tina4-js — The 1.5KB Reactive Core

Signals, Web Components, and a frontend you can fit in your head

v3.12.4 • tina4.com

The Intelligent Native Application 4ramework

Table of Contents

Getting Started

Your First 5 Minutes

A terminal. A single command. A browser tab. In five minutes you will have a running tina4-js project with reactive state, a rendered page, and a dev server with hot reload. The counter on screen will respond to your clicks before you understand why. That is the point -- you ship first, then you learn.

---

1. What Is tina4-js

tina4-js is a reactive JavaScript framework with a 1.5KB gzipped core. The full framework ships under 6KB gzipped. Eight modules, each solving one problem:

- **Signals** for reactive state -- no stores, no reducers, no actions
- **Tagged template literals** for DOM rendering -- no JSX, no virtual DOM, no compiler
- **Web Components** for encapsulation -- real custom elements, not a framework abstraction
- **A router** for SPA navigation
- **An HTTP client** built for tina4-php and tina4-python backends
- **WebSocket** with auto-reconnect and signal integration
- **PWA** support with one function call
- **A debug overlay** that shows you everything

One npm package. Zero dependencies. The entire framework weighs less than most favicons.

What It Is Not

tina4-js is not React. It has no virtual DOM. It does not diff trees. When a signal changes, the exact text node or attribute that depends on it updates. Nothing else moves.

tina4-js is not Angular. No dependency injection. No decorators. No module system. No zone.js.

tina4-js is not a meta-framework. No server-side rendering. No file-based routing. No data loading conventions. It is a client-side framework that does one thing well: build reactive UIs with the smallest possible footprint.

---

2. Prerequisites

You need three things. Nothing else.

- **Node.js 18 or later** -- check with:

```
node --version
```

You should see `v18.0.0` or higher.

- **npm** -- comes with Node.js. Check with:

```
npm --version
```

- **A text editor** -- VS Code, Cursor, Zed, Vim, anything.

---

3. Create a Project

The primary way to scaffold is the Tina4 Rust CLI:

```
tina4 init js my-app
```

This scaffolds a complete project with TypeScript, Vite, routing, and a sample page.

> ***The `tina4` Rust CLI is the unified installer across all Tina4 frameworks*** *(Python, PHP, Ruby, Node.js, JS). For npm-only environments where the CLI isn't available, every command has an `npx tina4js …` fallback -- for example `npx tina4js create my-app` is equivalent to `tina4 init js my-app`.*

Want the optional CSS framework included? The Rust CLI does not yet accept a `--css` flag on `init`, so scaffold first and then add the dependency:

```
tina4 init js my-app
```

To add Tina4 CSS, edit `package.json` to add the `tina4-css` dependency, or use the fallback `npx tina4js create my-app --css` which accepts the flag directly. This adds `tina4-css` to your dependencies -- a utility CSS library with reset, grid, buttons, forms, tables, cards, and dark mode built in. More on this in Chapter 10.

Want PWA support from the start? The Rust CLI also does not yet expose a `--pwa` flag on `init`. Scaffold with `tina4 init js my-app` and enable PWA manually (see Chapter 9), or use the fallback `npx tina4js create my-app --pwa` to get the PWA preset directly. You can combine the flags on the fallback: `npx tina4js create my-app --css --pwa`.

Now install and run:

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

Open `http://localhost:3000`. A welcome page appears with a counter. Click the minus button. Click the plus button. The number updates. No page reload. No visible delay. That is signals at work -- reactive state flowing from data to DOM without you writing a single line of update logic.

---

4. Project Structure

Look at what the CLI created:

```
my-app/
  index.html              # Entry point -- loads src/main.ts
  package.json            # Dependencies: tina4js, vite, typescript
```

```
tsconfig.json           # TypeScript config
vite.config.ts          # Vite dev server config
src/
  main.ts               # App entry -- imports routes, starts router
  routes/
    index.ts            # Route definitions
  pages/
    home.ts             # Home page handler
  components/
    app-header.ts       # Example web component
  public/
    css/
      default.css       # Default styles
```

The structure is intentional:

- `src/routes/` -- Route definitions. One file per feature or group.
- `src/pages/` -- Page handler functions. Each returns a template.
- `src/components/` -- Web Components (Tina4Element subclasses).
- `src/public/` -- Static assets (CSS, images, fonts).

This is a convention, not a requirement. tina4-js does not care where your files live. But this structure scales. Every tina4-js project looks the same, which matters when you onboard new team members or let AI generate code. Consistency compounds.

---

5. The Entry Point

Open `src/main.ts`:

```
import { signal, computed, html, route, router, navigate, api } from 'tina4js';
import './routes/index';

// Configure API (uncomment to connect to tina4-php/python backend)
// api.configure({ baseUrl: '/api', auth: true });

// Start router
router.start({ target: '#root', mode: 'hash' });
```

Three things happen here:

- **Import everything from one package.** All seven modules export from `tina4js`. No sub-package installs.
- **Import your routes.** The route file calls `route()` to register paths and handlers.
- **Start the router.** It finds `#root` in the DOM and renders matched routes into it.

The `mode: 'hash'` means URLs look like `http://localhost:3000/#/about`. For clean URLs without the hash, use `mode: 'history'` -- but you will need server-side URL rewriting in production.

---

6. Your First Signal

Open `src/pages/home.ts`:

```
import { signal, computed, html } from 'tina4js';

export function homePage() {
  const count = signal(0);
  const doubled = computed(() => count.value * 2);

  return html`
    <div class="page">
      <h1>Welcome</h1>

      <div class="counter">
        <button @click=${() => count.value--}>-</button>
        <span>${count}</span>
        <button @click=${() => count.value++}>+</button>
      </div>
      <p class="muted">Doubled: ${doubled}</p>
    </div>
  `;
}
```

Four things happen in this code. Each one is a core concept you will use in every tina4-js application.

`signal(0)` creates a reactive value. Read it with `count.value`. Write it with `count.value = 5`. When you write, everything that depends on it updates. Not eventually. Not on the next tick. Right now.

`computed(() => count.value * 2)` creates a derived signal. It reads `count.value` inside the function, so tina4-js knows to recompute whenever `count` changes. You cannot write to a computed -- it is read-only.

`html\...\`` is a tagged template literal. It returns a real DocumentFragment `-- actual DOM nodes, not a string, not a virtual tree. When you put` ${count} `in the template, tina4-js creates a text node that updates in place when` count` changes. No diffing. No reconciliation. Direct DOM mutation.

`@click=${() => count.value--}` adds a click event listener. The `@` prefix means "event handler." Since v1.0.9, event handlers are wrapped in `batch()`, so multiple signal writes inside one handler trigger one update.

---

7. Your First Route

Open `src/routes/index.ts`:

```
import { route, html } from 'tina4js';
import { homePage } from '../pages/home';

// Home
route('/', homePage);
```

```
// About
route('/about', () => html`
  <div class="page">
    <h1>About</h1>
    <p>Built with tina4-js.</p>
    <a href="/">Back home</a>
  </div>
`);

// 404
route('*', () => html`
  <div class="page">
    <h1>404</h1>
    <p>Page not found.</p>
    <a href="/">Go home</a>
  </div>
`);
```

`route(pattern, handler)` -- pattern is always the first argument. The handler is a function that returns a template. The router calls it when the URL matches.

`route('*', handler)` -- the wildcard catches any URL that no other route matched. Put it last.

Links work automatically. The router intercepts clicks on `<a>` tags that point to same-origin paths and navigates without a page reload. You do not need a special `<Link>` component.

---

8. Adding a Component

Open `src/components/app-header.ts`:

```
import { Tina4Element, html } from 'tina4js';

class AppHeader extends Tina4Element {
  static props = { title: String };
  static styles = `
    :host { display: block; padding: 1rem 0; border-bottom: 1px solid #e5e7eb; }
    h1 { margin: 0; font-size: 1.5rem; }
    nav { display: flex; gap: 1rem; margin-top: 0.5rem; }
    a { color: #2563eb; text-decoration: none; }
  `;

  render() {
    return html`
      <h1>${this.prop('title')}</h1>
      <nav>
        <a href="/">Home</a>
        <a href="/about">About</a>
      </nav>
    `;
  }
}

customElements.define('app-header', AppHeader);
```

Then use it in any template:

```
html`
```

```
    <app-header title="My App"></app-header>
    <div class="content">...</div>
  `
```

`static props` declares reactive attributes. When the `title` attribute changes, the component re-renders that part.

`static styles` scopes CSS to the component via Shadow DOM. Your styles cannot leak out. External styles cannot leak in.

`this.prop('title')` returns a signal for the `title` prop. Drop it in the template and it updates reactively.

`customElements.define()` registers the tag name. Use a hyphenated name -- that is a Web Components requirement, not a tina4-js thing.

---

9. The Build

Development is done. Time to ship:

```
npm run build
```

Vite bundles everything into `dist/`. The tina4-js runtime adds under 6KB to your bundle. Your entire app -- framework, routes, components, everything -- will likely be smaller than React's runtime alone.

To preview the production build:

```
npm run preview
```

---

10. What Just Happened

Five minutes. One command to scaffold. One command to install. One command to run. And you covered:

- Reactive state with `signal()`
- Derived state with `computed()`
- DOM rendering with `html` tagged templates
- Event handling with `@click`
- A web component with `Tina4Element`
- Client-side routing with `route()` and `router.start()`
- A production build with Vite

The rest of this book goes deep on each of these. But you already have a working app. You already have a production build. Everything from here is precision and power.

Summary

What	How
Create project	`tina4 init js my-app` (fallback: `npx tina4js create my-app`)
With CSS framework	`tina4 init js my-app` + add `tina4-css` dep (fallback: `npx tina4js create my-app --css`)
With PWA	`tina4 init js my-app` + enable PWA manually (fallback: `npx tina4js create my-app --pwa`)
Dev server	`npm run dev`
Production build	`npm run build`
Reactive state	`signal(initialValue)`
Derived state	`computed(() => expression)`
DOM rendering	`` ` html<p>${signal}</p> ` ``
Event handling	`@click=${handler}`
Components	`class X extends Tina4Element`
Routes	`route(pattern, handler)`
Start router	`router.start({ target, mode })`

Signals

Reactive State Without the Drama

A variable changes. The DOM does not. You write `document.getElementById('counter').textContent = count` in twelve places. You miss one. The bug report arrives on Friday afternoon.

Signals end this. A signal is a reactive value. Change it, and everything that depends on it -- text nodes, attributes, computed values, side effects -- updates. You never write update logic again.

This chapter covers the entire reactivity system in tina4-js: creating signals, deriving values, running side effects, batching updates, and avoiding the mistakes that trip up every new user.

---

1. What Is a Signal

A signal is a reactive value. When it changes, everything that depends on it updates.

```
import { signal } from 'tina4js';

const count = signal(0);

// Read
console.log(count.value); // 0

// Write
count.value = 5;
console.log(count.value); // 5
```

That is the entire API for basic signals. `.value` to read. `.value =` to write. Nothing else.

When you write a new value, every subscriber -- effects, computed signals, DOM bindings -- fires. Not eventually. Not on the next tick. Synchronously, right now.

---

2. Why Object.is() Matters

Signals use `Object.is()` to decide if a value changed. If the new value is the same as the old value by `Object.is()`, nothing happens. No subscribers fire. No DOM updates.

This matters for objects and arrays:

```
const items = signal(['apple', 'banana']);

// WRONG -- mutating in place, same reference, no update
items.value.push('cherry');
// Object.is(oldArray, newArray) === true -- nothing happens!

// RIGHT -- new array reference
items.value = [...items.value, 'cherry'];
// Object.is(oldArray, newArray) === false -- subscribers fire!
```

Same rule for objects:

```
const user = signal({ name: 'Alice', age: 30 });

// WRONG -- mutating in place
user.value.age = 31;

// RIGHT -- new object reference
user.value = { ...user.value, age: 31 };
```

This is the single most common mistake in tina4-js. UI not updating? Check this first. New reference, or mutation in place?

The rule fits on an index card: **always assign a new value to `.value`.** Spread arrays. Spread objects. Never mutate.

---

3. computed() -- Derived State

A shopping cart has items. Each item has a price and a quantity. The total is derived from those values. You do not store the total -- you compute it. When an item changes, the total recalculates.

That is what `computed()` does. It derives a value from other signals and keeps it current.

```
import { signal, computed } from 'tina4js';

const price = signal(10);
const quantity = signal(3);
const total = computed(() => price.value * quantity.value);

console.log(total.value); // 30

price.value = 20;
console.log(total.value); // 60
```

The function you pass to `computed()` runs immediately and then re-runs whenever any signal it reads changes. tina4-js tracks which signals were read inside the function -- you do not need to declare dependencies manually.

Computed Is Eager, Not Lazy

This is different from some other frameworks. In tina4-js, `computed()` re-evaluates as soon as a dependency changes, not when you read `.value`. This means:

- Computed values are always up to date
- There is no stale-read problem
- But unnecessary computeds do cost CPU

If you have an expensive computation that is only needed sometimes, do not put it in a computed. Use an effect with a conditional instead.

Computed Is Read-Only

```
const total = computed(() => price.value * quantity.value);

total.value = 100; // throws Error: '[tina4] computed signals are read-only'
```

If you need a writable derived value, use a signal and an effect.

---

4. effect() -- Side Effects

An effect runs a function immediately, then re-runs it whenever any signal it reads changes.

```
import { signal, effect } from 'tina4js';

const name = signal('World');

const dispose = effect(() => {
  console.log(`Hello, ${name.value}!`);
});
// logs: "Hello, World!"

name.value = 'Tina4';
// logs: "Hello, Tina4!"
```

Effects bridge reactive state to the outside world. Logging. DOM manipulation. Network requests. Analytics. Anything that needs to happen when data changes -- effects handle it.

Disposing Effects

`effect()` returns a dispose function. Call it to stop the effect from running:

```
const dispose = effect(() => {
  document.title = `Count: ${count.value}`;
});

// Later, when you are done:
dispose();

count.value = 99; // effect does NOT run
```

This matters for cleanup. Component unmounts. Route changes. Without disposal, effects keep running -- memory leaks and ghost updates accumulate. The router handles this for you: effects created during route rendering are disposed when the route changes. More on that in Chapter 5.

Auto-Tracking

You do not tell tina4-js which signals an effect depends on. It figures it out by running the function and seeing which `.value` getters are called:

```
const a = signal(1);
const b = signal(2);
const showB = signal(true);

effect(() => {
  if (showB.value) {
    console.log(a.value + b.value);
  } else {
```

```
    console.log(a.value);
  }
});
```

When `showB` is `true`, the effect tracks `showB`, `a`, and `b`. When `showB` becomes `false`, on the next run it only reads `showB` and `a` -- so changes to `b` no longer trigger the effect. The dependency set is dynamic.

Re-subscription on Re-run

Every time an effect re-runs, it unsubscribes from all previous signals and re-subscribes to whatever it reads this time. This is automatic. You do not need to manage subscriptions.

---

5. batch() -- Grouping Updates

Picture a form with a first name field and a last name field. A greeting effect reads both. Without batching, updating both fields fires the effect twice -- once with stale data, once with correct data. The user sees a flicker. The console logs an intermediate state that never should have existed.

```
const first = signal('Alice');
const last = signal('Smith');

effect(() => {
  console.log(`${first.value} ${last.value}`);
});
// logs: "Alice Smith"

first.value = 'Bob';   // logs: "Bob Smith"
last.value = 'Jones';  // logs: "Bob Jones"
```

That is two effect runs. With `batch()`, you get one:

```
import { batch } from 'tina4js';

batch(() => {
  first.value = 'Bob';
  last.value = 'Jones';
});
// logs: "Bob Jones" -- once
```

Inside a batch, signal writes are deferred. Subscribers are queued and only fire once when the batch completes.

When Do You Need batch()?

Since v1.0.9, **event handlers in templates are automatically batched.** So this already only triggers one update:

```
html`
  <button @click=${() => {
    first.value = 'Bob';
    last.value = 'Jones';
  }}>Update</button>
`
```

You need explicit `batch()` only when writing to multiple signals outside event handlers:

- `setTimeout` or `setInterval` callbacks
- `fetch().then()` handlers
- WebSocket message handlers
- Any async code

Inside a template event handler, batching is free. Everywhere else, wrap your writes.

---

6. peek() -- Read Without Tracking

An effect tracks every signal it reads. But sometimes you need a value without creating a dependency. You want to read a configuration signal once, not re-run the effect every time the config changes. That is what `peek()` does:

```
const count = signal(0);
const multiplier = signal(2);

effect(() => {
  // Tracks count, but NOT multiplier
  console.log(count.value * multiplier.peek());
});

count.value = 5;      // effect runs
multiplier.value = 3; // effect does NOT run
```

`peek()` returns the current value without registering the read. The effect does not re-run when `multiplier` changes.

This is useful for:

- Reading config values that rarely change
- Avoiding infinite loops when an effect writes to a signal it also reads
- Performance optimization -- skipping unnecessary re-runs

---

7. Debug Labels

Signals accept an optional second argument -- a debug label:

```
const count = signal(0, 'count');
const username = signal('', 'username');
const items = signal([], 'cart-items');
```

Labels do nothing in production. In the debug overlay (Chapter 9), labels appear in the Signals panel. Instead of staring at `Signal<number>` with value `7` and wondering which signal that is, you see `count: 7`. That difference matters at 2 AM when something is not updating.

Add labels to every signal you might need to debug. The overhead is zero when the debug module is not imported.

---

8. isSignal() -- Type Check

```
import { isSignal } from 'tina4js';

const count = signal(0);

isSignal(count);     // true
isSignal(42);        // false
isSignal(null);      // false
isSignal({ value: 1 }); // false -- must be a real tina4 signal
```

Useful when writing utilities that accept either a signal or a plain value.

---

9. Common Mistakes

Mistake 1: Reading .value in Templates

```
// WRONG -- evaluates once, never updates
html`<p>${count.value}</p>`

// RIGHT -- pass the signal directly
html`<p>${count}</p>`

// RIGHT -- use a function for expressions
html`<p>${() => count.value * 2}</p>`
```

When you write `${count.value}`, JavaScript evaluates `count.value` before the template function sees it. The template gets the number `0`, not the signal. It cannot subscribe to changes.

When you write `${count}`, the template receives the signal object. It creates a text node and subscribes to changes. The DOM updates automatically.

Mistake 2: Mutating in Place

```
// WRONG
items.value.push(newItem);

// RIGHT
items.value = [...items.value, newItem];
```

Covered above, but it is worth repeating. This is the number one bug in tina4-js applications.

Mistake 3: Creating Effects in Loops

```
// WRONG -- creates a new effect every time the list re-renders
html`${() => items.value.map(item => {
  effect(() => console.log(item));  // leaks!
  return html`<div>${item}</div>`;
})}`
```

Effects created inside reactive blocks are fine -- the router and html renderer handle cleanup. But manually creating effects in a loop that runs inside a reactive function will leak unless you dispose them. Let the template engine handle reactivity. Use `${signal}` and `${() => expr}` instead of manual effects for DOM updates.

Mistake 4: Forgetting That false Renders as Text

```
// WRONG -- if condition is false, renders the TEXT "false"
html`${showDetails && html`<div>Details here</div>`}`

// RIGHT -- use a ternary
html`${() => showDetails.value ? html`<div>Details here</div>` : null}`
```

In JavaScript, `false && anything` evaluates to `false`. The template receives the boolean `false` and renders it as the string `"false"`. Use the ternary pattern. Always.

---

10. Putting It Together -- A Todo List

Signals. Computed. Effects. Batch. Here they are, working together in a complete todo application:

```
import { signal, computed, effect, batch, html } from 'tina4js';

function todoApp() {
  const items = signal<{ text: string; done: boolean }[]>([], 'todo-items');
  const input = signal('', 'todo-input');
  const filter = signal<'all' | 'active' | 'done'>('all', 'todo-filter');

  const filtered = computed(() => {
    const list = items.value;
    if (filter.value === 'active') return list.filter(i => !i.done);
    if (filter.value === 'done') return list.filter(i => i.done);
    return list;
  });

  const remaining = computed(() =>
    items.value.filter(i => !i.done).length
  );

  // Side effect: update document title
  effect(() => {
    document.title = `Todos (${remaining.value} left)`;
  });

  const addItem = () => {
    const text = input.value.trim();
    if (!text) return;
    batch(() => {
      items.value = [...items.value, { text, done: false }];
      input.value = '';
    });
  };

  const toggleItem = (index: number) => {
    items.value = items.value.map((item, i) =>
      i === index ? { ...item, done: !item.done } : item
    );
  };

  const removeItem = (index: number) => {
    items.value = items.value.filter((_, i) => i !== index);
```

```
  };

  return html`
    <div>
      <h1>Todos</h1>

      <form @submit=${(e: Event) => { e.preventDefault(); addItem(); }}>
        <input
          type="text"
          placeholder="What needs to be done?"
          .value=${input}
          @input=${(e: Event) => { input.value = (e.target as HTMLInputElement).value; }}
        />
        <button type="submit">Add</button>
      </form>

      <div>
        <button @click=${() => filter.value = 'all'}>All</button>
        <button @click=${() => filter.value = 'active'}>Active</button>
        <button @click=${() => filter.value = 'done'}>Done</button>
      </div>

      <ul>
        ${() => filtered.value.map((item, index) => html`
          <li>
            <input
              type="checkbox"
              @click=${() => toggleItem(index)}
              ?checked=${item.done}
            />
            <span style="${item.done ? 'text-decoration: line-through' : ''}">${item.text}</span
            <button @click=${() => removeItem(index)}>x</button>
          </li>
        `)}
      </ul>

      <p>${remaining} items left</p>
    </div>
  `;
}
```

Every concept from this chapter is present:

- `items`, `input`, and `filter` are signals with debug labels -- visible in the overlay
- `filtered` and `remaining` are computed -- they recalculate when `items` or `filter` change
- `effect()` syncs the document title as a side effect
- `batch()` groups the add-and-clear into one update
- The list uses `${() => filtered.value.map(...)}` -- a reactive block that re-renders when `filtered` changes
- New references everywhere: `map()` returns a new array, `filter()` returns a new array, the spread creates new objects

No mutation. No manual DOM updates. No event bus. Signals handle all of it.

Summary

Concept	API	Purpose
Signal	`signal(value, label?)`	Reactive value
Read	`sig.value`	Get current value (tracks dependency)
Write	`sig.value = x`	Set value (notifies subscribers)
Computed	`computed(() => expr)`	Derived read-only signal
Effect	`effect(() => { ... })`	Side effect, returns dispose function
Batch	`batch(() => { ... })`	Group writes, one notification pass
Peek	`sig.peek()`	Read without tracking
Check	`isSignal(x)`	Returns true for tina4 signals
Label	`signal(0, 'name')`	Debug overlay identification

HTML Templates

DOM Without the Framework Tax

Open your browser's DevTools. Inspect a React application. You see a `<div id="root">` containing a tree of elements that a virtual DOM diffing algorithm built for you. Somewhere between your JSX and those elements, a reconciler compared two trees, computed a minimal set of patches, and applied them. For a counter displaying the number 5.

tina4-js skips all of that. The `html` tagged template creates real DOM nodes. It binds signals to specific text nodes and attributes. When a signal changes, that one text node updates. No tree comparison. No patch computation. No framework standing between your data and your DOM.

This chapter covers every binding syntax in the `html` tagged template -- reactive text, reactive blocks, events, boolean attributes, property bindings, and class bindings. You will know which ones update and which ones are static, and you will never fall into the `false`/`null` rendering trap.

---

1. What html`` Does

```
import { html } from 'tina4js';

const fragment = html`<h1>Hello, World!</h1>`;
document.body.appendChild(fragment);
```

`html` is a tagged template literal function. It parses your markup once, caches the result, then clones it for each call. The return value is a real `DocumentFragment` -- actual DOM nodes. Not a string. Not a virtual tree. Not an intermediate representation.

