Vue js

Course Outline

- Vue JS Overview (Project scaffolding & Use cases)
- Fundamental Concepts
  - Template Syntax
  - Reactivity
- Rendering Mechanism
  - Render Pipeline
  - Virtual DOM
  - Lifecycle hooks & watchers

Course Outline

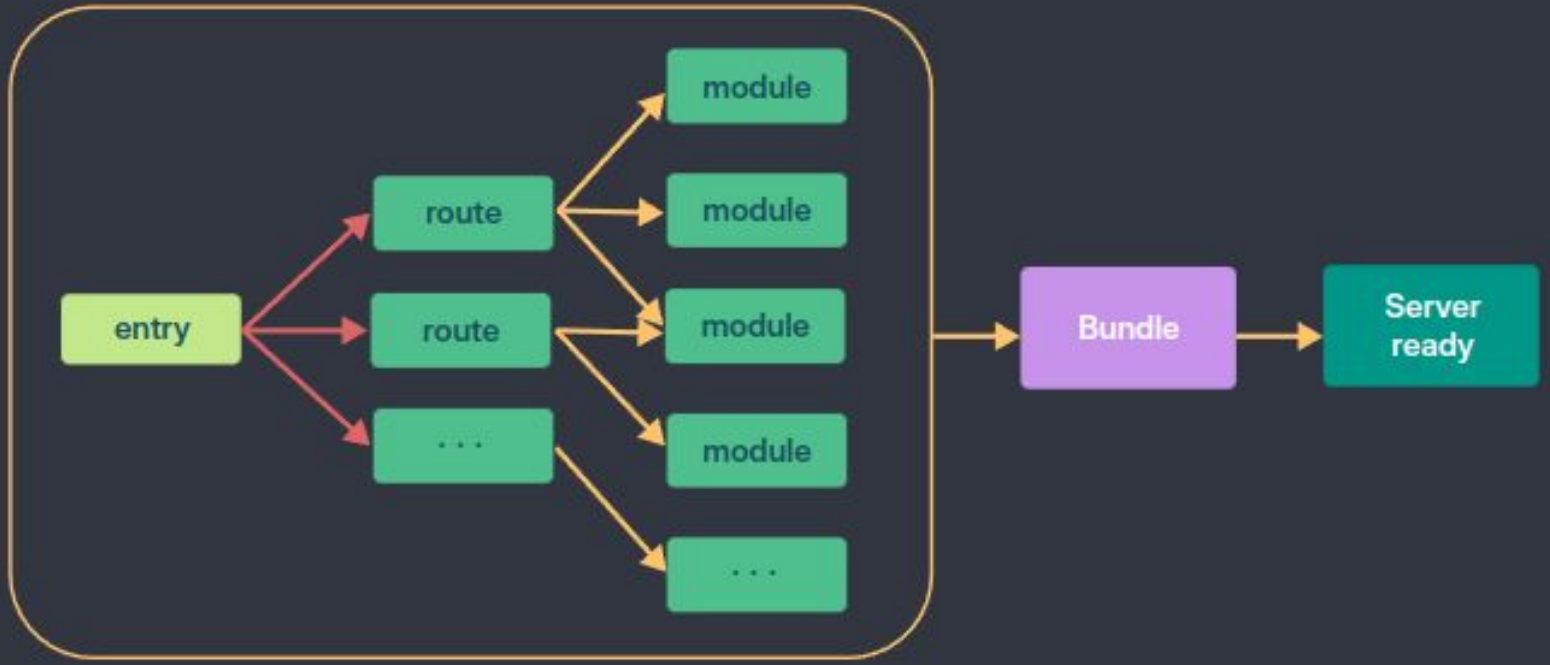
- State Management with Pinia
- Todo App
- Performance Optimization
- Production Deployment

