

Building with LARC: A Reference Manual

A Comprehensive Guide to Modern Web Applications

LARC Project Contributors

December 2025

Contents

1	Introduction	1
1.1	What Is LARC?	1
1.1.1	The Philosophy in Action	2
1.2	Who Should Use This Book	2
1.2.1	Who This Book Is NOT For	2
1.3	How to Use This Book	3
1.3.1	Two Ways to Read	3
1.3.2	What’s Inside	3
1.4	Relationship to “Learning LARC”	3
1.5	Prerequisites and Assumptions	4
1.5.1	Required Knowledge	4
1.5.2	Helpful But Not Required	4
1.5.3	What You Don’t Need	5
1.5.4	Software Requirements	5
1.6	Book Conventions and Notation	5
1.6.1	Typographical Conventions	5
1.6.2	Code Examples	6
1.6.3	Message Topics	6
1.6.4	Component Naming	6
1.6.5	Attribute Syntax	7
1.6.6	API Signatures	7
1.6.7	File Paths and Imports	7
1.6.8	Example Applications	8
1.7	What’s Next	8
2	The Philosophy of LARC	9
2.1	Introduction: Why Another Approach?	9
2.2	Build Tool Fatigue and the Web Components Promise	9
2.2.1	The Real Problem: Developer Onboarding Overhead	9
2.2.2	The Web Components Disappointment	10
2.2.3	The PAN Experiment: How Far Can We Go?	11
2.2.4	The Build System Burden	13
2.2.5	The Philosophy That Emerged	14
2.3	Message-Passing Architecture: Learning from Distributed Systems	15
2.3.1	The Inspiration: Actor Model and Message Queues	15
2.3.2	The PAN Bus: Pub/Sub for Components	15

2.3.3	Topic-Based Routing: Organization Without Central Control	16
2.3.4	Retained Messages: State Without Stores	16
2.3.5	Benefits of Message-Passing	17
2.4	DOM-Native Communication Principles	19
2.4.1	Leveraging Existing Standards	19
2.4.2	BroadcastChannel: The Foundation	19
2.4.3	Custom Elements: Native Components	20
2.5	Zero-Dependency, Zero-Build Philosophy	21
2.5.1	But What About Production?	21
2.6	Progressive Enhancement and Graceful Degradation	21
2.6.1	Layering Functionality	21
2.6.2	Graceful Degradation	23
2.6.3	Browser Support Strategy	24
2.7	Comparison to Other Approaches	25
2.7.1	LARC vs. Redux	25
2.7.2	Other State Management Approaches	26
2.7.3	Summary: Architectural Trade-offs	27
2.8	Additional Practical Benefits	27
2.8.1	Simplified Component Testing	27
2.8.2	Multi-Tab Synchronization	28
2.9	Conclusion: A Pragmatic Philosophy	29
3	The LARC Story	31
3.1	The Problem That Wouldn't Go Away	31
3.2	Enter LARC	31
3.3	Design Decisions and Trade-offs	32
3.3.1	The Great Hydration Debate	32
3.3.2	Reactivity Without the Reactivity Tax	32
3.3.3	HTML Templates: Tagged or Literal?	33
3.4	The PAN: A Network Protocol for Components	33
3.4.1	The Origin of PAN	34
3.4.2	PAN Design Principles	34
3.5	Evolution and Growing Pains	35
3.5.1	The Streaming Crisis	35
3.5.2	The TypeScript Years	35
3.5.3	The Build-Tool Wars	36
3.6	Real-World Use Cases	36
3.6.1	Live Data Dashboards	36
3.6.2	Progressive Enhancement Sites	36
3.6.3	IoT Control Panels	36
3.7	Community and Ecosystem	37
3.7.1	The Component Library	37
3.7.2	The Plugin Ecosystem	37
3.7.3	The Documentation Journey	38
3.8	Lessons Learned	38
3.9	The Future	38
3.10	Epilogue: The Name	39

4	Core Concepts	41
4.1	Introduction: The Building Blocks	41
4.2	The Message Bus: Your Application's Nervous System	41
4.2.1	What Is a Message Bus?	41
4.2.2	How Does It Work?	42
4.2.3	Configuration and Capabilities	43
4.3	Pub/Sub Pattern: Fire and Forget (But Don't Actually Forget)	43
4.3.1	The Classic Pattern	43
4.3.2	Wildcards: Subscribe to Patterns	45
4.3.3	The Global Wildcard Problem	45
4.4	Topics and Routing: Addressing Your Messages	46
4.4.1	Naming Conventions	46
4.4.2	Semantic Routing	47
4.4.3	Anti-Patterns to Avoid	47
4.5	Message Lifecycle: Birth, Death, and Resurrection	48
4.5.1	The Lifecycle of a Message	48
4.5.2	Message Structure	48
4.5.3	Validation and Size Limits	48
4.5.4	Retained Messages: The Last Value Cache	49
4.5.5	Memory Management	50
4.6	Components and Composition: Building Blocks	50
4.6.1	What Is a Component in LARC?	50
4.6.2	Communication Patterns	51
4.6.3	Composition Examples	52
4.6.4	The Autoloader: Zero-Config Imports	53
4.7	State Management Strategies	53
4.7.1	The Three Flavors of State	53
4.7.2	Local State: Keep It Simple	53
4.7.3	Shared State: Use Retained Messages	54
4.7.4	The State Publisher Pattern	55
4.7.5	Persistent State: Add Storage	56
4.8	Event Envelopes and Metadata	57
4.8.1	The Message Envelope	57
4.8.2	Message IDs	57
4.8.3	Timestamps	58
4.8.4	Custom Headers	58
4.8.5	The Request/Reply Pattern	59
4.9	Putting It All Together	60
4.10	What We've Learned	64
4.11	Key Takeaways	64
5	Getting Started	65
5.1	Prerequisites	65
5.2	Installation Options	65
5.2.1	Option 1: CDN (Fastest for Testing)	65
5.2.2	Option 2: NPM Installation	67
5.2.3	Option 3: Git Clone (Full Repository)	67
5.2.4	Option 4: Create LARC App (Coming Soon)	68

5.3	Development Environment Setup	68
5.3.1	Directory Structure	69
5.3.2	Editor Configuration	69
5.3.3	Local Development Server	70
5.4	Browser DevTools Setup	71
5.4.1	Chrome DevTools	71
5.4.2	Firefox Developer Tools	72
5.4.3	Safari Web Inspector	72
5.4.4	LARC DevTools Extension (Optional)	72
5.5	Your First LARC Application	73
5.5.1	Step 1: Project Setup	73
5.5.2	Step 2: Create Directory Structure	74
5.5.3	Step 3: Build the Task Form Component	74
5.5.4	Step 4: Build the Task List Component	76
5.5.5	Step 5: Build the Task Item Component	79
5.5.6	Step 6: Run the Application	82
5.5.7	What You Just Built	82
5.6	Serving Static Files	82
5.6.1	Development Servers (Local)	82
5.6.2	Production Servers	83
5.6.3	CORS and Module Loading	83
5.7	Browser Requirements and Compatibility	84
5.7.1	Supported Browsers	84
5.7.2	Feature Detection	84
5.7.3	Polyfills for Older Browsers	85
5.7.4	Mobile Browser Support	86
5.8	Common Troubleshooting	86
5.8.1	Problem: Components Don't Load	86
5.8.2	Problem: PAN Bus Messages Not Received	87
5.8.3	Problem: Styles Not Applying	87
5.8.4	Problem: localStorage Quota Exceeded	88
5.8.5	Problem: CORS Errors	88
5.9	Next Steps	89
6	Basic Message Flow	91
6.1	The Anatomy of a Message	91
6.2	Publishing Your First Message	92
6.2.1	Publishing from Components	92
6.3	Subscribing to Topics	94
6.3.1	Subscription Example: Notification System	94
6.4	Wildcard Patterns: The Power of Asterisks	95
6.4.1	Practical Wildcard Example: Audit Logger	96
6.5	Message Retention: The PAN Bus Remembers	98
6.5.1	Controlling Retention	99
6.5.2	Retention Gotchas	99
6.6	Message Ordering and Synchronization	99
6.7	Unsubscribing and Cleanup	100
6.7.1	Cleanup Patterns	100

6.8	Debugging Message Flow	102
6.8.1	Console Logging	102
6.8.2	Conditional Logging	102
6.8.3	Message Inspector Component	102
6.9	Performance Considerations	104
6.10	Common Patterns and Anti-Patterns	105
6.10.1	Pattern: Command-Query Separation	105
6.10.2	Pattern: Namespacing	105
6.10.3	Anti-Pattern: Publishing Without Data	105
6.10.4	Anti-Pattern: Overloading Topics	106
6.11	Wrapping Up	106
7	Working with Components	107
7.1	Web Components: A Brief Refresher	107
7.2	The Component Lifecycle	108
7.3	Shadow DOM: To Use or Not to Use?	109
7.3.1	When to Use Shadow DOM	109
7.3.2	When to Avoid Shadow DOM	111
7.4	Connecting Components via the PAN Bus	111
7.4.1	Component 1: Product Catalog	112
7.4.2	Component 2: Shopping Cart	113
7.4.3	Component 3: Cart Badge	114
7.4.4	Putting It All Together	115
7.5	Component Communication Patterns	115
7.5.1	Pattern: Request-Response	115
7.5.2	Pattern: Command Pattern	117
7.5.3	Pattern: State Projection	118
7.5.4	Pattern: Event Aggregation	119
7.6	Reusable Component Design	120
7.6.1	Principle 1: Single Responsibility	120
7.6.2	Principle 2: Clear API	120
7.6.3	Principle 3: Composition Over Configuration	121
7.6.4	Principle 4: Progressive Enhancement	121
7.6.5	Principle 5: Accessibility First	122
7.7	Advanced Component Techniques	123
7.7.1	Technique: Lazy Rendering	123
7.7.2	Technique: Virtual Scrolling	124
7.7.3	Technique: Memoization	126
7.8	Testing Components	126
7.9	Wrapping Up	127
8	State Management	129
8.1	Local vs. Shared State	129
8.2	The State Store Pattern	131
8.3	State Persistence with localStorage	133
8.3.1	localStorage Limitations	134
8.4	State Persistence with IndexedDB	134
8.5	State Persistence with OPFS	138

8.5.1	When to Use OPFS vs. IndexedDB	141
8.6	Synchronization Patterns	141
8.6.1	Pattern: Optimistic Updates	141
8.6.2	Pattern: Debounced Sync	143
8.6.3	Pattern: Polling	145
8.7	Conflict Resolution	146
8.7.1	Strategy: Last Write Wins	146
8.7.2	Strategy: Timestamps	146
8.7.3	Strategy: Version Vectors	147
8.7.4	Strategy: Conflict Detection and User Intervention	148
8.8	State Snapshots and Time Travel	148
8.9	Derived State	150
8.10	Performance Considerations	151
8.11	Wrapping Up	151
9	Routing and Navigation	153
9.1	Understanding Client-Side Routing	153
9.2	The pan-routes Component	154
9.2.1	Route Matching Order	154
9.3	Route Parameters and Pattern Matching	155
9.3.1	Basic Parameters	155
9.3.2	Multiple Parameters	156
9.3.3	Optional Parameters	156
9.4	Programmatic Navigation	156
9.4.1	Navigation Options	157
9.5	Navigation Guards	158
9.5.1	Implementing Auth Guards	158
9.5.2	Route Transition Guards	159
9.6	Deep Linking and URL State	160
9.6.1	Query Parameters	160
9.6.2	Hash Fragments	161
9.7	History Management	162
9.7.1	Push vs. Replace	162
9.7.2	Listening to History Changes	162
9.7.3	Preserving Scroll Position	163
9.8	Nested Routes and Layouts	164
9.9	Link Handling and Active States	165
9.10	SEO Considerations	165
9.10.1	Server-Side Rendering (SSR)	166
9.10.2	Meta Tags and Titles	166
9.10.3	Prerendering	167
9.11	Putting It All Together	168
10	Forms and User Input	171
10.1	The Fundamentals of Form Handling	171
10.1.1	Basic Form Structure	171
10.1.2	Two-Way Data Binding	173
10.2	Validation Strategies	175

10.2.1	HTML5 Built-in Validation	175
10.2.2	Custom Validation Logic	176
10.2.3	Debounced Validation	179
10.3	Schema-Driven Forms	180
10.3.1	Defining a Schema	180
10.3.2	Schema Form Component	182
10.3.3	Using the Schema Form	186
10.4	File Uploads	186
10.4.1	Basic File Upload	186
10.4.2	Upload Progress with XMLHttpRequest	189
10.5	Rich Text Editing	190
10.5.1	Markdown Editor	190
10.5.2	Integrating Third-Party Editors	192
10.6	Form State Management	194
10.7	Conclusion	197
11	Data Fetching and APIs	199
11.1	The Foundation: Fetch API	199
11.1.1	Basic GET Request	199
11.1.2	POST Request with JSON	201
11.1.3	Request Configuration	201
11.2	Error Handling and Retries	205
11.2.1	Retry Logic	205
11.2.2	Circuit Breaker Pattern	206
11.3	Caching Strategies	207
11.3.1	In-Memory Cache	207
11.3.2	LocalStorage Cache	209
11.3.3	Stale-While-Revalidate	211
11.4	GraphQL Integration	212
11.4.1	GraphQL Client	212
11.4.2	Using GraphQL Queries	213
11.5	WebSocket Communication	214
11.5.1	WebSocket Client	214
11.5.2	Real-Time Chat Component	216
11.6	Server-Sent Events (SSE)	218
11.6.1	SSE Client	218
11.6.2	Live Notifications	220
11.7	Request Cancellation	222
11.7.1	AbortController	222
11.8	Putting It All Together	224
12	Authentication and Authorization	227
12.1	Understanding Authentication vs. Authorization	227
12.2	JWT Authentication: The Token Economy	227
12.2.1	Creating an Authentication Service	228
12.2.2	Securing API Requests	232
12.3	Protected Routes and Navigation Guards	234
12.4	Role-Based Access Control (RBAC)	236

12.4.1	Designing a Flexible RBAC System	236
12.4.2	Using Authorization in Components	239
12.5	Session Management Best Practices	240
12.5.1	Implementing Secure Session Storage	240
12.5.2	Session Timeout and Activity Tracking	242
12.6	Security Best Practices	244
12.6.1	Input Validation and Sanitization	244
12.6.2	CSRF Protection	245
12.7	Putting It All Together: A Complete Login Flow	246
12.8	What We've Learned	251
13	Real-time Features	253
13.1	Understanding Real-time Communication	253
13.2	WebSocket Integration: The Two-Way Street	253
13.2.1	Building a WebSocket Client	254
13.2.2	Using WebSockets in Components	259
13.3	Server-Sent Events: One-Way Data Flow	264
13.3.1	Building an SSE Client	264
13.3.2	Live Activity Feed with SSE	267
13.4	BroadcastChannel: Cross-Tab Communication	272
13.4.1	Building a Tab Synchronization Service	272
13.4.2	Synchronizing Authentication Across Tabs	274
13.5	Web Workers: Background Processing	275
13.5.1	Creating a Data Processing Worker	275
13.5.2	Worker Manager	277
13.6	Real-time Collaboration Patterns	279
13.7	What We've Learned	283
14	File Management	285
14.1	Understanding OPFS: Your App's Private File System	285
14.2	Getting Started with OPFS	285
14.3	Building a File Browser Component	291
14.4	File Upload and Download	302
14.5	Storage Quota Management	309
14.6	File Type Filtering and Search	314
14.7	Putting It All Together	316
14.8	What We've Learned	319
15	Theming and Styling	321
15.1	CSS Custom Properties: Variables That Actually Work	321
15.1.1	Defining a Theme System	322
15.1.2	Dark Mode: Inverting Without Inverting	324
15.1.3	System Preference Detection	325
15.2	Theme Provider Component	325
15.3	Theme-Aware Components	329
15.4	Theme Switcher Component	330
15.5	Smooth Transitions Between Themes	332
15.6	Scoped Themes for Components	333

15.7	Responsive Design with Custom Properties	334
15.8	Multiple Brand Themes	335
15.9	Performance Considerations	336
15.9.1	1. Minimize Transitions	336
15.9.2	2. Use CSS Containment	337
15.9.3	3. Avoid Deep Custom Property Lookups	337
15.9.4	4. Batch Theme Changes	337
15.10	Accessibility in Theming	338
15.10.1.1	1. Maintain Contrast Ratios	338
15.10.2.2	2. Respect Reduced Motion	338
15.10.3.3	3. Provide High Contrast Mode	338
15.10.4.4	4. Test with Screen Readers	338
15.11	Wrapping Up	339
16	Performance Optimization	341
16.1	Message Filtering and Routing Efficiency	341
16.1.1	Pattern: Specific Topic Subscriptions	341
16.1.2	Pattern: Early Returns	342
16.1.3	Pattern: Unsubscribe Aggressively	342
16.1.4	Implementing Message Throttling	343
16.1.5	Debouncing Message Publishers	343
16.1.6	Batching Messages	344
16.2	Component Lazy Loading	345
16.2.1	Lazy Loading Off-Screen Components	345
16.2.2	Code Splitting Routes	347
16.3	Virtual Scrolling for Large Lists	348
16.3.1	Dynamic Item Heights	351
16.4	Debouncing and Throttling	352
16.4.1	Debounce Utility	352
16.4.2	Throttle Utility	353
16.4.3	RequestAnimationFrame Throttling	355
16.5	Memory Management	356
16.5.1	Pattern: Clean Up Subscriptions	356
16.5.2	Pattern: Remove Event Listeners	356
16.5.3	Pattern: Clear Timers	357
16.5.4	Pattern: Cancel Pending Promises	357
16.5.5	Pattern: Weak References for Caches	358
16.6	Bundle Size Optimization	358
16.6.1	1. Tree Shaking	358
16.6.2	2. Dynamic Imports	359
16.6.3	3. Avoid Large Dependencies	359
16.6.4	4. Code Splitting by Route	359
16.6.5	5. Minification	360
16.6.6	6. Compression	360
16.7	Performance Monitoring	360
16.8	Wrapping Up	362
17	Testing Strategies	363

17.1	The Testing Pyramid for LARC	363
17.2	Unit Testing Components	364
17.2.1	Setting Up Vitest	364
17.2.2	Testing a Simple Component	364
17.2.3	Testing Components with Attributes	366
17.3	Mocking the PAN Bus	368
17.3.1	Creating a Mock Bus	368
17.3.2	Using the Mock Bus	370
17.3.3	Testing Message-Driven Components	371
17.4	Integration Testing Message Flows	373
17.5	Testing Async Operations	375
17.5.1	Testing Promises	375
17.5.2	Testing with Async/Await	377
17.5.3	Testing Timeouts and Intervals	378
17.6	End-to-End Testing	379
17.6.1	Setting Up Playwright	379
17.6.2	Testing Theme Switching	381
17.7	Test Utilities and Helpers	381
17.7.1	Component Test Harness	382
17.7.2	Message Bus Test Helper	384
17.8	Test Coverage	386
17.9	Wrapping Up	386
18	Error Handling and Debugging	387
18.1	Error Boundaries: Containing the Chaos	387
18.1.1	Understanding Error Propagation	387
18.1.2	Implementing Error Handlers	388
18.1.3	Global Error Handler Component	389
18.2	Message Tracing: Following the Breadcrumbs	390
18.2.1	Built-in Message Tracing	390
18.2.2	Message Flow Visualization	392
18.3	DevTools Integration: Professional Debugging	394
18.3.1	Custom Console Formatters	394
18.3.2	Source Maps and Stack Traces	395
18.3.3	Performance Profiling	396
18.4	Logging Strategies: Write Once, Debug Forever	397
18.4.1	Structured Logging	397
18.4.2	Log Aggregation and Search	399
18.5	Common Pitfalls and Solutions	401
18.5.1	Pitfall #1: Message Type Typos	401
18.5.2	Pitfall #2: Infinite Message Loops	402
18.5.3	Pitfall #3: Stale Closures in Handlers	403
18.5.4	Pitfall #4: Async Race Conditions	403
18.6	Debugging Checklist	404
18.7	Conclusion	405
19	Advanced Patterns	407
19.1	Message Forwarding and Bridging	407

19.1.1	Basic Message Forwarding	407
19.1.2	Bidirectional Bridging	408
19.1.3	Message Translation	409
19.2	Multi-Bus Architectures	410
19.2.1	Domain-Segregated Buses	410
19.2.2	Hierarchical Bus Structure	412
19.3	Backend Integration Strategies	414
19.3.1	API Gateway Component	414
19.3.2	WebSocket Integration	416
19.3.3	GraphQL Integration	418
19.4	Micro-Frontends with LARC	421
19.4.1	Module Federation Pattern	421
19.4.2	Shell Application Pattern	423
19.5	Plugin Systems	424
19.5.1	Plugin Registry	424
19.6	Middleware Patterns	427
19.6.1	Message Middleware	427
19.6.2	Async Middleware with Error Handling	430
19.7	Conclusion	431
20	Deployment and Production	433
20.1	Build Considerations (Or Lack Thereof)	433
20.1.1	The No-Build Approach	433
20.1.2	When You Actually Need a Build Step	434
20.1.3	Minimal Build with esbuild	434
20.1.4	Code Splitting for Larger Apps	435
20.1.5	TypeScript Integration (Optional)	436
20.2	CDN Deployment	437
20.2.1	Static File Hosting	437
20.2.2	Configuration Files	437
20.2.3	Asset Fingerprinting	439
20.3	Caching Strategies	439
20.3.1	Browser Cache Headers	439
20.3.2	Service Worker Caching	440
20.3.3	API Response Caching	442
20.4	Performance Monitoring	444
20.4.1	Real User Monitoring (RUM)	444
20.4.2	Custom Performance Marks	446
20.4.3	Bundle Size Monitoring	447
20.5	Production Debugging	448
20.5.1	Source Maps	448
20.5.2	Remote Error Tracking	449
20.5.3	Feature Flags	450
20.6	Versioning and Upgrades	452
20.6.1	Semantic Versioning	452
20.6.2	Update Notifications	452
20.6.3	Rolling Deployments	453
20.7	Deployment Checklist	454

20.8	Monitoring Production Health	454
20.9	Conclusion	456
21	Core Components Reference	457
21.1	pan-bus	457
21.1.1	Overview	457
21.1.2	When to Use	458
21.1.3	Installation and Setup	458
21.1.4	Attributes	458
21.1.5	Methods	459
21.1.6	Events	461
21.1.7	Working Examples	464
21.1.8	Common Issues and Solutions	468
21.2	pan-theme-provider	469
21.2.1	Overview	469
21.2.2	When to Use	469
21.2.3	Installation and Setup	469
21.2.4	Attributes	470
21.2.5	Methods	470
21.2.6	Events	471
21.2.7	Working Examples	472
21.2.8	Common Issues and Solutions	475
21.3	pan-theme-toggle	476
21.3.1	Overview	476
21.3.2	When to Use	476
21.3.3	Installation and Setup	476
21.3.4	Attributes	477
21.3.5	Methods	477
21.3.6	Events	477
21.3.7	Working Examples	477
21.3.8	Common Issues and Solutions	480
21.4	pan-routes	480
21.4.1	Overview	480
21.4.2	When to Use	481
21.4.3	Installation and Setup	481
21.4.4	Methods	481
21.4.5	Route Configuration	486
21.4.6	Working Examples	489
21.4.7	Common Issues and Solutions	491
21.5	Summary	493
22	Data Components	495
22.1	Overview	495
22.2	pan-store: Reactive State Management	496
22.2.1	Purpose	496
22.2.2	When to Use	496
22.2.3	When Not to Use	496
22.2.4	Installation	496

22.2.5	API Reference	496
22.2.6	bind(element, store, mapping, options)	500
22.2.7	Complete Working Examples	501
22.2.8	Common Issues and Solutions	505
22.3	pan-idb: IndexedDB Integration	506
22.3.1	Purpose	506
22.3.2	When to Use	507
22.3.3	When Not to Use	507
22.3.4	Installation	507
22.3.5	Attributes Reference	507
22.3.6	PAN Topics	508
22.3.7	Methods Reference	511
22.3.8	Complete Working Examples	512
22.3.9	Common Issues and Solutions	518
22.4	Combining pan-store and pan-idb	520
22.5	Related Components	521
22.6	Best Practices	521
22.7	Conclusion	522
23	UI Components	523
23.1	pan-files: File System Browser	523
23.1.1	Overview and Purpose	523
23.1.2	When to Use pan-files	524
23.1.3	Installation and Setup	524
23.1.4	Attributes Reference	524
23.1.5	Methods Reference	525
23.1.6	Events Reference	527
23.1.7	Complete Working Example	528
23.1.8	Related Components	530
23.1.9	Common Issues and Solutions	530
23.2	pan-markdown-editor: Rich Markdown Editor	531
23.2.1	Overview and Purpose	531
23.2.2	When to Use pan-markdown-editor	531
23.2.3	Installation and Setup	531
23.2.4	Attributes Reference	532
23.2.5	Methods Reference	532
23.2.6	Toolbar Actions	533
23.2.7	Keyboard Shortcuts	534
23.2.8	Events Reference	534
23.2.9	Complete Working Example	535
23.2.10	Related Components	537
23.2.11	Common Issues and Solutions	537
23.3	pan-markdown-renderer: Markdown Display	538
23.3.1	Overview and Purpose	538
23.3.2	When to Use pan-markdown-renderer	539
23.3.3	Installation and Setup	539
23.3.4	Attributes Reference	539
23.3.5	Methods Reference	540

23.3.6	Supported Markdown Syntax	540
23.3.7	Styling with CSS Variables	542
23.3.8	Events Reference	542
23.3.9	Complete Working Example	543
23.3.10	Related Components	546
23.3.11	Common Issues and Solutions	546
23.4	Component Integration Patterns	547
23.4.1	Pattern 1: Markdown Note-Taking App	547
23.4.2	Pattern 2: Documentation Viewer	548
23.4.3	Pattern 3: Split-View Editor	548
23.5	Conclusion	549
24	Integration Components	551
24.1	pan-data-connector	551
24.1.1	Overview	551
24.1.2	When to Use	552
24.1.3	Installation and Setup	552
24.1.4	Attributes	553
24.1.5	Topics	554
24.1.6	Authentication Integration	556
24.1.7	Complete Examples	557
24.1.8	Related Components	560
24.1.9	Common Issues and Solutions	560
24.2	pan-graphql-connector	561
24.2.1	Overview	561
24.2.2	When to Use	561
24.2.3	Installation and Setup	562
24.2.4	Attributes	563
24.2.5	GraphQL Operation Scripts	563
24.2.6	Response Path Mapping	564
24.2.7	Topics	564
24.2.8	Authentication Integration	564
24.2.9	Complete Examples	564
24.2.10	Related Components	566
24.2.11	Common Issues and Solutions	566
24.3	pan-websocket	567
24.3.1	Overview	567
24.3.2	When to Use	567
24.3.3	Installation and Setup	568
24.3.4	Attributes	568
24.3.5	Topics	569
24.3.6	Methods	570
24.3.7	Message Format	571
24.3.8	Complete Examples	571
24.3.9	Related Components	574
24.3.10	Common Issues and Solutions	575
24.4	pan-sse	576
24.4.1	Overview	576

24.4.2	When to Use	576
24.4.3	Installation and Setup	576
24.4.4	Attributes	577
24.4.5	Server-Side SSE Format	577
24.4.6	Topics	578
24.4.7	Complete Examples	578
24.4.8	Related Components	581
24.4.9	Common Issues and Solutions	581
24.5	Architectural Patterns	582
24.5.1	Combining Multiple Connectors	582
24.5.2	Optimistic Updates with Rollback	585
24.5.3	Conflict Resolution	586
24.5.4	Offline Support	588
24.5.5	Rate Limiting and Backpressure	590
24.5.6	Security Considerations	591
24.6	Summary	593
25	Utility Components	595
25.1	Overview	595
25.2	pan-debug: Message Tracing and Debugging	595
25.2.1	Overview	595
25.2.2	When to Use	596
25.2.3	Installation and Setup	596
25.2.4	API Reference	597
25.2.5	Helper Functions	601
25.2.6	Complete Working Examples	602
25.2.7	Common Issues and Solutions	607
25.3	pan-forwarder: HTTP Message Forwarding	608
25.3.1	Overview	608
25.3.2	When to Use	608
25.3.3	Installation and Setup	609
25.3.4	Attributes	609
25.3.5	Properties	611
25.3.6	Request Body Format	611
25.3.7	Deduplication	611
25.3.8	Complete Working Examples	612
25.3.9	Server-Side Implementation	617
25.3.10	Related Components	618
25.3.11	Common Issues and Solutions	618
25.4	Best Practices	621
25.4.1	Debug Component Best Practices	621
25.4.2	Forwarder Best Practices	621
25.5	Summary	621
26	Message Topics Reference	623
26.1	Topic Format and Structure	623
26.1.1	Standard Format	623
26.1.2	Topic Examples	623

26.1.3	Naming Rules	624
26.2	Topic Patterns and Matching	624
26.2.1	Exact Match	624
26.2.2	Single-Segment Wildcard	624
26.2.3	Global Wildcard	625
26.2.4	Pattern Matching Examples	625
26.3	Reserved Topic Namespaces	625
26.3.1	pan:* Namespace (System Internal)	625
26.3.2	sys:* Namespace (System Reserved)	626
26.3.3	Application Namespaces	626
26.4	CRUD Topic Patterns	626
26.4.1	List Operations	626
26.4.2	Item Operations	628
26.4.3	Item Events	630
26.4.4	Per-Item State	630
26.5	State Management Topics	631
26.5.1	Global State	631
26.5.2	Scoped State	631
26.6	Events vs Commands	632
26.6.1	Events	632
26.6.2	Commands	632
26.7	Domain-Specific Patterns	633
26.7.1	Authentication	633
26.7.2	Navigation	634
26.7.3	UI Components	634
26.7.4	Forms	634
26.7.5	Data Synchronization	635
26.8	Best Practices	635
26.8.1	Topic Naming	635
26.8.2	Topic Catalog	635
26.8.3	Performance Considerations	636
26.9	Summary	636
27	Event Envelope Specification	639
27.1	Overview	639
27.2	Message Envelope Structure	639
27.2.1	Complete Format	639
27.2.2	Minimal Message	640
27.3	Field Specifications	640
27.3.1	topic (required)	640
27.3.2	data (required)	641
27.3.3	id (optional, auto-generated)	643
27.3.4	ts (optional, auto-generated)	643
27.3.5	retain (optional)	644
27.3.6	replyTo (optional)	645
27.3.7	correlationId (optional)	646
27.3.8	headers (optional)	646
27.3.9	clientId (internal)	648

27.4	Message Examples	648
27.4.1	Simple Event	648
27.4.2	Retained State	648
27.4.3	Request Message	649
27.4.4	Reply Message	649
27.4.5	Message with Headers	649
27.4.6	Error Response	650
27.5	Response Payload Conventions	650
27.5.1	Success Response	650
27.5.2	Error Response	650
27.5.3	Example Error Codes	651
27.6	Message Size Limits	651
27.6.1	Default Limits	651
27.6.2	Configuration	651
27.6.3	Size Estimation	651
27.6.4	Handling Size Limits	651
27.7	Validation	652
27.7.1	Topic Validation	652
27.7.2	Data Validation	652
27.7.3	Size Validation	653
27.8	Internal System Messages	653
27.8.1	pan:sys.ready	653
27.8.2	pan:sys.error	653
27.8.3	pan:sys.stats	654
27.9	Summary	654
28	Configuration Options	655
28.1	PAN Bus Configuration	655
28.1.1	Attribute Reference	655
28.1.2	Complete Configuration Example	659
28.2	PanClient Configuration	659
28.2.1	Constructor Options	660
28.2.2	Subscription Options	660
28.2.3	Request Options	661
28.3	Component Configuration	661
28.3.1	Standard Attributes	661
28.3.2	Component-Specific Configuration	662
28.4	Global Configuration	663
28.4.1	Window Configuration	663
28.4.2	Feature Flags	663
28.5	Environment Variables	663
28.5.1	NODE_ENV	663
28.5.2	LARC_DEBUG	663
28.5.3	LARC_MAX_RETAINED	664
28.5.4	LARC_RATE_LIMIT	664
28.6	Configuration Profiles	664
28.6.1	Development Profile	664
28.6.2	Production Profile	664

28.6.3	High-Throughput Profile	665
28.6.4	Memory-Constrained Profile	665
28.6.5	Testing Profile	665
28.7	Runtime Configuration	666
28.7.1	Get Current Configuration	666
28.7.2	Clear Retained Messages	666
28.7.3	Get Statistics	666
28.8	Configuration Best Practices	666
28.8.1	Start Conservative	666
28.8.2	Monitor and Tune	667
28.8.3	Environment-Specific Configuration	667
28.8.4	Document Your Configuration	668
28.9	Configuration Checklist	668
28.10	Summary	668
29	Migration Guide	671
29.1	General Migration Strategy	671
29.2	Version 0.x to 1.0 Migration	671
29.2.1	Component Registration Changes	671
29.2.2	PAN Bus API Refinement	672
29.2.3	Attribute Handling	672
29.3	Version 1.x to 2.0 Migration	673
29.3.1	TypeScript Integration	673
29.3.2	Shadow DOM Adoption	673
29.3.3	Async Component Initialization	674
29.4	Version 2.x to 3.0 Migration	674
29.4.1	Reactive State Management	674
29.4.2	PAN Bus Namespacing	675
29.4.3	Performance Optimizations	676
29.5	Deprecation Timeline	676
29.5.1	Currently Deprecated (Remove in 4.0)	676
29.5.2	Planned Deprecations (4.0+)	677
29.6	Breaking Changes Checklist	677
29.7	Migration Tools	677
29.7.1	Automated Refactoring	677
29.7.2	Manual Review Points	678
29.8	Rollback Strategy	678
29.9	Getting Help	678
30	Recipes and Patterns	679
30.1	Recipe 1: Lazy-Loading Components	679
30.2	Recipe 2: Form Validation Component	680
30.3	Recipe 3: Infinite Scroll List	683
30.4	Recipe 4: Toast Notification System	685
30.5	Recipe 5: Debounced Search Input	687
30.6	Recipe 6: Modal Dialog	689
30.7	Recipe 7: State Persistence	691
30.8	Recipe 8: Drag and Drop	692

30.9	Recipe 9: Responsive Image	694
30.10	Recipe 10: Event Bus Bridge	695
30.11	Common Patterns	696
30.11.1	Pattern: Component Composition	696
30.11.2	Pattern: Higher-Order Components	696
30.11.3	Pattern: Singleton Services	696
30.12	Anti-Patterns to Avoid	697
30.12.1	Anti-Pattern: Tight Coupling	697
30.12.2	Anti-Pattern: Massive Components	697
30.12.3	Anti-Pattern: Ignoring Lifecycle	698
30.12.4	Anti-Pattern: Manual Memory Leaks	698
31	Glossary	699
31.1	A	699
31.2	B	699
31.3	C	699
31.4	D	700
31.5	E	700
31.6	F	701
31.7	H	701
31.8	I	701
31.9	L	701
31.10	M	701
31.11	N	702
31.12	O	702
31.13	P	702
31.14	R	702
31.15	S	702
31.16	T	703
31.17	U	703
31.18	V	703
31.19	W	703
31.20	LARC-Specific Terms	704
31.21	Web Standards References	704
31.22	Acronyms and Abbreviations	704
31.23	Related Concepts	705
31.24	Further Reading	705
32	Resources	707
32.1	Official Documentation	707
32.1.1	Primary Documentation	707
32.1.2	Companion Books	707
32.2	Community Resources	708
32.2.1	Forums and Discussion	708
32.2.2	Social Media	708
32.3	Code Examples and Templates	708
32.3.1	Example Applications	708
32.3.2	Component Showcases	709

32.4	Development Tools	709
32.4.1	Browser Extensions	709
32.4.2	Editor Extensions	709
32.4.3	Command-Line Tools	709
32.5	Learning Resources	710
32.5.1	Video Tutorials	710
32.5.2	Blog Posts and Articles	710
32.5.3	Podcasts	710
32.6	Related Projects and Technologies	711
32.6.1	Web Components Standards	711
32.6.2	Message Bus Patterns	711
32.6.3	Complementary Technologies	711
32.7	Backend Integration	711
32.7.1	LARC-Compatible Backends	711
32.7.2	API Design Guides	711
32.8	Testing and Quality	712
32.8.1	Testing Resources	712
32.8.2	Performance Resources	712
32.9	Contributing and Extending	712
32.9.1	Contribution Guides	712
32.9.2	Governance and Roadmap	712
32.10	Package Registries	713
32.10.1	NPM Packages	713
32.10.2	CDN Distributions	713
32.11	Books and Long-Form Resources	713
32.11.1	Recommended Reading	713
32.11.2	Academic Papers	713
32.12	Deployment and Hosting	714
32.12.1	Hosting Platforms	714
32.12.2	Deployment Guides	714
32.13	Events and Training	714
32.13.1	Conferences	714
32.13.2	Workshops and Training	714
32.14	Community Projects	714
32.14.1	Notable Applications Built with LARC	714
32.14.2	Open Source Projects	715
32.15	Stay Updated	715
32.15.1	Newsletters	715
32.15.2	Release Notes	715
32.16	Getting Help	715
32.16.1	Support Options	715
33	Index	717
33.1	A	717
33.2	B	717
33.3	C	718
33.4	D	719
33.5	E	720

33.6 F	720
33.7 G	721
33.8 H	721
33.9 I	721
33.10J	721
33.11K	721
33.12L	721
33.13M	721
33.14N	722
33.15O	722
33.16P	722
33.17Q	723
33.18R	723
33.19S	724
33.20T	725
33.21U	725
33.22V	725
33.23W	726
33.24X	726
33.25Y	726
33.26Z	726
33.27Appendices	726
33.28Components Quick Reference	726
34 Colophon	729
34.1 About the Cover Animal	729
34.2 About the Cover Illustration	730
34.3 Fonts	730
34.4 Production Notes	730
34.5 Acknowledgments	730
35 Learning LARC	731
35.1 The Web Has Grown Up. It's Time Our Apps Did Too.	731
35.2 About the Author	731

Chapter 1

Introduction

Welcome to *Building with LARC: A Reference Manual*, your comprehensive guide to the Lightweight Asynchronous Relay Core framework. If you’re holding this book (or reading it on a screen, as modern humans do), you’re about to dive into one of the most refreshingly simple yet surprisingly powerful approaches to building web applications. No, we’re not overselling it. Well, maybe a little. But stick with us.

1.1 What Is LARC?

LARC (Lightweight Asynchronous Relay Core) is a zero-build, browser-native web component framework built around a message-passing architecture called PAN (Page Area Network). If that sentence made you think “Wait, another JavaScript framework?” — we get it. But LARC is different in ways that matter.

Here’s the elevator pitch: LARC gives you the power of modern component-based architecture without requiring build tools, dependency hell, or sacrificing your weekend to webpack configuration. It’s built entirely on web standards (Custom Elements, Shadow DOM, ES Modules), uses a DOM-native pub/sub messaging system to coordinate components, and can be added to any project with a single `<script>` tag.

```
<script type="module" src="/src/pan.mjs"></script>

<!-- That's it. You're done. Now use components: -->
<pan-card title="Hello World">
  <pan-button>Click me</pan-button>
</pan-card>
```

The secret sauce is the PAN bus — a lightweight messaging backbone inspired by the CAN (Controller Area Network) buses found in automobiles. Just as a car’s sensors, motors, and computers communicate over a shared bus without knowing about each other’s internals, LARC components coordinate through published messages and subscriptions. This solves the notorious “Web Component silo problem” where components can’t easily communicate without tight coupling.

Think of it this way: Web Components give you 80% of what you need to build modern applications. LARC provides the missing 20% — the coordination layer, auto-loading system, and

state management infrastructure that makes Web Components genuinely practical for real-world applications.

1.1.1 The Philosophy in Action

LARC embraces a “zero-build development, optimized production” philosophy. During development, you write code and refresh your browser — no webpack watch, no hot-module replacement gymnastics, no waiting for recompilation. In production, you can still use your favorite build tools to optimize bundles, but you’re not *required* to.

This isn’t about being anti-tooling or nostalgic for the “good old days.” Modern build tools solve real problems at scale. LARC simply argues they shouldn’t be mandatory for every project, especially during the exploratory and iterative phases of development. The browser has evolved significantly since 2015 — it’s time we trusted it to do what it does well.

1.2 Who Should Use This Book

This book is written for **experienced programmers** who understand web development fundamentals and want a comprehensive reference for building applications with LARC. You should be comfortable with:

- **JavaScript** (ES6+): You know your promises from your `async/await`, understand modules, destructuring, and arrow functions. You don’t need to be a TC39 committee member, but closures shouldn’t make you nervous.
- **HTML and CSS**: You can write semantic markup and understand the box model. Shadow DOM will be explained, but the basics should be familiar territory.
- **Web Components**: Some exposure to Custom Elements and Shadow DOM is helpful but not required. We’ll cover what you need, but won’t re-teach the entire Web Components spec.
- **Basic HTTP and REST principles**: You know what GET and POST mean, understand APIs at a conceptual level, and have integrated with a backend before.
- **Command-line basics**: You can start a local development server (whether that’s Python’s `http.server`, Node’s `http-server`, or something else) and navigate your file system.

1.2.1 Who This Book Is NOT For

If you’re completely new to web development, start with *Learning LARC* (more on that shortly). This reference manual assumes you already know how to build web applications and want to learn LARC specifically. We won’t explain what the DOM is or why JavaScript runs in browsers.

Similarly, if you’re looking for a gentle tutorial that holds your hand through your first “Hello World,” the companion book *Learning LARC* is a better starting point. This manual is comprehensive, thorough, and occasionally exhausting in its detail — perfect for reference, less ideal for bedtime reading.

1.3 How to Use This Book

Building with LARC is structured as a **reference manual**, not a tutorial. Think of it like the Perl Programming books or the classic O'Reilly references: comprehensive, authoritative, and designed for looking things up more than reading cover-to-cover (though you're welcome to try — we won't judge).

1.3.1 Two Ways to Read

As a Reference: Jump directly to the chapter covering your current problem. Building a file management system? Chapter 14 has you covered. Need to implement authentication? Chapter 12 is your friend. Each chapter is relatively self-contained, with cross-references when deeper context is needed.

As a Deep Dive: Read Part I (Foundations) to understand LARC's philosophy and architecture, then work through Part II (Building Applications) in order to see how the pieces fit together. Part III (Component Reference) becomes your API documentation, and Part IV (Appendices) serves as quick-reference material.

1.3.2 What's Inside

The book is organized into four parts:

Part I: Foundations (Chapters 1-5) covers the conceptual underpinnings: what LARC is, why it exists, how the PAN messaging architecture works, and how to set up your development environment. If you're new to LARC, read this first.

Part II: Building Applications (Chapters 6-20) is where the rubber meets the road. These task-oriented chapters show you how to accomplish specific goals: managing state, handling routing, fetching data, implementing authentication, optimizing performance, and deploying to production. Each chapter follows a consistent structure: problem statement, concepts, step-by-step implementation, complete example, variations, troubleshooting, and best practices.

Part III: Component Reference (Chapters 21-25) provides exhaustive documentation for every core LARC component. Think of it as your API reference — detailed attribute tables, method signatures, event specifications, and practical examples for components like `pan-bus`, `pan-store`, `pan-routes`, `pan-markdown-editor`, and dozens more.

Part IV: Appendices (A-G) contains supporting material: message topic conventions, event envelope specifications, configuration options, migration guides, code recipes, a glossary, and resource links.

1.4 Relationship to “Learning LARC”

If *Building with LARC* is the comprehensive reference manual, *Learning LARC* is its approachable older sibling — the tutorial-focused book that teaches LARC from the ground up through hands-on examples and clear explanations.

The two books complement each other:

Learning LARC (the tutorial book) is organized around **learning progressions**. It starts with “Hello World” and gradually builds to complex, real-world applications. Each chapter introduces

new concepts in a carefully scaffolded way, with exercises, quizzes, and projects that reinforce understanding. If you're new to LARC or prefer learning by doing, start there.

Building with LARC (this book) is organized around **tasks and references**. It assumes you already understand the basics and want to accomplish specific goals or look up specific component APIs. It's comprehensive where *Learning LARC* is curated, exhaustive where the tutorial is selective, and reference-oriented where the tutorial is narrative-driven.

Here's a practical guideline:

- **New to LARC?** Start with *Learning LARC*, then return to this book for reference and advanced topics.
- **Experienced with LARC?** This book is your primary resource. Keep it on your desk (or bookmarked in your browser).
- **Learning a specific feature?** Check *Learning LARC* for tutorial coverage, then consult this book's relevant chapters for comprehensive details.
- **Debugging or optimizing?** This book's troubleshooting sections, appendices, and detailed component references are what you need.

Think of it like the difference between *Learning Perl* and *Programming Perl* — one teaches you the language, the other documents it thoroughly. Both are valuable, just at different times.

1.5 Prerequisites and Assumptions

To get the most from this book, you should have:

1.5.1 Required Knowledge

1. **JavaScript Fundamentals:** You're comfortable with modern JavaScript (ES6+), including modules, `async/await`, destructuring, template literals, and arrow functions. You understand scope, closures, and prototypes at least conceptually.
2. **Web Development Basics:** You've built websites or web applications before. You understand client-server architecture, HTTP methods, and how browsers request resources.
3. **HTML/CSS:** You can write semantic HTML5 and know enough CSS to style components. You don't need to be a design wizard, but you should understand selectors, specificity, and layout basics.
4. **Development Environment:** You can set up a local development server and use a code editor. Experience with browser DevTools (Console, Network, Elements tabs) is highly recommended.

1.5.2 Helpful But Not Required

- **TypeScript:** LARC supports TypeScript with official type definitions (`@larcjs/core-types`), but JavaScript examples are used throughout this book for clarity and accessibility.
- **Build Tools:** While LARC is designed for zero-build development, understanding webpack, Rollup, or Vite helps when you want optimized production builds.

- **React/Vue/Angular:** Experience with component-based frameworks provides useful context for understanding LARC’s architecture, but isn’t necessary.
- **Web Components APIs:** We’ll explain what you need about Custom Elements, Shadow DOM, and HTML Templates, but prior exposure helps.

1.5.3 What You Don’t Need

You **don’t** need to know:

- Advanced computer science algorithms or data structures
- Backend programming (though integration examples are provided)
- DevOps or deployment infrastructure (basics are covered in Chapter 20)
- Browser internals or JavaScript engine implementations

This book meets you where experienced web developers typically are — comfortable with the fundamentals and ready to learn a new tool.

1.5.4 Software Requirements

Throughout this book, examples assume you have:

- A **modern web browser**: Chrome 90+, Firefox 88+, Safari 14+, or Edge 90+. Most examples work across all modern browsers, but specific features may note browser requirements.
- A **local development server**: Any static file server works. Examples use Python’s built-in HTTP server, but Node’s `http-server`, PHP’s built-in server, or VS Code’s Live Server extension are equally valid.
- A **code editor**: VS Code is recommended (with the LARC extension for enhanced support), but any editor with JavaScript/HTML syntax highlighting works.
- **Git** (optional): Useful for cloning examples and exploring the LARC source code.

No build tools, no Node.js, no npm (unless you want to install LARC from npm) — just a browser and a way to serve static files. That’s the point.

1.6 Book Conventions and Notation

This section consolidates all the conventions used throughout this book for easy reference.

1.6.1 Typographical Conventions

This book uses standard O’Reilly conventions:

- **Bold** indicates new terms, emphasis, or UI elements (“click the **Save** button”)
- *Italic* indicates filenames, URLs, example text, or emphasis
- **Constant width** indicates code, commands, component names, attributes, and technical terms

We also use special callout blocks to highlight important information:

NOTE: Additional context, clarifications, or interesting tangents that won’t break your code if you skip them.

WARNING: Pay attention here — this is where developers commonly make mistakes or encounter surprising behavior.

TIP: Practical advice from the trenches, often learned the hard way.

1.6.2 Code Examples

Throughout this book, you’ll see code examples in various formats:

Inline code appears like `this` for short snippets, commands, file names, and HTML attributes.

Block code appears in fenced sections with syntax highlighting:

```
// JavaScript examples look like this
class MyComponent extends HTMLElement {
  connectedCallback() {
    this.textContent = 'Hello from LARC!';
  }
}
```

```
<!-- HTML examples look like this -->
<pan-card title="Example">
  <p>Component content goes here</p>
</pan-card>
```

Command-line examples begin with a prompt:

```
$ python3 -m http.server 8000
$ npm install @larcjs/core
```

The `$` indicates your shell prompt — don’t type it.

