login page
   …..
  }
</script>
```

Router, Composition API

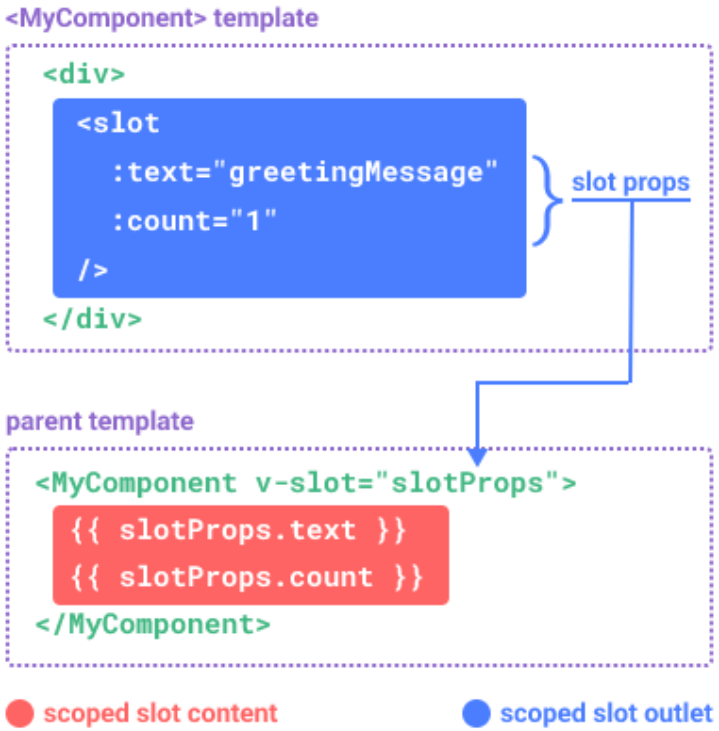
```
//inside Components:
<script>
  import { useRouter, useRoute } from 'vue-router'

  setup() {
    const router = useRouter()
    const route = useRoute()

    //go to About page
    router.push('/About')
    //get route params
    console.log(route.params)
  }
</script>
```

Slots


parent template
<FancyButton>
Click Me
</FancyButton>
replace
<FancyButton> template
<button>
<slot></slot>
</button>
slot content
slot outlet


parent template
<BaseLayout>
<template #header>
...
<template>
</BaseLayout>
replace
<BaseLayout> template

<slot name="header"/>
</button>
named slot
named slot content
named slot outlet


<MyComponent> template

<slot
:text="greetingMessage"
:count="1"
/>

slot props
parent template
<MyComponent v-slot="slotProps">
{{ slotProps.text }}
{{ slotProps.count }}
</MyComponent>
scoped slot content
scoped slot outlet

Vuex

```
<script>
  const store = createStore({
    state: {
      todos: [{ id: 1, text: '...', done: true }, { id: 2, text: '...', done: false }],
      count: 1                      //inside Component (Option API) this.$store.state.count
    },

    //the only way to actually change state in a Vuex store is by committing a mutation
    mutations: {
      increment (state, a) {        //to call mutation inside Component (Option API) - this.$store.commit('increment', 10)
        // mutate state
        state.count += a
      }
    },

    //Actions are similar to mutations, the differences being that: 1.Instead of mutating the state, actions commit mutations;
    //2.Actions can contain arbitrary asynchronous operations.
    actions: {
      async increment (context) {  //to call actions inside Component (Option API) - this.$store.dispatch('increment')
        const value = await axios.get('/api/value');
        context.commit('increment', value.data)
      }
    },

    //to get from state use getters
    getters: {
      doneTodos (state) {           //to call getters inside Component (Option API) - this.$store.getters.doneTodos
        return state.todos.filter(todo => todo.done)
      }
    }
  })
</script>
```

Vuex, modules

```
<script>
  const moduleA = {
    state: () => ({ ... }),
    mutations: { ... },
    actions: { ... },
    getters: { ... }
  }

  const moduleB = {
    state: () => ({ ... }),
    mutations: { ... },
    actions: { ... }
  }

  const store = createStore({
    modules: {
      a: moduleA,
      b: moduleB
    }
  })

  store.state.a // -> `moduleA`'s state
  store.state.b // -> `moduleB`'s state
</script>
```

Vuex, Composition API

