Vue

A Complete 12-Chapter Course

Topics covered:

SFC & Reactivity · Directives · v-if/v-for · Computed & Watchers · Components & Props

Events & v-model · Slots · Lifecycle Hooks · Composables · Vue Router · Pinia

Exercises: 36 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · React/Angular comparison tables

Table of Contents

CHAPTER 1 OF 12

What Vue Is, the SFC, Vite Setup

CHAPTER 1

What Vue Is, the SFC, and Vite Setup

A framework that sits between React and Angular — and borrows the best ideas of each

With both React and Angular already under your belt, Vue is best understood by where it lands between them. Like **React**, it's component-based, flexible, and unopinionated about much of your app's structure. Like **Angular**, it uses an HTML template with directives (`v-if`, `v-for`) rather than JSX, and ships official solutions for routing and state. Vue's own pitch is being approachable and incrementally adoptable — and in practice it feels like Angular's template ergonomics with React's lightness.

The Single-File Component (SFC)

```
<!-- Greeting.vue -->
<script setup>
const name = 'Philip';
</script>

<template>
  <h1>Hello, {{ name }}!</h1>
</template>

<style scoped>
  h1 { color: #42b883; }
</style>
```

A Vue component lives in a `.vue` file — a **Single-File Component** — with three blocks: `<script>` for logic, `<template>` for markup, and an optional `<style>` for CSS, all in one file. This is a genuine middle ground: React puts markup and logic together in JSX but CSS elsewhere; Angular splits all three into separate files; Vue keeps all three together but cleanly separated by block. `{{ name }}` is text interpolation — identical syntax to Angular, and the same role as React's `{name}`.

<script setup> and the Composition API

The `setup` attribute on `<script>` enables the **Composition API** — the modern way to write Vue, where everything declared in the script block (variables, functions, imports) is automatically available to the template.

There's no `return` statement, no boilerplate wiring: declare `const name = 'Philip'` and the template can use `{{ name }}` directly. This course uses `<script setup>` throughout.

> **YOU'LL ALSO SEE THE OPTIONS API IN OLDER CODE**
>
> Older Vue tutorials (and Vue 2 code) use the **Options API** — a component defined as an object with `data()`, `methods`, `computed`, etc. as separate named sections. It's still fully supported in Vue 3, but `<script setup>` with the Composition API is the recommended modern default — the same kind of "modern vs legacy" choice as standalone components over NgModules in Angular, or hooks over class components in React. Recognize the Options API when you meet it; this course teaches only the Composition API.

Style Scoping — Built In

The `scoped` attribute on `<style>` confines those styles to this component only — the same automatic style encapsulation Angular gives every component, and what React needs CSS Modules or a library to achieve. Two components can both style `h1` differently with zero collision, just by adding `scoped`.

Creating a Project with Vite

```
# scaffold a new Vue project
npm create vue@latest my-app

cd my-app
npm install
npm run dev
```

`npm create vue@latest` runs Vue's official scaffolding tool (built on Vite — the same build tool used throughout the React course), asking a few setup questions (TypeScript, router, Pinia — all optional and addable later). `npm run dev` starts the dev server with live reload, exactly as it did for React. The key generated files:

- `src/main.js` — the entry point; mounts the root `App` component onto the page.
- `src/App.vue` — the root SFC, the first component you'll edit.
- `index.html` — the single real HTML page everything mounts into, just like React's.

Mounting the Root Component

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

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

`createApp(App).mount('#app')` is Vue's bootstrap — it takes the root component and attaches it to the `<div id="app">` in `index.html`, the direct equivalent of React's `createRoot(...).render(<App />)` or Angular's `bootstrapApplication(AppComponent)`. Everything else renders inside that root.

CONCEPT	REACT	ANGULAR	VUE
Markup style	JSX	HTML template + directives	HTML template + directives
Component file	`.jsx`	`.ts` + `.html` + `.css`	`.vue` (all three blocks)
Interpolation	`{value}`	`{{ value }}`	`{{ value }}`
Scoped styles	CSS Modules / lib	Automatic	`<style scoped>`

Coding Challenges

CHALLENGE 1

Create a new Vue project, and edit App.vue so its template shows "Welcome, [your name]!" by interpolating a constant declared in <script setup>.

View solution

CHALLENGE 2

Build a Greeting.vue SFC with all three blocks — a name constant in script, an interpolated heading in the template, and a scoped style coloring that heading.

View solution

CHALLENGE 3

Build a second component (e.g. Footer.vue) and use it inside App.vue by importing it in <script setup> and placing its tag in the template — confirming a child component renders inside a parent.

View solution

CHAPTER 1 QUICK REFERENCE

- **Vue** — component-based and flexible like React, template+directive-driven like Angular
- **SFC (.vue file)** — `<script>`, `<template>`, `<style>` blocks together in one file
- **<script setup>** — the Composition API; everything declared is auto-available to the template
- **{{ value }}** — text interpolation, identical to Angular
- **<style scoped>** — automatic per-component style encapsulation
- **npm create vue@latest** / **npm run dev** — scaffold and run (Vite-based)
- **createApp(App).mount('#app')** — bootstraps the root component
- The older **Options API** appears in legacy code; this course uses `<script setup>` throughout
- **Next chapter:** the template, interpolation, and `ref` reactivity basics

CHAPTER 2 OF 12

Template, Interpolation, and ref Reactivity

CHAPTER 2

Template, Interpolation, and ref Reactivity

How Vue tracks changing values — and why a counter built with a plain variable wouldn't update

Chapter 1's interpolated values were constants that never changed. The moment a value needs to change and have the UI update, it must be **reactive** — Vue's tracking system that re-renders the template when a value changes. In the Composition API, the core tool for that is `ref`.

ref — A Reactive Value

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

const count = ref(0);

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

<template>
  <p>Count: {{ count }}</p>
  <button @click="increment">+1</button>
</template>
```

`ref(0)` creates a reactive container holding `0`. The catch that trips up everyone at first: in the `<script>`, you read and write the value through `.value` (`count.value++`) — but in the `<template>`, Vue unwraps it automatically, so it's just `{{ count }}`, no `.value`. This is Vue's equivalent of React's `useState`, but with two key differences: there's no separate setter function (you assign to `.value` directly), and the same `ref` is used both to read and to update.

.VALUE IN SCRIPT, NO .VALUE IN TEMPLATE

This split is the single most common Vue stumbling point. Inside `<script setup>`, always use `count.value`. Inside `<template>`, always use plain `count` — Vue unwraps it for you there. Forgetting `.value` in the script (writing `count++`) silently fails to update anything; adding `.value` in the template (`{{ count.value }}`) shows `undefined`.

Why Not a Plain Variable?

```
<!-- this does NOT work -->
<script setup>
let count = 0;
function increment() {
  count++;  // changes the variable, but Vue has no idea — the UI never updates
}
</script>
```

A plain `let count = 0` increments fine in memory, but Vue isn't tracking it — nothing tells the framework the UI needs re-rendering, so the displayed number never changes. Exactly the same trap as a plain variable in React (Fundamentals Chapter 3). `ref` is what makes the value reactive; the change to `.value` is what Vue detects.

reactive — For Objects

```
import { reactive } from 'vue';

const state = reactive({ name: 'Philip', age: 35 });

function birthday() {
  state.age++;  // no .value needed — reactive objects are accessed directly
}
```

`reactive` is an alternative for objects, making the whole object reactive with no `.value` — properties are accessed directly (`state.age++`). In practice, many Vue developers use `ref` for everything (it works for objects too, via `ref({...}).value`) to avoid juggling two mental models, but `reactive` is common and worth recognizing. The trade-off: `reactive` only works on objects/arrays, and you lose reactivity if you destructure it.

Expressions in the Template