1.6.3 Message Topics

LARC applications communicate through the PAN bus using hierarchical topic patterns. Throughout this book, topics follow conventions:

`namespace.entity.action`

For example:

- `user.auth.login` — User authentication login
- `cart.items.add` — Add item to shopping cart
- `theme.changed` — Theme change notification

Wildcards are common:

- `user.*` — All user-related messages
- `*.changed` — All change notifications
- `#` — All messages (use sparingly)

1.6.4 Component Naming

LARC components follow Web Component naming conventions:

- All lowercase with hyphens: `pan-button`, `pan-card`, `my-custom-component`
- Must contain at least one hyphen (per Web Component spec)
- Core LARC components start with `pan-` prefix
- Your components can use any prefix or no prefix

1.6.5 Attribute Syntax

Component attributes are shown in HTML as:

```
<pan-card
  title="Card Title"
  variant="elevated"
  theme="dark">
```

Boolean attributes (true when present, false when absent):

```
<pan-button disabled>Can't click me</pan-button>
<pan-markdown-editor readonly></pan-markdown-editor>
```

1.6.6 API Signatures

JavaScript APIs are documented with type annotations for clarity:

```
// Method signature
publish(topic: string, payload: any, options?: object): void

// Usage example
bus.publish('user.login', { userId: 123 });
```

These aren't real TypeScript — just pseudocode for clarity. TypeScript users should reference the official `@larcjs/core-types` package for actual type definitions.

1.6.7 File Paths and Imports

File paths are shown in UNIX format (`/src/components/my-component.mjs`) but translate naturally to Windows (`\src\components\my-component.mjs`).

Example projects use a consistent structure:

```
/src/
  /components/    # Your custom components
  /utils/         # Helper functions
  /styles/        # Global styles
/core/           # LARC core (@larcjs/core)
/ui/             # LARC components (@larcjs/ui)
index.html       # Entry point
larc-config.mjs  # Path configuration
```

Imports use ES6 module syntax:

```
import { PanBus } from '/core/pan-bus.mjs';
import MyComponent from '/my-component.mjs';
```

Production applications typically use import maps or CDN URLs (covered in Chapter 20).

1.6.8 Example Applications

Each chapter includes complete, runnable examples. You can:

1. **Type them manually** — Best for learning and retention
2. **Copy from the book** — Faster, still educational
3. **Clone from GitHub** — All examples are available at github.com/larcjs/examples

Examples are self-contained where possible, with any dependencies clearly noted.

1.7 What's Next

Now that you understand what LARC is, who this book is for, and how to use it effectively, you're ready to dive deeper. Chapter 2 explores the philosophy behind LARC — why message-passing architecture, why zero-build, and how LARC compares to other approaches. Understanding the “why” makes the “how” much clearer.

If you prefer learning by doing, feel free to skip ahead to Chapter 5 (Getting Started) and return to the philosophical foundations later. We won't tell anyone.

If you're the type who needs to understand principles before touching code (guilty), Chapter 2 awaits.

Either way, welcome to LARC. We think you're going to like it here.

“The web grew up. Now we get to build like it.” — Christopher Robison, Foreword to *Learning LARC*

Chapter 2

The Philosophy of LARC

2.1 Introduction: Why Another Approach?

If you’ve been developing web applications for any length of time, you’ve probably noticed that the complexity keeps climbing. Each year brings new tools, new frameworks, and new “best practices” that somehow require even more configuration files, build steps, and abstract concepts to master.

But here’s the uncomfortable truth: most of this complexity isn’t solving your actual problems. It’s solving problems created by previous layers of abstraction.

LARC takes a different approach. Instead of adding another layer of abstraction on top of the existing stack, it asks a more fundamental question: **What if we could build modern web applications using the platform itself?**

This chapter explores the philosophy behind LARC—the “why” that drives every design decision. Understanding this philosophy will help you use LARC more effectively and make better architectural decisions in your own projects.

2.2 Build Tool Fatigue and the Web Components Promise

2.2.1 The Real Problem: Developer Onboarding Overhead

Let’s be honest about what actually drove the creation of LARC: **build tool fatigue**.

After years of watching developers spend more time configuring webpack, fighting with Babel, debugging TypeScript configs, and learning complex build pipelines than actually building features, it became clear that something was fundamentally wrong. New team members would join a project and spend their first week (or month!) learning the build system, understanding the toolchain, and navigating the maze of configuration files—all before they could write a single line of actual application code.

The barrier to entry had become absurd:

```
# A typical modern project setup  
npm install  
# Wait 10 minutes  
# Install 1,200+ dependencies
```

```
# 400MB of node_modules

# Then fight with:
- webpack.config.js (200 lines)
- babel.config.js
- tsconfig.json
- .eslintrc.js
- postcss.config.js
- vite.config.js
- And dozens more...
```

This isn't what the web was supposed to be. The web platform itself requires none of this. You can write an HTML file, open it in a browser, and it works. So why did we accept all this complexity?

2.2.2 The Web Components Disappointment

Around the same time, Web Components promised to solve the reusability problem. The pitch was compelling: **write a component once, use it anywhere**. No framework lock-in. True portability. Native browser support. It sounded perfect.

But the reality was disappointing. Web Components solved the technical problem of creating custom elements, but they didn't solve the practical problem of building real applications. You still needed:

- A way to manage state across components
- A way for components to communicate
- A way to handle data fetching and updates
- Build tools (ironically) for anything non-trivial

The most frustrating part was this: every web component I built ended up tightly coupled to its current context anyway. A “user-profile” component needed direct access to the user object. A “product-card” needed specific methods from a parent component. A “notification-list” needed to import the notification service directly.

```
// This felt like defeat
class UserProfile extends HTMLElement {
  connectedCallback() {
    // Tightly coupled to global state
    const user = window.appState.user;

    // Tightly coupled to specific API
    this.api = window.userService;

    // Can't reuse this component in another project
    // because it depends on these specific globals
    this.render(user);
  }
}
```

What was the point? Web Components were supposed to be **reusable**, but I was building components that were just as tightly coupled as any framework component—except now with extra steps of

abstraction. The technology gave us encapsulation, but it didn't give us independence.

It felt like using a more verbose syntax to achieve the same result. Why write a Custom Element if it can't actually be portable? Why bother with the Web Components API if you still need to wire everything together manually with brittle global dependencies?

Web Components gave us the syntax for reusable components, but not the architecture for building with them. They became yet another piece that needed framework scaffolding around them to be useful.

2.2.3 The PAN Experiment: How Far Can We Go?

LARC started as a simple experiment with the **PAN (Page Area Network) concept**—a message bus for browser components inspired by MQTT and the Actor model. The initial question was straightforward: “What if components could communicate through messages instead of direct coupling?”

Of course, I knew about the pub/sub pattern. I knew that web components could technically communicate via `postMessage()` or `BroadcastChannel`. But here's the thing: both of those APIs are low-level primitives. They give you the **mechanism** for sending messages, but not the **architecture** for organizing them.

With `postMessage()`, you'd write code like this:

```
// Sender
window.postMessage({ type: 'USER_LOGIN', payload: user }, '*');

// Receiver
window.addEventListener('message', (event) => {
  if (event.data.type === 'USER_LOGIN') {
    handleLogin(event.data.payload);
  }
});
```

And with `BroadcastChannel`:

```
// Sender
const channel = new BroadcastChannel('app-events');
channel.postMessage({ type: 'USER_LOGIN', user: user });

// Receiver
const channel = new BroadcastChannel('app-events');
channel.onmessage = (event) => {
  if (event.data.type === 'USER_LOGIN') {
    handleLogin(event.data.user);
  }
};
```

Both approaches have the same problem: **every project rolls their own tightly coupled message format**. You're back to the same coupling issues, just at a different level. Instead of coupling to `window.appState`, you're coupling to a specific message structure: `{ type: 'USER_LOGIN',`

payload: ... } vs { type: 'USER_LOGIN', user: ... }. Different projects would have different conventions, different payload shapes, different type naming schemes.

I saw this pattern repeated everywhere. Developers were independently creating custom messaging buses on top of `postMessage` and `BroadcastChannel`—each slightly different, each solving the same problems in slightly different ways. Everyone was building their own topic routing, their own message envelope format, their own subscription management.

It struck me: **there should be a well-defined message standard**. Not just “send messages,” but a consistent format for:

- **Topic-based routing:** `user.login` not { type: 'USER_LOGIN' }
- **Message envelopes:** Consistent structure with data, metadata, timestamps
- **Subscription patterns:** Wildcards like `user.*` or `*.login`
- **Retained messages:** State that persists for late subscribers
- **Lifecycle management:** Automatic cleanup when components disconnect

The web had given us the transport layer (`BroadcastChannel`), but we needed an application layer—a protocol that components could depend on without coupling to specific implementations.

That’s when PAN moved from “let’s try message passing” to “let’s define a standard.”

The moment this clicked was transformative. Instead of:

```
// Before: Tightly coupled
class UserProfile extends HTMLElement {
  connectedCallback() {
    const user = window.appState.user; // Coupled to specific global
    this.render(user);
  }
}
```

Components could do this:

```
// After: Loosely coupled through messages
class UserProfile extends HTMLElement {
  connectedCallback() {
    // Subscribe to a topic - any component can publish to it
    panClient.subscribe('user.profile', ({ data }) => {
      this.render(data);
    });

    // Request current data
    panClient.publish('user.profile.request');
  }
}
```

Now the component doesn’t know **where** the user data comes from. It doesn’t import anything. It doesn’t depend on specific globals. It just subscribes to a topic. This component can be dropped into **any** project that has a PAN bus—different backend, different state management, different everything. As long as something publishes to ‘user.profile’, this component works.

This was the reusability promise that Web Components couldn’t deliver alone. The PAN bus

provided the missing piece: a standard way for components to communicate without coupling.

But that experiment led to a more interesting question: **“How far can we go without any external, heavy, locked-in framework?”**

Not from an anti-framework ideology—frameworks solve real problems and have their place. But from a pragmatic curiosity: the web platform has matured dramatically over the past decade. The problems React and its contemporaries solved 15 years ago—managing DOM updates, providing component models, handling events, supporting modern JavaScript—have largely been addressed by open standards now:

- **Custom Elements** provide a native component model
- **ES Modules** provide native code organization
- **Shadow DOM** provides style encapsulation
- **JavaScript itself** now has classes, `async/await`, destructuring, template literals
- **CSS** has custom properties, grid, flexbox, container queries
- **Fetch API** handles HTTP requests
- **BroadcastChannel** enables cross-context messaging

The question became: if we use these standards directly, without transpilation, without heavy frameworks, **can we build real applications that are actually simpler to understand and maintain?**

2.2.4 The Build System Burden

Here’s what really pushed the experiment forward: watching talented developers struggle not with code logic, not with algorithms, not with architecture—but with **build configuration**.

Consider this scenario (repeated countless times):

Developer: “I need to add a simple feature—just fetch some data and display it.”

Traditional Framework Workflow (React/Vue/Angular/etc.):

1. Pull latest code
2. Install/update dependencies (because the lockfile changed... again)
3. Create/update component file
4. Write fetch logic
5. Update state/store/actions
6. Update template/JSX
7. Update routing if needed
8. Update types/interfaces
9. Fix lint errors
10. Fix type errors
11. Start dev server (or wait for it to restart)
12. Wait for bundler to rebuild (don’t forget about the source maps!)
13. Debug through layers of framework abstractions
14. Update tests and mocks
15. Commit changes
16. Push branch
17. Wait for CI: transpile, bundle, test, lint, type-check
18. Fix config/environment differences CI complains about

19. Merge when green
20. Build pipeline runs again for production artifacts
21. Deploy artifacts to server
22. Verify it works in production

Time spent: 2 hours **Time actually coding:** 15 minutes

This is backwards. The tools should be invisible, not the primary challenge. Let's compare with the LARC workflow:

LARC Workflow

1. Pull latest code
2. Open component or create a new one
3. Write fetch logic using standard `fetch()`
4. Drop the results into the DOM (template literal, `innerHTML`, whatever fits)
5. Refresh browser tab
6. Debug directly in the browser with no abstraction layer
7. Commit + push
8. CI runs lint/tests (no build pipeline)
9. Deploy static files
10. Done

That's it.

No bundler.

No transpiler.

No dev server.

No JSX.

No toolchain waiting room.

2.2.5 The Philosophy That Emerged

These experiments crystallized LARC's core philosophy:

- **Use the platform.** The web has matured. Build on standards directly instead of abstracting them away.
- **Message-passing over shared state.** Components that communicate through messages can truly be reused anywhere.
- **Make builds optional.** Use builds for production optimization, not as a development requirement.
- **Enable true portability.** Components that depend only on web standards and a lightweight message bus work in any project.

This isn't anti-framework ideology—React, Vue, and Svelte solve real problems. But for many projects, the web platform itself is sufficient. When it is, why take on unnecessary complexity?

2.3 Message-Passing Architecture: Learning from Distributed Systems

2.3.1 The Inspiration: Actor Model and Message Queues

LARC's solution comes from distributed systems theory. When building systems with multiple independent processes, you don't use shared state—you use message passing. Each process maintains its own state and communicates with other processes by sending messages.

This pattern appears throughout computing:

- **Operating systems:** Processes communicate via message queues
- **Actor model:** Erlang, Akka, and Orleans use message passing for concurrency
- **Microservices:** Services communicate via HTTP, message queues, or event streams
- **MQTT:** IoT devices coordinate through pub/sub messaging
- **Event-driven architecture:** Systems react to events without direct coupling

These patterns work because they solve fundamental problems:

1. **Decoupling:** Components don't need to know about each other
2. **Scalability:** Add components without modifying existing ones
3. **Resilience:** Failures are isolated and don't cascade
4. **Flexibility:** Swap implementations without changing interfaces

LARC brings this pattern to the browser with the **PAN (Page Area Network) bus**—a publish/subscribe message system that allows components to communicate without knowing about each other.

2.3.2 The PAN Bus: Pub/Sub for Components

Here's how state management looks with the PAN bus:

```
// Publishing a message (any component can do this)
panClient.publish('cart.item.add', {
  id: 'product-123',
  name: 'Coffee Mug',
  price: 12.99,
  quantity: 1
}, { retain: true });

// Subscribing to messages (any component can listen)
panClient.subscribe('cart.item.add', ({ data }) => {
  console.log('Item added to cart:', data);
  updateCartDisplay();
});

// Multiple components can react to the same message
panClient.subscribe('cart.item.add', ({ data }) => {
  // Update cart count badge
  document.getElementById('cart-count').textContent =
    getCartItems().length;
});
```

```
});

panClient.subscribe('cart.item.add', ({ data }) => {
  // Show notification
  showNotification(`${data.name} added to cart`);
});
```

No action creators. No reducers. No `connect()` functions. Just messages flowing through the system.

2.3.3 Topic-Based Routing: Organization Without Central Control

The PAN bus uses topic-based routing, similar to MQTT or RabbitMQ. Topics are hierarchical strings separated by dots:

```
user.profile.update
cart.item.add
cart.item.remove
cart.checkout.start
inventory.product.update
ui.modal.open
ui.sidebar.toggle
```

Components can subscribe to specific topics or use wildcards:

```
// Subscribe to specific topic
panClient.subscribe('cart.item.add', handler);

// Subscribe to all cart events
panClient.subscribe('cart.*', handler);

// Subscribe to all add events
panClient.subscribe('*.item.add', handler);

// Subscribe to everything
panClient.subscribe('*', handler);
```

This creates natural organization:

- **Namespace separation:** Different domains have different topic prefixes
- **Granular subscriptions:** Subscribe only to what you need
- **Discoverability:** Topic names describe what they do
- **No central registry:** Components define their own topics

2.3.4 Retained Messages: State Without Stores

Traditional message buses are ephemeral—messages are delivered once and then disappear. But web applications need state: when a new component loads, it needs the current state, not just future updates.

LARC solves this with **retained messages**—the last message published to a topic can be stored and delivered to new subscribers:

```
// Publish with retention
panClient.publish('user.preferences.theme', 'dark', { retain: true });

// Later, a new component subscribes...
panClient.subscribe('user.preferences.theme', ({ data }) => {
  // Immediately receives 'dark' even though it subscribed after publication
  applyTheme(data);
});
```

This provides state management without a central store:

- **Current state:** Retrieved by subscribing to retained topics
- **State updates:** Published as new messages
- **No special API:** Same subscribe() method works for both
- **Automatic synchronization:** New components automatically get current state

Think of retained messages as “stateful topics”—each topic can hold exactly one value, similar to a key in a key-value store, but with pub/sub semantics.

2.3.5 Benefits of Message-Passing

Why is this better than centralized state management?

1. **** Zero coupling between components ****

Components don’t import each other. They don’t know each other exist. They just publish and subscribe to topics. This means:

- **Add components freely:** New components can subscribe to existing topics
- **Remove components safely:** Unsubscribing doesn’t break other components
- **Test in isolation:** Mock the bus, not the entire application state
- **Reuse anywhere:** Components work in any application with a PAN bus

2. **** Progressive complexity ****

Start simple and add sophistication only when needed:

```
// Simple: Direct message handling
panClient.subscribe('cart.item.add', ({ data }) => {
  items.push(data);
  render();
});

// Advanced: Add validation
panClient.subscribe('cart.item.add', ({ data }) => {
  if (validateItem(data)) {
    items.push(data);
    render();
  } else {
    panClient.publish('cart.error', 'Invalid item');
  }
});
```

```
// Sophisticated: Add persistence
panClient.subscribe('cart.item.add', async ({ data }) => {
  if (validateItem(data)) {
    items.push(data);
    await saveToLocalStorage(items);
    panClient.publish('cart.synced', items);
    render();
  }
});
```

No refactoring required. Just add features incrementally.

3. **** Natural debugging****

Every message flows through the bus. Want to debug state changes? Subscribe to all topics:

```
// Development debugging
panClient.subscribe('*', ({ topic, data, meta }) => {
  console.log(`[${topic}]`, data, meta);
});
```

Want to trace a specific feature? Subscribe to its topics:

```
// Track all cart operations
panClient.subscribe('cart.*', ({ topic, data }) => {
  console.log('Cart event:', topic, data);
});
```

Compare this to stepping through Redux reducers or tracing Vue reactivity. Message-passing makes data flow explicit and observable.

4. **** Multi-component coordination****

Traditional state management struggles with coordinating multiple components:

```
// Redux: Components must import actions and know about each other
import { openModal } from './modalActions';
import { pauseVideo } from './videoActions';
import { saveFormData } from './formActions';

function handleSaveAndClose() {
  dispatch(saveFormData(data));
  dispatch(pauseVideo());
  dispatch(openModal('confirmation'));
}
```

With message-passing, components coordinate through messages:

```
// Publisher doesn't know who's listening
panClient.publish('form.save.request', formData);

// Multiple components react independently
```

```
panClient.subscribe('form.save.request', ({ data }) => {
  // Form component saves data
  saveToDatabase(data);
});

panClient.subscribe('form.save.request', () => {
  // Video component pauses playback
  pauseVideo();
});

panClient.subscribe('form.save.request', () => {
  // Modal component shows confirmation
  showModal('confirmation');
});
```

Each component handles its own concerns. No central coordinator needed.

2.4 DOM-Native Communication Principles

2.4.1 Leveraging Existing Standards

LARC doesn't invent new communication patterns—it leverages patterns already present in the DOM:

1. **Events:** The DOM uses events for component interaction
2. **Attributes:** Components configure via attributes
3. **Properties:** JavaScript interfaces use properties
4. **Custom Elements:** The browser provides a component system

The PAN bus extends these patterns to handle application-level communication:

```
<!-- DOM events: Local communication -->
<button onclick="handleClick()">Click me</button>

<!-- PAN messages: Application-level communication -->
<pan-button topic="ui.action.click">Click me</pan-button>
```

This creates a natural mental model:

- **DOM events** = Component-to-parent communication
- **PAN messages** = Component-to-application communication
- **Attributes** = Configuration
- **Properties** = JavaScript API

2.4.2 BroadcastChannel: The Foundation

Under the hood, LARC can use the browser's `BroadcastChannel` API—a standard way to communicate between browser contexts (tabs, windows, iframes, workers):

```
// Native BroadcastChannel
const channel = new BroadcastChannel('my-app');
```

```
channel.postMessage({ type: 'update', data: 'value' });
channel.onmessage = (event) => {
  console.log('Received:', event.data);
};
```

The PAN bus builds on this foundation but adds:

- Topic-based routing with wildcards
- Retained messages for state
- Message metadata and timestamps
- Synchronous and asynchronous delivery
- Type safety and validation (optional)

By building on web standards, LARC remains simple, debuggable, and future-proof.

2.4.3 Custom Elements: Native Components

LARC components are standard Custom Elements:

```
class PanCard extends HTMLElement {
  connectedCallback() {
    this.render();

    // Subscribe to theme changes
    this.subscription = panClient.subscribe('theme.change', ({ data }) => {
      this.applyTheme(data);
    });
  }

  disconnectedCallback() {
    // Clean up subscriptions
    this.subscription.unsubscribe();
  }

  render() {
    this.innerHTML = `
      <div class="card">
        <slot></slot>
      </div>
    `;
  }
}

customElements.define('pan-card', PanCard);
```

No framework required. Just standard Web Components that happen to communicate via the PAN bus.

2.5 Zero-Dependency, Zero-Build Philosophy

We’ve already seen the problems with build tool complexity. LARC’s solution is straightforward: use only features that browsers understand natively—ES Modules, standard JavaScript (ES2015+), CSS Custom Properties, and Web Components.

The result is the simple development workflow shown earlier: edit a file, refresh the browser, see changes instantly. No bundler, no transpiler, no waiting.

2.5.1 But What About Production?

The zero-build philosophy applies to **development**, not necessarily production. For production, you can (and often should) use build tools:

```
# Production build with Vite (optional)
vite build

# Result: Minified, tree-shaken, optimized bundle
```

But here’s the key difference: **builds are optional, not required**.

- Developing locally? No build needed.
- Prototyping? No build needed.
- Small sites? Deploy directly, no build needed.
- Large applications? Add a build for optimization.

The build is an optimization, not a requirement.

2.6 Progressive Enhancement and Graceful Degradation

2.6.1 Layering Functionality

LARC embraces the web’s fundamental principle: **progressive enhancement**. Start with a basic HTML structure that works, then enhance it with JavaScript.

Level 0: Static HTML

```
<article class="card">
  <h2>Product Name</h2>
  <p>Description of the product</p>
  <a href="/product/123">View Details</a>
</article>
```

Works everywhere. No JavaScript required. Search engines can index it. Screen readers can navigate it.

Level 1: Basic Web Component

```
<script type="module">
  class ProductCard extends HTMLElement {
    connectedCallback() {
      const product = JSON.parse(this.getAttribute('data-product'));
      this.innerHTML = `
```

```

        <article class="card">
          <h2>${product.name}</h2>
          <p>${product.description}</p>
          <a href="/product/${product.id}">View Details</a>
        </article>
      `;
    }
  }
  customElements.define('product-card', ProductCard);
</script>

<product-card data-product='{ "name": "Product", "description": "Description", "id": "123" }'></product-card>

```

Enhanced with JavaScript, but the content is still in the DOM. Still indexable. Still accessible.

Level 2: PAN Integration

```

class ProductCard extends HTMLElement {
  connectedCallback() {
    // Subscribe to product updates
    this.subscription = panClient.subscribe(
      `product.${this.productId}.*`,
      ({ data }) => this.update(data)
    );

    // Request current data
    panClient.publish(`product.${this.productId}.fetch`, null);
  }

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

  update(data) {
    this.render(data);
  }
}

```

Now the component participates in the application's message flow. It reacts to updates, coordinates with other components, and manages state.

Level 3: Advanced Features

```

class ProductCard extends HTMLElement {
  connectedCallback() {
    // Theme support
    panClient.subscribe('theme.change', ({ data }) => {
      this.applyTheme(data);
    });
  }
}

```

```

// Internationalization
panClient.subscribe('locale.change', ({ data }) => {
  this.updateLocale(data);
});

// Real-time updates
panClient.subscribe(`product.${this.productId}.update`, ({ data }) => {
  this.animateUpdate(data);
});

// Analytics
this.addEventListener('click', () => {
  panClient.publish('analytics.event', {
    type: 'product-click',
    id: this.productId
  });
});
}
}

```

Each layer adds functionality without breaking previous layers. If JavaScript fails to load, the HTML still works. If the PAN bus isn't available, the component still renders.

2.6.2 Graceful Degradation

Progressive enhancement works in reverse too. As capabilities decrease, functionality gracefully degrades:

```

class EnhancedComponent extends HTMLElement {
  connectedCallback() {
    // Check for Web Components support (already guaranteed if this runs)

    // Check for ES Modules support
    if ('noModule' in HTMLScriptElement.prototype) {
      this.enableModuleFeatures();
    }

    // Check for Shadow DOM support
    if (this.attachShadow) {
      this.attachShadow({ mode: 'open' });
      this.useShadowDOM = true;
    } else {
      // Fallback: Light DOM with scoped styles
      this.useShadowDOM = false;
    }

    // Check for BroadcastChannel (for cross-tab sync)
    if ('BroadcastChannel' in window) {

```

```

    this.enableCrossTabSync();
  }

  // Check for IndexedDB (for persistence)
  if ('indexedDB' in window) {
    this.enablePersistence();
  } else {
    // Fallback: localStorage
    this.enableBasicStorage();
  }

  this.render();
}
}

```

The component adapts to available APIs. It doesn't require cutting-edge features—it enhances the experience when they're available.

2.6.3 Browser Support Strategy

LARC targets modern browsers (Chrome 90+, Firefox 88+, Safari 14+, Edge 90+) that support:

- Custom Elements v1
- Shadow DOM v1
- ES Modules
- ES2020 features

This covers 95%+ of global users. For the remaining 5%, you have options:

Option 1: Polyfills

Load polyfills for older browsers:

```

<script src="https://unpkg.com/@webcomponents/webcomponentsjs@2/webcomponents-loader.js"></script>
<script type="module" src="/src/app.mjs"></script>

```

Option 2: Build Step

For maximum compatibility, add a production build that transpiles to ES5:

```
vite build --target es2015
```

Option 3: Server-Side Rendering

Render content on the server for browsers without JavaScript:

```

// Node.js
import { renderToString } from '@larcjs/ssr';
const html = renderToString('<pan-card>Hello</pan-card>');

```

But here's the key: these are **optimizations**, not requirements. Start with modern browsers and add compatibility only if needed.

2.7 Comparison to Other Approaches

2.7.1 LARC vs. Redux

Redux:

```
// Redux requires significant boilerplate

// 1. Define action types
const ADD_ITEM = 'ADD_ITEM';
const REMOVE_ITEM = 'REMOVE_ITEM';
const UPDATE_ITEM = 'UPDATE_ITEM';

// 2. Create action creators
const addItem = (item) => ({ type: ADD_ITEM, payload: item });
const removeItem = (id) => ({ type: REMOVE_ITEM, payload: id });
const updateItem = (id, data) => ({ type: UPDATE_ITEM, payload: { id, data } });

// 3. Write reducer
function itemsReducer(state = [], action) {
  switch (action.type) {
    case ADD_ITEM:
      return [...state, action.payload];
    case REMOVE_ITEM:
      return state.filter(item => item.id !== action.payload);
    case UPDATE_ITEM:
      return state.map(item =>
        item.id === action.payload.id
          ? { ...item, ...action.payload.data }
          : item
      );
    default:
      return state;
  }
}

// 4. Create store
const store = createStore(combineReducers({ items: itemsReducer }));

// 5. Connect components
const mapStateToProps = (state) => ({ items: state.items });
const mapDispatchToProps = { addItem, removeItem, updateItem };
export default connect(mapStateToProps, mapDispatchToProps)(ItemList);
```

LARC:

```
// LARC uses message-passing - no boilerplate

// Publish messages
```

```

panClient.publish('items.add', item, { retain: true });
panClient.publish('items.remove', id);
panClient.publish('items.update', { id, data });

// Subscribe in components
panClient.subscribe('items.*', ({ topic, data }) => {
  if (topic.endsWith('.add')) {
    items.push(data);
  } else if (topic.endsWith('.remove')) {
    items = items.filter(item => item.id !== data);
  } else if (topic.endsWith('.update')) {
    items = items.map(item =>
      item.id === data.id ? { ...item, ...data.data } : item
    );
  }
  render();
});

```

Key Differences:

Redux	LARC
Centralized store, reducers, actions, selectors	Distributed messages, topics, subscribers
All state in one place	State distributed across components
Components coupled to store	Components coupled only to topics
Requires setup and configuration	Just publish and subscribe

When Redux is better:

- Need single source of truth
- Complex state transformations
- Time-travel debugging is essential
- Team is already trained in Redux

When LARC is better:

- Loosely coupled components
- Progressive enhancement
- Mix of frameworks
- Simple message-driven logic

2.7.2 Other State Management Approaches

The same pattern applies to other state management solutions. Vuex (Vue), MobX, and React Context all share similar characteristics: centralized state, framework-specific APIs, and various degrees of boilerplate. LARC's message-passing approach offers a framework-agnostic alternative with explicit data flow.

2.7.3 Summary: Architectural Trade-offs

Feature	Redux	Vuex	MobX	Context	LARC
Learning curve	Steep	Medium	Medium	Low	Low
Boilerplate	High	Medium	Low	Low	Minimal
Framework lock-in	No*	Yes	No*	Yes	No
Testability	Excellent	Good	Good	Fair	Excellent
DevTools	Excellent	Excellent	Good	Limited	Good
Bundle size	~8KB	~3KB	~16KB	0KB	~5KB
Async handling	Middleware	Built-in	Built-in	Manual	Manual
Type safety	Good	Fair	Excellent	Good	Optional
Cross-framework	Possible	No	Possible	No	Yes
Component coupling	Store	Store	Store	Context	Topics

*Requires framework integration libraries

LARC's sweet spot:

- You need loose coupling between components
- You're mixing frameworks or using vanilla JS
- You want simple, explicit data flow
- You value progressive enhancement
- You want minimal dependencies

LARC's limitations:

- No automatic reactivity (explicit subscriptions required)
- No built-in transaction/rollback
- No built-in middleware system (yet)
- No official browser DevTools extension (yet)

2.8 Additional Practical Benefits

Beyond the architectural advantages, LARC offers practical benefits for everyday development:

2.8.1 Simplified Component Testing

Traditional approach:

Testing a Redux-connected component requires:

```
import { Provider } from 'react-redux';
import configureMockStore from 'redux-mock-store';
import { render } from '@testing-library/react';

const mockStore = configureMockStore();
const store = mockStore({
  items: [],
  user: { name: 'Test' },
```

```

    theme: 'light'
  });

test('renders item list', () => {
  render(
    <Provider store={store}>
      <ItemList />
    </Provider>
  );
  // Test assertions...
});

```

You have to:

1. Mock the entire store
2. Provide store structure
3. Wrap component in Provider
4. Understand Redux internals

LARC approach:

```

import { mockPanClient } from '@larcjs/testing';

test('renders item list', () => {
  const client = mockPanClient();
  const itemList = document.createElement('item-list');
  document.body.appendChild(itemList);

  // Publish test data
  client.publish('items.list', [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' }
  ], { retain: true });

  // Test assertions on DOM...
});

```

Simpler. Mock the message bus, not the entire application state.

2.8.2 Multi-Tab Synchronization

Traditional approach:

Synchronizing state across browser tabs is painful:

```

// Tab 1: Update state
localStorage.setItem('user', JSON.stringify(user));

// Tab 2: Poll for changes
setInterval(() => {
  const stored = localStorage.getItem('user');

```

```

const user = JSON.parse(stored);
updateState(user);
}, 1000);

// Or use storage events (but they're clunky)
window.addEventListener('storage', (e) => {
  if (e.key === 'user') {
    updateState(JSON.parse(e.newValue));
  }
});

```

LARC approach:

LARC can use BroadcastChannel for cross-tab communication:

```

// Tab 1: Publish message
panClient.publish('user.update', user, { retain: true });

// Tab 2: Automatically receives message
panClient.subscribe('user.update', ({ data }) => {
  updateState(data);
});

// No polling. No storage events. Just messages.

```

Tabs stay synchronized automatically through the PAN bus.

2.9 Conclusion: A Pragmatic Philosophy

LARC's philosophy can be summarized in a few principles:

1. **Build on standards, not abstractions** — Use what browsers provide natively
2. **Message-passing over shared state** — Loose coupling through pub/sub
3. **Zero-build development** — Edit code and see results immediately
4. **Progressive enhancement** — Start simple, add complexity only when needed
5. **Framework-agnostic** — Components work everywhere
6. **Explicit over implicit** — Data flow should be obvious

This philosophy makes trade-offs:

What you gain: - Simplicity - Portability - Debuggability - Fast iteration - Low barrier to entry

What you give up:

- Automatic reactivity
- Framework-specific optimizations
- Established ecosystem
- Corporate backing

Is LARC right for your project? Consider these questions:

- Do you value simplicity over framework features?

- Do you need components that work across frameworks?
- Do you want fast iteration during development?
- Are you comfortable with explicit data flow?
- Do you prefer standards over abstractions?

If you answered “yes” to most of these, LARC might be a good fit.

In the next chapter, we’ll explore the story of LARC—how it came to be, the design decisions along the way, and the real-world use cases that shaped its development.

Chapter 3

The LARC Story

Or: How We Learned to Stop Worrying and Love Asynchronous Components

Every technology has an origin story. Some begin in garages, others in corporate research labs. LARC’s story begins with a simple observation: web development shouldn’t be this hard.

3.1 The Problem That Wouldn’t Go Away

Picture this: You’re building a web application in 2020. You need to fetch some data from an API, display it in a component, and maybe update it when the user clicks a button. Simple, right? Yet you find yourself drowning in boilerplate—state management libraries, effect hooks, loading states, error boundaries, and an ever-growing `node_modules` directory that could collapse into a black hole at any moment.

The React team gave us hooks. Vue gave us the composition API. Svelte gave us reactive declarations. Each solution was elegant in its own way, but they all danced around a fundamental truth: **components are inherently asynchronous**, yet we kept treating them as synchronous with bolted-on async features.

Think about it. A component might need to:

- Fetch data from an API
- Wait for user input
- Subscribe to real-time updates
- Coordinate with other components
- Handle errors and retries

Every single one of these is an asynchronous operation. Yet our component models were built on synchronous rendering with async tacked on as an afterthought. We were trying to fit a round peg into a square hole, and then wondering why we needed so much glue.

3.2 Enter LARC

LARC (Live Asynchronous Reactive Components) emerged from a deceptively simple question: *What if we built components async-first from the ground up?*

Not “async as a feature you can add.” Not “async as a pattern you can implement.” But async as the **fundamental paradigm**—the water the fish swims in, so natural it becomes invisible.

The core insight was this: if components are inherently asynchronous, let’s make them JavaScript Promises. Not wrapped in promises. Not returning promises. Actually **be** promises. After all, a promise is just a value that exists somewhere in time. A component is a UI element that exists somewhere in time. The parallel was too elegant to ignore.

```
// A LARC component is just an async function
async function UserProfile({ userId }) {
  const user = await fetch(`/api/users/${userId}`);
  return html`
    <div class="profile">
      <h2>${user.name}</h2>
      <p>${user.bio}</p>
    </div>
  `;
}
```

Look at that. No `useEffect`. No `useState`. No lifecycle methods. No loading states. The code reads exactly like what it does: fetch the data, then render it. The asynchrony is right there in the language, not hidden behind framework abstractions.

3.3 Design Decisions and Trade-offs

3.3.1 The Great Hydration Debate

Early in LARC’s development, we faced a critical decision: server-side rendering. The React world had spent years perfecting hydration—that delicate dance where server-rendered HTML comes alive on the client. Should LARC follow suit?

We chose a different path: **streaming server rendering** with progressive enhancement. Instead of sending static HTML that gets “rehydrated” into a full client-side app, LARC streams components as they resolve. A slow database query doesn’t block the entire page—it just means that component arrives a bit later.

This decision had consequences. You can’t “hydrate” a LARC app in the traditional sense. But you gain something more valuable: **true progressive rendering**. Your page loads fast because it’s actually fast, not because you’ve carefully orchestrated a theatrical performance of looking fast while secretly downloading megabytes of JavaScript.

Some people called this controversial. We called it honest.

3.3.2 Reactivity Without the Reactivity Tax

The next question: how do components update? Every framework has its answer:

- React: Immutable state and reconciliation
- Vue: Proxies and dependency tracking
- Svelte: Compile-time reactive statements
- Angular: Zone.js and change detection

LARC took yet another path: **explicit subscriptions**. If you want a component to update, subscribe to a signal, observable, or any async iterable. When the source emits, the component re-renders.

```
async function LiveCounter({ signal }) {
  for await (const count of signal) {
    return html`<div>Count: ${count}</div>`;
  }
}
```

This isn't the most magical solution. You can't just mutate a variable and expect the UI to update. But it's **explicit**, **predictable**, and has zero hidden costs. No virtual DOM diffing. No proxy overhead. No compiler magic. Just async iteration—a standard JavaScript feature since ES2018.

The trade-off? You have to think about your data flow. LARC won't guess what you meant. But in exchange, you get complete control and no surprising performance cliffs.

3.3.3 HTML Templates: Tagged or Literal?

Here's where we made our most controversial decision. JSX had won the mindshare wars. Even Vue 3 added JSX support. Surely LARC would use JSX, right?

Nope. We went with **tagged template literals**.

```
// LARC style
html`<div class="${className}">${content}</div>`

// Not JSX
<div className={className}>{content}</div>
```

Why? Three reasons:

1. **No build step required.** You can write LARC in a `<script type="module">` tag and it just works. Try that with JSX.
2. **It's actual HTML.** Copy-paste from your designer's mockup works. No translating `class` to `className` or `for` to `htmlFor`. HTML is HTML.
3. **Syntax highlighting is free.** Every editor that understands HTML already highlights these correctly. No plugins, no extensions, no configuration.

The downside? You lose some type safety and your linter can't validate your HTML structure. We considered this an acceptable trade-off for the ergonomic wins. LARC is for people who like writing HTML, not for people who tolerate it.

3.4 The PAN: A Network Protocol for Components

Here's where LARC gets weird (in a good way).

Traditional component communication follows familiar patterns:

- Props down, events up
- Global state management

- Context providers
- Dependency injection

These work, but they’re all based on a tree structure—parents, children, ancestors, descendants. Yet real applications aren’t trees. They’re **graphs**. A notification component needs to talk to the API client. The shopping cart needs to talk to the inventory system. The analytics tracker needs to know about everything.

We needed something different. Something that felt less like a hierarchy and more like... a network.

Enter the **PAN (Page Area Network)**.

3.4.1 The Origin of PAN

The name came from a brainstorming session that was getting nowhere. We’d considered “component mesh,” “reactive bus,” “signal network,” and other boring enterprise-y names. Then someone joked: “It’s like a personal area network, but for a page.”

PAN. It stuck immediately. It was short, memorable, and just slightly cheeky. Plus, the network metaphor was perfect. Components aren’t calling methods on each other—they’re **broadcasting** on channels and **listening** for messages. It’s publish-subscribe, but for UI components.

```
// Broadcasting on the PAN
pan.emit('user:login', { userId: 123, username: 'alice' });

// Listening on the PAN
async function WelcomeMessage() {
  for await (const { username } of pan.on('user:login')) {
    return html`<div>Welcome back, ${username}!</div>`;
  }
}
```

The PAN isn’t revolutionary technology. It’s event emitters and observables, patterns that date back decades. But **giving it a name** and **making it first-class** changed how people thought about component communication. You’re not fighting against the framework’s hierarchy—you’re using a purpose-built communication layer.

3.4.2 PAN Design Principles

The PAN follows a few key principles:

1. **Namespaced channels.** Events live in namespaces like `user:login` or `cart:add`. This prevents collisions and makes systems self-documenting.
2. **No required coordination.** Components can emit events that no one listens to. They can listen to events that never fire. The PAN doesn’t care. This makes components truly independent.
3. **Time-travel friendly.** Every event is timestamped and logged (in dev mode). You can replay sequences, debug race conditions, and understand causality. Because debugging async systems is hard enough without flying blind.

4. **Automatic instantiation.** You don't create a PAN—it's just there, automatically, like `console` or `window`. One less thing to configure, one less thing to inject, one less thing to get wrong.

The trade-off? The PAN is implicit global state. Some people hate this. They've been trained that global state is the devil, that everything should be explicitly passed through constructors and function parameters. They're not wrong—for most code.

But UI components are special. They're already global-ish—they exist in a single shared page. Making them pretend to be isolated, pure functions is ceremony without benefit. The PAN embraces this reality.

3.5 Evolution and Growing Pains

LARC didn't emerge fully formed. Version 0.1 was... let's call it "enthusiastic." It had ideas. It had ambition. It also had bugs, missing features, and APIs that made sense at 2am but not at 2pm.

3.5.1 The Streaming Crisis

Early versions of LARC tried to stream everything, all the time. Every component was a stream. Every update was a new stream message. This was theoretically beautiful and practically unusable.

The problem: **backpressure**. If a component updated faster than the browser could render, messages would queue up, memory would bloat, and eventually your tab would crash. We learned this the hard way when someone used LARC to display real-time stock prices. Thousands of updates per second met an unmovable object: the browser's rendering pipeline.

The fix required a philosophical shift. Not everything needs to be a stream. Sometimes, you just want the **latest value**. This led to the distinction between **signals** (which buffer the latest value) and **streams** (which preserve the full sequence). Most components want signals. Streams are for the rare cases where order and completeness matter.

3.5.2 The TypeScript Years

In 2021, we had a reckoning with TypeScript. LARC was written in vanilla JavaScript. The docs showed vanilla JavaScript. The examples used vanilla JavaScript. But increasingly, users were asking: "Where are the types?"

We resisted at first. LARC was supposed to be simple, lightweight, no-build-required. TypeScript felt like adding weight. But we were wrong. TypeScript wasn't about compiler strictness—it was about **developer experience**. Autocomplete, inline documentation, catching bugs before runtime. These weren't nice-to-haves; they were essential for anything beyond toy examples.

So we rewrote LARC in TypeScript. Not because we loved types (though they grew on us), but because our users did. The library stayed tiny—types are free at runtime—but the DX improved dramatically.

The lesson: **Listen to your users, even when they're asking you to compromise your aesthetic vision.** Especially then.

3.5.3 The Build-Tool Wars

LARC’s “no build step” philosophy hit reality hard when we tried to integrate with existing applications. Yes, you could write LARC in a `<script>` tag. But most real projects use Webpack, Vite, Rollup, or whatever the JavaScript ecosystem has decreed is cool this month.

We couldn’t fight the build tools—we had to join them. This meant:

- Writing plugins for every major bundler
- Supporting JSX (yes, really) as an alternative to tagged templates
- Providing pre-built bundles for CDNs
- Creating a CLI for scaffolding projects

Each addition felt like a betrayal of the original vision. But each also made LARC more usable in the real world. Purity is beautiful in theory. In practice, people need to ship code on Tuesday.

3.6 Real-World Use Cases

LARC found its audience in unexpected places.

3.6.1 Live Data Dashboards

Financial firms discovered LARC early. When you’re displaying real-time market data across hundreds of components, traditional frameworks struggle. Too much reconciliation overhead, too many re-renders, too much wasted work.

LARC’s streaming model fit perfectly. Each widget was an independent component subscribed to its own data feed. Updates flowed through the PAN. No global state to synchronize, no render batching to tune, no performance cliffs to hit. It just worked.

One trading desk reported replacing 10,000 lines of React + Redux with 2,000 lines of LARC. The new version was faster, more maintainable, and actually understandable by the junior developers.

3.6.2 Progressive Enhancement Sites

Ironically, LARC also found success in the opposite domain: simple content sites that wanted a touch of interactivity. A blog with a newsletter signup form. A marketing site with a live demo. An e-commerce store with real-time inventory.

These sites didn’t need massive client-side frameworks. They needed a sprinkle of JavaScript that enhanced the HTML without taking it over. LARC’s async components could render on the server, stream to the client, and add interactivity only where needed.

The “no build step” feature stopped being a compromise and became a selling point. Designers could edit HTML files directly. Developers could add components without configuring Webpack. It was web development like it used to be—just with better asynchrony.

3.6.3 IoT Control Panels

The Internet of Things people found LARC too. When you’re building a web interface for smart home devices, you’re dealing with:

- Unreliable networks

- Devices that appear and disappear
- Real-time state updates
- Lots of independent components

The PAN model mapped naturally to MQTT topics and WebSocket events. Each device became a channel. Each UI component subscribed to the devices it cared about. The system self-organized without centralized coordination.

One smart home startup built their entire control panel in LARC—500 devices, 2,000 data points, 60 FPS updates. It ran smoothly on a Raspberry Pi. Try that with a typical SPA framework.

3.7 Community and Ecosystem

LARC’s community didn’t grow explosively—it grew steadily. We never hit the front page of Hacker News for a week straight. We never became a meme on Twitter. But developers who tried LARC tended to stick around.

3.7.1 The Component Library

Early on, someone started a “LARC Components” repository. Basic stuff—buttons, forms, modals. It wasn’t fancy, but it was practical. More people contributed. Soon there were data tables, charts, calendars, and all the widgets you’d expect.

The library followed LARC’s philosophy: components were independent, async-first, and communicated via the PAN. A modal component didn’t need a “modal manager”—it just listened for `modal:open` events. A notification system didn’t need to be wired into every component—it subscribed to `notify:*` events.

The result felt different from other component libraries. Less configuration, more convention. Less wiring, more broadcasting. It wasn’t for everyone, but for those who got it, it was liberating.

3.7.2 The Plugin Ecosystem

Developers started writing PAN plugins. Some were simple utilities:

- `pan-persist`: Save PAN events to `localStorage`
- `pan-time-travel`: Replay event sequences for debugging
- `pan-analytics`: Track user interactions automatically

Others were full integrations:

- `pan-firebase`: Bridge Firebase Realtime Database to PAN events
- `pan-graphql`: Subscribe to GraphQL subscriptions via PAN
- `pan-webrtc`: Coordinate WebRTC connections through PAN

The plugin pattern emerged organically. Since the PAN was just an event emitter, plugins were just functions that added new behaviors. No framework API to learn, no plugin architecture to understand. Just JavaScript.

3.7.3 The Documentation Journey

LARC’s docs went through several iterations. The first version was technically accurate and completely opaque. We’d made the classic mistake: writing docs for people who already understood the system.

The rewrite focused on **mental models**. Instead of “here’s how to use this API,” we explained “here’s how to think about async components.” Instead of exhaustive API references, we provided guided examples that built intuition.

This book you’re reading now is the culmination of that journey. Not just a reference manual, but a guide to thinking in LARC.

3.8 Lessons Learned

Building LARC taught us things that no amount of theorizing could:

1. **Async-first is a different paradigm.** You can’t just add async to a sync model. You have to rebuild from the ground up. This is hard, but worth it.
2. **Trade-offs are real.** Every framework makes trade-offs. React trades simplicity for ecosystem. Svelte trades runtime flexibility for compile-time optimization. LARC trades magic for explicitness. Own your trade-offs.
3. **Developer experience matters more than you think.** The best API in the world is useless if developers hate using it. TypeScript support, good error messages, and clear documentation aren’t optional.
4. **Weird names stick.** “PAN” was a joke that became a feature. Sometimes the silly idea is the right idea.
5. **Community isn’t about numbers.** A small, engaged community beats a large, passive one every time. We’d rather have 1,000 developers who truly understand LARC than 100,000 who cargo-cult it.

3.9 The Future

Where does LARC go from here? We have ideas:

- Better streaming server rendering, possibly with resumability
- First-class support for edge computing platforms
- Enhanced time-travel debugging and replay tools
- Integration with emerging web standards like View Transitions
- Maybe, just maybe, a visual component builder (but probably not)

But the core philosophy remains: **components are asynchronous, and frameworks should embrace that reality rather than hide from it.**

LARC isn’t trying to replace React. It isn’t trying to kill Vue. It’s offering a different perspective—a path for developers who looked at the existing options and thought, “there has to be another way.”

Sometimes the best tool isn’t the most popular one. It’s the one that fits how you think, how you work, and how you want to build software.

If that sounds like LARC, welcome to the journey. If not, that’s okay too. The web is big enough for many approaches.

3.10 Epilogue: The Name

We should probably explain what LARC actually stands for. You’ve seen “Live Asynchronous Reactive Components” throughout this chapter. That’s the official expansion.

But here’s a secret: the name came first. Someone said “LARC” and we all liked how it sounded—like “spark” but with an L. The acronym came later, reverse-engineered to fit.

Some of the rejected expansions:

- “Lightweight Async Rendering Components”
- “Lazy Async Reactive Components”
- “Live Applications with Reactive Components”
- “Larry’s Awesome Reactive Components” (no one on the team was named Larry)

We eventually settled on Live Asynchronous Reactive Components because it captured the essence: components that live in time, embrace asynchrony, and react to change.

But really, LARC is just LARC. Sometimes a name is just a name.

And sometimes it’s the start of something bigger.

Next: Chapter 4 - Core Concepts, where we dive deep into the building blocks that make LARC tick.

Chapter 4

Core Concepts

4.1 Introduction: The Building Blocks

If you're coming from a traditional framework background, LARC might seem... sparse. There's no virtual DOM, no reconciliation algorithm, no elaborate lifecycle methods. What you get instead is something arguably more powerful: a set of composable primitives that work together through a simple, consistent interface.

This chapter covers the core concepts that make LARC tick. If you understand these fundamentals, you'll understand 90% of what you need to build production applications. The remaining 10% is just knowing which components already exist so you don't reinvent the wheel.