Project Scaffolding

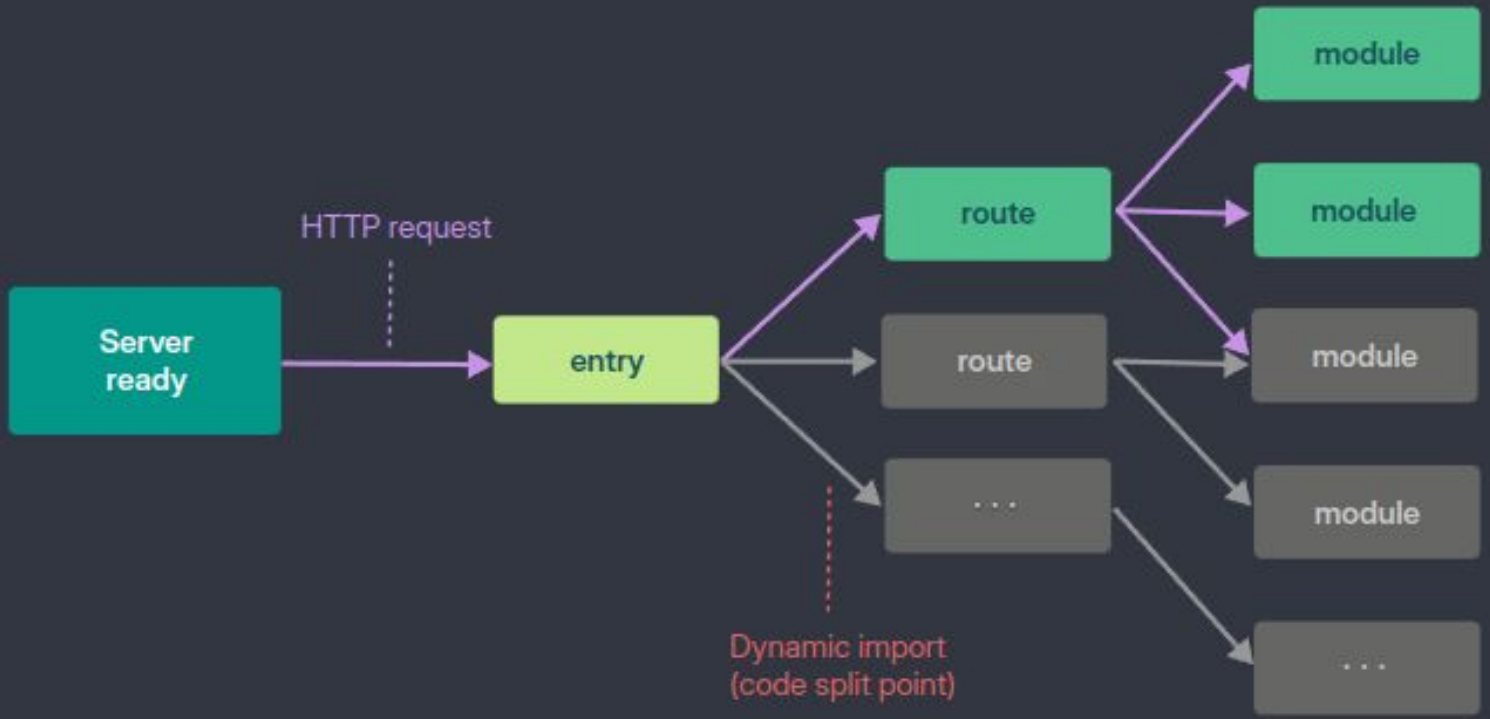
- Build Tool is Vite: Takes advantage of browsers native capability to use ESModules

- Divides application module into two parts
    - Dependencies: pre-bundled using esbuild , are strongly cached
    - Source Code: Non plain JS that needs transforming
        - Served over native ESM
        - Source code module requests are made conditional via 304 Not Modified requests (If requested module did not change, browser uses cached version)

Bundle based dev server



Native ESM based dev server



Vue Js Overview

- A framework that provides declarative & component-based architecture to build user interfaces

- Built on top of HTML, CSS & JS

- Reactive Data Binding: Automatically tracks dependencies and updates DOM

Vue Js - Use cases

1. Standalone script without a build step
    - Provides a pre compiled runtime from CDN that can render & process vue components in the browser
2. Embedded web components
    - Components that can be embedded in any HTML page by creating customElements
3. Single Page Applications
4. SSR & SSG
5. Targets Desktop, Mobile Apps , etc

Vue Js - Create a Vue App from CDN

Using the global ES Module build

```
<script type="module">
import { createApp, ref } from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'
createApp({
  setup() {
    const message = ref('Hello Vue!')
    return {
      message
    }
  }
}).mount('#app')
</script>
<div id="app">{{ message }}</div>
```

Vue Js - Creating a Vue SPA

1. Create an application instance using createApp()
2. Pass a root component
3. Mount the instance

```
// main.js
import { createApp } from 'vue'
import App from './App.vue'

const app = createApp(App)
app.use() // Add any plugins

app.mount('#app') // After all configs are added
```

