MiniVue — Build Vue 3 from Scratch

A framework from the ground up: reactivity, virtual DOM, compiler, components, router, store and SSR

Serik Muradov

Contents

Why Rebuild Vue from Scratch **1**
What we'll build . . . 1
The ground rules: no build step . . . 2
How the book and the repository are organized . . . 2
Who this is for . . . 2

The JavaScript You'll Need **3**
Values and variables . . . 3
Functions . . . 3
Objects . . . 4
Arrays . . . 4
Destructuring and spread . . . 4
Classes, getters, and setters . . . 5
Set, Map, and WeakMap . . . 6
Symbol . . . 6
Proxy and Reflect . . . 7
Object.is . . . 7
Modules: import and export . . . 7
optional chaining and `!!` . . . 8

Reactivity **9**
The problem in one sentence . . . 9
Two verbs: track and trigger . . . 9
An effect is a function you can re-run . . . 10
Where the dependencies live: targetMap . . . 11
reactive: intercepting an object with Proxy . . . 14
ref: reactivity for a single value . . . 16
computed: lazy and cached . . . 17
Why cleanup is needed: the branching problem . . . 18
watch: observe and get old/new . . . 19
How it all works together . . . 19
Check yourself . . . 20

The Virtual DOM and the Renderer **21**
The idea: describe first, compare second . . . 21
VNode: a node as an object . . . 21
The renderer doesn't know about the browser . . . 22
patch: compare old and new . . . 23

Updating in place . . . 24
Comparing lists by key . . . 24
A gotcha: unmount must walk the whole subtree . . . 25
How this ties into reactivity . . . 25
Check yourself . . . 25

Components **26**
What a component is . . . 26
The instance: a component's private record . . . 27
setup and the public context . . . 27
Tying into reactivity: render as an effect . . . 28
The scheduler: three changes — one re-render . . . 29
A gotcha: a job that re-queues itself . . . 30
props: data flowing top-down . . . 30
emit: events flowing bottom-up . . . 31
slots: markup from outside . . . 32
provide / inject: through the floors . . . 32
Lifecycle . . . 32
createApp: the entry point . . . 33
A gotcha: moving a keyed component needs its `el` . . . 33
Check yourself . . . 33

The Template Compiler **34**
Three translation steps . . . 34
Parsing: from text to tree . . . 34
A gotcha: a parser must always move forward . . . 36
Generation: from tree to code . . . 36
Directives . . . 36
From string to function: with(ctx) . . . 37
How the compiler hooks into the runtime . . . 38
What we simplified . . . 38
Check yourself . . . 39

The Router **40**
The idea: URL → component . . . 40
history: where the URL lives . . . 40
The matcher: parsing paths with parameters . . . 41
The reactive current route . . . 41
Navigation and guards . . . 42
RouterView and RouterLink . . . 43
Wiring into the app . . . 44
What we simplified . . . 45
Check yourself . . . 45

The Store **46**
The store is reactivity again . . . 46
createPinia: the store container . . . 46
defineStore: two styles . . . 47

One creation path . . . 48
proxyRefs: why store.count, not store.count.value . . . 48
storeToRefs: destructuring without losing reactivity . . . 49
Plugins . . . 49
What we simplified . . . 49
Check yourself . . . 49

SSR and Hydration **51**
Rendering to a string . . . 51
createSSRApp . . . 52
The whole server . . . 52
The problem: HTML without life . . . 53
hydrate: adopting the existing DOM . . . 53
A gotcha: the browser merges adjacent text nodes . . . 54
What we simplified . . . 55
Check yourself . . . 55
Wrap-up . . . 55

Forms and Bindings **56**
class and style bindings . . . 56
v-model: two-way binding . . . 57
Event modifiers . . . 57
What we simplified . . . 59
Check yourself . . . 59

Reactivity, Deeper **60**
watchEffect . . . 60
readonly . . . 61
shallowReactive and shallowRef . . . 62
markRaw . . . 62
Wrap-up . . . 62

Directives and Dynamic Components **63**
Custom directives . . . 63
How it works . . . 63
Dynamic components . . . 64
v-model on components . . . 65
What we simplified . . . 65
Check yourself . . . 65

Built-in Components **66**
Teleport: render somewhere else . . . 66
KeepAlive: hide, don't destroy . . . 67
A slots bug found along the way . . . 68
Async components . . . 69
What we simplified . . . 70
Check yourself . . . 70

Store Deep-Dive and the Capstone App **71**

$patch, $subscribe, $reset 71
The final app 72
Two more apps — and a bug the shop caught 72
What we covered 73
What's next 73
Check yourself 73

Why Rebuild Vue from Scratch

There are two ways to understand a tool. The first is to read the manual and start using it. The second is to take it apart and rebuild it yourself. This book is about the second way. We won't learn to write applications with Vue — we'll write Vue itself.

That sounds intimidating, but there's nothing to fear. Inside, Vue isn't magic — it's a handful of simple ideas stacked carefully on top of one another. Reactivity ("the data changed, so the screen updated itself") is just intercepting reads and writes. Components are functions that return a description of markup. The router is a dictionary of "address → what to show." Each of these ideas fits in a hundred or two hundred lines that read like plain prose. We'll go through all of them.

What we'll build

By the end of the book we'll have a working mini-framework with the whole ecosystem of real Vue 3:

- **Reactivity** — `ref`, `reactive`, `computed`, `watch`. The core that makes Vue exist in the first place.
- **Virtual DOM and the renderer** — how a description of the interface turns into real elements on the page, and how to update them surgically instead of redrawing everything.
- **Components** — `setup`, props, events (emit), slots, lifecycle hooks, `createApp`.
- **Template compiler** — how `<div>{{ count }}</div>` turns into a function.
- **Router** — an equivalent of Vue Router: addresses, params, `RouterView`, `RouterLink`.
- **Store** — an equivalent of Pinia: shared application state.
- **SSR** — server-side rendering and "coming alive" (hydration) on the client.

We deliberately use the same names as real Vue. Our `ref` is called `ref`, our `createApp` is called `createApp`. So everything you understand here maps one-to-one onto the real framework — there's just more code there, because it handles edge cases that we'll sometimes skip for the sake of clarity.

The ground rules: no build step

Normally, to run a Vue project, you install Node, then a bundler (Vite or Webpack), and it "magically" turns your files into something the browser understands. For a beginner that's the worst possible start: half of what's happening is hidden inside a tool you don't yet understand.

So we have a rule: **no build step**. We write in plain ES modules — the standard the browser understands on its own. A code file is wired into the page with a single `<script type="module">` line, and `import` works inside it. Open the page, and the code runs. Everything that happens, happens right in front of you, with no middlemen.

We'll need Node.js for exactly two things: running tests and spinning up a tiny local server so the browser will allow modules to load (from the `file://` scheme it won't load them, for security reasons). The framework itself doesn't depend on Node — until the chapter on SSR, where the server is the whole point.

How the book and the repository are organized

Each layer is, at the same time:

- a **chapter** here in the book, where the idea is explained in words and pictures;
- a **folder in `packages/`** with working code, heavily commented;
- **tests in `test/`** that prove the code does what it promises;
- a **demo in `playground/`** you can open in the browser and poke at by hand.

I recommend keeping the code and the chapter side by side: read the explanation, then open the file and find those exact lines. The code in the book and the code in the repository are literally the same lines.

Who this is for

For anyone who wants to understand, not just use. If you've never written JavaScript, don't worry: the next chapter gives you exactly the minimum of the language you need to read our code. If you already write Vue but it's a "black box" to you, this book opens the box.

One warning: don't try to memorize everything at once. Reactivity in the first chapter feels like a puzzle at first — that's normal. Get to the demo, poke the buttons, come back to the code. The idea will click into place. And once it does, everything else in Vue becomes an obvious consequence.

Here we go.

The JavaScript You'll Need

This chapter isn't a full language course — it's a glossary. It covers exactly the constructs that show up in our code, and nothing extra. If you already know JavaScript, skim it and move on to reactivity. If you don't, read it once and then come back to it as a reference whenever an unfamiliar symbol turns up in the code.

Values and variables

A value is a piece of data: the number `42`, the string `"hello"`, the boolean `true` or `false`, or "nothing" — `null` and `undefined`. A variable is a name for a value.

```
const name = 'Anya'  // const — a name you can't reassign
let count = 0        // let — a name you can change
count = count + 1    // now count is 1
```

`const` doesn't mean "the value is immutable" — it means "you can't reassign this name." You can still change the object a `const` name points to; you just can't say `name = another object`. Default to `const`, and reach for `let` only when you genuinely need to reassign.

Functions

A function is a reusable block of actions that you can run many times, feeding it different inputs (arguments).

```
function double(x) {
  return x * 2
}
double(21) // 42
```

There's a shorter form — the arrow function. We'll see it constantly.

```
const double = (x) => x * 2       // single expression — return is implied
const log = (msg) => {            // body in braces — you write return yourself
  console.log(msg)
}
const now = () => Date.now()      // no arguments — empty parentheses
```

You can pass a function to another function as an argument — that's a "callback," a handler function. All of reactivity is built on taking someone else's function and deciding when to call it:

```
effect(() => {
  console.log('rerun me when the data changes')
})
```

Objects

An object is a set of "key: value" pairs. Keys (properties) are read and written with a dot.

```
const user = { name: 'Anya', age: 30 }
user.name       // 'Anya'  — reading
user.age = 31   // writing
user.city = 'Almaty' // adding a new property
```

Remember the phrases **property read** and **property write** — the whole first chapter revolves around them. Reactivity is intercepting exactly these two operations.

Arrays

An array is an ordered list of values. Indexing starts at zero.

```
const items = [10, 20, 30]
items[0]        // 10
items.length    // 3
items.push(40)  // add to the end → [10, 20, 30, 40]
items.reduce((sum, x) => sum + x, 0) // 100 — "fold" into a single number
```

`reduce` walks the elements and accumulates a result: start at `0`, add each element's value. That's how we'll compute a cart total.

Destructuring and spread

You can "unpack" an object or array into separate variables:

```
const { name, age } = user       // name = 'Anya', age = 31
const [first, second] = items    // first = 10, second = 20
```

Three dots `...` mean "spread the rest." In our code you'll see it as "make a copy":

```
const copy = [...items]          // a new array with the same elements
```

This line will save us in reactivity: before iterating over a set of effects, we copy it so that changes made during iteration don't break the loop.

Classes, getters, and setters

A class is a template for creating objects with shared behavior. `new` creates an instance, `constructor` sets up the initial state, and `this` inside means "this particular instance."

```
class Counter {
  constructor(start) {
    this.value = start   // instance field
  }
  increment() {
    this.value++
  }
}
const c = new Counter(5)
c.increment()
c.value // 6
```

A special pair is the **getter** and **setter**. These are methods that look like an ordinary property from the outside, but run code on read/write. That's exactly how `ref` works: reaching for `.value` looks like a field, but actually runs a function.

```
class Box {
  constructor(v) { this._v = v }
  get value() {           // runs on a READ of box.value
    console.log('read')
    return this._v
  }
  set value(newV) {       // runs on a WRITE of box.value = ...
    console.log('wrote')
    this._v = newV
  }
```

```
}
const box = new Box(1)
box.value        // prints "read", returns 1
box.value = 2    // prints "wrote"
```

The underscore in _v is just a convention meaning "this is an internal field, don't touch it from outside." The language doesn't enforce it, but that's the custom.

Set, Map, and WeakMap

Besides arrays and objects, we need three special collections.

Set — a collection of unique values. Add the same thing twice, and it's stored once. Perfect for "the list of effects subscribed to a property": one effect shouldn't land there twice.

```
const s = new Set()
s.add('a'); s.add('a'); s.add('b')
s.size          // 2
s.has('a')      // true
s.delete('b')
```

Map — like an object, but the key can be anything, including another object. That's critical for us: we'll be linking a reactive object to its dependencies.

```
const m = new Map()
m.set('key', 123)
m.get('key')    // 123
```

WeakMap — the same as Map, but with "weak" object keys: if nothing else references a key object anymore, the garbage collector is free to remove it along with its entry. This way the dependency store doesn't keep objects alive that are no longer needed.

Symbol

`Symbol('name')` creates a unique marker value that's guaranteed not to collide with any string key. We use symbols as internal "hidden" properties — for example, a marker meaning "this object is already reactive," which can't be confused with the user's real data.

```
const IS_REACTIVE = Symbol('isReactive')
obj[IS_REACTIVE] // no ordinary property will ever land here by accident
```

Proxy and Reflect

Here's the main tool of reactivity. **Proxy** wraps an object and lets you intercept operations on it — first and foremost reads (`get`) and writes (`set`).

```
const raw = { count: 0 }
const proxy = new Proxy(raw, {
  get(target, key) {
    console.log('reading', key)
    return target[key]
  },
  set(target, key, value) {
    console.log('writing', key, '=', value)
    target[key] = value
    return true   // set must return true on success
  },
})
proxy.count       // prints "reading count", returns 0
proxy.count = 5   // prints "writing count = 5"
```

`reactive` grows out of this tiny bit of interception: in `get` we'll call "remember who read this," and in `set` we'll call "wake up whoever read it."

Reflect — a set of functions that mirror the standard operations. `Reflect.get(target, key, receiver)` is the same as `target[key]`, but it correctly passes `this` to getters and works with inheritance. Inside a Proxy the right thing is to use `Reflect`, not `target[key]` directly — otherwise tricky objects will break.

Object.is

Comparing "did the value change?" A plain `===` works almost everywhere, but it has some historical quirks: `NaN === NaN` gives `false`, even though it's "the same thing." `Object.is` fixes those. We'll ask "is the value really different?" via `!Object.is(old, new)` so we don't wake up the interface when assigning the same value.

Modules: import and export

Code is split into module files. A file shares things by `export`-ing them and grabs other files' things via `import`.

```
// effect.js
export function track() { /* ... */ }
export let activeEffect = undefined

// reactive.js
import { track, activeEffect } from './effect.js'
```

The path is written with the `.js` extension and with `./` for neighboring files — that's what the browser ES modules standard requires, and we use it without a bundler.

optional chaining and `!!`

`a?.b` reads `a.b`, but if `a` is `null` or `undefined`, it calmly returns `undefined` instead of throwing. `!!x` turns any value into an honest `true/false` (two "nots": the first inverts, the second flips it back). When you see `return !!(value && value.__isRef)`, it just means "return true if value has the `__isRef` flag."

This glossary is enough to read any line of our framework. Don't memorize it all — keep this chapter handy and come back as needed. Now, on to the heart of Vue.

