Learning LARC

LARC Community

2024

Contents

Learning LARC 1
Building Modern Web Applications with the Page Area Network Architecture 1
Praise for Learning LARC 1
Copyright 1
Table of Contents 2
Preface 2
Part I: Foundations 2
Part II: Building Components 2
Part III: Building Applications 2
Part IV: Advanced Topics 3
Part V: Ecosystem 3
Appendices 3
Index 3
About the Authors 4

Foreword 5

Chapter 1: Philosophy and Background 7
The Problem with Modern Web Development 7
The Rise of Complexity 7
The Platform Has Caught Up 8
A Common Scenario 8
A Return to Fundamentals 9
What LARC Is 10
What LARC Is Not 10
Core Principles 10
The LARC Philosophy 11
Composition Over Configuration 11
Convention Over Prescription 11
Explicit Over Implicit 12
Why “No Build” Matters 13
Development Speed 13
Debugging Simplicity 13
Deployment Simplicity 13
Lower Barrier to Entry 14
Sustainability 14

When to Use LARC . . . 14
LARC Excels At . . . 14
Consider Alternatives When . . . 15
Hybrid Approaches . . . 15
What You'll Build . . . 15
Chapter Examples . . . 15
Capstone Project: TaskFlow . . . 15
What You'll Learn . . . 16
Looking Ahead . . . 16
Summary . . . 16

Chapter 2: Core Concepts **19**
Web Components Refresher . . . 19
The Three Pillars . . . 19
Web Components vs Framework Components . . . 21
The Page Area Network (PAN) . . . 22
The Problem It Solves . . . 22
The PAN Bus Approach . . . 22
Topic Namespaces . . . 22
Message Patterns . . . 23
Why PAN Bus? . . . 24
Event-Driven Architecture . . . 24
Browser Events . . . 24
Custom Events . . . 25
PAN Bus Events . . . 25
When to Use Each . . . 26
State Management Philosophy . . . 26
State Hierarchy . . . 26
Reactive State (Optional) . . . 28
Module System . . . 29
Import/Export Basics . . . 29
Import Maps . . . 29
Module Organization . . . 30
The Component Lifecycle . . . 31
Lifecycle Callbacks . . . 31
Complete Lifecycle Example . . . 33
Summary . . . 36
Key Takeaways . . . 36

Chapter 3: Getting Started **37**
Setting Up Your Development Environment . . . 37
Requirements . . . 37
Quick Start with create-larc-app . . . 37
Manual Setup (No CLI) . . . 37
Development Tools . . . 39
Your First LARC Application . . . 40
Project Goal . . . 40
Step 1: Update index.html . . . 40

Step 2: Create app.js . . . 41
Step 3: Create counter-display.js . . . 42
Step 4: Create counter-controls.js . . . 43
Step 5: Test Your App . . . 45
What Just Happened? . . . 46
Project Structure . . . 46
File Organization Principles . . . 47
Import Maps Explained . . . 47
Basic Import Map . . . 47
Path Aliases . . . 48
Version Management . . . 48
Multiple CDNs . . . 49
Development vs Production . . . 49
Development Workflow . . . 50
Daily Development . . . 50
Debugging Tips . . . 50
Testing . . . 51
Common Patterns . . . 52
Pattern 1: Loading States . . . 52
Pattern 2: Form Handling . . . 53
Pattern 3: Conditional Rendering . . . 53
Pattern 4: Lists and Iteration . . . 54
Summary . . . 55
Exercises . . . 55

Chapter 4: Creating Web Components 57
Anatomy of a LARC Component . . . 57
Component Structure Breakdown . . . 64
Shadow DOM Deep Dive . . . 64
Creating Shadow DOM . . . 65
Shadow DOM vs Light DOM . . . 65
Style Encapsulation . . . 65
The :host Selector . . . 66
CSS Custom Properties (Variables) . . . 67
Parts and ::part() . . . 68
Attributes and Properties . . . 68
Attributes vs Properties . . . 68
Reflecting Properties to Attributes . . . 68
When to Use Each . . . 70
Type Conversion . . . 70
Boolean Attributes . . . 70
Component Styling . . . 71
Internal Styles . . . 71
External Stylesheets . . . 71
Adoptable Stylesheets . . . 71
Theming Strategies . . . 72
Lifecycle Methods (Advanced Patterns) . . . 74
Deferred Rendering . . . 74

Preventing Memory Leaks 74
Handling Rapid Reconnection 74
Testing Components 75
Unit Testing 75
Integration Testing 77
Visual Regression Testing 77
Summary 78
Best Practices 78

Chapter 5: The PAN Bus **81**
Understanding Pub/Sub Architecture 81
Traditional Communication 81
Pub/Sub Communication 81
The PAN Bus API 82
Topics and Namespaces 83
Topic Naming Conventions 83
Namespace Structure 83
Wildcards 84
Topic Best Practices 84
Publishing Messages 85
Basic Publishing 85
Publishing from Components 86
Event Metadata 87
Batch Publishing 87
Subscribing to Events 88
Basic Subscription 88
Component Lifecycle Integration 88
Multiple Subscriptions Helper 89
Conditional Subscriptions 90
Filtering Events 91
Message Patterns 91
1. Fire and Forget 91
2. Request/Response 91
3. Command Pattern 93
4. Event Sourcing 93
5. Aggregation Pattern 94
6. Saga Pattern 94
Debugging PAN Communication 96
Logging All Events 96
Event Inspector 96
Event Filtering 98
Performance Monitoring 98
Event Replay 99
Summary 101
Best Practices 101

Chapter 6: State Management **103**
Component-Local State 103

Instance Properties . . . 103
Private Fields . . . 104
State Objects . . . 105
Shared State Patterns . . . 106
Simple Global State . . . 107
Reactive State with Proxy . . . 108
Store Pattern . . . 109
The pan-store Component . . . 112
IndexedDB Integration . . . 113
Basic IndexedDB Wrapper . . . 113
Cache-First Strategy . . . 117
Persistence Strategies . . . 118
localStorage . . . 118
sessionStorage . . . 120
Hybrid Strategy . . . 120
Offline-First Applications . . . 122
Service Worker + State Management . . . 122
Sync Queue . . . 122
Summary . . . 126
Best Practices . . . 127

Chapter 7: Advanced Component Patterns **129**
Compound Components . . . 129
Basic Compound Component . . . 129
Context API for Compound Components . . . 132
Higher-Order Components . . . 134
Mixin Pattern . . . 134
Decorator Pattern . . . 137
Component Composition . . . 139
Container/Presentational Pattern . . . 139
Render Props Pattern . . . 140
Slots and Content Projection . . . 141
Named Slots . . . 141
Slot Change Detection . . . 142
Conditional Slots . . . 144
Dynamic Component Loading . . . 144
Lazy Loading . . . 144
Intersection Observer for Viewport Loading . . . 145
Performance Optimization . . . 146
Virtual Scrolling . . . 146
Memoization . . . 148
Debouncing and Throttling . . . 149
Summary . . . 151
Best Practices . . . 151

Chapter 8: Business Logic Patterns **153**
The Philosophy: Separation of Concerns . . . 153
Pattern 1: PAN Bus Listeners (Recommended) . . . 153

When to Use . . . 153
Basic Implementation . . . 154
Advantages . . . 158
Advanced: Composable Business Logic . . . 158
Pattern 2: Extending Components . . . 158
When to Use . . . 158
Implementation . . . 159
When This Makes Sense . . . 162
Pattern 3: Wrapper Components . . . 163
Implementation . . . 163
Pattern 4: Behavior Mixins . . . 165
Implementation . . . 165
Pattern 5: Service Layer . . . 168
Decision Matrix . . . 170
Real-World Example: E-Commerce Checkout . . . 170
Testing Business Logic . . . 175
Testing PAN Bus Listeners . . . 175
Testing Services . . . 176
Best Practices . . . 177
1. Keep Components Dumb . . . 177
2. Use Services for Complex Logic . . . 177
3. Make Business Logic Observable . . . 178
4. Document Business Rules . . . 178
5. Use Feature Flags . . . 179
Summary . . . 179

Chapter 8: Routing and Navigation 181
Client-Side Routing Basics . . . 181
The pan-router Component . . . 183
Route Parameters . . . 186
Route Guards . . . 186
Nested Routes . . . 187
Programmatic Navigation . . . 187
Query Parameters . . . 188
Summary . . . 189
Best Practices . . . 189

Chapter 9: Forms and Validation 191
Form Components . . . 191
Basic Form Component . . . 191
Two-Way Data Binding . . . 193
Validation Strategies . . . 195
Native HTML5 Validation . . . 195
Custom Validation . . . 195
Async Validation . . . 196
Error Handling . . . 197
File Uploads . . . 199
Form Submission . . . 201

Summary 203
Best Practices 204

Chapters 11-19: Summary Outlines **205**
Chapter 11: Data Fetching and APIs 205
Key Topics: 205
Code Example - API Client: 205
Chapter 12: Authentication and Security 206
Key Topics: 206
Code Example - Auth Service: 206
Chapter 13: Server Integration 207
Key Topics: 207
Node.js Example: 207
Chapter 14: Testing 207
Key Topics: 207
Test Example: 207
Chapter 15: Performance and Optimization 208
Key Topics: 208
Performance Patterns: 208
Chapter 16: Deployment 209
Key Topics: 209
Deployment Checklist: 209
Chapter 17: Component Library 209
Key Topics: 209
Registry Integration: 209
Chapter 18: Tooling 210
Key Topics: 210
VS Code Snippets: 210
Chapter 19: Real-World Applications 211
Case Study 1: E-Commerce Platform 211
Case Study 2: Dashboard Application 211
Case Study 3: Blog/CMS 211
Lessons Learned: 211

Appendices **213**
Appendix A: Web Components API Reference 213
Custom Elements 213
Lifecycle Callbacks 213
Shadow DOM 213
Appendix B: PAN Bus API Reference 213
Core Methods 213
Topic Patterns 214
Appendix C: Component API Reference 214
Built-in Components 214
Appendix D: Migration Guides 214
From React 214
From Vue 215
From Angular 215

Appendix E: Resources 215
Official Documentation 215
Web Standards 215
Community 216
Learning Resources 216
Tools 216
Example Projects 216

Learning LARC

Building Modern Web Applications with the Page Area Network Architecture

---

By the LARC Team

First Edition, 2025

Praise for Learning LARC

"LARC represents a return to web fundamentals while embracing modern capabilities. This book beautifully explains why that matters." — **Future Web Developer**

"Finally, a framework that respects the browser. Learning LARC shows you how to build without fighting the platform." — **Another Developer**

"The PAN bus architecture is elegant and powerful. This book makes it accessible to everyone." — **Yet Another Developer**

Copyright

Table of Contents

Preface

- Who This Book Is For
- What You'll Learn
- Conventions Used in This Book
- Using Code Examples
- How to Contact Us
- Acknowledgments

Part I: Foundations

Chapter 1: Philosophy and Background - The Problem with Modern Web Development - A Return to Fundamentals - The LARC Philosophy - Why "No Build" Matters - When to Use LARC - What You'll Build

Chapter 2: Core Concepts - Web Components Refresher - The Page Area Network (PAN) - Event-Driven Architecture - State Management Philosophy - Module System - The Component Lifecycle

Chapter 3: Getting Started - Setting Up Your Development Environment - Your First LARC Application - Project Structure - Import Maps Explained - Development Workflow - Common Patterns

Part II: Building Components

Chapter 4: Creating Web Components - Anatomy of a LARC Component - Shadow DOM Deep Dive - Attributes and Properties - Component Styling - Lifecycle Methods - Testing Components

Chapter 5: The PAN Bus - Understanding Pub/Sub Architecture - Topics and Namespaces - Publishing Messages - Subscribing to Events - Message Patterns - Debugging PAN Communication

Chapter 6: State Management - Component-Local State - Shared State Patterns - The pan-store Component - IndexedDB Integration - Persistence Strategies - Offline-First Applications

Chapter 7: Advanced Component Patterns - Compound Components - Higher-Order Components - Component Composition - Slots and Content Projection - Dynamic Component Loading - Performance Optimization

Part III: Building Applications

Chapter 8: Routing and Navigation - Client-Side Routing - The pan-router Component - Route Parameters - Nested Routes - Route Guards - History Management

Chapter 9: Forms and Validation - Form Components - Two-Way Data Binding - Validation Strategies - Error Handling - File Uploads - Form Submission

Chapter 10: Data Fetching and APIs - The pan-fetch Component - REST API Integration - GraphQL Support - WebSocket Communication - Server-Sent Events - Error Handling and Retry Logic

Chapter 11: Authentication and Security - Authentication Patterns - The pan-auth Component - JWT Token Management - Protected Routes - CORS Considerations - Security Best Practices

Part IV: Advanced Topics

Chapter 12: Server Integration - Backend Architecture - Node.js Integration - PHP Connector - Python/Django Integration - Database Patterns - Real-Time Communication

Chapter 13: Testing - Unit Testing Components - Integration Testing - End-to-End Testing - Visual Regression Testing - Performance Testing - Continuous Integration

Chapter 14: Performance and Optimization - Loading Strategies - Code Splitting - Lazy Loading Components - Caching Strategies - Bundle Size Optimization - Performance Monitoring

Chapter 15: Deployment - Static Hosting - CDN Configuration - Environment Variables - CI/CD Pipelines - Monitoring and Analytics - Production Best Practices

Part V: Ecosystem

Chapter 16: Component Library - Using the Component Registry - Contributing Components - Creating a Component Library - Documentation Strategies - Versioning and Releases

Chapter 17: Tooling - Development Tools - CLI Tools - VS Code Integration - Browser DevTools - Debugging Techniques

Chapter 18: Real-World Applications - Case Study: E-Commerce Platform - Case Study: Dashboard Application - Case Study: Blog/CMS - Lessons Learned - Best Practices

Appendices

Appendix A: Web Components API Reference - Custom Elements - Shadow DOM - HTML Templates - ES Modules

Appendix B: PAN Bus API Reference - Core Methods - Message Formats - Topic Patterns - Configuration Options

Appendix C: Component API Reference - Built-in Components - Component Properties - Events and Methods

Appendix D: Migration Guides - From React - From Vue - From Angular - From jQuery

Appendix E: Resources - Official Documentation - Community Resources - Video Tutorials - Example Projects

Index

About the Authors

The LARC Team is a group of developers passionate about web standards, simplicity, and performance. We believe the web platform has evolved to the point where many abstractions are no longer necessary, and we created LARC to prove it.

Foreword

by Christopher Robison

I didn't set out to build a framework. I set out to escape one — or at least escape the gravitational pull of the endless build pipeline.

After decades of building things for the web, my machine had become a storage exhibit of Node versions, Python versions, shims, wrappers, and dependency folders with the mass of small moons. Not because any of it was bad. Build tools are fine. For big projects, they're downright amazing. But somewhere along the way, we normalized the idea that even the simplest experiment needed a pipeline, a bundler, a transpiler, and a twelve-step hydration ritual before it could say "Hello, World."

That friction bothered me.

I wanted the feeling I had back in the early days: the joy of dropping a `<script>` tag into an HTML file and instantly seeing something come alive. No ceremony. No yak shaving. Just a browser, a file, and an idea.

Web Components felt close to that spirit — native modules, shadow DOM, real encapsulation — but they were oddly isolated. Each component was a self-contained island. Reusable, yes. Architecturally composable? Not really. Nothing tied them together except whatever glue code you wrote yourself. It felt like someone had shipped LEGO bricks without the ability to click them together.

That's when I remembered the CAN bus in cars.

The CAN bus is this beautifully simple ecosystem: dozens of systems — sensors, motors, controllers — all sharing a single communication line. Anybody can broadcast. Anybody can listen. Nobody needs to know who else exists. A message goes out, and the parts that care respond. It's loosely coupled machinery at its finest.

I wanted that for the web.

So I built the PAN bus — the Page Area Network — and started experimenting. Not with the intention of making A Real Framework™, but out of curiosity. How far could I push this idea? What could I build if every component on the page could talk over a shared bus, using nothing but browser-native APIs and one tiny script include?

That little experiment got out of hand in the best way.

I kept pushing it, partly out of stubbornness, partly out of sheer delight. I wanted to see if I could build real, full-blown applications with no build process at all — just a single script tag pointing to LARC and a page full of components chatting over the bus. And it turned out to be… fun. Refreshing. Capable. Liberating, even. A loose, elegant architecture emerged almost on its own.

Along the way, I realized something important: I'm not anti-build-tool. They solve real problems, especially at scale. But they shouldn't be mandatory for everything. And they shouldn't overshadow the fact that the browser today is powerful enough to build serious applications with a simple HTML page, a few components, and a shared message bus.

React, Angular, Vue — they solved problems that absolutely needed solving at the time. The web platform in 2015 was missing big pieces: templating, reactivity, routing, structured components, coherence. These frameworks carried the industry through that era. But the web has evolved since then. Many of those features now exist natively — standardized, built-in, fast, and universally available.

LARC isn't here to replace those frameworks. It complements them. It fills in the 20% Web Components never standardized — the shared communication fabric. The glue that lets components coexist instead of siloing themselves off. It also makes bundle sizes smaller and architectures cleaner, whether you're going framework-free or integrating with your existing stack.

If this book succeeds, you'll see what I saw: the thrill of rediscovering the browser as a first-class app platform. The joy of building big things out of small, decoupled pieces. And the surprising power of an architecture that starts with a simple HTML file and one script include.

The web grew up. Now we get to build like it.

— *Christopher Robison*

Chapter 1: Philosophy and Background

The Problem with Modern Web Development

If you've been building web applications for the past decade, you've likely experienced what many developers call "JavaScript fatigue." The modern web development landscape has become increasingly complex, with countless tools, frameworks, and build processes standing between you and shipping working code.

Consider a typical modern web project setup:

1. Initialize your project with a framework CLI (`create-react-app`, `vue create`, etc.)
2. Install hundreds or thousands of npm dependencies
3. Configure webpack, Babel, TypeScript, ESLint, Prettier
4. Set up build scripts for development, production, testing
5. Wait for builds to complete (sometimes minutes)
6. Debug build configuration issues when something breaks
7. Update dependencies regularly to patch security vulnerabilities
8. Repeat the cycle when frameworks release breaking changes

This complexity wasn't always necessary. In the early days of the web, you could create an HTML file, add some CSS and JavaScript, and open it directly in a browser. No build step. No toolchain. No configuration. Just code that runs.

What happened?

The Rise of Complexity

The web platform evolved, but it didn't evolve fast enough for ambitious developers. We wanted:

- **Component-based architecture** — but HTML didn't have custom elements yet
- **Module systems** — but JavaScript didn't have native imports
- **Reactive data binding** — but the DOM wasn't designed for it
- **Advanced syntax** — like JSX, TypeScript, or class properties

Frameworks filled these gaps by building abstractions on top of the web platform. But these abstractions came with costs:

- **Build toolchains** became mandatory to transpile code
- **Bundle sizes** grew as framework code was shipped to browsers

- **Learning curves** steepened as developers had to learn both the framework and the tools
- **Debugging** became harder with source maps and transpiled code
- **Performance** suffered from unnecessary abstraction layers

The irony? While we were busy building these elaborate toolchains, the web platform itself was evolving to support many of the features we wanted natively.

The Platform Has Caught Up

Today's web platform is remarkably capable. Modern browsers support:

- **Custom Elements** — native component definition
- **Shadow DOM** — true style encapsulation
- **ES Modules** — native JavaScript modules with imports
- **Import Maps** — dependency management without bundlers
- **Template Literals** — dynamic HTML without JSX
- **Proxy and Reflect** — reactive data patterns
- **CSS Custom Properties** — themeable components
- **Web Components** — standards-based component architecture

These aren't polyfills or experimental features. They're stable, well-supported standards that work across all modern browsers. Yet most web frameworks continue to build elaborate abstractions on top of the platform, ignoring these native capabilities.

A Common Scenario

Let's look at a real-world example. Imagine you're building a simple dashboard with a few interactive components: a card, a button, and a data table. Here's what this might look like in a typical React project:

The Setup:

```
npx create-react-app my-dashboard
cd my-dashboard
npm install styled-components react-router axios redux
# Wait 5-10 minutes for installation
# Project size: ~300MB, ~1000+ dependencies
```

The Code:

```
// Card.jsx
import React from 'react';
import styled from 'styled-components';

const StyledCard = styled.div`
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
`;

export default function Card({ title, children }) {
  return (
    <StyledCard>
      <h2>{title}</h2>
      {children}
    </StyledCard>
  );
}
```

The Build:

```
npm run build
# Wait 30-60 seconds
# Output: Minified, bundled, transpiled code
# Bundle size: 200-500KB (before your actual code)
```

Now, here's the same thing with native Web Components and LARC:

The Setup:

```
<!DOCTYPE html>
<html>
<head>
  <script type="importmap">
  {
    "imports": {
      "@larcjs/ui": "https://cdn.jsdelivr.net/npm/@larcjs/ui@1/dist/index.js"
    }
  }
  </script>
</head>
<body>
  <pan-card title="Dashboard">
    <p>Your content here</p>
  </pan-card>

  <script type="module">
    import '@larcjs/ui';
  </script>
</body>
</html>
```

The Build:

```
# There is no build step. Open the HTML file. It works.
```

Same functionality. Zero dependencies. No build process. No toolchain. Just HTML, CSS, and JavaScript working together as the platform intended.

A Return to Fundamentals

LARC (Lightweight Autonomous Reactive Components) represents a philosophical shift back to web fundamentals. But this isn't about going backward—it's about recognizing that the platform has evolved to the point where many of our abstractions are no longer necessary.

What LARC Is

LARC is a set of conventions, patterns, and utilities for building modern web applications using native web standards:

- **Web Components** for encapsulated, reusable UI elements
- **ES Modules** for code organization and imports
- **Import Maps** for dependency management
- **The PAN Bus** for component communication
- **Native APIs** for state, routing, and data fetching

LARC provides guidance and utilities, but it doesn't abstract away the platform. When you write LARC code, you're writing standard JavaScript, HTML, and CSS that runs directly in the browser.

What LARC Is Not

LARC is deliberately minimal. It is **not**:

- A framework with proprietary APIs you must learn
- A template language that requires compilation
- A state management system with complex rules
- A build tool that transforms your code
- A runtime that interprets your components

If you know HTML, CSS, and JavaScript, you already know most of LARC.

Core Principles

LARC is built on several core principles:

1. Standards First

LARC embraces web standards rather than fighting them. Every LARC component is a valid Web Component. Every LARC module is a valid ES Module. If you understand the standards, you understand LARC.

2. Zero Build for Development

During development, you should be able to edit a file and refresh the browser. No build step. No waiting. No configuration. The browser is your development environment.

This doesn't mean builds are forbidden—you can still optimize for production if needed. But they should be optional enhancements, not requirements.

3. Progressive Enhancement

Start simple and add complexity only when needed. A basic component can be a few lines of JavaScript. As requirements grow, add features incrementally: state management, routing, server integration, etc.

You're never locked into architectural decisions made at project initialization. LARC applications evolve naturally.

4. Local First, Network Aware

Components should work independently with local state. Network communication happens through explicit, observable patterns (the PAN bus). This makes components:

- Easier to test (no mocking required)
- More reusable (fewer dependencies)
- More resilient (graceful degradation)

5. Developer Experience Through Simplicity

Good DX doesn't require complex tooling. It comes from:

- Clear, predictable patterns
- Minimal abstractions
- Fast feedback loops
- Easy debugging
- Comprehensive documentation

When something breaks in LARC, you can open browser DevTools and debug standard JavaScript. No source maps. No transpiled code. No framework internals.

The LARC Philosophy

At its heart, LARC is about **respecting the platform**. The web is incredibly powerful, yet we've spent years building layers of abstraction that hide its capabilities. LARC removes those layers.

Composition Over Configuration

Rather than configuring a framework through JSON or CLI flags, LARC applications are composed from standard parts:

```html
<!-- Composition: Combine standard elements -->
<pan-router>
  <pan-route path="/" component="home-page"></pan-route>
  <pan-route path="/dashboard" component="dashboard-page"></pan-route>
</pan-router>
```

Each element is understandable in isolation. There's no magic configuration file that controls behavior across your entire application.

Convention Over Prescription

LARC suggests patterns but doesn't enforce them. There's no "one true way" to structure a LARC application. The conventions exist to make common tasks easier, but you can always drop down to standard APIs when needed.

For example, LARC recommends the PAN bus for component communication, but you can also use:

- Custom events
- Direct property access
- Shared state objects
- URL parameters
- LocalStorage
- Any other standard browser API

Choose the right tool for your specific use case.

Explicit Over Implicit

LARC favors explicitness. When a component fetches data, you see the fetch call. When state changes, you see the assignment. When events are dispatched, you see the dispatch.

Compare these two approaches:

Implicit (typical framework):

```
function UserProfile() {
  const [user, loading, error] = useUser(userId);

  if (loading) return <Spinner />;
  if (error) return <Error message={error} />;

  return <ProfileCard user={user} />;
}
```

Magic happens in `useUser`. Where does the data come from? When does it refetch? What triggers updates? You need to understand the framework's mental model.

Explicit (LARC):

```
class UserProfile extends HTMLElement {
  async connectedCallback() {
    this.render({ loading: true });

    try {
      const response = await fetch(`/api/users/${this.userId}`);
      const user = await response.json();
      this.render({ user });
    } catch (error) {
      this.render({ error: error.message });
    }
  }

  render(state) {
    if (state.loading) {
      this.innerHTML = '<loading-spinner></loading-spinner>';
    } else if (state.error) {
      this.innerHTML = `<error-message text="${state.error}"></error-message>`;
    } else {
      this.innerHTML = `<profile-card .user="${state.user}"></profile-card>`;
    }
  }
}
```

Every step is visible. You can trace exactly what happens and when. Debugging is straightforward because you’re working with standard JavaScript.

Why “No Build” Matters

The “no build” philosophy isn’t about being purist or rejecting tools. It’s about removing unnecessary complexity and its associated costs.

Development Speed

Without a build step, your development cycle is:

1. Edit code
2. Refresh browser
3. See changes

That’s it. No waiting for webpack to rebuild. No watching file watchers fail. No debugging build configurations.

This might seem like a small thing, but it compounds. Over a day of development, those 10-30 second build times add up to significant lost productivity. More importantly, they break flow state.

Debugging Simplicity

When you open browser DevTools in a LARC application, you see your actual code. No source maps needed. No transpiled output. No minified framework internals.

Set a breakpoint in your component’s `connectedCallback`. It stops exactly where you expect. The call stack is readable. Variables are named as you wrote them.

This makes debugging accessible to junior developers and reduces time spent fighting tools.

Deployment Simplicity

A LARC application can be deployed to any static host:

- GitHub Pages
- Netlify
- Vercel
- Amazon S3
- Any web server

No server-side rendering. No Node.js runtime. No build artifacts to manage. Just upload HTML, CSS, and JavaScript files.

Want to deploy to a CDN? Your entire application is already CDN-friendly because it’s just static files.

Lower Barrier to Entry

New developers can learn web development by:

1. Creating an HTML file
2. Adding some CSS and JavaScript
3. Opening it in a browser

No installation. No environment setup. No project configuration. This is how the web should work.

With build tools, new developers face:

1. Install Node.js
2. Learn npm/yarn
3. Understand package.json
4. Configure webpack/Babel
5. Troubleshoot build errors
6. Learn framework-specific tooling

Before writing a single line of application code, they've already encountered dozens of concepts unrelated to actual web development.

Sustainability

Build tools and frameworks change rapidly. A React application from 2015 likely needs significant updates to run today. Build configurations break. Dependencies become unmaintained. Migration guides are incomplete.

LARC applications use web standards. A LARC application from 2025 will still run in 2035 because it's built on stable browser APIs, not framework-specific abstractions.

This doesn't mean LARC applications never need updates—APIs evolve, best practices change. But the core architecture is built on a foundation that changes slowly and deliberately through standards processes.

When to Use LARC

LARC isn't the right choice for every project. Understanding when to use it (and when not to) helps you make informed decisions.

LARC Excels At

Small to Medium Applications Projects with 10-100 components where simplicity and maintainability matter more than framework ecosystem size.

Dashboard and Admin Panels Internal tools where the development team controls the environment and values fast iteration.

Progressive Web Apps Applications that benefit from offline-first architecture and minimal JavaScript overhead.

Learning Projects Teaching web development without the complexity of modern toolchains.

Embedded Widgets Reusable components that need to work in any environment without framework dependencies.