Vue js Fundamentals

Core Features

1. Declarative Writing: HTML like template syntax to describe HTML output based on JS state
2. Reactivity: Automatically tracks JS state changes on runtime & updates DOM
3. Utilizes Virtual DOM

Template Syntax

- HTML based template syntax
- Templates are compiled into highly optimized JS code at runtime (Render Functions)
- Data binding:

```
<span>Message: {{ msg }}</span> // Text interpolation using mustache syntax
```

Template Syntax

Directives: special built in attributes with the v- prefix that can be used inside HTML attributes

```
<p v-if="seen">Now you see me</p>

<div v-bind:id=”dynamicId”></div> or <div :id=”dynamicId”></div>

<button v-bind:disabled="disableBtn">Button</button> or <button
:disabled="disableBtn">Button</button>

<a v-on:click="doSomething"> ... </a> or <a @click="doSomething"> ... </a>
```

Modifiers

```
<form @submit.prevent="onSubmit">...</form>
```

Reactivity

- Dependency change based reactivity system
- Vue tracks every state used inside the template during rendering
- When that state mutates, Vue triggers a re-render
- Tracking is performed on runtime directly in the browser
    - Pros: Reactivity can work without a build step

Reactivity APIs - ref()

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

const count = ref(0) // A reactive state that returns a ref object

function increment() { // Mutates the state
 count.value++ // The ref object will contain a .value property
}
</script>

<template>
 <button @click="increment">
   {{ count }} // Automatically unwrapped inside templates
 </button>
</template>
```

Reactivity APIs - ref()

- Values are deeply reactive
- To opt out of deep reactivity & reduce observation cost, we can use shallowRef()
- Better for primitive values

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

const obj = ref({
 nested: { count: 0 },
 arr: ['foo', 'bar']
})

function mutateDeeply() {
 // these will work as expected.
 obj.value.nested.count++
 obj.value.arr.push('baz')
}

</script>
```

Reactivity APIs - reactive()

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

const state = reactive({count : 0}) // The object itself is reactive (no inner value wrapping)

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

<template>
 <button @click="increment">
   {{ state.count }}
 </button>
</template>
```

Reactivity APIs - reactive()

- Only works for object types (Object, array & collection types like Map, Set)
- Better for complex states & objects
- Are also deeply reactive
- To opt out of deep reactivity we can use shallowReactive()

reactive() limitations

- Only works for object types (Object, array & collection types like Map, Set)
- Can not replace the entire object

```
let state = reactive({ count: 0 })
state = reactive({ count: 1 }) ({ count: 0 }) is no longer being tracked
```

- Not destructure friendly

```
const state = reactive({ count: 0 })
 // count is disconnected from state.count when destructured or assigned
let { count } = state
let count = state.count
count++ // does not affect original state
state.count++ // this will still be reactive
```

Vue js Rendering in Depth

1. Observers

- When we create a reactive state using ref or reactive, Vue creates an observer under the hood
- The observer is responsible for intercepting & tracking property access updates using getters & setters
- When any change is detected, the observer will notify watchers

Reactivity in depth

```
// Using Ref (Getters & Setters)
function ref(value) {
 const refObject = {
   get value() { // track in getter
     track(refObject, 'value')
     return value
   },
   set value(newValue) { // triggered in setter
     value = newValue
     trigger(refObject, 'value')
   }
 }
 return refObject
}
```

```
// Using Reactive (Proxy)
function reactive(obj) {
 return new Proxy(obj, { // Returns a proxy
   get(target, key) { // track in getter
     track(target, key)
return target[key]
   },
   set(target, key, value) {
     target[key] = value
     trigger(target, key) // triggered in setter
   }))}
```

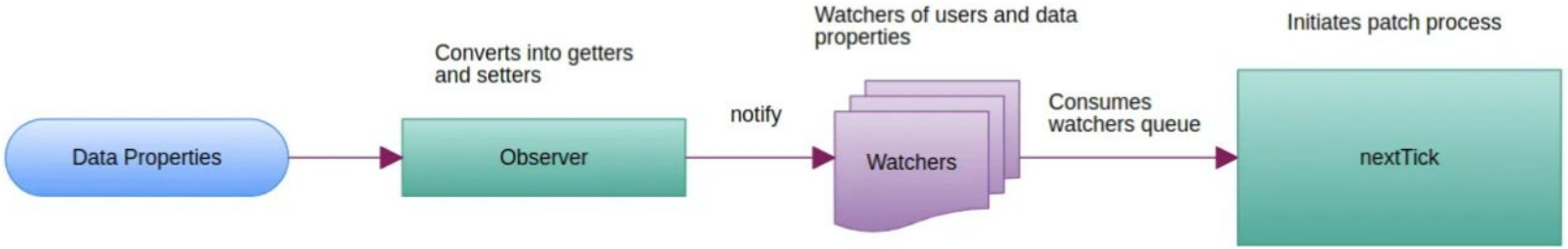
Vue js Rendering in Depth

2. Watcher:

- When a setter is triggered by an observer, watcher is notified which triggers the patch process

- Every component will have its own watcher instance when an application is initialized

Vue js Rendering in Depth



Reactivity in components

Two ways of using reactivity in components

1. Options API
2. Composition API

Options API