Let's start at the heart of it all: the message bus.

4.2 The Message Bus: Your Application's Nervous System

4.2.1 What Is a Message Bus?

Think of the message bus as your application's nervous system. Just as your nervous system carries signals between different parts of your body without those parts needing to know about each other directly, the message bus carries messages between components without creating coupling between them.

Here's the elegant part: the entire bus is just a custom element sitting in your DOM:

```
<!DOCTYPE html>
<html>
<head>
  <title>My App</title>
</head>
<body>
  <pan-bus></pan-bus>

  <!-- Your app goes here -->
  <my-dashboard></my-dashboard>
```

```
</body>
</html>
```

That’s it. No configuration files, no initialization boilerplate, no plugin registration. The `<pan-bus>` element listens for specific DOM events and routes them to interested parties. It’s just HTML doing HTML things.

4.2.2 How Does It Work?

The bus operates using the browser’s built-in event system. Components communicate by dispatching `CustomEvents` that bubble up through the DOM. The bus catches these events, processes them according to its routing rules, and dispatches delivery events to subscribers.

Here’s the beautiful part: because it’s all DOM events, it works across shadow DOM boundaries, through iframes (with appropriate setup), and with any framework that can dispatch events—which is to say, all of them.

Let’s look at a concrete example:

```
// Component A publishes a message
document.dispatchEvent(new CustomEvent('pan:publish', {
  detail: {
    topic: 'user.logged-in',
    data: { userId: '123', name: 'Alice' }
  },
  bubbles: true,
  composed: true
}));

// Component B subscribes and receives it
document.addEventListener('pan:deliver', (e) => {
  if (e.detail.topic === 'user.logged-in') {
    console.log('User logged in:', e.detail.data.name);
  }
});
```

But typing out `CustomEvent` constructors gets tedious fast. That’s why LARC provides the `PanClient` helper:

```
import { PanClient } from '@larc-app/core';

const client = new PanClient();
await client.ready();

// Publishing is now simple
client.publish({
  topic: 'user.logged-in',
  data: { userId: '123', name: 'Alice' }
});
```

```
// So is subscribing
client.subscribe('user.logged-in', (msg) => {
  console.log('User logged in:', msg.data.name);
});
```

Much better. But we can do more.

4.2.3 Configuration and Capabilities

The `<pan-bus>` element accepts configuration through attributes:

```
<pan-bus
  max-retained="1000"
  max-message-size="1048576"
  debug="true"
  allow-global-wildcard="false">
</pan-bus>
```

These settings control memory usage, security policies, and debugging output. In production, you'll want to tune these based on your app's needs. During development, `debug="true"` is invaluable for understanding message flow.

The bus also tracks statistics:

```
// Request stats
const response = await client.request('pan:sys.stats', {});
console.log(response.data);
// {
//   published: 1234,
//   delivered: 5678,
//   dropped: 0,
//   retained: 42,
//   subscriptions: 18,
//   clients: 5
// }
```

These metrics help you understand your application's communication patterns and spot potential performance issues before they become problems.

4.3 Pub/Sub Pattern: Fire and Forget (But Don't Actually Forget)

4.3.1 The Classic Pattern

Publish/subscribe (pub/sub) is the bread and butter of message-based architectures. A component publishes a message about something that happened. Other components subscribe to messages they care about. Neither knows the other exists.

Here's a real-world example from an e-commerce app:

```
// Shopping cart component
class ShoppingCart extends HTMLElement {
```

```

connectedCallback() {
  this.client = new PanClient(this);
  this.render();
}

async addItem(product) {
  this.items.push(product);

  // Tell the world what happened
  this.client.publish({
    topic: 'cart.item-added',
    data: {
      productId: product.id,
      name: product.name,
      price: product.price,
      quantity: 1
    }
  });

  this.render();
}
}

```

Now, anywhere in your application, components can react to items being added to the cart:

```

// Notification badge component
class CartBadge extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);
    this.count = 0;
    this.render();

    // Listen for cart changes
    this.client.subscribe('cart.item-added', () => {
      this.count++;
      this.render();
    });

    this.client.subscribe('cart.item-removed', () => {
      this.count--;
      this.render();
    });
  }
}

// Analytics component
class AnalyticsTracker extends HTMLElement {
  connectedCallback() {

```

```

    this.client = new PanClient(this);

    this.client.subscribe('cart.*', (msg) => {
      // Send to analytics service
      this.trackEvent(msg.topic, msg.data);
    });
  }

  trackEvent(action, data) {
    // Send to your analytics provider
    console.log('Analytics:', action, data);
  }
}

```

Notice how neither the cart badge nor the analytics tracker needed to be registered anywhere or injected with dependencies. They just listen for messages they care about. Add them to the DOM, and they work. Remove them, and they stop working. No cleanup code needed (the bus automatically removes dead subscriptions).

4.3.2 Wildcards: Subscribe to Patterns

One of the most powerful features of LARC's pub/sub system is pattern matching. Instead of subscribing to individual topics, you can subscribe to patterns:

```

// Subscribe to all cart-related messages
client.subscribe('cart.*', (msg) => {
  console.log('Cart event:', msg.topic, msg.data);
});

// Subscribe to all user-related messages
client.subscribe('users.*', (msg) => {
  console.log('User event:', msg.topic, msg.data);
});

// Subscribe to everything (use sparingly!)
client.subscribe('*', (msg) => {
  console.log('Any event:', msg.topic, msg.data);
});

```

The wildcard `*` matches any segment of a topic. So `cart.*` matches `cart.item-added` and `cart.checkout-started`, but not `cart.items.updated` (which has multiple segments after `cart`).

This makes it trivial to build components that react to entire categories of events without knowing the specific topics ahead of time.

4.3.3 The Global Wildcard Problem

You might be wondering: “What about security? Can any component spy on all messages?”

Yes, by default. That's actually intentional for most applications—it makes debugging and monitoring

much easier. But for sensitive applications, you can disable the global wildcard:

```
<pan-bus allow-global-wildcard="false"></pan-bus>
```

Now attempts to subscribe to `*` will be rejected. Components can still use specific wildcards like `users.*`, just not the nuclear option.

4.4 Topics and Routing: Addressing Your Messages

4.4.1 Naming Conventions

Topics in LARC follow a hierarchical naming convention similar to DNS or Java packages. The convention is:

`entity.resource.action`

For example:

- `users.list.state` - The current state of the user list
- `users.item.save` - Request to save a user item
- `cart.checkout.started` - Notification that checkout has started
- `api.users.error` - Error from the users API

This hierarchy serves two purposes:

1. **Organization:** It groups related topics together
2. **Routing:** It enables wildcard subscriptions and routing rules

Here are some real-world examples:

```
// State management topics
'users.list.state'      // Current list of users
'users.filter.state'    // Current filter settings
'users.pagination.state' // Current page/offset

// Action topics
'users.item.save'       // Save a user
'users.item.delete'     // Delete a user
'users.list.refresh'    // Refresh the list

// Event topics
'users.item.saved'      // User was saved
'users.item.deleted'    // User was deleted
'users.list.changed'    // List has changed

// API topics
'api.users.request'     // API request initiated
'api.users.success'     // API request succeeded
'api.users.error'       // API request failed
```

4.4.2 Semantic Routing

The beauty of hierarchical topics is that you can build semantic routing rules. For example, you might want to:

1. Log all API errors to your monitoring service
2. Cache all `*.state` messages for new components
3. Persist all `*.settings` changes to `localStorage`
4. Throttle high-frequency UI events

LARC's routing system (enabled with `enable-routing="true"`) lets you configure these behaviors declaratively. But even without routing, the topic structure helps you reason about message flow.

4.4.3 Anti-Patterns to Avoid

Some topic naming patterns to avoid:

Too Generic:

```
// Bad: What user? What data?  
'update'  
'change'  
'event'  
  
// Good: Specific and hierarchical  
'users.item.updated'  
'settings.theme.changed'  
'cart.item-added'
```

Too Specific:

```
// Bad: Can't subscribe to patterns  
'user-123-updated'  
'product-abc-added-to-cart'  
  
// Good: Use data payload for specifics  
'users.item.updated' // data: { userId: '123' }  
'cart.item-added'    // data: { productId: 'abc' }
```

Mixed Concerns:

```
// Bad: Mixing entity types  
'users-and-posts.updated'  
  
// Good: Separate topics  
'users.item.updated'  
'posts.item.updated'
```

4.5 Message Lifecycle: Birth, Death, and Resurrection

4.5.1 The Lifecycle of a Message

When you publish a message, it goes through several stages:

1. **Creation:** You publish the message via `client.publish()`
2. **Validation:** The bus validates message size and serializability
3. **Enrichment:** The bus adds metadata (id, timestamp)
4. **Routing:** The bus applies routing rules (if enabled)
5. **Delivery:** The bus dispatches to all matching subscribers
6. **Retention:** If marked `retain: true`, the message is cached
7. **Cleanup:** After delivery, the message object is eligible for GC

Let's look at each stage in detail.

4.5.2 Message Structure

A complete message has this shape:

```
{
  topic: 'users.item.saved',           // Required: hierarchical topic
  data: { id: '123', name: 'Alice' }, // Required: the payload
  id: 'a1b2c3d4-...',                 // Auto-generated UUID
  ts: 1698765432000,                  // Auto-generated timestamp
  retain: true,                       // Optional: cache this message
  replyTo: 'pan:$reply:...',          // Optional: for request/reply
  correlationId: 'req-123',           // Optional: for correlation
  headers: {                          // Optional: custom metadata
    'x-user-id': '123',
    'x-trace-id': 'abc-def'
  }
}
```

You only provide topic and data. The bus fills in the rest.

4.5.3 Validation and Size Limits

The bus validates messages before processing them:

```
// This will be rejected
client.publish({
  topic: 'users.item.save',
  data: {
    name: 'Alice',
    profilePicture: gigabyteSizedBinaryBlob // Too large!
  }
});
```

Default limits:

- Max message size: 1MB

- Max payload size: 512KB

Why two limits? The message size includes metadata, headers, and the payload. The payload limit is separate because payloads are what users control.

If you need to send large data, don't send it through the bus. Instead, send a reference:

```
// Good: Send a reference
client.publish({
  topic: 'upload.completed',
  data: {
    fileId: 'abc-123',
    url: '/api/files/abc-123',
    size: 10485760, // 10MB
    type: 'image/jpeg'
  }
});
```

4.5.4 Retained Messages: The Last Value Cache

One of the most useful features of the message bus is message retention. When you publish a message with `retain: true`, the bus caches it:

```
// Publish current state
client.publish({
  topic: 'users.list.state',
  data: { users: [...], total: 100 },
  retain: true
});
```

Now when a component subscribes to `users.list.state`, it immediately receives the last published value. This is perfect for state synchronization:

```
// New component gets current state immediately
class UserList extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);

    // Request retained messages
    this.client.subscribe('users.list.state', (msg) => {
      this.users = msg.data.users;
      this.render();
    }, { retained: true }); // <- This is the key
  }
}
```

The component doesn't need to know how to fetch the initial state. It doesn't need to make an API call. It just asks for retained messages and gets the current state instantly.

4.5.5 Memory Management

The bus limits retained messages to prevent memory leaks. By default, it keeps 1000 retained messages using an LRU (Least Recently Used) eviction policy. When the limit is reached, the oldest unused message is evicted.

You can tune this:

```
<pan-bus max-retained="5000"></pan-bus>
```

But be careful. Retained messages live in memory for the lifetime of the page. If you're retaining large objects or high-frequency updates, you can consume significant memory.

A good rule of thumb: only retain state snapshots, not events.

```
// Good: Retain state
client.publish({
  topic: 'users.list.state',
  data: { users: [...] },
  retain: true
});

// Bad: Don't retain events
client.publish({
  topic: 'users.item.clicked', // Ephemeral event
  data: { userId: '123' },
  retain: false // or just omit it
});
```

4.6 Components and Composition: Building Blocks

4.6.1 What Is a Component in LARC?

In LARC, a component is just a Web Component—a custom element that follows the W3C standard. No special base class, no framework-specific lifecycle methods. Just plain JavaScript classes extending `HTMLElement`:

```
class UserCard extends HTMLElement {
  connectedCallback() {
    // Element was added to DOM
    this.client = new PanClient(this);
    this.render();
  }

  disconnectedCallback() {
    // Element was removed from DOM
    // (PanClient automatically cleans up subscriptions)
  }

  render() {
    this.innerHTML = `
```

```

        <div class="user-card">
          <h3>${this.getAttribute('name')}</h3>
          <p>${this.getAttribute('email')}</p>
        </div>
      `;
    }
  }

  customElements.define('user-card', UserCard);

```

Use it like any HTML element:

```
<user-card name="Alice" email="alice@example.com"></user-card>
```

4.6.2 Communication Patterns

Components in LARC communicate through three primary patterns:

1. Attributes (Parent -> Child)

The standard HTML way. Parent sets attributes, child reads them:

```
<user-card user-id="123"></user-card>
```

```

class UserCard extends HTMLElement {
  static get observedAttributes() {
    return ['user-id'];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'user-id') {
      this.loadUser(newValue);
    }
  }
}

```

2. Events (Child -> Parent)

Components dispatch events to notify parents of changes:

```

class UserCard extends HTMLElement {
  handleClick() {
    this.dispatchEvent(new CustomEvent('user-selected', {
      detail: { userId: this.userId },
      bubbles: true
    }));
  }
}

```

```

// Parent listens
document.querySelector('user-card').addEventListener('user-selected', (e) => {

```

```
console.log('User selected:', e.detail.userId);
});
```

3. Messages (Anyone -> Anyone)

For cross-cutting concerns, use the message bus:

```
class UserCard extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);

    // Listen for updates to this user
    this.client.subscribe('users.item.updated', (msg) => {
      if (msg.data.id === this.userId) {
        this.update(msg.data);
      }
    });
  }

  handleSave() {
    // Notify the world
    this.client.publish({
      topic: 'users.item.updated',
      data: { id: this.userId, ...this.getData() }
    });
  }
}
```

4.6.3 Composition Examples

Here's how components compose in practice:

```
<!-- Dashboard composed of smaller components -->
<user-dashboard>
  <header-bar>
    <user-menu></user-menu>
    <notification-badge></notification-badge>
  </header-bar>

  <main-content>
    <user-list>
      <!-- user-card elements will be inserted here -->
    </user-list>

    <user-details>
      <!-- Details shown when user is selected -->
    </user-details>
  </main-content>
</user-dashboard>
```

Each component is independent. The `<user-menu>` publishes `user.logged-out` when the user logs out. The `<user-list>` subscribes to that message and clears itself. No direct coupling needed.

4.6.4 The Autoloader: Zero-Config Imports

One of LARC's killer features is the autoloader. Instead of explicitly importing every component:

```
// Traditional way (tedious!)
import './user-dashboard.js';
import './header-bar.js';
import './user-menu.js';
import './notification-badge.js';
import './main-content.js';
import './user-list.js';
import './user-details.js';
```

Just load the autoloader and use components:

```
<script type="module" src="/core/pan.mjs"></script>

<!-- Components load automatically when used -->
<user-dashboard></user-dashboard>
```

The autoloader uses IntersectionObserver to progressively load components as they approach the viewport. Components not in view aren't loaded until needed, saving bandwidth and parse time.

4.7 State Management Strategies

4.7.1 The Three Flavors of State

State in LARC comes in three flavors:

1. **Local State:** Confined to a single component
2. **Shared State:** Accessed by multiple components
3. **Persistent State:** Survives page reloads

Let's tackle each one.

4.7.2 Local State: Keep It Simple

For state that only matters to one component, use instance variables:

```
class Counter extends HTMLElement {
  constructor() {
    super();
    this.count = 0; // Local state
  }

  increment() {
    this.count++;
    this.render();
  }
}
```

```

}

render() {
  this.innerHTML = `
    <button onclick="this.parentElement.increment()">
      Count: ${this.count}
    </button>
  `;
}
}

```

No store needed. No reducers. Just regular JavaScript variables.

4.7.3 Shared State: Use Retained Messages

When multiple components need the same state, publish it as a retained message:

```

// Producer: Publishes state
class UserListProvider extends HTMLElement {
  async connectedCallback() {
    this.client = new PanClient(this);

    // Load users
    const users = await this.fetchUsers();

    // Publish as retained state
    this.client.publish({
      topic: 'users.list.state',
      data: { users },
      retain: true
    });
  }

  async fetchUsers() {
    const response = await fetch('/api/users');
    return response.json();
  }
}

// Consumer: Subscribes to state
class UserList extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);

    this.client.subscribe('users.list.state', (msg) => {
      this.users = msg.data.users;
      this.render();
    }, { retained: true }); // Get current value immediately
  }
}

```

```

    }
  }

  // Another consumer: Also subscribes
  class UserCount extends HTMLElement {
    connectedCallback() {
      this.client = new PanClient(this);

      this.client.subscribe('users.list.state', (msg) => {
        this.count = msg.data.users.length;
        this.render();
      }, { retained: true });
    }
  }
}

```

All three components are decoupled. The provider doesn't know about the consumers. The consumers don't know about each other. They just agree on a topic name.

4.7.4 The State Publisher Pattern

For complex state, create dedicated state publisher components:

```

class ShoppingCartState extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);
    this.items = [];

    // Listen for state changes
    this.client.subscribe('cart.item.add', (msg) => {
      this.items.push(msg.data);
      this.publishState();
    });

    this.client.subscribe('cart.item.remove', (msg) => {
      this.items = this.items.filter(i => i.id !== msg.data.id);
      this.publishState();
    });

    this.client.subscribe('cart.clear', () => {
      this.items = [];
      this.publishState();
    });

    // Publish initial state
    this.publishState();
  }

  publishState() {

```

```

    this.client.publish({
      topic: 'cart.state',
      data: {
        items: this.items,
        total: this.calculateTotal(),
        count: this.items.length
      },
      retain: true
    });
  }

  calculateTotal() {
    return this.items.reduce((sum, item) => sum + item.price, 0);
  }
}

```

Now any component can:

- Read the cart state by subscribing to `cart.state`
- Modify the cart by publishing to `cart.item.add`, `cart.item.remove`, etc.

The state component acts as a single source of truth, similar to a Redux store, but without the boilerplate.

4.7.5 Persistent State: Add Storage

For state that should survive page reloads, use the `<pan-storage>` component:

```

// Automatically persists to localStorage
class SettingsState extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);

    // Load from storage
    const stored = localStorage.getItem('settings');
    this.settings = stored ? JSON.parse(stored) : this.getDefault();

    this.publishState();

    // Listen for changes
    this.client.subscribe('settings.update', (msg) => {
      this.settings = { ...this.settings, ...msg.data };
      this.save();
      this.publishState();
    });
  }

  save() {
    localStorage.setItem('settings', JSON.stringify(this.settings));
  }
}

```

```
}

publishState() {
  this.client.publish({
    topic: 'settings.state',
    data: this.settings,
    retain: true
  });
}

getDefaults() {
  return {
    theme: 'light',
    language: 'en',
    notifications: true
  };
}
}
```

Or use LARC's built-in `<pan-storage>` component which handles persistence automatically:

```
<pan-storage
  key="settings"
  topic="settings.state"
  storage="localStorage">
</pan-storage>
```

Now any updates to `settings.state` are automatically persisted.

4.8 Event Envelopes and Metadata

4.8.1 The Message Envelope

Every message is wrapped in an envelope that carries metadata:

```
{
  topic: 'users.item.saved',
  data: { id: '123', name: 'Alice' },
  id: 'a1b2c3d4-5678-90ab-cdef-1234567890ab',
  ts: 1698765432000,
  headers: {
    'x-user-id': '123',
    'x-trace-id': 'trace-abc-def'
  }
}
```

4.8.2 Message IDs

Every message gets a unique ID (UUID v4). This enables:

1. **Deduplication:** Ignore messages you've already processed
2. **Tracing:** Track messages through your system
3. **Debugging:** Identify specific messages in logs

```
// Track processed messages
class DeduplicatingSubscriber extends HTMLElement {
  constructor() {
    super();
    this.processed = new Set();
  }

  connectedCallback() {
    this.client = new PanClient(this);

    this.client.subscribe('events.*', (msg) => {
      if (this.processed.has(msg.id)) {
        console.log('Duplicate message, ignoring:', msg.id);
        return;
      }

      this.processed.add(msg.id);
      this.process(msg);
    });
  }
}
```

4.8.3 Timestamps

Messages include a timestamp (milliseconds since epoch). Use it for:

1. **Ordering:** Process messages in chronological order
2. **TTL:** Ignore stale messages
3. **Metrics:** Measure message latency

```
// Ignore stale messages
this.client.subscribe('stock.price.updated', (msg) => {
  const age = Date.now() - msg.ts;

  if (age > 5000) { // More than 5 seconds old
    console.log('Ignoring stale price update');
    return;
  }

  this.updatePrice(msg.data);
});
```

4.8.4 Custom Headers

Add your own metadata with headers:

```
client.publish({
  topic: 'api.request',
  data: { endpoint: '/users' },
  headers: {
    'x-user-id': currentUser.id,
    'x-trace-id': traceId,
    'x-request-id': requestId
  }
});
```

Headers are perfect for:

- Correlation across service boundaries
- User context for multi-tenant systems
- Debugging and tracing
- Custom routing rules

4.8.5 The Request/Reply Pattern

The envelope supports request/reply with `replyTo` and `correlationId`:

```
// Under the hood, client.request() does this:
const correlationId = crypto.randomUUID();
const replyTo = `pan:$reply:${clientId}:${correlationId}`;

// Publish request
client.publish({
  topic: 'users.get',
  data: { id: '123' },
  replyTo,
  correlationId
});

// Subscribe to reply
client.subscribe(replyTo, (msg) => {
  if (msg.correlationId === correlationId) {
    console.log('Got reply:', msg.data);
  }
});
```

But you don't need to do this manually. Just use `client.request()`:

```
const response = await client.request('users.get', { id: '123' });
console.log('User:', response.data);
```

The client handles correlation automatically and returns a Promise that resolves with the reply or rejects on timeout.

4.9 Putting It All Together

Let's build a complete example that demonstrates all these concepts:

```
<!DOCTYPE html>
<html>
<head>
  <title>Task Manager</title>
  <script type="module" src="/core/pan.mjs"></script>
</head>
<body>
  <pan-bus debug="true"></pan-bus>

  <task-app>
    <task-form></task-form>
    <task-list></task-list>
    <task-stats></task-stats>
  </task-app>

  <script type="module">
    import { PanClient } from '/core/pan-client.mjs';

    // Task state manager
    class TaskState extends HTMLElement {
      connectedCallback() {
        this.client = new PanClient(this);
        this.tasks = [];

        // Listen for task operations
        this.client.subscribe('tasks.add', (msg) => {
          this.tasks.push({
            id: crypto.randomUUID(),
            ...msg.data,
            completed: false,
            createdAt: Date.now()
          });
          this.publishState();
        });

        this.client.subscribe('tasks.toggle', (msg) => {
          const task = this.tasks.find(t => t.id === msg.data.id);
          if (task) {
            task.completed = !task.completed;
            this.publishState();
          }
        });

        this.client.subscribe('tasks.delete', (msg) => {
```

```

        this.tasks = this.tasks.filter(t => t.id !== msg.data.id);
        this.publishState();
    });

    // Publish initial state
    this.publishState();
}

publishState() {
    this.client.publish({
        topic: 'tasks.state',
        data: {
            tasks: this.tasks,
            total: this.tasks.length,
            completed: this.tasks.filter(t => t.completed).length,
            pending: this.tasks.filter(t => !t.completed).length
        },
        retain: true
    });
}
}

// Task form
class TaskForm extends HTMLElement {
    connectedCallback() {
        this.client = new PanClient(this);
        this.render();
    }

    render() {
        this.innerHTML = `
            <form>
                <input type="text" id="title" placeholder="Task title" required>
                <button type="submit">Add Task</button>
            </form>
        `;

        this.querySelector('form').addEventListener('submit', (e) => {
            e.preventDefault();
            const title = this.querySelector('#title').value;

            this.client.publish({
                topic: 'tasks.add',
                data: { title }
            });

            this.querySelector('#title').value = '';
        });
    }
}

```

```

    });
  }
}

// Task list
class TaskList extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);

    this.client.subscribe('tasks.state', (msg) => {
      this.tasks = msg.data.tasks;
      this.render();
    }, { retained: true });
  }

  render() {
    this.innerHTML = `
      <ul>
        ${this.tasks || []}.map(task => `
          <li>
            <input
              type="checkbox"
              ${task.completed ? 'checked' : ''}
              onclick="this.closest('task-list').toggle('${task.id}')">
            <span style="${task.completed ? 'text-decoration: line-through' : ''}">
              ${task.title}
            </span>
            <button onclick="this.closest('task-list').delete('${task.id}')">
              Delete
            </button>
          </li>
        `).join('')}
      </ul>
    `;
  }

  toggle(id) {
    this.client.publish({
      topic: 'tasks.toggle',
      data: { id }
    });
  }

  delete(id) {
    this.client.publish({
      topic: 'tasks.delete',
      data: { id }
    });
  }
}

```

```

    });
  }
}

// Task stats
class TaskStats extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);

    this.client.subscribe('tasks.state', (msg) => {
      this.stats = msg.data;
      this.render();
    }, { retained: true });
  }

  render() {
    if (!this.stats) return;

    this.innerHTML = `
      <div>
        Total: ${this.stats.total} |
        Completed: ${this.stats.completed} |
        Pending: ${this.stats.pending}
      </div>
    `;
  }
}

// Register components
customElements.define('task-state', TaskState);
customElements.define('task-form', TaskForm);
customElements.define('task-list', TaskList);
customElements.define('task-stats', TaskStats);

// Create state manager
document.body.appendChild(document.createElement('task-state'));
</script>
</body>
</html>

```

This example demonstrates:

1. **Message Bus:** Coordinates all communication
2. **Pub/Sub:** Components publish and subscribe to topics
3. **Topics:** Hierarchical naming (`tasks.add`, `tasks.state`)
4. **Retained Messages:** State persists for new subscribers
5. **Component Composition:** Independent components work together
6. **State Management:** Centralized state with decoupled consumers

4.10 What We’ve Learned

You now understand the core concepts that make LARC work:

- **Message Bus:** Your application’s event-based nervous system
- **Pub/Sub:** Decoupled communication through topics
- **Topics:** Hierarchical addressing for organization and routing
- **Message Lifecycle:** Creation, validation, delivery, and retention
- **Components:** Standard Web Components using message passing
- **State Management:** Retained messages as distributed state
- **Envelopes:** Metadata for correlation, tracing, and debugging

These primitives combine to create a simple but powerful architecture. No elaborate state management libraries. No dependency injection frameworks. Just components communicating through a message bus.

In the next chapter, we’ll explore the component library and see how LARC provides higher-level abstractions on top of these primitives for common patterns like data binding, forms, and API integration.

4.11 Key Takeaways

1. The `<pan-bus>` element is just HTML—drop it in your page and it works
2. Use `PanClient` for a cleaner API than raw `CustomEvents`
3. Topic hierarchies enable pattern matching and semantic routing
4. Retained messages provide instant state synchronization
5. Components communicate via the bus, not direct references
6. State managers are just components that publish retained state
7. Message envelopes carry metadata for advanced patterns

Now you’re ready to build real applications with LARC. Let’s dive into the component library.

Chapter 5

Getting Started

Welcome to the hands-on portion of *Building with LARC*. This chapter takes you from zero to running code in minutes—no elaborate setup, no dependency nightmares, just a browser, a text editor, and a way to serve static files.

If you’ve worked with modern JavaScript frameworks, you might be bracing yourself for the usual ritual: install Node.js, run `npm install`, wait for 1,400 packages to download, configure webpack, debug build errors, and finally write your first line of application code. LARC skips all of that. You’ll have a working application before you finish reading this chapter.

This isn’t a toy example or a contrived demo. We’ll build something real: a task manager with multiple components, PAN bus messaging, persistent storage, and proper error handling. By the end, you’ll understand LARC’s development workflow and be ready to build your own applications.

5.1 Prerequisites

This chapter assumes you have the prerequisites covered in Chapter 1. The short version: a modern browser, a text editor, and a way to serve static files. That’s it — no Node.js, no npm, no build tools required.

TIP: If you need to set up a local server quickly, `python3 -m http.server 8000` works on most systems with Python installed.

5.2 Installation Options

LARC offers multiple installation paths. Choose the one that matches your workflow and project requirements.

5.2.1 Option 1: CDN (Fastest for Testing)

The quickest way to try LARC is loading it directly from a CDN. This is perfect for experiments, prototypes, and learning.

Create a file called `index.html`:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>LARC Quick Start</title>
</head>
<body>
  <!-- Load LARC from CDN -->
  <script type="module" src="https://unpkg.com/@larcjs/core@2.0.0/src/pan.mjs"></script>

  <!-- PAN bus is automatically instantiated -->

  <!-- Use components declaratively -->
  <pan-card title="Hello LARC">
    <p>This component loaded automatically from the CDN.</p>
    <pan-button>Click Me</pan-button>
  </pan-card>
</body>
</html>

```

Serve the file with any HTTP server:

```

# Python 3
$ python3 -m http.server 8000

# Python 2
$ python -m SimpleHTTPServer 8000

# PHP
$ php -S localhost:8000

# Node.js (if you have http-server installed)
$ npx http-server -p 8000

```

Open <http://localhost:8000> in your browser. You should see a styled card with a button. The component loaded automatically—no imports, no registration, no configuration.

What just happened?

1. The `pan.mjs` script loaded and initialized the PAN autoloader
2. The autoloader scanned your page for custom elements (tags with hyphens)
3. It discovered `<pan-card>` and `<pan-button>`
4. Using `IntersectionObserver`, it loaded these components as they entered the viewport
5. The components registered themselves and rendered their content

This is LARC’s “convention over configuration” approach in action.

When to use CDN: - Learning LARC - Quick experiments and prototypes - Simple single-page applications - Personal projects

When to avoid CDN:

- Production applications requiring optimization
- Offline-first applications
- Projects with strict CSP policies
- Applications needing specific version control

5.2.2 Option 2: NPM Installation

For projects with existing Node.js tooling or teams familiar with npm, install LARC as a package:

```
$ npm install @larcjs/core @larcjs/ui
```

Then import in your JavaScript:

```
// main.js
import '/node_modules/@larcjs/core/pan.mjs';

// Components auto-load from node_modules
// Configure the path resolver if needed
```

Or use an import map in your HTML:

```
<!DOCTYPE html>
<html>
<head>
  <script type="importmap">
  {
    "imports": {
      "@larcjs/core/": "./node_modules/@larcjs/core/",
      "@larcjs/ui/": "./node_modules/@larcjs/ui/"
    }
  }
</script>
  <script type="module" src="./main.js"></script>
</head>
<body>
  <pan-card title="From NPM">
    Components load from node_modules
  </pan-card>
</body>
</html>
```

When to use NPM: - Existing Node.js projects - Teams familiar with package.json workflows - Projects using bundlers (Vite, Webpack, Rollup) - TypeScript projects (add @larcjs/core-types)

5.2.3 Option 3: Git Clone (Full Repository)

For working with examples, contributing to LARC, or learning from source code:

```
# Clone with submodules
$ git clone --recurse-submodules https://github.com/larcjs/larc.git
```

```
$ cd larc

# Run setup script (handles submodule initialization)
$ ./setup.sh          # Mac/Linux
# or
$ setup.bat           # Windows

# Start a server
$ python3 -m http.server 8000

# Open http://localhost:8000/test-config.html
```

This gives you:

- Complete source code
- All examples
- Playground for testing components
- Documentation site
- DevTools extension

When to use Git clone:

- Learning from examples
- Contributing to LARC
- Exploring advanced features
- Building custom components

5.2.4 Option 4: Create LARC App (Coming Soon)

The LARC CLI provides scaffolding tools (currently in development):

```
# Create a new project
$ npx create-larc-app my-app
$ cd my-app
$ npm run dev

# Add components from registry
$ larc add pan-data-table

# Generate custom components
$ larc generate component my-widget
```

This creates a zero-config development environment with hot reload, component generation, and registry integration.

5.3 Development Environment Setup

Once you've chosen an installation method, set up your development environment for maximum productivity.

5.3.1 Directory Structure

LARC doesn't enforce a specific structure, but here's a recommended layout:

```
my-larc-app/
|-- index.html           # Entry point
|-- larc-config.mjs      # Path configuration (optional)
|-- src/
|   |-- components/      # Your custom components
|   |   |-- task-list.mjs
|   |   |-- task-item.mjs
|   |   |-- task-form.mjs
|   |-- utils/           # Helper functions
|   |   |-- api.mjs
|   |   |-- storage.mjs
|   |-- styles/          # Global styles
|   |   |-- main.css
|-- core/                # LARC core (if cloned)
|   |-- src/
|   |   |-- pan.mjs
|-- ui/                  # LARC components (if cloned)
|   |-- src/
|   |   |-- components/
```

Rationale: - `src/components/` - Your application components live here - `src/utils/` - Pure functions, API clients, shared logic - `src/styles/` - Global CSS, CSS custom properties, themes - `core/` and `ui/` - LARC source (only if using Git clone method)

This structure scales from small experiments to large applications while keeping concerns separated.

5.3.2 Editor Configuration

5.3.2.1 VS Code (Recommended)

Install the LARC extension (when available) for:

- Component auto-completion
- PAN topic IntelliSense
- Snippet library
- Integrated component browser

Manual setup without extension:

Create `.vscode/settings.json`:

```
{
  "files.associations": {
    "*.mjs": "javascript"
  },
  "emmet.includeLanguages": {
    "javascript": "html"
  },
}
```

```
"emmet.triggerExpansionOnTab": true
}
```

This enables Emmet HTML completion inside JavaScript template literals.

Recommended extensions:

- **Live Server** - Auto-reload on file changes
- **ES6 String HTML** - Syntax highlighting in template literals
- **ESLint** - JavaScript linting (configure for ES modules)
- **Path Intellisense** - Autocomplete for imports

5.3.2.2 Other Editors

Sublime Text: - Install “HTML-CSS-JS Prettify” - Use “JavaScript Next” syntax highlighting

Vim:

```
" In .vimrc
autocmd BufNewFile,BufRead *.mjs set filetype=javascript
```

WebStorm/IntelliJ:

- Built-in support for ES modules
- Configure web server for localhost testing

5.3.3 Local Development Server

LARC requires a local server (file:// URLs don’t work due to CORS and ES module restrictions). Choose any option:

5.3.3.1 Python (Built-in)

```
# Python 3 (most common)
$ python3 -m http.server 8000
# Serving HTTP on :: port 8000 (http://[::]:8000/) ...

# Python 2 (if needed)
$ python -m SimpleHTTPServer 8000
```

Pros: No installation, universally available, simple **Cons:** No hot reload, basic features

5.3.3.2 Node.js Options

http-server (simple):

```
$ npx http-server -p 8000 -c-1
# -c-1 disables caching (important during development)
```

live-server (with auto-reload):

```
$ npx live-server --port=8000 --no-browser
```

Vite (full-featured):

```
$ npx vite --port 8000
```

Vite provides:

- Instant hot module replacement
- Built-in TypeScript support
- Optimized production builds
- Plugin ecosystem

5.3.3.3 PHP (Built-in)

```
$ php -S localhost:8000
```

5.3.3.4 VS Code Extension

Install “Live Server” extension, then right-click `index.html` -> “Open with Live Server”. Changes reload automatically.

Recommended for beginners: Python (simplest) or Live Server extension (best DX)

Recommended for teams: Vite (most features, best performance)

5.4 Browser DevTools Setup

LARC applications are debuggable with standard browser DevTools. Here’s how to configure them effectively.

5.4.1 Chrome DevTools

Essential Panels:

1. **Console** - View PAN messages, errors, and logs
2. **Network** - Monitor component loading
3. **Elements** - Inspect Shadow DOM and component attributes
4. **Application** - Check localStorage, IndexedDB, and OPFS

Enable useful settings:

1. Open DevTools (F12 / Cmd+Option+I)
2. Settings (F1 / [gear] icon) -> Experiments
3. Enable:
 - “Show user agent shadow DOM” (to inspect component internals)
 - “CSS Authoring” (for live CSS editing)

Debugging PAN messages:

```
// Add to index.html during development
if (window.location.hostname === 'localhost') {
  // Subscribe to all messages
}
```

```
window.addEventListener('pan-ready', () => {
  window.panClient.subscribe('*', ({ topic, data, meta }) => {
    console.log(`[PAN] ${topic}`, data, meta);
  });
});
```

This logs every message flowing through the PAN bus.

5.4.2 Firefox Developer Tools

Firefox has excellent Web Components support:

1. Open DevTools (F12 / Cmd+Option+I)
2. Enable “Show Browser Styles” in Inspector settings
3. Use Console to inspect components:

```
// Get component instance
const card = document.querySelector('pan-card');
console.log(card);

// Inspect Shadow DOM
console.log(card.shadowRoot);

// Trigger methods
card.setAttribute('theme', 'dark');
```

5.4.3 Safari Web Inspector

Safari’s Web Inspector works well for debugging:

1. Enable Develop menu: Preferences -> Advanced -> Show Develop Menu
2. Develop -> Show Web Inspector (Cmd+Option+I)
3. Elements tab shows Shadow DOM boundaries clearly

Safari-specific considerations:

- OPFS not available (use IndexedDB fallback)
- BroadcastChannel available in Safari 15.4+
- Test iOS Safari separately (different engine version)

5.4.4 LARC DevTools Extension (Optional)

The LARC DevTools extension (in development) provides:

- Visual PAN message inspector
- Component tree visualization
- Performance profiling
- State inspection

Install from Chrome Web Store or Firefox Add-ons when available.

5.5 Your First LARC Application

Let's build a complete task manager to demonstrate LARC's development workflow. This isn't a trivial example—it includes multiple components, state management, persistence, and error handling.

5.5.1 Step 1: Project Setup

Create a new directory and add `index.html`:

```
$ mkdir task-manager
$ cd task-manager
```

`index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Task Manager - LARC Example</title>
  <style>
    * {
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }

    body {
      font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, sans-serif;
      background: #f5f5f5;
      padding: 2rem;
    }

    .container {
      max-width: 600px;
      margin: 0 auto;
    }

    h1 {
      margin-bottom: 2rem;
      color: #333;
    }
  </style>
</head>
<body>
  <div class="container">
    <h1>Task Manager</h1>
```

```
<!-- Load LARC from CDN -->
<script type="module" src="https://unpkg.com/@larcjs/core@2.0.0/src/pan.mjs"></script>

<!-- Load our custom components -->
<script type="module" src="./task-form.mjs"></script>
<script type="module" src="./task-list.mjs"></script>
<script type="module" src="./task-item.mjs"></script>

<!-- Use components -->
<task-form></task-form>
<task-list></task-list>
</div>
</body>
</html>
```

5.5.2 Step 2: Create Directory Structure

```
$ mkdir -p src/components src/utils
```

5.5.3 Step 3: Build the Task Form Component

Create `src/components/task-form.mjs`:

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

  connectedCallback() {
    this.render();
    this.setupEventListeners();
  }

  render() {
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
          margin-bottom: 2rem;
        }

        form {
          display: flex;
          gap: 0.5rem;
          background: white;
          padding: 1.5rem;
        }
      </style>
    `;
  }
}
```

```
    border-radius: 8px;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
  }

  input {
    flex: 1;
    padding: 0.75rem;
    border: 1px solid #ddd;
    border-radius: 4px;
    font-size: 1rem;
  }

  input:focus {
    outline: none;
    border-color: #4CAF50;
  }

  button {
    padding: 0.75rem 1.5rem;
    background: #4CAF50;
    color: white;
    border: none;
    border-radius: 4px;
    font-size: 1rem;
    cursor: pointer;
    transition: background 0.2s;
  }

  button:hover {
    background: #45a049;
  }

  button:disabled {
    background: #ccc;
    cursor: not-allowed;
  }
</style>

<form>
  <input
    type="text"
    placeholder="What needs to be done?"
    required
    autocomplete="off"
  />
  <button type="submit">Add Task</button>
</form>
```

```

    `;
  }

  setupEventListeners() {
    const form = this.shadowRoot.querySelector('form');
    const input = this.shadowRoot.querySelector('input');

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

      const text = input.value.trim();
      if (!text) return;

      // Publish task creation via PAN bus
      window.panClient.publish('tasks.add', {
        id: Date.now(),
        text,
        completed: false,
        createdAt: new Date().toISOString()
      });

      // Clear input
      input.value = '';
      input.focus();
    });
  }
}

customElements.define('task-form', TaskForm);

```

5.5.4 Step 4: Build the Task List Component

Create `src/components/task-list.mjs`:

```

class TaskList extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.tasks = [];
  }

  connectedCallback() {
    this.render();
    this.subscribeToMessages();
    this.loadTasks();
  }

  disconnectedCallback() {

```

```

    // Clean up subscriptions
    if (this.subscriptions) {
      this.subscriptions.forEach(sub => sub.unsubscribe());
    }
  }

  subscribeToMessages() {
    this.subscriptions = [
      // Listen for new tasks
      window.pancClient.subscribe('tasks.add', ({ data }) => {
        this.tasks.push(data);
        this.saveTasks();
        this.render();
      }),

      // Listen for task toggles
      window.pancClient.subscribe('tasks.toggle', ({ data }) => {
        const task = this.tasks.find(t => t.id === data.id);
        if (task) {
          task.completed = !task.completed;
          this.saveTasks();
          this.render();
        }
      }),

      // Listen for task deletions
      window.pancClient.subscribe('tasks.delete', ({ data }) => {
        this.tasks = this.tasks.filter(t => t.id !== data.id);
        this.saveTasks();
        this.render();
      })
    ];
  }

  loadTasks() {
    try {
      const stored = localStorage.getItem('larc-tasks');
      if (stored) {
        this.tasks = JSON.parse(stored);
        this.render();
      }
    } catch (error) {
      console.error('Failed to load tasks:', error);
    }
  }

  saveTasks() {

```

```

    try {
      localStorage.setItem('larc-tasks', JSON.stringify(this.tasks));
    } catch (error) {
      console.error('Failed to save tasks:', error);
    }
  }
}

render() {
  const total = this.tasks.length;
  const completed = this.tasks.filter(t => t.completed).length;

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

      .summary {
        background: white;
        padding: 1rem 1.5rem;
        border-radius: 8px;
        box-shadow: 0 2px 4px rgba(0,0,0,0.1);
        margin-bottom: 1rem;
        color: #666;
        font-size: 0.9rem;
      }

      .list {
        background: white;
        border-radius: 8px;
        box-shadow: 0 2px 4px rgba(0,0,0,0.1);
        overflow: hidden;
      }

      .empty {
        padding: 3rem;
        text-align: center;
        color: #999;
      }
    </style>

    ${total > 0 ? `
      <div class="summary">
        ${completed} of ${total} tasks completed
      </div>
    ` : ''}
  `
}

```

```

    <div class="list">
      ${total === 0 ? `
        <div class="empty">No tasks yet. Add one above!</div>
      ` : ''}
    </div>
  `;

  // Render task items
  const list = this.shadowRoot.querySelector('.list');
  this.tasks.forEach(task => {
    const item = document.createElement('task-item');
    item.setAttribute('task-id', task.id);
    item.setAttribute('text', task.text);
    if (task.completed) {
      item.setAttribute('completed', '');
    }
    list.appendChild(item);
  });
}
}

customElements.define('task-list', TaskList);

```

5.5.5 Step 5: Build the Task Item Component

Create src/components/task-item.mjs:

```

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

  static get observedAttributes() {
    return ['completed', 'text'];
  }

  connectedCallback() {
    this.render();
    this.setupEventListeners();
  }

  attributeChangedCallback() {
    if (this.shadowRoot) {
      this.render();
    }
  }
}

```

```
get taskId() {
  return parseInt(this.getAttribute('task-id'));
}

get text() {
  return this.getAttribute('text') || '';
}

get completed() {
  return this.hasAttribute('completed');
}

render() {
  this.shadowRoot.innerHTML = `
    <style>
      :host {
        display: block;
        border-bottom: 1px solid #eee;
      }

      :host(:last-child) {
        border-bottom: none;
      }

      .task {
        display: flex;
        align-items: center;
        gap: 1rem;
        padding: 1rem 1.5rem;
        transition: background 0.2s;
      }

      .task:hover {
        background: #f9f9f9;
      }

      input[type="checkbox"] {
        width: 20px;
        height: 20px;
        cursor: pointer;
      }

      .text {
        flex: 1;
        font-size: 1rem;
        transition: color 0.2s, text-decoration 0.2s;
      }
    `
}
```

```

        .text.completed {
            color: #999;
            text-decoration: line-through;
        }

        .delete {
            padding: 0.5rem 1rem;
            background: #f44336;
            color: white;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            font-size: 0.9rem;
            opacity: 0.7;
            transition: opacity 0.2s;
        }

        .delete:hover {
            opacity: 1;
        }
    </style>

    <div class="task">
        <input type="checkbox" ${this.completed ? 'checked' : ''} />
        <span class="text ${this.completed ? 'completed' : ''}">${this.text}</span>
        <button class="delete">Delete</button>
    </div>
    `;
}

setupEventListeners() {
    const checkbox = this.shadowRoot.querySelector('input[type="checkbox"]');
    const deleteBtn = this.shadowRoot.querySelector('.delete');

    checkbox.addEventListener('change', () => {
        window.pancClient.publish('tasks.toggle', {
            id: this.taskId
        });
    });

    deleteBtn.addEventListener('click', () => {
        window.pancClient.publish('tasks.delete', {
            id: this.taskId
        });
    });
}
}

```

```
customElements.define('task-item', TaskItem);
```

5.5.6 Step 6: Run the Application

Start your development server:

```
$ python3 -m http.server 8000
```

Open `http://localhost:8000` in your browser. You should see:

- A form to add tasks
- An empty state message
- Tasks appear when you add them
- Checkbox to toggle completion
- Delete button to remove tasks
- Task counter showing progress
- Persistence (refresh the page—tasks remain)

5.5.7 What You Just Built

This application demonstrates core LARC concepts:

1. **Component Composition** - Three components work together without direct coupling
2. **PAN Bus Messaging** - Components communicate through published messages
3. **Shadow DOM** - Each component has encapsulated styles
4. **Lifecycle Management** - Components clean up subscriptions properly
5. **State Persistence** - Data saved to `localStorage` automatically
6. **Progressive Enhancement** - Works without JavaScript for basic HTML

Try these experiments:

1. Open the Console and log all PAN messages:

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

2. Add a task and watch messages flow: `tasks.add`, updates trigger re-renders
3. Inspect Shadow DOM in Elements panel—each component's styles are isolated
4. Refresh the page—tasks persist via `localStorage`

5.6 Serving Static Files

LARC applications are just HTML, CSS, and JavaScript files. Any HTTP server works, but different servers offer different features.

5.6.1 Development Servers (Local)

Quick comparison:

Server	Setup	Hot Reload	HTTPS	Speed	Best For
Python	Built-in	No	No	Fast	Quick testing
Live Server	VS Code ext	Yes	Optional	Fast	Active development
http-server	npm	No	Optional	Fast	Node users
Vite	npm	Yes	Yes	Fastest	Full projects
PHP	Built-in	No	No	Fast	PHP developers

Recommended workflow:

1. **Learning/Experimenting:** Python (`python3 -m http.server`)
2. **Active Development:** Live Server extension or Vite
3. **Team Projects:** Vite or similar with shared config

5.6.2 Production Servers

Static hosting options:

1. **CDN-based:**
 - Netlify (drag-and-drop deployment)
 - Vercel (Git integration)
 - Cloudflare Pages (global edge network)
 - GitHub Pages (free for public repos)
2. **Traditional hosting:**
 - Nginx (most common)
 - Apache (with `mod_rewrite`)
 - Caddy (automatic HTTPS)
3. **Cloud platforms:**
 - AWS S3 + CloudFront
 - Google Cloud Storage + CDN
 - Azure Static Web Apps

Deployment is simple:

```
# Example: Netlify CLI
$ npm install -g netlify-cli
$ netlify deploy --dir=. --prod
```

Your LARC app deploys like any static site—just upload files. No build step required (though you can add optimization).

5.6.3 CORS and Module Loading

ES modules require proper MIME types. Most servers handle this automatically, but verify:

Nginx:

```
types {
    application/javascript mjs js;
}
```

Apache (.htaccess):

```
AddType application/javascript .mjs
```

Python: Works by default

If modules fail to load, check:

1. Console for CORS errors
2. Network tab for MIME type (`application/javascript`)
3. Server is running (`file://` URLs don't work)

5.7 Browser Requirements and Compatibility

LARC targets modern browsers with native Web Components support. Here's what you need to know.

5.7.1 Supported Browsers

Minimum versions (full support, no polyfills):

Browser	Version	Released	Market Share
Chrome	90+	April 2021	~65%
Edge	90+	April 2021	~5%
Firefox	88+	April 2021	~3%
Safari	14+	Sept 2020	~20%
Opera	76+	April 2021	~2%
Samsung	15+	April 2021	~3%

Total coverage: ~98% of global users (2025 data)

These versions support:

- Custom Elements v1
- Shadow DOM v1
- ES Modules
- IntersectionObserver
- MutationObserver
- Async/await

5.7.2 Feature Detection

Not all browsers support every optional feature. Use feature detection:

```
// Check for optional features
const features = {
  opfs: 'storage' in navigator && 'getDirectory' in navigator.storage,
  broadcastChannel: typeof BroadcastChannel !== 'undefined',
  resizeObserver: typeof ResizeObserver !== 'undefined',
  constructableStylesheets: 'adoptedStyleSheets' in Document.prototype
```

```
};

console.log('Available features:', features);

// Use features conditionally
if (features.opfs) {
  // Use Origin Private File System
} else {
  // Fall back to IndexedDB
}
```

5.7.3 Polyfills for Older Browsers

If you must support older browsers, add polyfills selectively:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>LARC with Polyfills</title>

  <!-- Feature detection -->
  <script>
    // Only load polyfills if needed
    if (!('customElements' in window)) {
      document.write('<script src="https://unpkg.com/@webcomponents/webcomponentsjs@2/webcomponentsjs"></script>');
    }

    if (typeof BroadcastChannel === 'undefined') {
      document.write('<script src="https://unpkg.com/broadcast-channel@4/dist/bundle.js"></script>');
    }
  </script>

  <!-- Load LARC after polyfills -->
  <script type="module" src="https://unpkg.com/@larcjs/core@2.0.0/src/pan.mjs"></script>
</head>
<body>
  <!-- Your app -->
</body>
</html>
```

Polyfill size impact:

- Web Components: ~30 KB gzipped
- BroadcastChannel: ~5 KB gzipped
- Total: ~35 KB for full compatibility

Recommendation: Target modern browsers only. Display upgrade message for IE11 and older browsers:

```
<!--[if IE]>
<div style="padding: 2rem; background: #fff3cd; border: 2px solid #ffc107;">
  <h2>Browser Update Required</h2>
  <p>This application requires a modern browser. Please upgrade to Chrome 90+, Firefox 88+, Sa.
</div>
<![endif]-->
```

5.7.4 Mobile Browser Support

LARC works on mobile browsers with the same requirements:

iOS: - Safari 14+ (iOS 14+) - Chrome 90+ (iOS) - Full support except OPFS (use IndexedDB fallback)

Android: - Chrome 90+ (Android 7+) - Samsung Internet 15+ - Full support including OPFS

Test on real devices:

```
# Start server accessible on network
$ python3 -m http.server 8000

# Find your local IP
$ ifconfig | grep "inet " # Mac/Linux
$ ipconfig                # Windows

# Access from mobile browser
# http://192.168.1.100:8000
```

5.8 Common Troubleshooting

5.8.1 Problem: Components Don't Load

Symptoms: Page blank, no errors, or “Failed to fetch” in Console

Solutions:

1. Check server is running:

```
$ python3 -m http.server 8000
# Should show "Serving HTTP on..."
```

2. Verify file paths:

```
// Check Network tab in DevTools
// Look for 404 errors
```

3. Check MIME types:

```
# Should be application/javascript
$ curl -I http://localhost:8000/task-form.mjs
```

4. Verify module syntax:

```
// Must use .mjs extension or type="module"
<script type="module" src="./app.js"></script>
```

5.8.2 Problem: PAN Bus Messages Not Received

Symptoms: Components don't update when publishing messages

Solutions:

1. Wait for PAN bus initialization:

```
// Wrong - panClient may not exist yet
window.panClient.publish('test', {});

// Right - wait for pan-ready event
window.addEventListener('pan-ready', () => {
  window.panClient.publish('test', {});
});
```

2. Check subscriptions:

```
// Debug: Log all messages
panClient.subscribe('*', ({ topic, data }) => {
  console.log('[PAN]', topic, data);
});
```

3. Verify topic patterns:

```
// Specific
panClient.subscribe('tasks.add', handler); // Only tasks.add

// Wildcard
panClient.subscribe('tasks.*', handler); // All tasks.* topics

// All
panClient.subscribe('*', handler); // Everything
```

5.8.3 Problem: Styles Not Applying

Symptoms: Components render but look unstyled

Solutions:

1. Check Shadow DOM encapsulation:

```
// Global styles don't penetrate Shadow DOM
// Use :host selector
this.shadowRoot.innerHTML = `
<style>
  :host {
    display: block; /* Applied to component itself */
  }
`
```

```

    .inner {
      color: blue;      /* Applied to internal elements */
    }
  </style>
`;

```

2. Use CSS Custom Properties for theming:

```

/* Global (outside Shadow DOM) */
:root {
  --primary-color: #4CAF50;
}

/* Component (inside Shadow DOM) */
button {
  background: var(--primary-color);
}

```

5.8.4 Problem: localStorage Quota Exceeded

Symptoms: “QuotaExceededError” in Console

Solutions:

1. Check storage usage:

```

if ('storage' in navigator && 'estimate' in navigator.storage) {
  navigator.storage.estimate().then(({ usage, quota }) => {
    console.log(`Using ${usage} of ${quota} bytes`);
  });
}

```

2. Implement data cleanup:

```

try {
  localStorage.setItem('data', JSON.stringify(data));
} catch (error) {
  if (error.name === 'QuotaExceededError') {
    // Clear old data
    localStorage.clear();
    // Try again
    localStorage.setItem('data', JSON.stringify(data));
  }
}

```

3. Use IndexedDB for large datasets:

```

// IndexedDB has much higher quota (typically 50%+ of disk space)

```

5.8.5 Problem: CORS Errors

Symptoms: “CORS policy” errors when loading components

Solutions:

1. Use proper server (not file://):

```
# Don't open index.html directly in browser
# Always use HTTP server
$ python3 -m http.server 8000
```

2. Check CDN CORS headers (usually automatic)

3. For API calls, configure server CORS:

```
// Server must send appropriate headers
Access-Control-Allow-Origin: *
```

5.9 Next Steps

You now have a working LARC development environment and understand the basic workflow:

1. Create HTML file with component tags
2. Load LARC autoloader
3. Write components as Custom Elements
4. Communicate through PAN bus
5. Test in browser with DevTools
6. Deploy as static files

In Chapter 6, we'll explore state management in depth—how to handle complex application state, implement undo/redo, synchronize across tabs, and persist data reliably. The task manager you built is a foundation; next, we'll scale it up.