Prototypes and MVPs Quickly validating ideas without upfront tooling investment.

Consider Alternatives When

Very Large Teams If you have 50+ developers working on a single codebase, framework opinions and tooling might provide valuable guardrails.

Heavy Framework Ecosystem Dependencies If your project critically relies on a specific framework's ecosystem (e.g., React Native integration, specific UI libraries), switching costs may be prohibitive.

Server-Side Rendering is Critical While LARC supports SSR, frameworks like Next.js have more mature SSR/SSG ecosystems.

Team Expertise If your entire team is deeply experienced in React/Vue/Angular and inexperienced with Web Components, the learning curve might slow initial development.

That said, LARC's simplicity often means the learning curve is shorter than expected. Most experienced developers can become productive with LARC in days, not weeks.

Hybrid Approaches

You don't have to go all-in on LARC. Consider hybrid approaches:

Progressive Migration Build new features in LARC while maintaining existing framework code. Web Components can coexist with React, Vue, or Angular.

Micro-frontends Use LARC for some micro-frontends and other frameworks for others. Web Components provide clean boundaries.

Component Libraries Build a LARC component library that can be consumed by any framework or vanilla JavaScript.

What You'll Build

Throughout this book, you'll build several progressively complex applications:

Chapter Examples

Each chapter includes focused examples demonstrating specific concepts:

- A **counter component** (Chapter 4) to understand component basics
- A **todo list** (Chapter 5) to learn PAN bus communication
- A **user profile form** (Chapter 9) to master form handling
- A **data table** (Chapter 10) to work with APIs and data

Capstone Project: TaskFlow

In the final chapters, you'll build **TaskFlow**, a complete project management application featuring:

- User authentication and authorization

- Real-time collaboration via WebSockets
- Offline-first architecture with IndexedDB
- Drag-and-drop task boards
- File attachments and comments
- Search and filtering
- Data visualization
- Mobile-responsive design

TaskFlow will demonstrate how LARC patterns scale to production applications while remaining maintainable and performant.

What You'll Learn

By the end of this book, you'll be able to:

- Build complex, maintainable applications using Web Components
- Design effective component communication patterns with the PAN bus
- Manage application state without external frameworks
- Integrate with backend APIs and real-time services
- Handle routing, forms, and authentication
- Write testable, reusable components
- Optimize performance and bundle size
- Deploy LARC applications to production
- Make informed decisions about when to use LARC vs. other approaches

Looking Ahead

The next chapter dives into LARC's core concepts: Web Components, the PAN bus, and event-driven architecture. You'll learn the fundamental patterns that make LARC applications work.

But before we get technical, take a moment to consider what drew you to this book. Perhaps you're tired of build tool complexity. Perhaps you want to understand how the web really works. Perhaps you're curious about a different approach.

Whatever your motivation, LARC offers something increasingly rare in modern web development: simplicity without sacrificing capability. You're about to learn how to build serious web applications using the platform itself, not abstractions on top of it.

Let's begin.

Summary

- Modern web development has become unnecessarily complex with build tools, frameworks, and abstractions
- The web platform has evolved to support features natively that once required frameworks
- LARC uses web standards (Web Components, ES Modules, Import Maps) to build applications without build steps

- Core principles: standards first, zero build for development, progressive enhancement, local first
- “No build” matters for development speed, debugging simplicity, deployment, and sustainability
- LARC works best for small-to-medium applications, dashboards, PWAs, and prototypes
- You’ll build real applications throughout this book, culminating in a production-ready project management app

Chapter 2: Core Concepts

Now that you understand LARC's philosophy, let's explore the technical foundation that makes it work. This chapter introduces the core concepts you'll use throughout the book: Web Components, the PAN bus, event-driven architecture, and the component lifecycle.

Don't worry if some of these concepts are new to you. We'll build understanding progressively, starting with the basics and working toward more sophisticated patterns.

Web Components Refresher

Web Components are a suite of browser APIs that let you create custom, reusable HTML elements. Unlike framework components, Web Components are browser standards supported natively across all modern browsers.

The Three Pillars

Web Components rest on three main technologies:

1. Custom Elements

Custom Elements let you define new HTML tags with custom behavior:

```
// Define a custom element
class HelloWorld extends HTMLElement {
  connectedCallback() {
    this.textContent = 'Hello, World!';
  }
}

// Register it
customElements.define('hello-world', HelloWorld);
```

Now you can use `<hello-world></hello-world>` in your HTML, and it works like any built-in element.

Key Points: - Element names must contain a hyphen (e.g., `my-component`, not `mycomponent`) - Custom elements inherit from `HTMLElement` or another HTML element - They have lifecycle callbacks for creation, connection, and removal

2. Shadow DOM

Shadow DOM provides style and markup encapsulation:

```
class FancyButton extends HTMLElement {
  constructor() {
    super();
    // Create shadow root
    this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    this.shadowRoot.innerHTML = `
      <style>
        button {
          background: blue;
          color: white;
          border: none;
          padding: 10px 20px;
          border-radius: 4px;
        }
      </style>
      <button>
        <slot></slot>
      </button>
    `;
  }
}

customElements.define('fancy-button', FancyButton);
```

The styles inside Shadow DOM don't leak out, and external styles don't leak in:

```
<!-- This button is blue (from shadow DOM) -->
<fancy-button>Click Me</fancy-button>

<!-- This button is not affected by fancy-button's styles -->
<button>Regular Button</button>

<style>
  /* This won't affect fancy-button's internal button */
  button { background: red; }
</style>
```

Key Points: - Shadow DOM creates an isolated scope for styles and DOM - Use `<slot>` elements to project content from light DOM into shadow DOM - `mode: 'open'` makes shadow root accessible via `element.shadowRoot`

3. HTML Templates

Templates define reusable chunks of markup that aren't rendered until activated:

```
<template id="card-template">
  <style>
    .card {
      border: 1px solid #ddd;
      border-radius: 8px;
      padding: 16px;
    }
  </style>
  <div class="card">
    <h2 class="title"></h2>
    <p class="content"></p>
  </div>
</template>

<script>
  class SimpleCard extends HTMLElement {
    connectedCallback() {
      const template = document.getElementById('card-template');
      const clone = template.content.cloneNode(true);

      clone.querySelector('.title').textContent = this.getAttribute('title');
      clone.querySelector('.content').textContent = this.getAttribute('content');

      this.attachShadow({ mode: 'open' });
      this.shadowRoot.appendChild(clone);
    }
  }

  customElements.define('simple-card', SimpleCard);
</script>
```

Key Points: - Template content is inert (scripts don't run, images don't load) - Templates can be defined in HTML or created programmatically - Clone template content before using it

Web Components vs Framework Components

It's worth understanding how Web Components differ from framework components:

Aspect	Web Components	React Components
Definition	Browser standard	Library-specific
Syntax	JavaScript classes	JSX or functions
Lifecycle	Native callbacks	Virtual DOM lifecycle
Reusability	Works everywhere	Requires React
Build step	Optional	Required (for JSX)
Encapsulation	Shadow DOM	CSS Modules/CSS-in-JS

Both approaches have their place. Web Components excel at true reusability and standards-based development. Framework components often provide better ergonomics within their specific ecosystem.

LARC chooses Web Components because they align with the "standards first" principle.

The Page Area Network (PAN)

The Page Area Network, or PAN bus, is LARC's event-driven communication system. It's inspired by microservices architecture but designed for browser components.

The Problem It Solves

In a traditional component tree, communication flows up and down:

```
App
  Header
      UserMenu
          LogoutButton
  Content
      UserProfile
```

If `LogoutButton` needs to notify `UserProfile` that the user logged out, you have several options:

1. **Pass callbacks down** through props (prop drilling)
2. **Lift state up** to a common ancestor
3. **Use context** or global state
4. **Dispatch custom events** that bubble up

Each approach has tradeoffs. Prop drilling creates tight coupling. Global state makes testing harder. Event bubbling is limited by DOM structure.

The PAN Bus Approach

The PAN bus provides a **decoupled pub/sub system**:

```
// LogoutButton publishes an event
pan.publish('user.logout', { userId: 123 });

// UserProfile subscribes to events (anywhere in the app)
pan.subscribe('user.logout', (data) => {
  console.log('User logged out:', data.userId);
  this.clearUserData();
});
```

Components don't need to know about each other. They communicate through topics (like `'user.logout'`) with no direct coupling.

Topic Namespaces

Topics use dot notation for organization:

```
'user.login'          // User logged in
'user.logout'         // User logged out
'user.profile.update' // Profile was updated

'cart.item.add'       // Item added to cart
'cart.item.remove'    // Item removed
'cart.checkout'       // Checkout initiated

'app.theme.change'    // Theme changed
'app.error'           // Application error
```

You can subscribe to specific topics or use wildcards:

```
// Specific topic
pan.subscribe('user.login', handler);

// Wildcard (all user events)
pan.subscribe('user.*', handler);

// All events (useful for debugging)
pan.subscribe('*', handler);
```

Message Patterns

The PAN bus supports several messaging patterns:

1. Fire and Forget

Most common pattern. Publish a message and continue:

```
pan.publish('notification.show', {
  type: 'success',
  message: 'Saved successfully'
});
```

2. Request/Response

Publish a message and wait for a response:

```
const result = await pan.request('api.fetch', {
  url: '/api/users',
  method: 'GET'
});
```

A subscriber handles the request and returns data:

```
pan.respond('api.fetch', async (data) => {
  const response = await fetch(data.url, { method: data.method });
  return response.json();
});
```

3. State Broadcast

Publish state changes that multiple components need:

```
// Theme switcher publishes
pan.publish('app.theme.change', { theme: 'dark' });

// Multiple components subscribe
class Header extends HTMLElement {
  connectedCallback() {
    pan.subscribe('app.theme.change', ({ theme }) => {
      this.applyTheme(theme);
    });
  }
}

class Sidebar extends HTMLElement {
  connectedCallback() {
    pan.subscribe('app.theme.change', ({ theme }) => {
      this.applyTheme(theme);
    });
  }
}
```

Why PAN Bus?

The PAN bus provides several advantages:

Loose Coupling Components don't need references to each other. Add or remove components without changing others.

Testability Test components in isolation. Mock the bus or test actual pub/sub behavior.

Debuggability Subscribe to `'*'` to log all messages. Visualize message flow easily.

Scalability Add new features by subscribing to existing topics. No need to modify existing code.

Flexibility Mix different communication patterns (events, requests, broadcasts) as needed.

Event-Driven Architecture

LARC applications use event-driven architecture (EDA) at multiple levels:

Browser Events

Standard DOM events for user interaction:

```
class ClickCounter extends HTMLElement {
  constructor() {
    super();
    this.count = 0;
  }
  connectedCallback() {
    this.innerHTML = `
      <button id="btn">Clicked ${this.count} times</button>
    `;

    this.querySelector('#btn').addEventListener('click', () => {
      this.count++;
      this.querySelector('#btn').textContent = `Clicked ${this.count} times`;
    });
  }
}
```

Custom Events

Components can dispatch custom events for parent components:

```
class ColorPicker extends HTMLElement {
  selectColor(color) {
    // Dispatch custom event
    this.dispatchEvent(new CustomEvent('colorchange', {
      detail: { color },
      bubbles: true,
      composed: true  // Cross shadow DOM boundary
    }));
  }
}

// Parent can listen
document.querySelector('color-picker').addEventListener('colorchange', (e) => {
  console.log('Selected color:', e.detail.color);
});
```

PAN Bus Events

For cross-component communication:

```
class SearchBox extends HTMLElement {
  handleInput(value) {
    pan.publish('search.query', { query: value });
  }
}

class SearchResults extends HTMLElement {
  connectedCallback() {
    pan.subscribe('search.query', ({ query }) => {
      this.search(query);
    });
  }
}
```

When to Use Each

Use DOM Events when: - Handling user interactions (click, input, focus, etc.) - Communication is parent-child relationship - Following HTML semantics matters

Use Custom Events when: - Component needs to notify parent/ancestors - Event should bubble up the DOM tree - Mimicking native element behavior

Use PAN Bus when: - Components are not in parent-child relationship - Multiple unrelated components need the same data - Decoupling is more important than DOM semantics - Building cross-cutting concerns (logging, analytics, etc.)

State Management Philosophy

LARC takes a pragmatic approach to state management: use the simplest solution that works, then scale up if needed.

State Hierarchy

State can exist at different levels:

1. Component-Local State

State that only matters to one component:

```
class TodoItem extends HTMLElement {
  constructor() {
    super();
    this.completed = false;  // Local state
  }

  toggle() {
    this.completed = !this.completed;
    this.render();
  }

  render() {
    this.classList.toggle('completed', this.completed);
  }
}
```

When to use: UI state, temporary values, component-specific configuration.

2. Shared State

State that multiple components need:

```
// Simple shared state object
const appState = {
  user: null,
  theme: 'light',
  notifications: []
};

// Components read from it
class UserMenu extends HTMLElement {
  connectedCallback() {
    this.render(appState.user);
  }
}

// Components write to it and notify via PAN
function updateTheme(theme) {
  appState.theme = theme;
  pan.publish('app.theme.change', { theme });
}
```

When to use: Application-wide settings, user data, feature flags.

3. Persistent State

State that survives page reloads:

```
class TodoList extends HTMLElement {
  loadTodos() {
    const saved = localStorage.getItem('todos');
    return saved ? JSON.parse(saved) : [];
  }

  saveTodos(todos) {
    localStorage.setItem('todos', JSON.stringify(todos));
  }
}
```

When to use: User preferences, draft content, offline data.

4. Server State

State that comes from and syncs with a server:

```
class UserProfile extends HTMLElement {
  async loadProfile() {
    const response = await fetch('/api/profile');
    this.profile = await response.json();
    this.render();
  }

  async saveProfile(updates) {
    await fetch('/api/profile', {
      method: 'PUT',
      body: JSON.stringify(updates)
    });
  }
}
```

When to use: Database records, API data, real-time updates.

Reactive State (Optional)

For more complex state needs, LARC provides reactive patterns using JavaScript Proxies:

```
function createStore(initialState) {
  const listeners = new Set();

  const state = new Proxy(initialState, {
    set(target, property, value) {
      target[property] = value;
      listeners.forEach(fn => fn(property, value));
      return true;
    }
  });

  return {
    state,
    subscribe(fn) {
      listeners.add(fn);
      return () => listeners.delete(fn);
    }
  };
}

// Usage
const store = createStore({ count: 0 });

class Counter extends HTMLElement {
  connectedCallback() {
    // Subscribe to changes
    this.unsubscribe = store.subscribe((prop, value) => {
      if (prop === 'count') this.render();
    });

    this.render();
  }

  disconnectedCallback() {
    this.unsubscribe();
  }
  render() {
    this.textContent = `Count: ${store.state.count}`;
  }
}

// Update state (automatically notifies subscribers)
store.state.count++;
```

This is similar to MobX or Vue's reactivity, but built with standard JavaScript.

Module System

LARC uses ES Modules, the native JavaScript module system.

Import/Export Basics

Export from a module:

```
// components/button.js
export class PanButton extends HTMLElement {
  // ...
}

export const BUTTON_TYPES = ['primary', 'secondary', 'danger'];

export default PanButton;
```

Import into another module:

```
// app.js
import PanButton, { BUTTON_TYPES } from './components/button.js';

// Or import everything
import * as Button from './components/button.js';
```

Import Maps

Import Maps let you define aliases for module paths:

```
<script type="importmap">
{
  "imports": {
    "@larcjs/core": "https://cdn.jsdelivr.net/npm/@larcjs/core@1/dist/index.js",
    "@larcjs/ui": "https://cdn.jsdelivr.net/npm/@larcjs/ui@1/dist/index.js",
    "app/": "/src/",
    "components/": "/src/components/"
  }
}
</script>
```

```html
<script type="module">
  // Use aliases
  import { pan } from '@larcjs/core';
  import { PanButton } from '@larcjs/ui';
  import { Header } from 'components/header.js';
</script>
```

This is similar to webpack's resolve aliases, but it's a browser standard.

Module Organization

A typical LARC project structure:

```
src/
   components/
       header.js
       footer.js
       sidebar.js
   lib/
       api.js
       auth.js
       utils.js
   pages/
       home.js
       dashboard.js
       profile.js
   app.js
```

Each file is a module with clear responsibilities:

```javascript
// src/lib/api.js
export async function fetchJSON(url, options = {}) {
  const response = await fetch(url, {
    ...options,
    headers: {
      'Content-Type': 'application/json',
      ...options.headers
    }
  });

  if (!response.ok) {
    throw new Error(`API error: ${response.status}`);
  }

  return response.json();
}

// src/components/user-list.js
import { fetchJSON } from '../lib/api.js';
```

```
export class UserList extends HTMLElement {
  async connectedCallback() {
    const users = await fetchJSON('/api/users');
    this.render(users);
  }
}

customElements.define('user-list', UserList);
```

The Component Lifecycle

Understanding the component lifecycle is essential for building robust LARC applications.

Lifecycle Callbacks

Web Components provide several lifecycle callbacks:

constructor()

Called when an instance is created:

```
class MyComponent extends HTMLElement {
  constructor() {
    // MUST call super() first
    super();

    // Initialize instance properties
    this.count = 0;
    this.data = null;

    // Attach shadow DOM if needed
    this.attachShadow({ mode: 'open' });

    // DON'T access attributes or children here
    // They might not be set yet
  }
}
```

Best practices: - Always call `super()` first - Initialize instance properties - Attach shadow DOM - Don't access attributes, children, or parent elements - Don't render here (use `connectedCallback` instead)

connectedCallback()

Called when the element is inserted into the DOM:

```
connectedCallback() {
  // Now it's safe to access attributes, children, parent
  const title = this.getAttribute('title');
  // Render initial content
  this.render();

  // Add event listeners
  this.addEventListener('click', this.handleClick);

  // Fetch data
  this.loadData();

  // Subscribe to PAN events
  this.unsubscribe = pan.subscribe('data.update', this.handleUpdate);
}
```

Best practices: - Render initial content - Add event listeners - Subscribe to events - Fetch initial data - Can be called multiple times if element is moved

disconnectedCallback()

Called when the element is removed from the DOM:

```
disconnectedCallback() {
  // Clean up event listeners
  this.removeEventListener('click', this.handleClick);

  // Unsubscribe from PAN events
  if (this.unsubscribe) {
    this.unsubscribe();
  }

  // Cancel pending operations
  if (this.fetchController) {
    this.fetchController.abort();
  }

  // Clear timers
  if (this.timer) {
    clearInterval(this.timer);
  }
}
```

Best practices: - Remove event listeners to prevent memory leaks - Unsubscribe from PAN events - Cancel pending async operations - Clear timers and intervals

attributeChangedCallback(name, oldValue, newValue)

Called when observed attributes change:

```
static get observedAttributes() {
  return ['title', 'count', 'active'];
}
```

```
attributeChangedCallback(name, oldValue, newValue) {
  // Called for each observed attribute that changes
  if (name === 'title') {
    this.updateTitle(newValue);
  } else if (name === 'count') {
    this.updateCount(Number(newValue));
  } else if (name === 'active') {
    this.updateActive(newValue !== null);
  }
}
```

Best practices: - Only observe attributes you actually use - Convert string values to appropriate types - Handle null/undefined values - Update only what changed (don't re-render everything)

adoptedCallback()

Called when the element is moved to a new document (rare):

```
adoptedCallback() {
  // Usually not needed
  // Called when element is moved between documents
  // (e.g., iframe scenarios)
}
```

Complete Lifecycle Example

Here's a full component showing proper lifecycle management:

```
class DataTable extends HTMLElement {
  // Define which attributes to observe
  static get observedAttributes() {
    return ['url', 'page-size'];
  }

  constructor() {
    super();
    this.attachShadow({ mode: 'open' });

    // Initialize state
    this.data = [];
    this.pageSize = 10;
    this.currentPage = 1;
  }

  async connectedCallback() {
    // Initial render
    this.render();

    // Load data if URL is set
    const url = this.getAttribute('url');
    if (url) {
      await this.loadData(url);
    }

    // Subscribe to events
    this.unsubscribePan = pan.subscribe('table.refresh', () => {
      this.refresh();
    });

    // Set up event listeners
    this.addEventListener('page-change', this.handlePageChange);
  }

  disconnectedCallback() {
    // Clean up subscriptions
    if (this.unsubscribePan) {
      this.unsubscribePan();
    }

    // Remove event listeners
    this.removeEventListener('page-change', this.handlePageChange);

    // Cancel pending fetch
    if (this.fetchController) {
      this.fetchController.abort();
    }
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (oldValue === newValue) return;

    if (name === 'url' && newValue) {
      this.loadData(newValue);
    } else if (name === 'page-size') {
      this.pageSize = Number(newValue) || 10;
      this.render();
    }
  }

  async loadData(url) {
    // Cancel previous fetch if any
    if (this.fetchController) {
      this.fetchController.abort();
    }

    this.fetchController = new AbortController();
    try {
      const response = await fetch(url, {
        signal: this.fetchController.signal
      });
      this.data = await response.json();
      this.render();
    } catch (error) {
      if (error.name !== 'AbortError') {
        console.error('Failed to load data:', error);
      }
    }
  }

  render() {
    // Render logic here
    this.shadowRoot.innerHTML = `
      <style>
        table { width: 100%; border-collapse: collapse; }
        th, td { padding: 8px; text-align: left; border-bottom: 1px solid #ddd; }
      </style>
      <table>
        <thead>
          <tr><th>ID</th><th>Name</th><th>Status</th></tr>
        </thead>
        <tbody>
          ${this.data.map(row => `
            <tr>
              <td>${row.id}</td>
              <td>${row.name}</td>
              <td>${row.status}</td>
            </tr>
          `).join('')}
        </tbody>
      </table>
    `;
  }

  handlePageChange = (event) => {
    this.currentPage = event.detail.page;
    this.render();
  }

  async refresh() {
    const url = this.getAttribute('url');
    if (url) {
      await this.loadData(url);
    }
  }
```

```
}

customElements.define('data-table', DataTable);
```

Summary

This chapter introduced LARC's core concepts:

- **Web Components** provide standard, reusable elements with Custom Elements, Shadow DOM, and Templates
- **The PAN Bus** enables decoupled pub/sub communication between components
- **Event-Driven Architecture** uses DOM events, custom events, and PAN messages for different scenarios
- **State Management** starts simple (local state) and scales to shared, persistent, and server state
- **ES Modules** organize code with standard imports/exports and import maps
- **Component Lifecycle** provides callbacks for creation, connection, attribute changes, and cleanup

In the next chapter, we'll put these concepts into practice by setting up a development environment and building your first LARC application.

Key Takeaways

- Web Components are browser standards, not framework abstractions
- Shadow DOM provides true style encapsulation
- The PAN bus decouples components through pub/sub messaging
- Use the simplest state management that works, then scale up
- ES Modules and Import Maps replace build-time bundling
- Proper lifecycle management prevents bugs and memory leaks
- Components should be self-contained but composable

Chapter 3: Getting Started

Theory is important, but there's no substitute for hands-on experience. In this chapter, you'll set up your development environment and build your first LARC application. By the end, you'll have a working project and understand the basic development workflow.

Setting Up Your Development Environment

One of LARC's strengths is minimal setup requirements. You don't need complex tooling or configuration—just a browser, a text editor, and a way to serve files.

Requirements

Essential: - **Modern browser** — Chrome, Firefox, Safari, or Edge (latest version) - **Text editor** — VS Code, Sublime Text, Atom, or any editor you prefer - **Local web server** — Python's SimpleHTTPServer, Node's `http-server`, or VS Code's Live Server extension

Optional but Recommended: - **VS Code** with the LARC extension for snippets and IntelliSense - **Browser DevTools** familiarity for debugging - **Git** for version control

Quick Start with create-larc-app

The fastest way to start is using the LARC CLI:

```
# Install globally
npm install -g create-larc-app

# Create a new project
create-larc-app my-first-app

# Start development server
cd my-first-app
larc dev
```

Open `http://localhost:3000` and you'll see your new LARC application running.

Manual Setup (No CLI)

Don't want to install Node.js? You can set up a LARC project manually:

1. Create project structure:

```
mkdir my-first-app
cd my-first-app
mkdir src
mkdir src/components
mkdir public
```

2. Create `index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>My First LARC App</title>

  <!-- Import Map for dependencies -->
  <script type="importmap">
  {
    "imports": {
      "@larcjs/core": "https://cdn.jsdelivr.net/npm/@larcjs/core@1/dist/index.js"
    }
  }
  </script>

  <style>
    body {
      font-family: system-ui, -apple-system, sans-serif;
      margin: 0;
      padding: 20px;
      background: #f5f5f5;
    }
  </style>
</head>
<body>
  <div id="app"></div>

  <script type="module" src="src/app.js"></script>
</body>
</html>
```

3. Create `src/app.js`:

```
import { pan } from '@larcjs/core';

// Import your components
import './components/hello-world.js';

// Initialize app
console.log('LARC app initialized');
pan.publish('app.ready');
```

```
// Add component to page
document.getElementById('app').innerHTML = '<hello-world></hello-world>';
```

4. Create `src/components/hello-world.js`:

```
class HelloWorld extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <div style="
        background: white;
        padding: 40px;
        border-radius: 8px;
        box-shadow: 0 2px 8px rgba(0,0,0,0.1);
        text-align: center;
      ">
        <h1>Hello, LARC!</h1>
        <p>Welcome to your first LARC application.</p>
      </div>
    `;
  }
}

customElements.define('hello-world', HelloWorld);
```

5. Serve the files:

```
# Python 3
python3 -m http.server 3000

# Or Python 2
python -m SimpleHTTPServer 3000

# Or with Node.js
npx http-server -p 3000

# Or use VS Code Live Server extension
# (right-click index.html → "Open with Live Server")
```

Open `http://localhost:3000` and you should see "Hello, LARC!" displayed.

That's it. No build step. No transpilation. No bundling. Just HTML, CSS, and JavaScript.

Development Tools

VS Code Extensions

Install these extensions for the best experience:

LARC Extension: - Snippets for components and PAN patterns - IntelliSense for LARC APIs - Commands for creating components

Install: Search "LARC" in VS Code extensions marketplace

Live Server: - Auto-reload when files change - Simple local web server - Right-click HTML file to start

Install: Search "Live Server" by Ritwick Dey

ES6 String HTML: - Syntax highlighting for template literals - Makes component templates more readable

Install: Search "ES6 String HTML"

Browser DevTools

Learn these DevTools features for LARC development:

Elements Panel: - Inspect shadow DOM (enable "Show user agent shadow DOM" in settings) - View Custom Elements with their properties - Debug CSS in shadow roots

Console: - Subscribe to all PAN messages: `pan.subscribe('*', console.log)` - Test components directly: `document.querySelector('my-component')` - Check Custom Elements registry: `customElements.get('my-component')`

Network Panel: - Verify ES modules load correctly - Check import map resolution - Monitor API calls

Sources Panel: - Set breakpoints in your source code (no source maps needed!) - Step through component lifecycle - Watch variables and state

Your First LARC Application

Let's build something more interesting than "Hello World"—a simple counter application with multiple components communicating via the PAN bus.

Project Goal

We'll create: - A counter display component - Increment and decrement buttons - A reset button - Communication via PAN bus (no prop drilling!)

Step 1: Update index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Counter App - LARC</title>

  <script type="importmap">
  {
    "imports": {
      "@larcjs/core": "https://cdn.jsdelivr.net/npm/@larcjs/core@1/dist/index.js"
      }
    }
  </script>

  <style>
    * {
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }

    body {
      font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
      background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
      min-height: 100vh;
      display: flex;
      justify-content: center;
      align-items: center;
    }

    #app {
      background: white;
      padding: 40px;
      border-radius: 16px;
      box-shadow: 0 20px 60px rgba(0, 0, 0, 0.3);
      min-width: 400px;
    }
  </style>
</head>
<body>
  <div id="app">
    <counter-display></counter-display>
    <counter-controls></counter-controls>
  </div>

  <script type="module" src="src/app.js"></script>
</body>
</html>
```

Step 2: Create app.js

```
// src/app.js
import { pan } from '@larcjs/core';

// Import components
import './components/counter-display.js';
import './components/counter-controls.js';
```

```javascript
// Initialize application state
let count = 0;

// Listen for increment requests
pan.subscribe('counter.increment', () => {
  count++;
  pan.publish('counter.updated', { count });
});

// Listen for decrement requests
pan.subscribe('counter.decrement', () => {
  count--;
  pan.publish('counter.updated', { count });
});

// Listen for reset requests
pan.subscribe('counter.reset', () => {
  count = 0;
  pan.publish('counter.updated', { count });
});

// Publish initial state
pan.publish('counter.updated', { count });

console.log('Counter app initialized');
```

Step 3: Create counter-display.js

```javascript
// src/components/counter-display.js
import { pan } from '@larcjs/core';

class CounterDisplay extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.count = 0;
  }

  connectedCallback() {
    // Subscribe to count updates
    this.unsubscribe = pan.subscribe('counter.updated', ({ count }) => {
      this.count = count;
      this.render();
    });

    this.render();
  }
  disconnectedCallback() {
    this.unsubscribe();
  }

  render() {
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
          text-align: center;
          margin-bottom: 30px;
        }

        .display {
          font-size: 72px;
          font-weight: bold;
          color: #667eea;
          margin-bottom: 10px;
          font-variant-numeric: tabular-nums;
        }

        .label {
          font-size: 18px;
          color: #666;
          text-transform: uppercase;
          letter-spacing: 2px;
        }
      </style>

      <div class="display">${this.count}</div>
      <div class="label">Current Count</div>
    `;
  }
}

customElements.define('counter-display', CounterDisplay);
```

Step 4: Create counter-controls.js

```
// src/components/counter-controls.js
import { pan } from '@larcjs/core';

class CounterControls extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }
```

```
connectedCallback() {
  this.render();
  this.attachEventListeners();
}

attachEventListeners() {
  this.shadowRoot.querySelector('#increment').addEventListener('click', () => {
    pan.publish('counter.increment');
  });

  this.shadowRoot.querySelector('#decrement').addEventListener('click', () => {
    pan.publish('counter.decrement');
  });

  this.shadowRoot.querySelector('#reset').addEventListener('click', () => {
    pan.publish('counter.reset');
  });
}

render() {
  this.shadowRoot.innerHTML = `
    <style>
      :host {
        display: block;
      }

      .controls {
        display: flex;
        gap: 10px;
        margin-bottom: 15px;
      }

      button {
        flex: 1;
        padding: 15px;
        font-size: 16px;
        font-weight: 600;
        border: none;
        border-radius: 8px;
        cursor: pointer;
        transition: all 0.2s;
      }

      button:hover {
        transform: translateY(-2px);
        box-shadow: 0 4px 12px rgba(0, 0, 0, 0.15);
      }
```

```
        button:active {
          transform: translateY(0);
        }

        #increment {
          background: #48bb78;
          color: white;
        }

        #increment:hover {
          background: #38a169;
        }

        #decrement {
          background: #f56565;
          color: white;
        }

        #decrement:hover {
          background: #e53e3e;
        }

        #reset {
          background: #4a5568;
          color: white;
          width: 100%;
        }

        #reset:hover {
          background: #2d3748;
        }
      </style>

      <div class="controls">
        <button id="decrement">- Decrement</button>
        <button id="increment">+ Increment</button>
      </div>
      <button id="reset">Reset</button>
    `;
  }
}

customElements.define('counter-controls', CounterControls);
```

Step 5: Test Your App

Start your local server and open the page. You should see:

- A large counter display showing "0"
- Increment and decrement buttons
- A reset button

Click the buttons. Notice how: - Components update immediately - State is managed centrally in `app.js` - Components don't reference each other directly - Adding new components is trivial (just subscribe to `counter.updated`)

What Just Happened?