Reactivity

Everything in Vue rests on one ability: data changes, and whatever depends on it updates by itself. You change `count`, and the number on screen becomes different, even though you never wrote "find the element and update its text." This chapter is about how that works. We'll build reactivity in full, and from here on the whole framework is just a layer on top of it.

The code for this chapter lives in `packages/reactivity/`. Tests are in `test/reactivity.test.mjs`, and the live demo is `playground/01-reactivity.html`.

The problem in one sentence

We want to be able to write this:

```
const count = ref(0)

effect(() => {
  document.title = 'Clicks: ' + count.value
})

count.value = 5   // and the tab title becomes "Clicks: 5" on its own
```

The idea is simple. There's data (`count`) and there's a function that uses it (`effect`). We want the function to re-run when the data changes — by itself, without our involvement. So the system needs to somehow learn that this function "depends" on `count`. And it can learn that at exactly one moment — when the function reads `count.value`. Everything is built around this idea.

Two verbs: track and trigger

Remember these two words; the rest of the book revolves around them.

- **track** — happens on a **read**. The meaning: "some function is running right now, and it read this value — let's remember the connection."

- **trigger** — happens on a **write**. The meaning: "the value changed — let's find every function that read it and re-run them."

Read — remember the dependency. Write — use the dependency. Everything else is details of how those dependencies are stored.

An effect is a function you can re-run

The function that has to re-run on changes is what we call an **effect**. So the system can manage it, we wrap it in an object. That way we can attach bookkeeping data to the function.

The key variable is `activeEffect`. While an effect's body is running, this variable holds that effect. Any data read at that moment knows "whom" to assign the dependency to. Outside effects it's `undefined`, and a read tracks nothing.

```
// packages/reactivity/effect.js
export let activeEffect = undefined
const effectStack = []
```

Why a stack? Effects can be nested (a component, and inside it a computed value). When the inner effect finishes, the outer one has to become "active" again. The stack handles exactly that: push on entry, pop on exit.

The effect itself is the `ReactiveEffect` class:

```
export class ReactiveEffect {
  constructor(fn, scheduler = null) {
    this.fn = fn
    this.scheduler = scheduler
    this.deps = []
    this.active = true
    this.onStop = null       // optional hook run once when the effect is stopped
    recordEffectScope(this)  // register in the current scope (see below)
  }

  run() {
    if (!this.active) return this.fn()
    cleanup(this)                 // forget old dependencies
    try {
      effectStack.push(this)
      activeEffect = this         // "I'm listening now"
      return this.fn()            // run it — tracks happen inside
```

```
    } finally {
      effectStack.pop()
      activeEffect = effectStack[effectStack.length - 1]
    }
  }
}
```

The magic is in the `run` method. Before calling the function it declares itself active (`activeEffect = this`). While the function runs and reads data, every read sees this `activeEffect` and records the dependency. As soon as the function finishes, we pop ourselves off the stack and restore whoever was active before us. The `try/finally` wrapper guarantees the active effect is restored even if an error happens inside.

The two extra lines in the constructor belong to `EffectScope` — a "basket" in the same file that collects every effect created while it's active (`recordEffectScope`), so a whole group can be stopped with one `scope.stop()`. The component layer will use it later: a component gathers its render effect and watchers into a scope and disconnects them all at once on unmount.

We'll cover `scheduler` and `cleanup` shortly — first let's see how the dependencies are stored.

Where the dependencies live: targetMap

We need to store which effects depend on which property of which object. The structure is three levels deep:

```
targetMap: WeakMap {
  reactiveObject → depsMap: Map {
    'property' → dep: Set(effect1, effect2, …)
  }
}
```

Read it as: "for this object, for this property of it — here is the set of effects that read it." The top level is a `WeakMap`, so forgotten objects aren't held in memory. The middle level is a `Map` keyed by property name. The bottom level is a `Set` of effects (unique, no duplicates).

The `track` function fills this structure in:

```
export function track(target, key) {
  if (!activeEffect || !shouldTrack) return  // no one listening (or tracking paused)

  let depsMap = targetMap.get(target)
  if (!depsMap) targetMap.set(target, (depsMap = new Map()))
```

```
  let dep = depsMap.get(key)
  if (!dep) depsMap.set(key, (dep = new Set()))

  trackEffects(dep)                  // link the active effect to this dep
}

export function trackEffects(dep) {
  if (!activeEffect || !shouldTrack) return
  if (dep.has(activeEffect)) return  // already linked during this run
  dep.add(activeEffect)              // dep knows about the effect
  activeEffect.deps.push(dep)        // the effect knows about the dep
}
```

Notice the two-way link in `trackEffects`. The `dep` remembers the effect — so it can wake it up later. But the effect also remembers the `dep` (in its `deps` array) — which we'll need for cleanup, covered below. The `shouldTrack` flag is a global pause switch (`pauseTracking`/`resetTracking` in the same file): array mutators like `push()` read the array as a side effect of writing, and without the pause an effect that pushes would subscribe to its own reads.

The reverse operation is `trigger`:

```
export function trigger(target, key, type = 'set', newValue) {
  const depsMap = targetMap.get(target)
  if (!depsMap) return               // no one read this object

  // Collect every dep set the change affects, then fire them all at once.
  const deps = []

  if (type === 'clear') {
    deps.push(...depsMap.values())   // collection.clear(): every key is gone
  } else if (key === 'length' && Array.isArray(target)) {
    // … arr.length shrank: wake 'length' readers and removed indices …
  } else {
    deps.push(depsMap.get(key))
    // … 'add'/'delete' (and map.set) also wake iteration effects:
    //   depsMap.get(ITERATE_KEY) or the array's 'length' …
  }

  // Merge into one set so an effect subscribed to several deps runs once.
```

```
  const effects = new Set()
  for (const dep of deps) {
    if (dep) dep.forEach((effect) => effects.add(effect))
  }
  triggerEffects(effects)
}

export function triggerEffects(dep) {
  const effects = [...dep]           // a copy! (explained below)

  // Pass 1: computed effects go FIRST — their caches must be invalidated
  // before plain effects read them.
  for (const effect of effects) {
    if (effect.computed) triggerEffect(effect)
  }
  // Pass 2: plain effects.
  // … the full file also skips effects that pass 1 already re-ran …
  for (const effect of effects) {
    if (!effect.computed) triggerEffect(effect)
  }
}

function triggerEffect(effect) {
  if (effect === activeEffect) return     // don't wake ourselves
  if (effect.scheduler) effect.scheduler()
  else effect.run()
}
```

Two subtleties, each of which would otherwise cost hours of debugging:

1. **A copy of the set** (`[...dep]`) before iterating. When an effect re-runs, its `cleanup` removes it from `dep` and `track` immediately adds it back. Mutating a `Set` you're currently looping over is a sure way to get an infinite loop or a skipped element. We iterate over the copy — let the original change freely.

2. **`if (effect === activeEffect) return`** in `triggerEffect` — protection against self-re-running. If you write `count.value++` inside an effect, it reads and writes `count` at the same time. Without this line it would wake itself up forever.

Two more decisions live in this code. The `type` argument distinguishes changing a value from adding or removing a key — the latter also wakes iteration effects (`for…in`, array `length`, Map/Set `size`), which subscribe under the special `ITERATE_KEY`, since an iteration has no single property to depend on. And `triggerEffects` makes two passes: computed effects run first, so a plain effect that reads both a value and a computed built on it never sees a stale cache (the classic "glitch").

reactive: intercepting an object with Proxy

Now let's make a plain object call `track` and `trigger` automatically. The tool is `Proxy`. In the `get` trap we call `track`, in the `set` trap we call `trigger`. The real file builds these traps with a factory — the same `createGetter`/`createSetter` serve `reactive`, `shallowReactive` and `readonly` (chapter 9), differing only in two flags:

```
// packages/reactivity/reactive.js  (the get and set traps, abridged)
function createGetter(isReadonly = false, isShallow = false) {
  return function get(obj, key, receiver) {
    // … internal markers, instrumented array methods, engine symbols …

    const result = Reflect.get(obj, key, receiver)

    if (!isReadonly) track(obj, key)   // "I was read"

    if (isShallow) return result

    // A ref inside a reactive object unwraps automatically: state.count,
    // not state.count.value (arrays are the exception).
    if (isRef(result)) {
      return Array.isArray(obj) && isIntegerKey(key) ? result : result.value
    }

    if (isObject(result)) {
      return isReadonly ? readonly(result) : reactive(result) // depth on demand
    }

    return result
  }
}

function createSetter(isShallow = false) {
```

```
  return function set(obj, key, value, receiver) {
    let oldValue = obj[key]
    // … a ref at this key absorbs the write into its .value;
    //   oldValue and value are compared "raw", via toRaw …

    const hadKey =
      Array.isArray(obj) && isIntegerKey(key)
        ? Number(key) < obj.length
        : Object.prototype.hasOwnProperty.call(obj, key)

    const result = Reflect.set(obj, key, value, receiver)
    // … prototype-chain guard: only the object actually written to triggers …

    if (!hadKey) {
      trigger(obj, key, 'add')         // a new key appeared
    } else if (hasChanged(oldValue, value)) {
      trigger(obj, key, 'set', value)  // an existing key got a new value
    }

    return result
  }
}

export function reactive(target) {
  return createReactiveObject(target, mutableHandlers, reactiveMap)
}
```

Four decisions worth spelling out.

Deep reactivity on demand. If a read property turns out to be an object itself, we wrap it in `reactive` right at that moment — not ahead of time, recursively across the whole tree (that's expensive and sometimes harmful), but lazily, when something actually reaches for it. So `state.user.name` is reactive at every level, but you don't pay the price until `user` is touched.

The `hasChanged` check. We call `trigger` only if the value is actually different. Assigning the same value shouldn't wake the interface.

`Reflect` instead of `obj[key]`. On objects with getters and inheritance, direct access loses the correct `this`. `Reflect.get/set` with `receiver` preserves it. On plain objects there's no difference, but it's worth getting used to the correct form.

Refs unwrap automatically. A ref stored inside a reactive object is read as `state.count`, not `state.count.value` — the getter returns its `.value`, and the setter writes an assigned non-ref into it. Arrays are the exception: `arr[0]` may legitimately hold a ref (Vue semantics).

`createReactiveObject` at the bottom of the file holds the plumbing: it caches the proxy per target (so `reactive(obj)` twice returns the same proxy), returns primitives, `markRaw`'d objects and non-proxyable ones (Date, RegExp, class instances with internal slots) unchanged, and picks a different handler set for Map/Set — their methods work on hidden internal slots a Proxy doesn't have, so the file replaces them with "instrumented" versions that call `track`/`trigger` by hand. Arrays get the same treatment for a handful of methods: `includes`/`indexOf`/`lastIndexOf` also search the raw array (elements come out wrapped, so an identity search would miss), and `push`/`pop`/`shift`/`unshift`/`splice` pause tracking while they run, so a writer doesn't subscribe to its own incidental reads. Iteration — `for…in`, `Object.keys`, a collection's `size` and `forEach` — is tracked under `ITERATE_KEY`, the key we saw in `trigger`.

ref: reactivity for a single value

`Proxy` intercepts an object's properties. But how do you make a plain number reactive? A number has no properties to intercept. The solution: put the value inside an object, in a `.value` field, and watch reads and writes of that field through a getter and a setter.

```
// packages/reactivity/ref.js
class RefImpl {
  constructor(value) {
    this._value = convert(value)   // an object inside a ref is made reactive too
    this._rawValue = value
    this.dep = new Set()           // its own set of effects (a ref has one "property")
    this.__isRef = true
  }
  get value() {
    if (activeEffect) trackEffects(this.dep)  // read → track
    return this._value
  }
  set value(newValue) {
    if (hasChanged(toRaw(newValue), this._rawValue)) {
      this._rawValue = toRaw(newValue)
      this._value = convert(newValue)
      triggerEffects(this.dep)                // write → trigger
    }
```

```
  }
}
```

This is why you always write `.value` on a `ref`: it's not a whim, it's the only hook through which the getter and setter can step in and call `track`/`trigger`. A `ref` keeps its own `dep` right inside the object — it doesn't need `targetMap`, since it has exactly one "property."

There's a handy helper, `proxyRefs` — it unwraps `.value` automatically so that in templates you can write `count` instead of `count.value`. Later the component layer applies it to the result of `setup`, and `.value` in markup disappears. Its code is short; take a look in `ref.js`.

computed: lazy and cached

A computed value is a formula on top of other reactive data, with two useful properties: it's evaluated **lazily** (only when the result is asked for) and it's **cached** (until the dependencies change, a repeat request returns the ready value).

It's assembled from parts we already have: an effect with lazy startup and a scheduler, plus a "dirty" flag.

```
// packages/reactivity/computed.js
class ComputedRefImpl {
  constructor(getter, setter) {
    this._value = undefined
    this._setter = setter
    this._dirty = true             // whether a recompute is needed
    this.dep = new Set()
    this.__isRef = true
    this.effect = new ReactiveEffect(getter, () => {
      if (!this._dirty) {          // a dependency changed →
        this._dirty = true         // mark as dirty
        triggerEffects(this.dep)   // and wake whoever read the computed
      }
    })
    this.effect.computed = true    // "invalidate me before plain effects"
  }
  get value() {
    if (activeEffect) trackEffects(this.dep)
    if (this._dirty) {             // recompute only if dirty
      this._value = this.effect.run()
```

```
      this._dirty = false
    }
    return this._value
  }
  // … set value(v) hands the write to the setter — a writable computed …
}
```

Here the **scheduler** (the second argument to `ReactiveEffect`) comes into play for the first time. An ordinary effect re-runs immediately when a dependency changes. But `computed` shouldn't evaluate immediately — it should evaluate lazily. So instead of recomputing, its scheduler only raises the `_dirty` flag and wakes the readers. The actual recompute happens in `get value`, and only if someone actually asks for the result. Read it twice without changes, and the second read comes from the cache — the `getter` doesn't run.

The `this.effect.computed = true` mark is the flag `triggerEffects` uses to invalidate computeds before plain effects run. The setter makes a writable computed possible: `computed({ get, set })` forwards assignments to your `set`, while a getter-only computed just warns.

Why cleanup is needed: the branching problem

Back to `cleanup`, which `run` calls before every run. Consider an effect with a condition:

```
effect(() => {
  out.value = show.value ? a.value : b.value
})
```

When `show` is true, the effect reads `show` and `a`, but **not** `b`. So it shouldn't subscribe to `b` — otherwise a change to `b`, which currently affects nothing, would re-run it for no reason. But on the next run `show` might become false, and then it needs `b`, not `a`.