No `render()` loop. No reconciliation. No diffing. The template creates DOM, binds signals to specific nodes, and walks away. Updates happen through direct signal subscriptions on individual nodes. The framework is not in the middle of the conversation between your data and your page.

---

2. Static Values -- ${value}

Plain values go in as text nodes. Once. Never again.

```
const name = 'Alice';
html`<p>Hello, ${name}</p>`
```

The template evaluates `name`, gets the string `"Alice"`, creates a text node, and inserts it. If `name` changes later, the DOM does not update. It is static.

Everything is XSS-safe by default. Values are inserted as text nodes, not HTML:

```
const userInput = '<script>alert("xss")</script>';
html`<p>${userInput}</p>`
// Renders as the literal text: <script>alert("xss")</script>
// NOT executed as HTML
```

You cannot accidentally inject HTML through interpolation. This is by design.

---

3. Reactive Text -- ${signal}

Pass a signal directly (not `.value`) to create a reactive text node:

```
import { signal, html } from 'tina4js';

const count = signal(0);

html`<p>Count: ${count}</p>`
```

When `count.value` changes, the text node updates. Nothing else in the DOM moves. The `<p>` element stays. The `"Count: "` text stays. Only the number changes.

This is the most common pattern in tina4-js. One signal. One text node. Live updates forever.

Critical rule: Pass the signal, not its value.

```
// WRONG -- static, evaluates once
html`<p>${count.value}</p>`

// RIGHT -- reactive, updates on change
html`<p>${count}</p>`
```

When you write `${count.value}`, JavaScript evaluates `count.value` (gets `0`) and passes the number `0` to the template. The template sees a plain number, not a signal. It creates a static text node.

When you write `${count}`, JavaScript passes the signal object. The template detects it (via `isSignal()`), creates a text node, and subscribes to changes. When the signal updates, the text node updates.

---

4. Reactive Blocks -- ${() => expr}

Functions are reactive blocks. The function runs immediately, and re-runs whenever any signal read inside it changes:

```
const show = signal(true);
const name = signal('Alice');

html`
  <div>
    ${() => show.value
      ? html`<p>Hello, ${name}!</p>`
      : null
    }
  </div>
`
```

When `show` or `name` changes, the function re-runs. The previous nodes are removed and new nodes are inserted. This is how you do conditional rendering and lists.

Conditional Rendering

```
const isLoggedIn = signal(false);

html`
  ${() => isLoggedIn.value
    ? html`<p>Welcome back!</p>`
    : html`<a href="/login">Log in</a>`
  }
`
```

Lists

```
const items = signal(['Apple', 'Banana', 'Cherry']);

html`
  <ul>
    ${() => items.value.map(item => html`<li>${item}</li>`)}
  </ul>
`
```

When `items` changes, the entire `<ul>` content is replaced. Old nodes out. New nodes in. This is not keyed reconciliation like React -- it is a full swap. For lists under a few hundred items, the browser handles this in under a millisecond. For massive lists, consider a virtualization library.

Nested Reactivity

Reactive blocks can contain signals:

```
const items = signal([
  { name: signal('Apple'), price: signal(1.50) },
]);

html`
  <ul>
    ${() => items.value.map(item => html`
      <li>${item.name} - $${item.price}</li>
    `)}
  </ul>
`
```

Each `${item.name}` is a reactive text node. If you change `items.value[0].name.value = 'Pear'`, only that text node updates. The list does not re-render.

---

5. DocumentFragments and Arrays

You can nest templates and pass arrays:

```
const header = html`<h1>Title</h1>`;
const items = ['one', 'two', 'three'];

html`
  ${header}
  <ul>
    ${items.map(i => html`<li>${i}</li>`)}
  </ul>
`
```

`DocumentFragment` values are inserted directly. Arrays are flattened -- each item is converted to nodes.

---

6. Event Handlers -- @event

The `@` prefix binds event listeners:

```
html`
  <button @click=${() => console.log('clicked!')}>Click me</button>
  <input @input=${(e: Event) => console.log((e.target as HTMLInputElement).value)} />
  <form @submit=${(e: Event) => {
    e.preventDefault();
    handleSubmit();
  }}>
    ...
  </form>
`
```

Any DOM event works. `@click`. `@input`. `@change`. `@submit`. `@keydown`. `@mouseenter`. `@focus`. `@blur`. If the browser fires it, tina4-js can bind it.

Auto-Batching

Since v1.0.9, all event handlers are automatically wrapped in `batch()`. This means you can write to multiple signals in one handler and only get one DOM update:

```
html`
  <button @click=${() => {
    firstName.value = 'Bob';
    lastName.value = 'Jones';
    age.value = 30;
  }}>Update All</button>
`
// Three signal writes, one DOM update
```

You do not need explicit `batch()` inside event handlers. It happens automatically.

---

7. Boolean Attributes -- ?attr

A button should be disabled while a form submits. A div should be hidden until data loads. A checkbox should be checked when a task is done. HTML boolean attributes -- `disabled`, `hidden`, `checked`, `readonly`, `required` -- need the `?` prefix:

```
const isDisabled = signal(false);
const isHidden = signal(true);

html`
  <button ?disabled=${isDisabled}>Submit</button>
  <div ?hidden=${isHidden}>Secret content</div>
`
```

When the value is truthy, the attribute is added (e.g., `<button disabled>`). When falsy, the attribute is removed entirely.

Without the `?` prefix, you get a string attribute:

```
// WRONG -- sets disabled="false" (which is still disabled!)
html`<button disabled=${isDisabled}>Submit</button>`

// RIGHT -- adds/removes the disabled attribute
html`<button ?disabled=${isDisabled}>Submit</button>`
```

This distinction catches everyone at least once. In HTML, `<button disabled="false">` is still disabled. The attribute exists. The browser does not read its value. The `?` prefix solves this by adding or removing the attribute entirely -- present means true, absent means false.

Boolean attributes accept signals, functions, and computed values:

```
// Signal
html`<button ?disabled=${isDisabled}>Submit</button>`

// Function
html`<button ?disabled=${() => items.value.length === 0}>Submit</button>`

// Computed
const canSubmit = computed(() => name.value.length > 0);
html`<button ?disabled=${() => !canSubmit.value}>Submit</button>`
```

---

8. Property Bindings -- .prop

The `.` prefix sets DOM properties (not HTML attributes):

```
const inputValue = signal('hello');

html`
  <input .value=${inputValue} />
  <div .innerHTML=${html`<strong>Bold text</strong>`} />
`
```

`.value` is the most common property binding. It sets the input's `value` property directly, which is different from the `value` attribute (the attribute is the initial value; the property is the current value).

`.innerHTML` is how you inject raw HTML. This is the only way to render HTML strings or inline SVG in tina4-js:

```
const svgIcon = '<svg viewBox="0 0 24 24"><path d="M12 2L2 22h20L12 2z"/></svg>';

// WRONG -- renders as escaped text
html`<div>${svgIcon}</div>`
// Shows: <svg viewBox="0 0 24 24">...

// RIGHT -- renders as HTML
html`<div .innerHTML=${svgIcon}></div>`
// Shows the actual SVG triangle
```

Warning: `.innerHTML` bypasses XSS protection. Only use it with trusted content. Never pass user input to `.innerHTML`.

When a signal is passed to a property binding, it updates reactively:

```
const content = signal('<em>Loading...</em>');

html`<div .innerHTML=${content}></div>`

// Later:
content.value = '<em>Done!</em>';
// The div's innerHTML updates
```

CRITICAL: Never Put Inputs Inside Reactive Blocks

> ***Inputs inside `${() => ...}` lose focus on every keystroke.*** *The most common mistake in tina4-js is wrapping `<input>` elements inside reactive arrow functions. When a signal changes, the reactive block re-renders its entire DOM subtree — destroying the input and creating a new one. The cursor disappears, the input loses focus, and typing is interrupted.*

BAD — input inside a reactive block (broken):

```
const name = signal('');
const showExtra = signal(false);

// WRONG: the input is INSIDE ${() => ...} — it gets destroyed on every keystroke
html`
  <div>
    ${() => html`
      <input .value=${name} @input=${(e: Event) => {
        name.value = (e.target as HTMLInputElement).value;
      }} />
      ${showExtra.value ? html`<p>Extra info</p>` : null}
    `}
  </div>
`;
```

GOOD — input outside, only dynamic content in reactive blocks:

```
const name = signal('');
const showExtra = signal(false);

// RIGHT: the input stays in the static template — only the conditional goes in ${() => ...}
html`
  <div>
    <input .value=${name} @input=${(e: Event) => {
      name.value = (e.target as HTMLInputElement).value;
    }} />
    ${() => showExtra.value ? html`<p>Extra info</p>` : null}
  </div>
`;
```

The rule: Inputs, textareas, and selects belong in the **static part** of your template. Use `.value`, `@input`, and `?disabled` bindings to make them reactive. Only wrap **computed output** (conditional messages, dynamic lists, calculated text) in `${() => ...}` blocks.

---

9. Dynamic Attributes -- attr=${value}

Regular attributes accept signals and functions for reactive updates:

```
const color = signal('red');
const className = signal('active');

html`
  <div class=${className} style=${() => `color: ${color.value}`}>
    Styled text
  </div>
`
```

When `className` changes, the `class` attribute updates. When `color` changes, the `style` attribute updates.

Reactive Classes

```
const isActive = signal(true);

// Simple string signal
const cls = signal('btn btn-primary');
html`<button class=${cls}>Click</button>`

// Function-based
html`<div class=${() => isActive.value ? 'tab active' : 'tab'}>Tab</div>`
```

---

10. The false/null/undefined Trap

You will hit this. Everyone does. Here are the rules:

```
${false}      // Renders nothing (empty) — treated the same as null
${null}       // Renders nothing (empty)
${undefined}  // Renders nothing (empty)
${true}       // Renders the TEXT "true"
${0}          // Renders the TEXT "0"
```

`false`, `null`, and `undefined` all render as nothing — they are explicitly swallowed by the template engine. Every other value is converted to text via `String(value)`.

This means the `&&` shortcut actually works for hiding content — `false` disappears cleanly:

```
// This works -- false renders nothing
html`${() => show.value && html`<p>Content</p>`}`

// This also works -- explicit ternary is clearer and preferred
html`${() => show.value ? html`<p>Content</p>` : null}`
```

The ternary is still the recommended style because intent is explicit and it handles both branches. But unlike React, you will not see the stray text `"false"` appear in the DOM if you forget.

---

11. Template Caching

The `html` tag caches parsed templates by the identity of the template strings array. The first call parses the HTML and creates a `<template>` element. Subsequent calls with the same template literal clone the cached template and bind fresh values.

This means:

- Templates in loops are only parsed once
- Cloning a `<template>` is fast (browser-native)
- The overhead per render is binding, not parsing

You do not need to do anything to benefit from this. It is automatic.

---

12. Putting It All Together

Static text. Reactive text. Reactive blocks. Event handlers. Boolean attributes. Property bindings. Conditional rendering. Here they all are, working together in a login form:

```
import { signal, computed, html } from 'tina4js';

function loginForm() {
  const email = signal('');
  const password = signal('');
  const loading = signal(false);
  const error = signal<string | null>(null);

  const isValid = computed(() =>
    email.value.includes('@') && password.value.length >= 8
  );

  const handleSubmit = async (e: Event) => {
    e.preventDefault();
    loading.value = true;
    error.value = null;

    try {
      const response = await fetch('/api/login', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          email: email.value,
          password: password.value,
        }),
      });

      if (!response.ok) throw new Error('Invalid credentials');
      // handle success...
    } catch (err) {
      error.value = (err as Error).message;
    } finally {
      loading.value = false;
    }
  };
```

```
  return html`
    <form @submit=${handleSubmit}>
      <h2>Login</h2>

      ${() => error.value
        ? html`<p style="color: red">${error}</p>`
        : null
      }

      <label>
        Email
        <input
          type="email"
          .value=${email}
          @input=${(e: Event) => { email.value = (e.target as HTMLInputElement).value; }}
          ?disabled=${loading}
        />
      </label>

      <label>
        Password
        <input
          type="password"
          .value=${password}
          @input=${(e: Event) => { password.value = (e.target as HTMLInputElement).value; }}
          ?disabled=${loading}
        />
      </label>

      <button
        type="submit"
        ?disabled=${() => !isValid.value || loading.value}
      >
        ${() => loading.value ? 'Logging in...' : 'Login'}
      </button>
    </form>
  `;
}
```

Every binding type from this chapter appears in this form:

- `.value=${email}` -- property binding keeps the input's DOM property in sync with the signal
- `@input` -- event handler updates the signal when the user types
- `?disabled=${loading}` -- boolean attribute toggles from a signal
- `?disabled=${() => !isValid.value || loading.value}` -- boolean attribute from a function
- `${() => loading.value ? 'Logging in...' : 'Login'}` -- reactive text block swaps the button label
- `${() => error.value ? html\...\ : null}` -- conditional rendering with the ternary pattern

One template. Six binding types. Zero manual DOM updates. The template engine handles every transition between states, and the form responds the moment data changes.

Summary

	What it does	Reactive?		
	Static text node, XSS-safe	No		
	Reactive text node	Yes		
`r}`	Reactive block (conditionals, lists)	Yes		
	Insert DocumentFragment	No		
	Render each item	No		
	null\	undefined}`	Renders nothing (empty)	-
`}`	Event listener (auto-batched)	-		
`{x}`	Boolean attribute (add/remove)	If signal/function		
	DOM property binding	If signal		
`${x}`	Raw HTML injection	If signal		
	Regular attribute	If signal/function		

Components

Web Components That Don't Suck

You build a button component in React. It works in React. You build one in Vue. It works in Vue. You build one in Angular. It works in Angular. Three frameworks. Three components. Same button.

Web Components solve this. One component. Every framework. Every page. Every context. The browser itself is the runtime.

The problem is that raw Web Components are verbose. Boilerplate for observed attributes. Manual attribute-to-property reflection. No reactive rendering. `Tina4Element` strips all of that away. You get reactive props, scoped styles, and template rendering -- with the full portability of native Web Components underneath.

---

1. What Are Tina4 Components

A Tina4 component is a Web Component. A real one. It extends `HTMLElement`, registers with `customElements.define()`, and works anywhere the browser runs. Drop it in a React app. Drop it in a static HTML page. Drop it in a WordPress theme. No framework runtime required to consume it.

`Tina4Element` adds three things on top of native Web Components:

- **Reactive props** -- attributes become signals
- **Scoped styles** -- CSS that cannot leak in or out (via Shadow DOM)
- **Template rendering** -- your `render()` method returns `html` tagged templates

Everything else is standard Web Components. No magic. No hidden state. No framework lock-in.

---

2. Your First Component

```
import { Tina4Element, html } from 'tina4js';

class GreetingCard extends Tina4Element {
  static props = { name: String };

  render() {
    return html`
      <div class="card">
        <h2>Hello, ${this.prop('name')}!</h2>
        <p>Welcome to tina4-js.</p>
      </div>
    `;
  }
}

customElements.define('greeting-card', GreetingCard);
```

Use it in HTML:

```
<greeting-card name="Alice"></greeting-card>
```

Or in a template:

```
html`<greeting-card name="Alice"></greeting-card>`
```

Change the attribute and the component updates:

```
document.querySelector('greeting-card')!.setAttribute('name', 'Bob');
// The heading updates to "Hello, Bob!"
```

---

3. static props -- Declaring Reactive Props

The `static props` object declares which attributes the component observes. Each key is an attribute name, and the value is a type constructor:

```
class UserCard extends Tina4Element {
  static props = {
    name: String,
    age: Number,
    active: Boolean,
  };

  render() {
    return html`
      <div>
        <h3>${this.prop('name')}</h3>
        <p>Age: ${this.prop('age')}</p>
        <p>${() => this.prop<boolean>('active').value ? 'Active' : 'Inactive'}</p>
      </div>
    `;
  }
}
```

Type Coercion

HTML attributes are always strings. tina4-js coerces them based on the type you declare:

Type	Coercion Rule	Example
`String`	Attribute value as-is, or `''` if absent	`name="Alice"` -> `'Alice'`
`Number`	`Number(value)`, or `0` if absent	`age="30"` -> `30`
`Boolean`	`true` if attribute exists, `false` if absent	`active` -> `true`, no attribute -> `false`

Boolean props follow the HTML convention: the attribute's presence means `true`, its absence means `false`. The attribute value does not matter.

```
<user-card name="Alice" age="30" active></user-card>
```

- `name` -> `'Alice'` (String)
- `age` -> `30` (Number)
- `active` -> `true` (Boolean, attribute is present)

```
<user-card name="Bob" age="25"></user-card>
```

- `active` -> `false` (Boolean, attribute is absent)

---

4. this.prop(name) -- Reading Props

`this.prop('name')` returns a **signal** for the named prop. This means you can:

- **Drop it in a template** for reactive rendering:

```
html`<span>${this.prop('name')}</span>`
```

- **Read its value** in methods:

```
const currentName = this.prop<string>('name').value;
```

- **Use it in computed/effects:**

```
const greeting = computed(() => `Hello, ${this.prop<string>('name').value}!`);
```

When the HTML attribute changes (via `setAttribute` or from a parent template), the prop signal updates and everything that depends on it re-renders.

---

5. static styles -- Scoped CSS

```
class StatusBadge extends Tina4Element {
  static props = { status: String };

  static styles = `
    :host {
      display: inline-block;
    }
    .badge {
      padding: 0.25rem 0.75rem;
      border-radius: 999px;
      font-size: 0.75rem;
      font-weight: 600;
      text-transform: uppercase;
    }
    .badge.active { background: #dcfce7; color: #166534; }
    .badge.inactive { background: #fee2e2; color: #991b1b; }
    .badge.pending { background: #fef3c7; color: #92400e; }
  `;

  render() {
    return html`
      <span class=${() => `badge ${this.prop<string>('status').value}`}>
        ${this.prop('status')}
      </span>
    `;
  }
}

customElements.define('status-badge', StatusBadge);
```

Styles are injected into the Shadow DOM `<style>` tag. They are completely scoped:

- `.badge` inside this component does not affect any `.badge` outside it
- External CSS does not affect elements inside this component
- The `:host` selector targets the component element itself

The :host Selector

`:host` is a Shadow DOM feature. It selects the component's outer element (the custom element tag). Use it to set `display`, `margin`, `padding`, and other box-model properties:

```
:host {
  display: block;
  margin-bottom: 1rem;
}
```

Without `:host { display: block }`, custom elements default to `display: inline`, which often causes unexpected layout behavior.

---

6. static shadow -- Light DOM vs Shadow DOM

By default, components use Shadow DOM (`static shadow = true`). Shadow DOM provides style encapsulation but has tradeoffs:

- **Shadow DOM (default):** Styles are scoped. External CSS cannot reach inside. Good for reusable components.
- **Light DOM:** No style encapsulation. Component renders directly into the page DOM. External CSS applies normally.

To use light DOM:

```
class PageSection extends Tina4Element {
  static shadow = false;
  static props = { title: String };

  render() {
    return html`
      <section>
        <h2>${this.prop('title')}</h2>
        <p>This content is rendered directly into the page DOM.</p>
      </section>
    `;
  }
}
```

When `shadow` is `false`:

- `static styles` is ignored (no Shadow DOM to scope them)
- The component renders its content directly as children of the custom element
- Global CSS applies to the component's internal elements
- `<slot>` elements do not work -- slots are a Shadow DOM feature. If you need content projection, use Shadow DOM (`static shadow = true`)
- This is useful for layout components that need to inherit page styles

---

7. this.emit() -- Custom Events

Props flow down. Events flow up. A parent tells a child what to display through attributes. The child tells the parent what happened through custom events. This is the same pattern React uses with callbacks and Vue uses with `$emit` -- but here it is native DOM events, and they work across framework boundaries.

```
class TodoItem extends Tina4Element {
  static props = { text: String, done: Boolean };

  render() {
    return html`
      <li>
        <input
          type="checkbox"
          ?checked=${this.prop('done')}
```

```
          @change=${() => this.emit('toggle')}
        />
        <span>${this.prop('text')}</span>
        <button @click=${() => this.emit('remove')}>x</button>
      </li>
    `;
  }
}

customElements.define('todo-item', TodoItem);
```

Listen for the event in the parent:

```
html`
  <todo-item
    text="Buy milk"
    done
    @toggle=${() => toggleItem(0)}
    @remove=${() => removeItem(0)}
  ></todo-item>
`
```

Passing Data with Events

```
this.emit('select', { detail: { id: 42, name: 'Alice' } });
```

Listen for it:

```
html`
  <user-list @select=${(e: CustomEvent) => {
    console.log(e.detail.id);   // 42
    console.log(e.detail.name); // 'Alice'
  }}></user-list>
`
```

Events are dispatched with `bubbles: true` and `composed: true` by default, so they cross Shadow DOM boundaries and bubble up the tree. You can listen for them anywhere above the component.

---

8. Lifecycle Hooks

Tina4Element provides two lifecycle hooks:

onMount()

Called after the component's first render. Use it for setup that needs DOM access:

```
class ChartWidget extends Tina4Element {
  static props = { data: String };

  onMount() {
    // DOM is ready, shadow root has content
    console.log('Chart mounted');
    // Initialize third-party library, set up intervals, etc.
  }

  render() {
```

```
    return html`<canvas id="chart"></canvas>`;
  }
}
```

onUnmount()

Called when the component is removed from the DOM. Use it for cleanup:

```
class LiveClock extends Tina4Element {
  private intervalId = 0;
  private time = signal(new Date().toLocaleTimeString());

  onMount() {
    this.intervalId = window.setInterval(() => {
      this.time.value = new Date().toLocaleTimeString();
    }, 1000);
  }

  onUnmount() {
    clearInterval(this.intervalId);
  }

  render() {
    return html`<span>${this.time}</span>`;
  }
}
```

Timers. Event listeners. WebSocket connections. Anything that outlives the DOM needs cleanup here. If you create it in `onMount()`, destroy it in `onUnmount()`. No exceptions.

---

9. Composing Components

Components compose naturally. Build small, focused components and combine them:

```
// status-badge.ts
class StatusBadge extends Tina4Element {
  static props = { status: String };
  static styles = `
    :host { display: inline-block; }
    span { padding: 0.2rem 0.5rem; border-radius: 4px; font-size: 0.75rem; }
    .online { background: #dcfce7; color: #166534; }
    .offline { background: #fee2e2; color: #991b1b; }
  `;

  render() {
    return html`<span class=${this.prop('status')}>${this.prop('status')}</span>`;
  }
}
customElements.define('status-badge', StatusBadge);

// user-row.ts
class UserRow extends Tina4Element {
  static props = { name: String, email: String, status: String };
  static styles = `
    :host { display: flex; align-items: center; gap: 1rem; padding: 0.5rem 0; }
    .name { font-weight: 600; }
    .email { color: #6b7280; }
```

```
  `;

  render() {
    return html`
      <span class="name">${this.prop('name')}</span>
      <span class="email">${this.prop('email')}</span>
      <status-badge status=${this.prop('status')}></status-badge>
    `;
  }
}
customElements.define('user-row', UserRow);
```

Use in a page:

```
html`
  <div>
    <user-row name="Alice" email="alice@test.com" status="online"></user-row>
    <user-row name="Bob" email="bob@test.com" status="offline"></user-row>
  </div>
`
```

---

10. The Store Pattern -- Shared Signals

A nav bar needs to know who is logged in. A dashboard needs the same data. A settings page can change it. Three components, one piece of state.