Before moving on, experiment with your task manager:

- Add filtering (all/active/completed)
- Implement task editing
- Add due dates
- Create categories
- Export/import tasks

Each feature is an opportunity to practice PAN messaging and component composition. The patterns you learn here apply to any LARC application.

Welcome to zero-build development. The browser is your development environment now.

Chapter 6

Basic Message Flow

“In the beginning was the Message, and the Message was with the Bus, and the Message was the Bus. And the Bus said, ‘Let there be publish-subscribe,’ and there was publish-subscribe, and it was good—mostly because it avoided callback hell.”

— The Book of Reactive Programming, Chapter 1, Verse 1

If you’ve made it through the previous chapters, you now understand the philosophical underpinnings of LARC, its architecture, and how to set up a basic application. But philosophy and architecture don’t ship features. Messages do.

In this chapter, we’ll dive deep into the beating heart of LARC: the message flow. We’ll explore how messages are published, how components subscribe to topics, how to use wildcard patterns to listen for multiple message types at once, and how to clean up after yourself when the party’s over. Think of this chapter as your field guide to the PAN bus—the communication backbone that makes LARC applications tick.

6.1 The Anatomy of a Message

Before we start slinging messages around like a caffeinated postal worker, let’s understand what a message actually is in LARC.

A message in LARC is delightfully simple: it’s a plain JavaScript object with two required properties:

```
{
  topic: "user.login",
  data: {
    userId: "12345",
    username: "alice",
    timestamp: Date.now()
  }
}
```

That’s it. The `topic` is a string that categorizes the message, and `data` is whatever payload you want to send along for the ride. This simplicity is intentional—LARC doesn’t impose schemas, validation, or type systems on your messages. It trusts you to be a responsible adult (though it

secretly hopes you're using TypeScript).

The topic follows a hierarchical naming convention using dots as separators, much like DNS names or Java package names. This convention enables powerful pattern matching, as we'll see shortly.

6.2 Publishing Your First Message

Publishing a message is as straightforward as calling a function. In fact, it *is* calling a function:

```
import { publish } from '@larc/core';

// Publish a message
publish('user.login', {
  userId: '12345',
  username: 'alice',
  timestamp: Date.now()
});
```

When you call `publish()`, LARC does several things:

1. It wraps your data in a message envelope with the specified topic
2. It routes the message to all subscribers interested in that topic
3. It optionally stores the message for later retrieval (more on this in a moment)
4. It returns immediately, because publishing is non-blocking

That last point is crucial. Publishing a message doesn't wait for subscribers to process it. It's fire-and-forget, like throwing a message in a bottle into the ocean, except the ocean is your application's memory space and the bottle is a JavaScript object. And unlike real bottles, these arrive instantly—or at least as instantly as the JavaScript event loop allows.

6.2.1 Publishing from Components

In most real applications, you'll publish messages from within web components. Here's a more realistic example:

```
class LoginForm extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <form id="login-form">
        <input type="text" id="username" placeholder="Username" />
        <input type="password" id="password" placeholder="Password" />
        <button type="submit">Log In</button>
      </form>
    `;

    this.querySelector('#login-form').addEventListener('submit', (e) => {
      e.preventDefault();
      this.handleLogin();
    });
  }
}
```

```
async handleLogin() {
  const username = this.querySelector('#username').value;
  const password = this.querySelector('#password').value;

  // Publish a login attempt message
  publish('auth.login.attempt', { username });

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

    if (response.ok) {
      const user = await response.json();

      // Publish success message
      publish('auth.login.success', {
        userId: user.id,
        username: user.username,
        roles: user.roles,
        timestamp: Date.now()
      });
    } else {
      // Publish failure message
      publish('auth.login.failure', {
        username,
        reason: 'Invalid credentials',
        timestamp: Date.now()
      });
    }
  } catch (error) {
    // Publish error message
    publish('auth.login.error', {
      username,
      error: error.message,
      timestamp: Date.now()
    });
  }
}

customElements.define('login-form', LoginForm);
```

Notice how we're publishing multiple messages at different stages of the login process. This granularity gives other parts of the application fine-grained awareness of what's happening. An

analytics component might care about login attempts, while a notification component only cares about successes and failures.

6.3 Subscribing to Topics

Publishing messages into the void is about as useful as shouting into a pillow. To make messages meaningful, you need subscribers—components that listen for specific topics and react accordingly.

Subscribing is just as simple as publishing:

```
import { subscribe } from '@larc/core';

// Subscribe to a topic
const unsubscribe = subscribe('user.login', (message) => {
  console.log('User logged in:', message.data);
});
```

The `subscribe()` function takes two arguments: a topic pattern and a callback function. When a message matching that pattern is published, your callback is invoked with the message object.

Notice that `subscribe()` returns a function. That function, conventionally called `unsubscribe`, removes your subscription when called. More on cleanup later.

6.3.1 Subscription Example: Notification System

Let's build a component that displays notifications for authentication events:

```
class NotificationCenter extends HTMLElement {
  connectedCallback() {
    this.subscriptions = [];
    this.innerHTML = '<div id="notifications"></div>';

    // Subscribe to success messages
    this.subscriptions.push(
      subscribe('auth.login.success', (msg) => {
        this.showNotification(
          `Welcome back, ${msg.data.username}!`,
          'success'
        );
      })
    );

    // Subscribe to failure messages
    this.subscriptions.push(
      subscribe('auth.login.failure', (msg) => {
        this.showNotification(
          `Login failed: ${msg.data.reason}`,
          'error'
        );
      })
    );
  }
}
```

```

    );

    // Subscribe to error messages
    this.subscriptions.push(
      subscribe('auth.login.error', (msg) => {
        this.showNotification(
          `An error occurred: ${msg.data.error}`,
          'error'
        );
      })
    );
  }

  showNotification(message, type) {
    const notification = document.createElement('div');
    notification.className = `notification notification-${type}`;
    notification.textContent = message;

    this.querySelector('#notifications').appendChild(notification);

    // Auto-remove after 5 seconds
    setTimeout(() => notification.remove(), 5000);
  }

  disconnectedCallback() {
    // Clean up subscriptions
    this.subscriptions.forEach(unsub => unsub());
  }
}

customElements.define('notification-center', NotificationCenter);

```

This component demonstrates several best practices:

1. **Store unsubscribe functions:** Keep references to all your subscriptions so you can clean them up later
2. **React to messages:** The callback functions update the UI in response to published messages
3. **Clean up in disconnectedCallback:** When the component is removed from the DOM, unsubscribe from all topics

6.4 Wildcard Patterns: The Power of Asterisks

Subscribing to individual topics is fine for simple cases, but it gets tedious fast. Imagine subscribing to `auth.login.success`, `auth.login.failure`, `auth.logout.success`, `auth.logout.failure`, `auth.refresh.success`, `auth.refresh.failure`—you’d need six separate subscriptions!

Enter wildcard patterns. LARC supports two wildcard characters:

- `*` matches a single topic segment

- `**` matches zero or more topic segments

Here are some examples:

```
// Match any auth-related login message
subscribe('auth.login.*', (msg) => {
  console.log('Login event:', msg.topic, msg.data);
});

// Match any auth message at any depth
subscribe('auth.**', (msg) => {
  console.log('Auth event:', msg.topic, msg.data);
});

// Match any success message for any operation
subscribe('*.*.success', (msg) => {
  console.log('Success:', msg.topic, msg.data);
});

// Match all messages (use sparingly!)
subscribe('**', (msg) => {
  console.log('All messages:', msg.topic, msg.data);
});
```

The single asterisk (*) matches exactly one segment. The pattern `auth.*.success` would match `auth.login.success` and `auth.logout.success`, but not `auth.success` (too few segments) or `auth.user.login.success` (too many segments).

The double asterisk (**) is greedier. It matches any number of segments, including zero. The pattern `auth.**` matches `auth.login`, `auth.login.success`, `auth.user.profile.update`, and even just `auth` (though publishing a message with a single-segment topic is unusual).

6.4.1 Practical Wildcard Example: Audit Logger

Let's build an audit logger that records all authentication-related activities:

```
class AuditLogger extends HTMLElement {
  connectedCallback() {
    this.logs = [];

    // Subscribe to all auth events
    this.unsubscribe = subscribe('auth.**', (msg) => {
      this.logEvent(msg);
    });

    this.render();
  }

  logEvent(msg) {
    const logEntry = {
```

```

        timestamp: new Date().toISOString(),
        topic: msg.topic,
        data: msg.data
    };

    this.logs.push(logEntry);

    // Persist to localStorage
    localStorage.setItem('audit-logs', JSON.stringify(this.logs));

    this.render();
}

render() {
    this.innerHTML = `
        <div class="audit-logger">
            <h2>Audit Log</h2>
            <table>
                <thead>
                    <tr>
                        <th>Timestamp</th>
                        <th>Event</th>
                        <th>Details</th>
                    </tr>
                </thead>
                <tbody>
                    ${this.logs.map(log => `
                        <tr>
                            <td>${log.timestamp}</td>
                            <td>${log.topic}</td>
                            <td>${JSON.stringify(log.data)}</td>
                        </tr>
                    `).join('')}
                </tbody>
            </table>
        </div>
    `;
}

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

customElements.define('audit-logger', AuditLogger);

```

This component uses `auth.**` to capture every authentication-related message, regardless of its specific operation or outcome. It's a powerful pattern for cross-cutting concerns like logging, analytics, or debugging.

6.5 Message Retention: The PAN Bus Remembers

One of the more clever features of LARC's PAN bus is message retention. By default, the PAN bus retains the most recent message for each topic. This means that when a component subscribes to a topic, it immediately receives the last published message, if one exists.

This behavior solves a common problem in reactive systems: the “late subscriber” problem. Imagine a component that displays the current user's profile. If it subscribes to `user.profile` after the profile has already been loaded, it would normally miss that message and show stale or empty data. With message retention, it gets the current profile immediately upon subscribing.

Here's an example:

```
// Somewhere early in the app lifecycle
publish('user.profile', {
  userId: '12345',
  username: 'alice',
  email: 'alice@example.com'
});

// Later, a component subscribes
class UserProfile extends HTMLElement {
  connectedCallback() {
    this.unsubscribe = subscribe('user.profile', (msg) => {
      this.render(msg.data);
    });
    // The callback fires immediately with the retained message
  }

  render(profile) {
    this.innerHTML = `
      <div class="user-profile">
        <h2>${profile.username}</h2>
        <p>${profile.email}</p>
      </div>
    `;
  }

  disconnectedCallback() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }
}
```

```
customElements.define('user-profile', UserProfile);
```

Even though `user-profile` subscribed after the message was published, it still receives the profile data immediately. This makes components more robust and eliminates race conditions.

6.5.1 Controlling Retention

Not all messages should be retained. Ephemeral events like `button.clicked` or `mouse.moved` would be pointless to retain—by the time a late subscriber arrives, the event is ancient history.

LARC allows you to control retention on a per-topic basis using a configuration object:

```
import { configure } from '@larc/core';

configure({
  retention: {
    'user.profile': true,           // Retain
    'user.settings': true,         // Retain
    'auth.login.attempt': false,   // Don't retain
    'mouse.*': false,              // Don't retain any mouse events
    '**': true                     // Default: retain everything else
  }
});
```

The retention configuration uses the same wildcard pattern matching as subscriptions. More specific patterns override less specific ones.

6.5.2 Retention Gotchas

Message retention is powerful, but it has pitfalls:

1. **Memory Usage:** Retained messages live in memory. If you're publishing thousands of unique topics, you'll accumulate thousands of messages. Consider using less granular topics or disabling retention for high-volume streams.
2. **Stale Data:** Retained messages can be stale. If a component subscribes to `user.profile` but the profile was loaded five minutes ago, is that data still valid? Always consider whether you need to refresh data after receiving a retained message.
3. **Surprising Callbacks:** Because subscriptions fire immediately if a retained message exists, your callback might execute synchronously during the `subscribe()` call. If your callback manipulates the DOM or performs side effects, ensure the component is fully initialized first.

6.6 Message Ordering and Synchronization

LARC processes messages synchronously in the order they're published. If you publish three messages in sequence:

```
publish('event.one', { value: 1 });
publish('event.two', { value: 2 });
publish('event.three', { value: 3 });
```

All subscribers will receive them in that exact order: one, two, three. This guarantee simplifies reasoning about message flow and eliminates many race conditions.

However, this guarantee only applies within a single JavaScript execution context. If you publish a message, then await an asynchronous operation, then publish another message, other code may publish messages in between:

```
publish('step.one', {});
await fetch('/api/data'); // Other code runs during this await
publish('step.two', {});
```

If you need strict ordering across asynchronous boundaries, consider batching messages or using sequence numbers:

```
let sequenceNumber = 0;

async function performOperation() {
  const seq = ++sequenceNumber;

  publish('operation.start', { sequence: seq });

  try {
    const result = await doAsyncWork();
    publish('operation.complete', { sequence: seq, result });
  } catch (error) {
    publish('operation.error', { sequence: seq, error: error.message });
  }
}
```

Subscribers can then use the sequence number to reorder messages if needed.

6.7 Unsubscribing and Cleanup

Every `subscribe()` call returns an `unsubscribe` function. Calling this function removes the subscription and prevents future messages from triggering the callback:

```
const unsubscribe = subscribe('user.login', (msg) => {
  console.log('User logged in:', msg.data);
});

// Later, when you're done listening
unsubscribe();
```

Failing to unsubscribe is a common source of memory leaks and bugs. If a component subscribes to a topic but never unsubscribes, the callback remains in memory even after the component is removed from the DOM. This keeps the component alive, prevents garbage collection, and may cause the callback to fire unexpectedly.

6.7.1 Cleanup Patterns

The most reliable cleanup pattern is to unsubscribe in the component's `disconnectedCallback()`:

```
class MyComponent extends HTMLElement {
  connectedCallback() {
    this.unsubscribe = subscribe('some.topic', (msg) => {
      this.handleMessage(msg);
    });
  }

  disconnectedCallback() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }
}
```

For multiple subscriptions, store them in an array:

```
class MyComponent extends HTMLElement {
  connectedCallback() {
    this.subscriptions = [
      subscribe('topic.one', this.handleOne.bind(this)),
      subscribe('topic.two', this.handleTwo.bind(this)),
      subscribe('topic.three', this.handleThree.bind(this))
    ];
  }

  disconnectedCallback() {
    this.subscriptions.forEach(unsub => unsub());
    this.subscriptions = [];
  }
}
```

Or, if you're feeling fancy, use a helper function:

```
class MyComponent extends HTMLElement {
  constructor() {
    super();
    this.subscriptions = new Set();
  }

  subscribe(topic, callback) {
    const unsub = subscribe(topic, callback);
    this.subscriptions.add(unsub);
    return unsub;
  }

  connectedCallback() {
    this.subscribe('topic.one', this.handleOne.bind(this));
    this.subscribe('topic.two', this.handleTwo.bind(this));
    this.subscribe('topic.three', this.handleThree.bind(this));
  }
}
```

```

}

disconnectedCallback() {
  this.subscriptions.forEach(unsub => unsub());
  this.subscriptions.clear();
}
}

```

This pattern wraps the `subscribe()` function and automatically tracks subscriptions, making cleanup effortless.

6.8 Debugging Message Flow

As your application grows, understanding message flow becomes increasingly important. LARC provides several tools to help debug and visualize messages.

6.8.1 Console Logging

The simplest debugging technique is to log all messages:

```

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

```

This logs every message published in your application. It's noisy, but invaluable when tracking down mysterious bugs or understanding component interactions.

6.8.2 Conditional Logging

For more targeted debugging, use patterns:

```

// Log only auth-related messages
subscribe('auth.**', (msg) => {
  console.log(`[AUTH] ${msg.topic}`, msg.data);
});

// Log only errors
subscribe('*.*.error', (msg) => {
  console.error(`[ERROR] ${msg.topic}`, msg.data);
});

```

6.8.3 Message Inspector Component

For a more sophisticated approach, build a message inspector component:

```

class MessageInspector extends HTMLElement {
  constructor() {
    super();
    this.messages = [];
    this.maxMessages = 100;
  }
}

```

```

    this.filter = '';
  }

  connectedCallback() {
    this.unsubscribe = subscribe('**', (msg) => {
      this.messages.unshift({
        timestamp: new Date().toISOString(),
        topic: msg.topic,
        data: msg.data
      });

      if (this.messages.length > this.maxMessages) {
        this.messages.pop();
      }

      this.render();
    });

    this.render();
  }

  render() {
    const filteredMessages = this.filter
      ? this.messages.filter(m => m.topic.includes(this.filter))
      : this.messages;

    this.innerHTML = `
    <div class="message-inspector">
      <h2>Message Inspector</h2>
      <input
        type="text"
        placeholder="Filter by topic..."
        value="${this.filter}"
        id="filter-input"
      />
      <table>
        <thead>
          <tr>
            <th>Time</th>
            <th>Topic</th>
            <th>Data</th>
          </tr>
        </thead>
        <tbody>
          ${filteredMessages.map(msg => `
            <tr>
              <td>${msg.timestamp}</td>

```

```

        <td><code>${msg.topic}</code></td>
        <td><pre>${JSON.stringify(msg.data, null, 2)}</pre></td>
      </tr>
    `).join('')}
  </tbody>
</table>
</div>
`;

const input = this.querySelector('#filter-input');
if (input) {
  input.addEventListener('input', (e) => {
    this.filter = e.target.value;
    this.render();
  });
}

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

customElements.define('message-inspector', MessageInspector);

```

Add this component to your app during development, and you'll have a real-time view of all message traffic, complete with filtering capabilities.

6.9 Performance Considerations

The PAN bus is fast, but it's not magic. Publishing messages and invoking callbacks takes time. Here are some guidelines for keeping performance optimal:

1. **Publish sparingly:** Don't publish messages inside tight loops or high-frequency events (like `mousemove`). If you must, throttle or debounce your publications.
2. **Keep callbacks fast:** Subscriber callbacks are invoked synchronously. If a callback does heavy computation or DOM manipulation, it blocks message processing. Consider deferring work with `requestAnimationFrame()` or `setTimeout()`.
3. **Unsubscribe aggressively:** Every active subscription consumes memory and adds overhead to message routing. Unsubscribe as soon as you no longer need messages.
4. **Use specific topics:** Wildcard subscriptions are powerful but expensive. A subscription to `**` matches every message, so its callback runs for every publication. Use the most specific pattern that meets your needs.

5. **Avoid retained message bloat:** If you have hundreds of unique topics, you'll have hundreds of retained messages. Consider whether retention is necessary for each topic.

6.10 Common Patterns and Anti-Patterns

6.10.1 Pattern: Command-Query Separation

Distinguish between commands (messages that request actions) and events (messages that announce completed actions):

```
// Command: requesting an action
publish('user.profile.update', { userId: '12345', name: 'Alice' });

// Event: announcing a completed action
publish('user.profile.updated', { userId: '12345', name: 'Alice' });
```

Commands are typically imperatives (“update”, “delete”, “send”), while events are past tense (“updated”, “deleted”, “sent”). This distinction makes message flow clearer.

6.10.2 Pattern: Namespacing

Use a consistent namespace hierarchy for topics:

```
// Good: hierarchical namespacing
publish('app.user.profile.updated', { ... });
publish('app.ui.theme.changed', { ... });
publish('app.data.sync.complete', { ... });

// Bad: flat namespace
publish('profileUpdated', { ... });
publish('themeChanged', { ... });
publish('syncComplete', { ... });
```

Hierarchical naming enables powerful wildcard subscriptions and makes the codebase easier to navigate.

6.10.3 Anti-Pattern: Publishing Without Data

Avoid publishing messages without meaningful data:

```
// Bad
publish('user.login', {});

// Good
publish('user.login', {
  userId: '12345',
  username: 'alice',
  timestamp: Date.now()
});
```

Even if subscribers don't currently need the data, they might in the future. Publishing rich data makes messages more useful and reduces the need for additional queries.

6.10.4 Anti-Pattern: Overloading Topics

Don't use the same topic for multiple purposes:

```
// Bad: same topic, different meanings
publish('user.action', { type: 'login', userId: '12345' });
publish('user.action', { type: 'logout', userId: '12345' });

// Good: distinct topics
publish('user.login', { userId: '12345' });
publish('user.logout', { userId: '12345' });
```

Overloading topics forces subscribers to inspect message data to determine intent, which defeats the purpose of topic-based routing.

6.11 Wrapping Up

You've now mastered the basics of message flow in LARC. You can publish messages, subscribe to topics, use wildcard patterns, leverage message retention, and clean up subscriptions. These are the fundamental skills you'll use in every LARC application.

In the next chapter, we'll build on this foundation and explore how to create reusable, composable web components that communicate seamlessly via the PAN bus. You'll learn about component lifecycle, Shadow DOM considerations, and patterns for building complex UIs from simple, loosely-coupled components.

But before we move on, take a moment to experiment. Fire up a LARC application, add a `message-inspector` component, and publish some messages. Watch them flow through the system. Subscribe with different wildcard patterns and see how they match. The best way to internalize these concepts is to play with them.

Remember: messages are the lifeblood of a LARC application. Treat them with care, name them thoughtfully, and they'll reward you with a system that's easy to understand, extend, and debug. And when things inevitably go wrong, you'll have the tools to trace message flow and identify the problem.

Now, onward to components.

Chapter 7

Working with Components

“Give a developer a component, and they’ll build a page. Teach a developer to build components, and they’ll build an empire—or at least a reasonably maintainable SPA.”

— Ancient Web Development Proverb (circa 2015)

If the PAN bus is the nervous system of a LARC application, components are the organs. They’re the visible, interactive pieces that users actually see and touch. They render UI, respond to user input, and communicate with each other through the message bus we explored in Chapter 6.

But components in LARC aren’t just any components—they’re web components, which means they’re built on browser standards rather than framework-specific abstractions. This gives them superpowers: they work anywhere, outlive framework churn, and compose beautifully with both LARC and non-LARC code.

In this chapter, we’ll explore how to create web components in LARC applications, understand their lifecycle, work with Shadow DOM, connect components via the PAN bus, and design reusable components that stand the test of time.

7.1 Web Components: A Brief Refresher

Before we dive into LARC-specific patterns, let’s review the three web standards that comprise “web components”:

1. **Custom Elements:** The API for defining new HTML elements with custom behavior
2. **Shadow DOM:** Encapsulated DOM trees that isolate styles and markup
3. **HTML Templates:** Reusable chunks of markup that can be cloned and inserted

LARC leans heavily on Custom Elements and uses Shadow DOM where appropriate. HTML Templates are less common in LARC applications because most components render dynamically based on message data, but they’re available if you need them.

Here’s the most basic custom element:

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

```

}

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

```

And here's how you use it:

```
<hello-world></hello-world>
```

That's all there is to it. No build step, no framework, no magic. Just JavaScript and HTML.

7.2 The Component Lifecycle

Custom elements have a well-defined lifecycle with four main callbacks:

1. **constructor()**: Called when an instance is created. Use this for initializing state, but don't manipulate the DOM or attributes here.
2. **connectedCallback()**: Called when the element is inserted into the DOM. This is where you should render content, set up subscriptions, and add event listeners.
3. **disconnectedCallback()**: Called when the element is removed from the DOM. Use this for cleanup: unsubscribe from messages, remove event listeners, and cancel any pending work.
4. **attributeChangedCallback(name, oldValue, newValue)**: Called when an observed attribute changes. Declare which attributes to observe with the static **observedAttributes** getter.

Here's a component that uses all four:

```

class UserBadge extends HTMLElement {
  static get observedAttributes() {
    return ['user-id'];
  }

  constructor() {
    super();
    this.userData = null;
  }

  connectedCallback() {
    // Subscribe to user data updates
    this.unsubscribe = subscribe('user.data', (msg) => {
      if (msg.data.userId === this.getAttribute('user-id')) {
        this.userData = msg.data;
        this.render();
      }
    });

    // Initial render
    this.render();
  }
}

```

```

disconnectedCallback() {
  // Clean up subscription
  if (this.unsubscribe) {
    this.unsubscribe();
  }
}

attributeChangedCallback(name, oldValue, newValue) {
  if (name === 'user-id' && oldValue !== newValue) {
    // Attribute changed, re-render
    this.render();
  }
}

render() {
  if (this.userData) {
    this.innerHTML = `
      <div class="user-badge">
        
        <span>${this.userData.name}</span>
      </div>
    `;
  } else {
    this.innerHTML = '<div class="user-badge loading">Loading...</div>';
  }
}
}

customElements.define('user-badge', UserBadge);

```

Notice how the component follows a clear pattern:

- Initialize state in `constructor()`
- Set up subscriptions and render in `connectedCallback()`
- Clean up in `disconnectedCallback()`
- React to attribute changes in `attributeChangedCallback()`

This pattern is robust and works for most LARC components.

7.3 Shadow DOM: To Use or Not to Use?

Shadow DOM is one of the more controversial features of web components. It provides encapsulation—styles inside the shadow tree don’t leak out, and styles outside don’t leak in—but this encapsulation comes with tradeoffs.

7.3.1 When to Use Shadow DOM

Use Shadow DOM when:

1. **You need style isolation:** Your component should look consistent regardless of the page's CSS
2. **You're building a library:** If others will use your component, Shadow DOM prevents style conflicts
3. **You want internal complexity hidden:** The shadow tree's internal structure is hidden from external JavaScript

Here's a component using Shadow DOM:

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

  connectedCallback() {
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: inline-block;
        }
        button {
          background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
          color: white;
          border: none;
          padding: 12px 24px;
          font-size: 16px;
          border-radius: 8px;
          cursor: pointer;
          transition: transform 0.2s;
        }
        button:hover {
          transform: scale(1.05);
        }
        button:active {
          transform: scale(0.95);
        }
      </style>
      <button><slot></slot></button>
    `;

    this.shadowRoot.querySelector('button').addEventListener('click', (e) => {
      this.dispatchEvent(new CustomEvent('fancy-click', {
        bubbles: true,
        composed: true
      }));
    });
  }
}
```

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

The `:host` selector styles the component itself, and `<slot>` projects content from the light DOM into the shadow DOM. The button's styles are completely isolated—no external CSS can affect them.

7.3.2 When to Avoid Shadow DOM

Avoid Shadow DOM when:

1. **You need global styles:** If your component should inherit the page's theme, Shadow DOM makes this harder
2. **You need simple DOM manipulation:** Shadow DOM adds complexity when you just want to insert some HTML
3. **You're building app-specific components:** For components that are tightly coupled to a single application, encapsulation is often overkill

Most LARC components don't use Shadow DOM. They rely on scoped CSS classes and BEM-style naming conventions instead:

```
class UserProfile extends HTMLElement {  
  connectedCallback() {  
    this.className = 'user-profile';  
    this.render();  
  }  
  
  render() {  
    this.innerHTML = `  
      <div class="user-profile__header">  
        <h2 class="user-profile__name">Alice</h2>  
      </div>  
      <div class="user-profile__details">  
        <p class="user-profile__email">alice@example.com</p>  
      </div>  
    `;  
  }  
}
```

```
customElements.define('user-profile', UserProfile);
```

This approach is simpler and allows global styles to influence the component, which is often desirable in application UIs.

7.4 Connecting Components via the PAN Bus

This is where LARC shines. Components don't call methods on each other or pass data through complex prop chains. Instead, they communicate through the PAN bus by publishing and subscribing

to messages.

Let's build a multi-component example: a simple shopping cart system.

7.4.1 Component 1: Product Catalog

```
class ProductCatalog extends HTMLElement {
  connectedCallback() {
    this.products = [
      { id: 1, name: 'Widget', price: 10 },
      { id: 2, name: 'Gadget', price: 20 },
      { id: 3, name: 'Doohickey', price: 30 }
    ];

    this.render();
  }

  render() {
    this.innerHTML = `
      <div class="product-catalog">
        <h2>Products</h2>
        ${this.products.map(product => `
          <div class="product">
            <h3>${product.name}</h3>
            <p>${product.price}</p>
            <button data-product-id="${product.id}>Add to Cart</button>
          </div>
        `).join('')}
      </div>
    `;

    this.querySelectorAll('button').forEach(button => {
      button.addEventListener('click', () => {
        const productId = parseInt(button.dataset.productId);
        const product = this.products.find(p => p.id === productId);

        // Publish a message when a product is added to the cart
        publish('cart.item.added', {
          productId: product.id,
          name: product.name,
          price: product.price,
          quantity: 1
        });
      });
    });
  }
}
```

```
customElements.define('product-catalog', ProductCatalog);
```

7.4.2 Component 2: Shopping Cart

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

  connectedCallback() {
    // Subscribe to cart events
    this.unsubscribe = subscribe('cart.item.added', (msg) => {
      this.addItem(msg.data);
    });

    this.render();
  }

  addItem(item) {
    const existing = this.items.find(i => i.productId === item.productId);

    if (existing) {
      existing.quantity += item.quantity;
    } else {
      this.items.push({ ...item });
    }

    // Publish updated cart state
    publish('cart.updated', {
      items: this.items,
      total: this.calculateTotal()
    });

    this.render();
  }

  calculateTotal() {
    return this.items.reduce((sum, item) => sum + (item.price * item.quantity), 0);
  }

  render() {
    this.innerHTML = `
      <div class="shopping-cart">
        <h2>Cart</h2>
        ${this.items.length === 0 ? '<p>Cart is empty</p>' : `
          <ul>
```

```

        ${this.items.map(item => `
            <li>
                ${item.name} x ${item.quantity} - $$${item.price * item.quantity}
            </li>
        `).join('')}
    </ul>
    <p><strong>Total: $$${this.calculateTotal()}</strong></p>
    `}
</div>
`;
}

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

customElements.define('shopping-cart', ShoppingCart);

```

7.4.3 Component 3: Cart Badge

```

class CartBadge extends HTMLElement {
    constructor() {
        super();
        this.itemCount = 0;
    }

    connectedCallback() {
        // Subscribe to cart updates
        this.unsubscribe = subscribe('cart.updated', (msg) => {
            this.itemCount = msg.data.items.reduce((sum, item) => sum + item.quantity, 0);
            this.render();
        });

        this.render();
    }

    render() {
        this.innerHTML = `
            <div class="cart-badge">
                [cart] ${this.itemCount}
            </div>
        `;
    }
}

```

```

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

customElements.define('cart-badge', CartBadge);

```

7.4.4 Putting It All Together

```

<!DOCTYPE html>
<html>
<head>
  <title>Shopping Demo</title>
  <script type="module" src="./app.js"></script>
</head>
<body>
  <header>
    <h1>My Store</h1>
    <cart-badge></cart-badge>
  </header>
  <main>
    <product-catalog></product-catalog>
    <shopping-cart></shopping-cart>
  </main>
</body>
</html>

```

Notice how these components have zero direct dependencies on each other. The `product-catalog` doesn't know about `shopping-cart`. The `cart-badge` doesn't know about either. They're completely decoupled, yet they work together seamlessly through the PAN bus.

This is the power of message-based architecture: you can add, remove, or replace components without touching existing code. Want to add a “Cart Saved” notification? Just create a component that subscribes to `cart.updated`. Want to log analytics when items are added? Subscribe to `cart.item.added`. The existing components don't care.

7.5 Component Communication Patterns

Let's explore some common patterns for component communication in LARC.

7.5.1 Pattern: Request-Response

Sometimes a component needs data from another component or service. Use a request-response pattern:

```

class DataLoader extends HTMLElement {
  connectedCallback() {
    // Subscribe to data requests
    this.unsubscribe = subscribe('data.request', async (msg) => {
      const { requestId, url } = msg.data;

      try {
        const response = await fetch(url);
        const data = await response.json();

        // Publish response
        publish('data.response', {
          requestId,
          data,
          error: null
        });
      } catch (error) {
        // Publish error
        publish('data.response', {
          requestId,
          data: null,
          error: error.message
        });
      }
    });
  }

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

customElements.define('data-loader', DataLoader);

```

A component that needs data publishes a request:

```

class DataConsumer extends HTMLElement {
  connectedCallback() {
    const requestId = `request-${Date.now()}-${Math.random()}`;

    // Subscribe to the response
    this.unsubscribe = subscribe('data.response', (msg) => {
      if (msg.data.requestId === requestId) {
        if (msg.data.error) {
          this.showError(msg.data.error);
        } else {

```

```

        this.showData(msg.data.data);
    }

    // Unsubscribe after receiving response
    this.unsubscribe();
}
});

// Publish the request
publish('data.request', {
    requestId,
    url: '/api/data'
});
}
}

```

The `requestId` ensures that the requester only processes its own response, not responses to other requests.

7.5.2 Pattern: Command Pattern

Use commands to trigger actions without caring who handles them:

```

// Component that issues commands
class CommandIssuer extends HTMLElement {
    connectedCallback() {
        this.innerHTML = `
            <button id="save-btn">Save</button>
            <button id="cancel-btn">Cancel</button>
        `;

        this.querySelector('#save-btn').addEventListener('click', () => {
            publish('command.save', { timestamp: Date.now() });
        });

        this.querySelector('#cancel-btn').addEventListener('click', () => {
            publish('command.cancel', { timestamp: Date.now() });
        });
    }
}

// Component that handles commands
class CommandHandler extends HTMLElement {
    connectedCallback() {
        this.subscriptions = [
            subscribe('command.save', () => this.handleSave()),
            subscribe('command.cancel', () => this.handleCancel())
        ];
    }
}

```

```

}

handleSave() {
  console.log('Saving...');
  // Perform save operation
}

handleCancel() {
  console.log('Canceling...');
  // Perform cancel operation
}

disconnectedCallback() {
  this.subscriptions.forEach(unsub => unsub());
}
}

```

7.5.3 Pattern: State Projection

Components can subscribe to state changes and project that state into the UI:

```

class CurrentUser extends HTMLElement {
  connectedCallback() {
    this.unsubscribe = subscribe('user.current', (msg) => {
      this.render(msg.data);
    });

    // Trigger initial render with retained message
    this.render(null);
  }

  render(user) {
    if (user) {
      this.innerHTML = `
        <div class="current-user">
          
          <span>${user.name}</span>
        </div>
      `;
    } else {
      this.innerHTML = '<div class="current-user">Not logged in</div>';
    }
  }

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

```

```

    }
  }
}

```

This component is purely presentational—it projects state into UI without managing any state itself.

7.5.4 Pattern: Event Aggregation

Some components aggregate events from multiple sources:

```

class ActivityFeed extends HTMLElement {
  constructor() {
    super();
    this.activities = [];
  }

  connectedCallback() {
    // Subscribe to multiple event types
    this.unsubscribe = subscribe('*.success', (msg) => {
      this.addActivity({
        type: 'success',
        topic: msg.topic,
        data: msg.data,
        timestamp: Date.now()
      });
    });
  }

  this.render();
}

addActivity(activity) {
  this.activities.unshift(activity);

  // Keep only the most recent 20 activities
  if (this.activities.length > 20) {
    this.activities.pop();
  }

  this.render();
}

render() {
  this.innerHTML = `
    <div class="activity-feed">
      <h2>Recent Activity</h2>
      <ul>
        ${this.activities.map(activity => `
          <li>

```

```

        <span class="activity-time">${new Date(activity.timestamp).toLocaleTimeString()}
        <span class="activity-type">${activity.topic}</span>
      </li>
    `).join('')}
  </ul>
</div>
`;
}

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

customElements.define('activity-feed', ActivityFeed);

```

7.6 Reusable Component Design

Creating reusable components is an art. Here are principles to guide your design:

7.6.1 Principle 1: Single Responsibility

Each component should do one thing well. Don't create a `UserProfileWithEditorAndNotifications` component—create `UserProfile`, `UserEditor`, and `UserNotifications` components that work together.

7.6.2 Principle 2: Clear API

A component's API consists of:

1. **Attributes:** Configuration that rarely changes
2. **Published messages:** Events or state changes the component announces
3. **Subscribed messages:** Messages the component reacts to

Document all three:

```

/**
 * UserAvatar Component
 *
 * Displays a user's avatar image with optional fallback to initials.
 *
 * Attributes:
 *
 *   - user-id: Required. The ID of the user to display.
 *   - size: Optional. Size in pixels (default: 40).
 *
 * Subscribes to:

```

```

* - user.data: Updates avatar when user data changes.
*
* Publishes:

* - user.avatar.clicked: When the avatar is clicked.
*/
class UserAvatar extends HTMLElement {
  // Implementation...
}

```

7.6.3 Principle 3: Composition Over Configuration

Rather than making components configurable with dozens of attributes, make them composable:

```

<!-- Bad: too many configuration options -->
<data-table
  show-header="true"
  show-footer="true"
  enable-sorting="true"
  enable-filtering="true"
  enable-pagination="true"
></data-table>

<!-- Good: compose smaller components -->
<data-table>
  <table-header></table-header>
  <table-body></table-body>
  <table-footer></table-footer>
</data-table>

```

7.6.4 Principle 4: Progressive Enhancement

Design components to work without JavaScript when possible, and enhance them progressively:

```

class ProgressiveForm extends HTMLElement {
  connectedCallback() {
    // The form works without JS (regular form submission)
    const form = this.querySelector('form');

    // Enhance with AJAX submission if JS is available
    form.addEventListener('submit', async (e) => {
      e.preventDefault();

      const formData = new FormData(form);
      const response = await fetch(form.action, {
        method: form.method,
        body: formData
      });
    });
  }
}

```

```

    if (response.ok) {
      publish('form.submitted', { formId: form.id });
    }
  });
}
}

```

7.6.5 Principle 5: Accessibility First

Always consider keyboard navigation, screen readers, and ARIA attributes:

```

class AccessibleDialog extends HTMLElement {
  connectedCallback() {
    this.setAttribute('role', 'dialog');
    this.setAttribute('aria-modal', 'true');

    this.innerHTML = `
      <div class="dialog-overlay">
        <div class="dialog-content">
          <button class="dialog-close" aria-label="Close dialog">x</button>
          <slot></slot>
        </div>
      </div>
    `;

    // Close on Escape key
    this.addEventListener('keydown', (e) => {
      if (e.key === 'Escape') {
        this.close();
      }
    });

    // Trap focus within dialog
    this.trapFocus();
  }

  trapFocus() {
    const focusableElements = this.querySelectorAll(
      'button, [href], input, select, textarea, [tabindex]:not([tabindex="-1"])'
    );

    if (focusableElements.length === 0) return;

    const firstElement = focusableElements[0];
    const lastElement = focusableElements[focusableElements.length - 1];

    this.addEventListener('keydown', (e) => {

```

```

    if (e.key === 'Tab') {
      if (e.shiftKey) {
        if (document.activeElement === firstElement) {
          e.preventDefault();
          lastElement.focus();
        }
      } else {
        if (document.activeElement === lastElement) {
          e.preventDefault();
          firstElement.focus();
        }
      }
    }
  });

  firstElement.focus();
}

close() {
  publish('dialog.closed', { dialogId: this.id });
  this.remove();
}
}

customElements.define('accessible-dialog', AccessibleDialog);

```

7.7 Advanced Component Techniques

7.7.1 Technique: Lazy Rendering

For components that manage large datasets, render lazily:

```

class LazyList extends HTMLElement {
  constructor() {
    super();
    this.items = [];
    this.visibleCount = 20;
  }

  connectedCallback() {
    this.unsubscribe = subscribe('list.items', (msg) => {
      this.items = msg.data.items;
      this.render();
    });

    this.render();
  }
}

```

```

render() {
  const visibleItems = this.items.slice(0, this.visibleCount);

  this.innerHTML = `
    <div class="lazy-list">
      <ul>
        ${visibleItems.map(item => `
          <li>${item.name}</li>
        `).join('')}
      </ul>
      ${this.items.length > this.visibleCount ? `
        <button id="load-more">Load More</button>
      ` : ''}
    </div>
  `;

  const loadMoreBtn = this.querySelector('#load-more');
  if (loadMoreBtn) {
    loadMoreBtn.addEventListener('click', () => {
      this.visibleCount += 20;
      this.render();
    });
  }
}

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

customElements.define('lazy-list', LazyList);

```

7.7.2 Technique: Virtual Scrolling

For truly massive lists, implement virtual scrolling:

```

class VirtualList extends HTMLElement {
  constructor() {
    super();
    this.items = [];
    this.itemHeight = 50;
    this.visibleCount = 20;
    this.scrollTop = 0;
  }
}

```

```

connectedCallback() {
  this.unsubscribe = subscribe('list.items', (msg) => {
    this.items = msg.data.items;
    this.render();
  });

  this.render();
}

render() {
  const startIndex = Math.floor(this.scrollTop / this.itemHeight);
  const endIndex = Math.min(
    startIndex + this.visibleCount,
    this.items.length
  );

  const visibleItems = this.items.slice(startIndex, endIndex);
  const totalHeight = this.items.length * this.itemHeight;
  const offsetY = startIndex * this.itemHeight;

  this.innerHTML = `
    <div class="virtual-list" style="height: 400px; overflow-y: auto;">
      <div style="height: ${totalHeight}px; position: relative;">
        <div style="transform: translateY(${offsetY}px);">
          ${visibleItems.map(item => `
            <div style="height: ${this.itemHeight}px;">${item.name}</div>
          `).join('')}
        </div>
      </div>
    </div>
  `;

  this.querySelector('.virtual-list').addEventListener('scroll', (e) => {
    this.scrollTop = e.target.scrollTop;
    this.render();
  });
}

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

customElements.define('virtual-list', VirtualList);

```

7.7.3 Technique: Memoization

Avoid re-rendering when nothing has changed:

```
class MemoizedComponent extends HTMLElement {
  constructor() {
    super();
    this.lastData = null;
  }

  connectedCallback() {
    this.unsubscribe = subscribe('data.updated', (msg) => {
      // Only re-render if data actually changed
      if (JSON.stringify(msg.data) !== JSON.stringify(this.lastData)) {
        this.lastData = msg.data;
        this.render();
      }
    });

    this.render();
  }

  render() {
    // Expensive rendering logic...
  }

  disconnectedCallback() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }
}
```

7.8 Testing Components

Components built with web standards are easy to test. Here's a simple test using a standard test framework:

```
import { expect } from 'chai';
import { publish, subscribe } from '@larc/core';
import './shopping-cart.js';

describe('ShoppingCart', () => {
  let cart;

  beforeEach(() => {
    cart = document.createElement('shopping-cart');
    document.body.appendChild(cart);
  });
```

```
afterEach(() => {
  cart.remove();
});

it('starts empty', () => {
  expect(cart.items).toHaveLength(0);
});

it('adds items when cart.item.added is published', (done) => {
  subscribe('cart.updated', (msg) => {
    expect(msg.data.items).toHaveLength(1);
    expect(msg.data.items[0].name).toEqual('Widget');
    done();
  });

  publish('cart.item.added', {
    productId: 1,
    name: 'Widget',
    price: 10,
    quantity: 1
  });
});

it('calculates total correctly', (done) => {
  subscribe('cart.updated', (msg) => {
    expect(msg.data.total).toEqual(30);
    done();
  });

  publish('cart.item.added', {
    productId: 1,
    name: 'Widget',
    price: 10,
    quantity: 3
  });
});
});
```

Because components communicate through messages, testing is straightforward: publish messages, subscribe to responses, and assert the results.