Conclusion: before every run the effect must forget all its previous dependencies and collect them again — only the ones actually read this time. That's what `cleanup` does:

```
function cleanup(effect) {
  for (const dep of effect.deps) dep.delete(effect)
  effect.deps.length = 0
}
```

It walks every `dep` where the effect is recorded (this is why the effect kept back references in `deps`), removes itself from them, and clears its own list. After `cleanup` the effect is "subscribed to nothing" — and then `fn()` subscribes it again, but this time based on what's actually read. In the tests

this is covered by the "branching — unsubscribe from an unused dependency" case: after switching branches, a change to the old variable no longer wakes the effect.

watch: observe and get old/new

`effect` just re-runs a function. `watch` is a layer on top that, on a change, calls a callback and passes it the new and old values. The source can be a `ref`, a getter function, an entire `reactive` object, or an array of such sources.

The idea: reduce any source to a getter function, wrap it in an effect, and put the whole "reaction" in the scheduler — it computes the new value and calls the callback.

```
// packages/reactivity/watch.js  (abridged)
let oldValue
const job = () => {
  const newValue = effect.run()
  // … skip if nothing actually changed …
  callback(newValue, oldValue, onCleanup)
  oldValue = newValue
}
// 'sync' fires right inside the mutation; the default ('pre') batches the job
// into a microtask queue — three synchronous writes mean one callback call.
const scheduler = flush === 'sync' ? job : () => queueWatchJob(job)
const effect = new ReactiveEffect(getter, scheduler)
oldValue = effect.run()          // first run: collect dependencies, remember the start
```

For a `reactive` object, to watch it "deeply," a `traverse` walk is used — it recursively reads every nested property, thereby subscribing the effect to each level (`deep: true` does the same for any source, and an array of sources is reduced to a getter over all of them). `immediate: true` makes the callback fire right away, `once: true` stops the watcher after the first call. The callback's third argument, `onCleanup`, registers a function that runs before the next invocation and on stop — the standard way to cancel a request that a newer value has made stale.

How it all works together

Let's put the picture together with an example from the demo. There's `count = ref(0)`, there's a `computed doubled = count.value * 2`, and there's an effect that writes both numbers into the DOM.

1. The effect runs for the first time. `activeEffect` is it. It reads `count.value` (→ `track`: `count.dep` remembered the effect) and `doubled.value`.

2. Reading `doubled.value` while `_dirty = true` runs its inner effect, which reads `count.value` (→ `count.dep` also remembered the computed effect). The result is cached, `_dirty = false`.
3. The user clicks "+1": `count.value = 1`. The `ref` setter calls `triggerEffects`.
4. `count.dep` now has two "listeners": our DOM effect and the computed effect. The computed has a scheduler — it sets `_dirty = true` and wakes the readers of `doubled`. The DOM effect has no scheduler — it just re-runs.
5. The DOM effect reads `doubled.value`. It's `_dirty`, so it recomputes (2) and caches. The numbers on screen are updated.

We didn't write a single line of "find the element, set the text" on the click. We only described the dependencies, and the system pulled the change through the chain by itself. That's the whole essence of Vue — and we've just built it.

Check yourself

Run the tests:

```
npm test
```

Fourteen checks in `test/reactivity.test.mjs` confirm every property: `ref` and `reactive` react, `computed` is lazy and caches, `cleanup` unsubscribes from the unneeded, `watch` hands back old and new. Then open the demo:

```
npm run serve
# http://localhost:5173/playground/01-reactivity.html
```

Click the buttons, watch the console to see the effect fire. Once reactivity has "clicked" for you, move on to the next layer — the virtual DOM. There we'll learn to turn data not into `document.title`, but into whole trees of elements, and to update them surgically.

The Virtual DOM and the Renderer

In the last chapter, reactivity updated `document.title` — a single line. A real interface is a tree of hundreds of elements, and rebuilding it whole on every change is too expensive: the browser stutters, input fields lose focus, scroll position jumps. We need a way to touch only what actually changed. That's the renderer's job, and the **virtual DOM** is what makes it possible.

The code for this chapter lives in `packages/runtime-core/` (the platform-independent core) and `packages/runtime-dom/` (the browser binding). Tests are in `test/renderer.test.mjs`, and the demo is `playground/02-vdom.html`.

The idea: describe first, compare second

Touching the page directly (`document.createElement`, `el.appendChild`) is slow and verbose. So we take a smarter route. First we describe what the interface should look like, using plain objects. Such a description object is called a **VNode** (virtual node). Then we hand that description to the renderer, and it figures out which real operations on the page are needed.

The payoff: when the data changes, we build a new description and place it next to the old one. The renderer compares the two descriptions (this is called a **diff**) and applies only the difference to the real page. No "tear it all down and repaint."

VNode: a node as an object

A single VNode looks like this:

```
{
  type,     // 'div' (tag), or Text, Fragment, or a component
  props,    // { id, class, onClick, ... }
  children, // a string, an array of child VNodes, or null
  key,      // key for matching items in lists (more on this below)
```

```
  el,       // reference to the real node — the renderer sets it on mount
}
```

`Text` and `Fragment` are special marker types (`Symbol`) with no tag. `Text` is just text. `Fragment` is a group of nodes without a shared wrapper (for when you need to return several elements without wrapping them in an extra `<div>`). In the renderer a fragment is bracketed by two invisible empty text nodes — a start anchor (stored in `vnode.el`) and an end anchor (`vnode.anchor`) — so the diff always has real nodes to point at: siblings insert before the start anchor, and the fragment's own children live strictly between the two.

Writing these objects by hand is awkward, so there's a function `h` (from "hyperscript"):

```
h('div', { id: 'app' }, 'hello')
h('ul', [ h('li', 'one'), h('li', 'two') ])
```

`h` can tell whether you passed `props` or went straight to children, and either way it assembles a correct VNode. `h` is exactly what our template compiler will call in layer 4 — so `<div>{{ x }}</div>` ends up becoming a call to `h`.

The renderer doesn't know about the browser

The key architectural decision is that the renderer isn't tied to the DOM. It receives every operation on real nodes from the outside, through an `options` object (we call it `nodeOps`): "create an element," "insert," "remove," "change text." The diff algorithm itself is independent of them.

```
export function createRenderer(options) {
  const {
    createElement: hostCreateElement,
    // ...
    insert: hostInsert,
    remove: hostRemove,
    patchProp: hostPatchProp,
    // ...
  } = options
  // ... all of the diff lives inside ...
  return { render, hydrate, createRenderer, patch, __installComponents }
}
```

Besides `render`, the returned object carries `hydrate` and a couple of service hooks — they belong to later layers (7 and 3); ignore them for now.

Why this abstraction? Because the same renderer then works anywhere. For the browser, the operations come from `runtime-dom` (files `nodeOps.js` and `patchProp.js`). For tests, an invented in-memory tree (`test/helpers/testHost.mjs`), where we exercise the diff without any browser. For the server in layer 7, there will be its own implementation. One algorithm, three environments. Real Vue is built exactly this way.

patch: compare old and new

The central function is `patch(n1, n2, container, anchor)`. It compares the old node `n1` and the new node `n2`:

- `n1 === null` — the node appeared for the first time, so we **mount** it;
- `n1.type !== n2.type` — the nodes are incompatible (you can't turn a `div` into a `span`), so we remove the old one and mount the new one from scratch;
- otherwise — the types match, so we **update in place**: the most common and cheapest path.

`anchor` is the node to insert before. It's what lets us land in the right spot among siblings; `null` means "at the end."

From there `patch` branches by the new node's type: text, fragment, element, or component. For an element appearing for the first time, `mountElement` runs:

```
function mountElement(vnode, container, anchor) {
  const { type, props, children } = vnode
  const el = (vnode.el = hostCreateElement(type)) // create and remember the node
  for (const key in props) {
    if (key === 'key') continue // key is diff metadata, not a DOM attribute
    hostPatchProp(el, key, null, props[key])       // set attributes
  }
  hostInsert(el, container, anchor)                // insert into the parent FIRST
  if (typeof children === 'string' || typeof children === 'number') {
    hostSetElementText(el, String(children))       // text content
  } else if (Array.isArray(children)) {
    mountChildren(children, el, null)              // or children recursively
  }
  invokeDirectives(vnode, 'mounted')               // custom directives (layer 8)
}
```

Notice `vnode.el = ...`. We store the reference to the real node right on the VNode. On the next update, the new VNode inherits this `el` — that's how we reuse the existing element instead of creating a new one.

The element goes into the parent *before* its children are mounted: by the time the `mounted` hooks of custom directives run (layer 8), the node is already attached to the document — `el.focus()` on a detached node would be a silent no-op.

Updating in place

When the types match, `patchElement` is called: it takes `el` from the old VNode, compares attributes (`patchProps`) and children (`patchChildren`).

`patchProps` walks the new props (updating changed ones, adding new ones) and the old props (removing ones that disappeared). The detail that's easy to forget is removal: if a node had a `class` and the new description doesn't, the attribute has to come off.

`patchChildren` handles the "was → became" cases. Children can be text, an array, or empty. Most combinations are simple (became text — set the text; was text, became an array — clear it and mount). One case is hard and deserves its own discussion — an array turning into an array.

Comparing lists by key

Picture a list of a thousand rows, and one gets added at the front. The naive approach would compare pairwise: first with first, second with second — and conclude that all thousand changed. A thousand needless edits instead of one insertion.

The rescue is **keys**. We ask you to tag list items with a unique `key` (usually an id from the data). Then the renderer matches nodes by key rather than by position: "the node with key 42 was third, now it's first — don't recreate it, just move it."

The `patchKeyedChildren` algorithm (the same one Vue 3 uses) works in several passes:

1. **From the start**: while keys match, update nodes in place and move forward.
2. **From the end**: the same from the tail of the list. These two passes instantly dispatch the most common cases — items added or removed at the start or the end.
3. If only new nodes remain after that — mount them.
4. If only old nodes remain — unmount them.
5. If the middle is shuffled — build a "key → new index" map, update the matched nodes, remove the extras, and move the rest.

The fifth step hides one more optimization. Not every node needs moving — some are already in the correct relative order. To find the largest such group, we compute the **longest increasing subsequence** (LIS, the `getSequence` function). Nodes in it we leave untouched, moving only the others. That keeps the number of DOM rearrangements minimal.

You can verify this in the test "reversing a list keeps the nodes": we record the real node objects before the rearrangement and confirm that after `[a,b,c] → [c,b,a]` they're the same objects, just reordered. Not one is recreated — which means focus, selection, and any other live state would have survived.

A gotcha: unmount must walk the whole subtree

Our first `unmount` took an element off the page with a single `hostRemove(vnode.el)` — and the DOM did disappear. But everything *inside* went down silently: components nested in that element (layer 3) never fired their unmount hooks and their effects kept running, while a Teleport's children (layer 11) stayed in their remote container forever. The fixed `unmount` always recurses through the children — passing `doRemove: false`, since the descendants' DOM leaves together with the ancestor — so cleanup runs for the whole subtree, and only the redundant per-node removals are skipped.

How this ties into reactivity

The renderer on its own knows nothing about reactivity — it's simply told "draw this." One `effect` connects the two layers. Look at the demo `02-vdom.html`:

```
effect(() => {
  render(view(), app) // view() builds a VNode tree from reactive state
})
```

`view()` reads reactive data (`people.value`), so the `effect` subscribed to it. Change `people` — the `effect` reruns, builds a new tree, and `render` diffs it and applies a targeted edit. Reactivity decides "when to repaint," the renderer decides "how to repaint with minimal work."

This is essentially Vue in miniature already. Only one thing is missing — being able to declare such a "state + view + effect" block as a reusable unit. That unit is called a **component**, and it's the subject of the next layer.

Check yourself

```
npm test          # reactivity and renderer tests
npm run serve     # http://localhost:5173/playground/02-vdom.html
```

The twelve checks in `renderer.test.mjs` cover mounting, attribute updates, all children transitions, and — most importantly — keyed diff in every mode: insertion, removal, reversal, complex rearrangement. In the demo, hit "Shuffle" with the inspector open — you'll see the `<li>` elements move rather than blink back into existence.

Components

By now we have reactivity (layer 1) and the renderer (layer 2). In the layer 2 demo we wired them together by hand with a single `effect`: it built an `h(...)` tree from state and called `render`. That worked, but the block couldn't be reused — state, markup, and the effect were floating free. A component packages exactly this bundle into a self-contained, reusable unit.

Chapter code: `packages/runtime-core/component.js`, `scheduler.js`, `apiLifecycle.js`, `apiInject.js`, `apiCreateApp.js`, `errorHandling.js`. Tests — `test/component.test.mjs`, demo — `playground/03-components.html`.

What a component is

A component is a plain object with a description: where state comes from (`setup`) and how markup is produced from it (`render`).

```
const Counter = {
  props: ['start'],
  setup(props) {
    const count = ref(props.start)
    return { count, inc: () => count.value++ }
  },
  render(ctx) {
    return h('button', { onClick: ctx.inc }, 'Clicks: ' + ctx.count)
  },
}
```

`setup` runs once when the component is created and returns whatever the markup will use. `render` is called every time the component needs to be drawn, and returns a VNode tree. Between them sits `ctx`, through which the markup reaches state. Here's how all of this comes alive.

The instance: a component's private record

The same `Counter` can be placed on a page ten times, and each one has its own counter. So every occurrence needs its own state. That state lives in an **instance** — an object created for each appearance of the component (`createComponentInstance`). It holds everything: parsed `props`, `slots`, the result of `setup` (`setupState`), the last rendered tree (`subTree`), the `isMounted`/`isUnmounted` flags, the effect scope (`scope`) that collects everything reactive the component creates, the `update` function, and a reference to the parent.

setup and the public context

`setupComponent` prepares the instance in three steps: it parses `props`, lays out `slots`, and runs `setup`.

```
let setupResult
try {
  setupResult = instance.scope.run(() => setup(instance.props, setupContext))
} catch (err) {
  handleError(err, instance, 'setup function') // user code may throw
} finally {
  setCurrentInstance(null) // always clear the current instance, even on a throw
}

if (typeof setupResult === 'function') {
  instance.render = setupResult          // setup returned the render function itself
} else if (setupResult && typeof setupResult === 'object') {
  instance.setupState = proxyRefs(setupResult) // the state object
}
```

Two details here. `setup` runs inside the instance's *effect scope*: every watcher and computed created in it lands in `instance.scope`, and unmount will stop them all with one call. And since `setup` is user code, a throw doesn't take the mount down — `handleError` routes it up the `onErrorCaptured` chain to `app.config.errorHandler` (see `errorHandling.js` and the lifecycle section below).

Note `proxyRefs` from layer 1. It wraps the returned object so that `.value` on refs is unwrapped automatically. That's why in `render` you write `ctx.count`, not `ctx.count.value` — `proxyRefs` handles that.