Let's examine the architecture:

Data Flow:

```
User clicks button
     ↓
Controls component publishes event
     ↓
App.js receives event and updates state
     ↓
App.js publishes updated state
     ↓
Display component receives update and re-renders
```

Key Points:

1. **Decoupled Components:** Display and controls don't know about each other
2. **Central State:** State lives in `app.js`, not in components
3. **Pub/Sub Communication:** All communication via PAN bus topics
4. **No Props:** No prop drilling or lifting state up
5. **Easy Testing:** Each component can be tested in isolation

Project Structure

As your application grows, organization becomes important. Here's a recommended structure:

```
my-app/
  index.html              # Entry point
  larc.config.json        # Optional config
  src/
     app.js              # Main application logic
     components/         # Reusable components
        ui/             # Generic UI components
           button.js
           card.js
           modal.js
        features/       # Feature-specific components
           user-profile.js
           todo-list.js
           dashboard.js
        layout/         # Layout components
```

```
            header.js
            sidebar.js
            footer.js
    lib/                # Utilities and helpers
        api.js          # API client
        auth.js         # Authentication
        router.js       # Routing logic
        utils.js        # General utilities
    pages/              # Page-level components
        home.js
        dashboard.js
        settings.js
    styles/             # Global styles
        reset.css
        variables.css
        utilities.css
public/                 # Static assets
    images/
    fonts/
    icons/
tests/                  # Test files
    components/
    integration/
```

File Organization Principles

Components: - One component per file - File name matches component name: `user-profile.js` defines `<user-profile>` - Keep related components together in subdirectories

Lib: - Utilities that don't render UI - API clients, helpers, formatters - Pure functions when possible

Pages: - Top-level route components - Compose smaller components - Handle page-specific logic

Styles: - Global styles in `styles/` - Component-specific styles in Shadow DOM - CSS custom properties for theming

Import Maps Explained

Import Maps are a browser standard that replaces the need for bundlers to resolve module paths.

Basic Import Map

```
<script type="importmap">
{
  "imports": {
    "lodash": "https://cdn.jsdelivr.net/npm/lodash-es@4/lodash.js",
    "dayjs": "https://cdn.jsdelivr.net/npm/dayjs@1/dayjs.min.js"
  }
}
</script>

<script type="module">
  // Use package names instead of URLs
  import _ from 'lodash';
  import dayjs from 'dayjs';

  console.log(dayjs().format('YYYY-MM-DD'));
</script>
```

Path Aliases

Create shortcuts for your own modules:

```
<script type="importmap">
{
  "imports": {
    "@/": "/src/",
    "components/": "/src/components/",
    "lib/": "/src/lib/",
    "@larcjs/core": "https://cdn.jsdelivr.net/npm/@larcjs/core@1/dist/index.js"
  }
}
</script>

<script type="module">
  // Instead of: import { api } from '../../../lib/api.js';
  import { api } from 'lib/api.js';

  // Instead of: import Button from '../components/ui/button.js';
  import Button from 'components/ui/button.js';

  // Instead of: import something from '../../../src/utils.js';
  import something from '@/utils.js';
</script>
```

Version Management

Pin dependencies to specific versions:

```
{
  "imports": {
    "@larcjs/core": "https://cdn.jsdelivr.net/npm/@larcjs/core@1.2.3/dist/index.js",
    "@larcjs/ui": "https://cdn.jsdelivr.net/npm/@larcjs/ui@2.0.1/dist/index.js"
  }
}
```

Or use version ranges for automatic updates:

```
{
  "imports": {
    "@larcjs/core": "https://cdn.jsdelivr.net/npm/@larcjs/core@1/dist/index.js",
    "@larcjs/ui": "https://cdn.jsdelivr.net/npm/@larcjs/ui@2/dist/index.js"
  }
}
```

Multiple CDNs

Add fallbacks for reliability:

```
{
  "imports": {
    "react": "https://esm.sh/react@18",
    "react-fallback": "https://cdn.skypack.dev/react@18"
  }
}
```

Then in code:

```
let React;
try {
  React = await import('react');
} catch {
  React = await import('react-fallback');
}
```

Development vs Production

Use different import maps for different environments:

development.importmap.json:

```
{
  "imports": {
    "@larcjs/core": "/node_modules/@larcjs/core/dist/index.js",
    "app/": "/src/"
  }
}
```

production.importmap.json:

```
{
  "imports": {
    "@larcjs/core": "https://cdn.jsdelivr.net/npm/@larcjs/core@1.2.3/dist/index.js",
    "app/": "/assets/js/"
  }
}
```

Load the appropriate map:

```
<script type="importmap" src="/config/production.importmap.json"></script>
```

Development Workflow

Daily Development

A typical development session:

1. Start dev server:

```
larc dev
```

This starts a local server with hot reload.

2. Edit files: Open your editor and make changes. The browser automatically reloads when you save.

3. Check the console: Open browser DevTools and check for errors or warnings.

4. Test in browser: Interact with your app, verify behavior, check responsive design.

5. Debug as needed: Set breakpoints, inspect elements, monitor network requests.

6. Repeat: The edit-refresh cycle is instant with no build step.

Debugging Tips

Log all PAN messages:

```javascript
pan.subscribe('*', (topic, data) => {
  console.log(`[PAN] ${topic}:`, data);
});
```

Inspect custom elements:

```javascript
// Get element
const el = document.querySelector('my-component');

// Check if defined
console.log(customElements.get('my-component'));

// Access shadow root
console.log(el.shadowRoot);

// Call methods directly
el.someMethod();
```

Monitor attribute changes:

```javascript
// Create observer
const observer = new MutationObserver((mutations) => {
  mutations.forEach(mutation => {
    console.log('Attribute changed:', mutation.attributeName);
  });
});
```

```
// Watch element
observer.observe(element, { attributes: true });
```

Testing

Run tests without a build step:

```
<!-- tests/counter.test.html -->
<!DOCTYPE html>
<html>
<head>
  <title>Counter Tests</title>
  <script type="importmap">
  {
    "imports": {
      "@larcjs/core": "../node_modules/@larcjs/core/dist/index.js"
    }
  }
  </script>
</head>
<body>
  <div id="test-container"></div>

  <script type="module">
    import { pan } from '@larcjs/core';
    import '../src/components/counter-display.js';

    // Simple test framework
    function test(name, fn) {
      try {
        fn();
        console.log(`  ${name}`);
      } catch (error) {
        console.error(`  ${name}:`, error);
      }
    }

    function assert(condition, message) {
      if (!condition) throw new Error(message || 'Assertion failed');
    }

    // Tests
    test('counter-display renders initial count', () => {
      const el = document.createElement('counter-display');
      document.getElementById('test-container').appendChild(el);

      const display = el.shadowRoot.querySelector('.display');
      assert(display.textContent === '0', 'Initial count should be 0');
      el.remove();
    });

    test('counter-display updates on PAN message', async () => {
      const el = document.createElement('counter-display');
      document.getElementById('test-container').appendChild(el);

      // Wait for component to connect
      await new Promise(resolve => setTimeout(resolve, 10));

      // Publish update
      pan.publish('counter.updated', { count: 42 });

      // Wait for render
      await new Promise(resolve => setTimeout(resolve, 10));

      const display = el.shadowRoot.querySelector('.display');
      assert(display.textContent === '42', 'Count should update to 42');

      el.remove();
    });

    console.log('All tests complete');
  </script>
</body>
</html>
```

Open `tests/counter.test.html` in your browser to run tests.

Common Patterns

Pattern 1: Loading States

```
class DataComponent extends HTMLElement {
  async connectedCallback() {
    this.render({ loading: true });

    try {
      const data = await this.fetchData();
      this.render({ data });
    } catch (error) {
      this.render({ error: error.message });
    }
  }

  render(state) {
    if (state.loading) {
      this.innerHTML = '<loading-spinner></loading-spinner>';
    } else if (state.error) {
      this.innerHTML = `<error-message>${state.error}</error-message>`;
    } else {
      this.innerHTML = `<data-display .data="${state.data}"></data-display>`;
    }
  }
}
```

Pattern 2: Form Handling

```
class LoginForm extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <form>
        <input type="email" name="email" required>
        <input type="password" name="password" required>
        <button type="submit">Login</button>
      </form>
    `;

    this.querySelector('form').addEventListener('submit', async (e) => {
      e.preventDefault();

      const formData = new FormData(e.target);
      const data = Object.fromEntries(formData);

      pan.publish('auth.login', data);
    });
  }
}
```

Pattern 3: Conditional Rendering

```
class UserMenu extends HTMLElement {
  constructor() {
    super();
    this.user = null;
  }

  connectedCallback() {
    pan.subscribe('auth.user.changed', ({ user }) => {
      this.user = user;
      this.render();
    });

    this.render();
  }
  render() {
    if (this.user) {
      this.innerHTML = `
        <div class="logged-in">
          <span>Hello, ${this.user.name}</span>
          <button id="logout">Logout</button>
        </div>
      `;

      this.querySelector('#logout').addEventListener('click', () => {
        pan.publish('auth.logout');
      });
    } else {
      this.innerHTML = `
        <button id="login">Login</button>
      `;

      this.querySelector('#login').addEventListener('click', () => {
        pan.publish('app.navigate', { path: '/login' });
      });
    }
  }
}
```

Pattern 4: Lists and Iteration

```
class TodoList extends HTMLElement {
  constructor() {
    super();
    this.todos = [];
  }

  connectedCallback() {
    pan.subscribe('todos.updated', ({ todos }) => {
      this.todos = todos;
      this.render();
    });

    this.render();
  }

  render() {
    this.innerHTML = `
      <ul>
        ${this.todos.map(todo => `
          <li>
            <input type="checkbox"
```

```
                ${todo.completed ? 'checked' : ''}
                data-id="${todo.id}">
          <span class="${todo.completed ? 'completed' : ''}">
            ${todo.text}
          </span>
        </li>
      `).join('')}
    </ul>
  `;

  // Attach event listeners after rendering
  this.querySelectorAll('input[type="checkbox"]').forEach(checkbox => {
    checkbox.addEventListener('change', (e) => {
      const id = e.target.dataset.id;
      pan.publish('todos.toggle', { id });
    });
  });
 }
}
```

Summary

In this chapter, you:

- Set up a LARC development environment (CLI or manual)
- Built your first multi-component application
- Learned project structure best practices
- Mastered Import Maps for dependency management
- Established an efficient development workflow
- Explored common component patterns

You now have a solid foundation for building LARC applications. The next chapter dives deeper into creating sophisticated Web Components with proper lifecycle management, styling, and interactivity.

Exercises

1. Enhance the Counter App: - Add a history component that shows past values - Add increment/decrement by custom amounts - Persist count to localStorage

2. Build a Todo List: - Add/remove todos - Mark as complete/incomplete - Filter by status (all/active/completed) - Use PAN bus for state management

3. Create a Theme Switcher: - Light/dark theme toggle - Publish theme changes via PAN - Multiple components respond to theme changes - Persist theme preference

4. Experiment with Import Maps: - Try different CDNs (jsDelivr, unpkg, esm.sh) - Add path aliases for your components - Import an external library (lodash, dayjs, etc.)

Take your time with these exercises. Understanding these patterns now will make the rest of the book much easier.

Chapter 4: Creating Web Components

Now that you've built your first LARC application, it's time to master the art of creating robust, reusable Web Components. This chapter covers everything from basic component anatomy to advanced patterns like composition, slots, and performance optimization.

By the end of this chapter, you'll be able to build production-quality components that are maintainable, testable, and performant.

Anatomy of a LARC Component

Let's dissect a well-structured LARC component to understand its parts:

```
// Import dependencies
import { pan } from '@larcjs/core';
import { formatDate } from '../lib/utils.js';

/**
 * A card component for displaying user information.
 *
 * @element user-card
 *
 * @attr {string} user-id - The ID of the user to display
 * @attr {boolean} compact - Display in compact mode
 *
 * @fires user-selected - Dispatched when card is clicked
 *
 * @slot - Default slot for additional content
 * @slot actions - Slot for action buttons
 */
class UserCard extends HTMLElement {
  // 1. Define observed attributes
  static get observedAttributes() {
    return ['user-id', 'compact'];
  }

  // 2. Constructor - initialize instance
```

```
constructor() {
  super();

  // Attach shadow DOM
  this.attachShadow({ mode: 'open' });

  // Initialize private state
  this._user = null;
  this._loading = false;
  this._error = null;

  // Bind event handlers
  this.handleClick = this.handleClick.bind(this);
}

// 3. Lifecycle: connected to DOM
connectedCallback() {
  this.render();

  // Load user data if ID is provided
  const userId = this.getAttribute('user-id');
  if (userId) {
    this.loadUser(userId);
  }

  // Subscribe to PAN events
  this.unsubscribe = pan.subscribe('user.updated', this.handleUserUpdate);

  // Add event listeners
  this.shadowRoot.addEventListener('click', this.handleClick);
}

// 4. Lifecycle: disconnected from DOM
disconnectedCallback() {
  // Clean up subscriptions
  if (this.unsubscribe) {
    this.unsubscribe();
  }

  // Remove event listeners
  this.shadowRoot.removeEventListener('click', this.handleClick);
}

// 5. Lifecycle: attributes changed
attributeChangedCallback(name, oldValue, newValue) {
  if (oldValue === newValue) return;

  if (name === 'user-id' && newValue) {
    this.loadUser(newValue);
  } else if (name === 'compact') {
    this.render();
  }
}

// 6. Public properties with getters/setters
get user() {
  return this._user;
}

set user(value) {
  this._user = value;
  this.render();
}

get loading() {
  return this._loading;
}

// 7. Public methods
async loadUser(userId) {
  this._loading = true;
  this._error = null;
  this.render();

  try {
    const response = await fetch(`/api/users/${userId}`);
    if (!response.ok) throw new Error('Failed to load user');

    this._user = await response.json();
    this._loading = false;
    this.render();
  } catch (error) {
    this._error = error.message;
    this._loading = false;
    this.render();
  }
}

refresh() {
  const userId = this.getAttribute('user-id');
  if (userId) {
    this.loadUser(userId);
  }
}

// 8. Private methods
```

```
handleClick(event) {
  if (!this._user) return;

  this.dispatchEvent(new CustomEvent('user-selected', {
    detail: { user: this._user },
    bubbles: true,
    composed: true
  }));
}

handleUserUpdate = (data) => {
  if (data.userId === this.getAttribute('user-id')) {
    this._user = data.user;
    this.render();
  }
}

// 9. Render method
render() {
  const compact = this.hasAttribute('compact');

  if (this._loading) {
    this.shadowRoot.innerHTML = this.renderLoading();
    return;
  }

  if (this._error) {
    this.shadowRoot.innerHTML = this.renderError();
    return;
  }

  if (!this._user) {
    this.shadowRoot.innerHTML = this.renderEmpty();
    return;
  }

  this.shadowRoot.innerHTML = compact
    ? this.renderCompact()
    : this.renderFull();
}

renderLoading() {
  return `
    <style>${this.styles()}</style>
    <div class="card loading">
      <div class="spinner"></div>
      <p>Loading...</p>
    </div>
  `;
}

renderError() {
  return `
    <style>${this.styles()}</style>
    <div class="card error">
      <p class="error-message">${this._error}</p>
      <button class="retry">Retry</button>
    </div>
  `;
}

renderEmpty() {
  return `
    <style>${this.styles()}</style>
    <div class="card empty">
      <p>No user data</p>
    </div>
  `;
}

renderCompact() {
  return `
    <style>${this.styles()}</style>
    <div class="card compact">
      <img src="${this._user.avatar}" alt="${this._user.name}">
      <div class="info">
        <h3>${this._user.name}</h3>
        <slot name="actions"></slot>
      </div>
    </div>
  `;
}

renderFull() {
  return `
    <style>${this.styles()}</style>
    <div class="card">
      <div class="header">
        <img src="${this._user.avatar}" alt="${this._user.name}" class="avatar">
        <div class="header-content">
          <h2>${this._user.name}</h2>
          <p class="email">${this._user.email}</p>
        </div>
      </div>
      <div class="body">
        <p class="bio">${this._user.bio || 'No bio available'}</p>
        <div class="meta">
          <span>Joined ${formatDate(this._user.createdAt)}</span>
        </div>
        <slot></slot>
      </div>
      <div class="footer">
        <slot name="actions"></slot>
      </div>
    </div>
  `;
}

// 10. Styles
styles() {
  return `
    :host {
      display: block;
      cursor: pointer;
    }

    .card {
      background: white;
      border-radius: 8px;
      box-shadow: 0 2px 4px rgba(0,0,0,0.1);
      padding: 16px;
      transition: box-shadow 0.2s;
    }

    .card:hover {
      box-shadow: 0 4px 12px rgba(0,0,0,0.15);
    }

    .header {
      display: flex;
      gap: 12px;
      margin-bottom: 16px;
    }

    .avatar {
      width: 48px;
      height: 48px;
      border-radius: 50%;
      object-fit: cover;
    }

    h2 {
      margin: 0;
      font-size: 18px;
      color: #333;
    }

    .email {
      margin: 4px 0 0 0;
      font-size: 14px;
      color: #666;
    }

    .bio {
      color: #444;
      line-height: 1.5;
    }

    .meta {
      font-size: 12px;
      color: #999;
      margin-top: 12px;
    }

    .loading, .error, .empty {
      text-align: center;
      padding: 40px 20px;
      color: #666;
    }

    .spinner {
      border: 3px solid #f3f3f3;
      border-top: 3px solid #667eea;
      border-radius: 50%;
      width: 40px;
      height: 40px;
      animation: spin 1s linear infinite;
      margin: 0 auto 16px;
    }

    @keyframes spin {
      to { transform: rotate(360deg); }
    }

    .error-message {
      color: #e53e3e;
    }

    .compact {
      display: flex;
      align-items: center;
      gap: 12px;
```

```
        padding: 12px;
      }

      .compact img {
        width: 40px;
        height: 40px;
        border-radius: 50%;
      }

      .compact h3 {
        margin: 0;
        font-size: 14px;
      }
    `;
  }
}

// 11. Register the custom element
customElements.define('user-card', UserCard);

// 12. Export for use in other modules
export default UserCard;
```

Component Structure Breakdown

1. Documentation: - JSDoc comments explain usage - Attribute, property, event, and slot documentation - Helps other developers understand the component

2. Static Properties: - `observedAttributes` defines which attributes trigger `attributeChangedCallback` - Keep this list minimal for performance

3. Constructor: - Initialize instance variables - Attach shadow DOM - Bind methods (for event handlers) - Don't access attributes or DOM here

4. Lifecycle Methods: - `connectedCallback`: Setup when added to DOM - `disconnectedCallback`: Cleanup when removed - `attributeChangedCallback`: Respond to attribute changes

5. Properties: - Use private fields (`_user`) for internal state - Provide getters/setters for public API - Setters can trigger re-renders

6. Methods: - Public methods for external use - Private methods (conventionally start with `_` or use `#` private fields) - Keep methods focused and single-purpose

7. Rendering: - Separate render logic from state management - Multiple render methods for different states - Extract styles to a separate method

Shadow DOM Deep Dive

Shadow DOM is one of the most powerful features of Web Components. It provides true encapsulation for both markup and styles.

Creating Shadow DOM

```
class MyComponent extends HTMLElement {
  constructor() {
    super();

    // Create shadow root
    this.attachShadow({ mode: 'open' });

    // mode: 'open' - shadow root accessible via element.shadowRoot
    // mode: 'closed' - shadow root not accessible (rarely used)
  }
}
```

Shadow DOM vs Light DOM

```
<my-component>
  <!-- This is Light DOM (regular DOM) -->
  <p>Visible content</p>
</my-component>

<script>
  class MyComponent extends HTMLElement {
    constructor() {
      super();
      this.attachShadow({ mode: 'open' });

      // This is Shadow DOM
      this.shadowRoot.innerHTML = `
        <div class="shadow-content">
          <h2>Shadow DOM Content</h2>
          <slot></slot>
        </div>
      `;
    }
  }

  customElements.define('my-component', MyComponent);
</script>
```

Result: - Light DOM (`<p>Visible content</p>`) is projected into the `<slot>` - Shadow DOM provides the structure and styling - Styles in shadow DOM don't leak out - Styles from light DOM don't leak in

Style Encapsulation

```
class StyledButton extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        /* These styles only affect this component */
        button {
          background: blue;
          color: white;
          border: none;
          padding: 10px 20px;
          border-radius: 4px;
          cursor: pointer;
        }

        button:hover {
          background: darkblue;
        }
      </style>
      <button><slot></slot></button>
    `;
  }
}
```

Key Points: - Styles inside shadow DOM are scoped - No conflicts with global styles - No CSS class name collisions - True component encapsulation

The :host Selector

Style the component itself:

```
:host {
  display: block;
  margin: 16px 0;
}

/* Style host when it has a class */
:host(.highlighted) {
  border: 2px solid gold;
}

/* Style host when it has an attribute */
:host([disabled]) {
  opacity: 0.5;
  pointer-events: none;
}

/* Style host in specific contexts */
:host-context(.dark-theme) {
  background: #333;
  color: white;
}
```

CSS Custom Properties (Variables)

CSS variables pierce the shadow DOM boundary:

```
// Component defines and uses variables
class ThemedCard extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });

    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
          background: var(--card-bg, white);
          color: var(--card-text, black);
          border: 1px solid var(--card-border, #ddd);
          border-radius: var(--card-radius, 8px);
          padding: var(--card-padding, 16px);
        }
      </style>
      <slot></slot>
    `;
  }
}
```

Usage:

```
<style>
  /* Override component variables from outside */
  themed-card {
    --card-bg: #f0f0f0;
    --card-text: #333;
    --card-border: #ccc;
    --card-radius: 12px;
  }

  themed-card.dark {
    --card-bg: #333;
    --card-text: #fff;
    --card-border: #555;
  }
</style>

<themed-card>Normal theme</themed-card>
<themed-card class="dark">Dark theme</themed-card>
```

This pattern allows theming while maintaining encapsulation.

Parts and ::part()

Expose specific shadow DOM elements for styling:

```
class FancyButton extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });

    this.shadowRoot.innerHTML = `
      <style>
        button { /* default styles */ }
        .icon { /* icon styles */ }
      </style>
      <button part="button">
        <span part="icon" class="icon">→</span>
        <slot></slot>
      </button>
    `;
  }
}
```

Style from outside:

```
fancy-button::part(button) {
  background: linear-gradient(135deg, #667eea, #764ba2);
}

fancy-button::part(icon) {
  color: gold;
}
```

This gives consumers more control while maintaining encapsulation.

Attributes and Properties

Understanding the difference between attributes and properties is crucial for component design.