7.9 Wrapping Up

You've now mastered the art of building components in LARC. You understand the component lifecycle, when to use Shadow DOM, how to connect components via the PAN bus, and how to design reusable, composable components that stand the test of time.

The key insight is this: components in LARC are independent, loosely-coupled modules that communicate through messages. They don't know about each other, don't depend on each other, and can be added, removed, or replaced without touching existing code. This architecture scales beautifully from tiny prototypes to massive applications.

In the next chapter, we'll tackle state management—one of the thorniest problems in modern web development. You'll learn how to manage local and shared state, persist data to IndexedDB and OPFS, synchronize state across components, and handle conflicts gracefully. Get ready—state management is where LARC's architecture truly shines.

But first, take a break. Build a few components. Connect them through the PAN bus. Watch them interact. The best way to internalize these patterns is to use them. And when you inevitably build a component that's too big, too complex, or too tightly coupled, you'll feel the pain firsthand—and you'll understand why the principles in this chapter matter.

See you in Chapter 8.

Chapter 8

State Management

“There are only two hard things in Computer Science: cache invalidation, naming things, and state management.”

— Phil Karlton (updated for modern web development)

State management is the art of keeping track of what’s true about your application right now. Which user is logged in? What items are in the shopping cart? Is the modal open or closed? Has the data been saved or is it still dirty?

Get state management right, and your application feels solid, predictable, and reliable. Get it wrong, and you’ll spend your days hunting down race conditions, stale data, and mysterious bugs that only reproduce on Tuesdays when Mercury is in retrograde.

In this chapter, we’ll explore how LARC approaches state management. You’ll learn the difference between local and shared state, strategies for persisting state to IndexedDB and OPFS (Origin Private File System), patterns for synchronizing state across components, and techniques for resolving conflicts when multiple sources of truth collide.

Fair warning: this chapter is dense. State management is hard, and anyone who tells you otherwise is selling something. But LARC’s message-based architecture provides a solid foundation for tackling this complexity. By the end of this chapter, you’ll have the tools to build applications that manage state gracefully, even under adverse conditions.

8.1 Local vs. Shared State

The first decision in state management is: where does this state live?

Local state belongs to a single component. It’s not shared, not synchronized, and not persisted. Examples include:

- Whether a dropdown is expanded
- The current input value in a form field
- The selected tab in a tab panel
- Animation state

Local state is simple. Store it in component properties:

```

class DropdownMenu extends HTMLElement {
  constructor() {
    super();
    this.isOpen = false; // Local state
  }

  connectedCallback() {
    this.render();
  }

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

  render() {
    this.innerHTML = `
      <div class="dropdown">
        <button id="toggle-btn">${this.isOpen ? 'Close' : 'Open'}</button>
        ${this.isOpen ? `
          <ul class="dropdown-menu">
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        ` : ''}
      </div>
    `;

    this.querySelector('#toggle-btn').addEventListener('click', () => {
      this.toggle();
    });
  }
}

customElements.define('dropdown-menu', DropdownMenu);

```

Local state requires no persistence, no synchronization, and no messaging. When the component is destroyed, the state disappears. This is fine—ephemeral state should be ephemeral.

Shared state is accessed by multiple components. Examples include:

- The current authenticated user
- Items in a shopping cart
- Application theme (light/dark mode)
- Cached API responses

Shared state lives outside individual components and flows through the PAN bus. Components subscribe to state changes and publish updates.

8.2 The State Store Pattern

For shared state, LARC applications typically use a “state store” component—a component whose sole job is to manage a piece of shared state.

Here’s a minimal example:

```
class UserStore extends HTMLElement {
  constructor() {
    super();
    this.currentUser = null;
  }

  connectedCallback() {
    // Subscribe to login events
    this.subscriptions = [
      subscribe('auth.login.success', (msg) => {
        this.setUser(msg.data);
      }),

      subscribe('auth.logout', () => {
        this.setUser(null);
      }),

      subscribe('user.profile.updated', (msg) => {
        if (this.currentUser && msg.data.userId === this.currentUser.userId) {
          this.setUser({ ...this.currentUser, ...msg.data });
        }
      })
    ];

    // Load persisted user from localStorage
    this.loadPersistedUser();
  }

  setUser(user) {
    this.currentUser = user;

    // Publish updated state
    publish('user.current', user);

    // Persist to localStorage
    if (user) {
      localStorage.setItem('currentUser', JSON.stringify(user));
    } else {
      localStorage.removeItem('currentUser');
    }
  }
}
```

```

loadPersistedUser() {
  const stored = localStorage.getItem('currentUser');
  if (stored) {
    try {
      const user = JSON.parse(stored);
      this.setUser(user);
    } catch (error) {
      console.error('Failed to load persisted user:', error);
    }
  }
}

disconnectedCallback() {
  this.subscriptions.forEach(unsub => unsub());
}

customElements.define('user-store', UserStore);

```

This store:

1. Listens for events that change user state
2. Updates its internal state
3. Publishes the new state to `user.current`
4. Persists the state to `localStorage`

Other components simply subscribe to `user.current`:

```

class UserGreeting extends HTMLElement {
  connectedCallback() {
    this.unsubscribe = subscribe('user.current', (msg) => {
      this.render(msg.data);
    });
  }

  render(user) {
    if (user) {
      this.textContent = `Hello, ${user.username}!`;
    } else {
      this.textContent = 'Please log in.';
    }
  }

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

```

```
customElements.define('user-greeting', UserGreeting);
```

Notice the separation of concerns: `UserStore` manages state, `UserGreeting` displays it. Neither component knows about the other.

8.3 State Persistence with localStorage

For simple persistence, `localStorage` is hard to beat. It's synchronous, widely supported, and requires no setup.

```
class SettingsStore extends HTMLElement {
  constructor() {
    super();
    this.settings = this.loadSettings();
  }

  connectedCallback() {
    this.unsubscribe = subscribe('settings.update', (msg) => {
      this.updateSettings(msg.data);
    });

    // Publish initial state
    publish('settings.current', this.settings);
  }

  loadSettings() {
    const stored = localStorage.getItem('settings');
    const defaults = {
      theme: 'light',
      fontSize: 16,
      notifications: true
    };

    if (stored) {
      try {
        return { ...defaults, ...JSON.parse(stored) };
      } catch (error) {
        console.error('Failed to load settings:', error);
        return defaults;
      }
    }

    return defaults;
  }

  updateSettings(updates) {
    this.settings = { ...this.settings, ...updates };
  }
}
```

```

    // Persist to localStorage
    localStorage.setItem('settings', JSON.stringify(this.settings));

    // Publish updated state
    publish('settings.current', this.settings);
  }

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

customElements.define('settings-store', SettingsStore);

```

8.3.1 localStorage Limitations

localStorage is convenient but has limitations:

1. **Size limit:** Typically 5-10 MB per origin
2. **Synchronous API:** Blocks the main thread (though usually fast)
3. **String-only storage:** Must serialize/deserialize data
4. **No structured queries:** You can't query localStorage like a database

For larger datasets or structured data, use IndexedDB.

8.4 State Persistence with IndexedDB

IndexedDB is a powerful, asynchronous, transactional database built into browsers. It can store much larger amounts of data than localStorage (often hundreds of megabytes or more) and supports structured queries.

However, IndexedDB's API is notoriously verbose. Here's a wrapper to make it more palatable:

```

class IndexedDBStore {
  constructor(dbName, storeName) {
    this.dbName = dbName;
    this.storeName = storeName;
    this.db = null;
  }

  async open() {
    return new Promise((resolve, reject) => {
      const request = indexedDB.open(this.dbName, 1);

      request.onerror = () => reject(request.error);
      request.onsuccess = () => {

```

```
    this.db = request.result;
    resolve(this.db);
  };

  request.onupgradeneeded = (event) => {
    const db = event.target.result;
    if (!db.objectStoreNames.contains(this.storeName)) {
      db.createObjectStore(this.storeName, { keyPath: 'id' });
    }
  };
});
}

async get(id) {
  if (!this.db) await this.open();

  return new Promise((resolve, reject) => {
    const transaction = this.db.transaction([this.storeName], 'readonly');
    const store = transaction.objectStore(this.storeName);
    const request = store.get(id);

    request.onerror = () => reject(request.error);
    request.onsuccess = () => resolve(request.result);
  });
}

async put(object) {
  if (!this.db) await this.open();

  return new Promise((resolve, reject) => {
    const transaction = this.db.transaction([this.storeName], 'readwrite');
    const store = transaction.objectStore(this.storeName);
    const request = store.put(object);

    request.onerror = () => reject(request.error);
    request.onsuccess = () => resolve(request.result);
  });
}

async delete(id) {
  if (!this.db) await this.open();

  return new Promise((resolve, reject) => {
    const transaction = this.db.transaction([this.storeName], 'readwrite');
    const store = transaction.objectStore(this.storeName);
    const request = store.delete(id);
```

```

        request.onerror = () => reject(request.error);
        request.onsuccess = () => resolve();
    });
}

async getAll() {
    if (!this.db) await this.open();

    return new Promise((resolve, reject) => {
        const transaction = this.db.transaction([this.storeName], 'readonly');
        const store = transaction.objectStore(this.storeName);
        const request = store.getAll();

        request.onerror = () => reject(request.error);
        request.onsuccess = () => resolve(request.result);
    });
}
}

```

Now use it in a store component:

```

class DocumentStore extends HTMLElement {
    constructor() {
        super();
        this.db = new IndexedDBStore('app-db', 'documents');
        this.documents = [];
    }

    async connectedCallback() {
        this.subscriptions = [
            subscribe('document.save', async (msg) => {
                await this.saveDocument(msg.data);
            }),

            subscribe('document.delete', async (msg) => {
                await this.deleteDocument(msg.data.id);
            }),

            subscribe('document.load', async (msg) => {
                await this.loadDocument(msg.data.id);
            })
        ];

        // Load all documents on startup
        await this.loadAllDocuments();
    }

    async loadAllDocuments() {

```

```
    try {
      this.documents = await this.db.getAll();
      publish('documents.loaded', { documents: this.documents });
    } catch (error) {
      console.error('Failed to load documents:', error);
      publish('documents.error', { error: error.message });
    }
  }

  async saveDocument(document) {
    try {
      await this.db.put(document);
      this.documents = await this.db.getAll();
      publish('document.saved', { document });
      publish('documents.loaded', { documents: this.documents });
    } catch (error) {
      console.error('Failed to save document:', error);
      publish('document.error', { error: error.message });
    }
  }

  async deleteDocument(id) {
    try {
      await this.db.delete(id);
      this.documents = await this.db.getAll();
      publish('document.deleted', { id });
      publish('documents.loaded', { documents: this.documents });
    } catch (error) {
      console.error('Failed to delete document:', error);
      publish('document.error', { error: error.message });
    }
  }

  async loadDocument(id) {
    try {
      const document = await this.db.get(id);
      publish('document.loaded', { document });
    } catch (error) {
      console.error('Failed to load document:', error);
      publish('document.error', { error: error.message });
    }
  }

  disconnectedCallback() {
    this.subscriptions.forEach(unsub => unsub());
  }
}
```

```
customElements.define('document-store', DocumentStore);
```

This store persists documents to IndexedDB and publishes events when documents are saved, deleted, or loaded. Other components react to these events without knowing anything about IndexedDB.

8.5 State Persistence with OPFS

The Origin Private File System (OPFS) is a newer browser API that provides high-performance file storage. Unlike IndexedDB, which is designed for structured data, OPFS is designed for files—making it ideal for large binary data like images, videos, or application data files.

Here's how to use OPFS:

```
class OPFSStore {
  constructor() {
    this.root = null;
  }

  async init() {
    if (!this.root) {
      this.root = await navigator.storage.getDirectory();
    }
  }

  async writeFile(path, data) {
    await this.init();

    const fileHandle = await this.root.getFileHandle(path, { create: true });
    const writable = await fileHandle.createWritable();
    await writable.write(data);
    await writable.close();
  }

  async readFile(path) {
    await this.init();

    try {
      const fileHandle = await this.root.getFileHandle(path);
      const file = await fileHandle.getFile();
      return await file.text();
    } catch (error) {
      if (error.name === 'NotFoundError') {
        return null;
      }
      throw error;
    }
  }
}
```

```

async deleteFile(path) {
  await this.init();

  try {
    await this.root.removeEntry(path);
  } catch (error) {
    if (error.name !== 'NotFoundError') {
      throw error;
    }
  }
}

async listFiles() {
  await this.init();

  const files = [];
  for await (const entry of this.root.values()) {
    if (entry.kind === 'file') {
      files.push(entry.name);
    }
  }
  return files;
}
}

```

Use OPFS for storing large files:

```

class FileStore extends HTMLElement {
  constructor() {
    super();
    this.opfs = new OPFSStore();
  }

  async connectedCallback() {
    this.subscriptions = [
      subscribe('file.save', async (msg) => {
        await this.saveFile(msg.data);
      }),

      subscribe('file.load', async (msg) => {
        await this.loadFile(msg.data.path);
      }),

      subscribe('file.delete', async (msg) => {
        await this.deleteFile(msg.data.path);
      })
    ];
  }
}

```

```
// Publish list of available files
const files = await this.opfs.listFiles();
publish('files.list', { files });
}

async saveFile({ path, content }) {
  try {
    await this.opfs.writeFile(path, content);
    publish('file.saved', { path });

    const files = await this.opfs.listFiles();
    publish('files.list', { files });
  } catch (error) {
    console.error('Failed to save file:', error);
    publish('file.error', { error: error.message });
  }
}

async loadFile(path) {
  try {
    const content = await this.opfs.readFile(path);
    publish('file.loaded', { path, content });
  } catch (error) {
    console.error('Failed to load file:', error);
    publish('file.error', { error: error.message });
  }
}

async deleteFile(path) {
  try {
    await this.opfs.deleteFile(path);
    publish('file.deleted', { path });

    const files = await this.opfs.listFiles();
    publish('files.list', { files });
  } catch (error) {
    console.error('Failed to delete file:', error);
    publish('file.error', { error: error.message });
  }
}

disconnectedCallback() {
  this.subscriptions.forEach(unsub => unsub());
}

customElements.define('file-store', FileStore);
```

8.5.1 When to Use OPFS vs. IndexedDB

Use **IndexedDB** when:

- You need structured data with queries
- You need transactions
- Data is primarily JSON or small blobs

Use **OPFS** when:

- You're working with large files (>1 MB)
- You need high-performance sequential access
- You're building a file-based application (e.g., document editor, media player)

Use **localStorage** when:

- Data is small (<100 KB)
- Simplicity matters more than performance
- You need synchronous access

8.6 Synchronization Patterns

When multiple components interact with shared state, synchronization becomes critical. Here are common patterns:

8.6.1 Pattern: Optimistic Updates

Update the UI immediately, then sync with the server in the background:

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

  connectedCallback() {
    this.subscriptions = [
      subscribe('todo.add', async (msg) => {
        await this.addToTodo(msg.data);
      }),

      subscribe('todo.complete', async (msg) => {
        await this.completeTodo(msg.data.id);
      })
    ];

    this.loadTodos();
  }

  async loadTodos() {
    try {
```

```

    const response = await fetch('/api/todos');
    this.todos = await response.json();
    publish('todos.loaded', { todos: this.todos });
  } catch (error) {
    console.error('Failed to load todos:', error);
  }
}

async addTodo(todo) {
  // Optimistic update: add to local state immediately
  const optimisticTodo = { id: `temp-${Date.now()}`, ...todo };
  this.todos.push(optimisticTodo);
  publish('todos.loaded', { todos: this.todos });

  try {
    // Sync with server
    const response = await fetch('/api/todos', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(todo)
    });

    const savedTodo = await response.json();

    // Replace optimistic todo with server response
    this.todos = this.todos.map(t =>
      t.id === optimisticTodo.id ? savedTodo : t
    );

    publish('todos.loaded', { todos: this.todos });
    publish('todo.synced', { todo: savedTodo });
  } catch (error) {
    // Rollback on error
    this.todos = this.todos.filter(t => t.id !== optimisticTodo.id);
    publish('todos.loaded', { todos: this.todos });
    publish('todo.error', { error: error.message });
  }
}

async completeTodo(id) {
  // Optimistic update: mark complete immediately
  const originalTodos = [...this.todos];
  this.todos = this.todos.map(t =>
    t.id === id ? { ...t, completed: true } : t
  );
  publish('todos.loaded', { todos: this.todos });
}

```

```

    try {
      // Sync with server
      await fetch(`/api/todos/${id}`, {
        method: 'PATCH',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ completed: true })
      });

      publish('todo.synced', { id });
    } catch (error) {
      // Rollback on error
      this.todos = originalTodos;
      publish('todos.loaded', { todos: this.todos });
      publish('todo.error', { error: error.message });
    }
  }
}

disconnectedCallback() {
  this.subscriptions.forEach(unsub => unsub());
}
}

customElements.define('todo-store', TodoStore);

```

Optimistic updates make the UI feel instant while handling network latency gracefully.

8.6.2 Pattern: Debounced Sync

For high-frequency updates, debounce synchronization to reduce server load:

```

class EditorStore extends HTMLElement {
  constructor() {
    super();
    this.content = '';
    this.syncTimer = null;
    this.syncDelay = 1000; // 1 second
  }

  connectedCallback() {
    this.unsubscribe = subscribe('editor.content.changed', (msg) => {
      this.updateContent(msg.data.content);
    });

    this.loadContent();
  }

  async loadContent() {
    try {

```

```
const response = await fetch('/api/document/current');
const data = await response.json();
this.content = data.content;
publish('editor.content.loaded', { content: this.content });
} catch (error) {
  console.error('Failed to load content:', error);
}
}

updateContent(content) {
  this.content = content;

  // Publish immediately for reactive UI
  publish('editor.content.updated', { content });

  // Debounce server sync
  clearTimeout(this.syncTimer);
  this.syncTimer = setTimeout(() => {
    this.syncToServer();
  }, this.syncDelay);
}

async syncToServer() {
  try {
    await fetch('/api/document/current', {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ content: this.content })
    });

    publish('editor.content.synced', { timestamp: Date.now() });
  } catch (error) {
    console.error('Failed to sync content:', error);
    publish('editor.sync.error', { error: error.message });
  }
}

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

  // Flush pending sync on disconnect
  clearTimeout(this.syncTimer);
  this.syncToServer();
}
}
```

```
customElements.define('editor-store', EditorStore);
```

8.6.3 Pattern: Polling

For real-time-ish updates without WebSockets, poll the server periodically:

```
class NotificationStore extends HTMLElement {
  constructor() {
    super();
    this.notifications = [];
    this.pollInterval = 30000; // 30 seconds
    this.pollTimer = null;
  }

  connectedCallback() {
    this.startPolling();
  }

  startPolling() {
    this.fetchNotifications();

    this.pollTimer = setInterval(() => {
      this.fetchNotifications();
    }, this.pollInterval);
  }

  async fetchNotifications() {
    try {
      const response = await fetch('/api/notifications');
      const notifications = await response.json();

      // Check for new notifications
      const newNotifications = notifications.filter(n =>
        !this.notifications.some(existing => existing.id === n.id)
      );

      if (newNotifications.length > 0) {
        publish('notifications.new', { notifications: newNotifications });
      }

      this.notifications = notifications;
      publish('notifications.updated', { notifications });
    } catch (error) {
      console.error('Failed to fetch notifications:', error);
    }
  }
}
```

```

    disconnectedCallback() {
      if (this.pollTimer) {
        clearInterval(this.pollTimer);
      }
    }
  }
}

customElements.define('notification-store', NotificationStore);

```

8.7 Conflict Resolution

When multiple sources can update the same state, conflicts arise. Here are strategies for resolving them:

8.7.1 Strategy: Last Write Wins

The simplest strategy: the most recent write wins, earlier writes are lost:

```

class SimpleStore extends HTMLElement {
  constructor() {
    super();
    this.data = {};
  }

  connectedCallback() {
    this.unsubscribe = subscribe('data.update', (msg) => {
      // Last write wins
      this.data = { ...this.data, ...msg.data };
      publish('data.current', this.data);
    });
  }
}

```

This works when conflicts are rare or unimportant.

8.7.2 Strategy: Timestamps

Use timestamps to determine which update is newer:

```

class TimestampedStore extends HTMLElement {
  constructor() {
    super();
    this.data = {};
    this.timestamps = {};
  }

  connectedCallback() {
    this.unsubscribe = subscribe('data.update', (msg) => {
      const { key, value, timestamp } = msg.data;

```

```

    // Only apply update if it's newer
    if (!this.timestamps[key] || timestamp > this.timestamps[key]) {
      this.data[key] = value;
      this.timestamps[key] = timestamp;
      publish('data.current', this.data);
    }
  });
}
}

```

This handles out-of-order updates gracefully.

8.7.3 Strategy: Version Vectors

For distributed systems, use version vectors to track causality:

```

class VersionedStore extends HTMLElement {
  constructor() {
    super();
    this.data = {};
    this.version = {}; // { clientId: sequence }
  }

  connectedCallback() {
    this.unsubscribe = subscribe('data.update', (msg) => {
      const { key, value, version } = msg.data;

      if (this.isNewer(version)) {
        this.data[key] = value;
        this.version = this.mergeVersions(this.version, version);
        publish('data.current', { data: this.data, version: this.version });
      }
    });
  }

  isNewer(incomingVersion) {
    // Check if incoming version is causally newer
    for (const clientId in incomingVersion) {
      if (incomingVersion[clientId] > (this.version[clientId] || 0)) {
        return true;
      }
    }
    return false;
  }

  mergeVersions(v1, v2) {
    const merged = { ...v1 };
    for (const clientId in v2) {

```

```

    merged[clientId] = Math.max(merged[clientId] || 0, v2[clientId]);
  }
  return merged;
}
}

```

This is overkill for most applications, but essential for offline-first or collaborative apps.

8.7.4 Strategy: Conflict Detection and User Intervention

When conflicts matter, detect them and let the user decide:

```

class ConflictAwareStore extends HTMLElement {
  constructor() {
    super();
    this.data = {};
    this.version = 0;
  }

  connectedCallback() {
    this.unsubscribe = subscribe('data.update', (msg) => {
      const { key, value, expectedVersion } = msg.data;

      if (expectedVersion !== this.version) {
        // Conflict detected
        publish('data.conflict', {
          key,
          currentValue: this.data[key],
          incomingValue: value,
          currentVersion: this.version,
          expectedVersion
        });
      } else {
        // No conflict, apply update
        this.data[key] = value;
        this.version++;
        publish('data.current', { data: this.data, version: this.version });
      }
    });
  }
}

```

A UI component can subscribe to `data.conflict` and show a dialog asking the user which value to keep.

8.8 State Snapshots and Time Travel

For debugging and undo/redox functionality, maintain a history of state snapshots:

```
class HistoryStore extends HTMLElement {
  constructor() {
    super();
    this.history = [];
    this.currentIndex = -1;
    this.maxHistory = 50;
  }

  connectedCallback() {
    this.subscriptions = [
      subscribe('state.update', (msg) => {
        this.addSnapshot(msg.data);
      }),

      subscribe('state.undo', () => {
        this.undo();
      }),

      subscribe('state.redo', () => {
        this.redo();
      })
    ];
  }

  addSnapshot(state) {
    // Remove any history after current index (user made changes after undo)
    this.history = this.history.slice(0, this.currentIndex + 1);

    // Add new snapshot
    this.history.push(JSON.parse(JSON.stringify(state)));
    this.currentIndex++;

    // Limit history size
    if (this.history.length > this.maxHistory) {
      this.history.shift();
      this.currentIndex--;
    }

    publish('state.current', state);
    publish('state.history.updated', {
      canUndo: this.canUndo(),
      canRedo: this.canRedo()
    });
  }

  undo() {
    if (this.canUndo()) {

```

```

        this.currentIndex--;
        const state = this.history[this.currentIndex];
        publish('state.current', state);
        publish('state.history.updated', {
            canUndo: this.canUndo(),
            canRedo: this.canRedo()
        });
    }
}

redo() {
    if (this.canRedo()) {
        this.currentIndex++;
        const state = this.history[this.currentIndex];
        publish('state.current', state);
        publish('state.history.updated', {
            canUndo: this.canUndo(),
            canRedo: this.canRedo()
        });
    }
}

canUndo() {
    return this.currentIndex > 0;
}

canRedo() {
    return this.currentIndex < this.history.length - 1;
}

disconnectedCallback() {
    this.subscriptions.forEach(unsub => unsub());
}
}

customElements.define('history-store', HistoryStore);

```

8.9 Derived State

Sometimes state is computed from other state. Rather than storing derived state redundantly, compute it on demand:

```

class CartStore extends HTMLElement {
    constructor() {
        super();
        this.items = [];
    }
}

```

```
connectedCallback() {
  this.unsubscribe = subscribe('cart.item.added', (msg) => {
    this.items.push(msg.data);
    this.publishDerivedState();
  });
}

publishDerivedState() {
  const itemCount = this.items.reduce((sum, item) => sum + item.quantity, 0);
  const subtotal = this.items.reduce((sum, item) => sum + (item.price * item.quantity), 0);
  const tax = subtotal * 0.08;
  const total = subtotal + tax;

  publish('cart.state', {
    items: this.items,
    itemCount,
    subtotal,
    tax,
    total
  });
}

disconnectedCallback() {
  if (this.unsubscribe) {
    this.unsubscribe();
  }
}
}
```

Components receive the fully computed state and don't need to recalculate it.

8.10 Performance Considerations

State management can be expensive. Here are tips for keeping it performant:

1. **Minimize state updates:** Only publish when state actually changes
2. **Batch updates:** If updating multiple fields, batch them into a single message
3. **Use immutable updates:** Create new objects rather than mutating existing ones
4. **Debounce high-frequency updates:** Don't publish on every keystroke
5. **Lazy load large datasets:** Load data on demand rather than upfront
6. **Prune old data:** Remove stale data from stores to prevent memory bloat

8.11 Wrapping Up

State management is hard, but LARC's message-based architecture provides a solid foundation. By separating state stores from UI components, using the PAN bus for state synchronization,

and choosing the right persistence strategy (localStorage, IndexedDB, or OPFS), you can build applications that manage state gracefully even under complex conditions.

The key insights:

- Local state lives in components; shared state lives in stores
- Stores subscribe to commands and publish state updates
- Components subscribe to state updates and render accordingly
- Persistence strategies vary by data size and access patterns
- Conflicts are inevitable; plan your resolution strategy
- Derived state should be computed, not stored

In the next chapter, we'll explore advanced topics like routing, code splitting, and progressive enhancement. But state management is the foundation—get this right, and everything else becomes easier.

Now go forth and manage some state. And when you inevitably encounter a conflict on a Tuesday when Mercury is in retrograde, you'll know exactly what to do.

Chapter 9

Routing and Navigation

In which we learn to guide users through our applications without getting lost in the woods (or the browser's back button)

Navigation is to web applications what hallways are to buildings: theoretically simple, but surprisingly easy to get wrong. You've probably experienced the horror of clicking the back button only to be ejected from the application entirely, or the confusion of bookmarking a URL that leads nowhere meaningful. LARC's routing system aims to prevent these digital disasters by making client-side navigation feel as natural as walking through a well-designed building.

In this chapter, we'll explore LARC's routing architecture, which leverages the **pan-routes** component to create seamless navigation experiences. We'll cover route definitions, pattern matching, navigation guards, deep linking, history management, and even touch on SEO considerations because, let's face it, even the most beautiful application is useless if no one can find it.

9.1 Understanding Client-Side Routing

Before we dive into LARC's implementation, let's establish what client-side routing actually means. In the ancient days of the web (circa 2005), every navigation triggered a full page reload. Click a link, wait for the server, watch the screen flash white, and finally see your new content. It was like rebooting your computer every time you wanted to switch applications.

Client-side routing changes this paradigm. Instead of requesting new HTML from the server for each navigation, the JavaScript application intercepts link clicks, updates the URL, and renders the appropriate component—all without reloading the page. It's like having a building where rooms can instantly rearrange themselves rather than making you walk outside and back in through a different door.

LARC implements client-side routing through the **pan-routes** component, which acts as a traffic controller for your application's navigation. It watches for URL changes, matches them against defined route patterns, and renders the corresponding components.

9.2 The pan-routes Component

The `pan-routes` component is your application's navigation hub. It sits in your main application component and declares all the routes your application recognizes. Here's a basic example:

```
<pan-app id="app">
  <pan-routes>
    <pan-route path="/" component="home-view"></pan-route>
    <pan-route path="/about" component="about-view"></pan-route>
    <pan-route path="/products" component="product-list"></pan-route>
    <pan-route path="/products/:id" component="product-detail"></pan-route>
    <pan-route path="/user/:username" component="user-profile"></pan-route>
    <pan-route path="*" component="not-found-view"></pan-route>
  </pan-routes>
</pan-app>
```

Each `pan-route` element defines a mapping between a URL path and a component. When the URL matches a route's path, LARC renders the corresponding component. Think of it as a telephone switchboard operator from the 1950s, connecting callers to the right extension—except digital and without the period-appropriate hairstyle.

9.2.1 Route Matching Order

Routes are evaluated in the order they're defined, which means specificity matters. The wildcard route (`path="*"`) should always come last, as it matches everything. If you put it first, your users will only ever see your 404 page, which is a bold design choice but probably not what you intended.

Here's a more realistic example showing route organization:

```
<pan-routes>
  <!-- Exact matches first -->
  <pan-route path="/" component="home-view"></pan-route>
  <pan-route path="/login" component="login-view"></pan-route>
  <pan-route path="/logout" component="logout-view"></pan-route>

  <!-- Static paths before dynamic ones -->
  <pan-route path="/products/new" component="product-create"></pan-route>
  <pan-route path="/products/:id" component="product-detail"></pan-route>

  <!-- More specific patterns before general ones -->
  <pan-route path="/admin/users/:id" component="admin-user-detail"></pan-route>
  <pan-route path="/admin/:section" component="admin-section"></pan-route>

  <!-- Catch-all last -->
  <pan-route path="*" component="not-found-view"></pan-route>
</pan-routes>
```

9.3 Route Parameters and Pattern Matching

Dynamic route parameters are where routing gets interesting. Instead of defining a separate route for every product, user, or blog post, you use parameter placeholders prefixed with a colon (`:parameter`). LARC extracts these values and makes them available to your components.

9.3.1 Basic Parameters

The most common pattern is a single dynamic segment:

```
<pan-route path="/products/:id" component="product-detail"></pan-route>
<pan-route path="/users/:username" component="user-profile"></pan-route>
<pan-route path="/posts/:year/:month/:slug" component="blog-post"></pan-route>
```

In your component, access these parameters through the route context:

```
class ProductDetail extends LarcComponent {
  constructor() {
    super();
    this.product = null;
  }

  onRoute(params) {
    // params.id contains the value from the URL
    this.loadProduct(params.id);
  }

  async loadProduct(id) {
    const response = await fetch(`/api/products/${id}`);
    this.product = await response.json();
    this.render();
  }

  template() {
    if (!this.product) {
      return '<div>Loading...</div>';
    }

    return `
      <div class="product-detail">
        <h1>${this.product.name}</h1>
        <p>${this.product.description}</p>
        <span class="price">${this.product.price}</span>
      </div>
    `;
  }
}
```

9.3.2 Multiple Parameters

Routes can contain multiple parameters, which is useful for hierarchical data:

```
<pan-route path="/store/:category/:subcategory/:productId"
           component="product-view"></pan-route>
```

```
class ProductView extends LarcComponent {
  onRoute(params) {
    // params = { category: 'electronics', subcategory: 'phones', productId: '123' }
    this.loadProduct(params.category, params.subcategory, params.productId);
  }
}
```

9.3.3 Optional Parameters

Sometimes you want a route to work with or without certain parameters. While LARC doesn't have built-in optional parameter syntax, you can achieve this with multiple route definitions:

```
<pan-routes>
  <pan-route path="/blog/:year/:month/:day" component="blog-archive"></pan-route>
  <pan-route path="/blog/:year/:month" component="blog-archive"></pan-route>
  <pan-route path="/blog/:year" component="blog-archive"></pan-route>
  <pan-route path="/blog" component="blog-archive"></pan-route>
</pan-routes>
```

```
class BlogArchive extends LarcComponent {
  onRoute(params) {
    const { year, month, day } = params;

    if (day) {
      this.loadPostsForDay(year, month, day);
    } else if (month) {
      this.loadPostsForMonth(year, month);
    } else if (year) {
      this.loadPostsForYear(year);
    } else {
      this.loadAllPosts();
    }
  }
}
```

9.4 Programmatic Navigation

Clicking links is great, but sometimes you need to navigate programmatically—after form submissions, authentication changes, or when playing a game of “redirect the user until they give up and close the tab.”

LARC provides the `navigate()` function for programmatic navigation:

```

import { navigate } from '@larc/core';

class LoginForm extends LarcComponent {
  async handleLogin(event) {
    event.preventDefault();

    const formData = new FormData(event.target);
    const credentials = {
      username: formData.get('username'),
      password: formData.get('password')
    };

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

      if (response.ok) {
        // Success! Navigate to dashboard
        navigate('/dashboard');
      } else {
        this.showError('Invalid credentials');
      }
    } catch (error) {
      this.showError('Network error');
    }
  }

  template() {
    return `
      <form onsubmit="this.handleLogin(event)">
        <input name="username" type="text" required>
        <input name="password" type="password" required>
        <button type="submit">Login</button>
      </form>
    `;
  }
}

```

9.4.1 Navigation Options

The `navigate()` function accepts an options object for controlling navigation behavior:

```

// Replace current history entry instead of pushing a new one
navigate('/login', { replace: true });

```

```
// Prevent navigation if user has unsaved changes
if (this.hasUnsavedChanges()) {
  const confirmed = confirm('You have unsaved changes. Leave anyway?');
  if (!confirmed) {
    return; // Don't navigate
  }
}
navigate('/other-page');

// Navigate back and forward
navigate(-1); // Go back
navigate(1);  // Go forward
navigate(-2); // Go back two pages
```

9.5 Navigation Guards

Navigation guards are like bouncers at an exclusive club—they decide who gets in and who gets redirected to the login page. Guards let you intercept navigation attempts and redirect, cancel, or allow them based on application state.

9.5.1 Implementing Auth Guards

A common use case is protecting routes that require authentication:

```
class AuthGuard {
  constructor() {
    this.user = null;
    this.loadUserFromStorage();
  }

  loadUserFromStorage() {
    const stored = localStorage.getItem('user');
    if (stored) {
      this.user = JSON.parse(stored);
    }
  }

  canActivate(route) {
    if (!this.user) {
      // Not logged in, redirect to login
      navigate('/login?redirect=' + encodeURIComponent(route.path));
      return false;
    }
    return true;
  }

  requiresRole(role) {
```

```

    return this.user && this.user.roles.includes(role);
  }
}

const authGuard = new AuthGuard();

```

Now integrate this guard into your components:

```

class DashboardView extends LarcComponent {
  beforeRoute(params) {
    if (!authGuard.canActivate(this.route)) {
      return false; // Cancel navigation
    }
    return true; // Allow navigation
  }

  onRoute(params) {
    this.loadDashboardData();
  }
}

class AdminPanel extends LarcComponent {
  beforeRoute(params) {
    if (!authGuard.requiresRole('admin')) {
      navigate('/unauthorized');
      return false;
    }
    return true;
  }
}

```

9.5.2 Route Transition Guards

Sometimes you need to prevent users from leaving a page—usually because they have unsaved changes and you’re trying to save them from themselves:

```

class PostEditor extends LarcComponent {
  constructor() {
    super();
    this.isDirty = false;
    this.originalContent = '';
  }

  beforeRouteLeave(to, from) {
    if (this.isDirty) {
      const answer = confirm(
        'You have unsaved changes. Are you sure you want to leave?'
      );
      return answer; // true = allow navigation, false = cancel
    }
  }
}

```

```

    }
    return true;
  }

  handleContentChange(event) {
    this.isDirty = event.target.value !== this.originalContent;
  }

  async handleSave() {
    await this.savePost();
    this.isDirty = false;
    this.originalContent = this.getEditorContent();
  }
}

```

9.6 Deep Linking and URL State

Deep linking is the practice of encoding application state in the URL so users can bookmark, share, or return to specific states. It's the difference between sharing “myapp.com” and sharing “myapp.com/products?category=electronics&sort=price&page=3”—one is helpful, the other is a digital shrug.

9.6.1 Query Parameters

Query parameters are perfect for filters, search terms, pagination, and other non-hierarchical state:

```

class ProductList extends LarcComponent {
  onRoute(params, query) {
    // params from route pattern, query from ?key=value
    const {
      category = 'all',
      sort = 'name',
      page = 1,
      search = ''
    } = query;

    this.loadProducts({ category, sort, page, search });
  }

  handleFilterChange(category) {
    const currentQuery = this.getQueryParams();
    navigate(`/products?${new URLSearchParams({
      ...currentQuery,
      category,
      page: 1 // Reset to first page when filter changes
    })}`);
  }
}

```

```

handleSortChange(sort) {
  const currentQuery = this.getQueryParams();
  navigate(`/products?${new URLSearchParams({
    ...currentQuery,
    sort
  })}`);
}

getQueryParams() {
  return Object.fromEntries(
    new URLSearchParams(window.location.search)
  );
}
}

```

9.6.2 Hash Fragments

Hash fragments (`#section-name`) are useful for scrolling to specific sections and maintaining scroll position:

```

class DocumentationView extends LarcComponent {
  onRoute(params) {
    this.loadDocument(params.docId);
  }

  afterRender() {
    // Scroll to hash target if present
    const hash = window.location.hash;
    if (hash) {
      const element = this.querySelector(hash);
      if (element) {
        element.scrollIntoView({ behavior: 'smooth' });
      }
    }
  }

  template() {
    return `
      <article>
        <h1 id="introduction">Introduction</h1>
        <p>Content here...</p>

        <h2 id="getting-started">Getting Started</h2>
        <p>More content...</p>

        <nav class="table-of-contents">
          <a href="#introduction">Introduction</a>

```

```

        <a href="#getting-started">Getting Started</a>
      </nav>
    </article>
  `;
}
}

```

9.7 History Management

The browser's history API is like a time machine, but one that only goes to boring places like “the page you were just on.” LARC wraps this API to make history management more pleasant.

9.7.1 Push vs. Replace

When navigating, you can either push a new entry onto the history stack or replace the current entry:

```

// Push new entry (default behavior)
// User can click back to return to previous page
navigate('/products/123');

// Replace current entry
// User clicks back and skips this page entirely
navigate('/login', { replace: true });

```

Replace is useful for:

- Redirect chains (login -> loading -> dashboard)
- Temporary states (splash screens, loading views)
- Fixing invalid URLs (redirect /old-path to /new-path)

9.7.2 Listening to History Changes

Sometimes you need to react to back/forward button clicks:

```

class App extends LarcComponent {
  constructor() {
    super();
    this.setupHistoryListener();
  }

  setupHistoryListener() {
    window.addEventListener('popstate', (event) => {
      // User clicked back or forward
      this.handleNavigation(event.state);
    });
  }

  handleNavigation(state) {

```

```

    // Restore application state from history state
    if (state && state.scrollTop) {
        window.scrollTo(0, state.scrollTop);
    }
}

saveScrollPosition() {
    history.replaceState({
        scrollTop: window.scrollY
    }, '');
}
}

```

9.7.3 Preserving Scroll Position

Nothing frustrates users more than losing their scroll position when navigating. Here's a pattern for preserving it:

```

class ScrollManager {
    constructor() {
        this.positions = new Map();
        this.setupListeners();
    }

    setupListeners() {
        // Save scroll position before navigating away
        window.addEventListener('beforeunload', () => {
            this.savePosition(window.location.pathname);
        });

        // Restore scroll position after navigation
        window.addEventListener('load', () => {
            this.restorePosition(window.location.pathname);
        });
    }

    savePosition(path) {
        this.positions.set(path, {
            x: window.scrollX,
            y: window.scrollY
        });
    }

    restorePosition(path) {
        const position = this.positions.get(path);
        if (position) {
            window.scrollTo(position.x, position.y);
        } else {

```

```

        window.scrollTo(0, 0); // Default to top
    }
}
}

```

9.8 Nested Routes and Layouts

Real applications have hierarchical navigation structures. You might have a main layout with a header and sidebar, then nested views that change based on the route. LARC supports this through component composition:

```

<pan-app id="app">
  <app-layout>
    <pan-routes>
      <pan-route path="/" component="home-view"></pan-route>
      <pan-route path="/products*" component="product-section"></pan-route>
      <pan-route path="/admin*" component="admin-section"></pan-route>
    </pan-routes>
  </app-layout>
</pan-app>

```

The product-section component contains its own nested routes:

```

class ProductSection extends LarcComponent {
  template() {
    return `
      <div class="product-section">
        <nav class="sidebar">
          <a href="/products">All Products</a>
          <a href="/products/categories">Categories</a>
          <a href="/products/new">Add New</a>
        </nav>

        <main class="content">
          <pan-routes>
            <pan-route path="/products" component="product-list"></pan-route>
            <pan-route path="/products/categories" component="category-list"></pan-route>
            <pan-route path="/products/new" component="product-form"></pan-route>
            <pan-route path="/products/:id" component="product-detail"></pan-route>
          </pan-routes>
        </main>
      </div>
    `;
  }
}

```

9.9 Link Handling and Active States

Navigation links should indicate which page is currently active. LARC provides utilities for this:

```
class NavBar extends LarcComponent {
  constructor() {
    super();
    this.currentPath = window.location.pathname;

    // Update active state when route changes
    window.addEventListener('popstate', () => {
      this.currentPath = window.location.pathname;
      this.render();
    });
  }

  isActive(path) {
    return this.currentPath === path;
  }

  isActivePrefix(prefix) {
    return this.currentPath.startsWith(prefix);
  }

  template() {
    return `
      <nav class="navbar">
        <a href="/" class="${this.isActive('/') ? 'active' : ''}">
          Home
        </a>
        <a href="/products" class="${this.isActivePrefix('/products') ? 'active' : ''}">
          Products
        </a>
        <a href="/about" class="${this.isActive('/about') ? 'active' : ''}">
          About
        </a>
      </nav>
    `;
  }
}
```

9.10 SEO Considerations

Client-side routing can be problematic for search engines if not handled properly. While modern search crawlers can execute JavaScript, it's still wise to follow best practices:

9.10.1 Server-Side Rendering (SSR)

For maximum SEO, consider implementing server-side rendering:

```
// server.js
import { renderToString } from '@larc/ssr';
import { App } from './app.js';

app.get('*', async (req, res) => {
  const html = await renderToString(App, {
    path: req.path,
    query: req.query
  });

  res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>${getTitle(req.path)}</title>
        <meta name="description" content="${getDescription(req.path)}">
      </head>
      <body>
        <div id="app">${html}</div>
        <script src="/bundle.js"></script>
      </body>
    </html>
  `);
});
```

9.10.2 Meta Tags and Titles

Update document title and meta tags when routes change:

```
class SEOManager {
  updateMeta(route, data) {
    // Update title
    document.title = data.title || 'Default Title';

    // Update description
    this.setMetaTag('description', data.description || '');

    // Update Open Graph tags for social sharing
    this.setMetaTag('og:title', data.title);
    this.setMetaTag('og:description', data.description);
    this.setMetaTag('og:url', window.location.href);

    // Update canonical URL
    this.setLinkTag('canonical', window.location.href);
  }
}
```

```

setMetaTag(name, content) {
  let element = document.querySelector(`meta[name="${name}"]`) ||
    document.querySelector(`meta[property="${name}"]`);

  if (!element) {
    element = document.createElement('meta');
    const attr = name.startsWith('og:') ? 'property' : 'name';
    element.setAttribute(attr, name);
    document.head.appendChild(element);
  }

  element.setAttribute('content', content);
}

setLinkTag(rel, href) {
  let element = document.querySelector(`link[rel="${rel}"]`);

  if (!element) {
    element = document.createElement('link');
    element.setAttribute('rel', rel);
    document.head.appendChild(element);
  }

  element.setAttribute('href', href);
}

const seoManager = new SEOManager();

class ProductDetail extends LarcComponent {
  async onRoute(params) {
    const product = await this.loadProduct(params.id);

    seoManager.updateMeta(this.route, {
      title: `${product.name} - Our Store`,
      description: product.description,
      image: product.imageUrl
    });
  }
}

```

9.10.3 Prerendering

For static content, consider prerendering routes at build time:

```

// build-prerender.js
import { prerender } from '@larc/prerender';

```

```
const routes = [
  '/',
  '/about',
  '/products',
  '/contact'
];

async function buildPrerenderedPages() {
  for (const route of routes) {
    const html = await prerender(route);
    const filename = route === '/' ? 'index.html' : `${route}/index.html`;
    await fs.writeFile(`dist/${filename}`, html);
  }
}

buildPrerenderedPages();
```

9.11 Putting It All Together

Let's create a complete example that demonstrates all these concepts:

```
// app.js
import { LarcComponent, navigate } from '@larc/core';

class MainApp extends LarcComponent {
  constructor() {
    super();
    this.authGuard = new AuthGuard();
    this.seoManager = new SEOManager();
    this.setupNavigation();
  }

  setupNavigation() {
    window.addEventListener('popstate', () => {
      this.render();
    });
  }

  template() {
    return `
      <div class="app">
        <app-header></app-header>
        <pan-routes>
          <pan-route path="/" component="home-view"></pan-route>
          <pan-route path="/products" component="product-list"></pan-route>
          <pan-route path="/products/:id" component="product-detail"></pan-route>
          <pan-route path="/cart" component="shopping-cart"></pan-route>
        </pan-routes>
      </div>
    `;
  }
}
```

```
        <pan-route path="/checkout" component="checkout-view"></pan-route>
        <pan-route path="/account*" component="account-section"></pan-route>
        <pan-route path="*" component="not-found-view"></pan-route>
      </pan-routes>
      <app-footer></app-footer>
    </div>
  `;
}
}

customElements.define('main-app', MainApp);
```

Routing in LARC transforms your application from a collection of disconnected pages into a cohesive, navigable experience. With proper route organization, parameter handling, navigation guards, and SEO considerations, you can build applications that feel responsive, intelligent, and easy to use—even when users inevitably click the back button seventeen times trying to find that one product they saw earlier.

In the next chapter, we'll tackle forms and user input, which is where users finally get to talk back to your application (and boy, do they have opinions).

Chapter 10

Forms and User Input

In which we learn to gracefully accept data from users, who are simultaneously your application's reason for existing and its greatest source of chaos

Forms are the primary way users communicate with your application, which means they're simultaneously the most important and most frustrating part of web development. Users will try to enter phone numbers with letters, paste entire essays into single-line inputs, and somehow manage to submit forms with negative quantities of products. Your job is to accept their input gracefully while gently steering them toward something your database can actually process.

In this chapter, we'll explore LARC's approach to form handling, from basic input binding to sophisticated schema-driven forms. We'll cover validation strategies that don't make users want to throw their keyboards, file upload patterns that work with modern APIs, and rich text editing that goes beyond the humble textarea. By the end, you'll be equipped to build forms that are both powerful and forgiving—a rare combination in web development.

10.1 The Fundamentals of Form Handling

Let's start with the basics. A form in LARC is just HTML with JavaScript event handling—no magic, no framework-specific syntax, just the web platform doing what it does best.

10.1.1 Basic Form Structure

Here's a simple login form:

```
class LoginForm extends LarcComponent {
  constructor() {
    super();
    this.error = null;
    this.loading = false;
  }

  async handleSubmit(event) {
    event.preventDefault(); // Prevent default form submission
  }
}
```

```
this.loading = true;
this.error = null;
this.render();

const formData = new FormData(event.target);
const credentials = {
  email: formData.get('email'),
  password: formData.get('password'),
  remember: formData.get('remember') === 'on'
};

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

  if (response.ok) {
    const user = await response.json();
    this.handleLoginSuccess(user);
  } else {
    const error = await response.json();
    this.error = error.message;
  }
} catch (err) {
  this.error = 'Network error. Please try again.';
} finally {
  this.loading = false;
  this.render();
}

handleLoginSuccess(user) {
  localStorage.setItem('user', JSON.stringify(user));
  navigate('/dashboard');
}

template() {
  return `
    <form class="login-form" onsubmit="this.handleSubmit(event)">
      <h2>Login</h2>

      ${this.error ? `<div class="error">${this.error}</div>` : ''}

      <div class="form-group">
        <label for="email">Email</label>
```

```

        <input
            type="email"
            id="email"
            name="email"
            required
            autocomplete="email">
    </div>

    <div class="form-group">
        <label for="password">Password</label>
        <input
            type="password"
            id="password"
            name="password"
            required
            autocomplete="current-password">
    </div>

    <div class="form-group">
        <label>
            <input type="checkbox" name="remember">
            Remember me
        </label>
    </div>