What is `ctx`? It's `instance.ctx` — a Proxy over the instance with access rules (`PublicInstanceHandlers`). When `render` reads `ctx.count`, the proxy looks for `count` first in `setupState`, then in `props`, then among the internal `$`-properties (`$emit`, `$slots`, `$attrs`). A single `ctx` is one access point to everything the markup can see.

One more subtlety: while `setup` runs, we mark the instance as "current" (`setCurrentInstance`). This lets `onMounted`, `provide`, and `inject` called inside `setup` know which component they belong to — without passing the instance as an argument.

Tying into reactivity: render as an effect

Here's the heart of the chapter. How do you make a component re-render itself when state changes? Wrap its rendering in a reactive effect from layer 1.

```
const componentUpdateFn = () => {
  if (instance.isUnmounted) return // a stale queued job must not touch a dead component
  // …
  if (!instance.isMounted) {
    const subTree = (instance.subTree = renderComponentRoot(instance))
    patch(null, subTree, container, anchor) // first time — mount
    instance.vnode.el = subTree.el
    instance.isMounted = true
    pendingPostHooks = instance.m // onMounted — fires later, see below
  } else {
    // …
    const nextTree = renderComponentRoot(instance)
    const prevTree = instance.subTree
    instance.subTree = nextTree
    patch(prevTree, nextTree, container, anchor) // after that — diff against the old tree
    instance.vnode.el = nextTree.el
    pendingPostHooks = instance.u // onUpdated
  }
}
const effect = instance.scope.run(
  () => new ReactiveEffect(componentUpdateFn, () => queueJob(instance.update)),
)
const update = (instance.update = () => {
  effect.run()
  // … the pending onMounted/onUpdated hooks fire here, outside the effect
})
update.id = instance.uid
update.i = instance
update() // first run = mount
```

When `componentUpdateFn` runs `render` for the first time, `render` reads reactive state — and the effect subscribes to it (exactly as in layer 1). State changes — the effect must re-run. But not immediately: its scheduler puts the component into a queue via `queueJob`. Why a queue — that's the next section.

On the first run we mount the subtree (`patch(null, subTree, ...)`); on later runs we compare the new tree against the old one (`patch(prevTree, nextTree, ...)`), and the diff from layer 2 makes the minimal change. So reactivity answers "when to re-render," and the renderer answers "how to re-render cheaply."

Three details are worth pausing on. The `isUnmounted` guard is the last line of defense: an update queued in the same tick the parent removed the component must not touch the DOM. The effect is created inside `instance.scope`, so unmounting stops it together with every watcher from `setup` in one call. And `onMounted`/`onUpdated` fire *after* `effect.run()` returns — inside the effect, reactivity's self-trigger guard would silently swallow any state change a hook makes, and a hook that mutates state could never schedule its follow-up render. The `id` on the job lets the scheduler keep parent-before-child order; `.i` tells it whose job this is, for error routing.

The scheduler: three changes — one re-render

If a handler changes three reactive values in a row, a naive effect would re-render the component three times, even though the user only cares about the final result. That's why component updates go through a queue (`scheduler.js`).

```
export function queueJob(job) {
  // dedupe — but only against the part of the queue that hasn't run yet
  if (!queue.includes(job, isFlushing ? flushIndex + 1 : 0)) {
    if (!isFlushing) {
      queue.push(job)
    } else {
      // … a job arriving mid-flush is inserted by id right after the running one
    }
    queueFlush()
  }
}
function queueFlush() {
  if (!isFlushPending && !isFlushing) {
    isFlushPending = true
    currentFlushPromise = resolvedPromise.then(flushJobs)
  }
}
```

`resolvedPromise.then(...)` defers processing the queue to a microtask — it runs as soon as the current synchronous code finishes. By that point all changes have already happened, and the component is in the queue once — so, one re-render. That's "batching." The test "several changes in a row = one re-render" confirms it. `flushJobs` sorts the queue by ascending job id (a parent has a smaller id than its child, so it updates first and may remove the child before the child's turn) and runs each job in its own try/catch: one component throwing in render doesn't cancel every other update in the tick — the error goes to the `onErrorCaptured` chain from `errorHandling.js`.

`nextTick` comes from the same place. Since the DOM update is deferred, sometimes you need to wait for it — for example, to measure the already-updated element. `await nextTick()` returns control after the queue has been processed and the DOM is current.

A gotcha: a job that re-queues itself

The first version deduplicated with a plain `queue.includes(job)`. Fair enough — until a job re-queues *itself* mid-flush: `onUpdated` changes state, the effect triggers, `queueJob` finds the currently running job still sitting in the array and silently drops the follow-up update forever. That's why the dedupe now searches only the part of the queue that hasn't run yet — `includes(job, flushIndex + 1)`: the running job may re-enter, and a job arriving mid-flush is inserted in id order so parents still update before children.

props: data flowing top-down

The parent passes data to a component through its VNode's attributes: `h(Child, { label: 'hello' })`. The component declares which of those are its `props` — either with a list, `props: ['label']`, or with an object of per-prop options: `props: { step: { type: Number, default: 1, required: false } }` (a bare constructor, `step: Number`, is shorthand for `{ type }`). Everything declared lands in `instance.props`; everything else (for example, a `class` attached from outside) goes into `attrs`.

```
for (const key in raw) {
  if (key === 'key') continue
  if (options.has(key)) props[key] = raw[key]  // a declared prop
  else attrs[key] = raw[key]                    // a "fallthrough" attribute
}
// Object-syntax extras: defaults for absent props, required/type warnings.
for (const [key, opt] of options) {
  if (key in props) {
    validatePropType(key, props[key], opt)
  } else {
    if (opt.required) console.warn(`[minivue] Missing required prop: "${key}"`)
```

```
    if ('default' in opt) props[key] = resolvePropDefault(opt)
  }
}
instance.props = reactive(props) // make props reactive
```

The object syntax is checked in dev style: a wrong type or a missing `required` prop produces a `console.warn`, never a throw. An object or array `default` must be a factory function (`resolvePropDefault` calls it) — otherwise every instance would share one mutable object.

`props` are reactive, and that matters. When the parent re-renders and passes a new value, `updateComponent` calls `updateProps`, which writes the new value into the reactive `props`, and anyone who read that prop in `render`, computed, or watch reacts. That's exactly what happens in the test "parent passes, child updates."

`attrs` don't just sit in the instance either. After every render, `renderComponentRoot` merges them onto the component's single element root — `class` and `style` combine, event handlers both fire, for the rest the outer value wins. That's what makes `h(Button, { class: 'primary' })` work without `Button` declaring `class`; a component opts out with `inheritAttrs: false`, and a fragment root gets nothing (there's nowhere to put them).

emit: events flowing bottom-up

The reverse direction. A component doesn't change someone else's data directly — it "shouts up" about an event, and the parent decides what to do. The component calls `emit('increment')`, and the parent listens for it as `onIncrement`.

```
function emit(instance, event, ...args) {
  const props = instance.vnode.props || {}
  // exact name first, then the camelized form ('value-change' → onValueChange)
  const handler = props[toHandlerKey(event)] || props[toHandlerKey(camelize(event))]
  if (handler) callWithErrorHandling(handler, instance, `"${event}" event handler`, args)
}
```

`emit('ping', 42)` looks in the component's props for a function `onPing` and calls it with `42`. So the child reports and the parent reacts — data flows down, events bubble up. This is the basic contract for component communication. Two touches match real Vue: a kebab-case event (`emit('value-change')`, the template style) is camelized so it still finds `onValueChange`, and the handler runs through `callWithErrorHandling` — a throw in the parent's handler is routed to the error chain instead of unwinding through the child's render that called `emit`.

slots: markup from outside

Sometimes the parent wants to place its own markup inside a component — like content in a card. These "holes" are called slots. A component's children are its slots:

```
h(Card, [ h('span', 'inside the card') ]) // this is the default slot
```

`normalizeSlots` turns children into an object of functions: an array/string becomes the `default` slot, an object `{ header, footer }` becomes named slots. The component renders them via `ctx.$slots.default()`:

```
const Card = {
  render(ctx) {
    return h('div', { class: 'card' }, ctx.$slots.default())
  },
}
```

A slot is a function (not a ready VNode) so that the content is computed at the right moment of rendering, rather than once and for all.

provide / inject: through the floors

Passing data through props is convenient between neighboring levels, but painful when you need to thread it deep down through many intermediate components. `provide` and `inject` bore a "tunnel": an ancestor puts down a value, and any descendant pulls it out directly.

The trick is prototype-based inheritance. Each component's `provides` object inherits the parent's `provides` (`Object.create`). So reading a key naturally walks up the chain of ancestors until it finds a value. The test "provide/inject: the value reaches the grandchild" checks this across two levels of nesting.

Lifecycle

A component goes through stages: created → mounted → updating → unmounted. You can hang your own code on each one with hooks `onMounted`, `onUpdated`, `onUnmounted`, and others. A hook simply adds your function to a list on the current instance, and the renderer, when it reaches the stage, calls the whole list. `onMounted` is handy for fetching data from the server (the element is already on the page), `onUnmounted` — for cleanup (clear a timer, unsubscribe).

Beyond those, `onActivated`/`onDeactivated` fire when `KeepAlive` (layer 11) hides a component instead of destroying it, and `onErrorCaptured` turns a component into an error boundary: an error thrown in a descendant's `setup`, `render`, hooks, or event handlers walks up the chain of `onErrorCaptured`

hooks (returning `false` stops it), then goes to `app.config.errorHandler`, then to `console.error` — the whole route lives in `errorHandling.js`. Hooks are user code too, so `invokeHooks` wraps each in `callWithErrorHandling`: one throwing `onMounted` won't abort the mount or the sibling hooks after it.

createApp: the entry point

All that's left is to assemble everything into an application. `createApp(RootComponent).mount('#app')` wraps the root component in a VNode, attaches the application context to it (for app-level `provide` and plugins), and calls `render`. The `use` method registers plugins — this is exactly how the router and store will slot in during later layers:

```
createApp(App)
  .use(router)   // plugin: router.install(app)
  .use(pinia)
  .mount('#app')
```

A gotcha: moving a keyed component needs its `el`

When a keyed list re-orders, the diff moves nodes by inserting `vnode.el` before an anchor. A component's re-render is *queued*, so `updateComponent` must copy the DOM node over synchronously — `n2.el = n1.el` — the moment it receives the new vnode. Skip that one line and a reordered list of components (say, sorting a product grid) throws `insertBefore ... is not of type Node`, because the moved vnode has no `el` yet. Plain elements never hit this — their `el` is assigned during patch — which is why the bug only surfaces once you build a keyed list of *components*.

Check yourself

```
npm test        # among others — 10 component tests
npm run serve   # http://localhost:5173/playground/03-components.html
```

The demo is a todo list: the parent holds the state, the child `TodoItem` receives a task through props and sends `toggle`/`remove` via emit, and the list of components uses the keyed diff from layer 2. All of it without a single template — pure `render` and `h`. Templates (`<div>{{ x }}</div>`) are the topic of the next layer: we'll write a compiler that turns familiar markup into exactly these `h` calls.

The Template Compiler

So far we've written markup with a `render` function and calls to `h`. That works, but it's clunky: compare `h('li', { key: t.id }, t.text)` with the familiar `<li :key="t.id">{{ t.text }}</li>`. Templates read like HTML, which makes them clearer. The compiler is a translator: it turns a template string into exactly that render function full of `h` calls. No new runtime magic appears — just convenient syntax on top.

Chapter code: `packages/compiler/`. Tests — `test/compiler.test.mjs`, demo — `playground/04-compiler.html`.

Three translation steps

The compiler works in three passes, and this is the classic scheme of any compiler:

1. **Parsing** (`parse.js`): string → tree of objects (AST). The text «`<p>{{ msg }}</p>`» turns into the structure «element p, containing interpolation msg».
2. **Transformation**: turn raw attributes into meaningful directives (`v-if`, `v-for`, `:bind`, `@on`). We fold this step into generation for simplicity.
3. **Generation** (`compile.js`): walk the tree and build a string of code `h('p', null, [_s(msg)])`, then turn that string into a real function.

The output is a function indistinguishable from one you'd write by hand. The runtime from layers 1–3 knows nothing about the compiler: it doesn't care where the render function came from.

Parsing: from text to tree

A computer doesn't understand text structurally — to it, «`<div>`» is just five characters. The parser walks the string left to right and, recognizing familiar shapes (tag, text, `{{`), builds a tree of nodes. We write it as a «recursive descent»: for nested tags the function calls itself.

```
function parseChildren(context) {
  const nodes = []
```

```
  while (!isEnd(context)) {
    // Remember how much input is left: every loop iteration MUST consume at
    // least one character, otherwise we would spin forever on the same spot.
    const lengthBefore = context.source.length

    const s = context.source
    let node = null
    if (s.startsWith('<!--')) {
      parseComment(context) // comments produce no output — just skip them
    } else if (s.startsWith('{{')) {
      node = parseInterpolation(context)
    } else if (s[0] === '<' && /[a-zA-Z]/.test(s[1])) {
      node = parseElement(context)
    } else {
      // Everything else is text — including a lone '<' that does not start a
      // tag (e.g. "5 < 10" or "i <3 vue").
      node = parseText(context)
    }
    if (node) nodes.push(node)

    // The progress guarantee. If a branch above ever consumes nothing, we fail
    // loudly instead of hanging the whole program in an infinite loop.
    if (context.source.length === lengthBefore) {
      parseError(context, 'Parser made no progress — cannot understand this input')
    }
  }
  return condenseWhitespace(nodes)
}
```

`context.source` is the remaining, not-yet-parsed part of the string. Each function «bites off» the recognized chunk from its start (`advanceBy`). `parseElement` reads the tag and attributes, then recursively calls `parseChildren` for the contents until it hits the closing `</tag>`. That's how a tree grows out of a flat string.

Two chores hide at the edges of the loop. Comments `<!-- -->` are recognized and skipped — they produce no node at all. And `condenseWhitespace` cleans up at the end: a whitespace-only text node with a newline in it is code formatting and gets dropped, while a deliberate space between inline elements survives as a single `' '`. Text and attribute values also decode HTML entities (`&` → `&`) — our templates never pass through the browser's parser, so nobody else would do it.

Nodes come in three types: `Element` (a tag with attributes and children), `Text` (plain text), and `Interpolation` (an insertion `{{ expression }}`). The parser collects attributes as raw `{ name, value }` pairs without probing their meaning — figuring out that `v-if` is a condition and `@click` is a handler is the next step's job.