Attributes vs Properties

Attributes: - HTML attributes (`<my-el foo="bar">`) - Always strings - Visible in HTML - Trigger `attributeChangedCallback`

Properties: - JavaScript properties (`element.foo = 123`) - Any type (string, number, object, etc.) - Not visible in HTML - Direct access, no callback

Reflecting Properties to Attributes

```
class ToggleButton extends HTMLElement {
  static get observedAttributes() {
    return ['checked'];
  }
  constructor() {
    super();
    this._checked = false;
  }

  // Property getter
  get checked() {
    return this._checked;
  }

  // Property setter - reflects to attribute
  set checked(value) {
    const isChecked = Boolean(value);

    if (isChecked) {
      this.setAttribute('checked', '');
    } else {
      this.removeAttribute('checked');
    }
  }

  // Attribute changed - updates property
  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'checked') {
      this._checked = newValue !== null;
      this.render();
    }
  }

  render() {
    this.innerHTML = `
      <button class="${this._checked ? 'checked' : ''}">
        ${this._checked ? ' ' : ' '}
      </button>
    `;
  }
}
```

Usage:

```
<!-- Set via attribute -->
<toggle-button checked></toggle-button>

<script>
  const toggle = document.querySelector('toggle-button');

  // Set via property
  toggle.checked = true;
  // Get property
  console.log(toggle.checked); // true

  // Check attribute
  console.log(toggle.hasAttribute('checked')); // true
</script>
```

When to Use Each

Use Attributes for: - Simple configuration (strings, numbers, booleans) - Values that should be visible in HTML - Initial configuration from HTML - Values that need to work with CSS selectors

Use Properties for: - Complex data (objects, arrays, functions) - Data that changes frequently - Large data that shouldn't serialize to HTML - Callback functions

Type Conversion

Attributes are always strings, so convert appropriately:

```
attributeChangedCallback(name, oldValue, newValue) {
  if (name === 'count') {
    this._count = Number(newValue) || 0;
  } else if (name === 'enabled') {
    this._enabled = newValue !== null; // Boolean attribute
  } else if (name === 'options') {
    try {
      this._options = JSON.parse(newValue);
    } catch {
      this._options = {};
    }
  }
}
```

Boolean Attributes

Follow HTML conventions:

```
// Boolean attribute: presence = true, absence = false
if (this.hasAttribute('disabled')) {
  // Is disabled
}

// Set boolean attribute
this.setAttribute('disabled', ''); // value doesn't matter

// Remove boolean attribute
this.removeAttribute('disabled');
```

Component Styling

Internal Styles

Most styles should be in shadow DOM:

```
styles() {
  return `
    :host {
      display: block;
    }

    .container {
      padding: 16px;
    }

    /* All your component styles */
  `;
}
```

External Stylesheets

For larger components, link external styles:

```
connectedCallback() {
  this.attachShadow({ mode: 'open' });

  this.shadowRoot.innerHTML = `
    <link rel="stylesheet" href="/styles/components/user-card.css">
    <div class="user-card">
      <!-- content -->
    </div>
  `;
}
```

Adoptable Stylesheets

Share styles between component instances:

```
// Create shared stylesheet once
const sheet = new CSSStyleSheet();
sheet.replaceSync(`
  .card {
    padding: 16px;
    border-radius: 8px;
    background: white;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
  }
`);
```

```
class CardComponent extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });

    // Adopt shared stylesheet (very fast)
    this.shadowRoot.adoptedStyleSheets = [sheet];

    this.shadowRoot.innerHTML = `
      <div class="card">
        <slot></slot>
      </div>
    `;
  }
}
```

Benefits: - Styles parsed once, shared across instances - Better performance with many components - Modify shared styles dynamically

Theming Strategies

Strategy 1: CSS Custom Properties

```
class ThemedComponent extends HTMLElement {
  styles() {
    return `
      :host {
        --primary-color: var(--app-primary, #667eea);
        --background: var(--app-bg, white);
        --text: var(--app-text, #333);
      }

      .content {
        background: var(--background);
        color: var(--text);
      }

      button {
        background: var(--primary-color);
      }
    `;
  }
}
```

Strategy 2: Class-Based Themes

```
class ThemeAwareComponent extends HTMLElement {
  connectedCallback() {
    // Observe theme changes on documentElement
    const observer = new MutationObserver(() => {
      this.updateTheme();
    });

    observer.observe(document.documentElement, {
      attributes: true,
      attributeFilter: ['data-theme']
    });

    this.updateTheme();
  }

  updateTheme() {
    const theme = document.documentElement.dataset.theme || 'light';
    this.setAttribute('theme', theme);
  }

  styles() {
    return `
      :host([theme="light"]) {
        background: white;
        color: black;
      }

      :host([theme="dark"]) {
        background: #333;
        color: white;
      }
    `;
  }
}
```

Strategy 3: PAN-Based Themes

```
import { pan } from '@larcjs/core';

class PanThemedComponent extends HTMLElement {
  connectedCallback() {
    this.unsubscribe = pan.subscribe('app.theme.changed', ({ theme }) => {
      this.applyTheme(theme);
    });

    // Request current theme
    pan.request('app.theme.get').then(theme => {
      this.applyTheme(theme);
    });
  }

  applyTheme(theme) {
    this.setAttribute('data-theme', theme);
  }
}
```

Lifecycle Methods (Advanced Patterns)

Deferred Rendering

Wait for dependencies before rendering:

```
class DataDisplay extends HTMLElement {
  async connectedCallback() {
    // Wait for dependencies to load
    await customElements.whenDefined('loading-spinner');
    await customElements.whenDefined('error-message');

    // Now render
    this.render();
  }
}
```

Preventing Memory Leaks

```
class WebSocketComponent extends HTMLElement {
  connectedCallback() {
    this.ws = new WebSocket('wss://api.example.com');

    this.ws.onmessage = (event) => {
      this.handleMessage(event.data);
    };

    this.ws.onerror = (error) => {
      console.error('WebSocket error:', error);
    };
  }

  disconnectedCallback() {
    // Clean up WebSocket connection
    if (this.ws) {
      this.ws.close();
      this.ws = null;
    }
  }
}
```

Handling Rapid Reconnection

Components can be disconnected and reconnected quickly:

```
class RobustComponent extends HTMLElement {
  connectedCallback() {
    // Might be called multiple times
    // Use a guard to prevent duplicate setup
    if (this._initialized) {
      return;
    }

    this._initialized = true;
    this.setup();
  }

  disconnectedCallback() {
    // Use setTimeout to debounce
    this._cleanupTimer = setTimeout(() => {
      this.cleanup();
      this._initialized = false;
    }, 100);
  }

  connectedCallback() {
    // Cancel cleanup if reconnected quickly
    if (this._cleanupTimer) {
      clearTimeout(this._cleanupTimer);
      this._cleanupTimer = null;
    }

    if (this._initialized) {
      return;
    }

    this._initialized = true;
    this.setup();
  }
}
```

Testing Components

Unit Testing

Test components in isolation:

```
// tests/user-card.test.js
import { expect } from '@open-wc/testing';
import '../src/components/user-card.js';

describe('UserCard', () => {
  let element;

  beforeEach(() => {
    element = document.createElement('user-card');
    document.body.appendChild(element);
  });

  afterEach(() => {
    element.remove();
  });

  it('renders empty state by default', () => {
    const emptyText = element.shadowRoot.querySelector('.empty');
    expect(emptyText).to.exist;
  });

  it('loads user when user-id attribute is set', async () => {
    // Mock fetch
    global.fetch = async () => ({
      ok: true,
      json: async () => ({ id: 1, name: 'John Doe', email: 'john@example.com' })
    });

    element.setAttribute('user-id', '1');

    // Wait for async operations
    await new Promise(resolve => setTimeout(resolve, 100));

    const name = element.shadowRoot.querySelector('h2');
    expect(name.textContent).to.equal('John Doe');
  });

  it('handles loading state', async () => {
    element.setAttribute('user-id', '1');

    const spinner = element.shadowRoot.querySelector('.spinner');
    expect(spinner).to.exist;
  });

  it('dispatches user-selected event on click', async () => {
    element._user = { id: 1, name: 'John' };
    element.render();

    let eventData = null;
    element.addEventListener('user-selected', (e) => {
      eventData = e.detail;
    });

    element.shadowRoot.querySelector('.card').click();

    expect(eventData).to.deep.equal({ user: { id: 1, name: 'John' } });
  });
});
```

Integration Testing

Test components working together:

```
// tests/counter-integration.test.js
describe('Counter Integration', () => {
  beforeEach(() => {
    document.body.innerHTML = `
      <counter-display></counter-display>
      <counter-controls></counter-controls>
    `;
  });

  it('updates display when controls are clicked', async () => {
    const display = document.querySelector('counter-display');
    const controls = document.querySelector('counter-controls');

    const incrementBtn = controls.shadowRoot.querySelector('#increment');
    incrementBtn.click();

    await new Promise(resolve => setTimeout(resolve, 50));

    const displayValue = display.shadowRoot.querySelector('.display').textContent;
    expect(displayValue).to.equal('1');
  });
});
```

Visual Regression Testing

Catch visual bugs:

```
// tests/visual.test.js
import puppeteer from 'puppeteer';
import pixelmatch from 'pixelmatch';

describe('Visual Regression', () => {
  let browser, page;

  beforeAll(async () => {
    browser = await puppeteer.launch();
    page = await browser.newPage();
  });

  afterAll(async () => {
    await browser.close();
  });
  it('user-card matches snapshot', async () => {
    await page.goto('http://localhost:3000/tests/user-card.html');

    const screenshot = await page.screenshot({ fullPage: true });
    const baseline = fs.readFileSync('tests/snapshots/user-card.png');

    const diff = pixelmatch(screenshot, baseline, null, 800, 600, {
      threshold: 0.1
    });

    expect(diff).to.be.lessThan(100); // Allow small differences
  });
});
```

Summary

This chapter covered:

- **Component Anatomy**: Structure, lifecycle, and organization
- **Shadow DOM**: Encapsulation, slots, and styling
- **Attributes vs Properties**: When to use each and how to reflect them
- **Component Styling**: Internal styles, theming, and CSS custom properties
- **Lifecycle Patterns**: Memory management and robust connection handling
- **Testing**: Unit, integration, and visual regression testing

You now know how to build production-quality Web Components. The next chapter explores the PAN bus in depth, showing you how to orchestrate component communication at scale.

Best Practices

1. **Always clean up in `disconnectedCallback`**
   - Remove event listeners
   - Cancel pending operations
   - Unsubscribe from events
2. **Use Shadow DOM for encapsulation**
   - Keep styles scoped
   - Avoid global style pollution
   - Use `:host` and CSS custom properties for theming
3. **Reflect important properties to attributes**
   - Makes state visible in HTML
   - Enables CSS selectors
   - Improves debugging
4. **Keep components focused**
   - Single responsibility principle
   - Compose larger components from smaller ones

   - Extract shared logic to utilities
5. **Test early and often**
   - Write tests as you build components
   - Test both happy paths and error cases
   - Use integration tests for component interaction

Chapter 5: The PAN Bus

The Page Area Network (PAN) bus is LARC's event-driven communication backbone. It enables decoupled, scalable component architectures by providing a pub/sub messaging system that works across your entire application.

In this chapter, you'll master the PAN bus: from basic publish/subscribe patterns to advanced message routing, error handling, and debugging techniques. By the end, you'll be able to build complex applications where components communicate seamlessly without tight coupling.

Understanding Pub/Sub Architecture

Publish/Subscribe (pub/sub) is a messaging pattern where senders (publishers) don't directly target specific receivers (subscribers). Instead, messages are sent to topics, and any component interested in those topics receives them.

Traditional Communication

Without pub/sub, components need direct references:

```
//  Tight coupling
class LoginButton {
  handleLogin() {
    const user = this.authenticate();

    // Direct reference to other components
    document.querySelector('user-menu').updateUser(user);
    document.querySelector('sidebar').showUserPanel();
    document.querySelector('notification').show('Welcome!');
  }
}
```

Problems: - LoginButton must know about all dependent components - Adding new components requires modifying LoginButton - Components can't work independently - Testing requires mocking all dependencies

Pub/Sub Communication

With the PAN bus:

```
//  Loose coupling
class LoginButton {
  handleLogin() {
    const user = this.authenticate();

    // Publish event - don't care who listens
    pan.publish('user.logged-in', { user });
  }
}

// Separate components subscribe independently
class UserMenu {
  connectedCallback() {
    pan.subscribe('user.logged-in', ({ user }) => {
      this.updateUser(user);
    });
  }
}

class Sidebar {
  connectedCallback() {
    pan.subscribe('user.logged-in', () => {
      this.showUserPanel();
    });
  }
}

class Notification {
  connectedCallback() {
    pan.subscribe('user.logged-in', () => {
      this.show('Welcome!');
    });
  }
}
```

Benefits: - LoginButton doesn't know about consumers - Add new subscribers without changing publishers - Components work independently - Easy to test in isolation

The PAN Bus API

The PAN bus provides three core operations:

```
import { pan } from '@larcjs/core';

// 1. Publish - send a message to a topic
pan.publish('topic.name', { data: 'value' });

// 2. Subscribe - listen for messages on a topic
const unsubscribe = pan.subscribe('topic.name', (data) => {
  console.log('Received:', data);
});

// 3. Unsubscribe - stop listening
unsubscribe();
```

That's the foundation. Everything else builds on these three operations.

Topics and Namespaces

Topics are the routing keys for messages. Well-designed topics make your application's data flow clear and maintainable.

Topic Naming Conventions

Use dot notation to create hierarchies:

```
domain.entity.action
```

Examples:

```
user.profile.updated
user.auth.login
user.auth.logout
user.settings.changed

cart.item.added
cart.item.removed
cart.total.calculated
cart.checkout.started
cart.checkout.completed

notification.info.show
notification.warning.show
notification.error.show

app.theme.changed
app.language.changed
app.route.changed
```

Namespace Structure

Organize topics by domain:

User Domain:

```
user.auth.login
user.auth.logout
user.auth.refresh
user.profile.fetch
user.profile.update
user.settings.fetch
user.settings.update
```

Shopping Cart Domain:

```
cart.init
cart.item.add
cart.item.remove
cart.item.update
cart.clear
cart.checkout
```

Application Domain:

```
app.ready
app.error
app.navigate
app.theme.change
app.modal.open
app.modal.close
```

Wildcards

Subscribe to multiple topics using wildcards:

```
// Subscribe to all user events
pan.subscribe('user.*', (data) => {
  console.log('User event:', data);
});

// Subscribe to all auth events across domains
pan.subscribe('*.auth.*', (data) => {
  console.log('Auth event:', data);
});

// Subscribe to ALL events (debugging)
pan.subscribe('*', (topic, data) => {
  console.log(`[${topic}]`, data);
});
```

Wildcard Patterns: - `user.*` - All user events (user.login, user.logout, etc.) - `*.created` - All create events (user.created, post.created, etc.) - `user.*.updated` - All user update events (user.profile.updated, user.settings.updated, etc.) - `*` - All events

Topic Best Practices

1. Be Specific:

```
//   Good - clear intent
pan.publish('cart.item.added', { item, quantity });
```

```
//  Bad - vague
pan.publish('cart.update', { type: 'add', item, quantity });
```

2. Use Consistent Tense:

```
//  Good - past tense for events that happened
pan.publish('user.logged-in', { user });
pan.publish('data.loaded', { data });

//  Bad - mixed tense
pan.publish('user.login', { user });  // Is this a command or event?
```

3. Include Context:

```
//  Good - data includes context
pan.publish('task.completed', {
  taskId: 123,
  userId: 456,
  completedAt: new Date()
});

//  Bad - missing context
pan.publish('task.done', { id: 123 });
```

4. Avoid Over-Nesting:

```
//  Good - clear and concise
pan.publish('user.profile.updated', { user });

//  Bad - too nested
pan.publish('app.domain.user.entity.profile.action.updated', { user });
```

Publishing Messages

Publishing is straightforward, but there are patterns and options to understand.

Basic Publishing

```
pan.publish('event.name', { any: 'data' });
```

The data can be anything JSON-serializable:

```
// Simple value
pan.publish('counter.updated', 42);

// Object
pan.publish('user.logged-in', {
  userId: 123,
  username: 'john',
  email: 'john@example.com'
});
```

```
// Array
pan.publish('items.loaded', [
  { id: 1, name: 'Item 1' },
  { id: 2, name: 'Item 2' }
]);

// Null/undefined
pan.publish('data.cleared', null);
```

Publishing from Components

Publish in response to user actions or state changes:

```
class AddToCartButton extends HTMLElement {
  connectedCallback() {
    this.addEventListener('click', this.handleClick);
  }

  async handleClick() {
    const productId = this.getAttribute('product-id');
    const quantity = parseInt(this.getAttribute('quantity') || 1);

    // Publish intent
    pan.publish('cart.item.add-requested', { productId, quantity });

    try {
      // Perform action
      await this.addToCart(productId, quantity);

      // Publish success
      pan.publish('cart.item.added', {
        productId,
        quantity,
        timestamp: Date.now()
      });
    } catch (error) {
      // Publish failure
      pan.publish('cart.item.add-failed', {
        productId,
        quantity,
        error: error.message
      });
    }
  }

  async addToCart(productId, quantity) {
    const response = await fetch('/api/cart/items', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ productId, quantity })
    });

    if (!response.ok) {
      throw new Error('Failed to add item to cart');
    }

    return response.json();
  }
}
```

Event Metadata

Include metadata for debugging and auditing:

```
function publishWithMetadata(topic, data) {
  pan.publish(topic, {
    ...data,
    _meta: {
      timestamp: Date.now(),
      source: 'UserComponent',
      userId: currentUser?.id,
      sessionId: sessionId
    }
  });
}

// Usage
publishWithMetadata('order.placed', {
  orderId: 12345,
  total: 99.99
});
```

Batch Publishing

Publish multiple events efficiently:

```
function syncLocalChanges(changes) {
  changes.forEach(change => {
    switch (change.type) {
      case 'add':
        pan.publish('data.item.added', change.item);
        break;
      case 'update':
        pan.publish('data.item.updated', change.item);
        break;
      case 'delete':
        pan.publish('data.item.deleted', { id: change.id });
        break;
    }
  });

  // Publish batch complete
  pan.publish('data.sync.completed', {
    changesCount: changes.length,
    timestamp: Date.now()
  });
}
```

Subscribing to Events

Subscriptions are how components react to events they care about.

Basic Subscription

```
const unsubscribe = pan.subscribe('event.name', (data) => {
  console.log('Received:', data);
});

// Later, when done
unsubscribe();
```

Component Lifecycle Integration

Subscribe in `connectedCallback`, unsubscribe in `disconnectedCallback`:

```
class NotificationDisplay extends HTMLElement {
  connectedCallback() {
    // Subscribe to notification events
    this.unsubscribeInfo = pan.subscribe('notification.info', this.showInfo);
    this.unsubscribeWarning = pan.subscribe('notification.warning', this.showWarning);
    this.unsubscribeError = pan.subscribe('notification.error', this.showError);
  }

  disconnectedCallback() {
    // Clean up subscriptions
    this.unsubscribeInfo();
    this.unsubscribeWarning();
    this.unsubscribeError();
  }

  showInfo = (data) => {
    this.showNotification('info', data.message);
  }

  showWarning = (data) => {
    this.showNotification('warning', data.message);
  }

  showError = (data) => {
    this.showNotification('error', data.message);
  }

  showNotification(type, message) {
    // Render notification UI
  }
}
```

Multiple Subscriptions Helper

Manage multiple subscriptions easily:

```
class SubscriptionManager {
  constructor() {
    this.subscriptions = [];
  }

  subscribe(topic, handler) {
    const unsubscribe = pan.subscribe(topic, handler);
    this.subscriptions.push(unsubscribe);
    return unsubscribe;
  }

  unsubscribeAll() {
    this.subscriptions.forEach(unsubscribe => unsubscribe());
    this.subscriptions = [];
  }
}

// Usage in component
class MyComponent extends HTMLElement {
  constructor() {
    super();
    this.subs = new SubscriptionManager();
  }

  connectedCallback() {
    this.subs.subscribe('user.login', this.handleLogin);
    this.subs.subscribe('user.logout', this.handleLogout);
    this.subs.subscribe('app.theme.changed', this.handleThemeChange);
  }

  disconnectedCallback() {
    this.subs.unsubscribeAll();
  }

  handleLogin = (data) => { /* ... */ }
  handleLogout = (data) => { /* ... */ }
  handleThemeChange = (data) => { /* ... */ }
}
```

Conditional Subscriptions

Subscribe only when conditions are met:

```
class UserDashboard extends HTMLElement {
  connectedCallback() {
    // Subscribe to user-specific events only when user is logged in
    this.unsubscribeAuth = pan.subscribe('auth.state.changed', ({ isAuthenticated, user }) =>
      if (isAuthenticated) {
        this.subscribeToUserEvents(user.id);
      } else {
        this.unsubscribeFromUserEvents();
      }
    });
  }

  subscribeToUserEvents(userId) {
    this.unsubscribeUserActivity = pan.subscribe('user.activity', (data) => {
      if (data.userId === userId) {
        this.updateActivity(data);
      }
    });

    this.unsubscribeUserNotifications = pan.subscribe('user.notifications', (data) => {
      if (data.userId === userId) {
        this.showNotification(data);
      }
    });
  }

  unsubscribeFromUserEvents() {
    if (this.unsubscribeUserActivity) {
      this.unsubscribeUserActivity();
      this.unsubscribeUserActivity = null;
    }

    if (this.unsubscribeUserNotifications) {
      this.unsubscribeUserNotifications();
      this.unsubscribeUserNotifications = null;
    }
  }
```

```
}
```

Filtering Events

Filter events in the subscriber:

```
pan.subscribe('task.updated', (task) => {
  // Only handle tasks assigned to current user
  if (task.assignedTo === currentUser.id) {
    this.updateTaskDisplay(task);
  }
});

pan.subscribe('notification.*', (notification) => {
  // Only show high-priority notifications
  if (notification.priority >= 3) {
    this.showNotification(notification);
  }
});
```

Message Patterns

The PAN bus supports several messaging patterns for different use cases.

1. Fire and Forget

Most common pattern. Publish and continue without waiting:

```
// Publisher
function saveSettings(settings) {
  localStorage.setItem('settings', JSON.stringify(settings));
  pan.publish('settings.saved', settings);
}

// Subscriber
pan.subscribe('settings.saved', (settings) => {
  console.log('Settings updated:', settings);
  updateUI(settings);
});
```

Use when: - Multiple components may react - You don't need confirmation - Action is non-critical

2. Request/Response

Request data and wait for a response:

```
// Responder
pan.respond('auth.token.get', async () => {
  return localStorage.getItem('authToken');
});
```

```
// Requester
const token = await pan.request('auth.token.get');
console.log('Token:', token);
```

Implementation:

```
// In PAN library
class PAN {
  request(topic, data, timeout = 5000) {
    return new Promise((resolve, reject) => {
      const responseId = `${topic}:${Date.now()}:${Math.random()}`;

      // Subscribe to response
      const unsubscribe = this.subscribe(`${topic}:response:${responseId}`, (response) => {
        unsubscribe();
        clearTimeout(timer);
        resolve(response);
      });

      // Set timeout
      const timer = setTimeout(() => {
        unsubscribe();
        reject(new Error(`Request timeout: ${topic}`));
      }, timeout);

      // Publish request
      this.publish(`${topic}:request`, {
        ...data,
        _responseId: responseId
      });
    });
  }

  respond(topic, handler) {
    return this.subscribe(`${topic}:request`, async (data) => {
      try {
        const result = await handler(data);
        this.publish(`${topic}:response:${data._responseId}`, result);
      } catch (error) {
        this.publish(`${topic}:response:${data._responseId}`, {
          error: error.message
        });
      }
    });
  }
}
```

Use when: - Need data from another component - Waiting for response is acceptable - Asynchronous operations

3. Command Pattern

Issue commands that components execute:

```
// Command issuer
pan.publish('modal.open', {
  component: 'user-profile',
  props: { userId: 123 }
});

// Command handler
pan.subscribe('modal.open', ({ component, props }) => {
  const modal = document.createElement('app-modal');
  modal.component = component;
  modal.props = props;
  document.body.appendChild(modal);
});
```

Use when: - Triggering actions in other components - Implementing undo/redo - Building command palette UIs

4. Event Sourcing

Store events for replay or auditing:

```
const eventStore = [];

// Store all events
pan.subscribe('*', (topic, data) => {
  eventStore.push({
    topic,
    data,
    timestamp: Date.now()
  });
});

// Replay events
function replayEvents(fromTimestamp) {
  eventStore
    .filter(event => event.timestamp >= fromTimestamp)
    .forEach(event => {
      pan.publish(event.topic, event.data);
    });
}

// Get events for debugging
function getEventHistory(topic) {
  return eventStore.filter(event =>
    event.topic === topic || event.topic.startsWith(topic + '.')
  );
}
```

Use when: - Debugging complex interactions - Implementing undo/redo - Auditing user actions - Syncing state across sessions

5. Aggregation Pattern

Collect multiple events before acting:

```
class DataAggregator extends HTMLElement {
  constructor() {
    super();
    this.pendingUpdates = new Set();
    this.debounceTimer = null;
  }

  connectedCallback() {
    pan.subscribe('data.item.updated', ({ id }) => {
      this.pendingUpdates.add(id);
      this.scheduleRefresh();
    });
  }

  scheduleRefresh() {
    clearTimeout(this.debounceTimer);

    this.debounceTimer = setTimeout(() => {
      this.refreshItems(Array.from(this.pendingUpdates));
      this.pendingUpdates.clear();
    }, 500);
  }

  async refreshItems(ids) {
    const items = await fetchItems(ids);
    this.render(items);
  }
}
```

Use when: - Avoiding excessive updates - Batching API requests - Debouncing rapid events

6. Saga Pattern

Coordinate multi-step processes:

```
class CheckoutSaga {
  constructor() {
    this.setupListeners();
  }
  setupListeners() {
    pan.subscribe('checkout.started', this.handleCheckoutStart);
    pan.subscribe('payment.completed', this.handlePaymentComplete);
    pan.subscribe('order.created', this.handleOrderCreated);
  }

  handleCheckoutStart = async ({ cart }) => {
    try {
      // Step 1: Validate cart
      pan.publish('checkout.validating', { cart });
      await this.validateCart(cart);

      // Step 2: Calculate totals
      pan.publish('checkout.calculating', { cart });
      const totals = await this.calculateTotals(cart);

      // Step 3: Request payment
      pan.publish('payment.requested', { totals });
    } catch (error) {
      pan.publish('checkout.failed', { error: error.message });
    }
  }

  handlePaymentComplete = async ({ paymentId, totals }) => {
    try {
      // Step 4: Create order
      pan.publish('order.creating', { paymentId });
      const order = await this.createOrder(paymentId, totals);

      pan.publish('order.created', { order });
    } catch (error) {
      // Compensating transaction: refund payment
      pan.publish('payment.refund-requested', { paymentId });
      pan.publish('checkout.failed', { error: error.message });
    }
  }

  handleOrderCreated = async ({ order }) => {
    // Step 5: Send confirmation
    pan.publish('order.confirmation-sending', { order });
    await this.sendConfirmation(order);

    // Step 6: Complete checkout
    pan.publish('checkout.completed', { order });
  }
}
```

Use when: - Complex multi-step workflows - Need to handle failures and rollbacks - Coordinating multiple services

Debugging PAN Communication

Debugging event-driven systems requires different techniques than traditional debugging.

Logging All Events

```
// Enable debug mode
pan.debug(true);

// Or manually subscribe to all events
pan.subscribe('*', (topic, data) => {
  console.group(`[PAN] ${topic}`);
  console.log('Data:', data);
  console.log('Timestamp:', new Date().toISOString());
  console.trace('Stack trace');
  console.groupEnd();
});
```

Event Inspector

Build a visual event inspector:

```
class PanInspector extends HTMLElement {
  constructor() {
    super();
    this.events = [];
    this.maxEvents = 100;
  }

  connectedCallback() {
    this.render();

    pan.subscribe('*', (topic, data) => {
      this.logEvent(topic, data);
    });
  }

  logEvent(topic, data) {
    this.events.unshift({
      topic,
      data,
      timestamp: Date.now()
    });

    if (this.events.length > this.maxEvents) {
      this.events.pop();
    }

    this.render();
  }

  render() {
    this.innerHTML = `
      <style>
        .pan-inspector {
          position: fixed;
          bottom: 0;
          right: 0;
          width: 400px;
          height: 300px;
          background: white;
          border: 1px solid #ccc;
          overflow: auto;
          font-family: monospace;
          font-size: 12px;
        }

        .event {
          padding: 8px;
          border-bottom: 1px solid #eee;
        }

        .event:hover {
          background: #f5f5f5;
        }

        .topic {
          font-weight: bold;
          color: #667eea;
        }

        .timestamp {
          color: #999;
          font-size: 10px;
        }

        .data {
          margin-top: 4px;
          color: #333;
        }
      </style>

      <div class="pan-inspector">
```

```
        <h3>PAN Event Inspector</h3>
        ${this.events.map(event => `
          <div class="event">
            <div class="topic">${event.topic}</div>
            <div class="timestamp">${new Date(event.timestamp).toLocaleTimeString()}</div>
            <div class="data">${JSON.stringify(event.data, null, 2)}</div>
          </div>
        `).join('')}
      </div>
    `;
  }
}

customElements.define('pan-inspector', PanInspector);
```

Event Filtering

Filter events for specific topics:

```
function filterEvents(pattern) {
  const regex = new RegExp(pattern.replace('*', '.*'));

  pan.subscribe('*', (topic, data) => {
    if (regex.test(topic)) {
      console.log(`[FILTERED] ${topic}:`, data);
    }
  });
}

// Usage
filterEvents('user.*');      // Only user events
filterEvents('*.error');     // All error events
filterEvents('cart|order'); // Cart or order events
```

Performance Monitoring

Track event frequency and performance:

```
class PanMonitor {
  constructor() {
    this.stats = new Map();

    pan.subscribe('*', (topic) => {
      const stat = this.stats.get(topic) || { count: 0, timestamps: [] };

      stat.count++;
      stat.timestamps.push(Date.now());

      // Keep only last 100 timestamps
      if (stat.timestamps.length > 100) {
        stat.timestamps.shift();
      }

      this.stats.set(topic, stat);
    });
  }

  getStats(topic) {
    const stat = this.stats.get(topic);
    if (!stat) return null;

    const timestamps = stat.timestamps;
    const duration = timestamps[timestamps.length - 1] - timestamps[0];
    const frequency = timestamps.length / (duration / 1000);

    return {
      topic,
      count: stat.count,
      frequency: frequency.toFixed(2) + ' events/sec',
      lastEvent: new Date(timestamps[timestamps.length - 1])
    };
  }

  getAllStats() {
    const results = [];

    this.stats.forEach((_, topic) => {
      results.push(this.getStats(topic));
    });

    return results.sort((a, b) => b.count - a.count);
  }

  reset() {
    this.stats.clear();
  }
}

// Usage
const monitor = new PanMonitor();

// Later, check stats
console.table(monitor.getAllStats());
```

Event Replay

Capture and replay events for testing:

```
class EventRecorder {
  constructor() {
    this.recording = false;
    this.events = [];
  }

  start() {
    this.recording = true;
    this.events = [];

    this.unsubscribe = pan.subscribe('*', (topic, data) => {
      if (this.recording) {
        this.events.push({ topic, data, timestamp: Date.now() });
      }
    });
  }

  stop() {
    this.recording = false;
    if (this.unsubscribe) {
      this.unsubscribe();
    }

    return this.events;
  }

  replay(events, speed = 1) {
    if (!events || events.length === 0) return;

    const startTime = events[0].timestamp;

    events.forEach((event, index) => {
      const delay = (event.timestamp - startTime) / speed;

      setTimeout(() => {
        pan.publish(event.topic, event.data);
      }, delay);
    });
  }

  save(name) {
    localStorage.setItem(`pan-recording-${name}`, JSON.stringify(this.events));
  }

  load(name) {
    const data = localStorage.getItem(`pan-recording-${name}`);
    return data ? JSON.parse(data) : null;
  }
}

// Usage
const recorder = new EventRecorder();

// Start recording
recorder.start();

// ... perform actions ...

// Stop and save
const events = recorder.stop();
recorder.save('my-test-scenario');

// Later, replay
const events = recorder.load('my-test-scenario');
recorder.replay(events, 2); // 2x speed
```

Summary

This chapter covered:

- **Pub/Sub Architecture**: Decoupled communication via topics
- **Topics and Namespaces**: Organizing events with hierarchical naming
- **Publishing**: Sending messages and event patterns
- **Subscribing**: Receiving and filtering events
- **Message Patterns**: Fire-and-forget, request/response, commands, sagas
- **Debugging**: Logging, inspection, monitoring, and replay tools

The PAN bus is central to LARC applications. Mastering it enables you to build scalable, maintainable applications where components collaborate without tight coupling.