```
<p>{{ count * 2 }}</p>
<p>{{ name.toUpperCase() }}</p>
<p>{{ isActive ? 'On' : 'Off' }}</p>
```

Interpolation isn't limited to a bare value — any single JavaScript *expression* works inside `{{ }}`: arithmetic, method calls, ternaries. The same as React's curly braces, and like Angular's interpolation, it's expressions only (no statements like `if` or `for` — those have their own directives, next chapter).

	REACT USESTATE	VUE REF
Create	`const [c, setC] = useState(0)`	`const c = ref(0)`
Read (script)	`c`	`c.value`
Read (markup)	`{c}`	`{{ c }}`
Update	`setC(c + 1)`	`c.value++`

Coding Challenges

CHALLENGE 1

Build a counter with a ref starting at 0 and a +1 button that increments it (remembering .value in the function), displaying the count via interpolation.

View solution

CHALLENGE 2

Build a component with a reactive object (via reactive) holding firstName and lastName, plus a button that changes one of them, displaying "Full name: [first] [last]" with both updating live.

View solution

CHALLENGE 3

Build a component with a price ref and a quantity ref, displaying their product directly in the template as a {{ }} expression (price * quantity), with buttons to change each — confirming the total recomputes automatically.

View solution

CHAPTER 2 QUICK REFERENCE

- **ref(value)** — a reactive value; Vue's equivalent of `useState`
- Read/write with **.value** in `<script>`; use the plain name (auto-unwrapped) in `<template>`
- No separate setter — assign to `.value` directly (`count.value++`)
- A plain variable won't trigger UI updates — only a `ref` (or `reactive`) is tracked
- **reactive({...})** — makes an object reactive with no `.value`; objects/arrays only
- **{{ }}** accepts any single JS expression (math, method calls, ternaries)

- **Next chapter:** directives — `v-bind`, `v-on`, `v-model`

CHAPTER 3 OF 12

Directives — v-bind, v-on, v-model

CHAPTER 3

Directives — v-bind, v-on, v-model

Connecting reactive data to attributes, events, and form inputs

Interpolation (Chapter 2) only handles text content between tags. For everything else — setting an attribute, responding to an event, syncing a form input — Vue uses **directives**: special `v-` prefixed attributes, the same conceptual mechanism as Angular's bindings, just with different syntax. These three are the workhorses.

v-bind — Binding an Attribute

```
<!-- full form -->
<img v-bind:src="photoUrl" />
<button v-bind:disabled="isSaving">Save</button>

<!-- shorthand: just a colon -->
<img :src="photoUrl" />
<button :disabled="isSaving">Save</button>
```

`v-bind` binds an attribute to a reactive expression rather than a static string — `:disabled="isSaving"` sets the real boolean `disabled` property from the `isSaving` ref. The `:` shorthand is near-universal in real code. This is the exact equivalent of Angular's `[disabled]="isSaving"` (square brackets) and serves the same purpose as React's `disabled={isSaving}` — passing a real value, not a string.

v-on — Listening for Events

```
<!-- full form -->
<button v-on:click="increment">+1</button>

<!-- shorthand: @ -->
<button @click="increment">+1</button>
<input @input="onInput" />
```

`v-on` attaches an event listener — already used informally in earlier chapters via its `@` shorthand. `@click="increment"` is Vue's version of Angular's `(click)="increment()"` and React's `onClick=`

`{increment}`. Two conveniences: you can write a method name (`@click="increment"`) or an inline expression (`@click="count++"`), and the native event object is available as `$event` when you need it (`@input="onInput($event)"`).

> **EVENT MODIFIERS — A VUE CONVENIENCE**
>
> Vue adds **modifiers** chained onto an event with dots: `@submit.prevent="onSubmit"` calls `event.preventDefault()` for you (no manual call like React's Fundamentals Chapter 4 needed), `@click.stop` stops propagation, `@keyup.enter` fires only on the Enter key. These small declarative shortcuts replace boilerplate you'd write by hand in React.

v-model — Two-Way Binding

```
<script setup>
import { ref } from 'vue';
const name = ref('');
</script>

<template>
  <input v-model="name" />
  <p>Hello, {{ name }}!</p>
</template>
```

`v-model` keeps a form input and a ref synchronized in both directions automatically — typing updates `name`, and changing `name` in code updates the input. This is the same end result as Angular's `[(ngModel)]`, and it collapses React's manual controlled-input pattern (`value` + `onChange`, Fundamentals Chapter 7) into a single directive. Under the hood it's just `:value` + `@input` combined — a shorthand Vue provides, and one you'll build yourself on a custom component in Chapter 7.

v-model on Other Input Types

```
<input type="checkbox" v-model="agreed" />        <!-- boolean ref -->
<select v-model="country">                  <!-- string ref -->
  <option value="ie">Ireland</option>
  <option value="hu">Hungary</option>
</select>
```

`v-model` automatically adapts to the input type — a checkbox binds to a boolean, a `<select>` binds to the chosen option's value, radio buttons to the selected value, all with the same `v-model` attribute. This is noticeably less fiddly than React, where a checkbox needed `checked` / `e.target.checked` while a text input

needed `value` / `e.target.value` (Fundamentals Chapter 7) — Vue figures out the right property and event for you.

V-MODEL NEEDS NO FORMSMODULE-STYLE IMPORT

Unlike Angular, where `[(ngModel)]` required importing `FormsModule` into the component, Vue's `v-model` is built into the template compiler and works out of the box — no import, no setup. One fewer thing to forget.

VUE	ANGULAR	REACT
`:attr="x"`	`[attr]="x"`	`attr={x}`
`@event="handler"`	`(event)="handler()"`	`onEvent={handler}`
`v-model="x"`	`[(ngModel)]="x"`	`value={x} onChange={...}`
`@submit.prevent`	manual `preventDefault()`	manual `preventDefault()`

Coding Challenges

CHALLENGE 1

Build a component with an isLocked ref, a Save button whose disabled attribute is v-bind-bound to it, and a second button (@click) that toggles isLocked. Use the shorthand syntaxes (: and @).

View solution

CHALLENGE 2

Build a single-field form using @submit.prevent on the form and v-model on a text input, logging the value on submit and clearing the input afterward — with no manual preventDefault call.

View solution

CHALLENGE 3

Build a component with a text input, a checkbox, and a select dropdown, each v-model-bound to its own ref (string, boolean, string), displaying all three current values live below the form.

View solution

CHAPTER 3 QUICK REFERENCE

- **v-bind:attr** (shorthand **:attr**) — bind an attribute to a reactive expression (= Angular `[attr]`)
- **v-on:event** (shorthand **@event**) — listen for an event (= Angular `(event)` / React `onEvent`)
- **$event** — the native event object, passed explicitly when needed
- **Modifiers**: `@submit.prevent`, `@click.stop`, `@keyup.enter` — declarative shortcuts
- **v-model** — two-way binding on a form input (= Angular `[(ngModel)]`), no import needed
- `v-model` auto-adapts to input type (text/checkbox/select/radio) — simpler than React's per-type handling
- **Next chapter:** conditional and list rendering — `v-if` and `v-for`

CHAPTER 4 OF 12

Conditional and List Rendering — v-if, v-for

CHAPTER 4

Conditional and List Rendering — v-if, v-for

Showing, hiding, and repeating elements — Vue's directive answer to JSX conditionals and .map()

React handles conditionals and lists with plain JavaScript in JSX (ternaries, `&&`, `.map()`); Angular uses `@if` / `@for` blocks. Vue uses directives on the element itself — `v-if` and `v-for` — closer to Angular's older `*ngIf` / `*ngFor` style, but with their own conveniences.

v-if / v-else-if / v-else

```
<p v-if="status === 'loading'">Loading...</p>
<p v-else-if="status === 'error'">Something went wrong.</p>
<p v-else>Ready!</p>
```

`v-if` conditionally renders an element based on a truthy expression, with optional `v-else-if` and `v-else` chained on adjacent elements. Like Angular's `@if` (and unlike a CSS `display: none`), a false `v-if` genuinely removes the element from the DOM entirely. The full `v-if/v-else-if/v-else` chain neatly handles the loading/error/success pattern that ran through the React projects, without a lookup object or chained ternaries.

v-show — Toggle Visibility, Keep It in the DOM