In React, you reach for Context or Redux. In Vue, you reach for Pinia. In tina4-js, you export signals from a module. That is it. No store library. No provider components. No boilerplate.

```
// store.ts
import { signal, computed } from 'tina4js';

export const user = signal<{ name: string; role: string } | null>(null, 'current-user');
export const isLoggedIn = computed(() => user.value !== null);
export const isAdmin = computed(() => user.value?.role === 'admin');

export function login(name: string, role: string) {
  user.value = { name, role };
}

export function logout() {
  user.value = null;
}
```

Any component can import and use these signals:

```
// nav-bar.ts
import { Tina4Element, html } from 'tina4js';
import { user, isLoggedIn, logout } from '../store';

class NavBar extends Tina4Element {
  static styles = `
    :host { display: flex; justify-content: space-between; padding: 1rem; }
    button { cursor: pointer; }
  `;

  render() {
```

```
    return html`
      <span>My App</span>
      <div>
        ${() => isLoggedIn.value
          ? html`
              <span>Welcome, ${user}!</span>
              <button @click=${() => logout()}>Logout</button>
            `
          : html`<a href="/login">Login</a>`
        }
      </div>
    `;
  }
}
customElements.define('nav-bar', NavBar);
```

No boilerplate. No providers. No context wrappers. No `mapStateToProps`. Signals are values. Import them. Use them. Every component that imports the same signal shares the same state, and every one of them updates when that state changes.

---

11. Complete Example -- A Card Component

```
import { Tina4Element, html, signal } from 'tina4js';

class ProductCard extends Tina4Element {
  static props = {
    name: String,
    price: Number,
    image: String,
    instock: Boolean,
  };

  static styles = `
    :host { display: block; border: 1px solid #e5e7eb; border-radius: 8px; overflow: hidden; }
    img { width: 100%; height: 200px; object-fit: cover; }
    .content { padding: 1rem; }
    h3 { margin: 0 0 0.5rem; }
    .price { font-size: 1.25rem; font-weight: 700; color: #059669; }
    .out-of-stock { color: #dc2626; font-size: 0.875rem; }
    button {
      width: 100%; padding: 0.75rem; border: none; border-radius: 4px;
      background: #2563eb; color: white; font-size: 1rem; cursor: pointer;
      margin-top: 0.5rem;
    }
    button:disabled { background: #9ca3af; cursor: not-allowed; }
  `;

  render() {
    const quantity = signal(1);

    return html`
      <img src=${this.prop('image')} alt=${this.prop('name')} />
      <div class="content">
        <h3>${this.prop('name')}</h3>
        <p class="price">$${this.prop('price')}</p>

        ${() => this.prop<boolean>('instock').value
          ? html`
              <div>
                <button @click=${() => { if (quantity.value > 1) quantity.value--; }}>-</button>
                <span>${quantity}</span>
                <button @click=${() => quantity.value++}>+</button>
              </div>
              <button @click=${() => {
                this.emit('add-to-cart', {
                  detail: {
                    name: this.prop<string>('name').value,
                    quantity: quantity.value,
                  },
                });
              }}>Add to Cart</button>
            `
          : html`<p class="out-of-stock">Out of Stock</p>`
        }
      </div>
    `;
  }
}
```

```
customElements.define('product-card', ProductCard);
```

Use it:

```
html`
  <product-card
    name="Wireless Mouse"
    price="29"
    image="/images/mouse.jpg"
    instock
    @add-to-cart=${(e: CustomEvent) => {
      console.log(`Added ${e.detail.quantity}x ${e.detail.name}`);
    }}
  ></product-card>
`
```

Summary

Feature	How
Define a component	`class X extends Tina4Element`
Declare props	`static props = { name: String }`
Read a prop	`this.prop('name')` returns a signal
Scoped styles	`static styles = '...'`
Light DOM mode	`static shadow = false`
Fire events	`this.emit('name', { detail })`
After first render	`onMount()`
Before removal	`onUnmount()`
Register tag	`customElements.define('tag-name', Class)`
Shared state	Export signals from a module

Routing

Navigation Without the Router Library

A user clicks "Dashboard." The page goes white. A full HTML document loads from the server. The CSS re-parses. The JavaScript re-executes. The user waits.

Single-page applications fix this. The browser stays on one HTML page. JavaScript swaps content in and out. The URL changes, but the page never reloads. Navigation feels instant because it is -- only the content that changed gets replaced.

tina4-js gives you this in about 20 lines of routing code. Parameterized routes. Guards that protect pages. A 404 catch-all. Programmatic navigation. No separate router library to install.

---

1. How Routing Works

The tina4-js router follows three steps:

- You call `route(pattern, handler)` to register paths
- You call `router.start({ target, mode })` to start listening
- When the URL changes, the router finds the matching route, calls the handler, and renders the result into the target element

No file-based routing. No dynamic imports by convention. No data loaders. You register routes with explicit calls, and the router invokes your handler function when the URL matches.

---

2. Registering Routes

```
import { route, html } from 'tina4js';

route('/', () => html`<h1>Home</h1>`);
route('/about', () => html`<h1>About</h1>`);
route('/contact', () => html`<h1>Contact</h1>`);
```

Pattern is always the first argument. This is a convention across all tina4 frameworks. Do not swap the arguments.

The handler is a function that returns content. It can return:

- A `DocumentFragment` (from `html` tagged templates)
- A `Node` (any DOM node)
- A `string` (set as innerHTML)
- A `Promise` that resolves to any of the above (async routes)

---

3. Route Parameters

Use `{param}` syntax in patterns to capture URL segments:

```
route('/users/{id}', ({ id }) => {
  return html`<h1>User ${id}</h1>`;
});

route('/posts/{year}/{slug}', ({ year, slug }) => {
  return html`<h1>${slug} (${year})</h1>`;
});
```

Parameters are extracted from the URL and passed to the handler as a `Record<string, string>`. All values are strings -- cast them yourself if you need numbers:

```
route('/products/{id}', ({ id }) => {
  const productId = parseInt(id, 10);
  return html`<product-detail id="${productId}"></product-detail>`;
});
```

How Matching Works

The router converts `{param}` to `([^/]+)` regex groups:

- `/users/{id}` matches `/users/42`, `/users/alice`, `/users/abc-123`
- `/posts/{year}/{slug}` matches `/posts/2024/my-post`
- It does not match `/users/` (trailing slash, no id) or `/users/42/edit` (extra segment)

Routes are checked in registration order. The first match wins. Put specific routes before general ones.

---

4. The Wildcard -- 404 Routes

```
route('*', () => html`
  <div>
    <h1>404</h1>
    <p>Page not found.</p>
    <a href="/">Go home</a>
  </div>
`);
```

The `*` pattern matches any path. Register it last so it only catches URLs that no other route matched.

---

5. Route Guards

A user types `/admin` into the address bar. They are not logged in. They should never see that page. Without guards, the handler runs, the admin panel renders, and your application has a security hole.

Guards protect routes. A guard runs before the handler. It can:

- Return `true` to allow navigation
- Return `false` to block navigation (nothing happens)
- Return a string to redirect to that path

```
import { route, html, signal, computed } from 'tina4js';

const token = signal<string | null>(null);
const isLoggedIn = computed(() => token.value !== null);

// Protected route
route('/dashboard', {
  guard: () => isLoggedIn.value || '/login',
  handler: () => html`<h1>Dashboard</h1>`,
});

// Login page
route('/login', () => {
  return html`
    <div>
      <h1>Login</h1>
      <button @click=${() => { token.value = 'abc123'; }}>
        Log In
      </button>
    </div>
  `;
});
```

When a user navigates to `/dashboard`:

- The guard runs: `isLoggedIn.value || '/login'`
- If `isLoggedIn` is `true`, the guard returns `true` and the handler renders
- If `isLoggedIn` is `false`, `false || '/login'` returns `'/login'` -- the router redirects

The redirect uses `navigate(path, { replace: true })`, so the protected URL does not appear in browser history. The user cannot press Back to get to the guarded page.

Admin Guard Pattern

```
const user = signal<{ role: string } | null>(null);

route('/admin', {
  guard: () => {
    if (!user.value) return '/login';
    if (user.value.role !== 'admin') return '/unauthorized';
    return true;
  },
  handler: () => html`<admin-panel></admin-panel>`,
});
```

---

6. Starting the Router

```
import { router } from 'tina4js';

router.start({
  target: '#root',
  mode: 'history',
});
```

target

A CSS selector for the element where route content renders. The router finds this element with `document.querySelector()`. If the element does not exist, the router throws.

```
<body>
  <nav>...</nav>
  <main id="root"></main>  <!-- routes render here -->
  <footer>...</footer>
</body>
```

mode

Two options:

Mode	URLs	Requires
`'history'`	`/about`, `/users/42`	Server-side URL rewriting
`'hash'`	`/#/about`, `/#/users/42`	Nothing -- works everywhere

Hash mode is the default in scaffolded projects. URLs carry a `#` prefix. No server configuration needed. Works on static hosts, GitHub Pages, S3, anywhere you can drop files.

History mode gives clean URLs without the hash. But the server must return `index.html` for all routes. When a user bookmarks `https://myapp.com/users/42` and loads it, the server needs to serve the SPA -- not search for a `/users/42` file that does not exist.

Vite dev server handles this automatically. For production, configure your web server:

```
# Nginx
location / {
  try_files $uri /index.html;
}
```

```
# Apache (.htaccess)
RewriteEngine On
RewriteRule ^index\.html$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.html [L]
```

---

7. Programmatic Navigation

```
import { navigate } from 'tina4js';

// Push to history (user can press Back)
navigate('/dashboard');

// Replace current history entry (no Back)
navigate('/login', { replace: true });
```

Use `navigate()` in event handlers, after API calls, in guards -- anywhere you need to move the user to a different page.

```
const handleLogin = async () => {
  const success = await api.post('/auth/login', credentials);
  if (success) {
    navigate('/dashboard');
  }
};
```

Link Interception

The router automatically intercepts clicks on `<a>` tags with same-origin `href` attributes. You do not need a special `<Link>` component:

```
html`<a href="/about">About</a>`
// Clicking this navigates without a page reload
```

The router ignores:

- Links with `target="_blank"` or any `target` attribute
- Links with the `download` attribute
- Links with `rel="external"`
- Modified clicks (Ctrl+click, Cmd+click, Shift+click, Alt+click)
- Links to different origins (external URLs)

---

8. Route Change Events

Listen for navigation events:

```
import { router } from 'tina4js';

const unsubscribe = router.on('change', ({ path, params, pattern, durationMs }) => {
  console.log(`Navigated to ${path} (matched ${pattern}) in ${durationMs}ms`);
  console.log('Params:', params);
});

// Later, to stop listening:
unsubscribe();
```

The change event includes:

Property	Type	Description

`path`	`string`	The current URL path
`params`	`Record<string, string>`	Extracted route parameters
`pattern`	`string`	The matched route pattern
`durationMs`	`number`	Time to render the route (ms)

Use this for analytics, breadcrumbs, active nav highlighting, or debugging:

```
// Analytics
router.on('change', ({ path }) => {
  analytics.pageView(path);
});

// Active nav highlighting
const currentPath = signal('/');
router.on('change', ({ path }) => {
  currentPath.value = path;
});

html`
  <nav>
    <a href="/" class=${() => currentPath.value === '/' ? 'active' : ''}>Home</a>
    <a href="/about" class=${() => currentPath.value === '/about' ? 'active' : ''}>About</a>
  </nav>
`
```

---

9. Async Routes

A user profile page needs data from the server. You cannot render the page until the data arrives. Route handlers can be async -- the router awaits the result before rendering:

```
route('/users/{id}', async ({ id }) => {
  const user = await api.get(`/users/${id}`);
  return html`
    <div>
      <h1>${user.name}</h1>
      <p>${user.email}</p>
    </div>
  `;
});
```

If the user navigates away before the async handler resolves, the stale result is discarded. The router tracks a version counter and renders only if the version matches. No race conditions. No stale data flashing on screen.

Loading States

For a better user experience, show a loading indicator:

```
route('/users/{id}', async ({ id }) => {
  const loading = signal(true);
  const user = signal<any>(null);

  // Show loading immediately
```

```
  const view = html`
    <div>
      ${() => loading.value
        ? html`<p>Loading...</p>`
        : html`
            <h1>${() => user.value?.name}</h1>
            <p>${() => user.value?.email}</p>
          `
      }
    </div>
  `;

  // Fetch in background
  api.get(`/users/${id}`).then(data => {
    user.value = data;
    loading.value = false;
  });

  return view;
});
```

---

10. Effect Cleanup on Route Change

You create a signal and an effect on one route. The user navigates away. Without cleanup, that effect keeps running -- updating state for a page that no longer exists, consuming memory for nodes that have been removed from the DOM.

The router handles this. When a route changes, effects created during the previous route's rendering are disposed. No memory leaks. No ghost updates.

```
route('/live', () => {
  const count = signal(0);

  // This effect is automatically cleaned up when the user navigates away
  effect(() => {
    const interval = setInterval(() => {
      count.value++;
    }, 1000);

    // But setInterval is NOT cleaned up automatically -- you need to handle that
  });

  return html`<p>Count: ${count}</p>`;
});
```

The router disposes the effect (unsubscribes from signals), but it cannot clean up timers, event listeners, or other resources you created outside the signal system. For those, use `onUnmount()` in a component, or track cleanup manually. The rule: if the router did not create it, the router cannot destroy it.

---

11. Complete Example -- Multi-Page App

```
// src/store.ts
import { signal, computed } from 'tina4js';

export const token = signal<string | null>(null, 'auth-token');
export const isLoggedIn = computed(() => token.value !== null);

// src/routes/index.ts
import { route, navigate, html, signal } from 'tina4js';
import { token, isLoggedIn } from '../store';

// Public routes
route('/', () => html`
  <div>
    <h1>Home</h1>
    <nav>
      <a href="/about">About</a>
      ${() => isLoggedIn.value
        ? html`<a href="/dashboard">Dashboard</a>`
        : html`<a href="/login">Login</a>`
      }
    </nav>
  </div>
`);

route('/about', () => html`
  <div>
    <h1>About</h1>
    <a href="/">Back</a>
  </div>
`);

// Login
route('/login', () => {
  const email = signal('');
  const password = signal('');

  return html`
    <div>
      <h1>Login</h1>
      <form @submit=${(e: Event) => {
        e.preventDefault();
        // Fake login
        token.value = 'fake-jwt-token';
        navigate('/dashboard');
      }}>
        <input
          type="email"
          placeholder="Email"
          @input=${(e: Event) => { email.value = (e.target as HTMLInputElement).value; }}
        />
        <input
          type="password"
          placeholder="Password"
          @input=${(e: Event) => { password.value = (e.target as HTMLInputElement).value; }}
        />
        <button type="submit">Login</button>
      </form>
```

```
    </div>
  `;
});

// Protected routes
route('/dashboard', {
  guard: () => isLoggedIn.value || '/login',
  handler: () => html`
    <div>
      <h1>Dashboard</h1>
      <p>You are logged in.</p>
      <button @click=${() => {
        token.value = null;
        navigate('/');
      }}>Logout</button>
    </div>
  `,
});

// 404
route('*', () => html`
  <div>
    <h1>404</h1>
    <a href="/">Go home</a>
  </div>
`);

// src/main.ts
import { router } from 'tina4js';
import './routes/index';

router.start({ target: '#root', mode: 'hash' });
```

Summary

	How	
r route	`route(pattern, handler)`	
params	`route('/users/{id}', ({ id }) => ...)`	
	`route('/x', { guard: () => bool\`	string, handler })`
d/404	`route('*', handler)`	
uter	`router.start({ target: '#root', mode: 'history' })`	
te	`navigate('/path')`	
e (no back)	`navigate('/path', { replace: true })`	
or changes	`router.on('change', ({ path, params, pattern, durationMs }) => ...)`	
outes	Handler returns a `Promise`	
erception	Automatic for same-origin `<a href>`	

API

Talking to Your Backend

Your frontend renders a list of users. The data lives in a database. Between your signal and that database sits an HTTP request -- and a surprising amount of ceremony. Auth headers. CSRF tokens. Token rotation. Error handling. JSON parsing. In most projects, you install Axios or ky, configure interceptors, and write wrapper functions before you make your first call.

tina4-js includes all of this. One import. No extra packages. If you use a tina4-php or tina4-python backend, auth and CSRF protection wire up with a single configuration flag.

---

1. The API Client

The built-in HTTP client wraps `fetch()` with the features you need in every application:

- Automatic `Authorization: Bearer` headers
- Token rotation via `FreshToken` response headers
- CSRF `formToken` injection in POST/PUT/PATCH/DELETE bodies
- Per-request headers and query params
- Request and response interceptors
- JSON parsing by default

```
import { api } from 'tina4js';
```

One import. No Axios. No ky. No dependencies. The API client ships with the framework.

---

2. Configuration

Call `api.configure()` once at app startup:

```
api.configure({
  baseUrl: 'https://api.example.com',
  auth: true,
});
```

Options

Option	Type	Default	Description
`baseUrl`	`string`	`''`	Prepended to all request paths
`auth`	`boolean`	`false`	Enable Bearer token and formToken
`tokenKey`	`string`	`'tina4_token'`	localStorage key for the auth token
`headers`	`Record<string, string>`	`{}`	Default headers on every request

A typical setup for a tina4-php backend:

```
api.configure({
  baseUrl: '/api',
  auth: true,
  headers: {
    'Accept': 'application/json',
  },
});
```

During development with Vite, proxy API calls to your backend:

```
// vite.config.ts
export default defineConfig({
  server: {
    proxy: {
      '/api': 'http://localhost:7145',
    },
  },
});
```

---

3. Making Requests

GET

```
const users = await api.get('/users');
```

With query parameters:

```
const users = await api.get('/users', {
  params: { page: 2, limit: 20, search: 'alice' },
});
// Request: GET /users?page=2&limit=20&search=alice
```

Parameters are automatically URL-encoded.

POST

```
await api.post('/users', {
  name: 'Alice',
  email: 'alice@example.com',
  role: 'editor',
});
```

The body is serialized as JSON. The `Content-Type: application/json` header is added automatically.

PUT (Full Replace)

```
await api.put('/users/42', {
  name: 'Alice',
  email: 'alice@new-email.com',
  role: 'admin',
});
```

PATCH (Partial Update)

```
await api.patch('/users/42', {
  role: 'admin',
});
```

DELETE

```
await api.delete('/users/42');
```

---

4. RequestOptions

Every method accepts an optional `RequestOptions` object:

```
interface RequestOptions {
  headers?: Record<string, string>;
  params?: Record<string, string | number | boolean>;
}
```

Per-Request Headers

```
const data = await api.get('/reports/export', {
  headers: {
    'Accept': 'text/csv',
    'X-Custom': 'value',
  },
});
```

Per-request headers merge with (and override) the default headers from `configure()`.

Query Params on Any Method

```
await api.post('/search', { query: 'tina4' }, {
  params: { format: 'detailed', lang: 'en' },
});
// POST /search?format=detailed&lang=en
// Body: { "query": "tina4" }
```

---

5. Auth Flow

One flag -- `auth: true` -- activates three mechanisms that handle authentication for every request your application makes:

1. Bearer Token

Every request includes an `Authorization` header:

```
Authorization: Bearer <token>
```

The token is read from `localStorage` using the `tokenKey` (default: `'tina4_token'`).

2. formToken

For POST, PUT, PATCH, and DELETE requests, a `formToken` property is injected into the JSON body:

```
// You send:
await api.post('/users', { name: 'Alice' });

// Actual request body:
{ "name": "Alice", "formToken": "the-current-token" }
```

This is CSRF protection for tina4-php and tina4-python backends. They validate the `formToken` on every write operation.

3. FreshToken Rotation

When the server responds with a `FreshToken` header, the client automatically stores it:

```
HTTP/1.1 200 OK
FreshToken: new-jwt-token-here
```

The new token replaces the old one in `localStorage` and is used for subsequent requests. This allows the backend to rotate tokens on every request or on a schedule.

Login Example

```
import { api, navigate, signal } from 'tina4js';

const loginError = signal<string | null>(null);

async function login(email: string, password: string) {
  try {
    const result = await api.post<{ token: string }>('/auth/login', {
      email,
      password,
    });

    // Store the token (the API client reads it from localStorage)
    localStorage.setItem('tina4_token', result.token);

    navigate('/dashboard');
  } catch (err: any) {
    loginError.value = err.data?.message ?? 'Login failed';
  }
}
```

---

6. Error Handling

A 404. A 500. A network timeout. Every API call can fail, and your application needs to handle the failure without crashing. When the server returns a non-2xx status, `api` throws the response object:

```
try {
  await api.get('/users/999');
} catch (err: any) {
  console.log(err.status);  // 404
  console.log(err.data);    // { message: "User not found" }
  console.log(err.ok);      // false
}
```

The thrown object has the `ApiResponse` shape:

```
interface ApiResponse<T = unknown> {
  status: number;
  data: T;
  ok: boolean;
  headers: Headers;
}
```

Pattern: Centralized Error Handling with Interceptors

```
api.intercept('response', (response) => {
  if (response.status === 401) {
    localStorage.removeItem('tina4_token');
    navigate('/login');
  }
  if (response.status === 403) {
    navigate('/unauthorized');
  }
});
```

---

7. Interceptors

Every request passes through a pipeline. Interceptors let you insert logic at two points: before the request leaves and after the response arrives. Add a client version header to every request. Unwrap a response envelope. Log slow calls. Redirect on 401. Interceptors handle all of it in one place.

Request Interceptor

```
api.intercept('request', (config) => {
  // Add a custom header to every request
  config.headers['X-Client-Version'] = '1.0.0';

  // Add a timestamp
  config.headers['X-Request-Time'] = new Date().toISOString();
});
```

The `config` parameter is a `RequestInit` with a `headers` record. Modify it in place or return a new object.

Response Interceptor

```
api.intercept('response', (response) => {
  // Log slow requests
  if (response.status === 200) {
    console.log(`API: ${response.status}`);
  }

  // Transform data
  if (response.data && typeof response.data === 'object') {
    // unwrap a common envelope
    const envelope = response.data as any;
    if (envelope.data) {
      return { ...response, data: envelope.data };
    }
  }
});
```

Return a modified response to transform the data before it reaches your application code. Return nothing (or `undefined`) to pass the response through unchanged.