Best Practices

1. **Use descriptive topic names**
   - `user.profile.updated` not `userUpdated`
   - Past tense for events that happened
   - Include context in message data
2. **Clean up subscriptions**
   - Always unsubscribe in `disconnectedCallback`
   - Use subscription managers for multiple subscriptions
   - Avoid memory leaks
3. **Avoid infinite loops**
   - Don't publish the same event you're subscribed to
   - Use different topics for input and output
   - Add loop detection in debug mode

4. **Keep handlers fast**
    - Don't block the event loop
    - Use async/await for long operations
    - Consider debouncing rapid events
5. **Include metadata**
    - Timestamp, source, user ID for debugging
    - Request IDs for tracing
    - Error details for failures
6. **Test event flows**
    - Use event recorders for integration tests
    - Mock pan.publish/subscribe in unit tests
    - Verify event contracts between components

Chapter 6: State Management

State management is one of the most critical aspects of application development. Poor state management leads to bugs, performance issues, and maintenance nightmares. Good state management makes applications predictable, testable, and maintainable.

LARC takes a pragmatic approach: start simple and scale complexity only when needed. This chapter explores state management at every level, from component-local state to distributed, offline-first architectures.

Component-Local State

The simplest form of state lives entirely within a single component. This is your first choice for most scenarios.

Instance Properties

Use instance properties for component-specific state:

```
class ToggleSwitch extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });

    // Local state
    this.isOn = false;
  }

  connectedCallback() {
    this.render();

    this.shadowRoot.querySelector('button').addEventListener('click', () => {
      this.isOn = !this.isOn;  // Update state
      this.render();            // Re-render

      // Notify others
      this.dispatchEvent(new CustomEvent('toggle', {
        detail: { isOn: this.isOn }
      }));
    });
  }

  render() {
    this.shadowRoot.innerHTML = `
      <style>
        button {
          background: ${this.isOn ? '#48bb78' : '#cbd5e0'};
          color: white;
          border: none;
          padding: 8px 16px;
          border-radius: 4px;
          cursor: pointer;
        }
      </style>
      <button>${this.isOn ? 'ON' : 'OFF'}</button>
    `;
  }
}
```

When to use: - UI state (expanded/collapsed, selected, etc.) - Temporary values (search input, form drafts) - Component-specific configuration

Advantages: - Simple and straightforward - No dependencies on external state - Easy to reason about - Easy to test

Private Fields

Use private fields (with `#`) for true encapsulation:

```
class Counter extends HTMLElement {
  // Private fields
  #count = 0;
  #max = 100;
  #min = 0;

  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  // Public getter
  get count() {
    return this.#count;
  }

  // Public setter with validation
  set count(value) {
    const newCount = Number(value);
    if (isNaN(newCount)) {
      throw new Error('Count must be a number');
    }

    if (newCount < this.#min || newCount > this.#max) {
      throw new Error(`Count must be between ${this.#min} and ${this.#max}`);
    }

    this.#count = newCount;
    this.render();
  }

  increment() {
    this.count = Math.min(this.#count + 1, this.#max);
  }

  decrement() {
    this.count = Math.max(this.#count - 1, this.#min);
  }

  render() {
    this.shadowRoot.innerHTML = `
      <div>${this.#count}</div>
    `;
  }
}
```

Benefits: - True privacy (can't access from outside) - Validation at setter boundaries - Clear public API

State Objects

Organize related state in objects:

```
class UserProfile extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });

    // Group related state
    this.state = {
      user: null,
      loading: false,
      error: null,
      editMode: false
    };
  }

  setState(updates) {
    // Merge updates into state
    this.state = {
      ...this.state,
      ...updates
    };

    this.render();
  }

  async loadUser(userId) {
    this.setState({ loading: true, error: null });

    try {
      const response = await fetch(`/api/users/${userId}`);
      const user = await response.json();

      this.setState({ user, loading: false });
    } catch (error) {
      this.setState({ error: error.message, loading: false });
    }
  }

  render() {
    const { user, loading, error, editMode } = this.state;

    if (loading) {
      this.shadowRoot.innerHTML = '<div>Loading...</div>';
    } else if (error) {
      this.shadowRoot.innerHTML = `<div class="error">${error}</div>`;
    } else if (user) {
      this.shadowRoot.innerHTML = `
        <div>
          <h2>${user.name}</h2>
          ${editMode ? this.renderEditForm() : this.renderDisplay()}
        </div>
      `;
    }
  }
}
```

Benefits: - Organized state structure - Single method to update state - Clear state shape - Easier debugging (log entire state)

Shared State Patterns

When multiple components need access to the same data, you need shared state.

Simple Global State

Create a shared state object:

```
// lib/state.js
export const appState = {
  user: null,
  theme: 'light',
  language: 'en',
  notifications: []
};

// Update state
export function updateState(updates) {
  Object.assign(appState, updates);
  pan.publish('app.state.changed', appState);
}

// Get state
export function getState() {
  return { ...appState };
}
```

Usage in components:

```
import { appState, updateState } from '../lib/state.js';

class ThemeSwitcher extends HTMLElement {
  connectedCallback() {
    // Read initial state
    this.render(appState.theme);

    // Subscribe to changes
    this.unsubscribe = pan.subscribe('app.state.changed', (state) => {
      this.render(state.theme);
    });

    // Add event listener
    this.addEventListener('click', () => {
      const newTheme = appState.theme === 'light' ? 'dark' : 'light';
      updateState({ theme: newTheme });
    });
  }

  disconnectedCallback() {
    this.unsubscribe();
  }

  render(theme) {
    this.textContent = `Theme: ${theme}`;
  }
}
```

Reactive State with Proxy

Make state changes automatically trigger updates:

```
// lib/reactive-state.js
export function createReactiveState(initialState) {
  const listeners = new Set();

  const state = new Proxy(initialState, {
    set(target, property, value) {
      const oldValue = target[property];
      target[property] = value;

      // Notify listeners
      listeners.forEach(listener => {
        listener(property, value, oldValue);
      });

      // Also publish via PAN
      pan.publish('state.changed', {
        property,
        value,
        oldValue
      });

      return true;
    },

    get(target, property) {
      return target[property];
    }
  });

  return {
    state,
    subscribe(listener) {
      listeners.add(listener);
      return () => listeners.delete(listener);
    },
    getState() {
      return { ...state };
    }
  };
}
```

Usage:

```
// Create reactive state
const { state, subscribe } = createReactiveState({
  count: 0,
  user: null,
  theme: 'light'
});

// Components automatically react to changes
class CountDisplay extends HTMLElement {
  connectedCallback() {
    // Subscribe to specific property changes
    this.unsubscribe = subscribe((property, value) => {
      if (property === 'count') {
        this.textContent = `Count: ${value}`;
      }
    });

    // Initial render
    this.textContent = `Count: ${state.count}`;
  }

  disconnectedCallback() {
    this.unsubscribe();
  }
}

// Update state (automatically triggers updates)
state.count++;  // All subscribers notified
state.count = 42;  // All subscribers notified
```

Store Pattern

Build a more sophisticated store:

```
// lib/store.js
class Store {
  constructor(initialState = {}) {
    this.state = initialState;
    this.listeners = new Map();
    this.middleware = [];
  }

  getState() {
    return { ...this.state };
  }

  setState(updates) {
    const oldState = { ...this.state };
    this.state = { ...this.state, ...updates };

    // Run middleware
    this.middleware.forEach(fn => fn(this.state, oldState));

    // Notify listeners
    this.listeners.forEach((listeners, key) => {
      if (key === '*' || key in updates) {
        listeners.forEach(listener => {
          listener(this.state, oldState);
        });
      }
    });
  }

  subscribe(key, listener) {
    if (!this.listeners.has(key)) {
      this.listeners.set(key, new Set());
    }

    this.listeners.get(key).add(listener);

    // Return unsubscribe function
    return () => {
      const listeners = this.listeners.get(key);
      if (listeners) {
        listeners.delete(listener);
      }
    };
  }

  use(middleware) {
    this.middleware.push(middleware);
  }

  dispatch(action) {
    // Action pattern: { type, payload }
    switch (action.type) {
      case 'user/login':
        this.setState({ user: action.payload });
        break;
      case 'user/logout':
        this.setState({ user: null });
        break;
      case 'theme/change':
        this.setState({ theme: action.payload });
        break;
      default:
        console.warn(`Unknown action: ${action.type}`);
    }
  }
}

// Create store instance
export const store = new Store({
  user: null,
  theme: 'light',
  notifications: []
});

// Add logging middleware
store.use((state, oldState) => {
  console.log('State changed:', { old: oldState, new: state });
});

// Add persistence middleware
store.use((state) => {
  localStorage.setItem('app-state', JSON.stringify(state));
});
```

Usage:

```
import { store } from '../lib/store.js';

class UserMenu extends HTMLElement {
  connectedCallback() {
    // Subscribe to user changes only
    this.unsubscribe = store.subscribe('user', (state) => {
      this.render(state.user);
    });

    // Initial render
    this.render(store.getState().user);
  }

  disconnectedCallback() {
    this.unsubscribe();
  }

  render(user) {
    if (user) {
      this.innerHTML = `
        <div>Hello, ${user.name}</div>
        <button id="logout">Logout</button>
      `;
      this.querySelector('#logout').addEventListener('click', () => {
        store.dispatch({ type: 'user/logout' });
      });
    } else {
      this.innerHTML = '<button id="login">Login</button>';

      this.querySelector('#login').addEventListener('click', () => {
        // Trigger login flow
        pan.publish('auth.login.requested');
      });
    }
  }
}
```

The pan-store Component

LARC provides a built-in component for state management:

```
<pan-store id="app-store" persist="true">
  <!-- Initial state -->
  <script type="application/json">
  {
    "user": null,
    "theme": "light",
    "cart": {
      "items": [],
      "total": 0
    }
  }
  </script>
</pan-store>

<script type="module">
  const store = document.getElementById('app-store');

  // Get state
  const state = store.getState();

  // Update state
  store.setState({ theme: 'dark' });

  // Subscribe to changes
  store.addEventListener('state-changed', (e) => {
    console.log('State changed:', e.detail);
  });

  // Or use PAN bus
  pan.subscribe('store.changed', (state) => {
    console.log('State via PAN:', state);
  });
</script>
```

Features: - Declarative state initialization - Optional persistence to localStorage - Integrates with PAN bus - Supports nested state updates - Time-travel debugging in dev mode

Advanced usage:

```
// Get nested state
const cartItems = store.getState('cart.items');

// Update nested state
store.setState('cart.items', [...items, newItem]);

// Subscribe to specific paths
store.subscribe('cart.total', (value) => {
  console.log('Cart total changed:', value);
});

// Computed properties
store.computed('cart.itemCount', (state) => {
  return state.cart.items.length;
});

// Actions
store.action('addToCart', (item) => {
  const cart = store.getState('cart');
  const items = [...cart.items, item];
  const total = items.reduce((sum, item) => sum + item.price, 0);

  store.setState({
    'cart.items': items,
    'cart.total': total
  });
});

// Use action
store.dispatch('addToCart', { id: 1, name: 'Product', price: 29.99 });
```

IndexedDB Integration

For large datasets or offline capability, use IndexedDB:

Basic IndexedDB Wrapper

```
// lib/db.js
class Database {
  constructor(name, version = 1) {
    this.name = name;
    this.version = version;
    this.db = null;
  }

  async open(stores) {
    return new Promise((resolve, reject) => {
      const request = indexedDB.open(this.name, this.version);

      request.onerror = () => reject(request.error);
      request.onsuccess = () => {
        this.db = request.result;
        resolve(this.db);
      };

      request.onupgradeneeded = (event) => {
        const db = event.target.result;

        stores.forEach(({ name, keyPath, indexes }) => {
          if (!db.objectStoreNames.contains(name)) {
            const store = db.createObjectStore(name, { keyPath });

            indexes?.forEach(({ name, keyPath, options }) => {
              store.createIndex(name, keyPath, options);
            });
          }
        });
      };
    });
  }

  async add(storeName, data) {
    const tx = this.db.transaction(storeName, 'readwrite');
    const store = tx.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.add(data);
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async get(storeName, key) {
    const tx = this.db.transaction(storeName, 'readonly');
    const store = tx.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.get(key);
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async getAll(storeName) {
    const tx = this.db.transaction(storeName, 'readonly');
    const store = tx.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.getAll();
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async update(storeName, data) {
    const tx = this.db.transaction(storeName, 'readwrite');
    const store = tx.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.put(data);
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async delete(storeName, key) {
    const tx = this.db.transaction(storeName, 'readwrite');
    const store = tx.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.delete(key);
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async clear(storeName) {
    const tx = this.db.transaction(storeName, 'readwrite');
    const store = tx.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.clear();
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }
```

```
}

// Initialize database
export const db = new Database('MyApp', 1);

await db.open([
  {
    name: 'todos',
    keyPath: 'id',
    indexes: [
      { name: 'by-status', keyPath: 'status' },
      { name: 'by-created', keyPath: 'createdAt' }
    ]
  },
  {
    name: 'users',
    keyPath: 'id'
  }
]);
```

Usage:

```
import { db } from '../lib/db.js';

class TodoList extends HTMLElement {
  async connectedCallback() {
    // Load todos from IndexedDB
    this.todos = await db.getAll('todos');
    this.render();

    // Subscribe to changes
    pan.subscribe('todo.added', async ({ todo }) => {
      await db.add('todos', todo);
      this.todos = await db.getAll('todos');
      this.render();
    });

    pan.subscribe('todo.updated', async ({ todo }) => {
      await db.update('todos', todo);
      this.todos = await db.getAll('todos');
      this.render();
    });

    pan.subscribe('todo.deleted', async ({ id }) => {
      await db.delete('todos', id);
      this.todos = await db.getAll('todos');
      this.render();
    });
  }
  render() {
    this.innerHTML = `
      <ul>
        ${this.todos.map(todo => `
          <li>
            <span>${todo.text}</span>
            <button data-id="${todo.id}">Delete</button>
          </li>
        `).join('')}
      </ul>
    `;
  }
}
```

Cache-First Strategy

Implement cache-first data loading:

```
class DataManager {
  constructor(storeName) {
    this.storeName = storeName;
    this.cache = new Map();
  }

  async get(id) {
    // 1. Check memory cache
    if (this.cache.has(id)) {
      return this.cache.get(id);
    }

    // 2. Check IndexedDB
    const cached = await db.get(this.storeName, id);
    if (cached) {
      this.cache.set(id, cached);
      return cached;
    }

    // 3. Fetch from API
    const data = await this.fetchFromAPI(id);

    // 4. Store in cache and IndexedDB
    this.cache.set(id, data);
    await db.add(this.storeName, data);

    return data;
  }
  async fetchFromAPI(id) {
    const response = await fetch(`/api/${this.storeName}/${id}`);
    return response.json();
  }

  async refresh(id) {
    // Force refresh from API
    const data = await this.fetchFromAPI(id);

    // Update cache and IndexedDB
    this.cache.set(id, data);
    await db.update(this.storeName, data);

    return data;
  }

  async getAll() {
    // Load from IndexedDB first
    const items = await db.getAll(this.storeName);

    // Cache in memory
    items.forEach(item => {
      this.cache.set(item.id, item);
    });

    return items;
  }
}

// Usage
const userManager = new DataManager('users');

// Always returns fast (from cache if available)
const user = await userManager.get(123);

// Force refresh
const freshUser = await userManager.refresh(123);
```

Persistence Strategies

localStorage

Simple key-value storage:

```
class PersistentState {
  constructor(key) {
    this.key = key;
    this.state = this.load();
  }

  load() {
    try {
      const data = localStorage.getItem(this.key);
      return data ? JSON.parse(data) : {};
    } catch (error) {
      console.error('Failed to load state:', error);
      return {};
    }
  }

  save() {
    try {
      localStorage.setItem(this.key, JSON.stringify(this.state));
    } catch (error) {
      console.error('Failed to save state:', error);
    }
  }

  get(path) {
    return this.getNestedValue(this.state, path);
  }

  set(path, value) {
    this.setNestedValue(this.state, path, value);
    this.save();
  }

  getNestedValue(obj, path) {
    return path.split('.').reduce((current, key) => current?.[key], obj);
  }

  setNestedValue(obj, path, value) {
    const keys = path.split('.');
    const lastKey = keys.pop();
    const target = keys.reduce((current, key) => {
      if (!(key in current)) current[key] = {};
      return current[key];
    }, obj);
    target[lastKey] = value;
  }

  clear() {
    this.state = {};
    localStorage.removeItem(this.key);
  }
}
```

```
// Usage
const settings = new PersistentState('app-settings');

settings.set('theme', 'dark');
settings.set('user.preferences.notifications', true);

console.log(settings.get('theme'));  // 'dark'
console.log(settings.get('user.preferences.notifications'));  // true
```

sessionStorage

For temporary session data:

```
class SessionState {
  constructor(key) {
    this.key = key;
  }

  set(data) {
    sessionStorage.setItem(this.key, JSON.stringify(data));
  }

  get() {
    const data = sessionStorage.getItem(this.key);
    return data ? JSON.parse(data) : null;
  }

  clear() {
    sessionStorage.removeItem(this.key);
  }
}

// Usage - data persists only for the session
const sessionData = new SessionState('form-draft');

// Save form draft
sessionData.set({ email: 'user@example.com', message: 'Draft...' });

// Restore on page reload (same session)
const draft = sessionData.get();
```

Hybrid Strategy

Combine localStorage and IndexedDB:

```
class HybridStorage {
  constructor(namespace) {
    this.namespace = namespace;
  }

  async set(key, value) {
    const fullKey = `${this.namespace}:${key}`;

    // Store small data in localStorage
    if (this.isSmall(value)) {
      localStorage.setItem(fullKey, JSON.stringify(value));
    } else {
      // Store large data in IndexedDB
      await db.update('storage', { key: fullKey, value });
    }
  }

  async get(key) {
    const fullKey = `${this.namespace}:${key}`;

    // Try localStorage first
    const local = localStorage.getItem(fullKey);
    if (local) {
      return JSON.parse(local);
    }

    // Try IndexedDB
    const result = await db.get('storage', fullKey);
    return result?.value;
  }

  isSmall(value) {
    const str = JSON.stringify(value);
    return str.length < 1024 * 10; // 10KB threshold
  }

  async clear() {
    // Clear localStorage items
    Object.keys(localStorage).forEach(key => {
      if (key.startsWith(`${this.namespace}:`)) {
        localStorage.removeItem(key);
      }
    });

    // Clear IndexedDB items
    const all = await db.getAll('storage');
    for (const item of all) {
      if (item.key.startsWith(`${this.namespace}:`)) {
        await db.delete('storage', item.key);
      }
    }
  }
}
```

Offline-First Applications

Build applications that work without connectivity:

Service Worker + State Management

```
// sw.js - Service Worker
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('v1').then((cache) => {
      return cache.addAll([
        '/',
        '/index.html',
        '/src/app.js',
        '/src/components/',
        // Cache critical assets
      ]);
    })
  );
});

self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      // Return cached version or fetch
      return response || fetch(event.request);
    })
  );
});
```

Sync Queue

Queue operations when offline:

```
// lib/sync-queue.js
class SyncQueue {
  constructor() {
    this.queue = this.loadQueue();
    this.processing = false;

    // Listen for online events
    window.addEventListener('online', () => {
      this.process();
    });
    // Start processing if online
    if (navigator.onLine) {
      this.process();
    }
  }

  loadQueue() {
    const data = localStorage.getItem('sync-queue');
    return data ? JSON.parse(data) : [];
  }

  saveQueue() {
    localStorage.setItem('sync-queue', JSON.stringify(this.queue));
  }

  add(operation) {
    this.queue.push({
      id: Date.now() + Math.random(),
      operation,
      timestamp: Date.now(),
      attempts: 0
    });

    this.saveQueue();

    if (navigator.onLine) {
      this.process();
    }
  }

  async process() {
    if (this.processing || this.queue.length === 0) {
      return;
    }

    this.processing = true;

    while (this.queue.length > 0 && navigator.onLine) {
      const item = this.queue[0];

      try {
        await this.executeOperation(item.operation);

        // Success - remove from queue
        this.queue.shift();
        this.saveQueue();

        pan.publish('sync.success', { operation: item.operation });
      } catch (error) {
        item.attempts++;

        if (item.attempts >= 3) {
          // Max attempts - remove and report error
          this.queue.shift();
          this.saveQueue();

          pan.publish('sync.failed', {
            operation: item.operation,
            error: error.message
          });
        } else {
          // Retry later
          break;
        }
      }
    }

    this.processing = false;
  }

  async executeOperation(operation) {
    switch (operation.type) {
      case 'CREATE':
        return this.create(operation.data);
      case 'UPDATE':
        return this.update(operation.data);
      case 'DELETE':
        return this.delete(operation.id);
      default:
        throw new Error(`Unknown operation: ${operation.type}`);
    }
  }

  async create(data) {
    const response = await fetch('/api/items', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });

    if (!response.ok) throw new Error('Create failed');
    return response.json();
  }

  async update(data) {
    const response = await fetch(`/api/items/${data.id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });

    if (!response.ok) throw new Error('Update failed');
    return response.json();
  }

  async delete(id) {
    const response = await fetch(`/api/items/${id}`, {
      method: 'DELETE'
    });

    if (!response.ok) throw new Error('Delete failed');
  }

  clear() {
    this.queue = [];
    this.saveQueue();
  }

  getStatus() {
    return {
      queued: this.queue.length,
      online: navigator.onLine,
      processing: this.processing
    };
  }
}

export const syncQueue = new SyncQueue();
```

Usage:

```
import { syncQueue } from '../lib/sync-queue.js';

class TodoManager {
  async addTodo(text) {
    const todo = {
      id: Date.now(),
      text,
      completed: false,
      createdAt: new Date()
    };

    // Save locally immediately
    await db.add('todos', todo);
    pan.publish('todo.added', { todo });
    // Queue for server sync
    if (!navigator.onLine) {
      syncQueue.add({
        type: 'CREATE',
        data: todo
      });

      pan.publish('notification.info', {
        message: 'Saved locally. Will sync when online.'
      });
    } else {
      // Online - sync immediately
      try {
        await this.syncToServer(todo);
      } catch (error) {
        // Failed - add to queue
        syncQueue.add({
          type: 'CREATE',
          data: todo
        });
      }
    }
  }

  async syncToServer(todo) {
    const response = await fetch('/api/todos', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(todo)
    });

    if (!response.ok) {
      throw new Error('Sync failed');
    }

    const result = await response.json();

    // Update local copy with server ID
    await db.update('todos', { ...todo, serverId: result.id });
  }
}
```

Summary

This chapter covered state management at every level:

- **Component-Local State**: Instance properties, private fields, and state objects

- **Shared State**: Global state, reactive proxies, and store patterns
- **pan-store**: Built-in state management component
- **IndexedDB**: Large dataset storage and offline capability
- **Persistence**: localStorage, sessionStorage, and hybrid strategies
- **Offline-First**: Service workers, sync queues, and conflict resolution

Choose the simplest solution that meets your needs, then scale up complexity as requirements grow.

Best Practices

1. **Start with local state**
   - Only share state when necessary
   - Keeps components independent
   - Easier to test and debug
2. **Use IndexedDB for large data**
   - localStorage limited to ~5-10MB
   - IndexedDB can store gigabytes
   - Better performance for large datasets
3. **Implement cache-first strategies**
   - Load from cache immediately
   - Update from server in background
   - Show stale data rather than loading spinner
4. **Queue offline operations**
   - Don't lose user data
   - Sync when connection restored
   - Show sync status to user
5. **Test offline scenarios**
   - Use DevTools to simulate offline
   - Test sync queue behavior
   - Verify conflict resolution
6. **Monitor storage usage**
   - Check quota before storing
   - Clean up old data
   - Provide clear error messages when full

Chapter 7: Advanced Component Patterns

As your LARC applications grow, you'll encounter scenarios that require sophisticated component architectures. This chapter explores advanced patterns that enable code reuse, flexible composition, and optimal performance.

These patterns come from years of component-based development across frameworks. LARC implements them using web standards, making them portable and future-proof.

Compound Components

Compound components work together as a set, sharing implicit state. Think of HTML's `<select>` and `<option>` elements—they form a cohesive unit.