```
<div v-show="isExpanded">Details here...</div>
```

`v-show` looks similar but works differently: the element *stays* in the DOM and is toggled with CSS `display`. Use `v-if` when the element is rarely shown or expensive to create (it's truly added/removed); use `v-show` when toggling frequently (e.g. a dropdown), since flipping a CSS property is cheaper than repeatedly creating and destroying the element. This explicit choice is something React and Angular handle more implicitly — Vue names it directly.

v-for — Rendering a List

```
<ul>
  <li v-for="fruit in fruits" :key="fruit">{{ fruit }}</li>
</ul>
```

`v-for="item in items"` repeats the element once per array entry — Vue's `.map()` equivalent. The `:key` binding is required (using `v-bind` from Chapter 3) and serves the identical purpose as React's `key` and Angular's `track`: a stable, unique identifier so Vue can update the DOM efficiently when the list changes. A real object array keys on its `id`:

```
<li v-for="todo in todos" :key="todo.id">{{ todo.text }}</li>
```

v-for with an Index

```
<li v-for="(todo, index) in todos" :key="todo.id">
  {{ index + 1 }}. {{ todo.text }}
</li>
```

A second variable in parentheses gives the current index — `(todo, index) in todos` — useful for numbering. Note the index is the *position*, not a good `:key` on its own (same caution as React's array index): prefer a stable `id` for the key whenever one exists.

> **DON'T USE V-IF AND V-FOR ON THE SAME ELEMENT**
>
> Putting both directives on one element is discouraged and has ambiguous precedence. To filter a list, either filter the array first (in a computed property, Chapter 5) or wrap the `v-for` in a `<template v-if>`. A common clean pattern is a computed `visibleTodos` that returns only the items to show, then `v-for` over that — exactly the "filter before mapping" approach from React's Fundamentals Chapter 6.

The <template> Wrapper

```
<template v-for="item in items" :key="item.id">
  <dt>{{ item.term }}</dt>
  <dd>{{ item.definition }}</dd>
</template>
```

To repeat or conditionally render *multiple* elements without adding a wrapper element to the DOM, put the directive on a `<template>` tag — an invisible grouping element that renders only its contents. This is Vue's equivalent of React's fragment (`<>...</>`), used here specifically to attach a directive to a group.

VUE	ANGULAR	REACT
`v-if / v-else-if / v-else`	`@if / @else if / @else`	Ternary / `&&`
`v-show`	`[hidden]` / `[style.display]`	Conditional class/style
`v-for + :key`	`@for + track`	`.map()` + `key`
`<template> wrapper`	`<ng-container>`	Fragment `<>...</>`

Coding Challenges

CHALLENGE 1

Build a component with a status ref ("loading"/"error"/"success") cycled by a button, using v-if/v-else-if/v-else to show different content for each status.

View solution

CHALLENGE 2

Given an array of objects ({ id, name }) in a ref, render them as a list with v-for and :key, plus a numbered version using the (item, index) form showing "1. Name", "2. Name", etc.

View solution

CHALLENGE 3

Build a component with a list of items and a "Show details" toggle button. Use v-show to reveal a details panel (kept in the DOM), and separately use v-if elsewhere to show a "No items" message when the array is empty.

View solution

CHAPTER 4 QUICK REFERENCE

- **v-if / v-else-if / v-else** — conditional rendering; truly adds/removes the element from the DOM
- **v-show** — toggles CSS `display`, keeping the element in the DOM (cheaper for frequent toggles)
- **v-for="item in items"** — list rendering; a required **:key** (= React's `key` / Angular's `track`)
- **(item, index) in items** — also exposes the index; don't use the index as the key if a stable id exists
- Don't combine `v-if` and `v-for` on one element — filter via a computed property instead
- **<template>** with a directive — group multiple elements without an extra DOM node (= React fragment)

- **Next chapter:** computed properties and watchers

CHAPTER 5 OF 12

Computed Properties and Watchers

CHAPTER 5

Computed Properties and Watchers

Deriving values that update themselves, and reacting to changes with side effects

Chapter 2 put a `{{ price * quantity }}` expression directly in the template. That works, but repeating it or running heavier logic inline gets messy fast. **Computed properties** are Vue's clean way to derive a reactive value from other reactive values; **watchers** are for running a side effect when something changes. Both will feel familiar from React's `useMemo` and `useEffect` — but with Vue's automatic dependency tracking.

computed — A Derived Reactive Value

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

const price = ref(100);
const quantity = ref(2);

const total = computed(() => price.value * quantity.value);
</script>

<template>
  <p>Total: {{ total }}</p>
</template>
```

`computed(() => ...)` returns a ref-like value that automatically recalculates whenever any reactive value it reads changes — here, whenever `price` or `quantity` changes. Two important properties: it's **cached** (only recomputes when a dependency actually changes, not on every render), and its dependencies are **auto-tracked** — there's no dependency array to maintain, unlike React's `useMemo`. Vue knows `total` depends on `price` and `quantity` simply because the function read them. In the template it's used as `{{ total }}` (auto-unwrapped, no `.value`); in script you'd read `total.value`.

IDENTICAL IN SPIRIT TO ANGULAR'S COMPUTED SIGNAL

If the Angular course's `computed()` signal felt clean, this is the same idea — a derived value that tracks its own dependencies and caches. Vue's `computed` predates Angular's by years; the convergence is no accident, as both

> frameworks moved toward fine-grained reactivity. Coming from React, think "`useMemo` with no dependency array, and you read it like a value."

Computed vs a Method

```
<!-- a method: re-runs on every single render -->
<p>{{ calculateTotal() }}</p>

<!-- a computed: cached, only re-runs when price/quantity change -->
<p>{{ total }}</p>
```

You could call a regular method in the template instead — but a method re-runs every time the component re-renders for *any* reason, even if its inputs didn't change. A `computed` caches its result and only recomputes when a dependency genuinely changes. For anything beyond a trivial expression, prefer `computed` — the caching is free performance.

Filtering a List with computed

```
const todos = ref([/* ... */]);
const showCompleted = ref(false);

const visibleTodos = computed(() =>
  showCompleted.value
    ? todos.value
    : todos.value.filter((t) => !t.done)
);
```

This is the clean fix for Chapter 4's "don't combine `v-if` and `v-for`" rule: a computed property returns exactly the items that should be shown, and the template does `v-for="todo in visibleTodos"`. The filter recomputes automatically when either `todos` or `showCompleted` changes — the same "filter before mapping" pattern React used in Fundamentals Chapter 6, here expressed reactively.

watch — Running a Side Effect on Change

```
import { ref, watch } from 'vue';

const searchTerm = ref('');

watch(searchTerm, (newValue, oldValue) => {
  console.log(`changed from "${oldValue}" to "${newValue}"`);
  // e.g. fire a search request here
});
```

`watch` runs a callback whenever a specific reactive source changes, receiving both the new and old value. It's for genuine **side effects** in response to a change — fetching data, saving to `localStorage`, logging — the role React's `useEffect` with a dependency plays. The key distinction from `computed`: `computed` *derives and returns a value*; `watch` *does something* and returns nothing. If you find yourself assigning to another ref inside a `watch`, a `computed` is usually the better tool.

watchEffect — Auto-Tracked Watching

```
import { watchEffect } from 'vue';

watchEffect(() => {
  console.log(`total is now ${price.value * quantity.value}`);
});
// runs immediately, then again whenever price or quantity changes
```

`watchEffect` is a variant that runs immediately and re-runs whenever *any* reactive value it reads changes — automatically tracked, no explicit source to name. It's closer in feel to React's `useEffect` with auto-detected dependencies. Use `watch` when you need the old value or want to watch one specific source; use `watchEffect` when you just want "re-run this whenever anything it uses changes."