---

8. Real Example: CRUD Data Table

A list of users. Add one. Edit one. Delete one. This is the bread and butter of business applications, and it exercises every feature of the API client -- GET for loading, POST for creating, PUT for updating, DELETE for removing, batch for coordinating signal updates:

```
import { signal, computed, html, api, batch } from 'tina4js';

interface User {
  id: number;
  name: string;
  email: string;
}

function usersPage() {
  const users = signal<User[]>([], 'users');
  const loading = signal(true, 'users-loading');
  const editingId = signal<number | null>(null);
  const editName = signal('');
  const editEmail = signal('');

  // Load users
  async function loadUsers() {
    loading.value = true;
    try {
      const data = await api.get<User[]>('/users');
      users.value = data;
    } finally {
      loading.value = false;
    }
  }
```

```
// Create
async function createUser(name: string, email: string) {
  const newUser = await api.post<User>('/users', { name, email });
  users.value = [...users.value, newUser];
}

// Update
async function updateUser(id: number) {
  const updated = await api.put<User>(`/users/${id}`, {
    name: editName.value,
    email: editEmail.value,
  });
  batch(() => {
    users.value = users.value.map(u => u.id === id ? updated : u);
    editingId.value = null;
  });
}

// Delete
async function deleteUser(id: number) {
  await api.delete(`/users/${id}`);
  users.value = users.value.filter(u => u.id !== id);
}

// Start editing
function startEdit(user: User) {
  batch(() => {
    editingId.value = user.id;
    editName.value = user.name;
    editEmail.value = user.email;
  });
}

// Initial load
loadUsers();

return html`
  <div>
    <h1>Users</h1>

    ${() => loading.value
      ? html`<p>Loading...</p>`
      : html`
          <table>
            <thead>
              <tr><th>Name</th><th>Email</th><th>Actions</th></tr>
            </thead>
            <tbody>
              ${() => users.value.map(user => html`
                <tr>
                  ${() => editingId.value === user.id
                    ? html`
                        <td>
                          <input .value=${editName}
                            @input=${(e: Event) => { editName.value = (e.target as HTMLInputEl
                        </td>
                        <td>
                          <input .value=${editEmail}
                            @input=${(e: Event) => { editEmail.value = (e.target as HTMLInputE
```

```
                        </td>
                        <td>
                          <button @click=${() => updateUser(user.id)}>Save</button>
                          <button @click=${() => { editingId.value = null; }}>Cancel</button>
                        </td>
                      `
                    : html`
                        <td>${user.name}</td>
                        <td>${user.email}</td>
                        <td>
                          <button @click=${() => startEdit(user)}>Edit</button>
                          <button @click=${() => deleteUser(user.id)}>Delete</button>
                        </td>
                      `
                }
              </tr>
            `)}
          </tbody>
        </table>
      `

    }

    <h2>Add User</h2>
    <form @submit=${(e: Event) => {
      e.preventDefault();
      const form = e.target as HTMLFormElement;
      const formData = new FormData(form);
      createUser(formData.get('name') as string, formData.get('email') as string);
      form.reset();
    }}>
      <input name="name" placeholder="Name" required />
      <input name="email" type="email" placeholder="Email" required />
      <button type="submit">Add</button>
    </form>
  </div>
`;
}
```

---

9. File Upload

The API client sends JSON by default. Files are not JSON. For file uploads, use `fetch()` with the same base URL and auth token. The browser sets the correct `Content-Type` with multipart boundaries -- do not set it yourself:

```
async function uploadFile(file: File) {
  const formData = new FormData();
  formData.append('file', file);

  const token = localStorage.getItem('tina4_token');

  const response = await fetch('/api/upload', {
    method: 'POST',
    headers: {
      ...(token ? { Authorization: `Bearer ${token}` } : {}),
    },
    body: formData, // Do NOT set Content-Type -- browser sets it with boundary
```

```
  });

  return response.json();
}
```

Use in a template:

```
html`
  <input type="file" @change=${(e: Event) => {
    const file = (e.target as HTMLInputElement).files?.[0];
    if (file) uploadFile(file);
  }} />
`
```

Summary

What	How
Configure	`api.configure({ baseUrl, auth, tokenKey, headers })`
GET	`api.get(path, options?)`
POST	`api.post(path, body?, options?)`
PUT	`api.put(path, body?, options?)`
PATCH	`api.patch(path, body?, options?)`
DELETE	`api.delete(path, options?)`
Query params	`{ params: { key: value } }`
Per-request headers	`{ headers: { key: value } }`
Auth token	Automatic Bearer header when `auth: true`
Token rotation	Automatic via FreshToken response header
CSRF protection	Automatic formToken in POST/PUT/PATCH/DELETE
Error handling	Non-2xx throws `ApiResponse`
Request interceptor	`api.intercept('request', fn)`
Response interceptor	`api.intercept('response', fn)`

WebSocket

Real-Time Without the Headache

A chat message arrives. A stock price changes. A deployment finishes. A sensor reading spikes. The data exists on the server right now, but your user stares at stale numbers until they refresh the page.

HTTP requests are pull. You ask, the server answers. WebSocket is push. The server sends data the moment it exists. The user sees it the moment it arrives.

The raw WebSocket API gives you this, but it hands you a connection object and wishes you luck. Reconnection? Write it yourself. State tracking? Manual. Feeding messages into your reactive UI? Also manual.

tina4-js wraps all of it. One function call opens the connection. Five reactive signals track the state. A pipe function streams messages into your signals with a reducer. Auto-reconnect handles dropped connections with exponential backoff.

---

1. The WebSocket Client

The tina4-js WebSocket client adds three things on top of the browser API:

- **Reactive signals** -- `status`, `connected`, `lastMessage`, `error`, `reconnectCount`
- **Auto-reconnect** -- exponential backoff, configurable attempts
- **Signal piping** -- stream messages into signals with a reducer

```
import { ws } from 'tina4js';
```

---

2. Connecting

```
const socket = ws.connect('wss://api.example.com/ws');
```

One line. The connection starts. The `socket` object gives you reactive state and methods to send, listen, pipe, and close.

With Options

```
const socket = ws.connect('wss://api.example.com/ws', {
  reconnect: true,           // auto-reconnect on disconnect (default: true)
  reconnectDelay: 1000,      // initial delay in ms (default: 1000)
  reconnectMaxDelay: 30000,  // max delay with backoff (default: 30000)
  reconnectAttempts: 10,     // max attempts (default: Infinity)
  protocols: 'chat-v1',     // WebSocket sub-protocols
});
```

Options Reference

Option	Type	Default	Description	
`reconnect`	`boolean`	`true`	Enable auto-reconnect	
`reconnectDelay`	`number`	`1000`	Initial reconnect delay (ms)	
`reconnectMaxDelay`	`number`	`30000`	Max delay after backoff (ms)	
`reconnectAttempts`	`number`	`Infinity`	Max reconnect attempts	
`protocols`	`string \	string[]`	`[]`	WebSocket sub-protocols

---

3. Reactive Signals

The socket exposes five signals:

```
const socket = ws.connect('wss://api.example.com/ws');

socket.status;          // Signal<'connecting' | 'open' | 'closed' | 'reconnecting'>
socket.connected;       // Signal<boolean>
socket.lastMessage;     // Signal<unknown>
socket.error;           // Signal<Event | null>
socket.reconnectCount;  // Signal<number>
```

Use them directly in templates:

```
import { html } from 'tina4js';

html`
  <div>
    <p>Status: ${socket.status}</p>
    <p>Connected: ${() => socket.connected.value ? 'Yes' : 'No'}</p>
    <p>Reconnect attempts: ${socket.reconnectCount}</p>
    <p>Last message: ${() => JSON.stringify(socket.lastMessage.value)}</p>
  </div>
`
```

These update the moment the connection state changes. No polling. No callbacks. No manual state management. Drop them in a template and the UI stays current.

Status Flow

```
connecting -> open -> closed -> reconnecting -> open -> ...
```

- `connecting` -- initial connection attempt
- `open` -- connected and ready

- `closed` -- disconnected
- `reconnecting` -- waiting to reconnect (auto-reconnect active)

---

4. Sending Messages

```
// Send a string
socket.send('hello');

// Send an object (auto-JSON.stringify)
socket.send({ type: 'chat', message: 'Hello!' });

// Send with a specific structure
socket.send({
  action: 'subscribe',
  channel: 'notifications',
});
```

Objects are automatically serialized with `JSON.stringify()`. Strings are sent as-is.

If the socket is not connected, `send()` throws:

```
try {
  socket.send('hello');
} catch (e) {
  console.error('Not connected');
}
```

Check the `connected` signal before sending:

```
if (socket.connected.value) {
  socket.send(data);
}
```

---

5. Listening for Events

The `on()` method returns an unsubscribe function:

```
// Messages
const unsub = socket.on('message', (data) => {
  console.log('Received:', data);
});

// Connection opened
socket.on('open', () => {
  console.log('Connected!');
  socket.send({ type: 'auth', token: 'my-token' });
});

// Connection closed
socket.on('close', (code, reason) => {
  console.log(`Closed: ${code} ${reason}`);
});

// Errors
```

```
socket.on('error', (event) => {
  console.error('WebSocket error:', event);
});

// Later, unsubscribe
unsub();
```

Message Parsing

Messages are automatically JSON-parsed. If the message is valid JSON, you get an object. If not, you get the raw string:

```
socket.on('message', (data) => {
  // data is already parsed if it was JSON
  if (typeof data === 'object') {
    console.log(data.type, data.payload);
  } else {
    console.log('Raw string:', data);
  }
});
```

---

6. pipe() -- Stream Messages Into Signals

This is where WebSocket meets reactivity. A message arrives. It flows into a signal through a reducer function. The signal updates. Every subscriber -- text nodes, computed values, effects -- responds. The data moves from server to DOM without a single manual step.

`pipe()` connects a WebSocket message stream to a signal with a reducer:

```
import { signal } from 'tina4js';

const messages = signal<string[]>([]);

socket.pipe(messages, (msg, current) => {
  const chatMsg = msg as { text: string };
  return [...current, chatMsg.text];
});
```

Every time a message arrives:

- The reducer runs with `(message, currentSignalValue)`
- The return value is assigned to `messages.value`
- Everything subscribed to `messages` updates

No event listener that grabs a signal reference and mutates it. No manual state management. The pipe is declarative, testable, and composable. Data flows in one direction: server to socket to reducer to signal to DOM.

Multiple Pipes

You can pipe the same socket to multiple signals:

```
const notifications = signal<any[]>([]);
const userCount = signal(0);
```

```
// Route messages to different signals based on type
socket.pipe(notifications, (msg, current) => {
  const m = msg as { type: string; data: any };
  if (m.type === 'notification') return [...current, m.data];
  return current; // ignore non-notification messages
});

socket.pipe(userCount, (msg, current) => {
  const m = msg as { type: string; count: number };
  if (m.type === 'user_count') return m.count;
  return current;
});
```

Unsubscribing

`pipe()` returns an unsubscribe function:

```
const unsub = socket.pipe(messages, reducer);

// Later
unsub();
```

---

7. Auto-Reconnect

By default, the socket reconnects automatically when the connection drops. The delay doubles after each attempt (exponential backoff):

- Attempt 1: 1000ms delay
- Attempt 2: 2000ms delay
- Attempt 3: 4000ms delay
- Attempt 4: 8000ms delay
- ...up to `reconnectMaxDelay` (default: 30000ms)

When reconnection succeeds:

- `status` goes to `'open'`
- `connected` becomes `true`
- `reconnectCount` resets to `0`
- The delay resets to the initial value

Track reconnection in the UI:

```
html`
  ${() => socket.status.value === 'reconnecting'
    ? html`<div class="banner">
        Reconnecting... (attempt ${socket.reconnectCount})
      </div>`
    : null
  }
`
```

Disabling Reconnect

```
const socket = ws.connect('wss://api.example.com/ws', {
  reconnect: false,
});
```

Limiting Attempts

```
const socket = ws.connect('wss://api.example.com/ws', {
  reconnectAttempts: 5, // give up after 5 tries
});
```

---

8. Closing the Connection

```
socket.close();
```

An intentional close stops reconnecting. The status goes to `'closed'` and stays there. The user logs out. The component unmounts. The page no longer needs live data. Call `close()` and the connection ends.

With code and reason:

```
socket.close(1000, 'User logged out');
```

---

9. Real Example: Chat Room

Two users. One room. Messages appear the moment they are sent. The connection status shows at the top. The input disables when the socket disconnects. Everything is reactive -- signals drive every piece of state.

```
import { signal, html, ws, batch } from 'tina4js';

function chatRoom() {
  const messages = signal<{ user: string; text: string; time: string }[]>([]);
  const input = signal('');
  const username = signal('Anonymous');
  const socket = ws.connect('wss://api.example.com/chat');

  // Pipe incoming messages into the messages signal
  socket.pipe(messages, (msg, current) => {
    const m = msg as { type: string; user: string; text: string; time: string };
    if (m.type === 'chat') {
      return [...current, { user: m.user, text: m.text, time: m.time }];
    }
    return current;
  });

  // Send on enter
  const sendMessage = () => {
    const text = input.value.trim();
    if (!text || !socket.connected.value) return;

    socket.send({
      type: 'chat',
```

```
      user: username.value,
      text,
    });
    input.value = '';
  };

  return html`
    <div>
      <h1>Chat</h1>

      ${() => !socket.connected.value
        ? html`<p>Connecting...</p>`
        : null
      }

      <div style="height: 400px; overflow-y: auto; border: 1px solid #e5e7eb; padding: 1rem;">
        ${() => messages.value.map(msg => html`
          <div>
            <strong>${msg.user}</strong>
            <span style="color: #6b7280; font-size: 0.75rem">${msg.time}</span>
            <p>${msg.text}</p>
          </div>
        `)}
      </div>

      <form @submit=${(e: Event) => { e.preventDefault(); sendMessage(); }}>
        <input
          type="text"
          placeholder="Type a message..."
          .value=${input}
          @input=${(e: Event) => { input.value = (e.target as HTMLInputElement).value; }}
          ?disabled=${() => !socket.connected.value}
        />
        <button type="submit" ?disabled=${() => !socket.connected.value}>
          Send
        </button>
      </form>
    </div>
  `;
}
```

---

10. Real Example: Live Dashboard

A server streams metrics every second. Active users. Requests per second. Error rate. CPU usage. Alerts arrive when thresholds are exceeded. The dashboard updates without a single API poll -- the data pushes itself to the screen.

```
import { signal, computed, html, ws } from 'tina4js';

function liveDashboard() {
  const metrics = signal({
    activeUsers: 0,
    requestsPerSecond: 0,
    errorRate: 0,
    cpuUsage: 0,
  });
```

```
const alerts = signal<string[]>([]);

const socket = ws.connect('wss://api.example.com/metrics');

// Pipe metrics updates
socket.pipe(metrics, (msg, current) => {
  const m = msg as { type: string; data: any };
  if (m.type === 'metrics') {
    return { ...current, ...m.data };
  }
  return current;
});

// Pipe alerts
socket.pipe(alerts, (msg, current) => {
  const m = msg as { type: string; message: string };
  if (m.type === 'alert') {
    return [...current.slice(-9), m.message]; // keep last 10
  }
  return current;
});

const cpuColor = computed(() => {
  const cpu = metrics.value.cpuUsage;
  if (cpu > 80) return '#dc2626';
  if (cpu > 50) return '#f59e0b';
  return '#059669';
});

return html`
  <div>
    <h1>Live Dashboard</h1>
    <p>Status: ${socket.status}</p>

    <div style="display: grid; grid-template-columns: repeat(4, 1fr); gap: 1rem;">
      <div class="card">
        <h3>Active Users</h3>
        <p style="font-size: 2rem">${() => metrics.value.activeUsers}</p>
      </div>
      <div class="card">
        <h3>Requests/s</h3>
        <p style="font-size: 2rem">${() => metrics.value.requestsPerSecond}</p>
      </div>
      <div class="card">
        <h3>Error Rate</h3>
        <p style="font-size: 2rem">${() => metrics.value.errorRate}%</p>
      </div>
      <div class="card">
        <h3>CPU</h3>
        <p style="font-size: 2rem; color: ${cpuColor}">
          ${() => metrics.value.cpuUsage}%
        </p>
      </div>
    </div>

    <h2>Alerts</h2>
    <ul>
      ${() => alerts.value.map(alert => html`
        <li style="color: #dc2626">${alert}</li>
```

```
          `)}
        </ul>
      </div>
    `;
  }
```

Summary

`nnect(url, options?)`		
`t.status` -- `'connecting' \	'open' \	'closed' \
`t.connected` -- boolean		
`t.lastMessage`		
`t.error`		
`t.reconnectCount`		
`t.send(data)` -- auto-stringify		
.on('message' \	'open' \	'close' \
`t.pipe(signal, (msg, current) wValue)`		
`t.close(code?, reason?)` -- econnect		
default, exponential backoff		

SSE / NDJSON Streaming

Streaming Without WebSocket

An AI chatbot types back one token at a time. A dashboard counter ticks up as orders arrive. A notification feed shows events the moment they happen. The data flows from server to client, but you don't need the complexity of WebSocket for it.

Server-Sent Events (SSE) is the browser's built-in streaming protocol. One-directional: server pushes, client listens. It reconnects automatically. It works over HTTP/2. It's simpler than WebSocket for every case where you don't need to send data back.

NDJSON (Newline-Delimited JSON) is the same idea over a regular fetch response. One JSON object per line, streamed as the server generates them. This is how most AI APIs stream tokens — and it supports POST requests and custom headers, which SSE's EventSource does not.

tina4-js wraps both in a single module. One function call opens the stream. Six reactive signals track the state. A pipe function accumulates messages into your signals with a reducer. Auto-reconnect handles dropped connections.

---

1. The SSE Client

The tina4-js SSE client provides:

- **Dual mode** — native EventSource for SSE, fetch+ReadableStream for NDJSON
- **Reactive signals** — `status`, `connected`, `lastMessage`, `lastEvent`, `error`, `reconnectCount`
- **Auto-reconnect** — exponential backoff, configurable attempts
- **Signal piping** — stream messages into signals with a reducer

```
import { sse } from 'tina4js';
```

---

2. EventSource Mode (Default)

```
const stream = sse.connect('/api/events');
```

One line. The browser opens an EventSource connection. Messages arrive as the server sends them.

```
import { sse, effect } from 'tina4js';

const stream = sse.connect('/api/events');

effect(() => {
  console.log('Status:', stream.status.value);
});

effect(() => {
```

```
  if (stream.lastMessage.value) {
    console.log('Received:', stream.lastMessage.value);
  }
});
```

---

3. Fetch Mode (NDJSON)

EventSource only supports GET requests with no custom headers. For POST requests, Bearer tokens, or NDJSON streaming, use fetch mode:

```
const stream = sse.connect('/api/chat', {
  mode: 'fetch',
  method: 'POST',
  headers: { 'Authorization': 'Bearer eyJ...' },
  body: { prompt: 'Explain signals in tina4-js' },
});
```

The client reads the response body as a stream, splits on newlines, and parses each line as JSON. Every parsed object becomes a message.

---

4. Options

```
const stream = sse.connect('/api/events', {
  mode: 'eventsource',    // 'eventsource' (default) or 'fetch'
  method: 'GET',           // HTTP method (fetch mode only)
  headers: {},             // Custom headers (fetch mode only)
  body: undefined,         // Request body (fetch mode only, auto-stringified)
  reconnect: true,         // Auto-reconnect on disconnect
  reconnectDelay: 1000,    // Initial delay (ms)
  reconnectMaxDelay: 30000,// Max delay after backoff (ms)
  reconnectAttempts: Infinity, // Max attempts
  events: [],              // Named SSE events (eventsource mode only)
  json: true,              // Auto-parse JSON (default: true)
});
```

Option	Default	Description
`mode`	`'eventsource'`	Transport: native EventSource or fetch+ReadableStream
`method`	`'GET'`	HTTP method (fetch mode only)
`headers`	`{}`	Custom headers (fetch mode only)
`body`	`undefined`	Request body (fetch mode, auto JSON.stringify)

`reconnect`	`true`	Auto-reconnect on disconnect
`reconnectDelay`	`1000`	Initial reconnect delay in ms
`reconnectMaxDelay`	`30000`	Max delay for exponential backoff
`reconnectAttempts`	`Infinity`	Max reconnect attempts
`events`	`[]`	Named SSE events to listen for
`json`	`true`	Auto-parse messages as JSON

---

5. Reactive Signals

Every stream exposes six reactive signals:

```
const stream = sse.connect('/api/events');

stream.status         // Signal<'connecting' | 'open' | 'closed' | 'reconnecting'>
stream.connected      // Signal<boolean>
stream.lastMessage    // Signal<unknown>  — last parsed message
stream.lastEvent      // Signal<string | null>  — SSE event name or null
stream.error          // Signal<Event | Error | null>
stream.reconnectCount // Signal<number>
```

Use them in effects, computed values, or html templates:

```
effect(() => {
  if (stream.connected.value) {
    console.log('Stream is live');
  }
});
```

---

6. Event Handlers

Register listeners for stream events. Every handler returns an unsubscribe function:

```
const unsub = stream.on('message', (data, event?) => {
  console.log('Data:', data);
  console.log('Event name:', event); // SSE event name or undefined
});

stream.on('open', () => console.log('Connected'));
stream.on('close', () => console.log('Disconnected'));
stream.on('error', (err) => console.error('Error:', err));

// Stop listening
unsub();
```

---

7. Named Events (EventSource Mode)

SSE supports named events. By default, EventSource only listens for unnamed `message` events. Pass event names in the options to listen for specific types:

```
const stream = sse.connect('/api/feed', {
  events: ['user_joined', 'message', 'user_left'],
});

stream.on('message', (data, event) => {
  switch (event) {
    case 'user_joined':
      console.log(`${data.name} joined`);
      break;
    case 'message':
      console.log(`${data.author}: ${data.text}`);
      break;
    case 'user_left':
      console.log(`${data.name} left`);
      break;
  }
});

// The lastEvent signal tracks the most recent event name
effect(() => console.log('Last event type:', stream.lastEvent.value));
```

---

8. Pipe to Signal

The pipe pattern streams messages into a signal through a reducer. This is the same pattern as the WebSocket module:

```
import { sse, signal } from 'tina4js';

const messages = signal([]);

const stream = sse.connect('/api/notifications');
stream.pipe(messages, (msg, current) => [...current, msg]);

// messages.value grows as notifications arrive
effect(() => {
  console.log(`${messages.value.length} notifications`);
});
```

Pipe returns an unsubscribe function:

```
const unsub = stream.pipe(messages, (msg, current) => [...current, msg]);

// Stop piping
unsub();
```

---

9. Auto-Reconnect

In EventSource mode, the browser handles reconnection natively. If the connection is fully closed, tina4-js schedules manual reconnection with exponential backoff.

In fetch mode, tina4-js handles all reconnection:

```
const stream = sse.connect('/api/stream', {
  mode: 'fetch',
  reconnect: true,
  reconnectDelay: 1000,       // Start at 1s
  reconnectMaxDelay: 30000,   // Cap at 30s
  reconnectAttempts: 10,      // Give up after 10 tries
});

effect(() => {
  if (stream.status.value === 'reconnecting') {
    console.log(`Reconnect attempt ${stream.reconnectCount.value}`);
  }
});
```

---

10. Closing

```
stream.close();
```

This stops the connection and prevents reconnection. In EventSource mode it calls `source.close()`. In fetch mode it aborts the fetch request.

---

11. SSE vs WebSocket

	SSE	WebSocket
Direction	Server → Client	Bidirectional
Protocol	HTTP	WS/WSS
Reconnect	Built-in (EventSource)	Manual (tina4-js handles it)
Headers	No (EventSource) / Yes (fetch mode)	Subprotocols only
POST body	No (EventSource) / Yes (fetch mode)	N/A
Binary data	No	Yes
HTTP/2 multiplexing	Yes	No
Use case	Notifications, feeds, AI streaming	Chat, gaming, live collaboration

Rule of thumb: If the client only needs to receive, use SSE. If the client needs to send too, use WebSocket.

---

12. Real-World Example: AI Chat Streaming

```
import { sse, signal, html } from 'tina4js';

const messages = signal([]);
const input = signal('');
const streaming = signal(false);

async function sendMessage() {
  const prompt = input.value.trim();
  if (!prompt) return;

  // Add user message
  messages.value = [...messages.value, { role: 'user', text: prompt }];
  input.value = '';

  // Add empty assistant message
  messages.value = [...messages.value, { role: 'assistant', text: '' }];
  streaming.value = true;

  // Stream tokens
  const stream = sse.connect('/api/chat', {
    mode: 'fetch',
    method: 'POST',
    headers: { 'Authorization': `Bearer ${localStorage.getItem('token')}` },
    body: { prompt },
  });

  stream.on('message', (data) => {
    const token = data.token || data;
    const msgs = [...messages.value];
    const last = msgs[msgs.length - 1];
    msgs[msgs.length - 1] = { ...last, text: last.text + token };
    messages.value = msgs;
  });

  stream.on('close', () => {
    streaming.value = false;
  });
}

const view = html`
  <div class="chat">
    ${() => messages.value.map(m => html`
      <div class="message ${m.role}">
        <strong>${m.role}:</strong> ${m.text}
      </div>
    `)}
    <div class="input-bar">
      <input
        type="text"
        .value=${input}
        @input=${(e) => { input.value = e.target.value; }}
        @keydown=${(e) => { if (e.key === 'Enter') sendMessage(); }}
        ?disabled=${streaming}
      />
      <button @click=${sendMessage} ?disabled=${streaming}>Send</button>
    </div>
  </div>
```

```
`;
```

---

13. Real-World Example: Live Notification Feed

```
import { sse, signal, html } from 'tina4js';

const notifications = signal([]);

const stream = sse.connect('/api/notifications', {
  events: ['info', 'warning', 'error'],
});

stream.pipe(notifications, (msg, current) => {
  return [{ ...msg, event: stream.lastEvent.value, time: new Date() }, ...current].slice(0, 50);
});

const view = html`
  <div class="feed">
    <h2>Notifications ${() => stream.connected.value ? '(live)' : '(disconnected)'}</h2>
    ${() => notifications.value.map(n => html`
      <div class="notification ${n.event}">
        <span class="badge">${n.event}</span>
        ${n.message}
        <small>${n.time.toLocaleTimeString()}</small>
      </div>
    `)}
  </div>
`;
```

Bundle Size

Module	Raw	Gzipped
SSE	3.42 KB	1.30 KB

Import only what you need:

```
import { sse } from 'tina4js/sse';  // 1.30 KB gzip
```

Summary

Task	Code
Connect (EventSource)	`sse.connect('/events')`
Connect (NDJSON/POST)	`sse.connect('/api', { mode: 'fetch', method: 'POST', body: {...} })`
Read status	`stream.status.value`
Listen for messages	`stream.on('message', (data, event?) => { ... })`
Named events	`sse.connect(url, { events: ['update', 'delete'] })`
Pipe to signal	`stream.pipe(signal, (msg, current) => [...current, msg])`
Close	`stream.close()`

GraphQL

One Method, Two Worlds

REST gives you URLs. GraphQL gives you a query language. Different philosophies, different trade-offs -- but your frontend shouldn't care. The `api` module you met in Chapter 6 already speaks GraphQL. Same auth. Same token rotation. Same error handling. One extra method and you're talking to a GraphQL endpoint.