A gotcha: a parser must always move forward

The progress check in `parseChildren` is a scar from a real bug. The original loop had no such guarantee, and a template with an innocent `<!-- -->` froze the page: there was no comment branch, `parseText` stops at every `<`, so an iteration consumed zero characters — and the `while` loop asked about the same spot forever. The fixed parser makes it a rule: every branch must consume at least one character, and an iteration that eats nothing is a loud compile error, not a silent hang.

Generation: from tree to code

Now we build a string of code from the tree. Each node type gets its own rule:

```
function genNode(node) {
  switch (node.type) {
    case 'Element':       return genElement(node)
    case 'Text':          return JSON.stringify(node.content)  // "hello"
    case 'Interpolation': return `_s(${node.content})`         // _s(msg)
  }
}
```

Text becomes a string literal (`JSON.stringify` escapes quotes along the way). The insertion `{{ msg }}` becomes `_s(msg)` — a call to the helper `_s`, which coerces the value to a string. An element becomes `h('tag', props, children)`, where `props` and `children` are generated recursively from the attributes and nested nodes.

For `<div>hello</div>` you get exactly `h("div", null, ["hello"])` — what you'd write yourself. You can confirm this in the test `codegen: element with text`, or right in the demo, where the generated code for the app's template is shown under the app.

Directives

The most interesting part is translating directives. When generating `props`, we classify each attribute:

- a plain `class="btn"` → a static pair `"class": "btn"`;
- `:id="bid"` (shorthand for `v-bind`) → `"id": (bid)` — the value is computed as an expression;

- `@click="inc"` (shorthand for `v-on`) → `"onClick": (inc)` — the event name gets an `on` prefix and a capital letter, and the runtime from layer 2 already knows how to attach such handlers.

Handlers have a subtlety. `@click="inc"` is a reference to a method, and we leave it as is: `(inc)`. But `@click="count++"` is an action expression, and it must be wrapped, otherwise `count++` runs once at generation time rather than on click:

```
function genHandler(on) {
  const exp = on.exp
  // ...
  const isMethodPath = /^[A-Za-z_$][\w$.]*$/.test(exp.trim())

  if (mods.length === 0) {
    return isMethodPath ? `(${exp})` : `$event => (${exp})`
  }
  // ... event modifiers (@click.stop) — that story comes in the forms chapter
}
```

`v-if` and `v-for` are «structural» directives: they control whether nodes appear at all, not their attributes. `v-for="item in list"` wraps the node in the helper `_l` (render list), which iterates over the list and returns an array of nodes:

```
// <li v-for="item in items">{{ item }}</li>  turns into:
h(Fragment, null, [..._l(items, (item) => h("li", null, [_s(item)]))])
```

An array is not a vnode, and `render` must return exactly one — so when `v-for` sits on the root element (perfectly legal, as here), the generator spreads the array into an invisible `Fragment`. Inside a parent element the `_l(...)` call is spread into that parent's children the same way.

`v-if` is generated at the level of the children list, because it needs access to a neighboring `v-else`. The chain `v-if / v-else-if / v-else` is assembled into a ternary operator:

```
// <span v-if="ok">yes</span><span v-else>no</span>  →
(ok) ? h("span", null, ["yes"]) : h("span", null, ["no"])
```

With no `v-else`, the branch becomes `: null`, meaning «render nothing».

From string to function: with(ctx)

The final trick is turning the generated string into a real function. Here a question comes up: in the code `h("p", null, [_s(msg)])`, where does `msg` come from? It has to come from the component's state, from `ctx`. Dragging `ctx.` in front of every name in the generator is a chore. Instead, we wrap the body in `with(ctx)`:

```
factory = new Function(
  'h',
  'Fragment',
  '_s',
  '_l',
  // ... plus the helpers of later chapters: _c, _key, _cd, _wd, _dir, _m, _th
  `return function render(ctx){ with(ctx){ return ${code} } }`,
)
```

`with(ctx)` forces every name inside to be looked up in `ctx` first. So `msg` resolves as `ctx.msg`, and `count` as `ctx.count`, without a single `ctx.` in the code. The helpers — `h`, `_s`, `_l` and, as the engine grows, seven more — come in as parameters of the outer factory function, bypassing `ctx`. One more real-world touch: the `new Function` call is wrapped in `try/catch` — a syntax error here means a broken expression *from the template* (say, `@click="count +"`), so the rethrown error quotes the generated code instead of pointing nowhere.

`with` is a rare and usually undesirable JavaScript operator, but here it fits, and Vue's real runtime compiler uses the same trick (`with(this)`). For `with` to decide correctly which name comes from `ctx` and which comes from the outer scope, we added a `has` trap to the component context proxy (layer 3). It answers «yes» for state and props, «no» for the compiler's helpers and a small whitelist of safe globals (`Math`, `Date`, `JSON`...), and claims every *other* unknown name for itself: the template then sees `undefined` and gets a one-time warning — instead of a typo falling through to `window` or exploding as a raw `ReferenceError` mid-render.

How the compiler hooks into the runtime

The runtime doesn't depend on the compiler directly — otherwise you couldn't build a «lightweight» version without it. The link is established through registration: importing `packages/compiler/index.js` calls `registerRuntimeCompiler(compile)`, and from that point on a component with a `template` property is compiled automatically at initialization (`finishComponentSetup` in layer 3). The full build `packages/minivue.js` does this import for you — the way the main `vue` package bundles both the runtime and the compiler. Compiling isn't free, so `finishComponentSetup` caches the result per component definition in a `WeakMap`: a list of a thousand identical components compiles its template once, not a thousand times.

What we simplified

Vue's real compiler does far more: static analysis and hoisting of unchanging nodes, patch flags to speed up diffing, handler caching, slots, compilation in a separate build step. (`v-model` and event

modifiers like `@click.stop`, still missing here, will be added in the forms chapter.) One shortcut is architectural: `compile.js` imports `h` and `Fragment` straight from runtime-core — a layer inversion compared to real Vue, where the compiler emits code that only *names* its helpers and the runtime binds them when it registers the compiler. Binding the helpers on the spot keeps the whole template→function story in one readable file, at the price of a compiler that cannot live without runtime-core; Vue pays the opposite price — an extra indirection layer — to ship a runtime-only build without the compiler and a compiler that runs at build time without the runtime. We took the core idea — parsing, generation, `with(ctx)` — and the most needed directives. That's enough to understand how `<div>{{ x }}</div>` becomes a working interface, and to write real applications with templates.

Check yourself

```
npm test        # among others — 10 compiler tests
npm run serve   # http://localhost:5173/playground/04-compiler.html
```

The tests check both the generated code (`compileToString`) and the full cycle «template → DOM»: interpolation, reactive updates, `@click`, `v-if/v-else`, `v-for`. In the demo the todo list is now written as a template, and at the bottom of the page you can see the code the compiler turned it into. Next up — the router: we'll teach the app to show different pages based on the browser's address.

The Router

A regular website asks the server for a fresh page on every navigation and reloads it. A single-page application (SPA) doesn't work that way: it loads once, then swaps its own content by looking at the URL in the browser's address bar. The router handles that. Ours is a scaled-down copy of Vue Router, and it rests entirely on the reactivity from layer 1.

Chapter code: `packages/router/`. Tests — `test/router.test.mjs`, demo — `playground/05-router.html`.

The idea: URL → component

The router solves one problem: pick which component to show based on the current URL. `/` — the `Home` component, `/about` — `About`, `/user/42` — `User` with the parameter `id = 42`. The chosen component is shown in a special place — `<RouterView>`. `<RouterLink>` links change the URL without reloading the page.

The whole trick is that the "current route" is a reactive object. `<RouterView>` reads it in its render function, so by the rules of layer 1 it subscribes to it. The URL changes — the route changes — `<RouterView>` re-renders with the new component. The router needs no special reactivity mechanism of its own; it reuses ours.

history: where the URL lives

The URL can be stored in different ways, depending on the environment. So the mechanism is hidden behind a `history` object with a single interface: `location` (the current path), `push` and `replace` (change it), `go` (walk the history stack), `listen` (subscribe to changes), `destroy` (remove the listeners). There are three implementations:

- **`createWebHistory`** — normal URLs like `/about` via the History API (`pushState`). Clean, but the server must serve `index.html` for any path.

- **`createWebHashHistory`** — URLs with a hash, `/#/about`. The part after `#` never reaches the server, so it works on any static host with no configuration.
- **`createMemoryHistory`** — the URL lives in a plain variable, with no `window`. Needed for tests and for the server (layer 7), where there's no browser.

The router doesn't care which one is plugged in — it talks through the same interface. Tests use `createMemoryHistory`, the demo uses `createWebHashHistory`.

The matcher: parsing paths with parameters

The route `/user/:id` should match the URL `/user/42` and produce `id = 42`. To do that, we turn the route's path into a regular expression, and the `:parameters` into capture groups:

```
function compileRoute(record) {
  const keys = []
  let score = 0 // each static segment adds a point — the most specific route wins
  const segments = record.path.split('/').filter(Boolean)
  const pattern = segments
    .map((segment) => {
      if (segment[0] === ':') {
        keys.push(segment.slice(1)) // remember the parameter name
        return '/([^/]+)'           // and match "any segment without a slash"
      }
      score++
      return '/' + escapeRegExp(segment) // static segment, regex chars escaped
    })
    .join('')
  const regex = new RegExp('^' + (pattern || '/') + '/?$') // '/?$' tolerates a trailing slash
  return { ...record, regex, keys, score }
}
```

`resolve(to)` first splits the hash and the query string off the URL (`?q=vue` becomes the object `{ q: 'vue' }` via `parseQuery`), then runs the path against every record and keeps the *most specific* match — the `score` is what stops `/user/:id` from shadowing `/user/new` just because it was declared first. For the winner, `params` are collected from the capture groups using the saved names. If nothing matches, an empty route is returned, and `<RouterView>` shows nothing.

The reactive current route

The heart of the router is the reactive object `currentRoute`:

```
const currentRoute = reactive({
  path: '/',
  fullPath: '/',
  query: {},
  hash: '',
  params: {},
  matched: [], // list of matching records
})

function applyRoute(route) {
  currentRoute.path = route.path
  currentRoute.fullPath = route.fullPath
  currentRoute.query = route.query
  currentRoute.hash = route.hash
  currentRoute.params = route.params
  currentRoute.matched = route.matched
}
```

`applyRoute` writes the resolved route into the reactive object — and that's the only thing needed to update the UI. Everything that read `currentRoute` reacts. The router also subscribes to the history: an external URL change (the browser's back/forward buttons) arrives as `navigate(path, 'pop')` — the same pipeline as our own `push`, guards included — and ends in the same re-render.

Navigation and guards

`push` doesn't change the URL right away — first it runs the navigation hooks (`beforeEach`). A hook receives where we're going and where we're coming from, and it can allow the transition, cancel it (by returning `false`), redirect (by returning a different path) — or return a Promise of any of these: async guards are awaited, and if a newer navigation starts during the wait, the stale one is abandoned:

```
async function navigate(to, mode, redirectDepth = 0, id = ++navigationId) {
  const toRoute = resolve(to)
  // …
  for (const guard of guards) {
    let result = guard(toRoute, currentRoute)
    if (result && typeof result.then === 'function') {
      result = await result // an async guard
      if (id !== navigationId) return false // a newer navigation won — abandon this one
    }
```

```
    if (result === false) {
      // On Back/Forward the browser has already moved the URL — restore it.
      if (mode === 'pop') history.replace(currentRoute.fullPath)
      // …
      return false
    }
    if (typeof result === 'string') {
      // Redirect — with a cap, so guards redirecting to each other can't loop forever.
      if (redirectDepth >= MAX_REDIRECTS) {
        // …
        return false
      }
      // …
      const nextMode = mode === 'push' ? 'push' : 'replace'
      return navigate(result, nextMode, redirectDepth + 1, id)
    }
  }
  applyRoute(toRoute) // the route first — and only now the URL
  if (mode === 'push' || mode === 'replace') history[mode](toRoute.fullPath)
  return true
}
```

`mode` says who started the navigation: `push`/`replace` come from code and write the URL on success; `pop` means the URL has already changed (browser back/forward) and only needs validating — with a rollback on cancel; `init` validates the initial URL when the app installs, so an auth guard fires even on a bookmarked private page. `push` returns a Promise resolving to `true` (navigated) or `false` (cancelled, superseded, or too many redirects — the chain is capped at `MAX_REDIRECTS = 10`, as in vue-router).

This is how you protect pages: “don’t let an unauthenticated user into `/admin`, send them to `/login`”. The tests cover both branches — cancellation and redirect.

RouterView and RouterLink

`<RouterView>` is a component of a few lines. It injects the current route and in render returns the component from `matched`:

```
const RouterView = {
  setup() {
    const route = inject(ROUTE_KEY)
```

```
    return () => {
      const matched = route.matched[0]
      return matched ? h(matched.component) : null
    }
  },
}
```

The key point is `return () => ...`. We return a render function from `setup`, and it reads `route.matched` on every call. So the component's effect is subscribed to the route, and on navigation `<RouterView>` re-renders itself.

`<RouterLink>` renders an ordinary `<a>`, but it intercepts the click: it cancels the browser's default navigation (`preventDefault`) and calls `router.push` — so the URL changes without a reload. It also compares its target with the current route and puts vue-router's classes on the `<a>`: `router-link-exact-active` when the paths match exactly, `router-link-active` when the current path lives "under" the link (`/user` is active while you're on `/user/42`).

Wiring into the app

The router is a plugin. `app.use(router)` calls its `install`, where the router hands itself and the route out through `provide`, registers `<RouterView>` / `<RouterLink>` as global components, and puts `$router` / `$route` into the global properties:

```
install(app) {
  app.provide(ROUTER_KEY, router)
  app.provide(ROUTE_KEY, currentRoute)
  app.component('RouterView', RouterView)
  app.component('router-view', RouterView)
  app.component('RouterLink', RouterLink)
  app.component('router-link', RouterLink)
  app.config.globalProperties.$router = router
  app.config.globalProperties.$route = currentRoute
  validateInitialNavigation() // the initial URL must pass the guards too
}
```

For `<RouterView>` in a template to turn into a component rather than an unknown HTML tag, the compiler (layer 4) got one extra ability: tags starting with an uppercase letter or containing a hyphen are generated via `_c('RouterView')` — resolving a component by name among the registered ones. Inside `setup`, the same router and route are reached with the `useRouter()` and `useRoute()` hooks.

What we simplified

The real Vue Router does far more: nested routes (several `<RouterView>` deep), named routes and views, lazy-loaded components, per-route guards (`beforeEnter`) and `afterEach` hooks, the `next()`-callback guard style, repeated query keys collected into arrays, scroll behavior, meta fields, and transition animations. We took the skeleton — history, a matcher with parameters and specificity, a reactive route with `query` and `hash`, `RouterView`/`RouterLink` with active classes, and `beforeEach` guards (sync and async, with a capped redirect chain) — which is enough to understand the principle and build a working multi-page application.