DON'T OVERUSE WATCH — REACH FOR COMPUTED FIRST

A very common beginner pattern is watching one ref to manually update another — that's almost always a `computed` in disguise, and the `computed` version is simpler, cached, and less bug-prone. Reserve `watch` / `watchEffect` for true side effects (I/O, logging, imperative DOM work). The mental rule: "deriving a value → `computed`; doing something → `watch`."

VUE	REACT	ANGULAR
`computed(() => ...)`	`useMemo` (no dep array)	`computed()` signal
`watch(src, cb)`	`useEffect` with a dep	`effect()` / RxJS `watch`
`watchEffect(cb)`	`useEffect` (auto deps)	`effect()` signal

Coding Challenges

CHALLENGE 1

Build a component with firstName and lastName refs and a fullName computed property combining them, with inputs to edit each — confirming fullName updates automatically.

View solution

CHALLENGE 2

Build a todo list with a "show completed" checkbox, using a computed visibleTodos that filters based on the checkbox, then v-for over visibleTodos — the clean fix for not combining v-if and v-for.

View solution

CHALLENGE 3

Build a component with a count ref and a watch that logs the new and old value whenever count changes, plus a watchEffect that logs a message referencing count — observing how the two differ (watchEffect runs immediately, watch does not).

View solution

CHAPTER 5 QUICK REFERENCE

- **computed(() => ...)** — a cached, auto-tracked derived value (= `useMemo` with no dep array)
- Prefer `computed` over a method for derived values — it caches; a method re-runs every render
- A `computed` filter is the clean fix for "don't combine v-if and v-for" (Chapter 4)
- **watch(source, (new, old) => ...)** — runs a side effect on change, with old/new values
- **watchEffect(() => ...)** — runs immediately and re-runs on any read dependency's change
- Rule of thumb: **deriving a value → computed; doing something → watch**
- **Next chapter:** components and props

CHAPTER 6 OF 12

Components and Props

CHAPTER 6

Components and Props

Passing data into a child component with defineProps — Vue's equivalent of React props and Angular @Input

Chapter 1's child components were static. **Props** make a component reusable by letting a parent pass data into it — the same concept as React props and Angular `@Input`. In `<script setup>`, props are declared with the compiler macro `defineProps`.

Declaring Props with defineProps

```
<!-- UserCard.vue -->
<script setup>
const props = defineProps(['name', 'age']);
</script>

<template>
  <h3>{{ name }}</h3>
  <p>Age: {{ age }}</p>
</template>
```

`defineProps(['name', 'age'])` declares the props this component accepts. It's a **compiler macro** — no import needed; Vue's compiler recognizes it inside `<script setup>`. In the template, props are used directly by name (`{{ name }}`), just like a local ref. In the script, they're accessed via the returned object (`props.name`).

Passing Props from a Parent

```vue
<!-- parent template -->
<script setup>
import UserCard from './UserCard.vue';
</script>

<template>
  <UserCard name="Philip" :age="35" />
  <UserCard name="Sam" :age="28" />
</template>
```

Import the child, then place its tag with the props as attributes. The crucial detail, carried over from `v-bind` in Chapter 3: `name="Philip"` (no colon) passes the literal *string* `"Philip"`, but `:age="35"` (with the colon) evaluates as an expression and passes the real *number* `35`. Forgetting the colon on a number/boolean/array prop passes a string instead — the same string-vs-expression distinction as React's quotes-vs-curly-braces and Angular's plain-vs-bracket attributes.

Typed Props with Defaults and Validation

```vue
const props = defineProps({
  name: { type: String, required: true },
  age: { type: Number, default: 0 },
  isAdmin: { type: Boolean, default: false },
});
```

The object form of `defineProps` adds type checking, a `required` flag, and `default` values — Vue warns in the console if a required prop is missing or the wrong type is passed. This is richer than React's plain props (which need PropTypes or TypeScript for the same), and comparable to Angular's typed `@Input`. Defaults work like Angular's, supplying a fallback when the parent omits the prop entirely.

Passing Different Data Types

```vue
<ProductCard
  title="Keyboard"           <!-- string -->
  :price="49.99"             <!-- number -->
  :in-stock="true"           <!-- boolean -->
  :tags="['new', 'sale']"    <!-- array -->
/>
```

Anything that isn't a plain string needs the `:` binding. Note also the naming convention: a prop declared as `inStock` (camelCase in JS) is passed as `:in-stock` (kebab-case in the template) — Vue maps between them

automatically, the standard HTML-attribute convention. You can use camelCase in the template too, but kebab-case is the common style.

PROPS ARE READ-ONLY — SAME AS EVERYWHERE

A child must never reassign a prop directly — Vue warns at runtime if you try. Props flow one way, parent to child, exactly as in React and Angular. If a child needs a local, editable copy, derive it (with a `computed`, or a `ref` initialized from the prop); to actually change the parent's data, emit an event — covered next chapter. Mutating an object/array prop's *contents* in place also works technically but is discouraged for the same reason: the parent loses control of its own data.

VUE	REACT	ANGULAR
`defineProps(['name'])`	Function params `{ name }`	`@Input() name`
`name="x" (string)`	`name="x"`	`name="x"`
`:age="35" (real value)`	`age={35}`	`[age]="35"`
`{ type, required, default }`	PropTypes / TS + defaults	Typed input + default

Coding Challenges

CHALLENGE 1

Build a MovieCard component with title and year props (declared via defineProps), rendering "Title (Year)". Use it three times in a parent with three different movies, passing year as a real number.

View solution

CHALLENGE 2

Build a Badge component using the object form of defineProps with a text prop (String, required) and a color prop (String, default "gray"). Render it once with a custom color and once without, confirming the default applies.

View solution

CHALLENGE 3

Build a ProductCard component receiving title (String), price (Number), and tags (Array) props, rendering the title, price, and the tags as a list. Pass tags as a real array using the : binding.

View solution

CHAPTER 6 QUICK REFERENCE

- **defineProps([...])** — declares a component's props; a compiler macro, no import needed
- Props used directly by name in the template; via `props.name` in the script
- **name="x"** passes a string; **:prop="expr"** passes a real value (number/boolean/array/object)
- Object form adds **type**, **required**, and **default** — richer than plain React props
- camelCase prop ↔ kebab-case attribute (`inStock` ↔ `:in-stock`) mapped automatically
- Props are **read-only** — flow one way; emit an event to change parent data (next chapter)
- **Next chapter:** emitting events with defineEmits, and v-model on components

CHAPTER 7 OF 12

Emitting Events and v-model on Components

CHAPTER 7

Emitting Events and v-model on Components

How a child talks back to its parent — and building a component that supports two-way binding

Props (Chapter 6) flow data *down* into a child. To send something back *up* — a click, a deletion, a new value — a child **emits an event** that the parent listens for. This is Vue's counterpart to React's "pass a callback down" and Angular's `@Output` / `EventEmitter`. Declared with the `defineEmits` macro.

Emitting an Event with defineEmits

```
<!-- TodoItem.vue -->
<script setup>
const props = defineProps(['id', 'text']);
const emit = defineEmits(['delete']);
</script>

<template>
  <li>
    {{ text }}
    <button @click="emit('delete', id)">Delete</button>
  </li>
</template>
```

```
<!-- parent template -->
<TodoItem
  :id="todo.id"
  :text="todo.text"
  @delete="removeTodo"
/>
```

`defineEmits(['delete'])` declares which events this component can emit and returns an `emit` function. Calling `emit('delete', id)` fires the `delete` event with `id` as its payload. The parent listens with the `@` event syntax from Chapter 3 — `@delete="removeTodo"` — and `removeTodo` receives the emitted `id` as its argument. Structurally identical to Angular's `@Output`, and the same role as a React callback prop, but modeled as a custom event the child raises (like Angular, unlike React).