No Apollo. No urql. No code generation. You write a query string, pass it to `api.graphql()`, and get back `{ data, errors }`. The framework handles the plumbing.

---

1. Your First Query

`api.graphql()` sends a POST with `{ query, variables }` and returns a typed response.

```
import { api } from "tina4-js";

api.configure({ baseUrl: "/api", auth: true });

const { data, errors } = await api.graphql("/graphql",
    "{ products { id name price } }"
);

if (errors) {
    console.error("GraphQL errors:", errors);
} else {
    console.log(data.products);
}
```

The response shape never changes:

```
{
    data: T | null;          // Your result, or null on failure
    errors?: Array<{ message: string }>;  // Present only when something went wrong
}
```

Query succeeds -- `data` holds your result, `errors` is undefined. Query fails -- `data` is null, `errors` tells you why. No status codes to decode. No response body formats to guess at.

---

2. Variables

Hardcoded values in query strings invite injection and kill reusability. Variables fix both problems. Pass them as the third argument.

```
const { data } = await api.graphql("/graphql",
    `query ($limit: Int!, $offset: Int!) {
        products(limit: $limit, offset: $offset) {
            id
            name
            price
```

```
            stock
        }
    }`,
    { limit: 10, offset: 0 }
);
```

A search query with user input:

```
const { data } = await api.graphql("/graphql",
    `query ($term: String!) {
        search_products(term: $term) {
            id
            name
            price
        }
    }`,
    { term: "widget" }
);
```

The variable goes into the JSON body, not the query string. The server handles escaping. Your frontend stays clean.

---

3. Mutations

Queries read. Mutations write. Same method, different keyword.

Create

```
const { data } = await api.graphql("/graphql",
    `mutation ($input: ProductInput!) {
        create_product(input: $input) {
            id
            name
            price
        }
    }`,
    {
        input: {
            name: "New Widget",
            price: 29.99,
            category_id: 1,
            stock: 100
        }
    }
);

console.log("Created:", data.create_product.id);
```

Update

```
const { data } = await api.graphql("/graphql",
    `mutation ($id: Int!, $input: ProductInput!) {
        update_product(id: $id, input: $input) {
            id
            name
            price
        }
    }`,
    { id: 42, input: { price: 24.99 } }
);
```

Delete

```
const { data } = await api.graphql("/graphql",
    `mutation ($id: Int!) {
        delete_product(id: $id) {
            success
        }
    }`,
    { id: 42 }
);
```

Three operations. One method. The server reads the `query` or `mutation` keyword and knows what to do.

---

4. Authentication

`api.graphql()` runs through the same auth pipeline as every other `api` method. Configure auth once and forget about it.

```
api.configure({
    baseUrl: "/api",
    auth: true,
});

// The Bearer token rides along
const { data } = await api.graphql("/graphql",
    "{ me { id name email role } }"
);
```

Token rotation works too. The server sends a `FreshToken` header. The client stores it. The next request uses the fresh token. You never touch any of this.

---

5. Error Handling

GraphQL responses carry two kinds of failure. The HTTP request itself can fail -- network down, server crashed, 500 response. Or the request succeeds but the query contains errors -- bad field name, permission denied, validation failure.

GraphQL Errors (query-level)

The server returns 200 but `errors` is populated. `data` might still hold partial results.

```
const { data, errors } = await api.graphql("/graphql",
    `{
        products { id name }
        categories { id name }
    }`
);

if (errors) {
    for (const err of errors) {
        console.warn("GraphQL error:", err.message);
    }
}

// Products might have loaded even if categories failed
if (data?.products) {
    renderProducts(data.products);
}
```

Network Errors (transport-level)

The HTTP request never completed. This throws.

```
try {
    const { data } = await api.graphql("/graphql", "{ products { id } }");
} catch (err) {
    console.error("Network error:", err);
}
```

Check `errors` for query problems. Catch exceptions for network problems. Handle both and your UI stays resilient.

---

6. Custom Headers

Per-request headers and query params go in the fourth argument. Same options object as `api.get()` and `api.post()`.

```
const { data } = await api.graphql("/graphql",
    "{ products { id name } }",
    {},
    {
        headers: { "X-Request-Id": "abc123" },
        params: { debug: "true" }
    }
);
```

The empty object in position three means "no variables." The options follow.

---

7. Tina4 Backend Integration

Every Tina4 backend -- Python, PHP, Ruby, Node.js -- generates a GraphQL endpoint from your ORM models. Register a model. The backend builds the schema. The frontend queries it. No SDL files. No resolvers to wire by hand.

The default endpoint is `/api/graphql`.

```
api.configure({ baseUrl: "" });

// The backend generated this query from the User model
const { data } = await api.graphql("/api/graphql",
    "{ users(limit: 10) { id name email } }"
);

// Mutations are generated too
const { data: created } = await api.graphql("/api/graphql",
    `mutation {
        create_user(name: "Alice", email: "alice@example.com") {
            id name email
        }
    }`
);
```

Your ORM models define the schema. The GraphQL layer reads it. The frontend consumes it. Three layers, zero duplication.

---

8. TypeScript

`api.graphql()` accepts a type parameter. Pass an interface that describes the response shape and TypeScript narrows the return type.

```
interface ProductsResponse {
    products: Array<{
        id: number;
        name: string;
        price: number;
    }>;
}

const { data } = await api.graphql<ProductsResponse>("/graphql",
    "{ products { id name price } }"
);

// data is ProductsResponse | null
data?.products.forEach(p => console.log(p.name));
```

The type flows through destructuring. Your editor autocompletes `data.products[0].name`. No type assertions needed.

---

9. Reactive Queries with Signals

GraphQL and signals fit together. A signal changes. An effect fires the query. The result lands in another signal. The UI updates.

```
import { signal, effect, html } from "tina4-js";
import { api } from "tina4-js/api";

const searchTerm = signal("");
const products = signal([]);
const loading = signal(false);

effect(async () => {
    const term = searchTerm.value;
    if (term.length < 2) return;

    loading.value = true;
    const { data } = await api.graphql("/api/graphql",
        `query ($term: String!) {
            search_products(term: $term) { id name price }
        }`,
        { term }
    );
    products.value = data?.search_products || [];
    loading.value = false;
});

const view = html`
    <input @input=${(e) => { searchTerm.value = e.target.value; }}
           placeholder="Search products...">
    ${() => loading.value
        ? html`<p>Loading...</p>`
        : html`<ul>${() => products.value.map(p =>
            html`<li>${p.name} - $${p.price}</li>`
          )}</ul>`
    }
`;
```

The user types. The signal updates. The effect queries. The list renders. No state management library. No cache normalisation layer. Signals and GraphQL, working together.

Summary

Call	What it does
`api.graphql(path, query)`	Run a query
`api.graphql(path, query, variables)`	Run a query with variables
`api.graphql(path, query, variables, options)`	Run a query with custom headers or params

One method. It sends `{ query, variables }` as JSON. It returns `{ data, errors }`. Auth, interceptors, and token rotation carry over from your `api.configure()` call. Everything you learned in Chapter 6 applies here -- GraphQL rides on the same transport.

PWA

Make It Installable

A user opens your app on their phone. They tap "Add to Home Screen." The app icon appears next to Instagram and WhatsApp. They open it on the subway with no signal, and it loads. Every page they visited before -- cached and waiting.

You built that with one function call.

---

1. What pwa.register() Does

```
import { pwa } from 'tina4js';

pwa.register({
  name: 'My App',
  shortName: 'MyApp',
  themeColor: '#1a1a2e',
  cacheStrategy: 'network-first',
  precache: ['/', '/src/public/css/default.css'],
});
```

That one call does three things:

- **Generates a web manifest** and injects it as a `<link rel="manifest">` in the document head
- **Sets the theme-color** meta tag
- **Generates and registers a service worker** with your chosen caching strategy

No build step. No config files. No `manifest.json` to maintain. Everything is generated at runtime from the config you pass.

---

2. Configuration

```
interface PWAConfig {
  name: string;                    // App name (shown in install prompt)
  shortName?: string;              // Short name (shown on home screen)
  themeColor?: string;             // Browser chrome color (default: '#000000')
  backgroundColor?: string;        // Splash screen background (default: '#ffffff')
  display?: 'standalone' | 'fullscreen' | 'minimal-ui' | 'browser';
  icon?: string;                   // Path to app icon
  cacheStrategy?: 'cache-first' | 'network-first' | 'stale-while-revalidate';
  precache?: string[];             // URLs to cache on install
  offlineRoute?: string;           // Fallback URL when offline and cache misses
}
```

Required

Option	Description
`name`	The full app name. Shown in the browser's install prompt and app switcher.

Optional

Option	Default	Description
`shortName`	Same as `name`	Shown on the home screen icon. Keep it short.
`themeColor`	`'#000000'`	Colors the browser chrome (address bar, status bar).
`backgroundColor`	`'#ffffff'`	Splash screen background while the app loads.
`display`	`'standalone'`	How the app looks when installed. `standalone` hides the browser chrome.
`icon`	none	Path to your app icon. Used for both 192x192 and 512x512 sizes.
`cacheStrategy`	`'network-first'`	How the service worker handles fetch requests.
`precache`	`[]`	URLs to download and cache on service worker install.
`offlineRoute`	none	URL to serve when the user is offline and the request is not in the cache.

---

3. Cache Strategies

The cache strategy determines what happens when the browser makes a network request. Three options. Each fits a different kind of application.

network-first (default)

The service worker tries the network. If the network fails, it falls back to the cache.

```
pwa.register({
  name: 'My App',
  cacheStrategy: 'network-first',
});
```

Best for apps where fresh data matters: news sites, dashboards, anything with content that changes by the hour.

How it works:

- Fetch from network
- If successful, cache the response and return it
- If network fails, return cached version
- If nothing cached, show offline fallback

cache-first

The service worker checks the cache first. It goes to the network only on a cache miss.

```
pwa.register({
  name: 'My App',
  cacheStrategy: 'cache-first',
});
```

Best for apps with stable content: documentation sites, reference apps, tools that should work in airplane mode.

How it works:

- Check cache
- If cached, return immediately (fast!)
- If not cached, fetch from network, cache it, return it
- If network fails and not cached, show offline fallback

stale-while-revalidate

The service worker returns the cached version now and updates the cache in the background. The user sees content in milliseconds. The next visit gets the fresh version.

```
pwa.register({
  name: 'My App',
  cacheStrategy: 'stale-while-revalidate',
});
```

Best for apps where speed matters but data should stay current: social feeds, product catalogs, wikis.

How it works:

- If cached, return cached version now
- In the background, fetch from network and update the cache
- Next visit gets the updated version
- If nothing cached, fetch from network

---

4. Precaching

Precache critical assets so they are available on the first offline visit -- before the user has navigated to them:

```
pwa.register({
  name: 'My App',
  cacheStrategy: 'network-first',
  precache: [
    '/',
    '/src/public/css/default.css',
    '/src/public/images/logo.png',
  ],
});
```

The service worker fetches and caches these URLs at install time. The user does not need to visit them first.

Precache your:

- HTML entry point (`/`)
- CSS files
- Critical images (logo, icons)
- Fonts

Do not precache:

- API responses (they change)
- Large files (waste bandwidth)
- User-specific content

The line between "precache" and "do not precache" is simple: if the asset is the same for every user and changes with deployments, precache it. If it varies per user or changes between requests, let the cache strategy handle it at runtime.

---

5. Offline Route

When the user is offline and requests a URL that is not in the cache, the service worker needs somewhere to redirect. That is the offline route:

```
pwa.register({
  name: 'My App',
  offlineRoute: '/offline.html',
  precache: ['/offline.html'],
});
```

Precache the offline route. If you skip this step, the fallback page itself will not be available when the user needs it most.

A simple offline page:

```
<!-- public/offline.html -->
```

```
<!DOCTYPE html>
<html>
<head><title>Offline</title></head>
<body>
  <h1>You are offline</h1>
  <p>Check your connection and try again.</p>
</body>
</html>
```

---

6. The Generated Manifest

`pwa.register()` generates a manifest like this:

```
{
  "name": "My App",
  "short_name": "MyApp",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#1a1a2e",
  "icons": [
    { "src": "/icon.png", "sizes": "192x192", "type": "image/png" },
    { "src": "/icon.png", "sizes": "512x512", "type": "image/png" }
  ]
}
```

It is injected as a Blob URL in a `<link rel="manifest">` tag. You do not need to create or maintain a `manifest.json` file.

Build-Time Generation

If you prefer a static manifest file (for CDN caching or server inspection), use the generation methods:

```
import { pwa } from 'tina4js';

const config = {
  name: 'My App',
  shortName: 'MyApp',
  themeColor: '#1a1a2e',
  icon: '/icon.png',
  cacheStrategy: 'network-first',
  precache: ['/', '/styles.css'],
};

// Get manifest as an object
const manifest = pwa.generateManifest(config);
// Write to file in your build script

// Get service worker as a string
const swCode = pwa.generateServiceWorker(config);
// Write to file: fs.writeFileSync('dist/sw.js', swCode);
```

Runtime generation works for most projects. Build-time generation is there when your deployment pipeline demands static files.

---

7. Complete Example

```
// src/main.ts
import { router, api } from 'tina4js';
import { pwa } from 'tina4js';
import './routes/index';

// Configure API
api.configure({ baseUrl: '/api', auth: true });

// Register PWA
pwa.register({
  name: 'Task Manager',
  shortName: 'Tasks',
  themeColor: '#2563eb',
  backgroundColor: '#f8fafc',
  display: 'standalone',
  icon: '/icon-512.png',
  cacheStrategy: 'network-first',
  precache: [
    '/',
    '/src/public/css/default.css',
    '/icon-512.png',
  ],
  offlineRoute: '/offline',
});

// Start router
router.start({ target: '#root', mode: 'history' });
```

When a user visits this app on a mobile browser, they see an "Add to Home Screen" prompt (if the browser supports it). After installing, the app launches without browser chrome, uses the blue theme color, and works offline for any cached page. The gap between "web app" and "native app" disappears -- and you closed it with twelve lines of configuration.

---

8. Tips

Icons: Provide at least one 512x512 PNG icon. Some browsers refuse to show the install prompt without it.

HTTPS required: Service workers only work over HTTPS (or localhost for development). Vite's dev server handles this for you.

Testing: In Chrome DevTools, use Application > Service Workers to see the registered service worker, and Application > Cache Storage to inspect cached assets. Use the Network panel's "Offline" checkbox to test offline behavior.

Updating: When you change your app and redeploy, the service worker detects the change and updates. The `skipWaiting()` call in the generated service worker ensures the new version activates without waiting for the user to close all tabs.

Scope: The service worker registers at the root scope. It intercepts all GET requests from your domain. Every fetch passes through your chosen cache strategy.

Summary

	How		
	`pwa.register(config)`		
	`name: 'My App'`		
	`themeColor: '#hex'`		
	`cacheStrategy: 'network-first' \	'cache-first' \	'stale-while-re
	`precache: ['/path1', '/path2']`		
	`offlineRoute: '/offline'`		
	`pwa.generateManifest(config)`		
	`pwa.generateServiceWorker(config)`		

Debug Overlay

See Everything

A signal changes. A component mounts. A route resolves. An API call returns a 401. You did not see any of it. You open the console, add a `console.log`, refresh, reproduce the bug, read the output, add another `console.log`, refresh again. Twenty minutes pass. The bug was a misspelled signal label.

The debug overlay shows you everything in real time. One import. One keyboard shortcut. Full visibility into your running application.

---

1. Enabling the Debug Overlay

```
import 'tina4js/debug';
```

That is the entire setup. One import. The overlay is active.

For production builds, conditionally import it:

```
if (import.meta.env.DEV) {
  import('tina4js/debug');
}
```

Vite tree-shakes the debug module away in production. Zero bytes in your final bundle.

When the debug module loads, you see a console message:

```
[tina4] Debug overlay enabled (Ctrl+Shift+D to toggle)
```

---

2. Opening the Overlay

Press **Ctrl+Shift+D** (or **Cmd+Shift+D** on Mac) to toggle the overlay. A panel appears at the bottom of the screen with four tabs: Signals, Components, Routes, and API.

Each tab is a window into a different layer of your application. Together, they replace dozens of `console.log` statements with a live, interactive dashboard.

---

3. The Signals Panel

This panel shows every signal in your application:

- **Label** -- the debug label you passed as the second argument to `signal()`
- **Current value** -- the live value, updating in real time
- **Subscriber count** -- how many effects, computed signals, and DOM bindings depend on this signal

- **Update count** -- how many times the value has changed since creation

This is why debug labels matter:

```
// Without label -- shows as "Signal<number>" in the panel
const count = signal(0);

// With label -- shows as "count" in the panel
const count = signal(0, 'count');
```

Add labels to every signal you might need to debug. The cost is zero when the debug module is not imported.

What to Look For

- **Subscriber count of 0:** This signal is not connected to anything. Either it is unused, or you read `.value` instead of passing the signal directly to a template binding.
- **High update count:** This signal is updating with unusual frequency. Check if you have an effect that writes to it in a loop.
- **Unexpected value:** The signal contains something you did not expect. Trace back to every place that writes to it. The update count tells you how many writes have occurred -- if the count is higher than you expect, something is writing that should not be.

---

4. The Components Panel

This panel shows every mounted `Tina4Element`:

- **Tag name** -- the custom element tag (e.g., `app-header`, `user-card`)
- **Mount status** -- whether the component is in the DOM

When a component mounts, it appears in the list. When it unmounts (removed from DOM), it disappears.

What to Look For

- **Too many instances:** You might be creating components inside a reactive block that re-renders, spawning new elements on every signal change. If you see `user-card` appear 50 times and you only have 10 users, the component is being recreated instead of updated.
- **Missing components:** A component you expected to see is absent. Check two things: does it extend `Tina4Element`, and did you call `customElements.define()` for it? Both must be true for the component to appear in the panel.

---

5. The Routes Panel

This panel shows navigation history:

- **Path** -- the URL that was navigated to
- **Pattern** -- the route pattern that matched

- **Params** -- extracted route parameters
- **Duration** -- how long the route handler took to render (ms)

Every navigation (initial load, link click, `navigate()` call, browser back/forward) creates an entry. The history builds up as you use the app, giving you a timeline of every page transition.

What to Look For

- **Slow routes:** A high `durationMs` value means the route handler is doing expensive work -- a heavy DOM build, a slow API call, or both. Compare durations across routes to find the outliers.
- **Wrong pattern matched:** If a path matched an unexpected pattern, your route registration order might be wrong. The router uses first-match-wins. A broad pattern registered before a specific one swallows the specific route.
- **Guard redirects:** If you see a navigation to a guarded route followed by a redirect to `/login`, the guard is working. If the guarded route renders without a redirect, the guard function has a bug.

---

6. The API Panel

This panel shows every HTTP request made through the `api` client:

- **Method and URL** -- GET /users, POST /auth/login, etc.
- **Status code** -- 200, 404, 500, etc.
- **Request headers** -- what was sent
- **Response data** -- what came back

What to Look For

- **401 responses:** The token might be expired or missing. Check if `auth: true` is configured and the token exists in localStorage.
- **Request body:** Verify that `formToken` is present in POST/PUT/PATCH/DELETE requests when auth is enabled. A missing `formToken` means the API client is not configured for authentication, or the token was cleared.
- **Missing requests:** If an API call is not appearing, the code might be using `fetch()` directly instead of the `api` client. The debug panel tracks only requests that go through `api.get/post/put/patch/delete`.

---

7. How It Works Internally

The debug module hooks into the framework at four points:

- **Signal hooks:** `__debugSignalCreate` and `__debugSignalUpdate` fire when signals are created and updated. These are null in production (tree-shaken away).

- **Component hooks:** `__debugComponentMount` and `__debugComponentUnmount` fire on connectedCallback/disconnectedCallback.
- **Route tracking:** The debug module subscribes to `router.on('change', ...)`.
- **API tracking:** Request and response interceptors register via `api.intercept()`.

All hooks are set to `null` by default. They activate only when `import 'tina4js/debug'` runs. If you never import the debug module, the hooks compile away through tree-shaking. Zero runtime cost in production.

The overlay itself is a Web Component (`<tina4-debug>`) appended to `document.body`. It uses its own Shadow DOM, so its styles never interfere with your application. Your CSS cannot break the overlay. The overlay cannot break your CSS. Complete isolation.

---

8. Best Practices

Always Use Debug Labels

```
// Do this for every signal you care about:
const users = signal<User[]>([], 'users');
const currentPage = signal(1, 'current-page');
const searchQuery = signal('', 'search-query');
const isLoading = signal(false, 'is-loading');
```

When you open the debug panel and see 15 signals, labels are the difference between understanding your app state in seconds and spending minutes cross-referencing source files to figure out which `Signal<string>` is which.

Import Conditionally

```
// src/main.ts
if (import.meta.env.DEV) {
  import('tina4js/debug');
}
```

Never ship the debug overlay to production. It adds weight and exposes internal state to anyone who opens DevTools.

Use During Development, Not Just Debugging

Do not wait for a bug to open the debug overlay. Keep it open while you develop. Watch signals update as you click buttons. Watch API requests flow as pages load. Watch routes resolve as you navigate. The overlay builds your intuition for how data moves through your application -- and that intuition is what makes you fast when a bug does appear.

Summary

What	How
Enable	`import 'tina4js/debug'`
Conditional	`if (import.meta.env.DEV) import('tina4js/debug')`
Toggle	Ctrl+Shift+D
Signals panel	Shows label, value, subscribers, update count
Components panel	Shows mounted Tina4Element instances
Routes panel	Shows navigation history with timing
API panel	Shows requests and responses
Debug labels	`signal(value, 'label')`
Production cost	Zero (tree-shaken away)

tina4-css

Optional Styling

You build a prototype. The HTML is clean. The logic works. But the buttons look like 1998, the form inputs differ between Chrome and Safari, and the layout breaks on mobile. You reach for a CSS framework. It pulls in 300KB of utility classes you will never use.

tina4-css is the alternative. One stylesheet. Good defaults. Drop it in and your app looks professional. Override what you want, ignore the rest. And if you already have your own CSS -- skip this chapter entirely.

---

1. What Is tina4-css

tina4-css is a standalone CSS library. It is not part of tina4-js. It is a separate npm package that gives you:

- A CSS reset
- A responsive grid system
- Styled buttons, forms, tables, cards, badges, alerts
- Navigation components
- Modal dialogs
- Pagination
- A dark theme

The library is designed to look good with zero configuration. No theme files. No customization wizard. No build-time compilation. One file, and your app has a consistent, polished appearance across every browser.

---

2. Installation

With the CLI

```
tina4 init js my-app
```

The Rust CLI does not yet accept a `--css` flag on `init`, so add `tina4-css` to `package.json` after scaffolding, or use the fallback `npx tina4js create my-app --css` which adds the dependency and includes the stylesheet link in `index.html` in one step.