Check yourself

```
npm test        # among other things — 6 router tests
npm run serve   # http://localhost:5173/playground/05-router.html
```

The tests run navigation against the in-memory history: the starting route, `push`, `:id` parameters, an empty route, and both kinds of guards. The demo is an app with a menu of `RouterLink`s and a user page that reads `:id`. Next up — the store: shared application state available to any component without passing it through props.

The Store

A component's state lives inside it, and that's correct: a button's counter shouldn't leak outside. But part of the state is shared across the whole application — the cart, the current user, the color theme. Threading it through props and events across the entire component tree is tedious and fragile. The store is a separate reactive storage that any component talks to directly. Our store is a scaled-down copy of Pinia.

Chapter code: `packages/store/`. Tests — `test/store.test.mjs`, demo — `playground/06-store.html`.

The store is reactivity again

The core idea: the store has no new mechanism. Its state is `reactive`/`ref` from layer 1, its computed fields are `computed`, its actions are plain functions that mutate state. A component that reads the store in `render` subscribes to it exactly as it does to its own `ref`. When the store changes, every component that read it re-renders, wherever in the tree it sits. The store just gives this shared reactive state a name and convenient access.

createPinia: the store container

`createPinia()` creates a container that plugs into the application as a plugin (`app.use(createPinia())`). Inside it holds a map of ready stores, shared reactive state, and a list of plugins:

```
export function createPinia() {
  const pinia = {
    _stores: new Map(), // id → ready store (created once, lazily)
    _plugins: [], // extensions (see use below)
    state: reactive({}), // shared state: state[id] = state of store id

    use(plugin) {
      pinia._plugins.push(plugin)
      return pinia
```

```
    },

    install(app) {
      setActivePinia(pinia)
      app.provide(PINIA_KEY, pinia)
      app.config.globalProperties.$pinia = pinia
    },
  }
  return pinia
}
```

`install` records the container as the "active" one and hands it out through `provide` — so any component can reach it.

defineStore: two styles

`defineStore(id, ...)` declares a store. As in Vue, two styles are supported — Options and Setup — and both collapse to the same thing inside.

Options — familiar and declarative:

```
const useCounter = defineStore('counter', {
  state: () => ({ count: 0 }),
  getters: { double: (s) => s.count * 2 },
  actions: { inc() { this.count++ } },
})
```

Setup — flexible, built on the Composition API:

```
const useCart = defineStore('cart', () => {
  const items = ref([])
  const total = computed(() => items.value.reduce((s, p) => s + p.price, 0))
  const add = (p) => (items.value = [...items.value, p])
  return { items, total, add }
})
```

`defineStore` returns not the store itself but a `useStore()` function. On the first call it creates the store; on later calls it hands back the same instance. The store is a singleton per application:

```
function useStore() {
  const pinia = getActivePinia()
  // …
```

```
  if (!pinia._stores.has(id)) {
    createStore(id, setupOrOptions, pinia)
  }
  return pinia._stores.get(id)
}
```

The "store is a singleton" test checks this: two calls to `useCounter()` return the same object, and a change through one is visible in the other.

One creation path

To avoid writing the logic twice, we turn an options store into a setup function (`optionsToSetup`) — and from there both styles follow the same road:

```
function setup() {
  const state = reactive(options.state ? options.state() : {})
  pinia.state[id] = state
  const parts = {}
  for (const key in state) parts[key] = toRef(state, key)          // state → refs
  for (const name in options.getters) parts[name] = computed(...)  // getters → computed
  for (const name in options.actions) parts[name] = (...a) => ...  // actions → functions
  return parts
}
```

The result is a `parts` object of refs, computeds, and functions. All that's left is to make it ergonomic.

proxyRefs: why store.count, not store.count.value

`parts` holds refs, which means from the outside you'd have to write `store.count.value`. Ugly. `proxyRefs` from layer 1 saves us — it wraps the object so that on read a ref unwraps automatically, and on write the value goes into `.value`:

```
store = proxyRefs(parts)
```

Now `store.count` returns a number, `store.count = 5` writes to state, and `store.inc()` calls the function. `proxyRefs` leaves functions alone — they pass through as is. One line turns a bag of refs into an ergonomic store.

Inside options actions `this` points at the store, so `this.count++` reads and writes state through the same `proxyRefs`, staying reactive. The "effect catches changes" test confirms it: a subscription to `store.count` fires on every `inc()`.

storeToRefs: destructuring without losing reactivity

It's tempting to write `const { count, double } = store`, but that loses reactivity — `count` becomes a plain number at the moment of destructuring. For state and getters there's `storeToRefs`: it hands them back as refs that keep their link to the store. Actions you take straight from the store (they're safe to destructure — they're functions):

```
const { count, double } = storeToRefs(store) // reactive refs
const { inc } = store                        // actions — as is
```

This works through `toRef(store, key)` from layer 1: the resulting ref reads and writes `store[key]`, which means it stays part of the reactive system.

Plugins

`pinia.use(plugin)` adds an extension that runs when each store is created and receives `{ store, pinia, id }`. Whatever the plugin returns is mixed into the store. That's how cross-cutting features are built: persisting a store to `localStorage`, logging actions, adding shared fields. For us it's a few lines in `createStore`.

What we simplified

Real Pinia does more: `$onAction` to intercept actions, hot module replacement during development, devtools integration, and type safety through TypeScript. The service methods, though, are all here: `$patch` applies a batched change (an object or a function), `$subscribe` fires one batched callback per change with mutation info (`{ type, storeId }`), `$state` of an options store can be read and even assigned — the assignment patches into the existing reactive object instead of replacing it, and `$reset` restores an options store to its initial state (a setup store throws, exactly like Pinia). We took the essence — reactive singleton state, getters on `computed`, actions, both declaration styles, `storeToRefs`, and plugins. That's enough to grasp the idea and share state between any components.

Check yourself

```
npm test        # among others — 7 store tests
npm run serve   # http://localhost:5173/playground/06-store.html
```

The tests cover both styles, reactivity of state and getters, singleton behavior, `storeToRefs`, and the store working inside a component. The demo is a mini-shop: the "header" and the "catalog" are two independent components, but they share the cart through the store, without a single prop between

them. One last layer remains — SSR: we'll learn to render the application to HTML on the server and "hydrate" it in the browser.

SSR and Hydration

Until now the app was born in the browser: JavaScript loaded, it built the DOM, and only then did the user see any content. That has two problems. First, a blank screen while the code loads and runs. Second, search crawlers often see an empty page. The fix is to draw the app ahead of time, on the server, and ship ready-made HTML. That is server-side rendering (SSR). And to make the shipped HTML come alive and become interactive, the client "wakes it up" — that is hydration. Both mechanisms are the topic of this final layer.

Code for this chapter: `packages/server-renderer/` (the server) and the `hydrate` function in `packages/runtime-core/renderer.js` (the client). Tests are in `test/ssr.test.mjs`; the live example is `playground/07-ssr/` (a Node server).

Rendering to a string

The key observation: a VNode tree doesn't have to turn into DOM specifically. It's just as easy to turn it into text — an HTML string. No browser is needed for that, only a walk over the tree and some string concatenation. That's the job of `renderToString`:

```
function renderVNode(vnode, parentComponent) {
  vnode = normalizeVNode(vnode)
  const { type } = vnode

  if (type === Text) {
    return escapeHtml(vnode.children)
  }
  if (type === Fragment) {
    return renderChildren(vnode.children, parentComponent)
  }
  if (typeof type === 'string') {
    return renderElement(vnode, parentComponent)
  }
```

```
  if (typeof type === 'object' || typeof type === 'function') {
    // component: create an instance, run setup, serialize the subtree
    const { instance, subTree } = createSSRComponent(vnode, parentComponent)
    // async setup() returns a Promise — a synchronous renderer must fail loudly
    if (instance.setupState && typeof instance.setupState.then === 'function') {
      throw new Error('async setup() is not supported by this renderToString')
    }
    return renderVNode(subTree, instance)
  }
  return ''
}
```

A text node becomes escaped text, an element becomes the string `<tag attributes>children</tag>` (void tags like `<img>` or `<input>` — the full HTML set of fourteen — get no closing tag), a component becomes the result of its `render`. For a component on the server we create an instance and run `setup` (`createSSRComponent`), but we don't set up a reactive effect: the server has nothing to re-render, it needs a single snapshot. And if `setup` returned a Promise (async `setup`), our synchronous renderer can't await it — instead of silently serializing `undefined` everywhere the state should be, it fails loudly with an explicit error.

Three details matter for correct HTML. First, **event handlers aren't serialized**: `onClick` makes no sense in HTML, the handler will be attached on the client. Second, **escaping**: user data goes through `escapeHtml`, otherwise the string `<script>` inside text would turn into executable code (XSS). The test "escaping against XSS" checks exactly this. Third, **attribute names**: a value can be escaped, a name cannot — HTML simply has no escape syntax there — so a name containing spaces, quotes, `<`, `>`, `/` or `=` would break out of its position and inject markup. Like Vue's `isSSRSafeAttrName`, we refuse to render such names.

createSSRApp

Just as the browser has `createApp`, the server has `createSSRApp`: the same interface with `use`, `provide`, `component`, but instead of `mount` there's a `renderToString` method that returns HTML. Plugins (router, store) are wired up the same way through `use`, so server rendering of a full app looks almost like the client version — that's the point of "isomorphism": one codebase for two environments.

The whole server

For every page request the server (`playground/07-ssr/server.js`) does three things:

```
const appHtml = renderToString(createVNode(App)) // 1. draw the app into a string
res.end(page(appHtml))                           // 2. embed it into the HTML shell
// 3. the shell contains the <script> for client hydration
```

The shell is a plain HTML page where `<div id="app">` already holds the finished markup, followed by the client script. The user sees the content instantly, before a single line of JavaScript runs. Open "View Source" and you'll see the counter already drawn there — that's SSR.

Notice: `App` is the very same module for server and client. The server imports it in Node, the client imports it in the browser. This is an isomorphic app.

The problem: HTML without life

The HTML the server ships is static. Buttons are drawn but don't react to clicks — they have no handlers (we didn't serialize them). The naive fix is to just run a normal `mount` on the client. But that would tear down all the server HTML and create an identical copy from scratch: wasted work, flicker, lost state (cursor position in an input, scroll offset). The right approach is not to create but to "adopt" the nodes that are already there. That's hydration.

hydrate: adopting the existing DOM

`hydrate` walks the VNode tree in parallel with the existing DOM. For each node it doesn't create a new one — it links the VNode to the real node it finds (`vnode.el = node`) and adds whatever the HTML lacks, above all event handlers:

```
function hydrateNode(node, vnode, container) {
  vnode = normalizeVNode(vnode)
  const { type } = vnode
  const parent = (node && hostParentNode(node)) || container

  if (type === Text) {
    // … mismatch checks; split a browser-merged text node if needed (see below)
    vnode.el = node
    return hostNextSibling(node)
  }
  // … (Fragment: insert the invisible anchors and hydrate the children in place)
  if (typeof type === 'string') {
    // … tag mismatch check → warn and fall back to a client render of this subtree
    vnode.el = node                              // adopt the existing element
```

```
    for (const key in vnode.props) {
      if (key !== 'key') hostPatchProp(node, key, null, vnode.props[key]) // attach events
    }
    if (Array.isArray(vnode.children)) {
      let cur = node.firstChild
      for (let i = 0; i < vnode.children.length; i++) {
        const child = (vnode.children[i] = normalizeVNode(vnode.children[i]))
        cur = hydrateNode(cur, child, node)      // recurse into children
      }
    }
    return hostNextSibling(node)
  }
  // component — adopted "on top of" the node, with a reactive effect set up
  hydrateComponentImpl(vnode, node, parent)
  return getNextHostNode(vnode)
}
```

For a component, hydration does the same thing as mounting, except the first render pass doesn't create the subtree through `patch(null, ...)` — it adopts the existing DOM through `hydrateNode`. A normal reactive effect is set up in the process — so from then on the component lives as usual: state changes, `patch` runs, but now over the adopted nodes.

And when the server DOM and the client vnode genuinely disagree — a different tag, different text — hydration doesn't crash: `handleHydrationMismatch` warns and falls back to a client-side render of that subtree, mounting the vnode fresh in place of the stale node (for text, it trusts the client and overwrites).

The test "hydrate adopts the DOM, attaches events, and updates" checks all of this strictly: the server-side button node and the client-side one after hydration are the same object (`Object.is`), meaning the node wasn't recreated; a handler appeared on it; and a click changes state and patches that same button. That's exactly how, after hydration, the static server HTML becomes a full-fledged SPA without a single needless recreation.

A gotcha: the browser merges adjacent text nodes

The server writes `<button>Clicks: 0</button>` as two text chunks — the static `Clicks:` and the dynamic `0` — but the HTML parser glues them into ONE DOM text node, while the client vdom still holds two text vnodes. Naive hydration adopted the merged node for the first vnode, asked for `nextSibling`, and walked off the real DOM — everything after that point was mismatched. The fix in `hydrateNode`:

if the DOM text *starts with* the vnode's text, `splitText` cuts off the adopted half and leaves the remainder for the next vnode.

What we simplified

Real SSR in Vue is more involved in the details: it inserts comment nodes as anchors for fragments and portals, serializes and transfers store state to the client (`window.__INITIAL_STATE__`), supports streaming rendering, async `setup` and `Suspense` (ours throws an explicit error instead of silently serializing a Promise), and fragment caching; its mismatch handling is also far subtler than our warn-and-rerender fallback. We built the skeleton — rendering a tree to a string with escaping, an isomorphic server, and hydration that adopts nodes, attaches events, survives browser-merged text nodes, and recovers from mismatches — which is enough to understand how SSR works and to build a working example.

Check yourself

```
npm test                          # among others — 7 tests for SSR and hydration
node playground/07-ssr/server.js  # then open http://localhost:5174/
```

The tests cover `renderToString` (attributes, absence of events, escaping, nested components, void tags, `createSSRApp`) and the full hydration cycle. In the Node example the server ships ready-made HTML and the client wakes it up — open the page source to see the server markup, and click the buttons to confirm it came alive.

Wrap-up

We've gone through all seven layers and built a working analog of Vue from scratch: reactivity, a virtual DOM with fine-grained diffing, components, a template compiler, a router, a store, and SSR. None of these mechanisms should be a black box to you anymore — you wrote each one yourself. Now, when you open the real Vue, you'll see not magic but familiar ideas, just polished to production quality. And that was the goal all along: to understand how it works, from the inside.