    <button type="submit" ?disabled="${this.loading}">
        ${this.loading ? 'Logging in...' : 'Login'}
    </button>
</form>
`
;
}
}

```

This example demonstrates several key patterns:

1. **preventDefault()** stops the browser's default form submission
2. **FormData** extracts values from form inputs
3. **Loading states** provide feedback during async operations
4. **Error handling** displays meaningful messages to users

10.1.2 Two-Way Data Binding

Sometimes you want form inputs to sync with component state in real-time. While LARC doesn't provide automatic two-way binding (we're not monsters), you can implement it easily:

```

class UserProfile extends LarcComponent {
    constructor() {
        super();
        this.user = {

```

```
    name: '',
    email: '',
    bio: '',
    notifications: true
  };
}

handleInput(field, event) {
  this.user[field] = event.target.value;
  // Optionally re-render to update dependent UI
  this.updatePreview();
}

handleCheckbox(field, event) {
  this.user[field] = event.target.checked;
  this.render();
}

updatePreview() {
  const preview = this.querySelector('.profile-preview');
  if (preview) {
    preview.textContent = this.user.bio || 'No bio provided';
  }
}

template() {
  return `
    <form class="profile-form">
      <div class="form-group">
        <label for="name">Name</label>
        <input
          type="text"
          id="name"
          value="${this.user.name}"
          oninput="this.handleInput('name', event)">
      </div>

      <div class="form-group">
        <label for="email">Email</label>
        <input
          type="email"
          id="email"
          value="${this.user.email}"
          oninput="this.handleInput('email', event)">
      </div>

      <div class="form-group">
```

```

        <label for="bio">Bio</label>
        <textarea
          id="bio"
          rows="4"
          oninput="this.handleInput('bio', event)">${this.user.bio}</textarea>
      </div>

      <div class="form-group">
        <label>
          <input
            type="checkbox"
            ?checked="${this.user.notifications}"
            onchange="this.handleCheckbox('notifications', event)">
            Email notifications
          </label>
        </div>

        <div class="profile-preview">
          ${this.user.bio || 'No bio provided'}
        </div>
      </form>
    `;
  }
}

```

10.2 Validation Strategies

Validation is like parenting: you need to set boundaries, but if you're too strict, everyone ends up frustrated. The key is providing helpful guidance without being obnoxious about it.

10.2.1 HTML5 Built-in Validation

Start with HTML5's native validation attributes—they're free, accessible, and work even if JavaScript fails:

```

<input type="email" required
  pattern="^[^@]+@[^@]+\.[^@]+"
  title="Please enter a valid email address">

<input type="tel"
  pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}"
  placeholder="123-456-7890"
  title="Format: 123-456-7890">

<input type="number"
  min="1"
  max="100"
  step="1">

```

```
<input type="url"
        placeholder="https://example.com">

<input type="text"
        minlength="3"
        maxlength="50"
        required>
```

10.2.2 Custom Validation Logic

For more sophisticated validation, implement custom logic:

```
class RegistrationForm extends LarcComponent {
  constructor() {
    super();
    this.errors = {};
    this.touched = {};
  }

  validateEmail(email) {
    if (!email) {
      return 'Email is required';
    }
    if (!/^[\s@]+@[^\s@]+\.[^\s@]+$/.test(email)) {
      return 'Please enter a valid email address';
    }
    return null;
  }

  validatePassword(password) {
    if (!password) {
      return 'Password is required';
    }
    if (password.length < 8) {
      return 'Password must be at least 8 characters';
    }
    if (!/[A-Z]/.test(password)) {
      return 'Password must contain at least one uppercase letter';
    }
    if (!/[a-z]/.test(password)) {
      return 'Password must contain at least one lowercase letter';
    }
    if (!/[0-9]/.test(password)) {
      return 'Password must contain at least one number';
    }
    return null;
  }
}
```

```
validatePasswordConfirm(password, confirm) {
  if (!confirm) {
    return 'Please confirm your password';
  }
  if (password !== confirm) {
    return 'Passwords do not match';
  }
  return null;
}

validateField(field, value, allValues = {}) {
  switch (field) {
    case 'email':
      return this.validateEmail(value);
    case 'password':
      return this.validatePassword(value);
    case 'passwordConfirm':
      return this.validatePasswordConfirm(allValues.password, value);
    default:
      return null;
  }
}

handleBlur(field, event) {
  this.touched[field] = true;
  const error = this.validateField(field, event.target.value, this.getFormValues());
  this.errors[field] = error;
  this.render();
}

handleSubmit(event) {
  event.preventDefault();

  const values = this.getFormValues();
  const newErrors = {};

  // Validate all fields
  Object.keys(values).forEach(field => {
    const error = this.validateField(field, values[field], values);
    if (error) {
      newErrors[field] = error;
    }
  });

  if (Object.keys(newErrors).length > 0) {
    this.errors = newErrors;
    this.touched = Object.keys(values).reduce((acc, key) => {
```

```

        acc[key] = true;
        return acc;
      }, {});
      this.render();
      return;
    }

    // Form is valid, submit it
    this.submitRegistration(values);
  }

  getFormValues() {
    const form = this.querySelector('form');
    const formData = new FormData(form);
    return {
      email: formData.get('email'),
      password: formData.get('password'),
      passwordConfirm: formData.get('passwordConfirm')
    };
  }

  template() {
    return `
      <form onsubmit="this.handleSubmit(event)">
        <div class="form-group ${this.errors.email && this.touched.email ? 'error' : ''}">
          <label for="email">Email</label>
          <input
            type="email"
            id="email"
            name="email"
            onblur="this.handleBlur('email', event)">
          ${this.errors.email && this.touched.email ?
            `<span class="error-message">${this.errors.email}</span>` : ''}
        </div>

        <div class="form-group ${this.errors.password && this.touched.password ? 'error' : ''}">
          <label for="password">Password</label>
          <input
            type="password"
            id="password"
            name="password"
            onblur="this.handleBlur('password', event)">
          ${this.errors.password && this.touched.password ?
            `<span class="error-message">${this.errors.password}</span>` : ''}
        </div>

        <div class="form-group ${this.errors.passwordConfirm && this.touched.passwordConfirm ?

```

```

        <label for="passwordConfirm">Confirm Password</label>
        <input
          type="password"
          id="passwordConfirm"
          name="passwordConfirm"
          onBlur="this.handleBlur('passwordConfirm', event)">
        ${this.errors.passwordConfirm && this.touched.passwordConfirm ?
          `<span class="error-message">${this.errors.passwordConfirm}</span>` : ''}
        </div>

        <button type="submit">Register</button>
      </form>
    `;
  }
}

```

10.2.3 Debounced Validation

For fields that require server-side validation (like username availability), debounce the requests:

```

class UsernameField extends LarcComponent {
  constructor() {
    super();
    this.username = '';
    this.checking = false;
    this.available = null;
    this.debounceTimer = null;
  }

  handleInput(event) {
    this.username = event.target.value;
    this.available = null; // Reset availability

    clearTimeout(this.debounceTimer);

    if (this.username.length >= 3) {
      this.checking = true;
      this.render();

      this.debounceTimer = setTimeout(() => {
        this.checkAvailability(this.username);
      }, 500); // Wait 500ms after user stops typing
    } else {
      this.checking = false;
      this.render();
    }
  }
}

```

```

async checkAvailability(username) {
  try {
    const response = await fetch(`/api/check-username?username=${encodeURIComponent(username)}`);
    const data = await response.json();
    this.available = data.available;
  } catch (err) {
    console.error('Error checking username:', err);
  } finally {
    this.checking = false;
    this.render();
  }
}

template() {
  return `
    <div class="form-group">
      <label for="username">Username</label>
      <input
        type="text"
        id="username"
        value="${this.username}"
        oninput="this.handleInput(event)"
        minlength="3"
        maxlength="20">

      ${this.checking ? '<span class="checking">Checking...</span>' : ''}

      ${this.available === true ?
        '<span class="success">[v] Available</span>' : ''}

      ${this.available === false ?
        '<span class="error">[x] Username taken</span>' : ''}
    </div>
  `;
}

```

10.3 Schema-Driven Forms

For complex forms, manually writing validation for each field becomes tedious. Schema-driven forms define the structure and rules in data, then generate the UI automatically.

10.3.1 Defining a Schema

```

const productSchema = {
  name: {

```

```
    type: 'text',
    label: 'Product Name',
    required: true,
    minLength: 3,
    maxLength: 100
  },
  description: {
    type: 'textarea',
    label: 'Description',
    required: true,
    minLength: 10,
    rows: 5
  },
  category: {
    type: 'select',
    label: 'Category',
    required: true,
    options: [
      { value: 'electronics', label: 'Electronics' },
      { value: 'clothing', label: 'Clothing' },
      { value: 'food', label: 'Food & Beverage' },
      { value: 'other', label: 'Other' }
    ]
  },
  price: {
    type: 'number',
    label: 'Price',
    required: true,
    min: 0.01,
    step: 0.01,
    prefix: '$'
  },
  inStock: {
    type: 'checkbox',
    label: 'In Stock',
    defaultValue: true
  },
  tags: {
    type: 'text',
    label: 'Tags (comma-separated)',
    placeholder: 'organic, gluten-free, local'
  }
};
```

10.3.2 Schema Form Component

```
class SchemaForm extends LarcComponent {
  constructor(schema, initialValues = {}) {
    super();
    this.schema = schema;
    this.values = { ...initialValues };
    this.errors = {};
    this.touched = {};
  }

  handleInput(field, event) {
    const fieldSchema = this.schema[field];

    if (fieldSchema.type === 'checkbox') {
      this.values[field] = event.target.checked;
    } else {
      this.values[field] = event.target.value;
    }

    // Clear error when user starts correcting
    if (this.errors[field]) {
      delete this.errors[field];
      this.render();
    }
  }

  handleBlur(field) {
    this.touched[field] = true;
    const error = this.validateField(field);
    if (error) {
      this.errors[field] = error;
      this.render();
    }
  }

  validateField(field) {
    const value = this.values[field];
    const fieldSchema = this.schema[field];

    if (fieldSchema.required && !value) {
      return `${fieldSchema.label} is required`;
    }

    if (fieldSchema.minLength && value.length < fieldSchema.minLength) {
      return `${fieldSchema.label} must be at least ${fieldSchema.minLength} characters`;
    }
  }
}
```

```

    if (fieldSchema.maxLength && value.length > fieldSchema.maxLength) {
      return `${fieldSchema.label} must be no more than ${fieldSchema.maxLength} characters`;
    }

    if (fieldSchema.min !== undefined && parseFloat(value) < fieldSchema.min) {
      return `${fieldSchema.label} must be at least ${fieldSchema.min}`;
    }

    if (fieldSchema.max !== undefined && parseFloat(value) > fieldSchema.max) {
      return `${fieldSchema.label} must be no more than ${fieldSchema.max}`;
    }

    if (fieldSchema.pattern && !new RegExp(fieldSchema.pattern).test(value)) {
      return fieldSchema.patternMessage || `${fieldSchema.label} is invalid`;
    }

    return null;
  }

  validateAll() {
    const newErrors = {};

    Object.keys(this.schema).forEach(field => {
      const error = this.validateField(field);
      if (error) {
        newErrors[field] = error;
      }
    });

    return newErrors;
  }

  renderField(fieldName) {
    const field = this.schema[fieldName];
    const value = this.values[fieldName] ?? field.defaultValue ?? '';
    const error = this.errors[fieldName] && this.touched[fieldName];

    const commonAttrs = `
      id="${fieldName}"
      name="${fieldName}"
      onBlur="this.handleBlur('${fieldName}')"
    `;

    let input;

    switch (field.type) {
      case 'textarea':

```

```

input = `
  <textarea ${commonAttrs}
    rows="${field.rows || 3}"
    oninput="this.handleInput('${fieldName}', event)"
    ${field.required ? 'required' : ''}>${value}</textarea>
`;
break;

case 'select':
input = `
  <select ${commonAttrs}
    onchange="this.handleInput('${fieldName}', event)"
    ${field.required ? 'required' : ''}>
    <option value="">Select ${field.label}</option>
    ${field.options.map(opt => `
      <option value="${opt.value}" ${value === opt.value ? 'selected' : ''}>
        ${opt.label}
      </option>
    `).join('')}
  </select>
`;
break;

case 'checkbox':
input = `
  <input type="checkbox" ${commonAttrs}
    onchange="this.handleInput('${fieldName}', event)"
    ${value ? 'checked' : ''}>
`;
break;

case 'number':
input = `
  ${field.prefix || ''}
  <input type="number" ${commonAttrs}
    value="${value}"
    oninput="this.handleInput('${fieldName}', event)"
    ${field.min !== undefined ? `min="${field.min}"` : ''}
    ${field.max !== undefined ? `max="${field.max}"` : ''}
    ${field.step !== undefined ? `step="${field.step}"` : ''}
    ${field.required ? 'required' : ''}>
  ${field.suffix || ''}
`;
break;

default: // text, email, tel, url, etc.
input = `

```

```

        <input type="${field.type}" ${commonAttrs}
            value="${value}"
            oninput="this.handleInput('${fieldName}', event)"
            ${field.placeholder ? `placeholder="${field.placeholder}"` : ''}
            ${field.required ? 'required' : ''}>
    `;
}

return `
    <div class="form-group ${error ? 'error' : ''}">
        <label for="${fieldName}">${field.label}</label>
        ${input}
        ${error ? `<span class="error-message">${this.errors[fieldName]}</span>` : ''}
    </div>
    `;
}

handleSubmit(event) {
    event.preventDefault();

    const errors = this.validateAll();

    if (Object.keys(errors).length > 0) {
        this.errors = errors;
        this.touched = Object.keys(this.schema).reduce((acc, key) => {
            acc[key] = true;
            return acc;
        }, {});
        this.render();
        return;
    }

    this.onSubmit(this.values);
}

onSubmit(values) {
    // Override in subclass or pass as parameter
    console.log('Form submitted:', values);
}

template() {
    return `
        <form class="schema-form" onsubmit="this.handleSubmit(event)">
            ${Object.keys(this.schema).map(field => this.renderField(field)).join('')}

            <div class="form-actions">
                <button type="submit">Submit</button>
    `;
}

```

```

        <button type="button" onclick="this.handleReset()">Reset</button>
      </div>
    </form>
  `;
}
}

```

10.3.3 Using the Schema Form

```

class ProductForm extends SchemaForm {
  constructor() {
    super(productSchema);
  }

  async onSubmit(values) {
    try {
      const response = await fetch('/api/products', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(values)
      });

      if (response.ok) {
        navigate('/products');
      } else {
        const error = await response.json();
        alert(`Error: ${error.message}`);
      }
    } catch (err) {
      alert('Network error. Please try again.');
```

10.4 File Uploads

File uploads are where form handling gets interesting (read: complicated). You need to handle previews, progress indicators, size limits, and mime type validation.

10.4.1 Basic File Upload

```

class FileUpload extends LarcComponent {
  constructor() {
    super();
  }

```

```
this.file = null;
this.preview = null;
this.uploading = false;
this.progress = 0;
}

handleFileSelect(event) {
  const file = event.target.files[0];

  if (!file) {
    return;
  }

  // Validate file type
  const allowedTypes = ['image/jpeg', 'image/png', 'image/gif'];
  if (!allowedTypes.includes(file.type)) {
    alert('Please select a valid image file (JPEG, PNG, or GIF)');
    return;
  }

  // Validate file size (max 5MB)
  const maxSize = 5 * 1024 * 1024;
  if (file.size > maxSize) {
    alert('File size must be less than 5MB');
    return;
  }

  this.file = file;
  this.generatePreview(file);
  this.render();
}

generatePreview(file) {
  const reader = new FileReader();

  reader.onload = (e) => {
    this.preview = e.target.result;
    this.render();
  };

  reader.readAsDataURL(file);
}

async handleUpload() {
  if (!this.file) {
    return;
  }
}
```

```
this.uploading = true;
this.progress = 0;
this.render();

const formData = new FormData();
formData.append('file', this.file);

try {
  const response = await fetch('/api/upload', {
    method: 'POST',
    body: formData
  });

  if (response.ok) {
    const result = await response.json();
    this.handleUploadSuccess(result);
  } else {
    alert('Upload failed');
  }
} catch (err) {
  alert('Network error');
} finally {
  this.uploading = false;
  this.render();
}

handleUploadSuccess(result) {
  console.log('File uploaded:', result.url);
  this.file = null;
  this.preview = null;
  this.render();
}

template() {
  return `
    <div class="file-upload">
      <input
        type="file"
        accept="image/*"
        onchange="this.handleFileSelect(event)"
        ?disabled="${this.uploading}">

      ${this.preview ? `
        <div class="preview">
          
          <button onclick="this.handleUpload()"`
      }
    `
  }
}
```

```

        ?disabled="${this.uploading}">
        ${this.uploading ? 'Uploading...' : 'Upload'}
    </button>
</div>
` : ''}

${this.uploading ? `
    <div class="progress">
        <div class="progress-bar" style="width: ${this.progress}%></div>
    </div>
    ` : ''}
</div>
`;
}
}

```

10.4.2 Upload Progress with XMLHttpRequest

For detailed progress tracking, use XMLHttpRequest instead of fetch:

```

uploadWithProgress(file, onProgress) {
    return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        const formData = new FormData();
        formData.append('file', file);

        xhr.upload.addEventListener('progress', (e) => {
            if (e.lengthComputable) {
                const progress = (e.loaded / e.total) * 100;
                onProgress(progress);
            }
        });

        xhr.addEventListener('load', () => {
            if (xhr.status === 200) {
                resolve(JSON.parse(xhr.responseText));
            } else {
                reject(new Error('Upload failed'));
            }
        });

        xhr.addEventListener('error', () => {
            reject(new Error('Network error'));
        });

        xhr.open('POST', '/api/upload');
        xhr.send(formData);
    });
}

```

```
}

async handleUpload() {
  this.uploading = true;
  this.render();

  try {
    const result = await this.uploadWithProgress(this.file, (progress) => {
      this.progress = progress;
      this.render();
    });

    this.handleUploadSuccess(result);
  } catch (err) {
    alert('Upload failed');
  } finally {
    this.uploading = false;
    this.render();
  }
}
```

10.5 Rich Text Editing

Sometimes a plain textarea isn't enough, and you need formatted text. You have two main approaches: WYSIWYG editors and markdown.

10.5.1 Markdown Editor

Markdown is developer-friendly and produces clean, semantic output:

```
class MarkdownEditor extends LarcComponent {
  constructor() {
    super();
    this.content = '';
    this.previewMode = false;
  }

  handleInput(event) {
    this.content = event.target.value;
    if (this.previewMode) {
      this.updatePreview();
    }
  }

  togglePreview() {
    this.previewMode = !this.previewMode;
    this.render();
  }
}
```

```

}

updatePreview() {
  const preview = this.querySelector('.markdown-preview');
  if (preview) {
    preview.innerHTML = this.renderMarkdown(this.content);
  }
}

renderMarkdown(text) {
  // Simple markdown parser (use a library like marked.js for production)
  return text
    .replace(/^### (.*)/gim, '<h3>$1</h3>')
    .replace(/^## (.*)/gim, '<h2>$1</h2>')
    .replace(/^# (.*)/gim, '<h1>$1</h1>')
    .replace(/\*(.*)\*/gim, '<strong>$1</strong>')
    .replace(/_(.*)_/gim, '<em>$1</em>')
    .replace(/!\[ (.*) \] \[ (.*) \]/gim, '')
    .replace(/\[ (.*) \] \[ (.*) \]/gim, '<a href="$2">$1</a>')
    .replace(/\n/gim, '<br>');
}

insertFormatting(format) {
  const textarea = this.querySelector('textarea');
  const start = textarea.selectionStart;
  const end = textarea.selectionEnd;
  const selectedText = this.content.substring(start, end);

  let insertion;
  switch (format) {
    case 'bold':
      insertion = `**${selectedText}**`;
      break;
    case 'italic':
      insertion = `_${selectedText}_`;
      break;
    case 'link':
      insertion = `[${selectedText}] (url)`;
      break;
    case 'heading':
      insertion = `## ${selectedText}`;
      break;
    default:
      return;
  }

  this.content = this.content.substring(0, start) +

```

```

        insertion +
        this.content.substring(end);

    this.render();
}

template() {
    return `
        <div class="markdown-editor">
            <div class="toolbar">
                <button type="button" onclick="this.insertFormatting('bold')">
                    <strong>B</strong>
                </button>
                <button type="button" onclick="this.insertFormatting('italic')">
                    <em>I</em>
                </button>
                <button type="button" onclick="this.insertFormatting('link')">
                    Link
                </button>
                <button type="button" onclick="this.insertFormatting('heading')">
                    H2
                </button>
                <button type="button" onclick="this.togglePreview()">
                    ${this.previewMode ? 'Edit' : 'Preview'}
                </button>
            </div>

            ${this.previewMode ? `
                <div class="markdown-preview">
                    ${this.renderMarkdown(this.content)}
                </div>
            ` : `
                <textarea
                    rows="10"
                    oninput="this.handleInput(event)">${this.content}</textarea>
            `}

        </div>
    `;
}

```

10.5.2 Integrating Third-Party Editors

For full-featured rich text editing, integrate libraries like Quill or TipTap:

```

import Quill from 'quill';

class RichTextEditor extends LarcComponent {

```

```
constructor() {
  super();
  this.content = '';
  this.editor = null;
}

afterRender() {
  if (!this.editor) {
    const container = this.querySelector('.editor-container');
    this.editor = new Quill(container, {
      theme: 'snow',
      modules: {
        toolbar: [
          ['bold', 'italic', 'underline', 'strike'],
          ['blockquote', 'code-block'],
          [{ 'header': 1 }, { 'header': 2 }],
          [{ 'list': 'ordered' }, { 'list': 'bullet' }],
          [{ 'indent': '-1' }, { 'indent': '+1' }],
          ['link', 'image'],
          ['clean']
        ]
      }
    });

    this.editor.on('text-change', () => {
      this.content = this.editor.root.innerHTML;
    });
  }
}

getContent() {
  return this.content;
}

setContent(html) {
  if (this.editor) {
    this.editor.root.innerHTML = html;
    this.content = html;
  }
}

template() {
  return '<div class="editor-container"></div>';
}
}
```

10.6 Form State Management

For complex forms with multiple steps or interdependent fields, centralized state management helps maintain sanity:

```
class FormState {
  constructor(initialValues = {}) {
    this.values = { ...initialValues };
    this.errors = {};
    this.touched = {};
    this.dirty = false;
    this.listeners = [];
  }

  subscribe(listener) {
    this.listeners.push(listener);
    return () => {
      this.listeners = this.listeners.filter(l => l !== listener);
    };
  }

  notify() {
    this.listeners.forEach(listener => listener(this.getState()));
  }

  getState() {
    return {
      values: { ...this.values },
      errors: { ...this.errors },
      touched: { ...this.touched },
      dirty: this.dirty
    };
  }

  setValue(field, value) {
    this.values[field] = value;
    this.dirty = true;
    this.notify();
  }

  setError(field, error) {
    if (error) {
      this.errors[field] = error;
    } else {
      delete this.errors[field];
    }
    this.notify();
  }
}
```

```

setTouched(field) {
  this.touched[field] = true;
  this.notify();
}

reset(values = {}) {
  this.values = { ...values };
  this.errors = {};
  this.touched = {};
  this.dirty = false;
  this.notify();
}

isValid() {
  return Object.keys(this.errors).length === 0;
}
}

```

Use this state manager in your forms:

```

class MultiStepForm extends LarcComponent {
  constructor() {
    super();
    this.currentStep = 1;
    this.formState = new FormState({
      // Step 1
      name: '',
      email: '',
      // Step 2
      address: '',
      city: '',
      // Step 3
      payment: ''
    });

    this.unsubscribe = this.formState.subscribe(() => {
      this.render();
    });
  }

  disconnectedCallback() {
    this.unsubscribe();
  }

  nextStep() {
    if (this.validateCurrentStep()) {
      this.currentStep++;
    }
  }
}

```

```

        this.render();
    }
}

previousStep() {
    this.currentStep--;
    this.render();
}

validateCurrentStep() {
    // Validate fields for current step
    return true;
}

template() {
    const state = this.formState.getState();

    return `
        <form class="multi-step-form">
            <div class="steps">
                ${this.currentStep === 1 ? this.renderStep1(state) : ''}
                ${this.currentStep === 2 ? this.renderStep2(state) : ''}
                ${this.currentStep === 3 ? this.renderStep3(state) : ''}
            </div>

            <div class="navigation">
                ${this.currentStep > 1 ? `
                    <button type="button" onclick="this.previousStep()">
                        Previous
                    </button>
                ` : ''}

                ${this.currentStep < 3 ? `
                    <button type="button" onclick="this.nextStep()">
                        Next
                    </button>
                ` : `
                    <button type="submit">Submit</button>
                `}
            </div>
        </form>
    `;
}
}

```

10.7 Conclusion

Forms are the battleground where user intent meets application logic. By combining HTML5's built-in capabilities with LARC's component model, you can create forms that validate intelligently, provide helpful feedback, and gracefully handle the chaos users inevitably introduce. Whether you're building simple login forms or complex multi-step wizards, the patterns in this chapter will help you create user inputs that are both powerful and forgiving.

In the next chapter, we'll explore data fetching and APIs—because forms are useless without somewhere to send their data.

Chapter 11

Data Fetching and APIs

In which we learn to retrieve data from distant servers without losing our minds (or our users' patience)

Modern web applications are essentially elaborate interfaces for remote data. They fetch JSON from APIs, subscribe to WebSocket streams, poll for updates, and cache responses like digital squirrels preparing for winter. The challenge isn't just getting data—it's getting it reliably, efficiently, and without making users stare at loading spinners longer than they stare at the actual content.

In this chapter, we'll explore LARC's approach to data fetching, from basic REST API calls to sophisticated real-time communication. We'll cover error handling strategies that acknowledge the chaos of distributed systems, caching patterns that balance freshness with performance, and retry logic that persists without becoming annoying. By the end, you'll be equipped to build applications that fetch data like they know what they're doing, even when the network doesn't.

11.1 The Foundation: Fetch API

JavaScript's Fetch API is the modern standard for making HTTP requests. It's promise-based, supports streaming, and doesn't require external libraries. Let's start with the basics and build up to production-ready patterns.

11.1.1 Basic GET Request

```
class ProductList extends LarcComponent {
  constructor() {
    super();
    this.products = [];
    this.loading = true;
    this.error = null;
  }

  async onMount() {
    await this.loadProducts();
  }
}
```

```
async loadProducts() {
  this.loading = true;
  this.error = null;
  this.render();

  try {
    const response = await fetch('/api/products');

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

    this.products = await response.json();
  } catch (error) {
    this.error = error.message;
    console.error('Failed to load products:', error);
  } finally {
    this.loading = false;
    this.render();
  }
}

template() {
  if (this.loading) {
    return '<div class="loading">Loading products...</div>';
  }

  if (this.error) {
    return `
      <div class="error">
        <p>Failed to load products: ${this.error}</p>
        <button onclick="this.loadProducts()">Retry</button>
      </div>
    `;
  }

  return `
    <div class="product-list">
      ${this.products.map(product => `
        <div class="product-card">
          <h3>${product.name}</h3>
          <p>${product.description}</p>
          <span class="price">${product.price}</span>
        </div>
      `).join('')}
    </div>
  `;
}
```

```

    }
  }
}

```

11.1.2 POST Request with JSON

```

class ProductForm extends LarcComponent {
  async submitProduct(formData) {
    const product = {
      name: formData.get('name'),
      description: formData.get('description'),
      price: parseFloat(formData.get('price')),
      category: formData.get('category')
    };

    try {
      const response = await fetch('/api/products', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Bearer ${this.getAuthToken()}`
        },
        body: JSON.stringify(product)
      });

      if (!response.ok) {
        const error = await response.json();
        throw new Error(error.message || 'Failed to create product');
      }

      const created = await response.json();
      navigate(`/products/${created.id}`);
    } catch (error) {
      this.showError(error.message);
    }
  }

  getAuthToken() {
    return localStorage.getItem('auth_token');
  }
}

```

11.1.3 Request Configuration

For consistent API communication, create a configured fetch wrapper:

```

class APIClient {
  constructor(baseUrl, options = {}) {

```

```

    this.baseUrl = baseUrl;
    this.defaultHeaders = options.headers || {};
    this.timeout = options.timeout || 30000;
  }

  async request(endpoint, options = {}) {
    const url = `${this.baseUrl}${endpoint}`;
    const config = {
      ...options,
      headers: {
        'Content-Type': 'application/json',
        ...this.defaultHeaders,
        ...options.headers
      }
    };

    // Add auth token if available
    const token = this.getAuthToken();
    if (token) {
      config.headers['Authorization'] = `Bearer ${token}`;
    }

    // Create timeout promise
    const timeoutPromise = new Promise((_, reject) => {
      setTimeout(() => reject(new Error('Request timeout')), this.timeout);
    });

    // Race between fetch and timeout
    try {
      const response = await Promise.race([
        fetch(url, config),
        timeoutPromise
      ]);

      if (!response.ok) {
        await this.handleHTTPError(response);
      }

      return await response.json();
    } catch (error) {
      throw this.enhanceError(error);
    }
  }

  async handleHTTPError(response) {
    let message = `HTTP ${response.status}: ${response.statusText}`;
  }

```

```
try {
  const body = await response.json();
  if (body.message) {
    message = body.message;
  }
} catch {
  // Response body wasn't JSON
}

const error = new Error(message);
error.status = response.status;
error.response = response;
throw error;
}

enhanceError(error) {
  if (error.name === 'AbortError') {
    error.message = 'Request was cancelled';
  } else if (!navigator.onLine) {
    error.message = 'No internet connection';
    error.offline = true;
  }
  return error;
}

getAuthToken() {
  return localStorage.getItem('auth_token');
}

// Convenience methods
get(endpoint, options) {
  return this.request(endpoint, { ...options, method: 'GET' });
}

post(endpoint, data, options) {
  return this.request(endpoint, {
    ...options,
    method: 'POST',
    body: JSON.stringify(data)
  });
}

put(endpoint, data, options) {
  return this.request(endpoint, {
    ...options,
    method: 'PUT',
    body: JSON.stringify(data)
  });
}
```

```

    });
  }

  patch(endpoint, data, options) {
    return this.request(endpoint, {
      ...options,
      method: 'PATCH',
      body: JSON.stringify(data)
    });
  }

  delete(endpoint, options) {
    return this.request(endpoint, { ...options, method: 'DELETE' });
  }
}

// Create global API client instance
const api = new APIClient('/api', {
  timeout: 10000,
  headers: {
    'X-App-Version': '1.0.0'
  }
});

```

Now use the API client throughout your application:

```

class ProductDetail extends LarcComponent {
  async onRoute(params) {
    try {
      this.product = await api.get(`/products/${params.id}`);
      this.render();
    } catch (error) {
      if (error.status === 404) {
        navigate('/not-found');
      } else {
        this.showError(error.message);
      }
    }
  }

  async handleDelete() {
    if (!confirm('Delete this product?')) {
      return;
    }

    try {
      await api.delete(`/products/${this.product.id}`);
      navigate('/products');
    }
  }
}

```

```

    } catch (error) {
      this.showError('Failed to delete product');
    }
  }
}

```

11.2 Error Handling and Retries

Networks are unreliable, servers crash, and APIs return errors. Good error handling is what separates professional applications from abandoned side projects.

11.2.1 Retry Logic

Implement exponential backoff for transient failures:

```

class RetryableAPIClient extends APIClient {
  async requestWithRetry(endpoint, options = {}, retries = 3) {
    let lastError;
    let delay = 1000; // Start with 1 second

    for (let attempt = 0; attempt <= retries; attempt++) {
      try {
        return await this.request(endpoint, options);
      } catch (error) {
        lastError = error;

        // Don't retry client errors (4xx) except 429 (rate limit)
        if (error.status >= 400 && error.status < 500 && error.status !== 429) {
          throw error;
        }

        // Don't retry if we're out of attempts
        if (attempt === retries) {
          break;
        }

        // Wait before retrying (exponential backoff with jitter)
        const jitter = Math.random() * 1000;
        await this.sleep(delay + jitter);
        delay *= 2; // Double the delay each time
      }
    }

    throw lastError;
  }

  sleep(ms) {

```

```

    return new Promise(resolve => setTimeout(resolve, ms));
  }

  // Override convenience methods to use retry logic
  get(endpoint, options) {
    return this.requestWithRetry(endpoint, { ...options, method: 'GET' });
  }

  post(endpoint, data, options) {
    return this.requestWithRetry(endpoint, {
      ...options,
      method: 'POST',
      body: JSON.stringify(data)
    });
  }
}

const api = new RetryableAPIClient('/api');
```

11.2.2 Circuit Breaker Pattern

Prevent cascading failures by temporarily disabling requests to failing services:

```

class CircuitBreaker {
  constructor(threshold = 5, timeout = 60000) {
    this.failureThreshold = threshold;
    this.timeout = timeout;
    this.failures = 0;
    this.state = 'CLOSED'; // CLOSED, OPEN, HALF_OPEN
    this.nextAttempt = Date.now();
  }

  async execute(fn) {
    if (this.state === 'OPEN') {
      if (Date.now() < this.nextAttempt) {
        throw new Error('Circuit breaker is OPEN');
      }
      // Try to recover
      this.state = 'HALF_OPEN';
    }

    try {
      const result = await fn();
      this.onSuccess();
      return result;
    } catch (error) {
      this.onFailure();
      throw error;
    }
  }
}
```

```

    }
}

onSuccess() {
    this.failures = 0;
    if (this.state === 'HALF_OPEN') {
        this.state = 'CLOSED';
    }
}

onFailure() {
    this.failures++;
    if (this.failures >= this.failureThreshold) {
        this.state = 'OPEN';
        this.nextAttempt = Date.now() + this.timeout;
    }
}

getState() {
    return this.state;
}
}

// Use with API client
class ResilientAPIClient extends APIClient {
    constructor(baseUrl, options = {}) {
        super(baseUrl, options);
        this.circuitBreaker = new CircuitBreaker();
    }

    async request(endpoint, options = {}) {
        return this.circuitBreaker.execute(async () => {
            return await super.request(endpoint, options);
        });
    }
}

```

11.3 Caching Strategies

Caching reduces server load, speeds up your application, and works when the network doesn't. But cache invalidation is one of computer science's hardest problems, so tread carefully.

11.3.1 In-Memory Cache

```

class CacheManager {
    constructor(defaultTTL = 5 * 60 * 1000) { // 5 minutes default

```

```
this.cache = new Map();
this.defaultTTL = defaultTTL;
}

set(key, value, ttl = this.defaultTTL) {
  this.cache.set(key, {
    value,
    expires: Date.now() + ttl
  });
}

get(key) {
  const item = this.cache.get(key);

  if (!item) {
    return null;
  }

  if (Date.now() > item.expires) {
    this.cache.delete(key);
    return null;
  }

  return item.value;
}

invalidate(key) {
  this.cache.delete(key);
}

invalidatePattern(pattern) {
  const regex = new RegExp(pattern);
  for (const key of this.cache.keys()) {
    if (regex.test(key)) {
      this.cache.delete(key);
    }
  }
}

clear() {
  this.cache.clear();
}

class CachedAPIClient extends APIClient {
  constructor(baseUrl, options = {}) {
    super(baseUrl, options);
  }
}
```

```

    this.cache = new CacheManager();
  }

  async get(endpoint, options = {}) {
    const cacheKey = this.getCacheKey('GET', endpoint);
    const cached = this.cache.get(cacheKey);

    if (cached && !options.bypassCache) {
      return cached;
    }

    const data = await super.get(endpoint, options);
    this.cache.set(cacheKey, data, options.cacheTTL);
    return data;
  }

  getCacheKey(method, endpoint) {
    return `${method}:${endpoint}`;
  }

  invalidateCache(endpoint) {
    this.cache.invalidatePattern(endpoint);
  }
}

const api = new CachedAPIClient('/api');

// Usage
class ProductList extends LarcComponent {
  async loadProducts() {
    // This will use cache if available
    this.products = await api.get('/products');
    this.render();
  }

  async refreshProducts() {
    // Bypass cache and get fresh data
    this.products = await api.get('/products', { bypassCache: true });
    this.render();
  }
}

```

11.3.2 LocalStorage Cache

For persistence across sessions:

```

class PersistentCache extends CacheManager {
  constructor(prefix = 'cache:', defaultTTL = 5 * 60 * 1000) {

```

```
super(defaultTTL);
this.prefix = prefix;
this.loadFromStorage();
}

loadFromStorage() {
  try {
    for (let i = 0; i < localStorage.length; i++) {
      const key = localStorage.key(i);
      if (key.startsWith(this.prefix)) {
        const data = JSON.parse(localStorage.getItem(key));
        const originalKey = key.substring(this.prefix.length);
        this.cache.set(originalKey, data);
      }
    }
  } catch (error) {
    console.error('Failed to load cache from storage:', error);
  }
}

set(key, value, ttl = this.defaultTTL) {
  super.set(key, value, ttl);

  try {
    localStorage.setItem(
      this.prefix + key,
      JSON.stringify({ value, expires: Date.now() + ttl })
    );
  } catch (error) {
    console.error('Failed to persist cache:', error);
  }
}

invalidate(key) {
  super.invalidate(key);
  localStorage.removeItem(this.prefix + key);
}

clear() {
  super.clear();
  const keysToRemove = [];

  for (let i = 0; i < localStorage.length; i++) {
    const key = localStorage.key(i);
    if (key.startsWith(this.prefix)) {
      keysToRemove.push(key);
    }
  }
}
```

```

    }

    keysToRemove.forEach(key => localStorage.removeItem(key));
  }
}

```

11.3.3 Stale-While-Revalidate

Serve cached data immediately while fetching fresh data in the background:

```

class SWRAPIClient extends APIClient {
  constructor(baseUrl, options = {}) {
    super(baseUrl, options);
    this.cache = new CacheManager();
  }

  async get(endpoint, options = {}) {
    const cacheKey = `GET:${endpoint}`;
    const cached = this.cache.get(cacheKey);

    // Return cached data immediately if available
    if (cached && !options.bypassCache) {
      // Fetch fresh data in background
      this.revalidate(endpoint, cacheKey, options);
      return cached;
    }

    // No cache, fetch fresh data
    const data = await super.get(endpoint, options);
    this.cache.set(cacheKey, data);
    return data;
  }

  async revalidate(endpoint, cacheKey, options) {
    try {
      const fresh = await super.get(endpoint, options);
      this.cache.set(cacheKey, fresh);

      // Notify subscribers of new data
      this.notifySubscribers(cacheKey, fresh);
    } catch (error) {
      console.error('Revalidation failed:', error);
    }
  }

  notifySubscribers(key, data) {
    const event = new CustomEvent('cache-update', {
      detail: { key, data }
    });
  }
}

```

```
});  
window.dispatchEvent(event);  
}  
}
```

11.4 GraphQL Integration

GraphQL provides a more flexible alternative to REST, allowing clients to request exactly the data they need.

11.4.1 GraphQL Client

```
class GraphQLClient {  
  constructor(endpoint) {  
    this.endpoint = endpoint;  
  }  
  
  async query(query, variables = {}) {  
    const response = await fetch(this.endpoint, {  
      method: 'POST',  
      headers: {  
        'Content-Type': 'application/json',  
        'Authorization': `Bearer ${this.getAuthToken()}`  
      },  
      body: JSON.stringify({ query, variables })  
    });  
  
    const result = await response.json();  
  
    if (result.errors) {  
      throw new Error(result.errors.map(e => e.message).join(', '));  
    }  
  
    return result.data;  
  }  
  
  async mutate(mutation, variables = {}) {  
    return this.query(mutation, variables);  
  }  
  
  getAuthToken() {  
    return localStorage.getItem('auth_token');  
  }  
}  
  
const graphql = new GraphQLClient('/graphql');
```

11.4.2 Using GraphQL Queries

```
class ProductList extends LarcComponent {
  async loadProducts() {
    const query = `
      query GetProducts($category: String, $limit: Int) {
        products(category: $category, limit: $limit) {
          id
          name
          description
          price
          category
          imageUrl
          inStock
        }
      }
    `;

    try {
      const data = await graphql.query(query, {
        category: this.selectedCategory,
        limit: 20
      });

      this.products = data.products;
      this.render();
    } catch (error) {
      this.showError(error.message);
    }
  }

  async createProduct(product) {
    const mutation = `
      mutation CreateProduct($input: ProductInput!) {
        createProduct(input: $input) {
          id
          name
          price
        }
      }
    `;

    try {
      const data = await graphql.mutate(mutation, {
        input: product
      });
    }
  }
}
```

```
        navigate(`/products/${data.createProduct.id}`);
    } catch (error) {
        this.showError(error.message);
    }
}
}
```

11.5 WebSocket Communication

WebSockets enable real-time bidirectional communication, perfect for chat applications, live updates, and collaborative features.

11.5.1 WebSocket Client

```
class WebSocketClient {
  constructor(url) {
    this.url = url;
    this.ws = null;
    this.listeners = new Map();
    this.reconnectDelay = 1000;
    this.maxReconnectDelay = 30000;
    this.reconnectAttempts = 0;
  }

  connect() {
    this.ws = new WebSocket(this.url);

    this.ws.onopen = () => {
      console.log('WebSocket connected');
      this.reconnectAttempts = 0;
      this.reconnectDelay = 1000;
      this.emit('connected');
    };

    this.ws.onmessage = (event) => {
      try {
        const message = JSON.parse(event.data);
        this.handleMessage(message);
      } catch (error) {
        console.error('Failed to parse message:', error);
      }
    };

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

```
};

this.ws.onclose = () => {
  console.log('WebSocket disconnected');
  this.emit('disconnected');
  this.attemptReconnect();
};
}

attemptReconnect() {
  this.reconnectAttempts++;
  const delay = Math.min(
    this.reconnectDelay * Math.pow(2, this.reconnectAttempts),
    this.maxReconnectDelay
  );

  console.log(`Reconnecting in ${delay}ms...`);

  setTimeout(() => {
    this.connect();
  }, delay);
}

handleMessage(message) {
  const { type, data } = message;
  this.emit(type, data);
}

send(type, data) {
  if (this.ws?.readyState === WebSocket.OPEN) {
    this.ws.send(JSON.stringify({ type, data }));
  } else {
    console.error('WebSocket not connected');
  }
}

on(event, callback) {
  if (!this.listeners.has(event)) {
    this.listeners.set(event, []);
  }
  this.listeners.get(event).push(callback);
}

off(event, callback) {
  const callbacks = this.listeners.get(event);
  if (callbacks) {
    const filtered = callbacks.filter(cb => cb !== callback);
  }
}
```

```

        this.listeners.set(event, filtered);
    }
}

emit(event, data) {
    const callbacks = this.listeners.get(event);
    if (callbacks) {
        callbacks.forEach(callback => callback(data));
    }
}

disconnect() {
    if (this.ws) {
        this.ws.close();
        this.ws = null;
    }
}
}

// Create global WebSocket client
const ws = new WebSocketClient('wss://api.example.com/ws');
ws.connect();

```

11.5.2 Real-Time Chat Component

```

class ChatRoom extends LarcComponent {
    constructor() {
        super();
        this.messages = [];
        this.connected = false;

        this.handleMessage = this.handleMessage.bind(this);
        this.handleConnected = this.handleConnected.bind(this);
        this.handleDisconnected = this.handleDisconnected.bind(this);
    }

    onMount() {
        ws.on('message', this.handleMessage);
        ws.on('connected', this.handleConnected);
        ws.on('disconnected', this.handleDisconnected);

        // Request message history
        ws.send('get_history', { room: this.roomId });
    }

    onUnmount() {
        ws.off('message', this.handleMessage);
    }
}

```

```
ws.off('connected', this.handleConnected);
ws.off('disconnected', this.handleDisconnected);
}

handleMessage(message) {
  this.messages.push(message);
  this.render();
  this.scrollToBottom();
}

handleConnected() {
  this.connected = true;
  this.render();
}

handleDisconnected() {
  this.connected = false;
  this.render();
}

sendMessage(event) {
  event.preventDefault();

  const input = this.querySelector('input[name="message"]');
  const message = input.value.trim();

  if (!message) {
    return;
  }

  ws.send('message', {
    room: this.roomId,
    text: message,
    timestamp: Date.now()
  });

  input.value = '';
}

scrollToBottom() {
  const container = this.querySelector('.messages');
  if (container) {
    container.scrollTop = container.scrollHeight;
  }
}

template() {
```

```

return `
  <div class="chat-room">
    <div class="status ${this.connected ? 'connected' : 'disconnected'}">
      ${this.connected ? 'Connected' : 'Disconnected'}
    </div>

    <div class="messages">
      ${this.messages.map(msg => `
        <div class="message">
          <span class="author">${msg.author}</span>
          <span class="text">${msg.text}</span>
          <span class="time">${this.formatTime(msg.timestamp)}</span>
        </div>
      `).join('')}
    </div>

    <form onsubmit="this.sendMessage(event)">
      <input
        type="text"
        name="message"
        placeholder="Type a message..."
        ?disabled="${!this.connected}">
      <button type="submit" ?disabled="${!this.connected}">
        Send
      </button>
    </form>
  </div>
`;
}

formatTime(timestamp) {
  return new Date(timestamp).toLocaleTimeString();
}
}

```

11.6 Server-Sent Events (SSE)

SSE provides one-way real-time communication from server to client—simpler than WebSockets but perfect for live updates, notifications, and progress tracking.

11.6.1 SSE Client

```

class SSEClient {
  constructor(url) {
    this.url = url;
    this.eventSource = null;
  }
}

```

```

    this.listeners = new Map();
  }

  connect() {
    this.eventSource = new EventSource(this.url);

    this.eventSource.onopen = () => {
      console.log('SSE connected');
      this.emit('connected');
    };

    this.eventSource.onerror = (error) => {
      console.error('SSE error:', error);
      this.emit('error', error);

      if (this.eventSource.readyState === EventSource.CLOSED) {
        this.emit('disconnected');
      }
    };

    this.eventSource.onmessage = (event) => {
      try {
        const data = JSON.parse(event.data);
        this.emit('message', data);
      } catch (error) {
        console.error('Failed to parse SSE data:', error);
      }
    };
  }

  on(event, callback) {
    if (!this.listeners.has(event)) {
      this.listeners.set(event, []);
    }
    this.listeners.get(event).push(callback);

    // Subscribe to custom event types
    if (event !== 'connected' && event !== 'error' && event !== 'disconnected' && event !== 'message') {
      this.eventSource?.addEventListener(event, (e) => {
        try {
          const data = JSON.parse(e.data);
          callback(data);
        } catch (error) {
          callback(e.data);
        }
      });
    }
  }

```

```

}

off(event, callback) {
  const callbacks = this.listeners.get(event);
  if (callbacks) {
    const filtered = callbacks.filter(cb => cb !== callback);
    this.listeners.set(event, filtered);
  }
}

emit(event, data) {
  const callbacks = this.listeners.get(event);
  if (callbacks) {
    callbacks.forEach(callback => callback(data));
  }
}

disconnect() {
  if (this.eventSource) {
    this.eventSource.close();
    this.eventSource = null;
  }
}
}

```

11.6.2 Live Notifications

```

class NotificationCenter extends LarcComponent {
  constructor() {
    super();
    this.notifications = [];
    this.sse = new SSEClient('/api/notifications/stream');

    this.handleNotification = this.handleNotification.bind(this);
  }

  onMount() {
    this.sse.on('notification', this.handleNotification);
    this.sse.connect();
  }

  onUnmount() {
    this.sse.off('notification', this.handleNotification);
    this.sse.disconnect();
  }

  handleNotification(notification) {

```

```

    this.notifications.unshift(notification);

    // Keep only last 50 notifications
    if (this.notifications.length > 50) {
        this.notifications = this.notifications.slice(0, 50);
    }

    this.render();
    this.showToast(notification);
}

showToast(notification) {
    // Show temporary toast notification
    const toast = document.createElement('div');
    toast.className = 'toast';
    toast.textContent = notification.message;
    document.body.appendChild(toast);

    setTimeout(() => {
        toast.classList.add('fade-out');
        setTimeout(() => toast.remove(), 300);
    }, 3000);
}

dismissNotification(id) {
    this.notifications = this.notifications.filter(n => n.id !== id);
    this.render();

    // Mark as read on server
    api.post(`/notifications/${id}/read`);
}

template() {
    return `
        <div class="notification-center">
            <h2>Notifications</h2>

            ${this.notifications.length === 0 ? `
                <p class="empty">No notifications</p>
            ` : `
                <ul class="notification-list">
                    ${this.notifications.map(notif => `
                        <li class="notification ${notif.read ? 'read' : 'unread'}">
                            <div class="content">
                                <strong>${notif.title}</strong>
                                <p>${notif.message}</p>
                                <time>${this.formatTime(notif.timestamp)}</time>
                    `}
                `}
                </ul>
            `}
        `;
}

```

```

        </div>
        <button onclick="this.dismissNotification('${notif.id}')">
            Dismiss
        </button>
    </li>
    `).join('')
</ul>
`}
</div>
`;
}

formatTime(timestamp) {
    const date = new Date(timestamp);
    const now = Date.now();
    const diff = now - date;

    if (diff < 60000) {
        return 'Just now';
    } else if (diff < 3600000) {
        return `${Math.floor(diff / 60000)}m ago`;
    } else if (diff < 86400000) {
        return `${Math.floor(diff / 3600000)}h ago`;
    } else {
        return date.toLocaleDateString();
    }
}
}
}

```

11.7 Request Cancellation

Long-running requests should be cancellable to avoid wasting resources and confusing users.