Manual Installation

```
npm install tina4-css
```

Then include it in your `index.html`:

```
<link rel="stylesheet" href="/node_modules/tina4-css/dist/tina4.min.css">
```

Or import it in your main TypeScript file (if your bundler supports CSS imports):

```
import 'tina4-css/dist/tina4.min.css';
```

---

3. The Reset

tina4-css includes a modern CSS reset. It:

- Removes default margins and padding
- Sets `box-sizing: border-box` on everything
- Uses system fonts
- Sets sensible defaults for headings, links, lists, and form elements

Every browser ships with different default styles. A `<button>` in Chrome looks different from a `<button>` in Firefox looks different from a `<button>` in Safari. The reset eliminates these differences. Your app starts from a clean, predictable baseline.

---

4. Grid System

A responsive grid based on CSS Grid:

```
<div class="grid grid-cols-3 gap-4">
  <div>Column 1</div>
  <div>Column 2</div>
  <div>Column 3</div>
</div>
```

Responsive variants:

```
<div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-4">
  <div>Item 1</div>
  <div>Item 2</div>
  <div>Item 3</div>
  <div>Item 4</div>
</div>
```

This renders 1 column on mobile, 2 on medium screens, and 4 on large screens. The layout adapts to the viewport. No JavaScript. No resize listeners. Pure CSS breakpoints.

---

5. Buttons

```
<button class="btn">Default</button>
<button class="btn btn-primary">Primary</button>
<button class="btn btn-secondary">Secondary</button>
<button class="btn btn-danger">Danger</button>
<button class="btn btn-outline">Outline</button>
<button class="btn btn-sm">Small</button>
<button class="btn btn-lg">Large</button>
```

Using with tina4-js:

```
html`
  <button class="btn btn-primary" @click=${handleClick}>
    Save
  </button>
  <button
    class="btn btn-danger"
    @click=${handleDelete}
    ?disabled=${isDeleting}
  >
    ${() => isDeleting.value ? 'Deleting...' : 'Delete'}
  </button>
`
```

The `?disabled` binding and the reactive text work with tina4-css classes without conflict. The framework handles the DOM. The stylesheet handles the appearance. Each stays in its lane.

---

6. Forms

tina4-css styles form elements to look consistent across browsers:

```
<form>
  <div class="form-group">
    <label>Name</label>
    <input type="text" class="form-control" placeholder="Enter name">
  </div>
  <div class="form-group">
    <label>Email</label>
    <input type="email" class="form-control" placeholder="Enter email">
  </div>
  <div class="form-group">
    <label>Role</label>
    <select class="form-control">
      <option>Admin</option>
      <option>Editor</option>
      <option>Viewer</option>
    </select>
  </div>
  <div class="form-group">
    <label>Notes</label>
    <textarea class="form-control" rows="3"></textarea>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

With tina4-js signals:

```
const name = signal('');
const email = signal('');

html`
  <form @submit=${(e: Event) => { e.preventDefault(); handleSubmit(); }}>
    <div class="form-group">
      <label>Name</label>
      <input
        type="text"
```

```
        class="form-control"
        .value=${name}
        @input=${(e: Event) => { name.value = (e.target as HTMLInputElement).value; }}
      />
    </div>
    <div class="form-group">
      <label>Email</label>
      <input
        type="email"
        class="form-control"
        .value=${email}
        @input=${(e: Event) => { email.value = (e.target as HTMLInputElement).value; }}
      />
    </div>
    <button type="submit" class="btn btn-primary">Save</button>
  </form>
`
```

The `.value` binding keeps the input synchronized with the signal. The `form-control` class keeps the input looking good. Signal-driven forms with polished styling -- no extra libraries required.

---

7. Tables

```
<table class="table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th>Role</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Alice</td>
      <td>alice@example.com</td>
      <td>Admin</td>
    </tr>
  </tbody>
</table>
```

Variants:

```
<table class="table table-striped">...</table>
<table class="table table-hover">...</table>
```

With reactive data:

```
const users = signal<User[]>([]);

html`
  <table class="table table-striped">
    <thead>
      <tr><th>Name</th><th>Email</th><th>Actions</th></tr>
    </thead>
    <tbody>
      ${() => users.value.map(user => html`
```

```
        <tr>
          <td>${user.name}</td>
          <td>${user.email}</td>
          <td>
            <button class="btn btn-sm" @click=${() => editUser(user)}>Edit</button>
          </td>
        </tr>
      `)}
    </tbody>
  </table>
`
```

The table re-renders when the `users` signal changes. Striped rows. Hover highlights. Edit buttons on every row. A data table with ten lines of template code.

---

8. Cards

```
<div class="card">
  <div class="card-header">Card Title</div>
  <div class="card-body">
    <p>Card content goes here.</p>
  </div>
  <div class="card-footer">
    <button class="btn btn-primary">Action</button>
  </div>
</div>
```

Card grid:

```
html`
  <div class="grid grid-cols-1 md:grid-cols-3 gap-4">
    ${() => products.value.map(product => html`
      <div class="card">
        <div class="card-body">
          <h3>${product.name}</h3>
          <p>$${product.price}</p>
        </div>
        <div class="card-footer">
          <button class="btn btn-primary btn-sm" @click=${() => addToCart(product)}>
            Add to Cart
          </button>
        </div>
      </div>
    `)}
  </div>
`
```

Three columns on desktop. One column on mobile. Each card has a body and a footer. The grid, the cards, and the reactive list all compose without friction.

---

9. Badges and Alerts

Badges

```
<span class="badge">Default</span>
<span class="badge badge-primary">Primary</span>
<span class="badge badge-success">Success</span>
<span class="badge badge-danger">Danger</span>
<span class="badge badge-warning">Warning</span>
```

Alerts

```
<div class="alert alert-info">This is an informational message.</div>
<div class="alert alert-success">Operation completed.</div>
<div class="alert alert-warning">Please check your input.</div>
<div class="alert alert-danger">Something went wrong.</div>
```

Reactive alerts:

```
const error = signal<string | null>(null);

html`
  ${() => error.value
    ? html`<div class="alert alert-danger">${error}</div>`
    : null
  }
`
```

The alert appears when the error signal has a value. It vanishes when the error clears. No show/hide logic. No CSS transitions to manage. Set the signal. The DOM follows.

---

10. Navigation

```
<nav class="navbar">
  <a class="navbar-brand" href="/">My App</a>
  <div class="navbar-nav">
    <a class="nav-link active" href="/">Home</a>
    <a class="nav-link" href="/about">About</a>
    <a class="nav-link" href="/contact">Contact</a>
  </div>
</nav>
```

With active route tracking:

```
const currentPath = signal('/');
router.on('change', ({ path }) => { currentPath.value = path; });

html`
  <nav class="navbar">
    <a class="navbar-brand" href="/">My App</a>
    <div class="navbar-nav">
      <a class=${() => `nav-link ${currentPath.value === '/' ? 'active' : ''}`} href="/">Home</a
      <a class=${() => `nav-link ${currentPath.value === '/about' ? 'active' : ''}`} href="/abou
    </div>
  </nav>
`
```

The `active` class moves to the current link as the user navigates. The navbar highlights where you are without a single line of imperative DOM manipulation.

---

11. Modals

```
<div class="modal" id="myModal">
  <div class="modal-dialog">
    <div class="modal-header">
      <h3>Confirm Action</h3>
    </div>
    <div class="modal-body">
      <p>Are you sure?</p>
    </div>
    <div class="modal-footer">
      <button class="btn">Cancel</button>
      <button class="btn btn-danger">Delete</button>
    </div>
  </div>
</div>
```

Control with a signal:

```
const showModal = signal(false);

html`
  ${() => showModal.value
    ? html`
        <div class="modal active">
          <div class="modal-dialog">
            <div class="modal-header">
              <h3>Confirm Delete</h3>
            </div>
            <div class="modal-body">
              <p>This action cannot be undone.</p>
            </div>
            <div class="modal-footer">
              <button class="btn" @click=${() => { showModal.value = false; }}>Cancel</button>
              <button class="btn btn-danger" @click=${() => { doDelete(); showModal.value = fals
                Delete
              </button>
            </div>
          </div>
        </div>
      `
    : null
  }
`
```

One signal. One boolean. The modal exists in the DOM when the signal is true and vanishes when it is false. No jQuery. No imperative show/hide methods. The template is the single source of truth for what appears on screen.

---

12. Dark Theme

tina4-css includes a dark theme. Activate it by toggling a class on the `<html>` element:

```
const darkMode = signal(false);

effect(() => {
  document.documentElement.classList.toggle('dark', darkMode.value);
});

html`
  <button @click=${() => { darkMode.value = !darkMode.value; }}>
    ${() => darkMode.value ? 'Light Mode' : 'Dark Mode'}
  </button>
`
```

All tina4-css components adapt to dark mode -- backgrounds, text colors, borders, form elements, cards, tables, alerts. One class toggles the entire palette. You write the toggle. tina4-css handles every surface.

---

13. Using with Shadow DOM Components

Shadow DOM components do not inherit external CSS. If you use `Tina4Element` with Shadow DOM (the default), tina4-css classes will not work inside the component. The Shadow DOM boundary blocks them.

Two options:

Option 1: Light DOM

```
class MyPage extends Tina4Element {
  static shadow = false; // External CSS applies

  render() {
    return html`
      <div class="card">
        <div class="card-body">
          <h3>This uses tina4-css</h3>
        </div>
      </div>
    `;
  }
}
```

Option 2: Import CSS into Shadow DOM

```
class MyComponent extends Tina4Element {
  static styles = `
    @import url('/node_modules/tina4-css/dist/tina4.min.css');
    /* Additional component styles */
  `;

  render() {
    return html`<button class="btn btn-primary">Works!</button>`;
  }
}
```

For most applications, the best balance is light DOM for pages (where tina4-css classes apply) and Shadow DOM for reusable widgets (where encapsulation matters). Pages get the full stylesheet. Widgets get isolation.

Summary

Component	Class
Buttons	`btn`, `btn-primary`, `btn-secondary`, `btn-danger`, `btn-outline`, `btn-sm`, `btn-lg`
Forms	`form-group`, `form-control`
Tables	`table`, `table-striped`, `table-hover`
Cards	`card`, `card-header`, `card-body`, `card-footer`
Badges	`badge`, `badge-primary`, `badge-success`, `badge-danger`, `badge-warning`
Alerts	`alert`, `alert-info`, `alert-success`, `alert-warning`, `alert-danger`
Navigation	`navbar`, `navbar-brand`, `navbar-nav`, `nav-link`, `active`
Modals	`modal`, `modal-dialog`, `modal-header`, `modal-body`, `modal-footer`, `active`
Grid	`grid`, `grid-cols-{n}`, `md:grid-cols-{n}`, `lg:grid-cols-{n}`, `gap-{n}`
Dark mode	Add `dark` class to `<html>`
Pagination	`pagination`

Backend Integration

The Full Stack

Your frontend fetches data. Your backend serves it. Between them sits authentication, CSRF protection, token rotation, and the question every team asks: how do we wire these two together without it becoming a mess?

This chapter answers that question. You will connect tina4-js to tina4-php and tina4-python backends, follow the auth flow from login to token expiry, build for backend embedding, and add interactive islands to server-rendered pages.

---

1. The Tina4 Stack

tina4-js was built to pair with tina4-php and tina4-python. The API client speaks their protocol:

- `Authorization: Bearer <token>` on every request when auth is enabled
- `formToken` injected into POST/PUT/PATCH/DELETE bodies for CSRF protection
- `FreshToken` response header read for token rotation
- JSON in, JSON out

But tina4-js works with any backend that sends JSON and accepts Bearer tokens. The API client is a `fetch()` wrapper with opinions. If your backend follows REST conventions, tina4-js connects to it without modification.

---

2. tina4-js + tina4-php

Setup

Backend (tina4-php):

```
tina4 create my-backend --php
cd my-backend
tina4 serve
# Running on http://localhost:7145
```

Frontend (tina4-js):

```
tina4 init js my-frontend
cd my-frontend
npm install
```

> *In npm-only environments where the `tina4` Rust CLI isn't available, the fallback is `npx tina4js create my-frontend`.*

Configure the Vite proxy to forward API calls:

```
// vite.config.ts
import { defineConfig } from 'vite';
```

```
export default defineConfig({
  server: {
    port: 3000,
    proxy: {
      '/api': 'http://localhost:7145',
    },
  },
});
```

Configure the API client:

```
// src/main.ts
import { api, router } from 'tina4js';
import './routes/index';

api.configure({
  baseUrl: '/api',
  auth: true,
});

router.start({ target: '#root', mode: 'hash' });
```

Now `api.get('/users')` hits `http://localhost:7145/api/users` during development. The Vite proxy handles the forwarding. The browser sees a same-origin request. No CORS headers needed.

tina4-php Routes

```
// src/routes/users.php
\Tina4\Get::add("/api/users", function(\Tina4\Response $response) {
    $users = (new User())->select("*")->asArray();
    return $response($users);
});

\Tina4\Post::add("/api/users", function(\Tina4\Response $response, \Tina4\Request $request) {
    $user = new User();
    $user->name = $request->data->name;
    $user->email = $request->data->email;
    $user->save();
    return $response($user->asArray());
});
```

tina4-js Frontend

```
// src/routes/users.ts
import { route, html, signal, api } from 'tina4js';

route('/users', async () => {
  const users = signal<any[]>([]);
  users.value = await api.get('/users');

  return html`
    <h1>Users</h1>
    <ul>
      ${() => users.value.map(u => html`<li>${u.name} (${u.email})</li>`)}
    </ul>
  `;
});
```

The backend defines the endpoints. The frontend consumes them. The proxy bridges the ports during development. In production, they share the same origin.

---

3. tina4-js + tina4-python

Setup

Backend (tina4-python):

```
tina4 create my-backend --python
cd my-backend
tina4 serve
# Running on http://localhost:7145
```

The Vite proxy and API configuration are identical to the PHP setup. Same config. Same proxy. Same port.

tina4-python Routes

```
# src/routes/users.py
from tina4_python import get, post

@get("/api/users")
async def get_users(request, response):
    users = User().select("*").as_list()
    return response(users)

@post("/api/users")
async def create_user(request, response):
    user = User()
    user.name = request.body.get("name")
    user.email = request.body.get("email")
    user.save()
    return response(user.as_dict())
```

The frontend code is the same whether the backend runs PHP or Python. The API client does not care what language processes the request. It sends JSON. It receives JSON. The language behind the endpoint is invisible to the browser.

---

4. Authentication End-to-End

Here is the complete auth flow, from login to token expiry, in five steps.

Step 1: Login

The frontend sends credentials:

```
const result = await api.post<{ token: string }>('/api/auth/login', {
  email: 'alice@example.com',
  password: 'secret',
});
localStorage.setItem('tina4_token', result.token);
```

The backend validates and returns a JWT:

```
// tina4-php
\Tina4\Post::add("/api/auth/login", function($response, $request) {
    $user = (new User())->find("email = '{$request->data->email}'");
    if ($user && password_verify($request->data->password, $user->password)) {
        $token = \Tina4\Auth::generateToken(["userId" => $user->id]);
        return $response(["token" => $token]);
    }
    return $response(["error" => "Invalid credentials"], 401);
});
```

Step 2: Authenticated Requests

With `auth: true`, every request now includes:

```
Authorization: Bearer <the-jwt-token>
```

For POST/PUT/PATCH/DELETE, the body also includes:

```
{ "name": "Alice", "formToken": "<the-jwt-token>" }
```

The backend validates both the header and the body token. Two layers of verification. The header proves identity. The body token prevents CSRF.

Step 3: Token Rotation

Tokens expire. The backend can issue a fresh token on any response by sending a `FreshToken` header:

```
$freshToken = \Tina4\Auth::generateToken(["userId" => $user->id]);
$response->addHeader("FreshToken", $freshToken);
return $response($data);
```

The API client stores the fresh token in localStorage. The next request uses the new token. The user never sees a login prompt as long as they stay active. Token rotation happens in the background, invisible and automatic.

Step 4: Route Guards

Protect frontend routes:

```
import { signal, computed } from 'tina4js';

const token = signal<string | null>(
  localStorage.getItem('tina4_token'),
  'auth-token'
);

const isLoggedIn = computed(() => token.value !== null);

route('/dashboard', {
  guard: () => isLoggedIn.value || '/login',
  handler: dashboardPage,
});
```

The guard runs before the route handler. If `isLoggedIn` is false, the router redirects to `/login`. The dashboard page never renders. The handler never executes. Guards are the frontend's first line of defense.

Step 5: 401 Handling

If the token expires and rotation did not happen, the backend returns 401. Handle it with a global interceptor:

```
api.intercept('response', (response) => {
  if (response.status === 401) {
    localStorage.removeItem('tina4_token');
    token.value = null;
    navigate('/login', { replace: true });
  }
});
```

Token gone. Signal cleared. User redirected. One interceptor handles every 401 across every API call in the application. No per-request error handling needed.

---

5. Building for Backend Embedding

When your tina4-js app is served by a tina4-php or tina4-python backend (not a separate frontend server), you build the JavaScript bundle and place it where the backend serves static files.

Build

```
npm run build
```

This creates a `dist/` folder with your bundled JavaScript and CSS.

Deploy to Backend

Copy the build output to the backend's public directory:

```
# For tina4-php
cp -r dist/* ../my-backend/src/public/js/

# For tina4-python
cp -r dist/* ../my-backend/src/public/js/
```

Or configure Vite to output to the backend directory:

```
// vite.config.ts
import { defineConfig } from 'vite';

export default defineConfig({
  build: {
    outDir: '../my-backend/src/public/js',
    emptyOutDir: true,
  },
});
```

The CLI Build Command

The tina4 CLI has a build command with target support:

```
npx tina4js build --target php
npx tina4js build --target python
```

This builds and places the output in the conventional location for each backend framework. One command. The files land where the backend expects them.

Backend Template

The backend serves an HTML page that loads the tina4-js bundle:

```
<!-- tina4-php: src/templates/index.twig -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>My App</title>
  <link rel="stylesheet" href="/js/style.css">
</head>
<body>
  <div id="root"></div>
  <script type="module" src="/js/main.js"></script>
</body>
</html>
```

The backend renders this template. The browser loads the JavaScript. The tina4-js router takes over. From the user's perspective, it is a single application. From the developer's perspective, it is two codebases that meet at the HTML page.

---

6. Islands Architecture

Most of this book assumes you are building a single-page application where tina4-js controls the entire page. Islands architecture flips that model. The server renders the page. tina4-js controls small, specific sections -- islands of interactivity in a sea of static HTML.

This approach works when:

- You have existing server-rendered pages and want to add interactivity without rewriting them
- You want fast initial loads with progressive enhancement
- SEO matters and you need server-rendered content that search engines can index

How It Works

- The backend renders the full HTML page
- tina4-js components hydrate specific sections
- Each component is independent -- it manages its own state and DOM

Example: Adding a Live Search to a Server-Rendered Page

The server renders the page:

```
<!-- Server-rendered page -->
<div class="product-listing">
  <h1>Products</h1>

  <!-- This island becomes interactive -->
```

```
  <product-search api-url="/api/products"></product-search>

  <!-- This stays static -->
  <footer>Copyright 2024</footer>
</div>

<script type="module">
  import { Tina4Element, html, signal, api } from 'tina4js';

  class ProductSearch extends Tina4Element {
    static props = { 'api-url': String };
    static shadow = false;

    render() {
      const query = signal('');
      const results = signal<any[]>([]);

      const search = async () => {
        if (query.value.length < 2) return;
        results.value = await api.get(
          this.prop<string>('api-url').value,
          { params: { q: query.value } }
        );
      };

      return html`
        <div>
          <input
            type="search"
            placeholder="Search products..."
            @input=${(e: Event) => {
              query.value = (e.target as HTMLInputElement).value;
              search();
            }}
          />
          <ul>
            ${() => results.value.map(p => html`
              <li>${p.name} - $${p.price}</li>
            `)}
          </ul>
        </div>
      `;
    }
  }

  customElements.define('product-search', ProductSearch);
</script>
```

The server sends HTML. The browser parses it. When it reaches the `<product-search>` tag, the custom element activates. The search input becomes live. The rest of the page stays static. The server did the heavy lifting. tina4-js added the interactivity.

Multiple Islands

Each island is a separate component. They do not need to know about each other:

```
<body>
  <!-- Island 1: Live search -->
  <product-search></product-search>
```

```
  <!-- Static server-rendered content -->
  <section class="featured">
    <h2>Featured Products</h2>
    <!-- server-rendered list -->
  </section>

  <!-- Island 2: Shopping cart -->
  <cart-widget></cart-widget>

  <!-- Island 3: Notification bell -->
  <notification-bell user-id="42"></notification-bell>
</body>
```

Each component self-initializes when the browser parses its tag. No router needed. No app shell. No bootstrapping ceremony. Place the tags where you want interactivity. The components wake up on their own.

Shared State Between Islands

If islands need to share state, use the store pattern from Chapter 4:

```
// store.ts
export const cart = signal<CartItem[]>([], 'cart');
export const cartCount = computed(() => cart.value.length);
```

Both `product-search` (add to cart) and `cart-widget` (display cart) import from the same store. When a user adds an item through the search island, the cart count updates in the cart widget. Two components. One signal. Synchronized without either knowing the other exists.

---

7. Development Workflow

Separate Frontend and Backend

Best for larger projects with dedicated frontend and backend teams.

```
my-project/
  frontend/      # tina4-js project
    src/
    package.json
    vite.config.ts
  backend/       # tina4-php or tina4-python
    src/
```

The frontend runs on `localhost:3000` with Vite. The backend runs on `localhost:7145`. Vite proxies API calls. Each team works in its own directory, its own repository, its own deployment pipeline.

Embedded Frontend

Best for smaller projects or when one team owns both layers.

```
my-project/
  src/
    routes/        # Backend routes
    templates/     # Backend templates
    orm/           # Backend ORM models
```

```
  public/
    js/          # Built tina4-js output goes here
frontend/        # tina4-js source
  src/
  package.json
  vite.config.ts  # builds to ../src/public/js/
```

One repository. One deployment. The frontend builds into the backend's public directory. The backend serves everything. For a team of one or two developers, this is the fastest path from code to production.

Summary

What	How
Proxy API in dev	Vite `proxy: { '/api': 'http://localhost:7145' }`
Configure API	`api.configure({ baseUrl: '/api', auth: true })`
Auth token	Stored in localStorage, auto-sent as Bearer
CSRF token	Auto-injected as `formToken` in request bodies
Token rotation	Automatic via `FreshToken` response header
401 handling	Response interceptor redirects to login
Build for backend	`npx tina4js build --target php` or `--target python`
Islands	Use `Tina4Element` components in server-rendered HTML
Shared state	Export signals from a store module

Building a Complete App

The Admin Dashboard

Picture the screen. A dark navbar across the top. Four stat cards showing live numbers -- total users, active sessions, revenue, orders -- updating without a page refresh. A notification bell with a red badge. A data table with search, pagination, and inline editing. A login page that guards everything behind it.

This chapter builds that dashboard from scratch. It uses every tina4-js module: signals, html, components, routing, API, WebSocket, and PWA. By the end, you will have a complete, deployable application -- and you will have seen how all the pieces from previous chapters fit together in production.

---

1. Project Setup

```
tina4 init js admin-dashboard
cd admin-dashboard
npm install
```

> *The Rust CLI `tina4 init js` does not yet accept `--css`. To pull in Tina4 CSS, either add `tina4-css` to `package.json` after scaffolding, or use the npm-only fallback `npx tina4js create admin-dashboard --css` which wires it in automatically.*

We will build this structure:

```
src/
  main.ts              # Entry point
  store.ts             # Global state
  routes/
    index.ts           # Route definitions
  pages/
    login.ts           # Login page
    dashboard.ts       # Dashboard with live stats
    users.ts           # CRUD user management
  components/
    app-layout.ts      # Layout wrapper with nav
    data-table.ts      # Reusable data table
    stat-card.ts       # Statistics card
    notification-bell.ts  # WebSocket notification bell
```

---

2. Global Store

Start with the shared state. Authentication drives the entire application -- which pages render, which API calls include tokens, what the navbar displays. The store is the foundation.