CUSTOM EVENTS READ LIKE NATIVE ONES

Because the parent listens with `@delete` — the same syntax as `@click` — custom events are conventionally named as plain events (`delete`, `save`, `change`), not `onDelete`. From the parent's perspective `<TodoItem @delete="..." />` reads just like listening to a built-in element's event. (This mirrors Angular's `@Output` naming convention exactly.)

Multiple Events and Validated Emits

```
const emit = defineEmits(['save', 'cancel']);

// or the object form, documenting each event's payload:
const emit = defineEmits({
  save: (payload) => typeof payload === 'object',  // returns true if valid
  cancel: null,  // no validation
});
```

A component can declare several events in the array. The object form additionally lets you validate each event's payload with a function (Vue warns if it returns false) — useful documentation of what each event carries, comparable to typing an Angular `EventEmitter<T>`.

v-model on a Component

Chapter 3 used `v-model` on a native `<input>`. You can also put `v-model` on a *custom* component, to build something like a reusable input control with two-way binding. Under the hood, `v-model` on a component is shorthand for a `modelValue` prop plus an `update:modelValue` event:

```
<!-- CustomInput.vue -->
<script setup>
defineProps(['modelValue']);
const emit = defineEmits(['update:modelValue']);
</script>

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

```
<!-- parent: v-model just works -->
<CustomInput v-model="name" />
```

The child takes the current value via the `modelValue` prop, binds it to its real input with `:value`, and emits `update:modelValue` whenever the input changes. The parent then uses plain `v-model="name"` — Vue wires the prop and event together automatically. This is the exact convention Angular uses for custom two-way binding (`value` + `valueChange`), and it's why `v-model` on native inputs "just works": those are built on the same mechanism.

DEFINEMODEL — THE NEWER SHORTHAND

Recent Vue versions (3.4+) add `defineModel()`, which collapses the whole `modelValue`-prop-plus-event pattern above into a single line: `const model = defineModel()`, then bind `v-model="model"` on the inner input. It's the recommended modern approach for new code — worth knowing the longer form above too, since it's what `defineModel` expands to and what you'll see in slightly older code.

A CHILD STILL CAN'T MUTATE THE PROP DIRECTLY

Even when building a `v-model` component, the child must not reassign `modelValue` directly — it emits `update:modelValue` and lets the parent update the source of truth, which flows back down as the new prop. The one-way-data-flow discipline from Chapter 6 holds: data down via props, changes up via events.

VUE	ANGULAR	REACT
`defineEmits(['delete'])`	`@Output() delete`	A callback prop (`onDelete`)
`emit('delete', id)`	`this.delete.emit(id)`	`onDelete(id)`
`@delete="handler"`	`(delete)="handler($event)"`	`onDelete={handler}`
`v-model on component`	`[(value)]` (value + valueChange)	value prop + onChange prop

Coding Challenges

CHALLENGE 1

Build a TodoItem component with id/text props and a delete emit. The parent holds an array of todos, renders one TodoItem each, and removes the matching todo when a delete event fires.

View solution

CHALLENGE 2

Build a CustomInput component supporting v-model via a modelValue prop and an update:modelValue emit, then use v-model on it in a parent and display the bound value live.

View solution

CHALLENGE 3

Rewrite Challenge 2's CustomInput using the newer defineModel() shorthand, confirming the parent's v-model usage is unchanged.

View solution

CHAPTER 7 QUICK REFERENCE

- **defineEmits([...])** — declares emittable events; returns an `emit` function
- **emit('event', payload)** — fires an event upward; parent listens with `@event="handler"`
- Name events plainly (`delete`, `save`), not `onX` — they read like native events
- **v-model on a component** = a `modelValue` prop + an `update:modelValue` event
- **defineModel()** — the modern one-line shorthand for that whole pattern (Vue 3.4+)
- The child never mutates the prop — it emits, and the parent updates the source of truth
- **Next chapter:** slots — Vue's content-projection / composition mechanism

CHAPTER 8 OF 12

Slots

CHAPTER 8

Slots

Letting a parent pass markup into a child — Vue's content-projection mechanism

Props pass *data* into a child. **Slots** pass *markup* — letting a parent drop arbitrary template content inside a child component. This is Vue's direct equivalent of React's `children` (Fundamentals Chapter 9) and Angular's `<ng-content>`, and it's how you build flexible wrappers like cards, modals, and layouts that don't need to know what's placed inside them.

The Default Slot

```
<!-- Card.vue -->
<template>
  <div class="card">
    <slot></slot>
  </div>
</template>
```

```
<!-- parent -->
<Card>
  <h2>Hello!</h2>
  <p>This markup was passed in from outside.</p>
</Card>
```

The `<slot></slot>` element is a placeholder — whatever the parent writes between the component's opening and closing tags gets rendered there. `Card` doesn't know or care it's rendering an `<h2>` and a `<p>`; it just wraps whatever it's given. This is exactly React's `{children}`, just declared with a `<slot>` tag in the template rather than read as a prop.

Fallback (Default) Content

```
<slot>No content provided.</slot>
```

Content placed *inside* the `<slot>` tag is fallback content — shown only when the parent passes nothing. A small convenience React lacks built-in (you'd write `{children ?? 'fallback'}` by hand); in Vue it's just whatever you put between the slot tags.

Named Slots — Multiple Insertion Points

```
<!-- Panel.vue -->
<template>
  <div class="panel">
    <header><slot name="header"></slot></header>
    <main><slot></slot></main>           <!-- the default slot -->
    <footer><slot name="footer"></slot></footer>
  </div>
</template>
```

```
<!-- parent: target each slot with #name (v-slot shorthand) -->
<Panel>
  <template #header><h2>Title</h2></template>
  <p>Main body content (goes in the default slot).</p>
  <template #footer><small>Footer text</small></template>
</Panel>
```

When a component needs *several* insertion points, give each `<slot>` a `name`. The parent targets a named slot with `<template #header>` (the `#` is shorthand for `v-slot:header`); anything not wrapped in a named template goes into the default slot. This is cleaner than React's approach for multi-region layouts, where you'd pass JSX through several separate named props (the `SplitLayout` pattern from React Fundamentals Chapter 9) — Vue keeps it all between one set of component tags.

Scoped Slots — Passing Data Back to the Slot Content

```
<!-- TodoList.vue -->
<template>
  <ul>
    <li v-for="todo in todos" :key="todo.id">
      <slot :todo="todo"></slot>   <!-- expose each todo to the parent's slot content -->
    </li>
  </ul>
</template>
```