11.7.1 AbortController

```

class SearchComponent extends LarcComponent {
    constructor() {
        super();
        this.query = '';
        this.results = [];
        this.searching = false;
        this.abortController = null;
    }

    async handleSearch(event) {
        this.query = event.target.value;
    }
}

```

```

    // Cancel previous search
    if (this.abortController) {
        this.abortController.abort();
    }

    if (!this.query) {
        this.results = [];
        this.render();
        return;
    }

    this.searching = true;
    this.render();

    // Create new abort controller
    this.abortController = new AbortController();

    try {
        const response = await fetch(
            `/api/search?q=${encodeURIComponent(this.query)}`,
            { signal: this.abortController.signal }
        );

        this.results = await response.json();
    } catch (error) {
        if (error.name === 'AbortError') {
            console.log('Search cancelled');
            return;
        }
        console.error('Search failed:', error);
    } finally {
        this.searching = false;
        this.abortController = null;
        this.render();
    }
}

template() {
    return `
        <div class="search">
            <input
                type="search"
                placeholder="Search..."
                value="${this.query}"
                oninput="this.handleSearch(event)">

            ${this.searching ? '<div class="spinner"></div>' : ''}
    `;
}

```

```

        <ul class="results">
          ${this.results.map(result => `
            <li>${result.title}</li>
          `).join('')}
        </ul>
      </div>
    `;
  }
}

```

11.8 Putting It All Together

Let's create a complete data layer that combines all these concepts:

```

// data-layer.js
class DataLayer {
  constructor() {
    this.api = new RetryableAPIClient('/api');
    this.cache = new PersistentCache();
    this.ws = null;
    this.sse = null;
  }

  // REST API methods
  async getProducts(options = {}) {
    return this.api.get('/products', options);
  }

  async getProduct(id) {
    return this.api.get(`/products/${id}`);
  }

  async createProduct(data) {
    const product = await this.api.post('/products', data);
    this.cache.invalidatePattern('/products');
    return product;
  }

  async updateProduct(id, data) {
    const product = await this.api.put(`/products/${id}`, data);
    this.cache.invalidate(`/products/${id}`);
    this.cache.invalidatePattern('/products');
    return product;
  }

  async deleteProduct(id) {
    await this.api.delete(`/products/${id}`);
  }
}

```

```
this.cache.invalidate(`/products/${id}`);
this.cache.invalidatePattern('/products');
}

// WebSocket methods
connectWebSocket(url) {
  this.ws = new WebSocketClient(url);
  this.ws.connect();
  return this.ws;
}

// SSE methods
subscribeToNotifications(callback) {
  if (!this.sse) {
    this.sse = new SSEClient('/api/notifications/stream');
    this.sse.connect();
  }
  this.sse.on('notification', callback);
}

unsubscribeFromNotifications(callback) {
  if (this.sse) {
    this.sse.off('notification', callback);
  }
}

// Cleanup
destroy() {
  this.ws?.disconnect();
  this.sse?.disconnect();
  this.cache.clear();
}
}

// Export singleton instance
export const dataLayer = new DataLayer();
```

Data fetching is the nervous system of your application—it connects your UI to the outside world and keeps everything in sync. With proper error handling, intelligent caching, and real-time communication, you can build applications that feel fast, reliable, and responsive, even when the network isn't cooperating. The patterns in this chapter will help you navigate the chaos of distributed systems and emerge with applications that users can actually depend on.

And with that, we've covered the essential patterns for building robust, maintainable applications with LARC. From routing to forms to data fetching, you now have the tools to create web applications that don't just work—they work well.

Chapter 12

Authentication and Authorization

In which we learn to keep the riffraff out, manage who gets to do what, and discover that security is less like a lock and more like an onion—layered, sometimes makes you cry, and absolutely essential.

Authentication and authorization are the bouncer and the VIP list of your application. Authentication answers “who are you?” while authorization answers “what are you allowed to do?” Get either wrong, and you’ll either lock out legitimate users or let chaos agents run wild through your carefully constructed digital empire.

In this chapter, we’ll explore how to implement robust authentication and authorization in LARC applications, from JWT tokens to role-based access control, all while maintaining the framework’s philosophy of explicit, testable, and maintainable code.

12.1 Understanding Authentication vs. Authorization

Before we dive into implementation, let’s clarify the distinction that trips up even experienced developers:

Authentication is proof of identity. When you show your driver’s license at airport security, that’s authentication. You’re proving you are who you claim to be.

Authorization is proof of permission. When you try to board the plane, the gate agent checks if you have a ticket for *this* flight. That’s authorization—verifying you’re allowed to do the specific thing you’re attempting.

In LARC applications, we typically handle authentication through JWT (JSON Web Tokens) and authorization through role-based or permission-based access control. Let’s build both systems from the ground up.

12.2 JWT Authentication: The Token Economy

JWT tokens are like those “Hello, My Name Is” stickers, except they’re cryptographically signed so people can’t just write whatever they want. A JWT contains claims about the user (their ID, username, roles) and a signature that proves the token hasn’t been tampered with.

12.2.1 Creating an Authentication Service

Let's build a comprehensive authentication service that handles login, token generation, and verification:

```
// services/auth.ts
import { api } from '@larc/lib';

interface LoginCredentials {
  username: string;
  password: string;
}

interface AuthTokens {
  accessToken: string;
  refreshToken: string;
  expiresIn: number;
}

interface UserClaims {
  userId: string;
  username: string;
  roles: string[];
  permissions: string[];
}

interface AuthState {
  isAuthenticated: boolean;
  user: UserClaims | null;
  tokens: AuthTokens | null;
}

// Storage keys
const STORAGE_KEYS = {
  ACCESS_TOKEN: 'auth.accessToken',
  REFRESH_TOKEN: 'auth.refreshToken',
  USER_DATA: 'auth.userData'
} as const;

class AuthenticationService {
  private state: AuthState = {
    isAuthenticated: false,
    user: null,
    tokens: null
  };

  // Initialize from stored tokens
  async initialize(): Promise<boolean> {
```

```

const accessToken = localStorage.getItem(STORAGE_KEYS.ACCESS_TOKEN);
const refreshToken = localStorage.getItem(STORAGE_KEYS.REFRESH_TOKEN);
const userData = localStorage.getItem(STORAGE_KEYS.USER_DATA);

if (!accessToken || !userData) {
  return false;
}

try {
  // Verify the token is still valid
  const user = JSON.parse(userData) as UserClaims;
  const isValid = await this.verifyToken(accessToken);

  if (isValid) {
    this.state = {
      isAuthenticated: true,
      user,
      tokens: {
        accessToken,
        refreshToken: refreshToken || '',
        expiresIn: this.getTokenExpiry(accessToken)
      }
    };
    return true;
  }

  // Token invalid, try refresh
  if (refreshToken) {
    return await this.refreshAccessToken(refreshToken);
  }

  // Can't authenticate, clear everything
  this.clearAuth();
  return false;
} catch (error) {
  console.error('Auth initialization failed:', error);
  this.clearAuth();
  return false;
}

// Login with credentials
async login(credentials: LoginCredentials): Promise<boolean> {
  try {
    const response = await api.post<AuthTokens & { user: UserClaims }>(
      '/auth/login',
      credentials
    );
  }
}

```

```

    );

    const { accessToken, refreshToken, expiresIn, user } = response;

    // Store tokens securely
    localStorage.setItem(STORAGE_KEYS.ACCESS_TOKEN, accessToken);
    localStorage.setItem(STORAGE_KEYS.REFRESH_TOKEN, refreshToken);
    localStorage.setItem(STORAGE_KEYS.USER_DATA, JSON.stringify(user));

    // Update state
    this.state = {
      isAuthenticated: true,
      user,
      tokens: { accessToken, refreshToken, expiresIn }
    };

    // Set up automatic token refresh
    this.scheduleTokenRefresh(expiresIn);

    return true;
  } catch (error) {
    console.error('Login failed:', error);
    return false;
  }
}

// Logout and clear all auth data
logout(): void {
  this.clearAuth();
  // Optionally call backend to invalidate tokens
  api.post('/auth/logout', {
    refreshToken: this.state.tokens?.refreshToken
  }).catch(err => console.error('Logout notification failed:', err));
}

// Verify token validity
private async verifyToken(token: string): Promise<boolean> {
  try {
    await api.get('/auth/verify', {
      headers: { Authorization: `Bearer ${token}` }
    });
    return true;
  } catch {
    return false;
  }
}

```

```

// Refresh the access token
private async refreshAccessToken(refreshToken: string): Promise<boolean> {
  try {
    const response = await api.post<AuthTokens>('/auth/refresh', {
      refreshToken
    });

    const { accessToken, refreshToken: newRefreshToken, expiresIn } = response;

    // Update stored tokens
    localStorage.setItem(STORAGE_KEYS.ACCESS_TOKEN, accessToken);
    localStorage.setItem(STORAGE_KEYS.REFRESH_TOKEN, newRefreshToken);

    // Update state
    if (this.state.tokens) {
      this.state.tokens = { accessToken, refreshToken: newRefreshToken, expiresIn };
    }

    this.scheduleTokenRefresh(expiresIn);
    return true;
  } catch (error) {
    console.error('Token refresh failed:', error);
    this.clearAuth();
    return false;
  }
}

// Schedule automatic token refresh before expiry
private scheduleTokenRefresh(expiresIn: number): void {
  // Refresh 5 minutes before expiry
  const refreshTime = (expiresIn - 300) * 1000;

  setTimeout(async () => {
    const refreshToken = this.state.tokens?.refreshToken;
    if (refreshToken) {
      await this.refreshAccessToken(refreshToken);
    }
  }, refreshTime);
}

// Extract expiry time from JWT
private getTokenExpiry(token: string): number {
  try {
    const payload = JSON.parse(atob(token.split('.')[1]));
    return payload.exp;
  } catch {
    return 0;
  }
}

```

```

    }
  }

  // Clear all authentication data
  private clearAuth(): void {
    localStorage.removeItem(STORAGE_KEYS.ACCESS_TOKEN);
    localStorage.removeItem(STORAGE_KEYS.REFRESH_TOKEN);
    localStorage.removeItem(STORAGE_KEYS.USER_DATA);

    this.state = {
      isAuthenticated: false,
      user: null,
      tokens: null
    };
  }

  // Get current authentication state
  getState(): Readonly<AuthState> {
    return { ...this.state };
  }

  // Get access token for API requests
  getAccessToken(): string | null {
    return this.state.tokens?.accessToken || null;
  }

  // Check if user has specific role
  hasRole(role: string): boolean {
    return this.state.user?.roles.includes(role) || false;
  }

  // Check if user has specific permission
  hasPermission(permission: string): boolean {
    return this.state.user?.permissions.includes(permission) || false;
  }
}

// Export singleton instance
export const authService = new AuthenticationService();

```

This service handles the complete authentication lifecycle: initialization from stored tokens, login with automatic token refresh scheduling, logout with cleanup, and convenient methods for checking authentication state.

12.2.2 Securing API Requests

Now let's create an API interceptor that automatically adds authentication tokens to requests:

```
// api/auth-interceptor.ts
import { api } from '@larc/lib';
import { authService } from '../services/auth';

// Add authentication header to all requests
api.interceptors.request.use(async (config) => {
  const token = authService.getAccessToken();

  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }

  return config;
});

// Handle 401 responses by refreshing token
api.interceptors.response.use(
  (response) => response,
  async (error) => {
    const originalRequest = error.config;

    // If 401 and we haven't already retried
    if (error.response?.status === 401 && !originalRequest._retry) {
      originalRequest._retry = true;

      try {
        // Try to refresh the token
        const state = authService.getState();
        if (state.tokens?.refreshToken) {
          await authService.refreshAccessToken(state.tokens.refreshToken);

          // Retry original request with new token
          const newToken = authService.getAccessToken();
          originalRequest.headers.Authorization = `Bearer ${newToken}`;
          return api(originalRequest);
        }
      } catch (refreshError) {
        // Refresh failed, logout user
        authService.logout();
        window.location.href = '/login';
        return Promise.reject(refreshError);
      }
    }

    return Promise.reject(error);
  }
);
```

This interceptor automatically adds the Bearer token to outgoing requests and handles 401 Unauthorized responses by attempting to refresh the token and retry the request—a pattern that keeps users logged in seamlessly.

12.3 Protected Routes and Navigation Guards

Authentication means nothing if users can still access protected pages by typing URLs directly. Let's build a routing system that enforces authentication:

```
// components/protected-route.ts
import { html, define, Component } from '@larc/lib';
import { authService } from '../services/auth';

interface ProtectedRouteProps {
  requiredRoles?: string[];
  requiredPermissions?: string[];
  fallbackPath?: string;
}

class ProtectedRoute extends Component<ProtectedRouteProps> {
  static tagName = 'protected-route';

  connectedCallback() {
    super.connectedCallback();
    this.checkAccess();
  }

  private checkAccess() {
    const state = authService.getState();

    // Check authentication
    if (!state.isAuthenticated) {
      this.redirectToLogin();
      return;
    }

    // Check roles if specified
    const { requiredRoles, requiredPermissions } = this.props;

    if (requiredRoles?.length) {
      const hasRole = requiredRoles.some(role =>
        authService.hasRole(role)
      );
      if (!hasRole) {
        this.redirectToForbidden();
        return;
      }
    }
  }
}
```

```

    // Check permissions if specified
    if (requiredPermissions?.length) {
      const hasPermission = requiredPermissions.every(permission =>
        authService.hasPermission(permission)
      );
      if (!hasPermission) {
        this.redirectToForbidden();
        return;
      }
    }
  }

  private redirectToLogin() {
    const currentPath = window.location.pathname;
    window.location.href = `/login?redirect=${encodeURIComponent(currentPath)}`;
  }

  private redirectToForbidden() {
    const { fallbackPath = '/forbidden' } = this.props;
    window.location.href = fallbackPath;
  }

  render() {
    const state = authService.getState();

    if (!state.isAuthenticated) {
      return html`<div>Redirecting to login...</div>`;
    }

    // Render children if authorized
    return html`<slot></slot>`;
  }
}

define(ProtectedRoute);

```

Use it in your application like this:

```

// app.ts
import { html, define, Component } from '@larc/lib';
import './components/protected-route';

class App extends Component {
  render() {
    return html`
      <nav-bar></nav-bar>
    `;
  }
}

```

```

<router-outlet>
  <!-- Public route -->
  <route-handler path="/" component="home-page"></route-handler>
  <route-handler path="/login" component="login-page"></route-handler>

  <!-- Protected route - requires authentication -->
  <route-handler path="/dashboard">
    <protected-route>
      <dashboard-page></dashboard-page>
    </protected-route>
  </route-handler>

  <!-- Protected route - requires admin role -->
  <route-handler path="/admin">
    <protected-route requiredRoles="$[['admin']]">
      <admin-panel></admin-panel>
    </protected-route>
  </route-handler>

  <!-- Protected route - requires specific permission -->
  <route-handler path="/reports">
    <protected-route requiredPermissions="$[['reports.view']]">
      <reports-page></reports-page>
    </protected-route>
  </route-handler>
</router-outlet>
`
;
}
}

define(App);

```

12.4 Role-Based Access Control (RBAC)

RBAC is like organizing your office with different colored keycards. Some doors everyone can open (the break room), some require special access (the server room), and some are only for the CEO (the executive washroom with the good soap).

12.4.1 Designing a Flexible RBAC System

Here's a comprehensive RBAC implementation that supports hierarchical roles and fine-grained permissions:

```

// services/authorization.ts
interface Permission {
  resource: string; // e.g., 'users', 'reports', 'settings'
  action: string;   // e.g., 'create', 'read', 'update', 'delete'
}

```

```

    scope?: string;    // e.g., 'own', 'team', 'all'
}

interface Role {
    id: string;
    name: string;
    permissions: Permission[];
    inherits?: string[]; // Inherit permissions from other roles
}

const ROLES: Record<string, Role> = {
    guest: {
        id: 'guest',
        name: 'Guest',
        permissions: [
            { resource: 'content', action: 'read', scope: 'public' }
        ]
    },

    user: {
        id: 'user',
        name: 'User',
        inherits: ['guest'],
        permissions: [
            { resource: 'profile', action: 'read', scope: 'own' },
            { resource: 'profile', action: 'update', scope: 'own' },
            { resource: 'content', action: 'create', scope: 'own' },
            { resource: 'content', action: 'update', scope: 'own' },
            { resource: 'content', action: 'delete', scope: 'own' }
        ]
    },

    moderator: {
        id: 'moderator',
        name: 'Moderator',
        inherits: ['user'],
        permissions: [
            { resource: 'content', action: 'update', scope: 'all' },
            { resource: 'content', action: 'delete', scope: 'all' },
            { resource: 'reports', action: 'read', scope: 'all' },
            { resource: 'reports', action: 'update', scope: 'all' }
        ]
    },

    admin: {
        id: 'admin',
        name: 'Administrator',

```

```

    inherits: ['moderator'],
    permissions: [
      { resource: 'users', action: 'create' },
      { resource: 'users', action: 'read' },
      { resource: 'users', action: 'update' },
      { resource: 'users', action: 'delete' },
      { resource: 'settings', action: 'read' },
      { resource: 'settings', action: 'update' }
    ]
  }
};

class AuthorizationService {
  // Get all permissions for a role (including inherited)
  getPermissions(roleId: string): Permission[] {
    const role = ROLES[roleId];
    if (!role) return [];

    const permissions = [...role.permissions];

    // Recursively add inherited permissions
    if (role.inherits) {
      for (const inheritedRoleId of role.inherits) {
        permissions.push(...this.getPermissions(inheritedRoleId));
      }
    }

    return permissions;
  }

  // Check if role has specific permission
  hasPermission(
    roleIds: string[],
    resource: string,
    action: string,
    scope?: string
  ): boolean {
    // Get all permissions for all user's roles
    const allPermissions = roleIds.flatMap(roleId =>
      this.getPermissions(roleId)
    );

    // Check if any permission matches
    return allPermissions.some(permission =>
      permission.resource === resource &&
      permission.action === action &&
      (!scope || !permission.scope || permission.scope === scope)
    );
  }
}

```

```

    );
  }

  // Check if user can perform action on specific resource
  canAccess(
    roleIds: string[],
    resource: string,
    action: string,
    ownerId?: string,
    userId?: string
  ): boolean {
    // First check for 'all' scope
    if (this.hasPermission(roleIds, resource, action, 'all')) {
      return true;
    }

    // Then check for 'own' scope if resource belongs to user
    if (ownerId && userId && ownerId === userId) {
      return this.hasPermission(roleIds, resource, action, 'own');
    }

    // Finally check for permission without scope
    return this.hasPermission(roleIds, resource, action);
  }

  // Get user-friendly permission label
  getPermissionLabel(permission: Permission): string {
    const scopeText = permission.scope ? ` (${permission.scope})` : '';
    return `${permission.action} ${permission.resource}${scopeText}`;
  }
}

export const authz = new AuthorizationService();

```

12.4.2 Using Authorization in Components

Let's create a component that conditionally renders content based on permissions:

```

// components/authorized-content.ts
import { html, define, Component } from '@larc/lib';
import { authService } from '../services/auth';
import { authz } from '../services/authorization';

interface AuthorizedContentProps {
  resource: string;
  action: string;
  fallback?: string;
}

```

```

class AuthorizedContent extends Component<AuthorizedContentProps> {
  static tagName = 'authorized-content';

  private isAuthorized(): boolean {
    const state = authService.getState();
    if (!state.user) return false;

    const { resource, action } = this.props;
    return authz.hasPermission(state.user.roles, resource, action);
  }

  render() {
    if (this.isAuthorized()) {
      return html`<slot></slot>`;
    }

    const { fallback } = this.props;
    if (fallback) {
      return html`<div class="unauthorized">${fallback}</div>`;
    }

    return html``;
  }
}

define(AuthorizedContent);

// Usage example:
// <authorized-content resource="users" action="delete">
//   <button onclick="${this.deleteUser}">Delete User</button>
// </authorized-content>

```

12.5 Session Management Best Practices

Session management is the art of remembering who users are across requests without making them log in every five seconds or leaving security holes big enough to drive a truck through.

12.5.1 Implementing Secure Session Storage

```

// utils/secure-storage.ts
class SecureStorage {
  private readonly prefix = '__secure__';

  // Store sensitive data with encryption (in production, use Web Crypto API)
  setSecure(key: string, value: string): void {
    try {

```

```

    // In production, encrypt the value before storing
    const encrypted = this.encrypt(value);
    sessionStorage.setItem(`${this.prefix}${key}`, encrypted);
  } catch (error) {
    console.error('Failed to store secure data:', error);
  }
}

getSecure(key: string): string | null {
  try {
    const encrypted = sessionStorage.getItem(`${this.prefix}${key}`);
    if (!encrypted) return null;
    return this.decrypt(encrypted);
  } catch (error) {
    console.error('Failed to retrieve secure data:', error);
    return null;
  }
}

removeSecure(key: string): void {
  sessionStorage.removeItem(`${this.prefix}${key}`);
}

clearSecure(): void {
  const keys = Object.keys(sessionStorage);
  keys.forEach(key => {
    if (key.startsWith(this.prefix)) {
      sessionStorage.removeItem(key);
    }
  });
}

// Simple encryption (use Web Crypto API in production!)
private encrypt(value: string): string {
  // This is a placeholder - use proper encryption in production
  return btoa(value);
}

private decrypt(value: string): string {
  // This is a placeholder - use proper decryption in production
  return atob(value);
}

export const secureStorage = new SecureStorage();

```

12.5.2 Session Timeout and Activity Tracking

Implement automatic session timeout to protect against abandoned sessions:

```
// services/session-manager.ts
import { authService } from './auth';

interface SessionConfig {
  timeoutMinutes: number;
  warningMinutes: number;
}

class SessionManager {
  private timeoutId: number | null = null;
  private warningId: number | null = null;
  private lastActivity: number = Date.now();

  constructor(private config: SessionConfig) {
    this.setupActivityListeners();
  }

  // Start session monitoring
  start(): void {
    this.resetTimeout();
  }

  // Stop session monitoring
  stop(): void {
    if (this.timeoutId) clearTimeout(this.timeoutId);
    if (this.warningId) clearTimeout(this.warningId);
  }

  // Reset timeout on user activity
  private resetTimeout(): void {
    this.lastActivity = Date.now();

    // Clear existing timers
    if (this.timeoutId) clearTimeout(this.timeoutId);
    if (this.warningId) clearTimeout(this.warningId);

    // Set warning timer
    const warningMs = this.config.warningMinutes * 60 * 1000;
    this.warningId = window.setTimeout(() => {
      this.showTimeoutWarning();
    }, warningMs);

    // Set timeout timer
    const timeoutMs = this.config.timeoutMinutes * 60 * 1000;
```

```
this.timeoutId = window.setTimeout(() => {
  this.handleTimeout();
}, timeoutMs);
}

// Setup listeners for user activity
private setupActivityListeners(): void {
  const events = ['mousedown', 'keydown', 'scroll', 'touchstart'];

  events.forEach(event => {
    document.addEventListener(event, () => {
      // Only reset if user is authenticated
      if (authService.getState().isAuthenticated) {
        this.resetTimeout();
      }
    }, { passive: true });
  });
}

// Show warning before timeout
private showTimeoutWarning(): void {
  const remainingMinutes = this.config.timeoutMinutes - this.config.warningMinutes;

  // Dispatch custom event that UI can listen to
  window.dispatchEvent(new CustomEvent('session-warning', {
    detail: { remainingMinutes }
  }));
}

// Handle session timeout
private handleTimeout(): void {
  authService.logout();

  // Dispatch timeout event
  window.dispatchEvent(new CustomEvent('session-timeout'));

  // Redirect to login
  window.location.href = '/login?reason=timeout';
}

// Get time until timeout
getTimeRemaining(): number {
  const elapsed = Date.now() - this.lastActivity;
  const timeoutMs = this.config.timeoutMinutes * 60 * 1000;
  return Math.max(0, timeoutMs - elapsed);
}
}
```

```
// Initialize with 30-minute timeout, 5-minute warning
export const sessionManager = new SessionManager({
  timeoutMinutes: 30,
  warningMinutes: 25
});
```

12.6 Security Best Practices

Security is like flossing—everyone knows they should do it, but it’s easy to skip until problems arise. Let’s make sure you’re following best practices.

12.6.1 Input Validation and Sanitization

Never trust user input. Ever. Here’s a validation utility:

```
// utils/validation.ts
export const validators = {
  email: (value: string): boolean => {
    const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return regex.test(value);
  },

  password: (value: string): { valid: boolean; errors: string[] } => {
    const errors: string[] = [];

    if (value.length < 8) {
      errors.push('Password must be at least 8 characters');
    }
    if (!/[A-Z]/.test(value)) {
      errors.push('Password must contain an uppercase letter');
    }
    if (!/[a-z]/.test(value)) {
      errors.push('Password must contain a lowercase letter');
    }
    if (!/[0-9]/.test(value)) {
      errors.push('Password must contain a number');
    }
    if (!/^[A-Za-z0-9]/.test(value)) {
      errors.push('Password must contain a special character');
    }

    return { valid: errors.length === 0, errors };
  },

  sanitize: (value: string): string => {
    return value
      .replace(/<>/g, '') // Remove angle brackets
  },
```

```

        .replace(/javascript:/gi, '') // Remove javascript: protocol
        .trim();
    }
};

```

12.6.2 CSRF Protection

Protect against Cross-Site Request Forgery:

```

// utils/csrf.ts
class CSRFProtection {
    private token: string = '';

    // Generate CSRF token
    generateToken(): string {
        this.token = this.randomString(32);
        sessionStorage.setItem('csrf-token', this.token);
        return this.token;
    }

    // Get current token
    getToken(): string {
        return this.token || sessionStorage.getItem('csrf-token') || '';
    }

    // Validate token
    validateToken(token: string): boolean {
        return token === this.getToken();
    }

    // Add token to request headers
    addToHeaders(headers: Record<string, string>): Record<string, string> {
        return {
            ...headers,
            'X-CSRF-Token': this.getToken()
        };
    }

    private randomString(length: number): string {
        const chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
        let result = '';
        const array = new Uint8Array(length);
        crypto.getRandomValues(array);

        for (let i = 0; i < length; i++) {
            result += chars[array[i] % chars.length];
        }
    }
}

```

```

    return result;
  }
}

export const csrf = new CSRFProtection();

```

12.7 Putting It All Together: A Complete Login Flow

Let's build a complete login component that demonstrates everything we've learned:

```

// components/login-form.ts
import { html, define, Component } from '@larc/lib';
import { authService } from '../services/auth';
import { validators } from '../utils/validation';
import { csrf } from '../utils/csrf';

interface LoginFormState {
  username: string;
  password: string;
  rememberMe: boolean;
  isLoading: boolean;
  error: string | null;
  fieldErrors: Record<string, string>;
}

class LoginForm extends Component {
  static tagName = 'login-form';

  state: LoginFormState = {
    username: '',
    password: '',
    rememberMe: false,
    isLoading: false,
    error: null,
    fieldErrors: {}
  };

  connectedCallback() {
    super.connectedCallback();
    csrf.generateToken(); // Generate CSRF token on mount
  }

  private validate(): boolean {
    const errors: Record<string, string> = {};

    if (!this.state.username) {
      errors.username = 'Username is required';
    }
  }

```

```
}

if (!this.state.password) {
  errors.password = 'Password is required';
}

this.setState({ fieldErrors: errors });
return Object.keys(errors).length === 0;
}

private async handleSubmit(e: Event) {
  e.preventDefault();

  if (!this.validate()) return;

  this.setState({ isLoading: true, error: null });

  try {
    const success = await authService.login({
      username: this.state.username,
      password: this.state.password
    });

    if (success) {
      // Get redirect URL from query params
      const params = new URLSearchParams(window.location.search);
      const redirect = params.get('redirect') || '/dashboard';
      window.location.href = redirect;
    } else {
      this.setState({
        error: 'Invalid username or password',
        isLoading: false
      });
    }
  } catch (error) {
    this.setState({
      error: 'An error occurred. Please try again.',
      isLoading: false
    });
  }
}

render() {
  const { username, password, rememberMe, isLoading, error, fieldErrors } = this.state;

  return html`
    <div class="login-container">
```

```

<form class="login-form" onsubmit="{this.handleSubmit}">
  <h2>Sign In</h2>

  ${error ? html`
    <div class="alert alert-error">
      ${error}
    </div>
  ` : ''}

  <div class="form-group">
    <label for="username">Username</label>
    <input
      type="text"
      id="username"
      value="{username}"
      oninput="{(e: Event) => this.setState({
        username: (e.target as HTMLInputElement).value
      })}"
      class="{fieldErrors.username ? 'error' : ''}"
      disabled="{isLoading}"
      autocomplete="username"
    />
    ${fieldErrors.username ? html`
      <span class="error-message">{fieldErrors.username}</span>
    ` : ''}
  </div>

  <div class="form-group">
    <label for="password">Password</label>
    <input
      type="password"
      id="password"
      value="{password}"
      oninput="{(e: Event) => this.setState({
        password: (e.target as HTMLInputElement).value
      })}"
      class="{fieldErrors.password ? 'error' : ''}"
      disabled="{isLoading}"
      autocomplete="current-password"
    />
    ${fieldErrors.password ? html`
      <span class="error-message">{fieldErrors.password}</span>
    ` : ''}
  </div>

  <div class="form-group checkbox">
    <label>

```

```

        <input
          type="checkbox"
          checked="{rememberMe}"
          onChange="{(e: Event) => this.setState({
            rememberMe: (e.target as HTMLInputElement).checked
          })}"
          disabled="{isLoading}"
        />
        Remember me
      </label>
    </div>

    <button
      type="submit"
      class="btn btn-primary"
      disabled="{isLoading}"
    >
      {isLoading ? 'Signing in...' : 'Sign In'}
    </button>

    <div class="form-footer">
      <a href="/forgot-password">Forgot password?</a>
    </div>
  </form>
</div>

<style>
.login-container {
  display: flex;
  justify-content: center;
  align-items: center;
  min-height: 100vh;
  padding: 1rem;
}

.login-form {
  width: 100%;
  max-width: 400px;
  padding: 2rem;
  border: 1px solid #ddd;
  border-radius: 8px;
  background: white;
}

.form-group {
  margin-bottom: 1rem;
}

```

```
.form-group label {
  display: block;
  margin-bottom: 0.5rem;
  font-weight: 500;
}

.form-group input[type="text"],
.form-group input[type="password"] {
  width: 100%;
  padding: 0.5rem;
  border: 1px solid #ddd;
  border-radius: 4px;
}

.form-group input.error {
  border-color: #dc3545;
}

.error-message {
  display: block;
  color: #dc3545;
  font-size: 0.875rem;
  margin-top: 0.25rem;
}

.alert {
  padding: 0.75rem;
  margin-bottom: 1rem;
  border-radius: 4px;
}

.alert-error {
  background: #f8d7da;
  color: #721c24;
  border: 1px solid #f5c6cb;
}

.btn {
  width: 100%;
  padding: 0.75rem;
  border: none;
  border-radius: 4px;
  font-size: 1rem;
  cursor: pointer;
}

.btn-primary {
```

```
        background: #007bff;
        color: white;
    }

    .btn:disabled {
        opacity: 0.6;
        cursor: not-allowed;
    }

    .form-footer {
        margin-top: 1rem;
        text-align: center;
    }
</style>
`
;
}
}

define(LoginForm);
```

12.8 What We've Learned

In this chapter, we've built a complete authentication and authorization system for LARC applications. You now know how to:

- Implement JWT-based authentication with automatic token refresh
- Create protected routes that enforce authentication and authorization
- Build a flexible role-based access control system
- Manage sessions securely with timeout and activity tracking
- Follow security best practices including input validation and CSRF protection
- Create production-ready login flows with proper error handling

Authentication and authorization are the foundation of application security. Get them right, and you'll sleep better at night knowing your users and their data are protected. Get them wrong, and you'll be explaining to your CEO why the company is on the front page of Hacker News for all the wrong reasons.

In the next chapter, we'll explore real-time features, where we'll need to authenticate WebSocket connections and authorize real-time events. Because what good is a secure application if it can't securely push updates to users in real-time?

Chapter 13

Real-time Features

In which we discover that the web doesn't have to reload every time something changes, explore the art of pushing data instead of polling, and learn that “real-time” doesn't mean “instantly”—it means “fast enough that users stop complaining.”

Real-time features have gone from “nice to have” to “why doesn't this update automatically?” The modern web expects live updates, collaborative editing, instant notifications, and data that refreshes faster than a TikTok feed. In this chapter, we'll build real-time features in LARC that are performant, reliable, and won't melt your servers or your users' browsers.

13.1 Understanding Real-time Communication

Before HTTP came along and ruined everything with its request-response pattern, computers communicated just fine by sending messages whenever they wanted. HTTP made us polite—the client asks nicely, and the server responds. But sometimes we want servers to speak when they have something to say, not just when asked.

We have three main tools for real-time communication on the web:

1. **WebSockets**: Full-duplex communication channels—both sides can talk whenever they want
2. **Server-Sent Events (SSE)**: One-way communication from server to client, simpler than WebSockets
3. **Polling**: The brute-force approach—asking “got anything new?” every few seconds

Additionally, we'll cover:

4. **BroadcastChannel**: Communication between tabs/windows in the same browser
5. **Web Workers**: Background processing for real-time data without blocking the UI

Let's build with all of these tools, starting with WebSockets.

13.2 WebSocket Integration: The Two-Way Street

WebSockets are like a phone call, while HTTP is like passing notes. Once the connection is established, both parties can send messages whenever they want without the overhead of setting up a new connection each time.

13.2.1 Building a WebSocket Client

Let's create a robust WebSocket client that handles connection lifecycle, reconnection, authentication, and message routing:

```
// services/websocket-client.ts
import { authService } from './auth';

interface WebSocketConfig {
  url: string;
  reconnectInterval?: number;
  maxReconnectAttempts?: number;
  heartbeatInterval?: number;
}

interface WebSocketMessage {
  type: string;
  payload: any;
  timestamp: number;
}

type MessageHandler = (payload: any) => void;

class WebSocketClient {
  private ws: WebSocket | null = null;
  private config: Required<WebSocketConfig>;
  private reconnectAttempts = 0;
  private reconnectTimeoutId: number | null = null;
  private heartbeatIntervalId: number | null = null;
  private messageHandlers = new Map<string, Set<MessageHandler>>();
  private isConnecting = false;

  constructor(config: WebSocketConfig) {
    this.config = {
      reconnectInterval: 5000,
      maxReconnectAttempts: 10,
      heartbeatInterval: 30000,
      ...config
    };
  }

  // Connect to WebSocket server
  async connect(): Promise<void> {
    if (this.ws?.readyState === WebSocket.OPEN) {
      console.log('Already connected');
      return;
    }
  }
```

```
if (this.isConnecting) {
  console.log('Connection in progress');
  return;
}

this.isConnecting = true;

try {
  // Get auth token
  const token = authService.getAccessToken();
  const url = token
    ? `${this.config.url}?token=${token}`
    : this.config.url;

  this.ws = new WebSocket(url);

  // Setup event handlers
  this.ws.onopen = () => this.handleOpen();
  this.ws.onmessage = (event) => this.handleMessage(event);
  this.ws.onerror = (error) => this.handleError(error);
  this.ws.onclose = (event) => this.handleClose(event);

  // Wait for connection
  await this.waitForConnection();
} finally {
  this.isConnecting = false;
}

// Wait for connection to open
private waitForConnection(): Promise<void> {
  return new Promise((resolve, reject) => {
    if (!this.ws) {
      reject(new Error('WebSocket not initialized'));
      return;
    }

    const timeout = setTimeout(() => {
      reject(new Error('Connection timeout'));
    }, 10000);

    this.ws.addEventListener('open', () => {
      clearTimeout(timeout);
      resolve();
    }, { once: true });

    this.ws.addEventListener('error', () => {
```

```

        clearTimeout(timeout);
        reject(new Error('Connection failed'));
    }, { once: true });
});
}

// Disconnect from WebSocket server
disconnect(): void {
    this.stopHeartbeat();
    this.stopReconnect();

    if (this.ws) {
        this.ws.close(1000, 'Client disconnect');
        this.ws = null;
    }
}

// Send message to server
send(type: string, payload: any): void {
    if (!this.isConnected()) {
        console.error('Cannot send message: not connected');
        return;
    }

    const message: WebSocketMessage = {
        type,
        payload,
        timestamp: Date.now()
    };

    this.ws!.send(JSON.stringify(message));
}

// Subscribe to messages of specific type
on(type: string, handler: MessageHandler): () => void {
    if (!this.messageHandlers.has(type)) {
        this.messageHandlers.set(type, new Set());
    }

    this.messageHandlers.get(type)!.add(handler);

    // Return unsubscribe function
    return () => {
        const handlers = this.messageHandlers.get(type);
        if (handlers) {
            handlers.delete(handler);
        }
    }
}

```

```
};  
}  
  
// Check if connected  
isConnected(): boolean {  
    return this.ws?.readyState === WebSocket.OPEN;  
}  
  
// Handle connection open  
private handleOpen(): void {  
    console.log('WebSocket connected');  
    this.reconnectAttempts = 0;  
    this.startHeartbeat();  
  
    // Notify listeners  
    this.notifyHandlers('connected', {});  
}  
  
// Handle incoming message  
private handleMessage(event: MessageEvent): void {  
    try {  
        const message = JSON.parse(event.data) as WebSocketMessage;  
  
        // Handle heartbeat response  
        if (message.type === 'pong') {  
            return;  
        }  
  
        // Notify type-specific handlers  
        this.notifyHandlers(message.type, message.payload);  
  
        // Notify global handlers  
        this.notifyHandlers('*', message);  
    } catch (error) {  
        console.error('Failed to parse WebSocket message:', error);  
    }  
}  
  
// Handle connection error  
private handleError(error: Event): void {  
    console.error('WebSocket error:', error);  
    this.notifyHandlers('error', { error });  
}  
  
// Handle connection close  
private handleClose(event: CloseEvent): void {  
    console.log('WebSocket closed:', event.code, event.reason);  
}
```

```

    this.stopHeartbeat();

    // Notify listeners
    this.notifyHandlers('disconnected', {
        code: event.code,
        reason: event.reason
    });

    // Attempt reconnection if not a normal close
    if (event.code !== 1000 && event.code !== 1001) {
        this.attemptReconnect();
    }
}

// Notify all handlers for a message type
private notifyHandlers(type: string, payload: any): void {
    const handlers = this.messageHandlers.get(type);
    if (handlers) {
        handlers.forEach(handler => {
            try {
                handler(payload);
            } catch (error) {
                console.error(`Handler error for type ${type}:`, error);
            }
        });
    }
}

// Start heartbeat to keep connection alive
private startHeartbeat(): void {
    this.stopHeartbeat();

    this.heartbeatIntervalId = window.setInterval(() => {
        if (this.isConnected()) {
            this.send('ping', {});
        }
    }, this.config.heartbeatInterval);
}

// Stop heartbeat
private stopHeartbeat(): void {
    if (this.heartbeatIntervalId !== null) {
        clearInterval(this.heartbeatIntervalId);
        this.heartbeatIntervalId = null;
    }
}

```

```

// Attempt to reconnect
private attemptReconnect(): void {
  if (this.reconnectAttempts >= this.config.maxReconnectAttempts) {
    console.error('Max reconnection attempts reached');
    this.notifyHandlers('reconnect-failed', {});
    return;
  }

  this.reconnectAttempts++;
  const delay = Math.min(
    this.config.reconnectInterval * this.reconnectAttempts,
    30000
  );

  console.log(`Reconnecting in ${delay}ms (attempt ${this.reconnectAttempts})`);

  this.reconnectTimeoutId = window.setTimeout(() => {
    console.log('Attempting to reconnect...');
    this.connect().catch(error => {
      console.error('Reconnection failed:', error);
    });
  }, delay);
}

// Stop reconnection attempts
private stopReconnect(): void {
  if (this.reconnectTimeoutId !== null) {
    clearTimeout(this.reconnectTimeoutId);
    this.reconnectTimeoutId = null;
  }
  this.reconnectAttempts = 0;
}

// Create and export singleton instance
export const wsClient = new WebSocketClient({
  url: process.env.WS_URL || 'ws://localhost:3000/ws'
});

```

13.2.2 Using WebSockets in Components

Now let's create a component that uses our WebSocket client to display real-time notifications:

```

// components/notification-feed.ts
import { html, define, Component } from '@larc/lib';
import { wsClient } from '../services/websocket-client';

interface Notification {

```

```

    id: string;
    type: 'info' | 'success' | 'warning' | 'error';
    title: string;
    message: string;
    timestamp: number;
  }

  interface NotificationFeedState {
    notifications: Notification[];
    isConnected: boolean;
  }

  class NotificationFeed extends Component {
    static tagName = 'notification-feed';

    state: NotificationFeedState = {
      notifications: [],
      isConnected: false
    };

    private unsubscribers: Array<() => void> = [];

    async connectedCallback() {
      super.connectedCallback();

      // Connect to WebSocket
      await wsClient.connect();

      // Subscribe to notification messages
      this.unsubscribers.push(
        wsClient.on('notification', (notification: Notification) => {
          this.addNotification(notification);
        })
      );

      // Subscribe to connection status
      this.unsubscribers.push(
        wsClient.on('connected', () => {
          this.setState({ isConnected: true });
        })
      );

      this.unsubscribers.push(
        wsClient.on('disconnected', () => {
          this.setState({ isConnected: false });
        })
      );
    }
  }

```

```

    // Update initial connection state
    this.setState({ isConnected: wsClient.isConnected() });
  }

  disconnectedCallback() {
    super.disconnectedCallback();

    // Unsubscribe from all messages
    this.unsubscribers.forEach(unsub => unsub());
    this.unsubscribers = [];
  }

  private addNotification(notification: Notification): void {
    this.setState({
      notifications: [notification, ...this.state.notifications].slice(0, 50)
    });

    // Auto-dismiss after 5 seconds
    setTimeout(() => {
      this.dismissNotification(notification.id);
    }, 5000);
  }

  private dismissNotification(id: string): void {
    this.setState({
      notifications: this.state.notifications.filter(n => n.id !== id)
    });
  }

  render() {
    const { notifications, isConnected } = this.state;

    return html`
      <div class="notification-feed">
        <div class="connection-status ${isConnected ? 'connected' : 'disconnected'}">
          ${isConnected ? '* Connected' : 'o Disconnected'}
        </div>

        <div class="notifications">
          ${notifications.map(notification => html`
            <div class="notification ${notification.type}" key="${notification.id}">
              <div class="notification-header">
                <strong>${notification.title}</strong>
                <button
                  class="dismiss"
                  onclick="${() => this.dismissNotification(notification.id)}"
                >

```

```

        x
      </button>
    </div>
    <div class="notification-body">
      ${notification.message}
    </div>
    <div class="notification-time">
      ${this.formatTime(notification.timestamp)}
    </div>
  </div>
</div>
`)}
</div>
</div>

<style>
.notification-feed {
  position: fixed;
  top: 1rem;
  right: 1rem;
  width: 320px;
  max-height: 80vh;
  overflow-y: auto;
  z-index: 1000;
}

.connection-status {
  padding: 0.5rem;
  margin-bottom: 0.5rem;
  border-radius: 4px;
  font-size: 0.875rem;
  text-align: center;
}

.connection-status.connected {
  background: #d4edda;
  color: #155724;
}

.connection-status.disconnected {
  background: #f8d7da;
  color: #721c24;
}

.notification {
  background: white;
  border-left: 4px solid;
  border-radius: 4px;

```

```

        padding: 1rem;
        margin-bottom: 0.5rem;
        box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
        animation: slideIn 0.3s ease-out;
    }

    .notification.info { border-color: #17a2b8; }
    .notification.success { border-color: #28a745; }
    .notification.warning { border-color: #ffc107; }
    .notification.error { border-color: #dc3545; }

    .notification-header {
        display: flex;
        justify-content: space-between;
        align-items: center;
        margin-bottom: 0.5rem;
    }

    .dismiss {
        background: none;
        border: none;
        font-size: 1.5rem;
        cursor: pointer;
        color: #999;
    }

    .notification-time {
        font-size: 0.75rem;
        color: #666;
        margin-top: 0.5rem;
    }

    @keyframes slideIn {
        from {
            transform: translateX(100%);
            opacity: 0;
        }
        to {
            transform: translateX(0);
            opacity: 1;
        }
    }
</style>
`;
}

private formatTime(timestamp: number): string {

```

```

    const date = new Date(timestamp);
    const now = new Date();
    const diff = now.getTime() - date.getTime();

    if (diff < 60000) return 'Just now';
    if (diff < 3600000) return `${Math.floor(diff / 60000)}m ago`;
    if (diff < 86400000) return `${Math.floor(diff / 3600000)}h ago`;
    return date.toLocaleDateString();
  }
}

define(NotificationFeed);

```

13.3 Server-Sent Events: One-Way Data Flow

Server-Sent Events (SSE) are perfect when you only need the server to push updates to the client. They're simpler than WebSockets, work over regular HTTP, and automatically reconnect when disconnected. Think of them as a fire hose of data from server to client.