Forms and Bindings

The core of the framework is done, but the first real form runs into three annoyances. An input field has to be both read and written by hand. Classes want to be toggled by condition, not glued together as a string. And an event handler constantly needs `preventDefault` and a check for which key was pressed. Vue covers this with three conveniences: `v-model`, object bindings for `class`/`style`, and event modifiers. All three are layers on top of what already exists, and they cost little to add.

Chapter code: `packages/shared.js` (class/style normalization), changes in `packages/compiler/compile.js` (`v-model`, modifiers) and `packages/runtime-dom/patchProp.js`. Tests — `test/forms.test.mjs`, demo — `playground/08-forms.html`.

class and style bindings

In layer 2, `class` could only be a string. An object ("class → is it on?") or an array is more convenient:

```
<span :class="{ active: isActive, done: isDone }" />
<span :class="['btn', isPrimary && 'btn-primary']" />
<div :style="{ color: 'red', fontSize: size + 'px' }" />
```

So that both the browser and SSR read every form the same way, we reduce it to a canonical shape in `shared.js`. `normalizeClass` turns an object/array into a string, `normalizeStyle` into an object, and `styleToString` into a string with kebab-case properties:

```
normalizeClass({ active: true, off: false })  // 'active'
normalizeClass(['a', false, 'b'])              // 'a b'
styleToString({ color: 'red', fontSize: '14px' }) // 'color:red;font-size:14px'
```

These helpers are called in two places: in the browser `patchProp` (when setting `class`/`style` on a real element) and in the server-side `renderAttrs`. One piece of logic for both paths. `fontSize` becomes `font-size` automatically, so a style object can use the familiar JS property names.

v-model: two-way binding

An input field is a two-way street: state should show up in the field, and the user's input should update state. By hand that's two things at once:

```
<input :value="name" @input="name = $event.target.value" />
```

`v-model` is sugar over exactly this pair. The compiler expands `<input v-model="name">` into a value binding plus a handler. Look at the implementation in `applyVModel`: it picks the property and the event based on the field type, because fields differ:

- text `input` and `textarea` → `:value` + `@input`;
- `checkbox` → `:checked` + `@change` (and we read `$event.target.checked`);
- `radio` → `:checked` + `@change`, where "checked" means "the model equals *this* input's own value" — writing `:value` here, like the text codegen does, would clobber the value that identifies the button;
- `select` → `:value` + `@change`.

```
// The value from the event, honoring the .trim / .number modifiers.
// Both may be present at once: trim first, then convert (like Vue).
let valueExpr = `$event.target.${field}`
if (mods.includes('trim')) valueExpr = `${valueExpr}.trim()`
if (mods.includes('number')) valueExpr = `Number(${valueExpr})`

directives.binds.push({ arg: prop, exp })
directives.ons.push({ event, exp: `${exp} = ${valueExpr}`, modifiers: [] })
```

From there it goes down the normal props-generation path — no special node. The modifiers are visible right in the excerpt: `.trim` and `.number` transform the value read from the event, and `.lazy` (just above this excerpt) swaps the event for `change` — sync on blur/Enter instead of every keystroke. The "input updates state" test checks both sides: that state shows up in the field, and that input changes it.

A subtlety on the runtime side: for `value` and `checked`, `patchProp` writes to the element property (`el.value`, `el.checked`), not the attribute. The attribute only sets the initial value; an already-rendered field can be updated only through the property — without this, `v-model` would get "stuck".

Event modifiers

Handlers almost always start with boilerplate: "don't reload the page", "don't bubble up", "only react to Enter". Vue moves these into modifiers after the event name:

```
<form @submit.prevent="save" />        <!-- $event.preventDefault() -->
<div @click.stop="onClick" />           <!-- $event.stopPropagation() -->
<input @keyup.enter="submit" />         <!-- only when Enter is pressed -->
```

The compiler parses `@submit.prevent` into an event plus a list of modifiers (`parseEvent`), and `genHandler` builds a wrapper with the needed guards:

```
function genHandler(on) {
  const exp = on.exp
  // Listener options (.once/.capture/.passive) live in the prop name, not in
  // the handler body — ignore them here.
  const mods = (on.modifiers || []).filter((m) => !OPTION_MODS.includes(m))
  const isMethodPath = /^[A-Za-z_$][\w$.]*$/.test(exp.trim())

  // No modifiers — same as before (short code, backwards compatible).
  if (mods.length === 0) {
    return isMethodPath ? `(${exp})` : `$event => (${exp})`
  }

  // With modifiers — wrap in an arrow function with the checks.
  const guards = []
  const keyMods = mods.filter((m) => KEY_MODS.includes(m))
  if (keyMods.length) guards.push(`if(!_key($event,${JSON.stringify(keyMods)}))return;`)
  if (mods.includes('stop')) guards.push('$event.stopPropagation();')
  if (mods.includes('prevent')) guards.push('$event.preventDefault();')
  if (mods.includes('self')) guards.push('if($event.target!==$event.currentTarget)return;')

  const call = isMethodPath ? `${exp}($event)` : `(${exp})`
  return `$event => { ${guards.join('')} ${call} }`
}
```

Key modifiers (`.enter`, `.esc`, `.up`, and others) are checked through the `_key` helper, which maps a modifier name to an `event.key` value (`enter` → `'Enter'`). The tests check both the generated code and the behavior: `@click.prevent` really does call `preventDefault`, and `@keyup.enter` fires only on Enter.

Three modifiers stand apart. `.once`, `.capture` and `.passive` are not checks inside the handler — they are options of `addEventListener` itself. The compiler encodes them as suffixes on the prop name (`@click.once` → `onClickOnce`, see `genProps`), and on the DOM side `patchProp` peels the suffixes back off into listener options. That is why `genHandler` filters them out (`OPTION_MODS`) before building its guards.

The “no modifiers” path stays as it was — a short reference `(inc)` or `$event => (count++)`. That way the old code and tests from layer 4 keep working unchanged, and the wrapper appears only where there actually are modifiers.

What we simplified

Real Vue supports more: multiple models on one component (`v-model:title` — a model with an argument), `v-model` on a multiple `select`, a dynamic `:type` (it needs a runtime directive — our compiler picks the field type only from a static `type`), system key modifiers (`.ctrl`, `.shift`, `.exact`) and mouse buttons. We took the most common cases — text fields, checkbox, radio and select, object/array `class`/`style`, `.stop`/`.prevent`/`.self` and keys, the value modifiers `.lazy`/`.trim`/`.number`, and the listener options `.once`/`.capture`/`.passive`. That’s enough to build a full-fledged form, and the principle behind the rest is exactly the same. `v-model` on your own components arrives in the directives chapter.

Check yourself

```
npm test        # among others — 11 tests for forms and bindings
npm run serve   # http://localhost:5173/playground/08-forms.html
```

The demo has a form with a text field, a `select`, and a checkbox on `v-model`, a badge whose class is toggled by a `:class` object, and submission via `@submit.prevent` and `@keyup.enter`. The “live state” under the form always matches the fields — that’s two-way binding in action.

Reactivity, Deeper

The reactivity core from layer 1 covers 90% of what you need, but real code brings up a few more tools. They're small and built entirely on the `effect`, `track`, and `trigger` we already have. This chapter is short — we're just picking up some convenient primitives.

Chapter code: `packages/reactivity/` (additions in `reactive.js`, `ref.js`, `watch.js`). Tests — `test/reactivity-extras.test.mjs`.

watchEffect

`watch` requires an explicit source and hands you the old and new value. Often that's overkill — you just want "run this and re-run it whenever something changes." That's what `watchEffect` is for:

```
const stop = watchEffect(() => {
  console.log('count =', count.value) // reads count → subscribes to it
})
```

It runs the function immediately (collecting dependencies from whatever it read along the way), then re-runs it when those change. The implementation is an effect whose scheduler queues a re-run in the same microtask queue `watch` uses (`flush: 'pre'`):

```
export function watchEffect(fn, options = {}) {
  // … onCleanup plumbing, same as in watch …
  const job = () => {
    if (!effect.active) return
    runCleanup()
    effect.run()
  }
  const scheduler = options.flush === 'sync' ? job : () => queueWatchJob(job)
  const effect = new ReactiveEffect(() => fn(onCleanup), scheduler)
  effect.onStop = runCleanup
```

```
  effect.run() // first run: both executes and collects dependencies
  return () => effect.stop()
}
```

It returns a stop function — call it and the watching stops (`effect.stop` runs `cleanup` and marks the effect inactive, see layer 1). The function itself receives `onCleanup` — same contract as in `watch`: whatever you register through it runs before each re-run and on stop.

readonly

Sometimes you want to hand out data "view only" — for example, a `provide` value that descendants shouldn't mutate. `readonly(obj)` returns a Proxy where reads work (including nested ones — those become `readonly` too), while writes and deletes are silently rejected with a warning:

```
const readonlyHandlers = {
  get: createGetter(true, false), // the shared getter factory, isReadonly flag on
  set(obj, key) {
    console.warn(`readonly: cannot modify property "${String(key)}"`)
    return true // the write is "swallowed" but not performed
  },
  deleteProperty(obj, key) {
    console.warn(`readonly: cannot delete property "${String(key)}"`)
    return true
  },
}

export function readonly(target) {
  return createReactiveObject(target, readonlyHandlers, readonlyMap, true)
}
```

There's nothing to track here — the value never changes through this proxy, so the getter (the same `createGetter` from layer 1, with the `isReadonly` flag on) skips `track`; nested objects come out wrapped in `readonly` too. One honest simplification: a Map or Set passed to `readonly` falls through to the shared collection handlers and gets no read-only view — writes to it are not blocked (Vue has such views; we keep the layer small).

shallowReactive and shallowRef

Deep reactivity is handy, but sometimes excessive. If you only ever change a large object by replacing it wholesale (rather than mutating its inner fields), there's no point wrapping every level. The "shallow" versions react only to the top level:

- `shallowReactive(obj)` — `state.count` is reactive, but `state.nested.x` isn't. In `get` we simply don't wrap nested objects.
- `shallowRef(v)` — reacts to replacing `.value` wholesale, but not to changing a field inside. If you did change a field and want to notify subscribers manually — there's `triggerRef(ref)`.

The test "shallowRef reacts to replacement but not to mutation" shows the difference: `s.value.count = 5` doesn't wake the effect, while `s.value = { count: 10 }` does.

markRaw

The opposite problem: mark an object so it **never** becomes reactive. You do this with heavy third-party objects (a map instance, a class from a library) that don't need reactivity and are even hurt by it performance-wise:

```
const map = markRaw(new ExpensiveThing())
const state = reactive({ map }) // state.map stays a plain object
```

The implementation is an invisible `Symbol` marker on the object that `reactive` checks before wrapping anything nested: see the marker, return it as is.

Wrap-up

All five additions are thin layers over the layer-1 machinery, with no new "engine." That's a good test of how clean the reactivity turned out: if new tools take just a few lines each to write, the foundation is right. Next up — directives and dynamic components, where we'll get real new mechanics again.

Directives and Dynamic Components

Three features that real apps rarely do without: your own directives (`v-focus`, `v-tooltip`), swapping a component based on data (`<component :is>`), and `v-model` on your own components. All three build on the component and compiler machinery we already have.

Chapter code: changes in `packages/compiler/compile.js`, `runtime-core/renderer.js`, `runtime-core/vnode.js`, `runtime-core/component.js`. Tests — `test/directives.test.mjs`, demo — `playground/10-directives.html`.

Custom directives

A directive is a way to attach low-level behavior to a plain element: set focus, wire up a third-party library, watch when the element appears on screen. A directive is an object with hooks tied to the element's lifecycle:

```
app.directive('focus', {
  mounted(el) { el.focus() }, // element entered the DOM — focus it
})
```

Hooks receive the real element `el` and a `binding` object with usage data: `value` (the current value of the expression), `oldValue` (the previous one, in `updated`), `arg` (the argument after the colon), and `modifiers`. So `v-color:bg.important="c"` gives the hook `binding.value = c`, `binding.arg = 'bg'`, `binding.modifiers = { important: true }`.

How it works

When the compiler meets an unknown `v-*`, it puts it in a separate list (`parseDirective`) and wraps the element in a `_wd` (withDirectives) call, passing a tuple of "directive, value, argument, modifiers" for each directive:

```
// <div v-focus></div> compiles to:
_wd(h("div", null, null), [[_dir("focus"), void 0, void 0, {}]])
```

`_dir` (`resolveDirective`) finds the directive by name — first in the component's local `directives` option, then among the global `app.directive` registrations. `withDirectives` just attaches the parsed bindings to `vnode.dirs`.

The renderer takes over from there. After inserting the element it calls the `mounted` hook, after an update — `updated` (having first stashed the old value into `oldValue`), and before removal — `beforeUnmount` and `unmounted`:

```
function invokeDirectives(vnode, name) {
  const dirs = vnode.dirs
  if (!dirs) return
  for (const binding of dirs) {
    const hook = binding.dir && binding.dir[name]
    if (hook) hook(vnode.el, binding, vnode)
  }
}
```

Directives are just extra calls at the same points in the element's lifecycle where the renderer was already doing work. The "mounted/updated/unmounted" test checks all three moments and that `binding` is correct.

Dynamic components

Sometimes the data decides which component to show: tabs, wizard steps, a list of heterogeneous blocks. For this there's a special tag `<component :is="...">`, where `:is` is an expression holding a component (an object) or its name (a string):

```
<component :is="currentTab" />
```

The compiler recognizes the `component` tag, takes its `:is`, and generates the node type through the `_cd` helper instead of a string tag:

```
const _cd = (is) => (typeof is === 'string' ? resolveComponent(is) : is)
// <component :is="cur"> → h(_cd(cur), null, [...])
```

If `is` is a string, `_cd` finds the component by name; if it's an object, it uses it directly. From there it's the same as any component: change `is` and `patch` sees a different `type`, unmounts the old component, and mounts the new one.

A practical subtlety shows up in the demo and the test: the component goes into a `shallowRef`, not a plain `ref`. Otherwise `ref` would try to make the component definition reactive (wrap it in a Proxy), which is pointless and harmful. The `shallowRef` from layer 9 stores the value as-is — exactly what these cases need.

v-model on components

We already covered `v-model` for input fields. On a component it works on the same "value down, event up" principle, only through the `modelValue` prop and the `update:modelValue` event:

```
<MoneyInput v-model="amount" />
<!-- compiles to: -->
<MoneyInput :modelValue="amount" @update:modelValue="amount = $event" />
```

In `applyVModel` the compiler sees that the tag is a component and expands `v-model` into that pair (not `:value`/`@input`, as for `<input>`). The component itself takes `modelValue` as a prop and reports changes through `emit`:

```
const MoneyInput = {
  props: ['modelValue'],
  setup(props, { emit }) {
    return { onIn: (e) => emit('update:modelValue', Number(e.target.value)) }
  },
  template: `<input :value="modelValue" @input="onIn" />`,
}
```

The parent and the component bind two-way while staying honest: data flows down through the prop, changes bubble up through the event. The "v-model on a component" test checks the full loop: typing into the child field updates the parent's state.