```
<script lang="ts">
export default {
 data() {
   return {
     count: 0 // reactive state
   }},
 methods: {
   increment() {
     this.count++
   }},
 mounted() {
   // methods can be called in lifecycle hooks, or other methods!
   this.increment()
 }}
</script>
```

Composition API

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

const count = ref(0)

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

<template>
 <button @click="increment">
   {{ count }}
 </button>
</template>
```

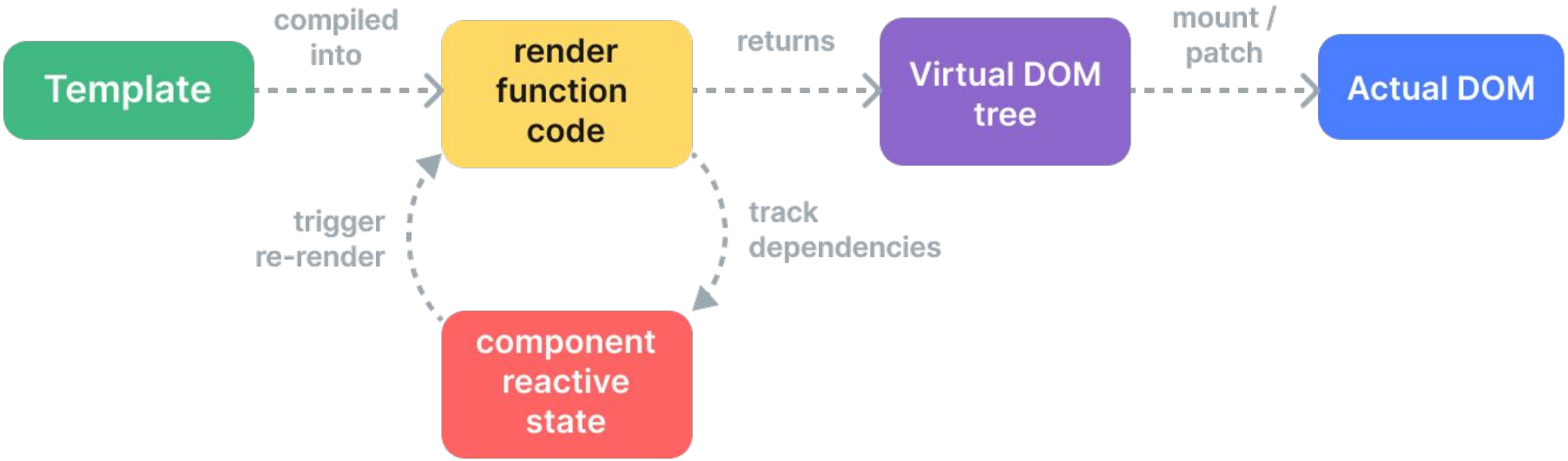
Renderer Mechanism

- Compile: Vue templates containing reactive objects must be compiled into a render function (Ahead of time via build step or on the fly using the runtime compiler)

- Mount: Runtime renderer invokes the render function which returns a vDOM tree. Creates a DOM based on the output. Keeps track of dependencies.