```
// src/store.ts
import { signal, computed } from 'tina4js';

// Auth
```

```
export const token = signal<string | null>(
  localStorage.getItem('tina4_token'),
  'auth-token'
);
export const user = signal<{ id: number; name: string; role: string } | null>(
  null,
  'current-user'
);
export const isLoggedIn = computed(() => token.value !== null);
export const isAdmin = computed(() => user.value?.role === 'admin');

export function setAuth(newToken: string, userData: { id: number; name: string; role: string }) {
  localStorage.setItem('tina4_token', newToken);
  token.value = newToken;
  user.value = userData;
}

export function clearAuth() {
  localStorage.removeItem('tina4_token');
  token.value = null;
  user.value = null;
}

// Notifications
export const notifications = signal<{ id: number; text: string; time: string }[]>(
  [],
  'notifications'
);
export const unreadCount = computed(() => notifications.value.length);
```

Every signal has a debug label. Every piece of derived state is a computed signal. The store exports functions for mutations, not raw signal writes. This discipline pays for itself the moment you open the debug overlay and see named, traceable state.

---

3. App Entry

```
// src/main.ts
import { api, router } from 'tina4js';
import { pwa } from 'tina4js';
import { token, clearAuth } from './store';
import { navigate } from 'tina4js';
import './routes/index';
import './components/app-layout';
import './components/data-table';
import './components/stat-card';
import './components/notification-bell';

// Debug overlay in dev only
if (import.meta.env.DEV) {
  import('tina4js/debug');
}

// API configuration
api.configure({
  baseUrl: '/api',
  auth: true,
});

// Global 401 handler
api.intercept('response', (response) => {
  if (response.status === 401) {
    clearAuth();
    navigate('/login', { replace: true });
  }
});

// PWA
pwa.register({
  name: 'Admin Dashboard',
  shortName: 'Admin',
  themeColor: '#1e293b',
  cacheStrategy: 'network-first',
  precache: ['/'],
});

// Start router
router.start({ target: '#root', mode: 'hash' });
```

The entry point is the wiring diagram. API configured. Interceptor registered. PWA enabled. Debug overlay loaded in development. Router started. Every cross-cutting concern lives here, declared once, applied everywhere.

---

4. Routes

```
// src/routes/index.ts
import { route } from 'tina4js';
import { isLoggedIn } from '../store';
import { loginPage } from '../pages/login';
import { dashboardPage } from '../pages/dashboard';
import { usersPage } from '../pages/users';
import { html } from 'tina4js';

// Public
route('/login', loginPage);

// Protected
route('/', {
  guard: () => isLoggedIn.value || '/login',
  handler: dashboardPage,
});

route('/users', {
  guard: () => isLoggedIn.value || '/login',
  handler: usersPage,
});

// 404
route('*', () => html`
  <app-layout>
    <h1>404 - Page Not Found</h1>
    <a href="/">Go to Dashboard</a>
  </app-layout>
`);
```

Four routes. One is public. Two are guarded. One catches everything else. The guard is the same computed signal for both protected routes -- `isLoggedIn` returns true or redirects to `/login`. The entire routing table fits on one screen.

---

5. Layout Component

```
// src/components/app-layout.ts
import { Tina4Element, html } from 'tina4js';
import { user, isLoggedIn, clearAuth, unreadCount } from '../store';
import { navigate } from 'tina4js';

class AppLayout extends Tina4Element {
  static shadow = false;

  render() {
    return html`
      <div style="min-height: 100vh; display: flex; flex-direction: column;">
        ${() => isLoggedIn.value
          ? html`
              <nav class="navbar" style="background: #1e293b; color: white; padding: 0.75rem 1.5
                <a class="navbar-brand" href="/" style="color: white; text-decoration: none; fon
                  Admin Dashboard
                </a>
                <div style="display: flex; align-items: center; gap: 1rem;">
                  <a href="/" style="color: #94a3b8; text-decoration: none;">Dashboard</a>
                  <a href="/users" style="color: #94a3b8; text-decoration: none;">Users</a>
                  <notification-bell></notification-bell>
                  <span style="color: #94a3b8;">${() => user.value?.name ?? ''}</span>
                  <button
                    class="btn btn-sm"
                    style="background: #475569; color: white; border: none;"
                    @click=${() => { clearAuth(); navigate('/login'); }}
                  >Logout</button>
                </div>
              </nav>
            `
          : null
        }
        <main style="flex: 1; padding: 1.5rem; max-width: 1200px; width: 100%; margin: 0 auto;">
          <slot></slot>
        </main>
      </div>
    `;
  }
}

customElements.define('app-layout', AppLayout);
```

The layout wraps every page. It renders the navbar when the user is logged in and hides it when they are not. The notification bell sits in the navbar. The user's name appears next to the logout button. The `<slot>` element passes through whatever the route handler renders.

One component. Every page gets a consistent shell.

---

6. Login Page

```
// src/pages/login.ts
import { signal, html, api, navigate, batch } from 'tina4js';
import { setAuth } from '../store';

export function loginPage() {
  const email = signal('', 'login-email');
  const password = signal('', 'login-password');
  const error = signal<string | null>(null, 'login-error');
  const loading = signal(false, 'login-loading');

  const handleLogin = async (e: Event) => {
    e.preventDefault();
    loading.value = true;
    error.value = null;

    try {
      const result = await api.post<{
        token: string;
        user: { id: number; name: string; role: string };
      }>('/auth/login', {
        email: email.value,
        password: password.value,
      });

      setAuth(result.token, result.user);
      navigate('/');
    } catch (err: any) {
      error.value = err.data?.message ?? 'Login failed. Please try again.';
    } finally {
      loading.value = false;
    }
  };

  return html`
    <div style="max-width: 400px; margin: 4rem auto; padding: 2rem;">
      <h1 style="text-align: center; margin-bottom: 2rem;">Admin Login</h1>

      ${() => error.value
        ? html`<div class="alert alert-danger">${error}</div>`
        : null
      }

      <form @submit=${handleLogin}>
        <div class="form-group">
          <label>Email</label>
          <input
            type="email"
            class="form-control"
            placeholder="admin@example.com"
            .value=${email}
            @input=${(e: Event) => { email.value = (e.target as HTMLInputElement).value; }}
            ?disabled=${loading}
            required
          />
        </div>

        <div class="form-group">
```

```
            <label>Password</label>
            <input
              type="password"
              class="form-control"
              placeholder="Enter password"
              .value=${password}
              @input=${(e: Event) => { password.value = (e.target as HTMLInputElement).value; }}
              ?disabled=${loading}
              required
            />
          </div>

          <button
            type="submit"
            class="btn btn-primary"
            style="width: 100%"
            ?disabled=${loading}
          >
            ${() => loading.value ? 'Logging in...' : 'Login'}
          </button>
        </form>
      </div>
    `;
  }
```

Four signals drive this page. `email` and `password` hold the form state. `error` displays feedback. `loading` disables the form during the API call. The login handler calls the store's `setAuth` on success and navigates to the dashboard. On failure, the error signal updates and the alert appears.

No form library. No validation library. Signals and HTML.

---

7. Stat Card Component

```
// src/components/stat-card.ts
import { Tina4Element, html } from 'tina4js';

class StatCard extends Tina4Element {
  static props = { label: String, value: String, color: String };

  static styles = `
    :host { display: block; }
    .stat {
      padding: 1.5rem;
      border-radius: 8px;
      background: white;
      border: 1px solid #e2e8f0;
    }
    .label { font-size: 0.875rem; color: #64748b; margin-bottom: 0.25rem; }
    .value { font-size: 2rem; font-weight: 700; }
  `;

  render() {
    return html`
      <div class="stat">
        <div class="label">${this.prop('label')}</div>
        <div class="value" style=${() => `color: ${this.prop<string>('color').value || '#1e293b'
          ${this.prop('value')}
        </div>
      </div>
    `;
  }
}

customElements.define('stat-card', StatCard);
```

A small, focused component. Three props. Shadow DOM for style isolation. The value updates reactively when the parent passes a new attribute. Drop four of these in a grid and you have a statistics dashboard.

---

8. Notification Bell Component

```
// src/components/notification-bell.ts
import { Tina4Element, html, signal } from 'tina4js';
import { notifications, unreadCount } from '../store';

class NotificationBell extends Tina4Element {
  static styles = `
    :host { display: inline-block; position: relative; cursor: pointer; }
    .bell { font-size: 1.25rem; }
    .badge {
      position: absolute; top: -6px; right: -8px;
      background: #ef4444; color: white;
      font-size: 0.625rem; font-weight: 700;
      width: 18px; height: 18px;
      border-radius: 50%;
      display: flex; align-items: center; justify-content: center;
    }
    .dropdown {
      position: absolute; top: 100%; right: 0; margin-top: 0.5rem;
      background: white; border: 1px solid #e2e8f0; border-radius: 8px;
      width: 300px; max-height: 400px; overflow-y: auto;
      box-shadow: 0 4px 12px rgba(0,0,0,0.1); z-index: 100;
    }
    .item { padding: 0.75rem 1rem; border-bottom: 1px solid #f1f5f9; }
    .item:last-child { border-bottom: none; }
    .time { font-size: 0.75rem; color: #94a3b8; }
    .empty { padding: 1rem; text-align: center; color: #94a3b8; }
  `;

  render() {
    const open = signal(false);

    return html`
      <div @click=${() => { open.value = !open.value; }}>
        <span class="bell">&#128276;</span>
        ${() => unreadCount.value > 0
          ? html`<span class="badge">${unreadCount}</span>`
          : null
        }
      </div>

      ${() => open.value
        ? html`
            <div class="dropdown">
              ${() => notifications.value.length === 0
                ? html`<div class="empty">No notifications</div>`
                : null
              }
              ${() => notifications.value.map(n => html`
                <div class="item">
                  <div>${n.text}</div>
                  <div class="time">${n.time}</div>
                </div>
              `)}
              ${() => notifications.value.length > 0
                ? html`
                    <div style="padding: 0.5rem; text-align: center;">
                      <button
```

```
                        style="border: none; background: none; color: #3b82f6; cursor: pointer;"
                        @click=${() => { notifications.value = []; }}
                      >Clear all</button>
                    </div>
                  `
                : null
              }
            </div>
          `
        : null
      }
    `;
  }
}

customElements.define('notification-bell', NotificationBell);
```

The bell reads from the global notification store. The badge appears when `unreadCount` is greater than zero. The dropdown opens on click. Each notification shows its text and timestamp. The "Clear all" button empties the array. All of this -- the badge count, the dropdown toggle, the notification list, the clear action -- runs on signals. No imperative state management. No event bus.

---

9. Dashboard Page with Live Stats

```
// src/pages/dashboard.ts
import { signal, computed, html, api, ws, batch } from 'tina4js';
import { user, notifications } from '../store';

export function dashboardPage() {
  // Stats from API
  const stats = signal({
    totalUsers: 0,
    activeUsers: 0,
    revenue: 0,
    orders: 0,
  }, 'dashboard-stats');

  const loading = signal(true, 'dashboard-loading');

  // Load initial stats
  async function loadStats() {
    loading.value = true;
    try {
      stats.value = await api.get('/dashboard/stats');
    } finally {
      loading.value = false;
    }
  }

  loadStats();

  // Live updates via WebSocket
  const socket = ws.connect(`${location.protocol === 'https:' ? 'wss:' : 'ws:'}//${location.host

  // Pipe live stat updates
  socket.pipe(stats, (msg, current) => {
    const m = msg as { type: string; data: any };
    if (m.type === 'stats_update') {
      return { ...current, ...m.data };
    }
    return current;
  });

  // Pipe notifications
  socket.pipe(notifications, (msg, current) => {
    const m = msg as { type: string; text: string; time: string };
    if (m.type === 'notification') {
      return [{ id: Date.now(), text: m.text, time: m.time }, ...current].slice(0, 50);
    }
    return current;
  });

  // Recent activity
  const activity = signal<{ action: string; user: string; time: string }[]>([], 'recent-activity

  socket.pipe(activity, (msg, current) => {
    const m = msg as { type: string; action: string; user: string; time: string };
    if (m.type === 'activity') {
      return [{ action: m.action, user: m.user, time: m.time }, ...current].slice(0, 20);
    }
    return current;
```

```
});

return html`
  <app-layout>
    <h1>Dashboard</h1>
    <p style="color: #64748b; margin-bottom: 1.5rem;">
      Welcome back, ${() => user.value?.name ?? 'User'}
    </p>

    ${() => loading.value
      ? html`<p>Loading stats...</p>`
      : html`
          <div style="display: grid; grid-template-columns: repeat(4, 1fr); gap: 1rem; margin-
            <stat-card
              label="Total Users"
              value=${() => String(stats.value.totalUsers)}
              color="#3b82f6"
            ></stat-card>
            <stat-card
              label="Active Now"
              value=${() => String(stats.value.activeUsers)}
              color="#10b981"
            ></stat-card>
            <stat-card
              label="Revenue"
              value=${() => `$${stats.value.revenue.toLocaleString()}`}
              color="#8b5cf6"
            ></stat-card>
            <stat-card
              label="Orders Today"
              value=${() => String(stats.value.orders)}
              color="#f59e0b"
            ></stat-card>
          </div>
        `
    }

    <div style="display: grid; grid-template-columns: 1fr 1fr; gap: 1.5rem;">
      <div>
        <h2>Live Connection</h2>
        <p style="margin-bottom: 1rem;">
          Status:
          <span style=${() =>
            socket.status.value === 'open'
              ? 'color: #10b981'
              : 'color: #ef4444'
          }>${socket.status}</span>
        </p>
      </div>

      <div>
        <h2>Recent Activity</h2>
        <div style="max-height: 300px; overflow-y: auto;">
          ${() => activity.value.length === 0
            ? html`<p style="color: #94a3b8;">No recent activity</p>`
            : null
          }
          ${() => activity.value.map(a => html`
            <div style="padding: 0.5rem 0; border-bottom: 1px solid #f1f5f9;">
```

```
                <strong>${a.user}</strong> ${a.action}
                <span style="color: #94a3b8; font-size: 0.75rem; margin-left: 0.5rem;">${a.time}
              </div>
            `)}
          </div>
        </div>
      </div>
    </app-layout>
  `;
}
```

The page loads stats from the API, then opens a WebSocket for live updates. Three `pipe()` calls route incoming messages to three different signals: stats, notifications, and activity. The stat cards update in real time. Notifications push into the bell component. Activity entries stream into the feed.

The API provides the initial snapshot. The WebSocket provides the deltas. The signals merge both into a single reactive view.

---

10. CRUD Users Page

```
// src/pages/users.ts
import { signal, computed, html, api, batch } from 'tina4js';

interface User {
  id: number;
  name: string;
  email: string;
  role: string;
  active: boolean;
}

export function usersPage() {
  const users = signal<User[]>([], 'users-list');
  const loading = signal(true, 'users-loading');
  const search = signal('', 'users-search');
  const page = signal(1, 'users-page');
  const perPage = 10;

  // Editing state
  const editingUser = signal<User | null>(null, 'editing-user');
  const showCreateForm = signal(false, 'show-create-form');
  const formName = signal('');
  const formEmail = signal('');
  const formRole = signal('editor');
  const formError = signal<string | null>(null);

  // Filtered users
  const filtered = computed(() => {
    const q = search.value.toLowerCase();
    if (!q) return users.value;
    return users.value.filter(u =>
      u.name.toLowerCase().includes(q) ||
      u.email.toLowerCase().includes(q)
    );
  });

  // Paginated
  const paginated = computed(() => {
    const start = (page.value - 1) * perPage;
    return filtered.value.slice(start, start + perPage);
  });

  const totalPages = computed(() => Math.ceil(filtered.value.length / perPage));

  // Load
  async function loadUsers() {
    loading.value = true;
    try {
      users.value = await api.get<User[]>('/users');
    } finally {
      loading.value = false;
    }
  }

  // Create
  async function createUser() {
    formError.value = null;
```

```
  try {
    const newUser = await api.post<User>('/users', {
      name: formName.value,
      email: formEmail.value,
      role: formRole.value,
    });
    batch(() => {
      users.value = [...users.value, newUser];
      showCreateForm.value = false;
      formName.value = '';
      formEmail.value = '';
      formRole.value = 'editor';
    });
  } catch (err: any) {
    formError.value = err.data?.message ?? 'Failed to create user';
  }
}

// Update
async function updateUser() {
  if (!editingUser.value) return;
  formError.value = null;
  try {
    const updated = await api.put<User>(`/users/${editingUser.value.id}`, {
      name: formName.value,
      email: formEmail.value,
      role: formRole.value,
    });
    batch(() => {
      users.value = users.value.map(u => u.id === updated.id ? updated : u);
      editingUser.value = null;
    });
  } catch (err: any) {
    formError.value = err.data?.message ?? 'Failed to update user';
  }
}

// Delete
async function deleteUser(id: number) {
  try {
    await api.delete(`/users/${id}`);
    users.value = users.value.filter(u => u.id !== id);
  } catch (err: any) {
    formError.value = err.data?.message ?? 'Failed to delete user';
  }
}

// Start editing
function startEdit(user: User) {
  batch(() => {
    editingUser.value = user;
    showCreateForm.value = false;
    formName.value = user.name;
    formEmail.value = user.email;
    formRole.value = user.role;
    formError.value = null;
  });
}
```

```
function cancelEdit() {
  batch(() => {
    editingUser.value = null;
    showCreateForm.value = false;
    formError.value = null;
  });
}

loadUsers();

return html`
  <app-layout>
    <div style="display: flex; justify-content: space-between; align-items: center; margin-bot
      <h1>Users</h1>
      <button class="btn btn-primary" @click=${() => {
        batch(() => {
          showCreateForm.value = true;
          editingUser.value = null;
          formName.value = '';
          formEmail.value = '';
          formRole.value = 'editor';
          formError.value = null;
        });
      }}>Add User</button>
    </div>

    ${() => formError.value
      ? html`<div class="alert alert-danger">${formError}</div>`
      : null
    }

    ${() => (showCreateForm.value || editingUser.value)
      ? html`
          <div class="card" style="margin-bottom: 1.5rem;">
            <div class="card-header">
              ${() => editingUser.value ? 'Edit User' : 'Create User'}
            </div>
            <div class="card-body">
              <div style="display: grid; grid-template-columns: 1fr 1fr 1fr auto; gap: 1rem; a
                <div class="form-group">
                  <label>Name</label>
                  <input class="form-control" .value=${formName}
                    @input=${(e: Event) => { formName.value = (e.target as HTMLInputElement).v
                </div>
                <div class="form-group">
                  <label>Email</label>
                  <input class="form-control" type="email" .value=${formEmail}
                    @input=${(e: Event) => { formEmail.value = (e.target as HTMLInputElement).
                </div>
                <div class="form-group">
                  <label>Role</label>
                  <select class="form-control" .value=${formRole}
                    @change=${(e: Event) => { formRole.value = (e.target as HTMLSelectElement)
                    <option value="admin">Admin</option>
                    <option value="editor">Editor</option>
                    <option value="viewer">Viewer</option>
                  </select>
                </div>
                <div style="display: flex; gap: 0.5rem;">
```

```
              <button class="btn btn-primary"
                @click=${() => editingUser.value ? updateUser() : createUser()}>
                ${() => editingUser.value ? 'Update' : 'Create'}
              </button>
              <button class="btn" @click=${cancelEdit}>Cancel</button>
            </div>
          </div>
        </div>
      </div>
    `
  : null
}

<div style="margin-bottom: 1rem;">
  <input
    type="search"
    class="form-control"
    placeholder="Search users..."
    .value=${search}
    @input=${(e: Event) => {
      search.value = (e.target as HTMLInputElement).value;
      page.value = 1;
    }}
  />
</div>

${() => loading.value
  ? html`<p>Loading users...</p>`
  : html`
      <table class="table table-striped table-hover">
        <thead>
          <tr>
            <th>Name</th>
            <th>Email</th>
            <th>Role</th>
            <th>Status</th>
            <th>Actions</th>
          </tr>
        </thead>
        <tbody>
          ${() => paginated.value.map(u => html`
            <tr>
              <td>${u.name}</td>
              <td>${u.email}</td>
              <td><span class="badge">${u.role}</span></td>
              <td>
                <span class=${`badge ${u.active ? 'badge-success' : 'badge-danger'}`}>
                  ${u.active ? 'Active' : 'Inactive'}
                </span>
              </td>
              <td>
                <button class="btn btn-sm" @click=${() => startEdit(u)}>Edit</button>
                <button class="btn btn-sm btn-danger" @click=${() => deleteUser(u.id)}>Del
              </td>
            </tr>
          `)}
        </tbody>
      </table>
```

```
            <div style="display: flex; justify-content: space-between; align-items: center; marg
              <span style="color: #64748b;">${filtered} total users</span>
              <div style="display: flex; gap: 0.5rem;">
                <button class="btn btn-sm" ?disabled=${() => page.value <= 1}
                  @click=${() => { page.value = page.value - 1; }}>Previous</button>
                <span style="padding: 0.25rem 0.5rem;">
                  Page ${page} of ${totalPages}
                </span>
                <button class="btn btn-sm" ?disabled=${() => page.value >= totalPages.value}
                  @click=${() => { page.value = page.value + 1; }}>Next</button>
              </div>
            </div>
          `
      }
    </app-layout>
  `;
}
```

This is the most complex page in the application, and it demonstrates the full CRUD lifecycle. Load data from the API. Display it in a searchable, paginated table. Create new records. Edit existing ones. Delete them. Handle errors. Reset form state.

The `batch()` calls group multiple signal writes into single DOM updates. The `computed` signals derive the filtered and paginated lists from the raw data. The search input resets pagination to page 1. Every interaction flows through signals.

---

11. What We Built

This dashboard uses every module in the framework:

Module	Usage
Signals	`token`, `user`, `notifications`, `stats`, `users`, form state
Computed	`isLoggedIn`, `isAdmin`, `filtered`, `paginated`, `totalPages`, `unreadCount`
Effects	Debug overlay auto-tracking, signal subscriptions in templates
Batch	Grouping form state resets, edit mode transitions
html	Every page and component template
Tina4Element	`stat-card`, `notification-bell`, `app-layout`
Routing	Login, dashboard, users, 404 with guards
API	Login, CRUD operations, stats loading, 401 interceptor

WebSocket	Live stats, notifications, activity feed
PWA	Installable with network-first caching
Debug	Labels on every signal, conditional import

Every module. One application. Deployable as-is.

The entire frontend -- framework, components, routes, state management, everything -- ships under 10KB gzipped. Try building this dashboard with React, Vue, or Angular. Count the packages. Measure the bundle.

Summary

A complete, production-grade admin dashboard does not need 50 npm packages, a state management library, a routing library, a form library, and a CSS-in-JS solution. It needs tina4-js and clear thinking.

The patterns repeat:

- **Signals for state.** One signal per piece of data.
- **Computed for derived data.** Filtering, pagination, auth checks.
- **Batch for grouped writes.** Form resets, mode transitions.
- **html for templates.** Reactive text, reactive blocks, events.
- **Components for reuse.** Props in, events out.
- **Routes with guards.** Public and protected.
- **API with interceptors.** Auth, error handling.
- **WebSocket for live data.** Pipe into signals.

Eight patterns. One framework. Now build your own.

Patterns and Pitfalls

What We Learned the Hard Way

You push a feature to production. The list does not update when items are added. You stare at the code for ten minutes. The logic is correct. The API returns the right data. The signal receives the value. But the DOM does not move.

Then you see it: `items.value.push(newItem)`. A mutation. No new reference. The signal never fired.

Every pitfall in this chapter was hit by a real developer on a real project. Every pattern was discovered through trial and error. This is the chapter you read before you ship -- and the chapter you return to when something breaks and you cannot figure out why.

---

1. When to Use tina4-js

Use tina4-js when:

- **Bundle size matters.** 1.5KB core gzipped, under 6KB for the full framework. Your app loads in under a second on a 3G connection.
- **You want simplicity.** Seven modules. One package. No decisions about state management libraries, routing libraries, or CSS-in-JS solutions.
- **You are building for a tina4 backend.** The API client speaks tina4-php and tina4-python natively -- auth tokens, CSRF protection, and token rotation work without configuration.
- **You want real Web Components.** Components that work outside the framework, in plain HTML, in other frameworks, in any context where custom elements are supported.
- **You are building islands.** Adding interactivity to server-rendered pages without taking over the entire DOM.
- **You are learning.** The entire source code is under 1,000 lines. You can read every module in an afternoon and understand how the framework works at the implementation level.

Use React, Vue, or Svelte when:

- **You need server-side rendering (SSR).** tina4-js is client-side only. If you need the server to render HTML for SEO or initial load performance, use a framework with SSR support.
- **You need a massive ecosystem.** Thousands of third-party components, hooks, plugins, and integrations.
- **Your team already knows React/Vue.** Developer familiarity has real value. Retraining a team has real cost.
- **You need fine-grained list reconciliation.** tina4-js re-renders entire lists when the signal changes. For lists of thousands of items with frequent individual updates, keyed reconciliation (React's virtual DOM, Svelte's each blocks) is the better tool.

Pick the right tool for the job. tina4-js is not trying to replace React. It is trying to replace the 90% of projects that never needed React in the first place.

---

2. The New-Reference Rule

This is the most important rule in tina4-js. It trips up everyone -- beginners and experienced developers alike.

Signals use `Object.is()` for change detection. `Object.is()` compares by reference for objects and arrays. If you mutate in place, the reference does not change. The signal does not fire. The DOM does not update. Your code is correct in every way except the one that matters.