Basic Compound Component

```
// tabs.js - Container component
class TabGroup extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.activeTab = 0;
  }

  connectedCallback() {
    this.render();
    this.setupTabs();
  }

  setupTabs() {
    // Get all tab headers
    const headers = this.querySelectorAll('tab-header');
    headers.forEach((header, index) => {
      header.addEventListener('click', () => {
        this.activeTab = index;
        this.updateTabs();
      });
    });

    this.updateTabs();
  }

  updateTabs() {
    // Update headers
    const headers = this.querySelectorAll('tab-header');
    headers.forEach((header, index) => {
      header.active = index === this.activeTab;
    });

    // Update panels
    const panels = this.querySelectorAll('tab-panel');
    panels.forEach((panel, index) => {
      panel.active = index === this.activeTab;
    });
  }

  render() {
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
        }
        .headers {
          display: flex;
          border-bottom: 2px solid #e2e8f0;
        }
        .panels {
          padding: 16px 0;
        }
      </style>
      <div class="headers">
        <slot name="headers"></slot>
      </div>
      <div class="panels">
        <slot name="panels"></slot>
      </div>
    `;
  }
}

// tab-header.js
class TabHeader extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  set active(value) {
    this._active = value;
    this.render();
  }

  connectedCallback() {
    this.render();
  }

  render() {
    this.shadowRoot.innerHTML = `
      <style>
        button {
          padding: 12px 24px;
          border: none;
          background: ${this._active ? '#667eea' : 'transparent'};
          color: ${this._active ? 'white' : '#4a5568'};
          cursor: pointer;
          font-weight: ${this._active ? '600' : '400'};
          transition: all 0.2s;
        }
        button:hover {
          background: ${this._active ? '#5a67d8' : '#f7fafc'};
        }
      </style>
      <button><slot></slot></button>
    `;
  }
}

// tab-panel.js
class TabPanel extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  set active(value) {
    this._active = value;
    this.style.display = value ? 'block' : 'none';
  }

  connectedCallback() {
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
        }
      </style>
      <slot></slot>
    `;
  }
}

customElements.define('tab-group', TabGroup);
customElements.define('tab-header', TabHeader);
customElements.define('tab-panel', TabPanel);
```

Usage:

```
<tab-group>
  <tab-header slot="headers">Profile</tab-header>
  <tab-header slot="headers">Settings</tab-header>
  <tab-header slot="headers">Billing</tab-header>

  <tab-panel slot="panels">
    <h2>Profile Content</h2>
    <p>User profile information...</p>
  </tab-panel>

  <tab-panel slot="panels">
    <h2>Settings Content</h2>
    <p>Application settings...</p>
  </tab-panel>

  <tab-panel slot="panels">
    <h2>Billing Content</h2>
    <p>Billing information...</p>
  </tab-panel>
</tab-group>
```

Context API for Compound Components

Share state without prop drilling:

```
// lib/context.js
const contexts = new WeakMap();

export function createContext(defaultValue) {
  return {
    Provider: class extends HTMLElement {
      constructor() {
        super();
        this.value = defaultValue;
        contexts.set(this, this.value);
      }

      provide(value) {
        this.value = value;
        contexts.set(this, value);
        this.notifyConsumers();
      }

      notifyConsumers() {
        const consumers = this.querySelectorAll('[data-context-consumer]');
        consumers.forEach(consumer => {
          if (consumer.onContextChange) {
            consumer.onContextChange(this.value);
          }
        });
      }

      connectedCallback() {
        this.innerHTML = `<slot></slot>`;
      }
    },

    Consumer: class extends HTMLElement {
      connectedCallback() {
        this.setAttribute('data-context-consumer', '');

        // Find provider up the tree
        let provider = this.closest('[data-context-provider]');
        if (provider && contexts.has(provider)) {
          this.onContextChange(contexts.get(provider));
        }
      }

      onContextChange(value) {
        // Override in subclasses
      }
    }
  };
}
```

Usage:

```
// Create context
const ThemeContext = createContext({ theme: 'light' });

// Provider component
class ThemeProvider extends ThemeContext.Provider {
  connectedCallback() {
    super.connectedCallback();
    this.setAttribute('data-context-provider', '');

    this.provide({
      theme: 'light',
      toggleTheme: () => {
        const newTheme = this.value.theme === 'light' ? 'dark' : 'light';
        this.provide({ ...this.value, theme: newTheme });
      }
    });
  }
}

// Consumer component
class ThemedButton extends ThemeContext.Consumer {
  onContextChange(context) {
    this.context = context;
    this.render();
  }

  render() {
    const { theme } = this.context || { theme: 'light' };

    this.innerHTML = `
      <button style="
        background: ${theme === 'dark' ? '#333' : '#fff'};
        color: ${theme === 'dark' ? '#fff' : '#333'};
      ">
        <slot></slot>
      </button>
    `;
  }
}

customElements.define('theme-provider', ThemeProvider);
customElements.define('themed-button', ThemedButton);
```

```
<theme-provider>
  <themed-button>Light/Dark</themed-button>
  <themed-button>Another Button</themed-button>
</theme-provider>
```

Higher-Order Components

Higher-order components (HOCs) wrap other components to add functionality.

Mixin Pattern

JavaScript mixins add functionality to classes:

```
// mixins/observable.js
export const ObservableMixin = (Base) => class extends Base {
  constructor() {
    super();
    this._observers = new Map();
  }

  observe(property, callback) {
    if (!this._observers.has(property)) {
      this._observers.set(property, new Set());
    }
    this._observers.get(property).add(callback);

    // Return unobserve function
    return () => {
      this._observers.get(property)?.delete(callback);
    };
  }

  notify(property, value) {
    this._observers.get(property)?.forEach(callback => {
      callback(value);
    });
  }

  set(property, value) {
    this[`_${property}`] = value;
    this.notify(property, value);
  }

  get(property) {
    return this[`_${property}`];
  }
};

// mixins/resizable.js
export const ResizableMixin = (Base) => class extends Base {
  connectedCallback() {
    super.connectedCallback?.();

    this.resizeObserver = new ResizeObserver((entries) => {
      for (const entry of entries) {
        this.onResize?.(entry.contentRect);
      }
    });

    this.resizeObserver.observe(this);
  }
  disconnectedCallback() {
    super.disconnectedCallback?.();
    this.resizeObserver?.disconnect();
  }
};

// mixins/loading.js
export const LoadingMixin = (Base) => class extends Base {
  constructor() {
    super();
    this._loading = false;
  }

  startLoading() {
    this._loading = true;
    this.setAttribute('loading', '');
    this.onLoadingChange?.(true);
  }

  stopLoading() {
    this._loading = false;
    this.removeAttribute('loading');
    this.onLoadingChange?.(false);
  }

  get loading() {
    return this._loading;
  }
};
```

Usage:

```
import { ObservableMixin } from './mixins/observable.js';
import { ResizableMixin } from './mixins/resizable.js';
import { LoadingMixin } from './mixins/loading.js';

// Compose multiple mixins
class DataTable extends LoadingMixin(ResizableMixin(ObservableMixin(HTMLElement))) {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  async connectedCallback() {
    super.connectedCallback();

    // Use Observable mixin
    this.observe('data', (data) => {
      console.log('Data changed:', data);
      this.render();
    });

    // Use Loading mixin
    this.startLoading();
    const data = await this.fetchData();
    this.set('data', data);
    this.stopLoading();
  }

  // Use Resizable mixin
  onResize(rect) {
    console.log('Component resized:', rect.width, rect.height);
    this.updateLayout();
  }

  onLoadingChange(loading) {
    this.render();
  }

  async fetchData() {
    const response = await fetch('/api/data');
    return response.json();
  }

  render() {
    // Render based on state
  }
}

customElements.define('data-table', DataTable);
```

Decorator Pattern

Wrap components to enhance them:

```
// decorators/with-loading.js
export function withLoading(ComponentClass) {
  return class extends ComponentClass {
    constructor() {
      super();
      this._originalConnectedCallback = this.connectedCallback;
    }

    connectedCallback() {
      // Inject loading overlay
      const loadingOverlay = document.createElement('div');
      loadingOverlay.className = 'loading-overlay';
      loadingOverlay.style.cssText = `
        position: absolute;
        top: 0;
        left: 0;
        right: 0;
        bottom: 0;
        background: rgba(255,255,255,0.8);
        display: none;
        align-items: center;
        justify-content: center;
      `;
      loadingOverlay.innerHTML = '<div class="spinner"></div>';

      this.appendChild(loadingOverlay);
      this._loadingOverlay = loadingOverlay;

      // Call original
      if (this._originalConnectedCallback) {
        this._originalConnectedCallback.call(this);
      }
    }

    showLoading() {
      if (this._loadingOverlay) {
        this._loadingOverlay.style.display = 'flex';
      }
    }

    hideLoading() {
      if (this._loadingOverlay) {
        this._loadingOverlay.style.display = 'none';
      }
    }
  };
}

// Usage
class UserProfile extends HTMLElement {
  async connectedCallback() {
    this.showLoading();

    const user = await fetch('/api/user').then(r => r.json());
    this.render(user);

    this.hideLoading();
  }
  render(user) {
    this.innerHTML = `<h1>${user.name}</h1>`;
  }
}

// Apply decorator
const UserProfileWithLoading = withLoading(UserProfile);
customElements.define('user-profile', UserProfileWithLoading);
```

Component Composition

Build complex UIs from simple, focused components.

Container/Presentational Pattern

Separate logic from presentation:

```
// Presentational - no logic, just rendering
class UserCard extends HTMLElement {
  set user(value) {
    this._user = value;
    this.render();
  }

  render() {
    if (!this._user) return;

    this.innerHTML = `
      <div class="card">
        <img src="${this._user.avatar}" alt="${this._user.name}">
        <h3>${this._user.name}</h3>
        <p>${this._user.email}</p>
        <button class="follow-btn">Follow</button>
      </div>
    `;

    // Emit events, don't handle logic
    this.querySelector('.follow-btn').addEventListener('click', () => {
      this.dispatchEvent(new CustomEvent('follow', {
        detail: { userId: this._user.id }
      }));
    });
  }
}

// Container - handles logic and data
class UserCardContainer extends HTMLElement {
  async connectedCallback() {
    const userId = this.getAttribute('user-id');

    // Fetch data
    this.user = await this.fetchUser(userId);

    // Create presentational component
    const card = document.createElement('user-card');
    card.user = this.user;

    // Handle events
    card.addEventListener('follow', (e) => {
      this.followUser(e.detail.userId);
    });

    this.appendChild(card);
  }

  async fetchUser(id) {
    const response = await fetch(`/api/users/${id}`);
    return response.json();
  }

  async followUser(userId) {
    await fetch(`/api/users/${userId}/follow`, { method: 'POST' });
    pan.publish('user.followed', { userId });
  }
}

customElements.define('user-card', UserCard);
customElements.define('user-card-container', UserCardContainer);
```

Render Props Pattern

Pass rendering logic as a slot:

```
class DataProvider extends HTMLElement {
  async connectedCallback() {
    const url = this.getAttribute('url');

    // Render loading state
    this.innerHTML = '<slot name="loading">Loading...</slot>';

    try {
      const response = await fetch(url);
      const data = await response.json();

      // Render with data
      const renderSlot = this.querySelector('[slot="render"]');
      if (renderSlot) {
        renderSlot.data = data;
        this.innerHTML = '';
        this.appendChild(renderSlot);
      }
    } catch (error) {
      // Render error state
      this.innerHTML = `<slot name="error">Error: ${error.message}</slot>`;
    }
  }
}

customElements.define('data-provider', DataProvider);
```

Usage:

```
<data-provider url="/api/users">
  <div slot="loading">
    <spinner-component></spinner-component>
  </div>

  <user-list slot="render"></user-list>

  <div slot="error">
    <error-message></error-message>
  </div>
</data-provider>
```

Slots and Content Projection

Slots are powerful for flexible component composition.

Named Slots

```
class CardComponent extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });

    this.shadowRoot.innerHTML = `
      <style>
        .card {
          border: 1px solid #e2e8f0;
          border-radius: 8px;
          overflow: hidden;
        }
        .header {
          background: #f7fafc;
          padding: 16px;
          border-bottom: 1px solid #e2e8f0;
        }
        .body {
          padding: 16px;
        }
        .footer {
          background: #f7fafc;
          padding: 12px 16px;
          border-top: 1px solid #e2e8f0;
          display: flex;
          justify-content: flex-end;
          gap: 8px;
        }
      </style>

      <div class="card">
        <div class="header">
          <slot name="header">Default Header</slot>
        </div>
        <div class="body">
          <slot></slot>
        </div>
        <div class="footer">
          <slot name="footer"></slot>
        </div>
      </div>
    `;
  }
}

customElements.define('card-component', CardComponent);
```

Usage:

```html
<card-component>
  <h2 slot="header">User Profile</h2>

  <!-- Default slot -->
  <p>User profile content goes here...</p>

  <div slot="footer">
    <button>Save</button>
    <button>Cancel</button>
  </div>
</card-component>
```

Slot Change Detection

React to slot content changes:

```
class DynamicList extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });

    this.shadowRoot.innerHTML = `
      <style>
        .count { font-weight: bold; color: #667eea; }
      </style>
      <div class="count"></div>
      <slot></slot>
    `;

    // Listen for slot changes
    const slot = this.shadowRoot.querySelector('slot');
    slot.addEventListener('slotchange', () => {
      this.updateCount();
    });

    this.updateCount();
  }

  updateCount() {
    const slot = this.shadowRoot.querySelector('slot');
    const elements = slot.assignedElements();

    const count = this.shadowRoot.querySelector('.count');
    count.textContent = `${elements.length} items`;
  }
}

customElements.define('dynamic-list', DynamicList);
```

Usage:

```
<dynamic-list>
  <div>Item 1</div>
  <div>Item 2</div>
  <div>Item 3</div>
</dynamic-list>

<script>
  const list = document.querySelector('dynamic-list');

  // Add item dynamically
  const newItem = document.createElement('div');
  newItem.textContent = 'Item 4';
  list.appendChild(newItem);
  // Count automatically updates!
</script>
```

Conditional Slots

Show/hide content based on slot presence:

```
class ConditionalCard extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });

    const hasHeader = this.querySelector('[slot="header"]') !== null;
    const hasFooter = this.querySelector('[slot="footer"]') !== null;

    this.shadowRoot.innerHTML = `
      <style>
        .card { border: 1px solid #ddd; border-radius: 8px; }
        .header, .footer { background: #f5f5f5; padding: 16px; }
        .body { padding: 16px; }
        .hidden { display: none; }
      </style>

      <div class="card">
        <div class="header ${hasHeader ? '' : 'hidden'}">
          <slot name="header"></slot>
        </div>
        <div class="body">
          <slot></slot>
        </div>
        <div class="footer ${hasFooter ? '' : 'hidden'}">
          <slot name="footer"></slot>
        </div>
      </div>
    `;
  }
}

customElements.define('conditional-card', ConditionalCard);
```

Dynamic Component Loading

Load components on demand for better performance.

Lazy Loading

```
class LazyLoader extends HTMLElement {
  async connectedCallback() {
    const component = this.getAttribute('component');
    const src = this.getAttribute('src');

    // Show placeholder
    this.innerHTML = '<div>Loading component...</div>';
    try {
      // Dynamically import component
      await import(src);

      // Wait for component to be defined
      await customElements.whenDefined(component);

      // Create and append component
      const element = document.createElement(component);

      // Copy attributes
      Array.from(this.attributes).forEach(attr => {
        if (attr.name !== 'component' && attr.name !== 'src') {
          element.setAttribute(attr.name, attr.value);
        }
      });

      this.innerHTML = '';
      this.appendChild(element);
    } catch (error) {
      this.innerHTML = `<div class="error">Failed to load component: ${error.message}</div>`;
    }
  }
}

customElements.define('lazy-loader', LazyLoader);
```

Usage:

```
<!-- Component loads when added to DOM -->
<lazy-loader
  component="heavy-chart"
  src="/components/heavy-chart.js"
  data-url="/api/chart-data">
</lazy-loader>
```

Intersection Observer for Viewport Loading

Load components when they enter the viewport:

```
class ViewportLoader extends HTMLElement {
  connectedCallback() {
    this.observer = new IntersectionObserver((entries) => {
      entries.forEach(entry => {
        if (entry.isIntersecting && !this.loaded) {
          this.load();
        }
      });
    }, {
      rootMargin: '50px'  // Start loading 50px before visible
    });

    this.observer.observe(this);
  }

  disconnectedCallback() {
    this.observer?.disconnect();
  }

  async load() {
    this.loaded = true;
    const component = this.getAttribute('component');
    const src = this.getAttribute('src');

    await import(src);
    await customElements.whenDefined(component);

    const element = document.createElement(component);
    Array.from(this.attributes).forEach(attr => {
      if (!['component', 'src'].includes(attr.name)) {
        element.setAttribute(attr.name, attr.value);
      }
    });

    this.appendChild(element);
  }
}

customElements.define('viewport-loader', ViewportLoader);
```

Usage:

```
<!-- Heavy image gallery - only loads when scrolled into view -->
<viewport-loader
  component="image-gallery"
  src="/components/image-gallery.js"
  album-id="123">
</viewport-loader>
```

Performance Optimization

Virtual Scrolling

Render only visible items in long lists:

```
class VirtualList extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });

    this.items = [];
    this.itemHeight = 50;
    this.visibleCount = 20;
    this.scrollTop = 0;
  }

  set data(items) {
    this.items = items;
    this.render();
  }

  connectedCallback() {
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
          height: 100%;
          overflow-y: auto;
          position: relative;
        }
        .viewport {
          position: relative;
        }
        .item {
          position: absolute;
          left: 0;
          right: 0;
          height: ${this.itemHeight}px;
          display: flex;
          align-items: center;
          padding: 0 16px;
          border-bottom: 1px solid #eee;
        }
      </style>
      <div class="viewport"></div>
    `;

    this.viewport = this.shadowRoot.querySelector('.viewport');

    this.addEventListener('scroll', () => {
      this.scrollTop = this.scrollTop;
      this.renderVisibleItems();
    });
  }
  render() {
    if (!this.viewport) return;

    // Set total height
    const totalHeight = this.items.length * this.itemHeight;
    this.viewport.style.height = `${totalHeight}px`;

    this.renderVisibleItems();
  }

  renderVisibleItems() {
    const startIndex = Math.floor(this.scrollTop / this.itemHeight);
    const endIndex = Math.min(
      startIndex + this.visibleCount,
      this.items.length
    );

    // Clear existing items
    this.viewport.innerHTML = '';

    // Render only visible items
    for (let i = startIndex; i < endIndex; i++) {
      const item = document.createElement('div');
      item.className = 'item';
      item.style.top = `${i * this.itemHeight}px`;
      item.textContent = this.items[i];

      this.viewport.appendChild(item);
    }
  }
}

customElements.define('virtual-list', VirtualList);
```

Usage:

```
const list = document.createElement('virtual-list');
list.data = Array.from({ length: 10000 }, (_, i) => `Item ${i + 1}`);
list.style.height = '400px';
document.body.appendChild(list);
```

Memoization

Cache expensive computations:

```
class MemoizedComponent extends HTMLElement {
  constructor() {
    super();
    this.cache = new Map();
  }

  memoize(fn, keyFn) {
    return (...args) => {
      const key = keyFn ? keyFn(...args) : JSON.stringify(args);

      if (this.cache.has(key)) {
        return this.cache.get(key);
      }

      const result = fn(...args);
      this.cache.set(key, result);

      return result;
    };
  }

  computeExpensiveValue = this.memoize(
    (data) => {
      // Expensive computation
      console.log('Computing...');
      return data.reduce((acc, val) => acc + val.price, 0);
    },
    (data) => data.map(d => d.id).join(',')
  );

  connectedCallback() {
    const data = [
      { id: 1, price: 100 },
      { id: 2, price: 200 }
    ];

    // First call - computes
    console.log(this.computeExpensiveValue(data));

    // Second call - cached
    console.log(this.computeExpensiveValue(data));
  }
}
```

Debouncing and Throttling

Limit expensive operations:

```
// lib/performance.js
export function debounce(fn, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);

    timeoutId = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
}

export function throttle(fn, limit) {
  let inThrottle;

  return function(...args) {
    if (!inThrottle) {
      fn.apply(this, args);
      inThrottle = true;

      setTimeout(() => {
        inThrottle = false;
      }, limit);
    }
  };
}

// Usage
class SearchBox extends HTMLElement {
  constructor() {
    super();

    // Debounce search - wait for user to stop typing
    this.handleSearch = debounce(this.search.bind(this), 300);

    // Throttle scroll - limit updates
    this.handleScroll = throttle(this.onScroll.bind(this), 100);
  }

  connectedCallback() {
    this.innerHTML = '<input type="search" placeholder="Search...">';

    this.querySelector('input').addEventListener('input', (e) => {
      this.handleSearch(e.target.value);
    });

    window.addEventListener('scroll', this.handleScroll);
  }

  search(query) {
    console.log('Searching for:', query);
    // Perform search
  }

  onScroll() {
    console.log('Scrolled');
    // Update UI based on scroll
  }
}
```

Summary

This chapter explored advanced component patterns:

- **Compound Components**: Components that work together as a cohesive unit
- **Higher-Order Components**: Mixins and decorators for code reuse
- **Component Composition**: Container/presentational pattern and render props
- **Slots**: Named slots, slot change detection, and conditional rendering
- **Dynamic Loading**: Lazy loading and viewport-based loading
- **Performance**: Virtual scrolling, memoization, debouncing, and throttling

These patterns enable you to build sophisticated, performant applications while keeping code maintainable and testable.

Best Practices

1. **Favor composition over inheritance**
   - Build complex components from simple ones
   - Use slots for flexibility
   - Keep components focused
2. **Use mixins for cross-cutting concerns**
   - Observable behavior
   - Resize handling
   - Loading states
3. **Separate logic from presentation**
   - Container components handle data
   - Presentational components handle UI
   - Easier to test and reuse
4. **Lazy load heavy components**
   - Reduce initial bundle size
   - Load on demand or when visible
   - Show loading states
5. **Optimize expensive operations**
   - Memoize pure functions
   - Debounce user input
   - Throttle scroll/resize handlers
   - Use virtual scrolling for long lists

6. **Keep performance in mind**
    - Profile before optimizing
    - Measure impact of changes
    - Don't over-optimize prematurely

Chapter 8: Business Logic Patterns

In the previous chapters, we've learned how to build components, communicate via the PAN bus, and manage state. But when building real-world applications, you'll inevitably need to inject your own custom business logic: validation rules, pricing calculations, access control, analytics tracking, and countless other domain-specific concerns.

A common question developers ask when adopting LARC is: *"Where do I put my business logic?"* This chapter explores the architectural patterns for integrating business logic into LARC applications, helping you make informed decisions about code organization and separation of concerns.

The Philosophy: Separation of Concerns

LARC's architecture naturally encourages a clean separation between:

- **Components**: UI and interaction concerns
- **PAN Bus**: Communication layer
- **Business Logic**: Domain rules and workflows

This separation isn't just academic—it makes your code:

- **Testable**: Logic can be tested independently of UI
- **Maintainable**: Changes to business rules don't require touching components
- **Reusable**: Logic can be shared across multiple components
- **Flexible**: Easy to modify workflows without refactoring components

Let's explore the patterns that make this possible.

Pattern 1: PAN Bus Listeners (Recommended)

The most common and recommended approach is to create separate modules that listen to PAN bus events and implement your business logic. This pattern treats business logic as a **first-class concern**, separate from both UI components and state management.

When to Use

Use PAN bus listeners when you need to:

- Coordinate behavior across multiple components
- Implement cross-cutting concerns (analytics, logging, validation)
- Add business rules that aren't tied to a specific component

- Keep components generic and reusable

Basic Implementation

Let's build an e-commerce application where we need to enforce business rules around cart operations:

```
// business-logic/cart-rules.js
import { pan } from '@larcjs/core';

class CartBusinessRules {
  constructor() {
    this.maxItemsPerOrder = 50;
    this.maxQuantityPerItem = 10;
  }

  init() {
    // Subscribe to cart events
    pan.subscribe('cart.item.add', this.handleItemAdd.bind(this));
    pan.subscribe('cart.item.update', this.handleItemUpdate.bind(this));
    pan.subscribe('cart.checkout.start', this.handleCheckout.bind(this));
  }

  async handleItemAdd(data) {
    console.log('Business rule: Validating item add', data);

    // Check current cart state
    const currentCart = await pan.request('cart.get');

    // Business Rule 1: Maximum items per order
    if (currentCart.items.length >= this.maxItemsPerOrder) {
      pan.publish('cart.error', {
        code: 'MAX_ITEMS_EXCEEDED',
        message: `Cannot add more than ${this.maxItemsPerOrder} items to cart`
      });
      return;
    }

    // Business Rule 2: Check inventory
    const available = await this.checkInventory(data.product.id);
    if (!available || available < data.quantity) {
      pan.publish('cart.error', {
        code: 'INSUFFICIENT_INVENTORY',
        message: 'This item is currently out of stock',
        product: data.product
      });
      return;
    }
    // Business Rule 3: Apply pricing
    const pricing = await this.calculatePrice(data.product, data.quantity);

    // All validations passed - allow the add and publish enriched data
    pan.publish('cart.item.validated', {
      ...data,
      pricing,
      timestamp: Date.now()
    });
  }

  async handleItemUpdate(data) {
    // Business Rule: Quantity limits
    if (data.quantity > this.maxQuantityPerItem) {
      pan.publish('cart.error', {
        code: 'MAX_QUANTITY_EXCEEDED',
        message: `Maximum ${this.maxQuantityPerItem} per item`
      });
      return;
    }

    // Check inventory for new quantity
    const available = await this.checkInventory(data.productId);
    if (available < data.quantity) {
      pan.publish('cart.error', {
        code: 'INSUFFICIENT_INVENTORY',
        message: `Only ${available} available`,
        available
      });
      return;
    }

    pan.publish('cart.item.update.validated', data);
  }

  async handleCheckout(data) {
    // Business Rule: Minimum order value
    const cart = await pan.request('cart.get');
    const total = cart.items.reduce((sum, item) => sum + item.total, 0);

    if (total < 10) {
      pan.publish('checkout.error', {
        code: 'MINIMUM_ORDER_NOT_MET',
        message: 'Minimum order value is $10',
        current: total,
        required: 10
      });
      return;
    }

    // Business Rule: User must be logged in
    const user = await pan.request('auth.user.get');
    if (!user) {
      pan.publish('checkout.error', {
        code: 'AUTH_REQUIRED',
        message: 'Please log in to continue'
      });
      return;
    }

    pan.publish('checkout.validated', { cart, user });
  }

  async checkInventory(productId) {
    // In real app, this would call your backend
    const response = await fetch(`/api/inventory/${productId}`);
    const data = await response.json();
    return data.available;
  }

  async calculatePrice(product, quantity) {
    // Apply business logic: bulk discounts, promotions, etc.
    let unitPrice = product.price;

    // Bulk discount: 10% off for 5+ items
    if (quantity >= 5) {
      unitPrice = unitPrice * 0.9;
    }

    // TODO: Check for active promotions
    // TODO: Apply user-specific pricing

    return {
      unitPrice,
      quantity,
      subtotal: unitPrice * quantity,
      discount: quantity >= 5 ? (product.price - unitPrice) * quantity : 0
    };
  }
}

// Initialize and export
const cartRules = new CartBusinessRules();
export default cartRules;
```

Now in your main application file:

```
// app.js
import { pan } from '@larcjs/core';
import cartRules from './business-logic/cart-rules.js';

// Initialize business logic
cartRules.init();

// Your components just publish events - the business logic handles the rest
// No business logic in components themselves!
```

Your components remain simple and focused on UI:

```
// components/product-card.js
class ProductCard extends HTMLElement {
  // ... component setup ...

  handleAddToCart() {
    // Just publish the event - business logic will validate
    pan.publish('cart.item.add', {
      product: this.product,
      quantity: this.quantity
    });

    // Show optimistic UI
    this.showAddingState();
  }

  connectedCallback() {
    super.connectedCallback();

    // Listen for validation results
    this.unsubscribers = [
      pan.subscribe('cart.item.validated', (data) => {
        if (data.product.id === this.product.id) {
          this.showSuccess();
        }
      }),

      pan.subscribe('cart.error', (error) => {
        this.showError(error.message);
      })
    ];
  }
}
```

Advantages

This pattern provides several key benefits:

1. **Separation of Concerns**: Components handle UI, business logic modules handle rules
2. **Easy Testing**: Test business logic without rendering components
3. **Centralized Rules**: All cart rules in one place, easy to modify
4. **Reusable**: Multiple components can trigger the same logic
5. **Flexible**: Easy to add, remove, or modify rules

Advanced: Composable Business Logic

For larger applications, you can compose multiple business logic modules:

```
// business-logic/index.js
import { pan } from '@larcjs/core';
import cartRules from './cart-rules.js';
import pricingRules from './pricing-rules.js';
import inventoryRules from './inventory-rules.js';
import analyticsRules from './analytics-rules.js';

export function initBusinessLogic() {
  console.log('Initializing business logic...');

  // Initialize all business logic modules
  cartRules.init();
  pricingRules.init();
  inventoryRules.init();
  analyticsRules.init();

  console.log('Business logic ready');
}
```

```
// app.js
import { initBusinessLogic } from './business-logic/index.js';

// Single call to initialize all business logic
initBusinessLogic();
```

Pattern 2: Extending Components

Sometimes you need to add business logic directly to a component, especially when:

- The logic is specific to one component type
- You need to override component behavior
- You're creating specialized versions of generic components

When to Use

Use component extension when:

- Logic is tightly coupled to component rendering
- You need access to component internals (Shadow DOM, private methods)
- Creating specialized variants of base components
- Logic doesn't need to be shared across different component types

Implementation

Let's extend a generic product card with business-specific behavior:

```
// components/base/product-card.js
export class ProductCard extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });
    this.render();
  }

  render() {
    this.shadowRoot.innerHTML = `
      <style>
        /* Base styles */
      </style>
      <div class="card">
        <img src="${this.product.image}" alt="${this.product.name}">
        <h3>${this.product.name}</h3>
        <p class="price">${this.formatPrice(this.product.price)}</p>
        <button class="add-to-cart">Add to Cart</button>
      </div>
    `;

    this.shadowRoot.querySelector('.add-to-cart')
      .addEventListener('click', () => this.handleAddToCart());
  }

  handleAddToCart() {
    pan.publish('cart.item.add', {
      product: this.product,
      quantity: 1
    });
  }

  formatPrice(price) {
    return `$${price.toFixed(2)}`;
  }

  get product() {
    return JSON.parse(this.getAttribute('product'));
  }
}
```