```vue
<!-- parent: receive the slot's data via v-slot -->
<TodoList :todos="todos">
  <template #default="{ todo }">
    <strong>{{ todo.text }}</strong>
  </template>
</TodoList>
```

A **scoped slot** lets the child pass data *up* to the markup the parent provides — the child binds values onto its `<slot>` (`:todo="todo"`), and the parent receives them by destructuring in `v-slot` (`#default="{ todo }"`). This is the powerful case: `TodoList` owns the looping and data, but the parent decides exactly how each item looks. It's the direct equivalent of React's render-props pattern (Advanced Chapter 5) — "the component owns behavior, the caller controls rendering" — but expressed declaratively in the template.

SLOTS COVER WHAT PROPS AND RENDER-PROPS DID SEPARATELY IN REACT

In React you'd use `children` for simple content, named props for multi-region layouts, and the render-props pattern for "expose data to the caller's markup." Vue's slot system handles all three with one consistent mechanism: default slot, named slots, and scoped slots. If render props felt clunky in React, scoped slots are the cleaner, template-native version of the same idea.

DON'T CONFUSE SLOTS (MARKUP) WITH PROPS (DATA)

A frequent early mix-up: trying to pass a chunk of markup through a prop, or trying to pass plain data through a slot. The rule is clean — **props for values, slots for markup**. If you want the parent to control *what's rendered*, that's a slot; if you want it to control *a value the child renders*, that's a prop. Scoped slots blur the line slightly (data flows back to slot markup), but the content itself is still markup.

VUE	REACT	ANGULAR
`<slot> (default)`	`{children}`	`<ng-content>`
Named slots (#header)	Named JSX props	`<ng-content select="...">`
Scoped slots	Render props	Structural directive / template context
Fallback in `<slot>`	Manual `children ?? fallback`	Default ng-content content

Coding Challenges

CHALLENGE 1

Build a Card component with a default slot wrapped in a styled div, then use it twice in a parent with completely different inner markup (a heading vs an image + caption). Add fallback slot text shown when no content is passed.

View solution

CHALLENGE 2

Build a Panel component with named header and footer slots plus a default slot for the body, and fill all three from a parent using <template #header> / default content / <template #footer>.

View solution

CHALLENGE 3

Build a List component that takes an items array prop, loops over it, and exposes each item to a scoped slot (:item="item"). In the parent, use v-slot to render each item your own way (e.g. bold name + muted id).

View solution

CHAPTER 8 QUICK REFERENCE

- **<slot></slot>** — placeholder for parent-provided markup (= React's `children`)
- Content inside the `<slot>` tag is **fallback**, shown when the parent passes nothing
- **Named slots** (`<slot name="header">`) — multiple insertion points; parent targets with `#header`
- **Scoped slots** — child binds data onto `<slot :x="x">`; parent receives via `#default="{ x }"` (= render props)
- Rule of thumb: **props for values, slots for markup**
- One slot mechanism covers what React split across children, named props, and render props
- **Next chapter:** lifecycle hooks — onMounted, onUnmounted, and friends

CHAPTER 9 OF 12

Lifecycle Hooks

CHAPTER 9

Lifecycle Hooks

Running code at key moments in a component's life — setup, mount, and teardown

A component is created, mounted to the DOM, updates as its data changes, and is eventually unmounted. **Lifecycle hooks** let you run code at these moments — most often to do setup work after mounting (like fetching data) and cleanup before unmounting (like clearing a timer). In the Composition API, these are functions you call inside `<script setup>`.

onMounted — After the Component Is in the DOM

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

const users = ref([]);

onMounted(async () => {
  const res = await fetch('/api/users');
  users.value = await res.json();
});
</script>
```

`onMounted` registers a callback that runs once, right after the component is inserted into the DOM — the standard place for an initial data fetch, the equivalent of React's `useEffect(() => {...}, [])` and Angular's `ngOnInit`. By this point the real DOM exists, so it's also safe to access template elements (via template refs) here. Note you can pass an `async` function directly — Vue doesn't have React's restriction about effect callbacks not being async.

CODE AT THE TOP LEVEL OF <SCRIPT SETUP> IS THE "CREATED" PHASE

Unlike React (where everything runs inside the component function every render) or Angular (where the constructor and `ngOnInit` are distinct), much of Vue's setup just lives at the top level of `<script setup>` — refs, computeds, and functions declared there run once when the component is created, *before* it mounts. You only reach for `onMounted` specifically when you need the DOM to exist first (or want to defer work until after the first render).

onUnmounted — Cleanup Before Removal

```
import { onMounted, onUnmounted } from 'vue';

let intervalId;

onMounted(() => {
  intervalId = setInterval(() => console.log('tick'), 1000);
});

onUnmounted(() => {
  clearInterval(intervalId);  // stop the timer when the component is removed
});
```

`onUnmounted` runs just before the component is destroyed — the place for cleanup, mirroring React's `useEffect` cleanup function and Angular's `ngOnDestroy`. The same universal rule applies: anything that "starts" something ongoing (a timer, an event listener, a subscription) needs a matching "stop" here, or it leaks. A `setInterval` not cleared in `onUnmounted` keeps firing after the component is gone.

onUpdated — After a Re-render

```
import { onUpdated } from 'vue';

onUpdated(() => {
  console.log('the component re-rendered and the DOM is updated');
});
```

`onUpdated` runs after the component re-renders due to a reactive change and the DOM has been patched. It's used less often than `onMounted` / `onUnmounted` — for most "react to a value changing" needs, a `watch` (Chapter 5) targeting the specific value is cleaner and more precise than this catch-all "something updated" hook.

The Common Pattern — Mount + Cleanup Together

```
onMounted(() => {
  window.addEventListener('resize', handleResize);
});

onUnmounted(() => {
  window.removeEventListener('resize', handleResize);
});
```

Setup in `onMounted` paired with matching teardown in `onUnmounted` is the everyday pattern — adding then removing an event listener, starting then stopping a subscription. In Chapter 10 you'll see how a **composable** bundles this paired logic into one reusable function, so the mount/unmount pairing lives in one place rather than scattered across every component that needs it — Vue's equivalent of a React custom hook with a cleanup return.

HOOKS MUST BE CALLED SYNCHRONOUSLY IN SETUP

Lifecycle hooks like `onMounted` must be registered synchronously at the top level of `<script setup>` — not inside an `if`, a callback, or after an `await`. This is the same rule-of-hooks constraint as React: Vue needs to associate the hook with the current component instance at setup time, which only works if it's called during the synchronous setup run. Calling `onMounted` inside an async function after an `await`, for instance, silently fails to register.

VUE	REACT	ANGULAR
top of <script setup>	component function body	constructor
onMounted	`useEffect(..., [])`	`ngOnInit`
onUpdated	`useEffect` (no dep array)	`ngOnChanges` / `ngDoCheck`
onUnmounted	`useEffect` cleanup	`ngOnDestroy`

Coding Challenges

CHALLENGE 1

Build a component that fetches a list from any free public API in onMounted and stores it in a ref for display, with a loading state shown until the data arrives.

View solution

CHALLENGE 2

Build a Clock component that starts a setInterval in onMounted (updating a time ref every second) and clears it in onUnmounted. Toggle the component with v-if in a parent to confirm the ticking stops when removed.

View solution

CHALLENGE 3

Build a component that tracks the window's width: add a resize event listener in onMounted (updating a width ref), and remove it in onUnmounted. Display the live width.

📄 View solution

CHAPTER 9 QUICK REFERENCE

- Refs/computeds/functions at the top of `<script setup>` run at **creation**, before mount
- **onMounted** — runs once after the component is in the DOM (= `useEffect(.., [])` / `ngOnInit`)
- **onUnmounted** — runs before removal; the place for cleanup (= cleanup fn / `ngOnDestroy`)
- **onUpdated** — after a re-render; usually a precise `watch` is better
- Pair setup in `onMounted` with teardown in `onUnmounted` (listeners, timers, subscriptions)
- Register hooks **synchronously** in setup — not after an `await` or inside a conditional
- **Next chapter:** composables — bundling reusable stateful logic (Vue's custom hooks)

CHAPTER 10 OF 12

Composables

CHAPTER 10

Composables

Extracting reusable stateful logic into a function — Vue's answer to React custom hooks

A **composable** is a function that uses Vue's reactivity (`ref` , `computed` , lifecycle hooks) to package up reusable, stateful logic — exactly what React custom hooks do, and what Vue calls its own version of the pattern. There's no special syntax: it's just a plain function, by convention named `useSomething` , that calls reactivity APIs inside and returns reactive values.

A First Composable — useCounter

```
// composables/useCounter.js
import { ref } from 'vue';

export function useCounter(initial = 0) {
  const count = ref(initial);

  function increment() { count.value++; }
  function reset() { count.value = initial; }

  return { count, increment, reset };
}
```

```
<!-- usage in any component -->
<script setup>
import { useCounter } from './composables/useCounter';
const { count, increment, reset } = useCounter();
</script>

<template>
  <p>{{ count }}</p>
  <button @click="increment">+1</button>
</template>
```

`useCounter` bundles a `ref` and its behaviour into one reusable function, returning the reactive `count` and the functions to change it. Any component imports it and destructures what it needs — and crucially, each call

creates its own independent state, just like calling a React hook. This maps directly onto React's `useToggle` from Intermediate Chapter 4; the only real syntax difference is returning an object (so you destructure by name) rather than a tuple.