```
<script>
  import { useStore } from 'vuex'

  setup() {
    const store = useStore()

    //how to operate with state
    store.state.count

    //access a mutation
    store.commit('increment')

    //access an action
    store.dispatch('asyncIncrement')

    //access a getter
    store.getters.double
  }
</script>
```

Pinia

```
<script>
  import { defineStore } from 'pinia'

  // useStore could be anything like useUser, useCart. The first argument is a unique id of the store across your application
  export const useStore = defineStore('main', {
    state: () => {            // arrow function recommended for full type inference
      return {
        counter: 0, name: 'Eduardo', isAdmin: true, userData: null,
      }
    },

    //Actions are the equivalent of methods in components, can be asynchronous - you can await inside of them any API call or even
    other actions
    actions: {
      increment() {
        this.counter++
      },
      async registerUser(login, password) {
        try {
          this.userData = await api.post({ login, password })
        } catch (error) {
          return error
        }
      },
    },

    //Getters are exactly the equivalent of computed values for the state of a Store
    getters: {
      doubleCount: (state) => state.counter * 2,
    },
  })
</script>
```

Pinia, Options API

```
<script>
  //to access pretty much everything from the store use
  mapStores() with computed option

  import { mapStores } from 'pinia‘
  import { useStore, anotherStore } from ‘../store‘


  export default {
    computed: {
      //not passing an array, just one store after the other
      ...mapStores(useStore, anotherStore)
    },

    //each store will be accessible as its id + 'Store’
    //use this.mainStore anywhere!
    //you can directly read and write state
    //can directly call any action as a method of the store
    //directly access any getter as a property of the store

    methods: {
      myMethod() {
        this.mainStore.increment()
      },
    },
  }
</script>
```

Pinia, Composition API

```
<script>
  import { useCounterStore } from '../stores/counterStore'

  export default {
    setup() {
      const counterStore = useCounterStore()

      counterStore.counter++  //directly read and write state
      counterStore.$reset()   //reset to its initial value

      //call the $patch method to mutate state or part of state
      counterStore.$patch({
        counter: counterStore.counter + 1,
        name: 'Abalam',
      })

      //use $patch with a function
      counterStore.$patch((state) => {
        state.items.push({ name: 'shoes', quantity: 1 })
        state.hasChanged = true
      })

      //call any action as a method of the store
      counterStore.randomizeCounter()

      //directly access any getter as a property of the store
      counterStore.doubleCount

      return { counterStore }
    },
  }
</script>
```

Composition API

```
<template>
 …
</template>

<script>
  //import

  export default {
    //common options          (9)

    //component registration (9)
    //props registration     (10)
    //emits registration     (10)

    //setup function {
         //reactive data
         //methods                (21)
          //computed              (22)
          //watch or watchEffect  (22-23)
          //lifecycle hooks       (24)

          //component instance feature
        return {
          //export to template
        }
    }
  }
</script>

<style scoped lang="SCSS">
 …
</style>
```

See Options API

Setup(), Composition API

```
<script>
  //importing nessesary fucntions from Vue
  import { ref, reactive, toRefs, toRef } from 'vue'

  export default {
/*  The setup() function is a key entry point for Composition API
setup() replaces data() function from Option API, all other options  - methods,
computed, watch, lifecycle hooks declare inside setup() function.
    Setup() has no "this", setup() called after beforeCreate hook and before Created
hook.
    Setup() has two arguments: props and context = {attrs, slots, emit, expose} objects
*/
    setup(props, context) {
      //reactive() takes an object and return a reactive object
      const ob = reactive({key1: value1, key2: value2})

      /* ref() returns a reactive object, which has a single property .value     that
points to the inner value. ref() uses for a primitive data and for objects also. If the
object contains nested refs, they will be deeply reactive */
      //count = reactive object of {value:0}
      const count = ref(0)
      //namesArr = reacative object of {value: ['Joe', 'Bob', [1, 2]]}
      const namesArr = ref(['Joe', 'Bob', [1, 2]])
      //export to template, ref Unwrapping in Templates
      return {
        count,                  //it is 'ref' format
        ...toRefs(ob),          //used toRefs() to transform to "ref" format
        toRef(props, 'title'),  //transform to "ref" format props.title
      }
    },
  }
</script>
```

Reactivity API: Utilities

```
<script>
  //isRef() - Checks if a value is a ref object.