- Patch (Diffing / Reconciliation): When a dependency used during mount changes, the effect reruns & and a new vDOM is created and necessary changes are applied to the actual DOM

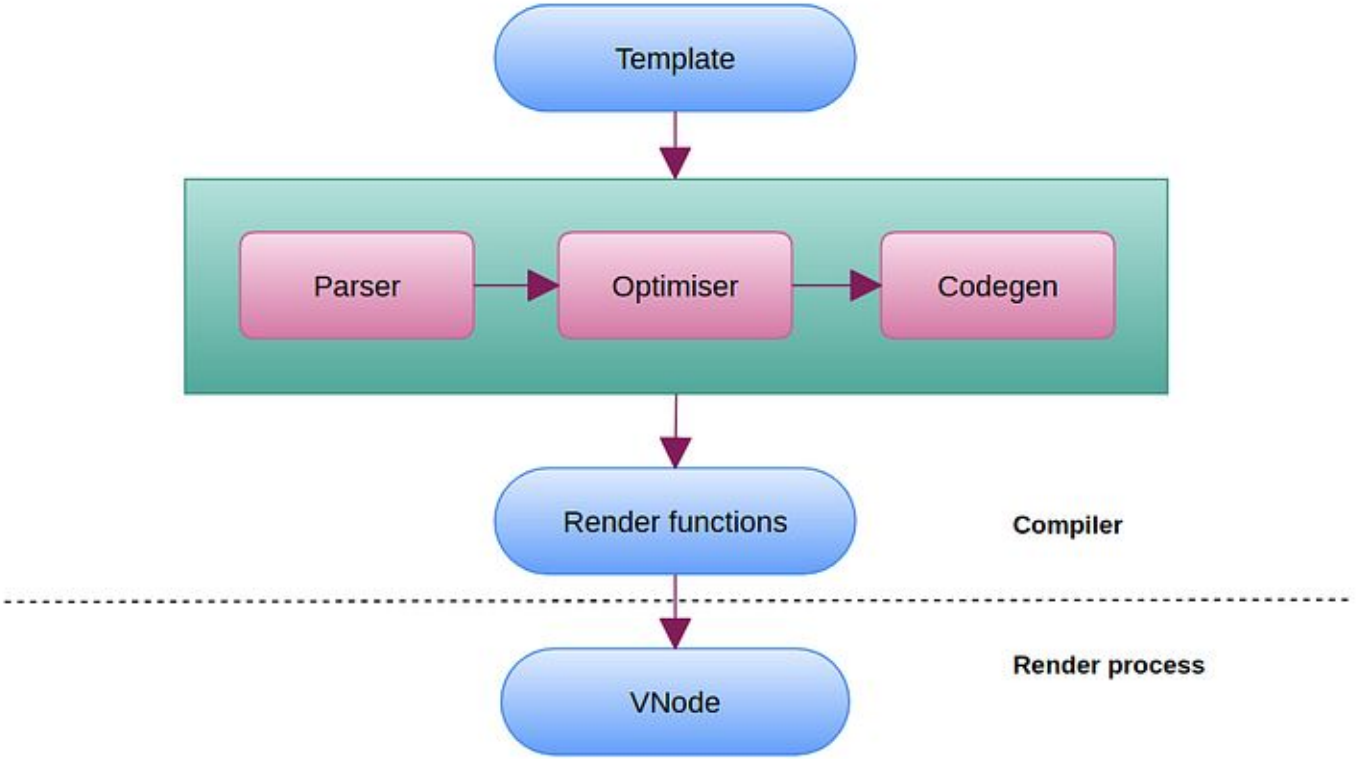
Render Pipeline



Vue js Compiler in Depth

- Vue templates containing reactive objects go through a series of steps to result in the render functions that ultimately output the Virtual Node which is used by the Patch Process to create the actual DOM

Vue js Compiler in Depth



Vue js Compiler in Depth

1. Parsing:
   - Template syntax is converted into AST

```
<div>
  <span>{{ message }}</span>
</div>
```



```
{
 type: 1,
 tag: 'div',
 children: [
   {
     type: 1,
     tag: 'span',
     children: [
       {
         type: 2,
         expression: '_s(message)',
         text: '{{ message }}'
       }]}]}
```

Vue js Compiler in Depth

2. Optimizing:

- Walks through the AST & identifies sub trees that are purely static (part of the DOM that does not need to change) and marks them as static
- This is called Static Hoisting & Vue will not create fresh nodes for them on each re-render.
- These nodes will be skipped completely during the patching process of the virtual DOM.