A Composable with Lifecycle — useMousePosition

```
// composables/useMousePosition.js
import { ref, onMounted, onUnmounted } from 'vue';

export function useMousePosition() {
  const x = ref(0);
  const y = ref(0);

  function update(e) {
    x.value = e.pageX;
    y.value = e.pageY;
  }

  onMounted(() => window.addEventListener('mousemove', update));
  onUnmounted(() => window.removeEventListener('mousemove', update));

  return { x, y };
}
```

This is the real payoff teased in Chapter 9: the `onMounted` / `onUnmounted` listener-pairing — the kind of thing you'd otherwise repeat in every component needing mouse position — lives in *one* composable. A composable can call lifecycle hooks itself, and Vue ties them to whichever component is using it. Any component just does `const { x, y } = useMousePosition()` and gets a self-cleaning, reactive position — the setup and teardown handled automatically.

useLocalStorage — A Persisted ref

```
// composables/useLocalStorage.js
import { ref, watch } from 'vue';

export function useLocalStorage(key, initial) {
  const saved = localStorage.getItem(key);
  const value = ref(saved ? JSON.parse(saved) : initial);

  watch(value, (newVal) => {
    localStorage.setItem(key, JSON.stringify(newVal));
  }, { deep: true });

  return value;
}
```

The same `useLocalStorage` that was a React custom hook (Intermediate Chapter 4) and an Angular signal+effect (Angular Chapter 13) — here, a composable combining a `ref` initialized from storage with a `watch` that re-saves on every change. `const notes = useLocalStorage('notes', [])` returns a normal-looking `ref` that transparently persists. The `{ deep: true }` option makes the watch fire on nested changes inside objects/arrays, not just reassignment.

COMPOSABLES VS RENDER-LESS COMPONENTS VS MIXINS

Older Vue 2 code shared logic with **mixins** — an approach with the same naming-collision and "where did this property come from?" problems that pushed React away from HOCs/mixins toward hooks. Composables are Vue 3's clear successor, for the same reasons React hooks replaced those patterns: explicit imports, no hidden merging, and each call gets isolated state. If you see a `mixins:` option in old code, a composable is the modern replacement.

RETURN REFS, NOT UNWRAPPED VALUES

A composable should return the `ref`s themselves (or a reactive object), not `ref.value`. Returning `count.value` hands back a plain number — a one-time snapshot that won't stay reactive in the component. Return `count` (the ref) so the component keeps the live, reactive connection. This is the composable equivalent of the same care React takes to return state and its setter, not a stale value.

VUE COMPOSABLE	REACT CUSTOM HOOK	ANGULAR EQUIVALENT
useCounter()	`useToggle()` / custom hook	Injectable service
Returns an object (destructure)	Returns a tuple/object	Class instance
Can call onMounted/onUnmounted	Can call useEffect	Lifecycle in the service host
Each call = isolated state	Each call = isolated state	One shared instance (singleton)

Coding Challenges

CHALLENGE 1

Write a useCounter composable returning { count, increment, decrement, reset }, and use it in two separate components — confirming each gets its own independent count.

View solution

CHALLENGE 2

Write a useWindowWidth composable that tracks window.innerWidth via a resize listener added in onMounted and removed in onUnmounted, returning the reactive width — then use it in a component, confirming the listener cleans up when the component is removed.

View solution

CHALLENGE 3

Write a useLocalStorage composable that returns a ref initialized from localStorage and persists it on change via a watch, then use it to persist a counter's value so it survives a page refresh.

View solution

CHAPTER 10 QUICK REFERENCE

- A **composable** is a function (conventionally `useX`) packaging reusable, stateful reactive logic — Vue's custom hook
- No special syntax — call `ref` / `computed` /lifecycle hooks inside, return reactive values
- Returns an **object** (destructure by name); each call gets **isolated state**
- A composable can call `onMounted` / `onUnmounted` — bundling setup+teardown in one place
- Return the **refs**, not `ref.value`, to keep the reactive connection live
- Composables replace Vue 2 **mixins**, for the same reasons hooks replaced HOCs/mixins in React
- **Next chapter:** Vue Router — multiple pages in a single-page app

CHAPTER 11 OF 12

Vue Router

CHAPTER 11

Vue Router

Multiple pages in a single-page app — the official routing library, deeply familiar by now

Routing in Vue is its own official package (`vue-router`), like React Router was a separate install — but the concepts are now thoroughly familiar from both the React and Angular courses. Define routes mapping paths to components, render the matched one in an outlet, link without reloading, read params, navigate from code, and guard routes. Same ideas, Vue's API.

Defining Routes

```
// router/index.js  (npm install vue-router)
import { createRouter, createWebHistory } from 'vue-router';
import Home from '../views/Home.vue';
import About from '../views/About.vue';

const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  { path: '/product/:id', component: ProductDetail },
  { path: '/:pathMatch(.*)*', component: NotFound },
];

export const router = createRouter({
  history: createWebHistory(),
  routes,
});
```

Routes are an array of `{ path, component }` objects, the same shape as Angular's route config and React Router's `<Route>` list. `:id` is a dynamic segment; `/:pathMatch(.*)*` is Vue Router's (slightly cryptic) catch-all for unmatched URLs. `createWebHistory()` enables clean URLs (no `#`). The router is then registered in `main.js` with `app.use(router)` — the same plugin pattern Pinia will use next chapter.

RouterView and RouterLink

```vue
<!-- App.vue -->
<template>
  <nav>
    <RouterLink to="/">Home</RouterLink>
    <RouterLink to="/about">About</RouterLink>
  </nav>
  <RouterView />
</template>
```

`<RouterView />` is the outlet where the matched route's component renders — the equivalent of React Router's `<Outlet />` and Angular's `<router-outlet>`. `<RouterLink to="/about">` navigates without a full page reload, like React's `<Link>` and Angular's `routerLink`. A `RouterLink` automatically gets an active CSS class when its route is current, so highlighting the active nav item needs no manual URL comparison.

Reading Route Params with useRoute

```vue
<!-- ProductDetail.vue -->
<script setup>
import { useRoute } from 'vue-router';

const route = useRoute();
const id = route.params.id;
</script>

<template>
  <p>Showing product {{ route.params.id }}</p>
</template>
```

`useRoute()` returns the current route object; `route.params.id` reads the `:id` segment — the equivalent of React Router's `useParams()` and Angular's `ActivatedRoute.paramMap`. The `route` object is reactive, so if the same component stays mounted while only the param changes (navigating `/product/1` → `/product/2`), a `watch` (Chapter 5) on `route.params.id` reacts to that change.

Programmatic Navigation with useRouter

```
import { useRouter } from 'vue-router';

const router = useRouter();

function goToProduct(id) {
  router.push(`/product/${id}`);
  // or: router.push({ path: '/product/' + id })
}
```

`useRouter()` returns the router instance; `router.push(...)` navigates from code — after a form submits, a login succeeds, an action completes — the equivalent of React Router's `useNavigate()` and Angular's `Router.navigate()`. Note the two distinct hooks: `useRoute` (singular, the current route's data) vs `useRouter` (the router, for navigating) — an easy pair to mix up by name.

Navigation Guards

```
// a global guard, in router/index.js
router.beforeEach((to, from) => {
  const isLoggedIn = checkAuth();
  if (to.path === '/dashboard' && !isLoggedIn) {
    return '/login';  // redirect by returning a path
  }
  // return true or nothing to allow
});
```

A **navigation guard** runs before each route change — `beforeEach` receives the target (`to`) and current (`from`) routes, and either allows navigation (return `true` or nothing) or redirects (return a path). This is Vue's equivalent of Angular's `canActivate` guard, and the standard way to keep unauthenticated users out of protected pages. Guards can also be defined per-route (`beforeEnter`) or inside a component.