```
// These do NOT trigger updates:
items.value.push(newItem);
items.value.splice(0, 1);
items.value.sort();
items.value[0] = 'new';
user.value.name = 'Alice';
config.value.theme = 'dark';

// These DO trigger updates:
items.value = [...items.value, newItem];
items.value = items.value.filter((_, i) => i !== 0);
items.value = [...items.value].sort();
items.value = items.value.map((item, i) => i === 0 ? 'new' : item);
user.value = { ...user.value, name: 'Alice' };
config.value = { ...config.value, theme: 'dark' };
```

The pattern: create a new array or object. The spread operator is your primary tool. Every write to an array or object signal should produce a new reference. No exceptions.

If this feels wasteful, it is not. JavaScript engines optimize short-lived object allocations. The performance cost of spreading an array of 100 items is negligible compared to the debugging cost of a mutation that silently fails.

Helper Functions

If you find yourself writing the same spread patterns over and over, make helpers:

```
function pushSignal<T>(sig: Signal<T[]>, item: T) {
  sig.value = [...sig.value, item];
}

function removeSignal<T>(sig: Signal<T[]>, index: number) {
  sig.value = sig.value.filter((_, i) => i !== index);
}

function updateSignal<T extends object>(sig: Signal<T>, partial: Partial<T>) {
  sig.value = { ...sig.value, ...partial };
}
```

Three functions. They encode the new-reference rule so you do not have to think about it on every write.

---

3. Computed Is Eager, Not Lazy

In some frameworks, computed values are lazy -- they recalculate only when you read them. In tina4-js, computed values are eager. They recalculate the moment a dependency changes.

This means:

```
const expensive = computed(() => {
  // This runs EVERY TIME any dependency changes,
  // even if nobody is reading expensive.value
  return heavyCalculation(data.value);
});
```

If you have an expensive computation that is only needed in certain states:

```
// Instead of computed, use a signal + effect with a guard
const result = signal<Result | null>(null);
const needsResult = signal(false);

effect(() => {
  if (needsResult.value) {
    result.value = heavyCalculation(data.value);
  }
});
```

Or compute on demand without caching:

```
function getExpensiveResult() {
  return heavyCalculation(data.value);
}
```

For most computed values -- filtering lists, calculating totals, deriving display strings -- eagerness is fine. The recalculation takes microseconds. You only need to worry about this when the computation involves heavy iteration, network-dependent data, or large dataset transformations. If the computed function takes more than a millisecond, consider guarding it.

---

4. Inputs Must Stay Outside Reactive Blocks

> ***This is the most common tina4-js bug.*** *Wrapping* `<input>` *elements inside* `${() => ...}` *reactive blocks causes them to lose focus on every keystroke. The reactive block destroys and recreates its entire DOM subtree when any signal it reads changes — so the input element is replaced mid-typing.*

The rule: Form elements (`<input>`, `<textarea>`, `<select>`) go in the **static** part of the template. Use `.value`, `@input`, and `?disabled` bindings to make them reactive. Only wrap **computed output** (conditional messages, dropdown options, dynamic lists) in `${() => ...}`.

```
// WRONG — input inside reactive block, loses focus on every keystroke
html`${() => html`<input .value=${name} @input=${(e: Event) => {
  name.value = (e.target as HTMLInputElement).value;
}} />`}`

// RIGHT — input in static template, only output is reactive
html`
  <input .value=${name} @input=${(e: Event) => {
```

```
    name.value = (e.target as HTMLInputElement).value;
  }} />
  <p>${() => name.value ? `Hello, ${name.value}!` : 'Type your name'}</p>
`
```

---

5. Event Handler Auto-Batching

Since v1.0.9, all `@event` handlers in templates are wrapped in `batch()` for you. This means:

```
html`
  <button @click=${() => {
    a.value = 1;
    b.value = 2;
    c.value = 3;
  }}>Update</button>
`
// One DOM update, not three
```

You do NOT need to wrap event handlers in `batch()`. It happens behind the scenes.

But this applies only to handlers registered with the `@event` syntax in `html` templates. If you add event listeners through other means, you need explicit batching:

```
// Manual addEventListener -- NOT auto-batched
element.addEventListener('click', () => {
  batch(() => {
    a.value = 1;
    b.value = 2;
  });
});
```

The same applies to:

- `setTimeout` / `setInterval` callbacks
- `fetch().then()` handlers
- WebSocket `on('message')` handlers
- `Promise` handlers
- `requestAnimationFrame` callbacks

Any code that runs outside the `@event` handler context needs explicit `batch()` when writing to multiple signals. When in doubt, wrap in `batch()`. Batching an already-batched operation is harmless -- batches nest without side effects.

---

6. Effect Cleanup on Route Navigation

When the router navigates to a new route, it disposes all effects that were created during the previous route's handler execution. This is automatic and prevents memory leaks.

But it only disposes **effects**. It does not clean up:

- `setInterval` / `setTimeout` timers

- Manual `addEventListener` calls
- WebSocket connections
- Third-party library instances

For those, use a component with `onUnmount()`:

```
class LiveWidget extends Tina4Element {
  private interval = 0;
  private socket: ManagedSocket | null = null;

  onMount() {
    this.interval = window.setInterval(() => { /* ... */ }, 1000);
    this.socket = ws.connect('wss://...');
  }

  onUnmount() {
    clearInterval(this.interval);
    this.socket?.close();
  }

  render() { /* ... */ }
}
```

When the route changes and this component leaves the DOM, `onUnmount()` fires. The timer stops. The socket closes. No lingering connections. No phantom intervals ticking in the background.

The rule: if you create it in a component, destroy it in `onUnmount()`. If you create it in a route handler, put it in a component so `onUnmount()` can reach it.

---

7. Do Not Mix addEventListener Inside Reactive Blocks

```
// WRONG -- adds a new listener every time count changes
html`
  ${() => {
    document.addEventListener('keydown', handleKey); // LEAK!
    return html`<p>Count: ${count.value}</p>`;
  }}
`
```

Every time the reactive block re-runs, a new event listener is added. The old ones are never removed. After ten signal updates, ten listeners fire on every keypress. After a hundred updates, a hundred listeners fire.

Instead:

```
// RIGHT -- add the listener once, outside the template
document.addEventListener('keydown', handleKey);

html`<p>${count}</p>`
```

Or use a component:

```
class KeyHandler extends Tina4Element {
  private handler = (e: KeyboardEvent) => { /* ... */ };
```

```
  onMount() {
    document.addEventListener('keydown', this.handler);
  }

  onUnmount() {
    document.removeEventListener('keydown', this.handler);
  }

  render() { return html`<slot></slot>`; }
}
```

The component adds the listener once on mount and removes it on unmount. Clean lifecycle. No leaks. No accumulation.

---

8. The false/null Rendering Trap

This is covered in Chapter 3 but deserves repetition because it causes subtle bugs that pass code review:

```
// WRONG -- renders "false" as text when show is false
html`${show.value && html`<div>Content</div>`}`

// RIGHT -- returns null for nothing
html`${() => show.value ? html`<div>Content</div>` : null}`
```

The `&&` pattern comes from React JSX, where `false` is a rendering no-op. In tina4-js, `false` converts to the text `"false"` and displays on screen. The word "false" appears in your UI. Always use ternary. Always return `null` for "render nothing."

The same trap applies to zero:

```
// WRONG -- shows "0" when count is zero
html`${count.value && html`<p>${count.value} items</p>`}`

// RIGHT
html`${() => count.value > 0 ? html`<p>${count.value} items</p>` : null}`
```

`0 && anything` evaluates to `0`, which renders as the text `"0"`. A zero appears in your UI with no context. The ternary with an explicit comparison avoids both traps.

---

9. Signal Scope and Lifetime

Signals live until they are garbage collected. If nothing references a signal, the runtime cleans it up. But if an effect or DOM binding holds a reference, the signal stays alive.

Avoid Global Signals for Temporary State

```
// BAD -- these signals live forever
const formName = signal('');
const formEmail = signal('');

route('/users/new', () => {
  return html`<input .value=${formName} ... />`;
});
```

When the user navigates away and comes back, `formName` still holds the old value. The form is pre-filled with stale data. The user did not expect that.

```
// GOOD -- signals scoped to the route handler
route('/users/new', () => {
  const formName = signal('');
  const formEmail = signal('');

  return html`<input .value=${formName} ... />`;
});
```

Each visit creates fresh signals. The old ones are garbage collected when the route changes and effects are disposed. The form starts empty every time.

Use Global Signals for Shared State

```
// store.ts -- these SHOULD be global
export const user = signal<User | null>(null, 'user');
export const token = signal<string | null>(null, 'token');
export const theme = signal<'light' | 'dark'>('light', 'theme');
```

Global signals are appropriate for state that persists across route changes: authentication, user preferences, cart contents, notification counts. The distinction is simple: if the data should survive navigation, make it global. If the data belongs to a single page visit, scope it to the route handler.

---

10. Debugging Techniques

Add Labels to Every Signal

```
const count = signal(0, 'count');         // DO THIS
const count = signal(0);                    // NOT THIS
```

When you open the debug overlay and see 20 unnamed signals, you will spend more time figuring out which signal is which than fixing the bug you came to investigate.

Check Subscriber Counts

In the debug overlay, a signal with 0 subscribers is either:

- Unused (remove it)
- Bound incorrectly (`${count.value}` instead of `${count}`)

A signal with an unexpectedly high subscriber count might be referenced in a reactive block that re-runs often, creating new subscriptions each time. The subscriber count is the fastest way to spot a binding error or a subscription leak.

Use effect() for Debugging

```
effect(() => {
  console.log('User changed:', user.value);
  console.log('Token changed:', token.value);
});
```

This logs whenever `user` or `token` changes. It captures every write, from every source, in chronological order. Remove it before shipping -- but while debugging, it is more reliable than breakpoints because it shows you the data flow without pausing execution.

---

11. Performance Patterns

Keep Lists Reasonable

tina4-js re-renders entire lists when the signal changes. For lists under 200 items, the re-render takes under a millisecond. For larger lists:

- Paginate on the server (return 20 items per page, not 10,000)
- Use virtual scrolling (a separate library that renders only visible rows)
- Keep the visible list small with filtering and pagination

Avoid Unnecessary Computed

```
// UNNECESSARY -- computed with one dependency that transforms a value
const upperName = computed(() => name.value.toUpperCase());

// SIMPLER -- just use a function in the template
html`<p>${() => name.value.toUpperCase()}</p>`
```

Use computed when multiple parts of your app need the derived value. Use inline functions when only one template needs it. A computed signal adds a subscription, a cached value, and a node in the dependency graph. An inline function adds a function call. For single-use derivations, the function is the lighter tool.

Batch Async Results

```
// Multiple signal writes from an API response
const data = await api.get('/dashboard');
batch(() => {
  users.value = data.users;
  stats.value = data.stats;
  lastUpdated.value = new Date().toISOString();
});
```

Without batch, each assignment triggers a separate DOM update. Three writes, three re-renders. With batch, three writes, one re-render. The difference is invisible for two signals. It is visible for ten.

---

12. Common Error Messages

"[tina4] computed signals are read-only"

You tried to write to a computed signal. Computed values are derived -- they calculate from source signals. Write to the source signals instead, and the computed value updates on its own.

"[tina4] WebSocket is not connected"

You called `socket.send()` when the socket was not open. Check `socket.connected.value` before sending, or queue messages and flush them when the `status` signal changes to `'open'`.

"[tina4] Router target '...' not found in DOM"

The CSS selector you passed to `router.start({ target })` does not match any element. The element must exist in your HTML before `router.start()` runs. If you are using a component that renders the target element, make sure the component mounts before the router starts.

"[tina4] Prop '...' not declared in static props"

You called `this.prop('name')` but did not declare `name` in the component's `static props` object. The framework requires prop declarations so it can set up attribute observation. Add the prop to the static object and the error disappears.

Summary

Pitfall	Fix
Mutating arrays/objects in place	Always create new references (spread)
`${count.value}` in templates	Use `${count}` (pass the signal)
`${condition && html\`...\`}`	Use ternary: `${() => cond ? html\`...\` : null}`
addEventListener in reactive blocks	Add listeners once, outside templates
No cleanup for timers/sockets	Use `onUnmount()` in components
Expensive computed	Guard with a condition or compute on demand
Unnamed signals	Always add debug labels
Multiple writes without batch	Wrap async signal writes in `batch()`
Global signals for form state	Scope signals to route handlers or components

Vibe Coding with AI

Let AI Write Your tina4-js

You type a prompt: "Build a todo app." The AI generates 200 lines of code. You paste it in. The build fails. The state management is wrong. The router syntax is backwards. You spend 30 minutes fixing what the AI wrote. You could have written it from scratch in 20.

That is what happens with ambiguous frameworks. tina4-js is different. One state primitive. One template system. One router. One way to do everything. The AI has no wrong choices to make -- and the code it generates works on the first try.

---

1. Why tina4-js Works Well with AI

Most JavaScript frameworks are ambiguous. React offers three ways to manage state (useState, useReducer, external stores). Vue offers four ways to style components (scoped, modules, Tailwind, CSS-in-JS). An AI agent faces these choices and guesses. It often guesses wrong.

tina4-js has one way to do everything:

- One state primitive: `signal()`
- One template system: `html` tagged templates
- One component base: `Tina4Element`
- One router: `route()` + `router.start()`
- One API client: `api`
- One WebSocket client: `ws`

One choice. One answer. One correct output.

When an AI reads the tina4-js documentation, there is no ambiguity. No "it depends on your preference." No "you could also use X." The AI generates correct code on the first try because there is only one way to write it.

---

2. CLAUDE.md -- The AI's Instruction Manual

Every tina4-js project includes a `CLAUDE.md` file at the root. AI assistants (Claude Code, Cursor, GitHub Copilot) read this file when they open the project.

`CLAUDE.md` contains:

- **Build and test commands** -- `npm run build`, `npm run test`, `npm run dev`
- **Project structure** -- where signals, components, routes, and pages live
- **Key conventions** -- `route(pattern, handler)` pattern-first, `api.get(path, options)` with `{ params, headers }`

- **Template binding syntax** -- the complete table of `${signal}`, `@click`, `?disabled`, `.value`, etc.
- **Package exports** -- all seven entry points
- **Publishing instructions** -- version, build, publish

When an AI opens a tina4-js project and reads `CLAUDE.md`, it knows:

- How to build and test
- Where to put new code
- What syntax to use in templates
- What the API signatures look like
- What pitfalls to avoid

No guessing. No hallucinating method names. No inventing conventions. The instruction manual is in the repository, and the AI reads it before writing a single line.

---

3. llms.txt -- AI Context for the Framework

The tina4-js npm package ships an `llms.txt` file. This is a standardized format that AI tools consume to understand a library's API. It includes:

- Every exported function with its signature
- Every interface and type
- Usage examples for each module
- Common patterns

AI agents that support `llms.txt` load this as context when generating tina4-js code, even if they have never encountered the framework before. The file bridges the gap between "I have never seen this library" and "I know every function signature and every convention."

---

4. The tina4-js Skill

Beyond `CLAUDE.md` and `llms.txt`, tina4-js ships with an AI skill file at `.claude/skills/tina4-js/SKILL.md`. This is a deep reference that teaches AI agents the three rules that fix 90% of mistakes:

Rule 1: Static vs Reactive

```
// WRONG -- AI often generates this
html`<p>${count.value}</p>`

// RIGHT -- the skill teaches this
html`<p>${count}</p>`
```

AI agents love to be explicit. They write `count.value` because it shows they are reading the signal's current value. But in tina4-js templates, `.value` makes the binding static -- evaluated once, never updated. Passing the signal itself creates a reactive binding that updates when the signal changes. The skill file drills this distinction until the AI internalizes it.

Rule 2: New References

```
// WRONG -- AI loves Array.push()
items.value.push(newItem);

// RIGHT
items.value = [...items.value, newItem];
```

AI agents trained on React often generate mutation patterns because React's `useState` works with `setState(prev => [...prev, item])`. In tina4-js, the signal itself must receive a new reference. No setter function wraps the mutation. The signal compares references. The skill teaches the spread pattern for every array and object operation.

Rule 3: Boolean Attributes

```
// WRONG -- AI writes this because it looks logical
html`<button disabled=${isDisabled}>Click</button>`

// RIGHT
html`<button ?disabled=${isDisabled}>Click</button>`
```

Without the `?` prefix, `disabled` is set as a string attribute. `disabled="false"` still disables the button in HTML -- the attribute's presence is what matters, not its value. The `?` prefix tells tina4-js to add or remove the attribute based on the signal's truthiness. The skill file explains this with examples that prevent the mistake before it happens.

---

5. What AI Gets Wrong

Even with the skill file, AI agents make these mistakes. Watch for them in generated code:

Using .value in Templates

```
// AI generates:
html`<p>Count: ${count.value}</p>`
html`<button ?disabled=${loading.value}>Submit</button>`

// Should be:
html`<p>Count: ${count}</p>`
html`<button ?disabled=${loading}>Submit</button>`
```

When you see `.value` inside a template interpolation, it is almost always wrong. The signal itself should be passed for reactive binding. The `.value` version renders once and freezes.

The && Pattern

```
// AI generates (React habit):
html`${isLoading.value && html`<p>Loading...</p>`}`

// Should be:
html`${() => isLoading.value ? html`<p>Loading...</p>` : null}`
```

AI agents trained on JSX use `&&` for conditional rendering. In tina4-js, `false` renders as the text "false." The ternary with `null` is the correct pattern. Every conditional in a template should use `() =>` with a ternary.

Missing @event Syntax

```
// AI generates:
html`<button onclick=${handler}>Click</button>`

// Should be:
html`<button @click=${handler}>Click</button>`
```

The `@` prefix is tina4-js syntax for event binding. Native `onclick` as an attribute does not work with template interpolation. The AI sometimes falls back to vanilla HTML event attributes out of habit.

Forgetting Reactive Blocks for Conditionals

```
// AI generates:
html`
  <div>
    ${showDetails ? html`<p>Details</p>` : html`<p>Summary</p>`}
  </div>
`

// Should be:
html`
  <div>
    ${() => showDetails.value ? html`<p>Details</p>` : html`<p>Summary</p>`}
  </div>
`
```

The function wrapper `() =>` is required for the template to re-evaluate when `showDetails` changes. Without it, the condition evaluates once at render time and never updates. The page stays frozen on whichever branch was true at creation.

Using route() with Handler First

```
// AI sometimes generates:
route(() => html`<h1>Home</h1>`, '/');

// Should be:
route('/', () => html`<h1>Home</h1>`);
```

Pattern first. Handler second. Always.

---

6. Working with Claude Code on tina4-js Projects

Claude Code reads `CLAUDE.md` on project open. When you work on a tina4-js project in Claude Code, the AI knows the framework. Here is how to get the best results:

Be Specific About Modules

```
"Add a route at /users/{id} that fetches the user from /api/users/:id and displays their name an
```

This tells Claude which modules to use (route, api, html) and what the result should look like. Specific prompts produce specific code.

Reference Existing Patterns

```
"Add a new page similar to src/pages/users.ts but for products"
```

Claude reads the existing file and replicates the pattern -- same signal structure, same API calls, same template style. Your codebase stays consistent.

Let It Run the Dev Server

```
"Create a counter component and show me it works"
```

Claude creates the component, adds it to a route, and runs `npm run dev` to verify. The feedback loop is immediate.

Review Generated Templates

After Claude generates tina4-js code, scan for five things:

- `.value` inside `${}` in templates -- should it be the signal itself?
- `&&` patterns -- should it be a ternary with `null`?
- `disabled=` without `?` prefix -- should it be `?disabled=`?
- Missing `() =>` wrappers around conditional expressions
- Array mutations without new references

Five checks. Ten seconds. They catch the mistakes that would cost you ten minutes of debugging.

---

7. The AI Advantage

tina4-js was designed with AI-assisted development in mind. The framework's constraints are the AI's strengths:

- **One way to do things** means the AI never picks the wrong approach
- **Small API surface** means the AI does not hallucinate non-existent methods
- **Consistent conventions** mean generated code matches existing code
- `CLAUDE.md` **and skills** mean the AI has perfect context from the first prompt

Tell an AI: "Build a todo app with tina4-js." The AI needs to decide on state management, components, routing, styling, and API communication. With React, each decision has three to five options. The AI picks a combination. It might work. It might not. With tina4-js, there is one answer for each decision. The AI generates working code faster with fewer errors because fewer decisions means fewer wrong decisions.

This is not accidental. When every tina4 project follows the same structure, uses the same conventions, and ships the same instruction files, AI assistance becomes reliable. Not probabilistic. Not "usually right." Reliable.

---

8. Setting Up Your Project for AI

To get the best AI experience with tina4-js:

- **Keep `CLAUDE.md` up to date.** If you add custom conventions -- naming patterns, file organization rules, API endpoint structures -- document them. The AI reads what you write.
- **Use debug labels on signals.** AI agents read your code to understand patterns. `signal(0, 'cart-count')` communicates intent. `signal(0)` communicates nothing.
- **Follow the project structure.** `src/routes/`, `src/pages/`, `src/components/`, `src/store.ts`. When the AI sees consistent structure, it generates code that fits. When the structure is chaotic, the AI generates code that adds to the chaos.
- **Write TypeScript.** Type annotations give AI agents precise context about what functions expect and return. A typed interface is worth a paragraph of explanation.
- **Use the store pattern.** Export signals from `store.ts`. AI agents recognize this pattern and replicate it. Scattered signals in random files confuse AI agents the same way they confuse human developers.

One framework. One structure. One set of conventions. The AI becomes an extension of your thinking instead of a source of surprises.

Summary

What	Where
Project instructions	`CLAUDE.md` at project root
Framework API reference	`llms.txt` in the npm package
Deep AI skill	`.claude/skills/tina4-js/SKILL.md`
Common AI mistake 1	`${count.value}` instead of `${count}`
Common AI mistake 2	`&&` pattern instead of ternary with `null`
Common AI mistake 3	`disabled=` instead of `?disabled=`
Common AI mistake 4	Missing `() =>` for reactive conditionals
Common AI mistake 5	Array mutation instead of new reference