What we simplified

Real Vue supports more nuance: the directive hooks `created`/`beforeMount`/ `beforeUpdate`, directives on components, the shorthand form (a function = mounted + updated), multiple `v-model` with an argument (`v-model:title`), `keep-alive` for dynamic components (the next chapter). We took the essentials — full-featured hooks for element directives, `<component :is>` for objects and names, `v-model` on a component — which is enough for the vast majority of tasks.

Check yourself

```
npm test        # among others — 5 tests for directives and dynamic components
npm run serve   # http://localhost:5173/playground/10-directives.html
```

In the demo: `v-focus` sets focus on the field when it appears, the tabs switch the component through `<component :is>`, and `MoneyInput` is a custom component with `v-model`. Next up — Vue's built-in components: `Teleport`, `KeepAlive`, and async components.

Built-in Components

Vue ships a few "system" components that don't behave like ordinary ones: `Teleport` renders its children somewhere else on the page, `KeepAlive` keeps inactive components alive, and `defineAsyncComponent` loads code on demand. They're special — the renderer and the component system recognize them by markers and handle them differently.

Chapter code: `packages/runtime-core/builtins.js` plus changes in `renderer.js` and `component.js`. Tests — `test/builtins.test.mjs`, demo — `playground/11-builtins.html`.

Teleport: render somewhere else

A modal logically belongs to the component that opened it, but physically it should live at the end of `<body>` — otherwise a parent's `overflow: hidden` clips it or someone else's `z-index` buries it. `Teleport` breaks that tie: in the markup the children sit inside the component, but in the DOM they land in the target container.

```
<Teleport to="#modals">
  <div class="modal">...</div>
</Teleport>
```

The component itself is just a marker — `{ __isTeleport: true }`. All the work happens in the renderer, which, on seeing this marker, sends the children to the target container instead of the current one:

```
function processTeleport(n1, n2, container, anchor) {
  if (n1 == null) {
    n2.el = hostCreateText('')           // empty anchor at the original spot
    hostInsert(n2.el, container, anchor)
    const target = (n2.target = resolveTeleportTarget(n2.props))
    if (target && Array.isArray(n2.children)) mountChildren(n2.children, target, null)
  } else {
    n2.el = n1.el
    const prevTarget = n1.target
```

```
    const nextTarget = resolveTeleportTarget(n2.props)
    if (prevTarget) {
      n2.target = prevTarget
      patchChildren(n1, n2, prevTarget, null) // diff in the OLD target first
      if (nextTarget && nextTarget !== prevTarget) {
        n2.target = nextTarget
        for (const child of n2.children) move(child, nextTarget, null) // `to` changed
      }
    }
    // … target appeared only now → mount the children fresh
  }
}
```

An empty text anchor stays at the original spot so sibling nodes don't get thrown off, while the content lives in the target container. The "children render in the target container" test checks exactly this: the original spot is empty, all the content is in the target. On update the children are diffed in the *old* target (its anchors are stable), and only then, if `to` changed, carried over into the new container — ignoring the freshly resolved target here was a real bug: the computed value used to be thrown away and the children stayed put forever.

KeepAlive: hide, don't destroy

Normally, switching a dynamic component destroys the old one along with all its state: type text into a form, move to another tab, come back — the field is empty. `KeepAlive` fixes that: an inactive component isn't destroyed, it's hidden, keeping its instance and state.

```
<KeepAlive><component :is="currentTab" /></KeepAlive>
```

The implementation is a combination of "markers on the vnode + special handling in the component system." `KeepAlive` itself keeps a cache of "key → vnode with a live instance" and sets the markers:

```
if (cache.has(key)) {
  vnode.component = cache.get(key).component // reuse the live instance
  vnode.__keptAlive = true // the component system "reactivates" instead of remounting
}
cache.set(key, vnode) // always store the FRESH vnode — it carries the latest props
vnode.__shouldKeepAlive = true  // on leave — stash, don't destroy
vnode.__keepAliveOwner = instance
```

The component system (`component.js`) reacts to the markers in two places. On unmount, instead of destroying the component, `unmountComponent` moves its DOM into off-screen storage without touching the instance — and fires `onDeactivated`:

```
if (vnode.__shouldKeepAlive && owner && !owner.__keepAliveTearingDown) {
  move(instance.subTree, keepAliveStorage(), null) // stashed away
  instance.isDeactivated = true
  invokeHooks(instance.da, instance, 'deactivated hook') // onDeactivated
  return
}
```

And when "mounting" a cached component, `activateComponent` doesn't create it anew — it brings the stashed DOM back, then runs a normal update against the new vnode (its props and slots may have changed while the component slept) and fires `onActivated`:

```
function activateComponent(vnode, container, anchor) {
  const instance = vnode.component // set by KeepAlive's render from its cache
  move(instance.subTree, container, anchor)
  updateComponent(instance.vnode, vnode)
  vnode.el = instance.subTree.el
  instance.isDeactivated = false
  invokeHooks(instance.a, instance, 'activated hook') // onActivated
}
```

The instance stays alive the whole time, its reactive effect is never stopped — which is why the state is right where you left it. The "state is preserved across switching" test proves it: a counter incremented on tab A stays put after moving to B and back, instead of resetting.

The `__keepAliveOwner` marker matters at the very end. When the `KeepAlive` itself is unmounted for real, `unmountComponent` flags it as tearing down: its children are no longer stashed, and everything still hiding in the cache gets a genuine unmount — hooks fire, effects stop, the storage DOM is freed. Without an owner, "unmount" would keep stashing forever and leak every cached component for the life of the page.

A slots bug found along the way

`KeepAlive` exposed a subtle bug. Its `setup` returns `() => slots.default()`, capturing `slots` in a closure. But our code, when updating a component, **reassigned** `instance.slots` to a new object — and the closure kept seeing the old one. Because of this, `KeepAlive` always showed the first tab. The fix: update the contents of the same `slots` object instead of replacing the reference (`updateSlots` in `component.js`). A good lesson: reference stability matters anywhere something gets captured in a closure.

Async components

There's no reason to load a large app all at once — rarely used screens can be pulled in later. `defineAsyncComponent` wraps a loader (usually a dynamic `import`) in a component that shows "loading" while the code is on its way, then swaps in the real component:

```
const Chart = defineAsyncComponent(() => import('./Chart.js'))
```

The implementation is, again, just reactivity. The wrapper's `setup` kicks off the loader and holds state `refs`; when the promise resolves, a `ref` flips — and the wrapper re-renders with the real component:

```
setup() {
  const instance = getCurrentInstance()
  const loaded = ref(false)
  const error = ref(null)

  load() // one shared in-flight request for every instance of the wrapper
    .then(() => {
      if (!instance || !instance.isUnmounted) loaded.value = true
    })
    .catch((err) => {
      if (!instance || !instance.isUnmounted) error.value = err
    })

  return () => {
    if (loaded.value && resolvedComponent) return h(resolvedComponent)
    if (error.value) {
      return options.errorComponent ? h(options.errorComponent) : h('span', 'Loading failed')
    }
    return options.loadingComponent ? h(options.loadingComponent) : h('span', 'Loading…')
  }
}
```

No special machinery — reactivity switches the view on its own. Two guards are worth noting: `load()` caches a single in-flight promise, so ten async components in a list trigger one fetch, not ten (a failed load clears the slot to allow a retry); and the `isUnmounted` checks keep a wrapper that was removed before the chunk arrived from flipping state and mounting the late component into a container that's already gone. The tests cover both outcomes: a successful load and an error (in which case `errorComponent` is shown).

What we simplified

We left out `Suspense` — the coordinator for several async dependencies with a shared fallback and async `setup`. It hooks into deeper promise handling inside the render and would deserve a whole chapter of its own; for teaching purposes `defineAsyncComponent` is enough to show the essence of "loading → ready." Also, in real Vue `KeepAlive` supports `include`/`exclude` and a cache limit (`max`), while `Teleport` has `disabled`. We took the core of each — though the `activated`/`deactivated` hooks did make it in: `onActivated`/`onDeactivated` fire exactly where the stash-and-restore happens.

Check yourself

```
npm test        # among others — 5 tests for the built-in components
npm run serve   # http://localhost:5173/playground/11-builtins.html
```

In the demo: a modal moves into `#modals` via `Teleport`, tabs under `KeepAlive` retain the text you typed, and the async block appears after a delay. One last layer remains — store extensions and the big final app, where everything comes together.

Store Deep-Dive and the Capstone App

The last layer rounds out the store's conveniences and brings everything we've built together into one living app. There's almost no new deep machinery here — this is where the picture gets completed.

Chapter code: additions in `packages/store/index.js`, the final app in `playground/12-capstone.html`.

$patch, $subscribe, $reset

Three utility methods on the store that you reach for constantly in real work.

`$patch` changes several fields at once — with an object or a function. An object is handy for straight replacements, a function for computing from the current state:

```
store.$patch({ count: 5, name: 'y' })      // merge an object
store.$patch((s) => { s.count++; s.done = true }) // mutate with a function
```

The implementation picks the target of the change: for an options store it's the reactive state `pinia.state[id]`, for a setup store it's the store itself (a write through `proxyRefs` lands in the `.value` of the right ref). One change, one path.

`$subscribe` runs a callback on any state change. It's a deep `watch` on top of our reactivity, and it batches like Pinia: any number of synchronous mutations (say, one `$patch` of three keys) collapse into a single callback on the next microtask. The callback receives mutation info, `{ type, storeId }`, where `type` is `'direct'`, `'patch object'` or `'patch function'`:

```
store.$subscribe((mutation, state) => console.log(mutation.type, state))
```

A subtlety a test exposed: for a setup store you can't watch both the state and the getters at once. Otherwise a single change to `items` would fire twice — once from the ref itself and once from the `computed` that depends on it. So we filter out the computeds (they carry a `.effect`) and deep-watch only the state refs — `{ deep: true }` makes the watcher walk inside ref values, so `items.value.push(...)` is seen too. A small detail, but it shows how the layers connect: the store subscription leans on how `computed` and `watch` from layer 1 are built.

`$reset` returns the state to its initial values. It works for options stores, where the `state()` function that produces fresh values is known. A setup store has no initial snapshot, so `$reset` throws — exactly like Pinia (a setup store can expose its own reset action if it wants one).

The final app

`playground/12-capstone.html` is a "Notes" app where everything runs at once:

- **Router** — three routes: the list `/`, editing `/note/:id` with a param, and an "About" page. Navigation through `RouterLink`, rendering through `RouterView`.
- **Store** — `useNotes` in setup style holds the notes; `storeToRefs` hands reactive refs to the component; `$subscribe` logs changes to the console.
- **Forms** — adding and editing through `v-model`, submitting on `@keyup.enter`.
- **Teleport** — the delete confirmation is moved into a modal above the page.
- **Components** — the list and the detail talk only through the store and the router, with no props passed between them.

Not a single third-party line: reactivity, rendering, template compilation, the router, and the store — all ours, assembled over twelve layers. If this app works, the whole framework works.

Two more apps — and a bug the shop caught

The `examples/` folder holds two larger apps built entirely on this framework:

- **MiniTrello** (`examples/kanban/`) — a Kanban board: router, a persisted store, `v-model` forms, a Teleport modal, KeepAlive tabs, an async panel and custom directives. Running it end-to-end in a real browser is what first caught several renderer bugs.
- **MiniShop** (`examples/shop/`) — a storefront fed by a fake REST API: async loading with error states, a `/product/:id` page that refetches through `watchEffect`, a persisted cart in a Teleport drawer, plus live search, category filtering and sorting.

MiniShop earned its keep immediately. Its product grid is a **keyed list of components** (`<ProductCard :key="id">`), and sorting it re-orders those components. That reordering threw `insertBefore ... is not of type Node` — the exact gotcha from chapter 3. A component's re-render is queued, so `updateComponent` has to carry its DOM node over synchronously (`n2.el = n1.el`) for the keyed diff to move it. Our earlier apps only re-ordered keyed *elements*, whose `el` is assigned during `patch`, so the bug stayed hidden until a real product grid needed sorting. One more reminder that the surest way to find the gaps in a framework is to build something real on it.

What we covered

Twelve layers, each built on the ones before it:

1. **Reactivity** — `track`/`trigger`, `ref`, `reactive`, `computed`, `watch`.
2. **Virtual DOM** — `h`, the renderer, keyed diff via the longest increasing subsequence.
3. **Components** — `setup`, the reactive render effect, the scheduler, props/emit, slots, provide/inject, the lifecycle, `createApp`.
4. **Compiler** — from a template to a render function through `with(ctx)`.
5. **Router** — reactive route, history, params, guards.
6. **Store** — `defineStore`, getters on `computed`, actions, plugins.
7. **SSR** — `renderToString` and hydration by adopting the DOM.
8. **Forms** — `v-model`, class/style binding, event modifiers.
9. **Reactivity extensions** — `watchEffect`, `readonly`, `shallow*`, `markRaw`.
10. **Directives and dynamics** — custom directives, `<component :is>`, `v-model` on components.
11. **Built-in components** — `Teleport`, `KeepAlive`, async components.
12. **Store extensions** — `$patch`, `$subscribe`, `$reset` — and this app.

All of them grew from the single idea of chapter one: "the data changed, so whatever depends on it updated itself." Reactivity turned out to be not one feature among many but the foundation everything else stands on — rendering, components, the router, the store, even SSR. That's the book's main takeaway: big systems rarely rest on many complex mechanisms. More often it's one right idea, carried carefully into every corner.

What's next

Real Vue differs from ours not in ideas but in finish: patch flags to speed up the diff, static hoisting in the compiler, `Suspense` and streaming SSR, devtools, strict typing, thousands of tested edge cases. All of it is engineering polish over exactly the mechanisms you've now written yourself. Open Vue's source: you'll find familiar `track`, `trigger`, `patchKeyedChildren`, `setupComponent` — just larger and more careful. You'll read them now not as magic but as an extended version of your own code.

Thanks for going the whole way. You didn't learn Vue — you wrote it.

Check yourself

```
npm test        # all 247 tests across twelve layers
npm run serve   # http://localhost:5173/playground/12-capstone.html
```

The “Notes” app is the final check: add a note, open it, edit it, delete it through the modal, click through the routes. It all runs on a framework that twelve chapters ago was an empty folder.