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

Now extend it with business-specific logic:

```
// components/premium-product-card.js
import { ProductCard } from './base/product-card.js';

export class PremiumProductCard extends ProductCard {
  connectedCallback() {
    super.connectedCallback();

    // Add business-specific subscriptions
    this._unsubscribers = [
      pan.subscribe('pricing.update', this.handlePriceUpdate.bind(this)),
      pan.subscribe('user.tier.changed', this.handleTierChange.bind(this))
    ];

    // Initialize premium features
    this.loadMemberPricing();
  }

  async loadMemberPricing() {
    const user = await pan.request('auth.user.get');
    if (user?.tier === 'premium') {
      this.applyPremiumDiscount();
    }
  }

  applyPremiumDiscount() {
    // Business Rule: 15% discount for premium members
    const discount = 0.15;
    const originalPrice = this.product.price;
    const discountedPrice = originalPrice * (1 - discount);

    this.product.price = discountedPrice;
    this.product.originalPrice = originalPrice;

    this.render(); // Re-render with new price
  }

  render() {
    // Call parent render
    super.render();

    // Add premium badge if applicable
    if (this.product.originalPrice) {
      this.addPremiumBadge();
    }
```

```
}

addPremiumBadge() {
  const badge = document.createElement('div');
  badge.className = 'premium-badge';
  badge.innerHTML = `
    <style>
      .premium-badge {
        position: absolute;
        top: 10px;
        right: 10px;
        background: gold;
        color: black;
        padding: 5px 10px;
        border-radius: 3px;
        font-weight: bold;
      }
      .original-price {
        text-decoration: line-through;
        color: #999;
        font-size: 0.9em;
      }
    </style>
    <span>Premium Member</span>
    <div class="original-price">
      ${this.formatPrice(this.product.originalPrice)}
    </div>
  `;

  this.shadowRoot.querySelector('.card').prepend(badge);
}

async handleAddToCart() {
  // Business validation before adding
  const canAddPremiumItem = await this.validatePremiumAccess();

  if (!canAddPremiumItem) {
    pan.publish('app.error', {
      message: 'Premium membership required for this product'
    });
    return;
  }

  // Track premium conversions
  this.trackPremiumConversion();

  // Call parent behavior
  super.handleAddToCart();
  }

  async validatePremiumAccess() {
    if (!this.product.premiumOnly) return true;

    const user = await pan.request('auth.user.get');
    return user?.tier === 'premium';
  }

  trackPremiumConversion() {
    pan.publish('analytics.track', {
      event: 'premium_product_add_to_cart',
      product: this.product.id,
      price: this.product.price,
      discount: this.product.originalPrice - this.product.price
    });
  }

  handlePriceUpdate(data) {
    if (data.productId === this.product.id) {
      this.product.price = data.newPrice;
      this.render();
    }
  }

  handleTierChange(data) {
    // User tier changed - recalculate pricing
    this.loadMemberPricing();
  }

  disconnectedCallback() {
    // Clean up subscriptions
    this._unsubscribers.forEach(unsub => unsub());
    super.disconnectedCallback?.();
  }
}

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

When This Makes Sense

Component extension works well when:

1. **The logic changes how the component renders or behaves**
2. **You need multiple variants of a base component** (premium, free, guest, etc.)
3. **Business logic is closely tied to component lifecycle**

However, be cautious: overuse of extension can lead to:

- Tight coupling between business logic and UI
- Harder to test business rules independently
- Duplication if multiple components need the same logic

Pattern 3: Wrapper Components

Wrapper components let you add behavior around existing components without modifying them. This is useful when you want to:

- Add behavior to third-party components
- Keep base components pristine
- Compose behaviors dynamically

Implementation

```
// components/business-wrapper.js
class BusinessWrapper extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });

    // Intercept events from slotted content
    this.addEventListener('add-to-cart', this.handleBusinessLogic.bind(this));

    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
        }
        .validation-message {
          color: red;
          padding: 10px;
          background: #fee;
          border-radius: 4px;
          margin-bottom: 10px;
        }
        .validation-message.hidden {
          display: none;
        }
      </style>
      <div class="validation-message hidden"></div>
      <slot></slot>
    `;
  }

  async handleBusinessLogic(e) {
    // Stop the event from propagating immediately
    e.stopPropagation();
    // Apply business validation
    const validation = await this.validateBusinessRules(e.detail);

    if (!validation.valid) {
      this.showError(validation.message);
      return;
    }

    // Validation passed - let the event continue
    pan.publish('cart.item.add', e.detail);
  }

  async validateBusinessRules(data) {
    // Check user eligibility
    const user = await pan.request('auth.user.get');
    if (!user) {
      return {
        valid: false,
        message: 'Please log in to add items to cart'
      };
    }

    // Check age restriction
    if (data.product.ageRestricted && user.age < 21) {
      return {
        valid: false,
        message: 'This product requires age verification (21+)'
      };
    }

    // Check geographic restriction
    if (data.product.geoRestricted && !this.isAllowedRegion(user.region)) {
      return {
        valid: false,
        message: 'This product is not available in your region'
      };
    }

    return { valid: true };
  }

  isAllowedRegion(region) {
    // Business logic for regional restrictions
    const allowedRegions = ['US', 'CA', 'UK'];
    return allowedRegions.includes(region);
  }

  showError(message) {
    const errorEl = this.shadowRoot.querySelector('.validation-message');
    errorEl.textContent = message;
    errorEl.classList.remove('hidden');

    setTimeout(() => {
      errorEl.classList.add('hidden');
    }, 5000);
  }
}

customElements.define('business-wrapper', BusinessWrapper);
```

Usage:

```
<!-- Wrap any component with business logic -->
<business-wrapper>
  <product-card product-id="123"></product-card>
</business-wrapper>

<business-wrapper>
  <quick-buy-button product-id="456"></quick-buy-button>
</business-wrapper>
```

The wrapper intercepts events and applies business logic **without modifying** the wrapped components.

Pattern 4: Behavior Mixins

Mixins let you share behavior across multiple component types. This is useful for cross-cutting concerns like analytics, logging, or validation.

Implementation

```
// mixins/analytics-mixin.js
export const AnalyticsMixin = (BaseClass) => class extends BaseClass {
  track(event, data = {}) {
    pan.publish('analytics.track', {
      event,
      data,
      component: this.tagName.toLowerCase(),
      timestamp: Date.now(),
      ...this.getAnalyticsContext()
    });
  }

  trackInteraction(element, action) {
    this.track(`${element}.${action}`, {
      element,
      action
    });
  }

  getAnalyticsContext() {
    // Add common context to all analytics events
    return {
      page: window.location.pathname,
      referrer: document.referrer
    };
  }

  connectedCallback() {
    super.connectedCallback?.();
    this.track('component.mounted', { id: this.id });
  }

  disconnectedCallback() {
    this.track('component.unmounted', { id: this.id });
    super.disconnectedCallback?.();
  }
};

// mixins/validation-mixin.js
export const ValidationMixin = (BaseClass) => class extends BaseClass {
  async validate(data, rules) {
    const errors = [];

    for (const [field, rule] of Object.entries(rules)) {
      const value = data[field];

      if (rule.required && !value) {
        errors.push(`${field} is required`);
      }

      if (rule.min && value < rule.min) {
        errors.push(`${field} must be at least ${rule.min}`);
      }

      if (rule.max && value > rule.max) {
        errors.push(`${field} must be at most ${rule.max}`);
      }

      if (rule.pattern && !rule.pattern.test(value)) {
        errors.push(`${field} is invalid`);
      }

      if (rule.custom) {
        const customError = await rule.custom(value, data);
        if (customError) errors.push(customError);
      }
    }

    return {
      valid: errors.length === 0,
      errors
    };
  }

  showValidationErrors(errors) {
    pan.publish('validation.errors', {
      component: this.tagName.toLowerCase(),
      errors
    });
  }
};
```

Use mixins to compose behavior:

```
import { AnalyticsMixin } from './mixins/analytics-mixin.js';
import { ValidationMixin } from './mixins/validation-mixin.js';

class CheckoutForm extends ValidationMixin(AnalyticsMixin(HTMLElement)) {
  async handleSubmit() {
    // Use validation from mixin
    const validation = await this.validate(this.formData, {
      email: {
        required: true,
        pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/
      },
      cardNumber: {
        required: true,
        custom: async (value) => {
          const valid = await this.validateCard(value);
          return valid ? null : 'Invalid card number';
        }
      }
    });

    if (!validation.valid) {
      this.showValidationErrors(validation.errors);
      return;
    }

    // Use analytics from mixin
    this.track('checkout.submit', {
      amount: this.total,
      items: this.items.length
    });

    // Process checkout
    this.processOrder();
  }
}
```

Pattern 5: Service Layer

For complex business logic, create a dedicated service layer that components and PAN listeners can both use:

```
// services/pricing-service.js
class PricingService {
  async calculatePrice(product, quantity, user) {
    let price = product.basePrice;

    // Business Rule: Volume discounts
    if (quantity >= 10) price *= 0.85;
    else if (quantity >= 5) price *= 0.90;

    // Business Rule: Member discounts
    if (user?.tier === 'premium') {
      price *= 0.85;
    } else if (user?.tier === 'gold') {
      price *= 0.90;
    }

    // Business Rule: Active promotions
    const promotions = await this.getActivePromotions(product.id);
    for (const promo of promotions) {
      price = this.applyPromotion(price, promo);
    }

    return {
      unitPrice: price,
      quantity,
      subtotal: price * quantity,
      savings: (product.basePrice - price) * quantity
    };
  }

  async getActivePromotions(productId) {
    const response = await fetch(`/api/promotions?product=${productId}`);
    return response.json();
  }

  applyPromotion(price, promotion) {
    if (promotion.type === 'percentage') {
      return price * (1 - promotion.value / 100);
    } else if (promotion.type === 'fixed') {
      return Math.max(0, price - promotion.value);
    }
    return price;
  }

  async getTax(subtotal, region) {
    const taxRates = {
      'CA': 0.0725,
      'NY': 0.08,
      'TX': 0.0625
    };

    return subtotal * (taxRates[region] || 0);
  }
}

export default new PricingService();
```

Use the service from both components and PAN listeners:

```
// In a component
import pricingService from './services/pricing-service.js';

class ProductCard extends HTMLElement {
  async updatePrice() {
    const user = await pan.request('auth.user.get');
    const pricing = await pricingService.calculatePrice(
      this.product,
      this.quantity,
      user
    );

    this.displayPrice(pricing);
  }
}

// In business logic
import pricingService from './services/pricing-service.js';

class CartBusinessLogic {
  init() {
    pan.subscribe('cart.item.add', async (data) => {
      const user = await pan.request('auth.user.get');
      const pricing = await pricingService.calculatePrice(
        data.product,
        data.quantity,
        user
      );

      pan.publish('cart.item.priced', { ...data, pricing });
    });
  }
}
```

Decision Matrix

Here's how to choose the right pattern:

Scenario	Recommended Pattern	Why
Cross-component coordination	PAN Bus Listeners	Decoupled, flexible
Analytics/logging	Mixins	Reusable across all components
Validation before actions	PAN Bus Listeners	Centralized rules
Component-specific UI logic	Extend Component	Access to internals
Add behavior to third-party components	Wrapper	Non-invasive
Complex business calculations	Service Layer	Testable, reusable
Component variants (premium, free)	Extend Component	Clear inheritance
Feature flags / A-B testing	Wrapper or PAN Listeners	Easy to toggle

Real-World Example: E-Commerce Checkout

Let's see how these patterns work together in a complete checkout flow:

```
// services/checkout-service.js
class CheckoutService {
  async processOrder(cart, paymentInfo, shippingInfo) {
    // Complex business logic
    const pricing = await this.calculateFinalPricing(cart);
    const shipping = await this.calculateShipping(cart, shippingInfo);
    const tax = await this.calculateTax(pricing.subtotal, shippingInfo.state);

    return {
      items: cart.items,
      pricing,
      shipping,
      tax,
      total: pricing.subtotal + shipping.cost + tax
    };
  }

  async calculateFinalPricing(cart) {
    // Apply all discounts, coupons, etc.
    let subtotal = 0;
    let savings = 0;

    for (const item of cart.items) {
      const itemPricing = await pricingService.calculatePrice(
        item.product,
        item.quantity,
        cart.user
      );
      subtotal += itemPricing.subtotal;
      savings += itemPricing.savings;
    }

    return { subtotal, savings };
  }

  async calculateShipping(cart, shippingInfo) {
    // Shipping business rules
    if (cart.total >= 50) {
      return { method: 'standard', cost: 0, freeShipping: true };
    }

    const weight = cart.items.reduce((sum, item) => sum + item.weight, 0);
    const zone = this.getShippingZone(shippingInfo.state);

    return {
      method: 'standard',
      cost: this.calculateShippingCost(weight, zone),
      freeShipping: false
    };
  }

  calculateShippingCost(weight, zone) {
    const baseRate = { 1: 5, 2: 7, 3: 10 };
    return baseRate[zone] + (weight > 5 ? (weight - 5) * 0.5 : 0);
  }

  getShippingZone(state) {
    const zones = {
      1: ['CA', 'OR', 'WA'],
      2: ['NV', 'AZ', 'UT', 'ID'],
      3: [] // All other states
    };

    for (const [zone, states] of Object.entries(zones)) {
      if (states.includes(state)) return parseInt(zone);
    }
    return 3;
  }

  async calculateTax(subtotal, state) {
    return pricingService.getTax(subtotal, state);
  }
}

export default new CheckoutService();

// business-logic/checkout-rules.js
import checkoutService from '../services/checkout-service.js';

class CheckoutBusinessRules {
  init() {
    pan.subscribe('checkout.start', this.handleCheckoutStart.bind(this));
    pan.subscribe('checkout.submit', this.handleCheckoutSubmit.bind(this));
  }

  async handleCheckoutStart(data) {
    // Business validations
    const cart = await pan.request('cart.get');
    const user = await pan.request('auth.user.get');

    // Validation 1: Cart not empty
    if (!cart.items.length) {
      pan.publish('checkout.error', {
        code: 'EMPTY_CART',
        message: 'Your cart is empty'
      });
      return;
    }

    // Validation 2: User logged in
    if (!user) {
      pan.publish('checkout.error', {
        code: 'AUTH_REQUIRED',
        message: 'Please log in to continue'
      });
      return;
    }
    // Validation 3: Inventory check
    for (const item of cart.items) {
      const available = await this.checkInventory(item.product.id);
      if (available < item.quantity) {
        pan.publish('checkout.error', {
          code: 'INSUFFICIENT_INVENTORY',
          message: `Only ${available} of "${item.product.name}" available`,
          item
        });
        return;
      }
    }

    // All validations passed
    pan.publish('checkout.validated', { cart, user });
  }

  async handleCheckoutSubmit(data) {
    try {
      // Process order through service
      const order = await checkoutService.processOrder(
        data.cart,
        data.paymentInfo,
        data.shippingInfo
      );

      // Submit to backend
      const response = await fetch('/api/orders', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(order)
      });

      if (!response.ok) {
        throw new Error('Order submission failed');
      }

      const result = await response.json();

      // Success
      pan.publish('checkout.success', {
        orderId: result.orderId,
        order: result
      });

      // Clear cart
      pan.publish('cart.clear');
    } catch (error) {
      pan.publish('checkout.error', {
        code: 'SUBMISSION_FAILED',
        message: 'Unable to process order. Please try again.',
        error
      });
    }
  }

  async checkInventory(productId) {
    const response = await fetch(`/api/inventory/${productId}`);
    const data = await response.json();
    return data.available;
  }
}

export default new CheckoutBusinessRules();
```

The checkout component stays simple:

```
// components/checkout-form.js
class CheckoutForm extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });
    this.render();
    this.attachEventListeners();
    this.subscribeToEvents();
  }

  subscribeToEvents() {
    this._unsubscribers = [
      pan.subscribe('checkout.validated', () => {
        this.showCheckoutForm();
      }),

      pan.subscribe('checkout.error', (error) => {
        this.showError(error.message);
      }),

      pan.subscribe('checkout.success', (data) => {
        this.showSuccess(data.orderId);
      })
    ];
  }

  handleSubmit(e) {
    e.preventDefault();

    // Just collect data and publish - business logic handles the rest
    pan.publish('checkout.submit', {
      cart: this.cart,
      paymentInfo: this.getPaymentInfo(),
      shippingInfo: this.getShippingInfo()
    });

    this.showProcessing();
  }

  // UI methods only - no business logic
  showCheckoutForm() { /* ... */ }
  showError(message) { /* ... */ }
  showSuccess(orderId) { /* ... */ }
  showProcessing() { /* ... */ }
}
```

Testing Business Logic

One of the biggest advantages of separating business logic is testability. Here's how to test each pattern:

Testing PAN Bus Listeners

```
// __tests__/cart-rules.test.js
import { describe, it, expect, beforeEach, vi } from 'vitest';
import { pan } from '@larcjs/core';
import cartRules from '../business-logic/cart-rules.js';

describe('Cart Business Rules', () => {
  beforeEach(() => {
    // Reset PAN bus between tests
    pan.clear();
    cartRules.init();
  });

  it('should reject adding more than max items', async () => {
    // Mock cart with max items
    pan.respond('cart.get', () => ({
      items: new Array(50).fill({})
    }));

    const errorHandler = vi.fn();
    pan.subscribe('cart.error', errorHandler);

    // Try to add another item
    await pan.publish('cart.item.add', {
      product: { id: 1, name: 'Test' },
      quantity: 1
    });

    expect(errorHandler).toHaveBeenCalledWith({
      code: 'MAX_ITEMS_EXCEEDED',
      message: expect.stringContaining('50 items')
    });
  });

  it('should apply bulk discount for 5+ items', async () => {
    const validated = vi.fn();
    pan.subscribe('cart.item.validated', validated);

    await pan.publish('cart.item.add', {
      product: { id: 1, name: 'Test', price: 100 },
      quantity: 5
    });

    expect(validated).toHaveBeenCalledWith(
      expect.objectContaining({
        pricing: expect.objectContaining({
          unitPrice: 90, // 10% discount
          discount: 50
        })
      })
    );
  });
});
```

Testing Services

```
// __tests__/pricing-service.test.js
import { describe, it, expect } from 'vitest';
import pricingService from '../services/pricing-service.js';

describe('Pricing Service', () => {
  it('should apply volume discount', async () => {
    const product = { basePrice: 100 };
    const pricing = await pricingService.calculatePrice(product, 10, null);

    expect(pricing.unitPrice).toBe(85); // 15% off for 10+
    expect(pricing.subtotal).toBe(850);
  });

  it('should stack member and volume discounts', async () => {
    const product = { basePrice: 100 };
    const user = { tier: 'premium' };

    const pricing = await pricingService.calculatePrice(product, 10, user);
    // 15% volume + 15% premium = 72.25
    expect(pricing.unitPrice).toBe(72.25);
  });
});
```

Best Practices

1. Keep Components Dumb

Components should focus on UI and user interaction. They publish events but don't implement business rules.

Good:

```
handleAddToCart() {
  pan.publish('cart.item.add', { product: this.product });
}
```

Bad:

```
async handleAddToCart() {
  // Business logic in component - hard to test and reuse
  const inventory = await fetch('/api/inventory');
  if (inventory < this.quantity) {
    alert('Out of stock');
    return;
  }

  const user = await fetch('/api/user');
  if (user.age < 21 && this.product.ageRestricted) {
    alert('Age restricted');
    return;
  }

  // ... more business logic
}
```

2. Use Services for Complex Logic

If business logic involves multiple steps, calculations, or external APIs, put it in a service:

```
// Good: Service handles complexity
const pricing = await pricingService.calculatePrice(product, quantity, user);

// Bad: Business logic scattered across components and PAN listeners
const basePrice = product.price;
const volumeDiscount = quantity >= 10 ? 0.15 : 0;
const memberDiscount = user?.tier === 'premium' ? 0.15 : 0;
// ... etc
```

3. Make Business Logic Observable

Use PAN bus to make business logic transparent:

```
class OrderProcessor {
  async processOrder(order) {
    pan.publish('order.processing.start', { orderId: order.id });

    try {
      await this.validateOrder(order);
      pan.publish('order.validated', { orderId: order.id });

      await this.chargePayment(order);
      pan.publish('order.charged', { orderId: order.id });

      await this.createShipment(order);
      pan.publish('order.shipped', { orderId: order.id });

      pan.publish('order.complete', { orderId: order.id });
    } catch (error) {
      pan.publish('order.failed', { orderId: order.id, error });
    }
  }
}
```

Now other parts of your app can react to these events (analytics, notifications, UI updates, etc.).

4. Document Business Rules

Make business rules explicit and documented:

```
/**
 * Shopping Cart Business Rules
 *
 * 1. Maximum 50 items per order
 * 2. Maximum 10 quantity per item
 * 3. Free shipping over $50
 * 4. Volume discounts:
 *    - 5-9 items: 10% off
 *    - 10+ items: 15% off
 * 5. Member discounts:
 *    - Premium: 15% off
 *    - Gold: 10% off
 * 6. Minimum order value: $10
 */
class CartBusinessRules {
  // Implementation
}
```

5. Use Feature Flags

Make business logic toggleable:

```
class CheckoutRules {
  constructor() {
    this.features = {
      guestCheckout: true,
      expressCheckout: false,
      digitalWallet: true
    };
  }

  async handleCheckout(data) {
    if (!this.features.guestCheckout && !data.user) {
      pan.publish('checkout.error', {
        message: 'Account required for checkout'
      });
      return;
    }

    // ... rest of logic
  }
}
```

Summary

When integrating business logic into LARC applications:

1. **Default to PAN Bus listeners** for most business logic - it's decoupled, testable, and flexible
2. **Use services** for complex calculations and workflows
3. **Extend components** only when logic is tightly coupled to UI
4. **Use mixins** for cross-cutting concerns like analytics
5. **Wrap components** when adding behavior to third-party code
6. **Keep components dumb** - they publish events, business logic handles the rest

This separation of concerns makes your application: - **Easier to test** - business logic without rendering components - **More maintainable** - business rules in one place - **More flexible** - easy to change rules without touching UI - **More reusable** - logic can be shared across components

In the next chapter, we'll explore routing and navigation, building on these patterns to create complete single-page applications.

Chapter 8: Routing and Navigation

Client-side routing enables single-page applications (SPAs) to feel like multi-page websites without full page reloads. LARC provides routing through web standards and the PAN bus, keeping things simple and framework-free.

Client-Side Routing Basics

Client-side routing intercepts link clicks and updates the URL without reloading:

```
// lib/router.js
class Router {
  constructor() {
    this.routes = new Map();
    this.currentRoute = null;

    // Intercept link clicks
    document.addEventListener('click', (e) => {
      if (e.target.matches('a[href^="/"]')) {
        e.preventDefault();
        this.navigate(e.target.getAttribute('href'));
      }
    });

    // Handle browser back/forward
    window.addEventListener('popstate', () => {
      this.handleRoute(window.location.pathname);
    });
  }

  register(path, handler) {
    this.routes.set(path, handler);
  }

  navigate(path, state = {}) {
    window.history.pushState(state, '', path);
    this.handleRoute(path);

    // Publish navigation event
    pan.publish('router.navigated', { path, state });
  }

  handleRoute(path) {
    // Find matching route
    for (const [pattern, handler] of this.routes) {
      const params = this.matchRoute(pattern, path);
      if (params) {
        this.currentRoute = { path, pattern, params };
        handler(params);
        return;
      }
    }

    // 404 - no match
    pan.publish('router.not-found', { path });
  }

  matchRoute(pattern, path) {
    // Simple pattern matching
    const patternParts = pattern.split('/').filter(Boolean);
    const pathParts = path.split('/').filter(Boolean);

    if (patternParts.length !== pathParts.length) {
      return null;
    }

    const params = {};

    for (let i = 0; i < patternParts.length; i++) {
      const patternPart = patternParts[i];
      const pathPart = pathParts[i];

      if (patternPart.startsWith(':')) {
        // Dynamic segment
        params[patternPart.slice(1)] = pathPart;
      } else if (patternPart !== pathPart) {
        // Mismatch
        return null;
      }
    }

    return params;
  }

  start() {
    this.handleRoute(window.location.pathname);
  }
```

```
}

export const router = new Router();
```

Usage:

```
import { router } from './lib/router.js';

// Register routes
router.register('/', () => {
  document.getElementById('app').innerHTML = '<home-page></home-page>';
});

router.register('/about', () => {
  document.getElementById('app').innerHTML = '<about-page></about-page>';
});

router.register('/users/:id', (params) => {
  const page = document.createElement('user-page');
  page.setAttribute('user-id', params.id);
  document.getElementById('app').innerHTML = '';
  document.getElementById('app').appendChild(page);
});

// Start router
router.start();
```

The pan-router Component

LARC provides a declarative router component:

```
<pan-router>
  <pan-route path="/" component="home-page"></pan-route>
  <pan-route path="/about" component="about-page"></pan-route>
  <pan-route path="/users/:id" component="user-page"></pan-route>
  <pan-route path="/posts/:postId/comments/:commentId" component="comment-page"></pan-route>
  <pan-route path="*" component="not-found-page"></pan-route>
</pan-router>
```

Implementation:

```
class PanRouter extends HTMLElement {
  connectedCallback() {
    this.routes = Array.from(this.querySelectorAll('pan-route')).map(route => ({
      path: route.getAttribute('path'),
      component: route.getAttribute('component'),
      guard: route.getAttribute('guard')
    }));

    // Create outlet
    this.outlet = document.createElement('div');
    this.outlet.className = 'router-outlet';
    this.appendChild(this.outlet);

    // Listen for navigation
    pan.subscribe('router.navigate', ({ path, params }) => {
      this.navigate(path, params);
    });

    // Handle browser navigation
    window.addEventListener('popstate', () => {
      this.handleRoute(window.location.pathname);
    });

    // Intercept links
    document.addEventListener('click', (e) => {
      const link = e.target.closest('a[href^="/"]');
      if (link) {
        e.preventDefault();
        this.navigate(link.getAttribute('href'));
      }
    });

    // Initial route
    this.handleRoute(window.location.pathname);
  }

  navigate(path, params = {}) {
    window.history.pushState(params, '', path);
    this.handleRoute(path);
  }

  async handleRoute(path) {
    // Find matching route
    for (const route of this.routes) {
      const params = this.matchRoute(route.path, path);

      if (params) {
        // Check route guard
        if (route.guard) {
          const canActivate = await this.runGuard(route.guard, params);
          if (!canActivate) {
            return;
          }
        }

        // Render component
        await this.renderComponent(route.component, params);
        return;
      }
    }

    // 404
    pan.publish('router.not-found', { path });
  }

  matchRoute(pattern, path) {
    if (pattern === '*') return {};

    const patternParts = pattern.split('/').filter(Boolean);
    const pathParts = path.split('/').filter(Boolean);

    if (patternParts.length !== pathParts.length) return null;

    const params = {};

    for (let i = 0; i < patternParts.length; i++) {
      if (patternParts[i].startsWith(':')) {
        params[patternParts[i].slice(1)] = pathParts[i];
      } else if (patternParts[i] !== pathParts[i]) {
        return null;
      }
    }

    return params;
  }

  async runGuard(guardName, params) {
    const result = await pan.request(`guard.${guardName}`, params);
    return result !== false;
  }

  async renderComponent(componentName, params) {
    // Wait for component to be defined
    await customElements.whenDefined(componentName);

    // Create component
    const component = document.createElement(componentName);

    // Pass route params
    Object.entries(params).forEach(([key, value]) => {
      component.setAttribute(key, value);
    });

    // Clear outlet and add component
    this.outlet.innerHTML = '';
```

```
    this.outlet.appendChild(component);

    // Publish route change
    pan.publish('router.changed', { component: componentName, params });
  }
}

customElements.define('pan-router', PanRouter);
customElements.define('pan-route', class extends HTMLElement {});
```

Route Parameters

Access route parameters in components:

```
class UserPage extends HTMLElement {
  static get observedAttributes() {
    return ['user-id'];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'user-id' && newValue) {
      this.loadUser(newValue);
    }
  }

  async loadUser(id) {
    const response = await fetch(`/api/users/${id}`);
    const user = await response.json();
    this.render(user);
  }

  render(user) {
    this.innerHTML = `
      <h1>${user.name}</h1>
      <p>${user.email}</p>
    `;
  }
}

customElements.define('user-page', UserPage);
```

Route Guards

Protect routes with authentication checks:

```
// Respond to auth guard
pan.respond('guard.auth', async () => {
  const token = localStorage.getItem('authToken');
  if (!token) {
    // Redirect to login
    pan.publish('router.navigate', { path: '/login' });
    return false;
  }

  // Verify token
  try {
    const response = await fetch('/api/auth/verify', {
      headers: { 'Authorization': `Bearer ${token}` }
    });

    return response.ok;
  } catch {
    return false;
  }
});

// Respond to admin guard
pan.respond('guard.admin', async () => {
  const user = await pan.request('auth.user.get');
  return user?.role === 'admin';
});
```

Usage:

```
<pan-router>
  <pan-route path="/login" component="login-page"></pan-route>
  <pan-route path="/dashboard" component="dashboard-page" guard="auth"></pan-route>
  <pan-route path="/admin" component="admin-page" guard="admin"></pan-route>
</pan-router>
```

Nested Routes

Support hierarchical routing:

```
<pan-router>
  <pan-route path="/settings" component="settings-layout">
    <pan-route path="/settings/profile" component="profile-settings"></pan-route>
    <pan-route path="/settings/security" component="security-settings"></pan-route>
    <pan-route path="/settings/billing" component="billing-settings"></pan-route>
  </pan-route>
</pan-router>
```

Programmatic Navigation

Navigate from JavaScript:

```
// Navigate to a path
pan.publish('router.navigate', { path: '/users/123' });

// Navigate with state
pan.publish('router.navigate', {
  path: '/search',
  state: { query: 'web components' }
});

// Go back
pan.publish('router.back');

// Go forward
pan.publish('router.forward');

// Replace current route (no history entry)
pan.publish('router.replace', { path: '/new-path' });
```

Query Parameters

Parse and use query parameters:

```
class SearchPage extends HTMLElement {
  connectedCallback() {
    // Parse query params
    const params = new URLSearchParams(window.location.search);
    const query = params.get('q');
    const page = parseInt(params.get('page') || '1');

    this.performSearch(query, page);

    // Listen for query changes
    pan.subscribe('router.changed', () => {
      const params = new URLSearchParams(window.location.search);
      const newQuery = params.get('q');
      const newPage = parseInt(params.get('page') || '1');

      if (newQuery !== query || newPage !== page) {
        this.performSearch(newQuery, newPage);
      }
    });
  }

  performSearch(query, page) {
    // Search implementation
  }
}
```

Update query params:

```
function updateQuery(params) {
  const url = new URL(window.location);

  Object.entries(params).forEach(([key, value]) => {
    url.searchParams.set(key, value);
  });

  pan.publish('router.navigate', { path: url.pathname + url.search });
}

// Usage
updateQuery({ q: 'web components', page: '2' });
```

Summary

LARC routing provides: - Client-side navigation without page reloads - Declarative route configuration - Route parameters and guards - Nested routing support - Browser history integration - PAN bus integration

---

Best Practices

1. **Use declarative routing** - Prefer `<pan-router>` over imperative API
2. **Implement route guards** - Protect sensitive routes
3. **Handle 404s gracefully** - Always include catch-all route
4. **Preserve scroll position** - Restore scroll on back navigation
5. **Use query params for filters** - Makes URLs shareable

Chapter 9: Forms and Validation

Forms are the primary way users input data into web applications. LARC provides patterns for building accessible, validated forms using web standards and the PAN bus.

Form Components

Basic Form Component

```
class ContactForm extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    this.render();
    this.attachEventListeners();
  }

  attachEventListeners() {
    const form = this.shadowRoot.querySelector('form');

    form.addEventListener('submit', async (e) => {
      e.preventDefault();

      if (this.validate()) {
        const data = this.getFormData();
        await this.handleSubmit(data);
      }
    });
  }

  getFormData() {
    const form = this.shadowRoot.querySelector('form');
    const formData = new FormData(form);
    return Object.fromEntries(formData);
  }
```

```
validate() {
  const form = this.shadowRoot.querySelector('form');
  return form.checkValidity();
}

async handleSubmit(data) {
  try {
    const response = await fetch('/api/contact', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });

    if (response.ok) {
      pan.publish('form.submitted', { form: 'contact', data });
      this.showSuccess();
    } else {
      throw new Error('Submission failed');
    }
  } catch (error) {
    this.showError(error.message);
  }
}

showSuccess() {
  pan.publish('notification.success', { message: 'Form submitted successfully!' });
  this.shadowRoot.querySelector('form').reset();
}

showError(message) {
  pan.publish('notification.error', { message });
}

render() {
  this.shadowRoot.innerHTML = `
    <style>
      form { max-width: 500px; }
      .field { margin-bottom: 16px; }
      label {
        display: block;
        margin-bottom: 4px;
        font-weight: 600;
      }
      input, textarea {
        width: 100%;
        padding: 8px 12px;
        border: 1px solid #cbd5e0;
        border-radius: 4px;
        }
        input:invalid, textarea:invalid {
          border-color: #fc8181;
        }
        button {
          background: #667eea;
          color: white;
          padding: 10px 24px;
          border: none;
          border-radius: 4px;
          cursor: pointer;
        }
      </style>

      <form>
        <div class="field">
          <label for="name">Name *</label>
          <input type="text" id="name" name="name" required minlength="2">
        </div>

        <div class="field">
          <label for="email">Email *</label>
          <input type="email" id="email" name="email" required>
        </div>

        <div class="field">
          <label for="message">Message *</label>
          <textarea id="message" name="message" required minlength="10" rows="5"></textarea>
        </div>

        <button type="submit">Send Message</button>
      </form>
    `;
  }
}

customElements.define('contact-form', ContactForm);
```

Two-Way Data Binding

Sync form inputs with component state:

```
class DataBoundForm extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.state = {
      firstName: '',
      lastName: '',
      email: ''
    };
  }

  connectedCallback() {
    this.render();
    this.bindInputs();
  }

  bindInputs() {
    const inputs = this.shadowRoot.querySelectorAll('input');

    inputs.forEach(input => {
      // Update state when input changes
      input.addEventListener('input', (e) => {
        this.state[e.target.name] = e.target.value;
        pan.publish('form.state.changed', { state: this.state });
      });

      // Update input when state changes
      pan.subscribe('form.state.update', (updates) => {
        if (updates[input.name] !== undefined) {
          input.value = updates[input.name];
          this.state[input.name] = updates[input.name];
        }
      });
    });
  }

  render() {
    this.shadowRoot.innerHTML = `
      <form>
        <input type="text" name="firstName" value="${this.state.firstName}" placeholder="First
        <input type="text" name="lastName" value="${this.state.lastName}" placeholder="Last Nar
        <input type="email" name="email" value="${this.state.email}" placeholder="Email">
      </form>
      <div class="preview">
        <p>Hello, ${this.state.firstName} ${this.state.lastName}!</p>
        <p>Email: ${this.state.email}</p>
      </div>
    `;
  }
}
```

Validation Strategies

Native HTML5 Validation

```html
<input type="email" required>
<input type="number" min="1" max="100">
<input type="text" pattern="[A-Za-z]{3,}" title="At least 3 letters">
<input type="url" required>
```

Custom Validation

```javascript
class ValidatedInput extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <input type="text" id="input">
      <span class="error"></span>
    `;

    const input = this.querySelector('input');
    const error = this.querySelector('.error');

    input.addEventListener('blur', () => {
      const validationResult = this.customValidate(input.value);

      if (!validationResult.valid) {
        error.textContent = validationResult.message;
        input.classList.add('invalid');
      } else {
        error.textContent = '';
        input.classList.remove('invalid');
      }
    });
  }

  customValidate(value) {
    // Custom validation logic
    if (value.length < 3) {
      return { valid: false, message: 'Must be at least 3 characters' };
    }

    if (!/^[a-zA-Z]+$/.test(value)) {
      return { valid: false, message: 'Only letters allowed' };
    }

    return { valid: true };
  }
}
```

Async Validation

```
class UsernameInput extends HTMLElement {
  connectedCallback() {
    this.render();

    const input = this.querySelector('input');
    let timeoutId;

    input.addEventListener('input', (e) => {
      clearTimeout(timeoutId);

      timeoutId = setTimeout(async () => {
        await this.checkAvailability(e.target.value);
      }, 500);
    });
  }

  async checkAvailability(username) {
    const status = this.querySelector('.status');

    if (username.length < 3) {
      status.textContent = '';
      return;
    }

    status.textContent = 'Checking...';

    try {
      const response = await fetch(`/api/check-username?username=${username}`);
      const { available } = await response.json();

      if (available) {
        status.textContent = '  Available';
        status.className = 'status success';
      } else {
        status.textContent = '  Already taken';
        status.className = 'status error';
      }
    } catch (error) {
      status.textContent = 'Could not check availability';
      status.className = 'status error';
    }
  }

  render() {
    this.innerHTML = `
      <label>Username</label>
      <input type="text" placeholder="Choose a username">
      <span class="status"></span>
    `;
  }
}
```

Error Handling

Display validation errors elegantly:

```
class FormWithErrors extends HTMLElement {
  constructor() {
    super();
    this.errors = {};
  }

  connectedCallback() {
    this.render();

    const form = this.querySelector('form');

    form.addEventListener('submit', (e) => {
      e.preventDefault();

      this.clearErrors();
      const errors = this.validateForm();

      if (Object.keys(errors).length === 0) {
        this.handleSubmit();
      } else {
        this.showErrors(errors);
      }
    });
  }

  validateForm() {
    const errors = {};
    const inputs = this.querySelectorAll('input');

    inputs.forEach(input => {
      if (!input.validity.valid) {
        errors[input.name] = this.getErrorMessage(input);
      }
    });

    return errors;
  }
```

```
getErrorMessage(input) {
  if (input.validity.valueMissing) {
    return 'This field is required';
  }
  if (input.validity.typeMismatch) {
    return `Please enter a valid ${input.type}`;
  }
  if (input.validity.tooShort) {
    return `Must be at least ${input.minLength} characters`;
  }
  if (input.validity.tooLong) {
    return `Must be no more than ${input.maxLength} characters`;
  }
  if (input.validity.patternMismatch) {
    return input.title || 'Invalid format';
  }

  return 'Invalid input';
}

showErrors(errors) {
  Object.entries(errors).forEach(([fieldName, message]) => {
    const field = this.querySelector(`[name="${fieldName}"]`);
    const errorEl = field.parentElement.querySelector('.error');

    if (errorEl) {
      errorEl.textContent = message;
      field.classList.add('invalid');
    }
  });
}

clearErrors() {
  this.querySelectorAll('.error').forEach(el => {
    el.textContent = '';
  });

  this.querySelectorAll('.invalid').forEach(el => {
    el.classList.remove('invalid');
  });
}

render() {
  this.innerHTML = `
    <form>
      <div class="field">
        <label>Email</label>
        <input type="email" name="email" required>
```

```
          <span class="error"></span>
        </div>

        <div class="field">
          <label>Password</label>
          <input type="password" name="password" required minlength="8">
          <span class="error"></span>
        </div>

        <button type="submit">Submit</button>
      </form>
    `;
  }
}
```

File Uploads

Handle file uploads with progress tracking:

```
class FileUpload extends HTMLElement {
  connectedCallback() {
    this.render();

    const input = this.querySelector('input[type="file"]');
    const button = this.querySelector('button');

    input.addEventListener('change', (e) => {
      const file = e.target.files[0];
      if (file) {
        this.showPreview(file);
        button.disabled = false;
      }
    });

    button.addEventListener('click', () => {
      const file = input.files[0];
      if (file) {
        this.uploadFile(file);
      }
    });
  }

  showPreview(file) {
    const preview = this.querySelector('.preview');

    if (file.type.startsWith('image/')) {
      const reader = new FileReader();
      reader.onload = (e) => {
        preview.innerHTML = `<img src="${e.target.result}" alt="Preview">`;
      };
      reader.readAsDataURL(file);
    } else {
      preview.innerHTML = `
        <p>${file.name}</p>
        <p>${this.formatFileSize(file.size)}</p>
      `;
    }
  }

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

    const xhr = new XMLHttpRequest();

    xhr.upload.addEventListener('progress', (e) => {
      const percent = (e.loaded / e.total) * 100;
      this.updateProgress(percent);
    });

    xhr.addEventListener('load', () => {
      if (xhr.status === 200) {
        pan.publish('file.uploaded', {
          filename: file.name,
          response: JSON.parse(xhr.response)
        });
        this.showSuccess();
      } else {
        this.showError('Upload failed');
      }
    });

    xhr.addEventListener('error', () => {
      this.showError('Upload failed');
    });

    xhr.open('POST', '/api/upload');
    xhr.send(formData);
  }

  updateProgress(percent) {
    const progress = this.querySelector('.progress-bar');
    progress.style.width = `${percent}%`;
    progress.textContent = `${Math.round(percent)}%`;
  }
  formatFileSize(bytes) {
    if (bytes < 1024) return bytes + ' B';
    if (bytes < 1024 * 1024) return (bytes / 1024).toFixed(1) + ' KB';
    return (bytes / (1024 * 1024)).toFixed(1) + ' MB';
  }

  showSuccess() {
    this.querySelector('.status').innerHTML = '  Uploaded successfully';
  }

  showError(message) {
    this.querySelector('.status').innerHTML = `  ${message}`;
  }

  render() {
    this.innerHTML = `
      <div class="upload-container">
        <input type="file" accept="image/*">
        <div class="preview"></div>
        <div class="progress">
          <div class="progress-bar"></div>
        </div>
        <button disabled>Upload</button>
        <div class="status"></div>
      </div>
    `;
  }
}

customElements.define('file-upload', FileUpload);
```

Form Submission

Handle form submission with loading states and error recovery:

```
class SmartForm extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.submitting = false;
  }

  connectedCallback() {
    this.render();

    this.shadowRoot.querySelector('form').addEventListener('submit', async (e) => {
      e.preventDefault();
      if (this.submitting) return;

      this.submitting = true;
      this.disableForm();

      try {
        const data = this.getFormData();
        await this.submitForm(data);
        this.handleSuccess();
      } catch (error) {
        this.handleError(error);
      } finally {
        this.submitting = false;
        this.enableForm();
      }
    });
  }

  getFormData() {
    const form = this.shadowRoot.querySelector('form');
    const formData = new FormData(form);
    return Object.fromEntries(formData);
  }

  async submitForm(data) {
    const response = await fetch('/api/submit', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });

    if (!response.ok) {
      const error = await response.json();
      throw new Error(error.message || 'Submission failed');
    }

    return response.json();
  }

  disableForm() {
    const inputs = this.shadowRoot.querySelectorAll('input, button, textarea');
    inputs.forEach(el => el.disabled = true);

    this.shadowRoot.querySelector('.loading').style.display = 'block';
  }

  enableForm() {
    const inputs = this.shadowRoot.querySelectorAll('input, button, textarea');
    inputs.forEach(el => el.disabled = false);

    this.shadowRoot.querySelector('.loading').style.display = 'none';
  }

  handleSuccess() {
    pan.publish('notification.success', { message: 'Form submitted successfully!' });
    this.shadowRoot.querySelector('form').reset();
  }

  handleError(error) {
    pan.publish('notification.error', { message: error.message });
  }

  render() {
    this.shadowRoot.innerHTML = `
      <style>
        .loading {
          display: none;
          text-align: center;
          padding: 16px;
        }
      </style>

      <form>
        <!-- Form fields -->
        <button type="submit">Submit</button>
      </form>

      <div class="loading">
        <div class="spinner"></div>
        <p>Submitting...</p>
      </div>
    `;
  }
}

customElements.define('smart-form', SmartForm);
```

Summary

This chapter covered: - Building accessible form components - Two-way data binding patterns - Validation strategies (native and custom) - Error handling and display - File upload with progress tracking - Form submission with loading states

Best Practices

1. **Use native validation first** - HTML5 provides powerful built-in validation
2. **Provide clear error messages** - Tell users exactly what's wrong
3. **Validate on blur** - Don't show errors while user is typing
4. **Disable during submission** - Prevent double-submission
5. **Show progress for uploads** - Users want to see progress
6. **Handle errors gracefully** - Network can fail, handle it well

Chapters 11-19: Summary Outlines

Chapter 11: Data Fetching and APIs

Key Topics:

- **REST API Integration**: Using fetch() with proper error handling
- **GraphQL Support**: Query/mutation patterns with LARC
- **WebSocket Communication**: Real-time bi-directional communication
- **Server-Sent Events**: One-way server push for live updates
- **Caching Strategies**: Cache-first, network-first, stale-while-revalidate
- **Retry Logic**: Exponential backoff and circuit breakers

Code Example - API Client:

```
class ApiClient {
  async fetch(endpoint, options = {}) {
    const response = await fetch(`/api${endpoint}`, {
      ...options,
      headers: {
        'Content-Type': 'application/json',
        ...options.headers
      }
    });

    if (!response.ok) throw new Error(`API Error: ${response.status}`);
    return response.json();
  }

  async get(endpoint) {
    return this.fetch(endpoint);
  }

  async post(endpoint, data) {
    return this.fetch(endpoint, {
      method: 'POST',
      body: JSON.stringify(data)
    });
  }
}
```

Chapter 12: Authentication and Security

Key Topics:

- **JWT Token Management**: Storing, refreshing, and validating tokens
- **The pan-auth Component**: Centralized authentication state
- **Protected Routes**: Route guards for authenticated pages
- **CORS Handling**: Cross-origin resource sharing configuration
- **XSS Prevention**: Sanitizing user input
- **CSRF Protection**: Token-based request validation

Code Example - Auth Service:

```
class AuthService {
  async login(credentials) {
    const response = await fetch('/api/auth/login', {
      method: 'POST',
      body: JSON.stringify(credentials)
    });

    const { token, user } = await response.json();

    localStorage.setItem('authToken', token);
    pan.publish('auth.login', { user });

    return { token, user };
  }

  async refresh() {
    const token = localStorage.getItem('authToken');
    const response = await fetch('/api/auth/refresh', {
      headers: { 'Authorization': `Bearer ${token}` }
    });

    const { token: newToken } = await response.json();
    localStorage.setItem('authToken', newToken);
  }

  logout() {
    localStorage.removeItem('authToken');
    pan.publish('auth.logout');
  }
}
```

Chapter 13: Server Integration

Key Topics:

- **Node.js/Express Backend**: RESTful API design for LARC
- **PHP Integration**: Connecting LARC to PHP backends
- **Python/Django**: Django REST framework integration
- **Database Patterns**: ORM usage and raw SQL
- **Real-Time**: WebSocket servers with Socket.io
- **File Serving**: Static assets and CDN integration

Node.js Example:

```
// server.js
const express = require('express');
const app = express();

app.use(express.json());
app.use(express.static('public'));

app.get('/api/users', async (req, res) => {
  const users = await db.users.findAll();
  res.json(users);
});

app.post('/api/users', async (req, res) => {
  const user = await db.users.create(req.body);
  res.json(user);
});

app.listen(3000);
```

Chapter 14: Testing

Key Topics:

- **Unit Testing**: Testing components in isolation with Web Test Runner
- **Integration Testing**: Testing component interactions
- **E2E Testing**: Playwright/Puppeteer for full user flows
- **Visual Regression**: Percy or BackstopJS for UI testing
- **Mocking**: Fetch mocks and PAN bus mocks
- **CI/CD**: GitHub Actions test automation

Test Example:

```
import { expect, fixture, html } from '@open-wc/testing';
import '../src/components/user-card.js';

describe('UserCard', () => {
  it('renders user data', async () => {
    const el = await fixture(html`
      <user-card .user=${{ name: 'John', email: 'john@example.com' }}>
      </user-card>
    `);

    expect(el.shadowRoot.querySelector('h2').textContent).to.equal('John');
    expect(el.shadowRoot.querySelector('.email').textContent).to.equal('john@example.com');
  });

  it('dispatches follow event on button click', async () => {
    const el = await fixture(html`<user-card></user-card>`);

    let eventData = null;
    el.addEventListener('follow', (e) => {
      eventData = e.detail;
    });

    el.shadowRoot.querySelector('button').click();

    expect(eventData).to.exist;
  });
});
```

Chapter 15: Performance and Optimization

Key Topics:

- **Code Splitting**: Dynamic imports for lazy loading
- **Tree Shaking**: Removing unused code
- **Lazy Loading**: Intersection Observer patterns
- **Image Optimization**: WebP, lazy loading, responsive images
- **Caching**: Service Worker caching strategies
- **Performance Monitoring**: Web Vitals and metrics

Performance Patterns:

```
// Lazy load on interaction
button.addEventListener('click', async () => {
  const { HeavyComponent } = await import('./heavy-component.js');
  // Use component
}, { once: true });

// Intersection Observer for images
const observer = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      entry.target.src = entry.target.dataset.src;
      observer.unobserve(entry.target);
    }
  });
});

document.querySelectorAll('img[data-src]').forEach(img => {
  observer.observe(img);
});
```

Chapter 16: Deployment

Key Topics:

- **Static Hosting**: Netlify, Vercel, GitHub Pages
- **CDN Configuration**: CloudFlare, AWS CloudFront
- **Environment Variables**: Managing config across environments
- **Build Scripts**: Optional production optimization
- **CI/CD Pipelines**: Automated deployment workflows
- **Monitoring**: Error tracking and analytics

Deployment Checklist:

- ☐ Minify JavaScript (optional but recommended)
- ☐ Optimize images
- ☐ Set up CDN for assets
- ☐ Configure caching headers
- ☐ Enable HTTPS
- ☐ Set up error monitoring (Sentry)
- ☐ Configure analytics (Plausible, Fathom)
- ☐ Test in all target browsers
- ☐ Set up automated deployments

Chapter 17: Component Library

Key Topics:

- **Using the Registry**: Finding and installing components
- **Contributing Components**: Publishing to the registry
- **Component Quality**: Tests, types, documentation
- **Versioning**: Semantic versioning and changelogs
- **Documentation**: API docs and usage examples
- **Design Systems**: Building consistent component libraries

Registry Integration:

```
# Install component from registry
larc add @larcjs/ui
```

```
# Publish component to registry
larc publish ./components/my-component.js
```

Chapter 18: Tooling

Key Topics:

- **LARC CLI**: create-larc-app, dev server, generators
- **VS Code Extension**: Snippets, IntelliSense, commands
- **Browser DevTools**: Debugging Web Components and Shadow DOM
- **Hot Module Reload**: Live updates without full refresh
- **Linting**: ESLint configuration for LARC
- **Formatting**: Prettier setup

VS Code Snippets:

```
{
  "LARC Component": {
    "prefix": "larc-component",
    "body": [
      "class ${1:ComponentName} extends HTMLElement {",
      "  constructor() {",
      "    super();",
      "    this.attachShadow({ mode: 'open' });",
      "  }",
      "  ",
      "  connectedCallback() {",
      "    this.render();",
      "  }",
      "  ",
      "  render() {",
      "    this.shadowRoot.innerHTML = \\`",
      "      <style>",
      "        :host { display: block; }",
      "      </style>",
      "      $2",
      "    \\`;",
      "  }",
      "}",
      "",
      "customElements.define('${3:component-name}', ${1:ComponentName});"
    ]
  }
}
```

Chapter 19: Real-World Applications

Case Study 1: E-Commerce Platform

Features: - Product catalog with search and filters - Shopping cart with persistence - Checkout flow with payment integration - User authentication and profiles - Order history and tracking

Architecture: - Components: product-card, cart-widget, checkout-form - State: IndexedDB for cart, localStorage for preferences - API: REST backend with Stripe integration - Routing: /products, /cart, /checkout, /orders

Case Study 2: Dashboard Application

Features: - Real-time data visualization - User permissions and roles - Data export functionality - Responsive layout - Dark mode support

Architecture: - Components: chart-widget, data-table, filter-bar - State: Reactive store with WebSocket updates - API: GraphQL for flexible queries - Real-time: WebSocket for live updates

Case Study 3: Blog/CMS

Features: - Markdown editor - Draft auto-save - Media library - SEO optimization - Static site generation

Architecture: - Components: markdown-editor, media-upload, post-list - State: IndexedDB for drafts - API: Headless CMS (Contentful/Strapi) - Build: Optional SSG for production

Lessons Learned:

1. Start simple, add complexity as needed
2. Use the PAN bus for cross-component communication
3. Implement offline-first for better UX
4. Test early and often
5. Profile before optimizing
6. Document your components
7. Use TypeScript for larger projects
8. Implement error boundaries
9. Monitor performance in production
10. Build progressively

Appendices

Appendix A: Web Components API Reference

Custom Elements

- `customElements.define(name, constructor, options)`
- `customElements.get(name)`
- `customElements.whenDefined(name)`
- `customElements.upgrade(root)`

Lifecycle Callbacks

- `constructor()`
- `connectedCallback()`
- `disconnectedCallback()`
- `attributeChangedCallback(name, oldValue, newValue)`
- `adoptedCallback()`

Shadow DOM

- `element.attachShadow({ mode: 'open'|'closed' })`
- `element.shadowRoot`
- `slot.assignedNodes()`
- `slot.assignedElements()`

Appendix B: PAN Bus API Reference

Core Methods

```
// Publish
pan.publish(topic, data)

// Subscribe
const unsubscribe = pan.subscribe(topic, handler)

// Request/Response
const result = await pan.request(topic, data, timeout)
pan.respond(topic, handler)
```

```
// Unsubscribe
unsubscribe()
```

Topic Patterns

- `user.login` - Specific event
- `user.*` - All user events
- `*.error` - All error events
- `*` - All events (debugging)

Appendix C: Component API Reference

Built-in Components

pan-store - Attributes: `persist`, `namespace` - Methods: `getState()`, `setState(updates)`, `subscribe(path, handler)` - Events: `state-changed`

pan-router - Child: `<pan-route path="" component="" guard="">` - Events: `router.navigated`, `router.not-found` - PAN Topics: `router.navigate`, `router.back`, `router.forward`

pan-fetch - Attributes: `url`, `method`, `auto` - Properties: `data`, `loading`, `error` - Events: `data-loaded`, `fetch-error`

pan-auth - Methods: `login(credentials)`, `logout()`, `refresh()` - Properties: `user`, `authenticated` - Events: `auth-changed`

Appendix D: Migration Guides

From React

Concepts: - JSX → Template literals - Props → Attributes/properties - State → Instance properties - Context → PAN bus or Context API pattern - Hooks → Lifecycle callbacks - Redux → pan-store or custom store

Example:

```
// React
function UserCard({ user }) {
  const [expanded, setExpanded] = useState(false);

  return (
    <div onClick={() => setExpanded(!expanded)}>
      <h2>{user.name}</h2>
      {expanded && <p>{user.bio}</p>}
    </div>
  );
}

// LARC
class UserCard extends HTMLElement {
  constructor() {
    super();
    this.expanded = false;
  }

  set user(value) {
    this._user = value;
    this.render();
  }

  connectedCallback() {
    this.addEventListener('click', () => {
      this.expanded = !this.expanded;
      this.render();
    });
    this.render();
  }

  render() {
    this.innerHTML = `
      <div>
        <h2>${this._user.name}</h2>
        ${this.expanded ? `<p>${this._user.bio}</p>` : ''}
      </div>
    `;
  }
}
```

From Vue

Concepts: - Templates → Template literals - v-model → Two-way binding patterns - Computed → Getters - Watch → Observe patterns - Vuex → pan-store

From Angular

Concepts: - Decorators → Static properties - Dependency Injection → Constructor patterns - Services → Modules - RxJS → PAN bus observables - NgRx → pan-store

Appendix E: Resources

Official Documentation

- LARC Docs: https://larcjs.com/docs
- Component Registry: https://components.larcjs.com
- GitHub: https://github.com/larcjs

Web Standards

- MDN Web Components: https://developer.mozilla.org/en-US/docs/Web/Web_Components
- Custom Elements Spec: https://html.spec.whatwg.org/multipage/custom-elements.html

- Shadow DOM Spec: https://dom.spec.whatwg.org/#shadow-trees

Community

- Discord: https://discord.gg/larcjs
- Forum: https://forum.larcjs.com
- Twitter: @larcjs

Learning Resources

- Web Components Tutorial: https://webcomponents.org
- ES Modules Guide: https://javascript.info/modules
- IndexedDB Tutorial: https://javascript.info/indexeddb

Tools

- LARC CLI: https://www.npmjs.com/package/create-larc-app
- VS Code Extension: Search "LARC" in marketplace
- Component Analyzer: https://github.com/larcjs/analyzer

Example Projects

- TodoMVC: https://github.com/larcjs/todomvc
- E-Commerce: https://github.com/larcjs/ecommerce-example
- Dashboard: https://github.com/larcjs/dashboard-example