  //unref() - Returns the inner value if the argument is a ref, otherwise return the argument itself.

  /* toRef() - Can be used to create a ref for a property on a source reactive object. The created ref is synced with its source property: mutating the source property will update the ref, and vice-versa.*/
  const state = reactive({
    foo: 1,
    bar: 2
  })

  const fooRef = toRef(state, 'foo')

  // mutating the ref updates the original
  fooRef.value++
  console.log(state.foo) // 2
  // mutating the original also updates the ref
  state.foo++
  console.log(fooRef.value) // 3

  //toRef() is useful when you want to pass the ref of a prop:
  useSomeFeature(toRef(props, 'foo'))

  /* toRefs() converts a reactive object to a plain object.Each individual ref is created using toRef() */
</script>
```

Methods, Composition API

```
<script>
  //importing necessary functions from Vue
  import { ref, reactive, toRefs, computed } from 'vue'

  export default {
    setup(props, context) {

      const ob = reactive({key1: value1, key2: value2})
      const count = ref(0)


      //methods declared as functions
      function increaseCount() {
        count.value + ob.key1          //use .value with "refs"
        context.emit('changed-value') //emit event
      }

      //export to template
      return {
        increaseCount, //export functions/methods/computed
      }
    }
</script>
```

Computed, Composition API

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

  export default {
    setup(props, context) {

      const firstName = ref('John')
      const lastName = ref('Doe')

      //computed
      const compValue = computed(() => {
        return firstName.value + props.num
      })

      //computed with getter and setter
      const fullName = computed({
        // getter
        get() {
          return firstName.value + ' ' + lastName.value
        },
        // setter
        set(newValue) {
          firstName.value = newValue.split(' ')[0]
          lastName.value = newValue.split(' ')[1]

        }
      })

      return { compValue, fullName }
    }
  }
</script>
```

Watch, Composition API

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

  export default {
    setup(props, context) {

      const x = ref(0); const y = ref(0)
      const obj = reactive({ count: 0 })

      // single ref
      watch(x, (newX) => {console.log(`x is ${newX}`)})

      // getter function
      watch(
         () => x.value + y.value,
         (sum) => {console.log(`sum of x + y is: ${sum}`)}
      )

      // array of multiple sources
      watch([x, () => y.value], ([newX, newY]) => {
         console.log(`x is ${newX} and y is ${newY}`)
      })

      // watcher for reactive value
      watch(
         () => obj.count,
         (count) => {console.log(`count is: ${count}`)},
         //additional options - see Option API:
         { deep:.., immediate:..., flush:.... } )
    }
  }
</script>
```

WatchEffect, Composition API

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

  export default {
    setup(props, context) {

      const url = ref('https://...')
      const data = ref(null)

      async function fetchData() {
        const response = await fetch(url.value)
        data.value = await response.json()
      }

      // fetch immediately
      fetchData()
      // ...then watch for url change
      watch(url, fetchData)

    }
  }
</script>
```

can be rewritten as

```
//watchEffect
//runs a function immediately while reactively tracking its
dependencies
//it is like a Computed but without return

/* watch vs. watchEffect
watch and watchEffect both allow us to reactively perform side
effects. Their main difference:

 - watch only tracks the explicitly watched source.
It won't track anything accessed inside the callback.

 - watchEffect combines dependency tracking and side effect
into one phase. It automatically tracks every reactive property
accessed during its synchronous execution. This is more
convenient and typically results in terser code, but makes its
reactive dependencies less explicit.
*/
```

```
watchEffect(async () => {
  const response = await fetch(url.value)
  data.value = await response.json()
})
```