LAZY-LOADING ROUTES — CODE SPLITTING BUILT IN

Like Angular's `loadComponent`, Vue Router supports lazy-loading a route's component so its code only downloads when the route is first visited: `{ path: '/admin', component: () => import('../views/Admin.vue') }`. A dynamic `import()` as the `component` is all it takes — no separate `Suspense` boundary needed (the React equivalent from Advanced Chapter 3). This is the standard way to keep the initial bundle small.

ROUTERLINK, NOT A PLAIN ANCHOR

Using `<a href="/about">` instead of `<RouterLink to="/about">` triggers a real full-page reload, throwing away all in-memory state — the same warning as React Router and Angular. `RouterLink` intercepts the click and updates the URL via the History API without a reload. Always use it for in-app navigation.

VUE ROUTER	REACT ROUTER	ANGULAR
`routes array + createRouter`	`<Routes>` / `<Route>`	routes + provideRouter
`<RouterView />`	`<Outlet />`	`<router-outlet>`
`<RouterLink to>`	`<Link to>`	`routerLink`
`useRoute().params`	`useParams()`	`ActivatedRoute.paramMap`
`useRouter().push()`	`useNavigate()`	`Router.navigate()`
`beforeEach guard`	Hand-rolled wrapper	`canActivate`

Coding Challenges

CHALLENGE 1

Set up a 3-page app (Home, About, Contact) with a routes array, a nav using RouterLink, a RouterView, and a catch-all route showing a NotFound component for unmatched URLs.

View solution

CHALLENGE 2

Add a /product/:id route. From a product list, use RouterLink to navigate to a detail page that reads the id via useRoute and displays the matching product.

View solution

CHALLENGE 3

Add a beforeEach navigation guard protecting a /dashboard route, redirecting to /login when a simple isLoggedIn flag is false, with a button toggling the flag to test both outcomes.

View solution

CHAPTER 11 QUICK REFERENCE

- **createRouter** + a routes array (path → component); register with `app.use(router)`
- **<RouterView />** renders the matched route; **<RouterLink to>** navigates without reload
- **:id** dynamic segments; **/:pathMatch(.*)*** is the catch-all route
- **useRoute()** — current route data (`route.params.id`); **useRouter()** — navigate (`router.push`)

- **beforeEach((to, from) => ...)** — a navigation guard; return a path to redirect, true/nothing to allow
- Lazy-load a route with `component: () => import('...')` — built-in code splitting
- Always use `RouterLink`, never a plain `<a href>`, for in-app navigation
- **Next chapter:** state management with Pinia, plus a data-fetching pattern (final chapter)

CHAPTER 12 OF 12

State Management with Pinia, and Data Fetching

CHAPTER 12

State Management with Pinia, and Data Fetching

The final chapter — global state the official way, plus tying fetching into a store

For state shared across many distant components, Vue's official answer is **Pinia**. Like Zustand and Redux for React, or a service for Angular, it holds state outside the component tree so any component can reach it — but a Pinia store is built from the very same reactivity primitives you already know (`ref`, `computed`), making it feel like a composable that happens to be global.

Defining a Store

```
// stores/counter.js  (npm install pinia)
import { defineStore } from 'pinia';
import { ref, computed } from 'vue';

export const useCounterStore = defineStore('counter', () => {
  const count = ref(0);
  const double = computed(() => count.value * 2);

  function increment() { count.value++; }

  return { count, double, increment };
});
```

`defineStore('counter', () => {...})` — the "setup store" form — is essentially a composable: declare `ref`s (the state), `computed`s (derived values, called "getters"), and functions (the "actions") inside, and return them. The first argument is a unique store id. If this looks almost identical to a composable from Chapter 10, that's the point — the difference is that a Pinia store is a single shared instance, not fresh state per call.

Setup, and Using the Store

```
// main.js
import { createPinia } from 'pinia';
createApp(App).use(createPinia()).mount('#app');
```

```
<!-- any component, anywhere -->
<script setup>
import { useCounterStore } from './stores/counter';
const store = useCounterStore();
</script>

<template>
  <p>{{ store.count }} (double: {{ store.double }})</p>
  <button @click="store.increment">+1</button>
</template>
```

Register Pinia once with `app.use(createPinia())` (the same plugin pattern as the router). Then any component calls `useCounterStore()` and accesses `store.count`, `store.double`, `store.increment` — and two completely unrelated components calling it share the exact same state, no props or events between them. This is the same problem Context/Zustand solved in React and a service solved in Angular, here with Vue's own reactivity underneath.

DESTRUCTURING A STORE LOSES REACTIVITY — USE STORETOREFS

Writing `const { count } = useCounterStore()` breaks reactivity — `count` becomes a disconnected snapshot (the same `reactive`-destructuring trap from Chapter 2). To pull out reactive state, use `storeToRefs`: `const { count, double } = storeToRefs(store)`. Actions (functions) can be destructured normally — only state and getters need `storeToRefs`.

Data Fetching Inside a Store

```
// stores/users.js
export const useUsersStore = defineStore('users', () => {
  const users = ref([]);
  const status = ref('idle');  // idle | loading | error | success

  async function fetchUsers() {
    status.value = 'loading';
    try {
      const res = await fetch('/api/users');
      if (!res.ok) throw new Error('Request failed');
      users.value = await res.json();
      status.value = 'success';
    } catch (e) {
      status.value = 'error';
    }
  }

  return { users, status, fetchUsers };
});
```

An **action** can be `async` — so a store is a natural home for API calls, exactly as a service was in Angular (Chapter 11). The `status` ref drives loading/error/success rendering, the same single-status pattern used across the React projects. A component calls `store.fetchUsers()` in `onMounted` and reads `store.users` / `store.status` in the template — and because the result lives in the store, several components share the fetched data without re-requesting it, the same benefit React Query gave (React Advanced Chapter 2).

PINIA VS A PLAIN COMPOSABLE FOR SHARED STATE

You could share state with a composable that creates its `ref`s in module scope (outside the function) — and for small cases that works. Pinia adds what a real app wants on top: a single well-defined store id, devtools integration (time-travel, inspecting state), server-side-rendering support, and plugins. Reach for a store when state is genuinely global and app-significant; a plain composable stays fine for localized reusable logic.

VUE (PINIA)	REACT	ANGULAR
`defineStore + useXStore()`	Zustand `create()` / Context	Injectable service
`state = ref(...)`	store state	service properties
`getters = computed(...)`	derived/selectors	getters / computed
`actions = functions`	store actions	service methods
`async action for fetching`	React Query / thunk	service + HttpClient

Coding Challenges

CHALLENGE 1

Set up Pinia and define a counter store (state: count; getter: double; actions: increment, reset). Use it from two unrelated components, confirming both share the same count with nothing passed between them.

View solution

CHALLENGE 2

Build a cart store (state: items; action addToCart with the duplicate-quantity check; getter totalItems) and use it from a product list and a separate cart-display component. Use storeToRefs to read items reactively.

View solution

CHALLENGE 3

Build a users store with an async fetchUsers action and a status ref (idle/loading/error/success). A component calls fetchUsers in onMounted and renders loading/error/success states from the store.

View solution

CHAPTER 12 QUICK REFERENCE

- **Pinia** — Vue's official global state; a store is essentially a shared, named composable
- **defineStore('id', () => {...})** — state (`ref`), getters (`computed`), actions (functions), then return them
- Register once with `app.use(createPinia())` ; use via `useXStore()` in any component
- Two components using the same store share one instance — no props, no events
- **storeToRefs(store)** — destructure state/getters reactively (actions destructure normally)
- An **async action** makes a store a natural home for API calls + loading/error/success state
- Use a store for genuinely global state; a plain composable for localized reusable logic

🎉 That completes the Vue course — all 12 chapters, from the SFC and reactivity through composables, routing, and Pinia.