Vue js Compiler in Depth

```
<div>
<div> foo </div>  <!-- hoisted   -->
<div> bar </div>  <!-- hoisted  -->
 <div>{{ dynamicContent }}</div>
</div>
```

Vue js Compiler in Depth

3. Code Gen

- Render functions are generated

- These render functions are used to create VNodes while triggering the Render Process

Generated Render Function

```
function render() {
 with (this) {
   return _c('div', [
     _c('span', [
       _v(_s(message))
     ])
   ])
 }
}
```

_c is an alias for createElement

used to create vDOM nodes

_v is an alias for createTextVNode

function which created vDOM text nodes

Vue js Patching

- DOM and vDOM interaction using snabbdom library
- Old vDOM & New vDOM comparison takes place
- Nodes flagged as static will remain untouched

React vDOM vs Vue vDOM

- React
  - Purely Runtime
  - Reconciliation algorithm can not make any assumptions about incoming vDOM tree
  - Has to fully traverse the tree & diff the props of every vNode
  - Even if part of a tree never changes, new vNodes are always created on each re-render
  - Brute force comparison of vDOM that sacrifices efficiency for correctness

React vDOM vs Vue vDOM

- Vue (Compiler Informed vDOM)
  - Controls both the compiler & the runtime
  - Compiler statically analyzes the templates & leaves hins for the runtime
  - Static Hoisting
  - Tree Flattening: Reduces the number of nodes needed to be traversed

React vDOM vs Vue vDOM

- Patch Flags used by runtime renderer:

```
<div :class="{ active }"></div>  <!-- class binding only -->

createElementVNode("div", {class: _normalizeClass({ active: _ctx.active })
}, null, 2 /* Class Flag */) // The type of update needed is encoded in the
vNode
```

Lifecycle

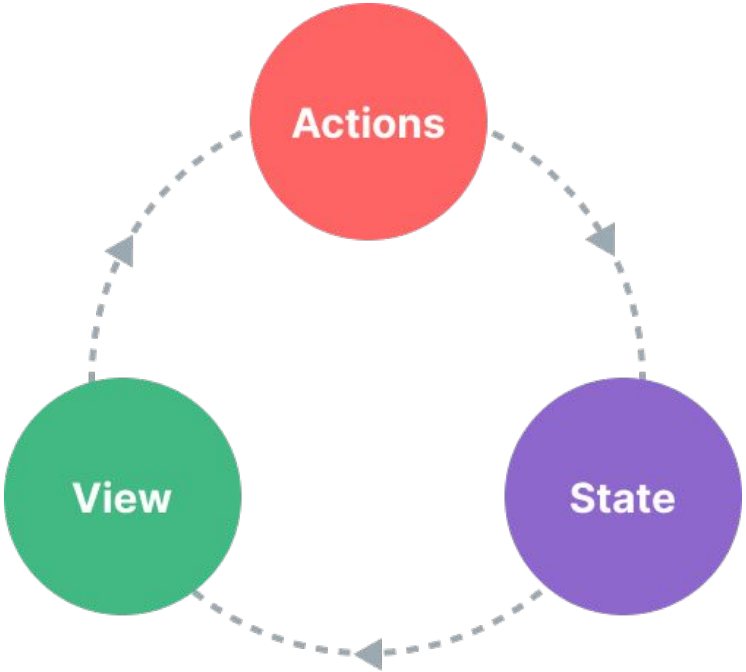
- Creation
  - beforeCreation: Collecting watchers and reactive state dependencies
  - created: setUp data & watchers are set up
- Monting
  - beforeMount: Before patch process - vNodes are getting created based on data & watchers
  - mount: After Patch Process
- Updating
  - beforeUpdate: Watcher updates vNode & re-initialized patch process again if data changes
  - Update: Patch process is done
- Destroying
  - beforeDestroy
  - Destroyed: Removes watchers , event listeners and child components

Life cycle hooks & watchers

- onMounted, onUpdated, onUnmounted
- Watchers are used to handle side effects in reaction to state changes
- watch() function
- watchEffect() function

State Management

- Each vue component instance already manages its own reactive state



State Management - Pinia

- Centralized store
- Integration with DevTools, in-component inspection & time travel debugging
- Supports Composition API
- Feature can be extended using plugins

Demo

Performance Optimization

- Code Splitting
- Tree Shakable APIs, Add packages that offer ES module formats that are tree-shaking friendly (lodash-es over lodash)
- Props Stability
- v-once & v-memo, shallowRef & shallowReactive
- List virtualization: Rendering only items that are in or close to the viewport
- Avoid unnecessary component abstractions

Production Deployment

- Bundling code with tree-shaking, lazy-loading & chunk splitting
- Take advantage of Vite’s pre configured build command (Rollup)
- During build step, templates are pre-compiled. No need to ship Vue compiler to the browser (Avoids runtime compilation cost)

Why Vue?

- Light weight, Comprehensive Ecosystem, Excellent Performance
- Intuitive reactivity system
- Efficient virtual DOM implementation
- Gentle learning curve

Thank you!