Lifecycle hooks, Composition API

```
<script>
import {onBeforeMount, onMounted, onBeforeUpdate, onUpdated,
onBeforeUnmount, onUnmounted } from 'vue'

    //called right before the component is to be mounted
    onBeforeMount(() => {...})

    //called after the component has been mounted
    onMounted(() => {...})

    //called right before the component is about to update its
    DOM tree due to a reactive state change.
    onBeforeUpdate(() => {...})

    //called after the component has updated its DOM tree due to
    a reactive state change
    onUpdated(() => {...})

    //called right before a component instance is to be
    unmounted
    onBeforeUnmount(() => {...})

    //called after the component has been unmounted
    onUnmounted(() => {...})

    //additional hooks
    import {onErrorCaptured, onActivated, onDeactivated,
onServerPrefetch, ...} from 'vue‘

</script>
```

DOM, Composition API

```
<template>
  <input :ref="inputEl"/>
</template>

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

export default {
  setup() {
    const inputEl =ref(null)
    console.log(inputEl.value) //input value
    return {inputEl}
  }
}
</script>
```

Provide/inject, Composition API

```
<script>
import { ref, provide, inject } from 'vue';

export default {
  setup() {
    // provide reactive value
    const count = ref(0)
    provide('count', count)

    // inject reactive value
    const count = inject('count')
  }
}
</script>
```

NextTick, Composition API

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

export default {
  setup() {
    function increment() {
      state.count++
      nextTick(() => {
        // access updated DOM
      })
    }
  }
}
</script>
```

Vue instance, Composition API

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

export default {
  setup() {
    const internalInstance = getCurrentInstance()

    internalInstance.appContext.config.globalProperties
  }
}
</script>
```

Script Setup API

```
<script setup> //syntactic sugar for using Composition API
  //import section, can be used in Template, no return!!!
  import { ref, computed, watch, onMounted, provide, inject } from 'vue'
  import MyComponent from './MyComponent.vue'

  provide('message', 'hello!')
  const message = inject('message')

  //declare props
  const props = defineProps({foo: String})
  //declare emits
  const emit = defineEmits(['change', 'delete'])

  //methods-functions
  function submitForm(email, password) {emit('submit', { email, password })}

  //variable
  const msg = 'Hello!'
  //reactivity
  const count = ref(0); const x = ref(0); const y = ref(0)

  //computed
  const b = computed(() => count.value + 2)
  //watch
  watch(x, (newX) => { console.log(`x is ${newX}`) })

  //lifecycle hooks
  onMounted(() => { input.value.focus()} )

  //await functions
  const post = await fetch(`/api/post/1`).then((r) => r.json())
</script>
```

Script Setup API example 1, 2

```vue
<script setup>
  import { ref } from 'vue'
  const c = ref(0); const f = ref(32)

  function setC(e, v = +e.target.value) {
    c.value = v
    f.value = v * (9 / 5) + 32
  }
  function setF(e, v = +e.target.value) {
    f.value = v
    c.value = (v - 32) * (5 / 9)
  }
</script>

<template>
  <input type="number" :value="c" @change="setC"/> Celsius
  <input type="number" :value="f" @change="setF" /> Fahrenheit
</template>
```

```vue
<MyComponent v-model:title="bookTitle" />

<!-- MyComponent.vue -->
<script setup>
  defineProps(['title'])
  defineEmits(['update:title'])
</script>

<template>
  <input type="text“ :value="title"
    @input="$emit('update:title', $event.target.value)"
  />
</template>
```

Script Setup API example 3

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

  const groceryList = ref([
   { id: 0, text: 'Vegetables' },
   { id: 1, text: 'Cheese' },
   { id: 2, text: 'Whatever else humans are supposed to eat' }
  ])
</script>

<template>
  <ol>
    <TodoItem
      v-for="item in groceryList"
      :todo="item"
      :key="item.id"
    ></TodoItem>
  </ol>
</template>

<!-- TodoItem.vue -->
<script setup>
  const props = defineProps({
    todo: Object
  })
</script>

<template>
  <li>{{ todo.text }}</li>
</template>
```