13.3.1 Building an SSE Client

```

// services/sse-client.ts
interface SSEConfig {
  url: string;
  withCredentials?: boolean;
  reconnectDelay?: number;
}

type SSEEventHandler = (data: any) => void;

class SSEClient {
  private eventSource: EventSource | null = null;
  private config: Required<SSEConfig>;
  private eventHandlers = new Map<string, Set<SSEEventHandler>>();
  private isConnecting = false;

  constructor(config: SSEConfig) {
    this.config = {
      withCredentials: true,
      reconnectDelay: 3000,
      ...config
    };
  }

  // Connect to SSE endpoint
  connect(): void {

```

```

if (this.eventSource?.readyState === EventSource.OPEN) {
    console.log('Already connected to SSE');
    return;
}

if (this.isConnecting) {
    console.log('SSE connection in progress');
    return;
}

this.isConnecting = true;

try {
    this.eventSource = new EventSource(this.config.url, {
        withCredentials: this.config.withCredentials
    });

    this.eventSource.onopen = () => this.handleOpen();
    this.eventSource.onerror = (error) => this.handleError(error);

    // Listen for default message event
    this.eventSource.onmessage = (event) => {
        this.handleEvent('message', event.data);
    };
} finally {
    this.isConnecting = false;
}
}

// Disconnect from SSE endpoint
disconnect(): void {
    if (this.eventSource) {
        this.eventSource.close();
        this.eventSource = null;
    }
}

// Subscribe to named events
on(eventName: string, handler: SSEEventHandler): () => void {
    if (!this.eventHandlers.has(eventName)) {
        this.eventHandlers.set(eventName, new Set());
    }

    // Register event listener with EventSource
    if (this.eventSource && eventName !== 'message') {
        this.eventSource.addEventListener(eventName, (event: MessageEvent) => {
            this.handleEvent(eventName, event.data);
        });
    }
}

```

```

    }
}

this.eventHandlers.get(eventName)!.add(handler);

// Return unsubscribe function
return () => {
    const handlers = this.eventHandlers.get(eventName);
    if (handlers) {
        handlers.delete(handler);
    }
};
}

// Check if connected
isConnected(): boolean {
    return this.eventSource?.readyState === EventSource.OPEN;
}

// Handle connection open
private handleOpen(): void {
    console.log('SSE connected');
    this.notifyHandlers('connected', {});
}

// Handle error
private handleError(error: Event): void {
    console.error('SSE error:', error);

    if (this.eventSource?.readyState === EventSource.CLOSED) {
        console.log('SSE connection closed, reconnecting...');
        this.notifyHandlers('disconnected', {});

        // Reconnect after delay
        setTimeout(() => {
            this.connect();
        }, this.config.reconnectDelay);
    }
}

// Handle incoming event
private handleEvent(eventName: string, data: string): void {
    try {
        const parsed = JSON.parse(data);
        this.notifyHandlers(eventName, parsed);
    } catch {
        // If not JSON, pass raw data
    }
}

```

```

        this.notifyHandlers(eventName, data);
    }
}

// Notify all handlers for an event
private notifyHandlers(eventName: string, data: any): void {
    const handlers = this.eventHandlers.get(eventName);
    if (handlers) {
        handlers.forEach(handler => {
            try {
                handler(data);
            } catch (error) {
                console.error(`Handler error for event ${eventName}:`, error);
            }
        });
    }
}

export const sseClient = new SSEClient({
    url: '/api/events'
});

```

13.3.2 Live Activity Feed with SSE

Let's build a live activity feed that displays real-time updates using SSE:

```

// components/activity-feed.ts
import { html, define, Component } from '@larc/lib';
import { sseClient } from '../services/sse-client';

interface Activity {
    id: string;
    userId: string;
    userName: string;
    action: string;
    target: string;
    timestamp: number;
}

interface ActivityFeedState {
    activities: Activity[];
    isConnected: boolean;
}

class ActivityFeed extends Component {
    static tagName = 'activity-feed';

```

```
state: ActivityFeedState = {
  activities: [],
  isConnected: false
};

private unsubscribers: Array<() => void> = [];

connectedCallback() {
  super.connectedCallback();

  // Connect to SSE
  sseClient.connect();

  // Subscribe to activity events
  this.unsubscribers.push(
    sseClient.on('activity', (activity: Activity) => {
      this.addActivity(activity);
    })
  );

  // Subscribe to connection events
  this.unsubscribers.push(
    sseClient.on('connected', () => {
      this.setState({ isConnected: true });
    })
  );

  this.unsubscribers.push(
    sseClient.on('disconnected', () => {
      this.setState({ isConnected: false });
    })
  );

  // Load initial activities
  this.loadActivities();
}

disconnectedCallback() {
  super.disconnectedCallback();
  this.unsubscribers.forEach(unsub => unsub());
}

private async loadActivities(): Promise<void> {
  try {
    const response = await fetch('/api/activities?limit=20');
    const activities = await response.json();
    this.setState({ activities });
  }
}
```

```

    } catch (error) {
      console.error('Failed to load activities:', error);
    }
  }

  private addActivity(activity: Activity): void {
    // Add to beginning and limit to 100 items
    this.setState({
      activities: [activity, ...this.state.activities].slice(0, 100)
    });
  }

  render() {
    const { activities, isConnected } = this.state;

    return html`
      <div class="activity-feed">
        <div class="feed-header">
          <h3>Activity Feed</h3>
          <span class="status ${isConnected ? 'live' : 'offline'}">
            ${isConnected ? '* Live' : 'o Offline'}
          </span>
        </div>

        <div class="activities">
          ${activities.length === 0 ? html`
            <div class="empty">No recent activity</div>
          ` : activities.map(activity => html`
            <div class="activity-item" key="${activity.id}">
              <div class="activity-avatar">
                ${activity.userName.charAt(0).toUpperCase()}
              </div>
              <div class="activity-content">
                <div class="activity-text">
                  <strong>${activity.userName}</strong>
                  ${activity.action}
                  <em>${activity.target}</em>
                </div>
                <div class="activity-time">
                  ${this.formatTime(activity.timestamp)}
                </div>
              </div>
            </div>
          `)}
        </div>
      </div>
    `;
  }

```

```
<style>
  .activity-feed {
    background: white;
    border: 1px solid #ddd;
    border-radius: 8px;
    overflow: hidden;
  }

  .feed-header {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 1rem;
    border-bottom: 1px solid #ddd;
    background: #f8f9fa;
  }

  .feed-header h3 {
    margin: 0;
  }

  .status {
    font-size: 0.875rem;
    font-weight: 500;
  }

  .status.live { color: #28a745; }
  .status.offline { color: #6c757d; }

  .activities {
    max-height: 500px;
    overflow-y: auto;
  }

  .activity-item {
    display: flex;
    gap: 1rem;
    padding: 1rem;
    border-bottom: 1px solid #eee;
    animation: fadeIn 0.3s ease-out;
  }

  .activity-item:last-child {
    border-bottom: none;
  }

  .activity-avatar {
```

```

        width: 40px;
        height: 40px;
        border-radius: 50%;
        background: #007bff;
        color: white;
        display: flex;
        align-items: center;
        justify-content: center;
        font-weight: bold;
        flex-shrink: 0;
    }

    .activity-content {
        flex: 1;
    }

    .activity-text {
        margin-bottom: 0.25rem;
    }

    .activity-time {
        font-size: 0.75rem;
        color: #6c757d;
    }

    .empty {
        padding: 2rem;
        text-align: center;
        color: #6c757d;
    }

    @keyframes fadeIn {
        from { opacity: 0; transform: translateY(-10px); }
        to { opacity: 1; transform: translateY(0); }
    }
</style>
`;
}

private formatTime(timestamp: number): string {
    const date = new Date(timestamp);
    const now = new Date();
    const diff = now.getTime() - date.getTime();

    if (diff < 60000) return 'Just now';
    if (diff < 3600000) return `${Math.floor(diff / 60000)} minutes ago`;
    if (diff < 86400000) return `${Math.floor(diff / 3600000)} hours ago`;

```

```

    return date.toLocaleDateString();
  }
}

define(ActivityFeed);

```

13.4 BroadcastChannel: Cross-Tab Communication

BroadcastChannel lets different tabs and windows of your application communicate with each other. It's perfect for keeping UI state synchronized across multiple tabs—like ensuring all tabs show “logged out” when a user logs out in one tab.

13.4.1 Building a Tab Synchronization Service

```

// services/tab-sync.ts
interface SyncMessage {
  type: string;
  payload: any;
  timestamp: number;
  tabId: string;
}

type SyncHandler = (payload: any, tabId: string) => void;

class TabSyncService {
  private channel: BroadcastChannel;
  private tabId: string;
  private handlers = new Map<string, Set<SyncHandler>>();

  constructor(channelName: string = 'app-sync') {
    this.channel = new BroadcastChannel(channelName);
    this.tabId = this.generateTabId();

    // Listen for messages
    this.channel.onmessage = (event) => {
      this.handleMessage(event.data);
    };

    // Announce this tab
    this.broadcast('tab-connected', { tabId: this.tabId });

    // Cleanup on page unload
    window.addEventListener('beforeunload', () => {
      this.broadcast('tab-disconnected', { tabId: this.tabId });
      this.channel.close();
    });
  }
}

```

```
}

// Broadcast message to all tabs
broadcast(type: string, payload: any): void {
  const message: SyncMessage = {
    type,
    payload,
    timestamp: Date.now(),
    tabId: this.tabId
  };

  this.channel.postMessage(message);
}

// Subscribe to message type
on(type: string, handler: SyncHandler): () => void {
  if (!this.handlers.has(type)) {
    this.handlers.set(type, new Set());
  }

  this.handlers.get(type)!.add(handler);

  return () => {
    const handlers = this.handlers.get(type);
    if (handlers) {
      handlers.delete(handler);
    }
  };
}

// Handle incoming message
private handleMessage(message: SyncMessage): void {
  // Ignore messages from this tab
  if (message.tabId === this.tabId) {
    return;
  }

  this.notifyHandlers(message.type, message.payload, message.tabId);
}

// Notify handlers
private notifyHandlers(type: string, payload: any, tabId: string): void {
  const handlers = this.handlers.get(type);
  if (handlers) {
    handlers.forEach(handler => {
      try {
        handler(payload, tabId);
      }
    });
  }
}
```

```

    } catch (error) {
      console.error(`Handler error for type ${type}:`, error);
    }
  });
}
}

// Generate unique tab ID
private generateTabId(): string {
  return `tab-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`;
}

// Get this tab's ID
getTabId(): string {
  return this.tabId;
}
}

export const tabSync = new TabSyncService();

```

13.4.2 Synchronizing Authentication Across Tabs

Let's use BroadcastChannel to keep authentication state synchronized:

```

// services/auth-sync.ts
import { tabSync } from './tab-sync';
import { authService } from './auth';

class AuthSyncService {
  constructor() {
    // Listen for logout in other tabs
    tabSync.on('auth-logout', () => {
      console.log('Logout detected in another tab');
      authService.logout();
      window.location.href = '/login?reason=logout-other-tab';
    });

    // Listen for login in other tabs
    tabSync.on('auth-login', (user) => {
      console.log('Login detected in another tab');
      // Reload to pick up new auth state
      window.location.reload();
    });

    // Listen for token refresh in other tabs
    tabSync.on('auth-token-refresh', () => {
      console.log('Token refresh detected in another tab');
      // Re-initialize auth from storage
    });
  }
}

```

```

    authService.initialize();
  });
}

// Broadcast logout to other tabs
broadcastLogout(): void {
  tabSync.broadcast('auth-logout', {});
}

// Broadcast login to other tabs
broadcastLogin(user: any): void {
  tabSync.broadcast('auth-login', user);
}

// Broadcast token refresh to other tabs
broadcastTokenRefresh(): void {
  tabSync.broadcast('auth-token-refresh', {});
}
}

export const authSync = new AuthSyncService();

```

13.5 Web Workers: Background Processing

Web Workers let you run JavaScript in background threads, keeping your UI responsive while processing data, crunching numbers, or handling real-time updates. They're like hiring an intern who works in another room and sends you updates via email.

13.5.1 Creating a Data Processing Worker

```

// workers/data-processor.worker.ts
interface ProcessRequest {
  id: string;
  type: 'sort' | 'filter' | 'aggregate';
  data: any[];
  options: any;
}

interface ProcessResponse {
  id: string;
  result: any;
  error?: string;
}

// Worker message handler
self.onmessage = (event: MessageEvent<ProcessRequest>) => {

```

```

const { id, type, data, options } = event.data;

try {
  let result: any;

  switch (type) {
    case 'sort':
      result = sortData(data, options);
      break;
    case 'filter':
      result = filterData(data, options);
      break;
    case 'aggregate':
      result = aggregateData(data, options);
      break;
    default:
      throw new Error(`Unknown operation type: ${type}`);
  }

  const response: ProcessResponse = { id, result };
  self.postMessage(response);
} catch (error) {
  const response: ProcessResponse = {
    id,
    result: null,
    error: error instanceof Error ? error.message : 'Unknown error'
  };
  self.postMessage(response);
}
};

function sortData(data: any[], options: any): any[] {
  const { field, order = 'asc' } = options;
  return [...data].sort((a, b) => {
    const aVal = a[field];
    const bVal = b[field];
    const comparison = aVal < bVal ? -1 : aVal > bVal ? 1 : 0;
    return order === 'asc' ? comparison : -comparison;
  });
}

function filterData(data: any[], options: any): any[] {
  const { field, value, operator = 'equals' } = options;

  return data.filter(item => {
    const itemValue = item[field];

```

```

    switch (operator) {
      case 'equals':
        return itemValue === value;
      case 'contains':
        return String(itemValue).includes(String(value));
      case 'greater':
        return itemValue > value;
      case 'less':
        return itemValue < value;
      default:
        return true;
    }
  });
}

function aggregateData(data: any[], options: any): any {
  const { operation, field } = options;

  switch (operation) {
    case 'count':
      return data.length;
    case 'sum':
      return data.reduce((sum, item) => sum + (item[field] || 0), 0);
    case 'average':
      const sum = data.reduce((s, item) => s + (item[field] || 0), 0);
      return data.length > 0 ? sum / data.length : 0;
    case 'min':
      return Math.min(...data.map(item => item[field]));
    case 'max':
      return Math.max(...data.map(item => item[field]));
    default:
      return null;
  }
}

```

13.5.2 Worker Manager

```

// services/worker-manager.ts
class WorkerManager {
  private worker: Worker | null = null;
  private requestId = 0;
  private pendingRequests = new Map<string, {
    resolve: (result: any) => void;
    reject: (error: Error) => void;
  }>();

  constructor(workerUrl: string) {

```

```

this.worker = new Worker(workerUrl);

this.worker.onmessage = (event) => {
  const { id, result, error } = event.data;
  const pending = this.pendingRequests.get(id);

  if (pending) {
    this.pendingRequests.delete(id);

    if (error) {
      pending.reject(new Error(error));
    } else {
      pending.resolve(result);
    }
  }
};

this.worker.onerror = (error) => {
  console.error('Worker error:', error);
};
}

// Send request to worker
async process(type: string, data: any[], options: any): Promise<any> {
  if (!this.worker) {
    throw new Error('Worker not initialized');
  }

  const id = `req-${++this.requestId}`;

  return new Promise((resolve, reject) => {
    this.pendingRequests.set(id, { resolve, reject });

    this.worker!.postMessage({
      id,
      type,
      data,
      options
    });

    // Timeout after 30 seconds
    setTimeout(() => {
      if (this.pendingRequests.has(id)) {
        this.pendingRequests.delete(id);
        reject(new Error('Request timeout'));
      }
    }, 30000);
  });
}

```

```

    });
  }

  // Terminate worker
  terminate(): void {
    if (this.worker) {
      this.worker.terminate();
      this.worker = null;
    }

    // Reject all pending requests
    this.pendingRequests.forEach(({ reject }) => {
      reject(new Error('Worker terminated'));
    });
    this.pendingRequests.clear();
  }
}

export const dataWorker = new WorkerManager(
  new URL('../workers/data-processor.worker.ts', import.meta.url).href
);

```

13.6 Real-time Collaboration Patterns

Let's build a collaborative document editor that demonstrates real-time collaboration:

```

// components/collaborative-editor.ts
import { html, define, Component } from '@larc/lib';
import { wsClient } from '../services/websocket-client';

interface EditorState {
  content: string;
  collaborators: Map<string, { name: string; cursor: number }>;
  docId: string;
}

class CollaborativeEditor extends Component {
  static tagName = 'collaborative-editor';

  state: EditorState = {
    content: '',
    collaborators: new Map(),
    docId: this.props.documentId || 'default'
  };

  private editorRef: HTMLTextAreaElement | null = null;
  private unsubscribers: Array<() => void> = [];

```

```
private localChanges = false;

async connectedCallback() {
  super.connectedCallback();

  await wsClient.connect();

  // Join document collaboration
  wsClient.send('join-document', { docId: this.state.docId });

  // Subscribe to document updates
  this.unsubscribers.push(
    wsClient.on('document-update', (update: any) => {
      if (!this.localChanges) {
        this.applyRemoteUpdate(update);
      }
    })
  );

  // Subscribe to collaborator updates
  this.unsubscribers.push(
    wsClient.on('collaborator-joined', (collaborator: any) => {
      const collaborators = new Map(this.state.collaborators);
      collaborators.set(collaborator.userId, collaborator);
      this.setState({ collaborators });
    })
  );

  this.unsubscribers.push(
    wsClient.on('collaborator-left', (data: any) => {
      const collaborators = new Map(this.state.collaborators);
      collaborators.delete(data.userId);
      this.setState({ collaborators });
    })
  );

  // Load initial document content
  await this.loadDocument();
}

disconnectedCallback() {
  super.disconnectedCallback();

  // Leave document
  wsClient.send('leave-document', { docId: this.state.docId });

  this.unsubscribers.forEach(unsub => unsub());
}
```

```

}

private async loadDocument(): Promise<void> {
  try {
    const response = await fetch(`/api/documents/${this.state.docId}`);
    const data = await response.json();
    this.setState({ content: data.content });
  } catch (error) {
    console.error('Failed to load document:', error);
  }
}

private handleInput(e: Event): void {
  const textarea = e.target as HTMLTextAreaElement;
  const newContent = textarea.value;
  const cursorPosition = textarea.selectionStart;

  this.localChanges = true;
  this.setState({ content: newContent });

  // Send update to server
  wsClient.send('document-update', {
    docId: this.state.docId,
    content: newContent,
    cursor: cursorPosition
  });

  // Reset local changes flag after a delay
  setTimeout(() => {
    this.localChanges = false;
  }, 100);
}

private applyRemoteUpdate(update: any): void {
  this.setState({ content: update.content });
}

render() {
  const { content, collaborators } = this.state;

  return html`
    <div class="collaborative-editor">
      <div class="editor-header">
        <div class="collaborators">
          ${Array.from(collaborators.values()).map(collab => html`
            <div class="collaborator-badge" title="${collab.name}">
              ${collab.name.charAt(0).toUpperCase()}
            </div>
          `)}
        </div>
      </div>
    </div>
  `;
}

```

```

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

  <textarea
    class="editor-content"
    value="${content}"
    oninput="${this.handleInput}"
    placeholder="Start typing..."
    ref="${(el: HTMLTextAreaElement) => this.editorRef = el}"
  ></textarea>
</div>

<style>
  .collaborative-editor {
    display: flex;
    flex-direction: column;
    height: 100%;
    border: 1px solid #ddd;
    border-radius: 8px;
    overflow: hidden;
  }

  .editor-header {
    padding: 1rem;
    background: #f8f9fa;
    border-bottom: 1px solid #ddd;
  }

  .collaborators {
    display: flex;
    gap: 0.5rem;
  }

  .collaborator-badge {
    width: 32px;
    height: 32px;
    border-radius: 50%;
    background: #007bff;
    color: white;
    display: flex;
    align-items: center;
    justify-content: center;
    font-weight: bold;
  }

```

```

        .editor-content {
            flex: 1;
            padding: 1rem;
            border: none;
            resize: none;
            font-family: 'Monaco', 'Courier New', monospace;
            font-size: 14px;
            line-height: 1.5;
        }

        .editor-content:focus {
            outline: none;
        }
    </style>
`
;
}
}

define(CollaborativeEditor);

```

13.7 What We've Learned

In this chapter, we've built comprehensive real-time features for LARC applications:

- **WebSocket integration** with automatic reconnection, heartbeat, and message routing
- **Server-Sent Events** for one-way server-to-client updates
- **BroadcastChannel** for synchronizing state across browser tabs
- **Web Workers** for background processing without blocking the UI
- **Real-time collaboration** patterns for building multi-user experiences

Real-time features transform applications from static pages into living, breathing experiences. Users no longer need to refresh to see updates—the updates come to them. Just remember: with great real-time power comes great responsibility to handle connection failures, race conditions, and the inevitable “why isn't it updating?” support tickets.

In the next chapter, we'll explore file management, where we'll learn to work with the Origin Private File System (OPFS), build file browsers, and handle uploads and downloads—because what's a modern application without the ability to handle files?

Chapter 14

File Management

In which we discover that the browser can store files just like a real computer, learn that “unlimited storage” means “we’ll delete your stuff if we feel like it,” and find out that OPFS is the best-kept secret in modern web development.

File management used to be simple: users uploaded files to your server, and your server stored them in folders. But modern web applications demand more—they need to work offline, handle large files without overwhelming your bandwidth budget, and provide instant access to previously loaded content. Enter the Origin Private File System (OPFS), the browser’s own file system that gives your web app genuine file storage capabilities.

In this chapter, we’ll build comprehensive file management features in LARC: file browsers, upload and download handlers, directory navigation, and smart quota management. By the end, you’ll have a file system in your web app that rivals native applications.

14.1 Understanding OPFS: Your App’s Private File System

OPFS is like giving your web application its own private hard drive. Unlike `LocalStorage` (limited to 5-10MB) or `IndexedDB` (better but still clunky for files), OPFS provides:

- **Large storage capacity:** Gigabytes, not megabytes
- **High performance:** Direct file I/O operations
- **Private to your origin:** Other sites can’t access your files
- **Persistent:** Survives page reloads and browser restarts
- **Streaming support:** Handle large files without loading everything into memory

The catch? Storage is “best effort”—browsers can delete it under pressure. But with proper quota management, OPFS is remarkably reliable.

14.2 Getting Started with OPFS

Let’s build a comprehensive OPFS wrapper that provides a clean API for file operations:

```
// services/file-system.ts
interface FileInfo {
  name: string;
```

```

    size: number;
    type: string;
    handle: FileSystemFileHandle;
    lastModified: number;
}

interface DirectoryInfo {
    name: string;
    handle: FileSystemDirectoryHandle;
    parent?: FileSystemDirectoryHandle;
}

class FileSystemService {
    private root: FileSystemDirectoryHandle | null = null;

    // Initialize the file system
    async initialize(): Promise<void> {
        try {
            this.root = await navigator.storage.getDirectory();
            console.log('OPFS initialized');
        } catch (error) {
            console.error('Failed to initialize OPFS:', error);
            throw new Error('File system not available');
        }
    }

    // Get root directory handle
    private async getRoot(): Promise<FileSystemDirectoryHandle> {
        if (!this.root) {
            await this.initialize();
        }
        return this.root!;
    }

    // Create or get a directory
    async createDirectory(path: string): Promise<FileSystemDirectoryHandle> {
        const root = await this.getRoot();
        const parts = path.split('/').filter(p => p.length > 0);

        let current = root;
        for (const part of parts) {
            current = await current.getDirectoryHandle(part, { create: true });
        }

        return current;
    }
}

```

```

// Get directory handle
async getDirectory(path: string): Promise<FileSystemDirectoryHandle> {
  const root = await this.getRoot();
  const parts = path.split('/').filter(p => p.length > 0);

  let current = root;
  for (const part of parts) {
    current = await current.getDirectoryHandle(part);
  }

  return current;
}

// List files in a directory
async listFiles(path: string = '/'): Promise<FileInfo[]> {
  try {
    const dir = path === '/'
      ? await this.getRoot()
      : await this.getDirectory(path);

    const files: FileInfo[] = [];

    for await (const [name, handle] of dir.entries()) {
      if (handle.kind === 'file') {
        const fileHandle = handle as FileSystemFileHandle;
        const file = await fileHandle.getFile();

        files.push({
          name,
          size: file.size,
          type: file.type,
          handle: fileHandle,
          lastModified: file.lastModified
        });
      }
    }

    return files.sort((a, b) => a.name.localeCompare(b.name));
  } catch (error) {
    console.error('Failed to list files:', error);
    return [];
  }
}

// List directories
async listDirectories(path: string = '/'): Promise<DirectoryInfo[]> {
  try {

```

```

const dir = path === '/'
  ? await this.getRoot()
  : await this.getDirectory(path);

const directories: DirectoryInfo[] = [];

for await (const [name, handle] of dir.entries()) {
  if (handle.kind === 'directory') {
    directories.push({
      name,
      handle: handle as FileSystemDirectoryHandle,
      parent: dir
    });
  }
}

return directories.sort((a, b) => a.name.localeCompare(b.name));
} catch (error) {
  console.error('Failed to list directories:', error);
  return [];
}
}

// Write file
async writeFile(
  path: string,
  fileName: string,
  content: Blob | ArrayBuffer | string
): Promise<void> {
  try {
    const dir = path === '/'
      ? await this.getRoot()
      : await this.createDirectory(path);

    const fileHandle = await dir.getFileHandle(fileName, { create: true });
    const writable = await fileHandle.createWritable();

    if (typeof content === 'string') {
      await writable.write(content);
    } else if (content instanceof ArrayBuffer) {
      await writable.write(new Blob([content]));
    } else {
      await writable.write(content);
    }

    await writable.close();
  } catch (error) {

```

```

        console.error('Failed to write file:', error);
        throw error;
    }
}

// Read file
async readFile(path: string, fileName: string): Promise<File> {
    try {
        const dir = path === '/'
            ? await this.getRoot()
            : await this.getDirectory(path);

        const fileHandle = await dir.getFileHandle(fileName);
        return await fileHandle.getFile();
    } catch (error) {
        console.error('Failed to read file:', error);
        throw error;
    }
}

// Delete file
async deleteFile(path: string, fileName: string): Promise<void> {
    try {
        const dir = path === '/'
            ? await this.getRoot()
            : await this.getDirectory(path);

        await dir.removeEntry(fileName);
    } catch (error) {
        console.error('Failed to delete file:', error);
        throw error;
    }
}

// Delete directory
async deleteDirectory(path: string, recursive: boolean = false): Promise<void> {
    try {
        const parts = path.split('/').filter(p => p.length > 0);
        const dirName = parts.pop()!;
        const parentPath = parts.join('/');

        const parent = parentPath
            ? await this.getDirectory(parentPath)
            : await this.getRoot();

        await parent.removeEntry(dirName, { recursive });
    } catch (error) {

```

```
        console.error('Failed to delete directory:', error);
        throw error;
    }
}

// Check if file exists
async fileExists(path: string, fileName: string): Promise<boolean> {
    try {
        await this.readFile(path, fileName);
        return true;
    } catch {
        return false;
    }
}

// Get file handle
async getFileHandle(
    path: string,
    fileName: string
): Promise<FileSystemFileHandle> {
    const dir = path === '/'
        ? await this.getRoot()
        : await this.getDirectory(path);

    return await dir.getFileHandle(fileName);
}

// Copy file
async copyFile(
    sourcePath: string,
    sourceFile: string,
    destPath: string,
    destFile: string
): Promise<void> {
    const file = await this.readFile(sourcePath, sourceFile);
    const content = await file.arrayBuffer();
    await this.writeFile(destPath, destFile, content);
}

// Move file
async moveFile(
    sourcePath: string,
    sourceFile: string,
    destPath: string,
    destFile: string
): Promise<void> {
    await this.copyFile(sourcePath, sourceFile, destPath, destFile);
}
```

```

    await this.deleteFile(sourcePath, sourceFile);
  }

  // Get storage estimate
  async getStorageInfo(): Promise<{
    usage: number;
    quota: number;
    percentUsed: number;
  }> {
    const estimate = await navigator.storage.estimate();
    const usage = estimate.usage || 0;
    const quota = estimate.quota || 0;
    const percentUsed = quota > 0 ? (usage / quota) * 100 : 0;

    return {
      usage,
      quota,
      percentUsed
    };
  }

  // Calculate directory size
  async getDirectorySize(path: string = '/'): Promise<number> {
    let totalSize = 0;

    const files = await this.listFiles(path);
    for (const file of files) {
      totalSize += file.size;
    }

    const directories = await this.listDirectories(path);
    for (const dir of directories) {
      const subPath = path === '/' ? dir.name : `${path}/${dir.name}`;
      totalSize += await this.getDirectorySize(subPath);
    }

    return totalSize;
  }
}

export const fileSystem = new FileSystemService();

```

14.3 Building a File Browser Component

Now let's create a beautiful, functional file browser that lets users navigate directories and manage files:

```

// components/file-browser.ts
import { html, define, Component } from '@larc/lib';
import { fileSystem, FileInfo, DirectoryInfo } from '../services/file-system';

interface FileBrowserState {
  currentPath: string;
  files: FileInfo[];
  directories: DirectoryInfo[];
  selectedItems: Set<string>;
  isLoading: boolean;
  viewMode: 'list' | 'grid';
  sortBy: 'name' | 'size' | 'date';
  sortOrder: 'asc' | 'desc';
}

class FileBrowser extends Component {
  static tagName = 'file-browser';

  state: FileBrowserState = {
    currentPath: '/',
    files: [],
    directories: [],
    selectedItems: new Set(),
    isLoading: false,
    viewMode: 'list',
    sortBy: 'name',
    sortOrder: 'asc'
  };

  async connectedCallback() {
    super.connectedCallback();
    await fileSystem.initialize();
    await this.loadDirectory('/');
  }

  private async loadDirectory(path: string): Promise<void> {
    this.setState({ isLoading: true, currentPath: path });

    try {
      const [files, directories] = await Promise.all([
        fileSystem.listFiles(path),
        fileSystem.listDirectories(path)
      ]);

      this.setState({
        files: this.sortItems(files),
        directories,

```

```

        selectedItems: new Set(),
        isLoading: false
    });
} catch (error) {
    console.error('Failed to load directory:', error);
    this.setState({ isLoading: false });
}
}

private sortItems(files: FileInfo[]): FileInfo[] {
    const { sortBy, sortOrder } = this.state;
    const sorted = [...files];

    sorted.sort((a, b) => {
        let comparison = 0;

        switch (sortBy) {
            case 'name':
                comparison = a.name.localeCompare(b.name);
                break;
            case 'size':
                comparison = a.size - b.size;
                break;
            case 'date':
                comparison = a.lastModified - b.lastModified;
                break;
        }

        return sortOrder === 'asc' ? comparison : -comparison;
    });

    return sorted;
}

private async navigateToDirectory(dirName: string): Promise<void> {
    const newPath = this.state.currentPath === '/'
        ? `/${dirName}`
        : `${this.state.currentPath}/${dirName}`;

    await this.loadDirectory(newPath);
}

private async navigateUp(): Promise<void> {
    const parts = this.state.currentPath.split('/').filter(p => p.length > 0);
    parts.pop();
    const newPath = parts.length > 0 ? `/${parts.join('/')}` : '/';
    await this.loadDirectory(newPath);
}

```

```

}

private toggleSelection(name: string): void {
  const selectedItems = new Set(this.state.selectedItems);

  if (selectedItems.has(name)) {
    selectedItems.delete(name);
  } else {
    selectedItems.add(name);
  }

  this.setState({ selectedItems });
}

private async deleteSelected(): Promise<void> {
  if (!confirm('Delete selected items?')) return;

  const { currentPath, selectedItems } = this.state;

  for (const name of selectedItems) {
    try {
      await fileSystem.deleteFile(currentPath, name);
    } catch (error) {
      console.error(`Failed to delete ${name}:`, error);
    }
  }

  await this.loadDirectory(currentPath);
}

private async createFolder(): Promise<void> {
  const name = prompt('Folder name:');
  if (!name) return;

  try {
    const newPath = this.state.currentPath === '/'
      ? name
      : `${this.state.currentPath}/${name}`;

    await fileSystem.createDirectory(newPath);
    await this.loadDirectory(this.state.currentPath);
  } catch (error) {
    console.error('Failed to create folder:', error);
    alert('Failed to create folder');
  }
}

```

```

private async downloadFile(file: FileInfo): Promise<void> {
  try {
    const fileData = await fileSystem.readFile(this.state.currentPath, file.name);
    const url = URL.createObjectURL(fileData);
    const a = document.createElement('a');
    a.href = url;
    a.download = file.name;
    a.click();
    URL.revokeObjectURL(url);
  } catch (error) {
    console.error('Failed to download file:', error);
    alert('Failed to download file');
  }
}

render() {
  const {
    currentPath,
    files,
    directories,
    selectedItems,
    isLoading,
    viewMode,
    sortBy,
    sortOrder
  } = this.state;

  return html`
    <div class="file-browser">
      <!-- Toolbar -->
      <div class="toolbar">
        <div class="breadcrumbs">
          <button
            class="breadcrumb"
            onclick="${() => this.loadDirectory('/')}"
          >
            Home
          </button>
          ${currentPath.split('/').filter(p => p).map((part, i, arr) => {
            const path = '/' + arr.slice(0, i + 1).join('/');
            return html`
              <span class="separator">/</span>
              <button
                class="breadcrumb"
                onclick="${() => this.loadDirectory(path)}"
              >
                ${part}
            `
          })}
        </div>
      </div>
    </div>
  `
}

```

```

        </button>
      `;
    }
  }
</div>

<div class="toolbar-actions">
  <button
    class="btn btn-icon"
    onclick="${this.createFolder}"
    title="New Folder"
  >
    [folder]
  </button>

  <button
    class="btn btn-icon"
    onclick="${this.deleteSelected}"
    disabled="${selectedItems.size === 0}"
    title="Delete Selected"
  >
    [trash]
  </button>

  <button
    class="btn btn-icon"
    onclick="${() => this.setState({
      viewMode: viewMode === 'list' ? 'grid' : 'list'
    })}"
    title="Toggle View"
  >
    ${viewMode === 'list' ? '[+]' : '[menu]'}
  </button>
</div>
</div>

<!-- Sort Controls -->
<div class="sort-controls">
  <select
    value="${sortBy}"
    onchange="${(e: Event) => {
      this.setState({
        sortBy: (e.target as HTMLSelectElement).value as any,
        files: this.sortItems(files)
      });
    }}"
  >
    <option value="name">Name</option>

```

```

    <option value="size">Size</option>
    <option value="date">Date</option>
  </select>

  <button
    class="btn btn-icon"
    onclick="${() => {
      this.setState({
        sortOrder: sortOrder === 'asc' ? 'desc' : 'asc',
        files: this.sortItems(files)
      });
    }}"
  >
    ${sortOrder === 'asc' ? '^' : 'v'}
  </button>
</div>

<!-- Content Area -->
<div class="content ${viewMode}">
  ${isLoading ? html`
    <div class="loading">Loading...</div>
  ` : html`
    <!-- Parent Directory Link -->
    ${currentPath !== '/' ? html`
      <div class="item directory" onclick="${this.navigateUp}">
        <div class="item-icon">[folder]</div>
        <div class="item-name">..</div>
      </div>
    ` : ''}

    <!-- Directories -->
    ${directories.map(dir => html`
      <div
        class="item directory"
        onclick="${() => this.navigateToDirectory(dir.name)}"
      >
        <div class="item-icon">[folder]</div>
        <div class="item-name">${dir.name}</div>
      </div>
    `)}

    <!-- Files -->
    ${files.map(file => html`
      <div
        class="item file ${selectedItems.has(file.name) ? 'selected' : ''}"
        onclick="${(e: MouseEvent) => {
          if (e.ctrlKey || e.metaKey) {

```

```

        this.toggleSelection(file.name);
    } else {
        this.downloadFile(file);
    }
    }}"
    >
    <div class="item-icon">${this.getFileIcon(file.type)}</div>
    <div class="item-name">${file.name}</div>
    <div class="item-size">${this.formatSize(file.size)}</div>
    <div class="item-date">
        ${new Date(file.lastModified).toLocaleDateString()}
    </div>
</div>
`))}

${files.length === 0 && directories.length === 0 ? html`
    <div class="empty">This folder is empty</div>
` : ''}
`}
</div>
</div>

<style>
.file-browser {
    display: flex;
    flex-direction: column;
    height: 100%;
    background: white;
    border: 1px solid #ddd;
    border-radius: 8px;
    overflow: hidden;
}

.toolbar {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 1rem;
    border-bottom: 1px solid #ddd;
    background: #f8f9fa;
}

.breadcrumbs {
    display: flex;
    align-items: center;
    gap: 0.25rem;
}

```

```
.breadcrumb {
  background: none;
  border: none;
  padding: 0.25rem 0.5rem;
  cursor: pointer;
  color: #007bff;
  border-radius: 4px;
}

.breadcrumb:hover {
  background: rgba(0, 123, 255, 0.1);
}

.separator {
  color: #6c757d;
}

.toolbar-actions {
  display: flex;
  gap: 0.5rem;
}

.btn-icon {
  padding: 0.5rem;
  border: 1px solid #ddd;
  background: white;
  border-radius: 4px;
  cursor: pointer;
  font-size: 1.2rem;
}

.btn-icon:hover:not(:disabled) {
  background: #f8f9fa;
}

.btn-icon:disabled {
  opacity: 0.5;
  cursor: not-allowed;
}

.sort-controls {
  display: flex;
  gap: 0.5rem;
  padding: 0.5rem 1rem;
  border-bottom: 1px solid #ddd;
  background: #f8f9fa;
}
```

```
.content {
  flex: 1;
  overflow-y: auto;
  padding: 1rem;
}

.content.list {
  display: flex;
  flex-direction: column;
  gap: 0.5rem;
}

.content.grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(120px, 1fr));
  gap: 1rem;
}

.item {
  display: flex;
  align-items: center;
  gap: 0.75rem;
  padding: 0.75rem;
  border: 1px solid #ddd;
  border-radius: 4px;
  cursor: pointer;
  transition: all 0.2s;
}

.content.grid .item {
  flex-direction: column;
  text-align: center;
}

.item:hover {
  background: #f8f9fa;
  border-color: #007bff;
}

.item.selected {
  background: #e7f3ff;
  border-color: #007bff;
}

.item.directory {
  font-weight: 500;
}
```

```

        .item-icon {
            font-size: 2rem;
        }

        .item-name {
            flex: 1;
            overflow: hidden;
            text-overflow: ellipsis;
            white-space: nowrap;
        }

        .content.grid .item-name {
            width: 100%;
        }

        .item-size, .item-date {
            font-size: 0.875rem;
            color: #6c757d;
        }

        .content.grid .item-size,
        .content.grid .item-date {
            display: none;
        }

        .loading, .empty {
            padding: 2rem;
            text-align: center;
            color: #6c757d;
        }
    </style>
`
;
}

private getFileIcon(type: string): string {
    if (type.startsWith('image/')) return '[image]';
    if (type.startsWith('video/')) return '[video]';
    if (type.startsWith('audio/')) return '[audio]';
    if (type.startsWith('text/')) return '[file]';
    if (type === 'application/pdf') return '[PDF]';
    if (type.includes('zip') || type.includes('compressed')) return '[package]';
    return '[file]';
}

private formatSize(bytes: number): string {
    if (bytes === 0) return '0 B';
    const k = 1024;

```

```

    const sizes = ['B', 'KB', 'MB', 'GB'];
    const i = Math.floor(Math.log(bytes) / Math.log(k));
    return Math.round(bytes / Math.pow(k, i) * 100) / 100 + ' ' + sizes[i];
  }
}

define(FileBrowser);

```

14.4 File Upload and Download

Let's create a comprehensive upload component with drag-and-drop, progress tracking, and validation:

```

// components/file-upload.ts
import { html, define, Component } from '@larc/lib';
import { fileSystem } from '../services/file-system';

interface FileUploadState {
  uploads: Map<string, {
    file: File;
    progress: number;
    status: 'pending' | 'uploading' | 'complete' | 'error';
    error?: string;
  }>;
  isDragging: boolean;
}

interface FileUploadProps {
  path?: string;
  maxSize?: number;
  acceptedTypes?: string[];
  multiple?: boolean;
  onUploadComplete?: (files: File[]) => void;
}

class FileUpload extends Component<FileUploadProps> {
  static tagName = 'file-upload';

  state: FileUploadState = {
    uploads: new Map(),
    isDragging: false
  };

  private fileInputRef: HTMLInputElement | null = null;

  private handleDragEnter = (e: DragEvent) => {
    e.preventDefault();
  }
}

```

```

    this.setState({ isDragging: true });
  };

  private handleDragLeave = (e: DragEvent) => {
    e.preventDefault();
    this.setState({ isDragging: false });
  };

  private handleDragOver = (e: DragEvent) => {
    e.preventDefault();
  };

  private handleDrop = async (e: DragEvent) => {
    e.preventDefault();
    this.setState({ isDragging: false });

    const files = Array.from(e.dataTransfer?.files || []);
    await this.processFiles(files);
  };

  private handleFileSelect = async (e: Event) => {
    const input = e.target as HTMLInputElement;
    const files = Array.from(input.files || []);
    await this.processFiles(files);

    // Reset input
    input.value = '';
  };

  private async processFiles(files: File[]): Promise<void> {
    const { maxSize, acceptedTypes, multiple } = this.props;

    // Filter and validate files
    const validFiles = files.filter(file => {
      // Check size
      if (maxSize && file.size > maxSize) {
        this.addError(file, `File too large (max ${this.formatSize(maxSize)})`);
        return false;
      }
    });

    // Check type
    if (acceptedTypes && acceptedTypes.length > 0) {
      const isAccepted = acceptedTypes.some(type => {
        if (type.endsWith('/*')) {
          const category = type.split('/')[0];
          return file.type.startsWith(category + '/');
        }
      });
    }
  }

```

```

        return file.type === type;
    });

    if (!isAccepted) {
        this.addError(file, 'File type not accepted');
        return false;
    }
}

return true;
});

// Limit to one file if not multiple
const filesToUpload = multiple ? validFiles : validFiles.slice(0, 1);

// Start uploads
for (const file of filesToUpload) {
    await this.uploadFile(file);
}
}

private addError(file: File, error: string): void {
    const uploads = new Map(this.state.uploads);
    uploads.set(file.name, {
        file,
        progress: 0,
        status: 'error',
        error
    });
    this.setState({ uploads });
}

private async uploadFile(file: File): Promise<void> {
    const uploads = new Map(this.state.uploads);

    // Add to upload list
    uploads.set(file.name, {
        file,
        progress: 0,
        status: 'pending'
    });
    this.setState({ uploads });

    try {
        // Update status to uploading
        const upload = uploads.get(file.name)!;
        upload.status = 'uploading';
    }
}

```

```

    this.setState({ uploads: new Map(uploads) });

    // Simulate progress (in real app, track actual upload progress)
    const progressInterval = setInterval(() => {
      const current = uploads.get(file.name);
      if (current && current.status === 'uploading' && current.progress < 90) {
        current.progress += 10;
        this.setState({ uploads: new Map(uploads) });
      }
    }, 100);

    // Write file to OPFS
    const path = this.props.path || '/';
    const content = await file.arrayBuffer();
    await fileSystem.writeFile(path, file.name, content);

    clearInterval(progressInterval);

    // Mark as complete
    upload.progress = 100;
    upload.status = 'complete';
    this.setState({ uploads: new Map(uploads) });

    // Notify parent
    if (this.props.onUploadComplete) {
      this.props.onUploadComplete([file]);
    }

    // Remove from list after 2 seconds
    setTimeout(() => {
      const current = new Map(this.state.uploads);
      current.delete(file.name);
      this.setState({ uploads: current });
    }, 2000);
  } catch (error) {
    const upload = uploads.get(file.name)!;
    upload.status = 'error';
    upload.error = error instanceof Error ? error.message : 'Upload failed';
    this.setState({ uploads: new Map(uploads) });
  }
}

private formatSize(bytes: number): string {
  if (bytes === 0) return '0 B';
  const k = 1024;
  const sizes = ['B', 'KB', 'MB', 'GB'];
  const i = Math.floor(Math.log(bytes) / Math.log(k));

```

```

    return Math.round(bytes / Math.pow(k, i) * 100) / 100 + ' ' + sizes[i];
  }

  render() {
    const { uploads, isDragging } = this.state;
    const { multiple = true, acceptedTypes } = this.props;

    return html`
      <div class="file-upload">
        <div
          class="drop-zone ${isDragging ? 'dragging' : ''}"
          ondragenter="${this.handleDragEnter}"
          ondragleave="${this.handleDragLeave}"
          ondragover="${this.handleDragOver}"
          ondrop="${this.handleDrop}"
          onclick="${() => this.fileInputRef?.click()}"
        >
          <div class="drop-zone-content">
            <div class="upload-icon">[upload]</div>
            <div class="upload-text">
              <strong>Drop files here</strong> or click to browse
            </div>
            ${acceptedTypes ? html`
              <div class="accepted-types">
                Accepted: ${acceptedTypes.join(', ')}
              </div>
            ` : ''}
          </div>
        </div>

        <input
          type="file"
          ref="${(el: HTMLInputElement) => this.fileInputRef = el}"
          onchange="${this.handleFileSelect}"
          multiple="${multiple}"
          accept="${acceptedTypes?.join(',') || ''}"
          style="display: none;"
        />

        ${uploads.size > 0 ? html`
          <div class="upload-list">
            ${Array.from(uploads.entries()).map(([name, upload]) => html`
              <div class="upload-item ${upload.status}>
                <div class="upload-info">
                  <div class="upload-name">${name}</div>
                  <div class="upload-size">${this.formatSize(upload.file.size)}</div>
                </div>
            `)}
          </div>
        `}
      </div>
    `;
  }
}

```

```

    ${upload.status === 'uploading' ? html`
      <div class="progress-bar">
        <div
          class="progress-fill"
          style="width: ${upload.progress}%"
        ></div>
      </div>
    ` : ''}

    ${upload.status === 'complete' ? html`
      <div class="upload-status success">[v] Complete</div>
    ` : ''}

    ${upload.status === 'error' ? html`
      <div class="upload-status error">
        [x] ${upload.error}
      </div>
    ` : ''}
  </div>
)}
</div>
` : ''}
</div>

<style>
  .file-upload {
    display: flex;
    flex-direction: column;
    gap: 1rem;
  }

  .drop-zone {
    border: 2px dashed #ddd;
    border-radius: 8px;
    padding: 3rem;
    text-align: center;
    cursor: pointer;
    transition: all 0.2s;
    background: #f8f9fa;
  }

  .drop-zone:hover {
    border-color: #007bff;
    background: #e7f3ff;
  }

  .drop-zone.dragging {

```

```
    border-color: #007bff;
    background: #e7f3ff;
    transform: scale(1.02);
}

.upload-icon {
    font-size: 3rem;
    margin-bottom: 1rem;
}

.upload-text {
    font-size: 1.1rem;
    margin-bottom: 0.5rem;
}

.accepted-types {
    font-size: 0.875rem;
    color: #6c757d;
}

.upload-list {
    display: flex;
    flex-direction: column;
    gap: 0.75rem;
}

.upload-item {
    padding: 1rem;
    border: 1px solid #ddd;
    border-radius: 4px;
    background: white;
}

.upload-info {
    display: flex;
    justify-content: space-between;
    margin-bottom: 0.5rem;
}

.upload-name {
    font-weight: 500;
}

.upload-size {
    color: #6c757d;
    font-size: 0.875rem;
}
```

```

    .progress-bar {
      height: 4px;
      background: #e9ecef;
      border-radius: 2px;
      overflow: hidden;
    }

    .progress-fill {
      height: 100%;
      background: #007bff;
      transition: width 0.3s ease;
    }

    .upload-status {
      margin-top: 0.5rem;
      font-size: 0.875rem;
      font-weight: 500;
    }

    .upload-status.success {
      color: #28a745;
    }

    .upload-status.error {
      color: #dc3545;
    }
  </style>
`
};
}
}

define(FileUpload);

```

14.5 Storage Quota Management

Managing storage quotas is crucial—browsers can be generous with space, but they can also delete your data without asking. Let's build a quota monitoring component:

```

// components/storage-quota.ts
import { html, define, Component } from '@larc/lib';
import { fileSystem } from '../services/file-system';

interface StorageQuotaState {
  usage: number;
  quota: number;
  percentUsed: number;
  isLoading: boolean;
}

```

```

}

class StorageQuota extends Component {
  static tagName = 'storage-quota';

  state: StorageQuotaState = {
    usage: 0,
    quota: 0,
    percentUsed: 0,
    isLoading: true
  };

  async connectedCallback() {
    super.connectedCallback();
    await this.updateQuota();

    // Update every 30 seconds
    setInterval(() => {
      this.updateQuota();
    }, 30000);
  }

  private async updateQuota(): Promise<void> {
    try {
      const info = await fileSystem.getStorageInfo();
      this.setState({
        ...info,
        isLoading: false
      });
    } catch (error) {
      console.error('Failed to get storage info:', error);
      this.setState({ isLoading: false });
    }
  }

  private async requestPersistence(): Promise<void> {
    if (!navigator.storage?.persist) {
      alert('Persistent storage not supported');
      return;
    }

    try {
      const isPersisted = await navigator.storage.persist();
      if (isPersisted) {
        alert('Storage is now persistent!');
      } else {
        alert('Persistence request denied');
      }
    }
  }
}

```

```

    }
  } catch (error) {
    console.error('Failed to request persistence:', error);
    alert('Failed to request persistence');
  }
}

render() {
  const { usage, quota, percentUsed, isLoading } = this.state;

  if (isLoading) {
    return html`<div class="storage-quota loading">Loading...</div>`;
  }

  const getStatusClass = () => {
    if (percentUsed > 90) return 'danger';
    if (percentUsed > 75) return 'warning';
    return 'normal';
  };

  return html`
    <div class="storage-quota">
      <div class="quota-header">
        <h4>Storage Usage</h4>
        <button
          class="btn btn-small"
          onclick="${this.requestPersistence}"
        >
          Request Persistent Storage
        </button>
      </div>

      <div class="quota-bar ${getStatusClass()}">
        <div class="quota-fill" style="width: ${percentUsed}%"></div>
      </div>

      <div class="quota-info">
        <div class="quota-stat">
          <span class="label">Used:</span>
          <span class="value">${this.formatBytes(usage)}</span>
        </div>
        <div class="quota-stat">
          <span class="label">Available:</span>
          <span class="value">${this.formatBytes(quota)}</span>
        </div>
        <div class="quota-stat">
          <span class="label">Percentage:</span>

```

```

    <span class="value">${percentUsed.toFixed(1)}%</span>
  </div>
</div>

${percentUsed > 75 ? html`
  <div class="quota-warning">
    [warning] Running low on storage space. Consider cleaning up old files.
  </div>
  ` : ''}
</div>

<style>
  .storage-quota {
    padding: 1.5rem;
    border: 1px solid #ddd;
    border-radius: 8px;
    background: white;
  }

  .quota-header {
    display: flex;
    justify-content: space-between;
    align-items: center;
    margin-bottom: 1rem;
  }

  .quota-header h4 {
    margin: 0;
  }

  .btn-small {
    padding: 0.5rem 1rem;
    border: 1px solid #007bff;
    background: white;
    color: #007bff;
    border-radius: 4px;
    cursor: pointer;
    font-size: 0.875rem;
  }

  .btn-small:hover {
    background: #007bff;
    color: white;
  }

  .quota-bar {
    height: 24px;

```

```
    background: #e9ecef;
    border-radius: 12px;
    overflow: hidden;
    margin-bottom: 1rem;
  }

  .quota-fill {
    height: 100%;
    transition: width 0.3s ease;
  }

  .quota-bar.normal .quota-fill { background: #28a745; }
  .quota-bar.warning .quota-fill { background: #ffc107; }
  .quota-bar.danger .quota-fill { background: #dc3545; }

  .quota-info {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    gap: 1rem;
    margin-bottom: 1rem;
  }

  .quota-stat {
    display: flex;
    flex-direction: column;
    gap: 0.25rem;
  }

  .quota-stat .label {
    font-size: 0.875rem;
    color: #6c757d;
  }

  .quota-stat .value {
    font-size: 1.25rem;
    font-weight: 500;
  }

  .quota-warning {
    padding: 0.75rem;
    background: #fff3cd;
    border: 1px solid #ffc107;
    border-radius: 4px;
    color: #856404;
  }
</style>
`;
```

```

}

private formatBytes(bytes: number): string {
  if (bytes === 0) return '0 B';
  const k = 1024;
  const sizes = ['B', 'KB', 'MB', 'GB', 'TB'];
  const i = Math.floor(Math.log(bytes) / Math.log(k));
  return Math.round(bytes / Math.pow(k, i) * 100) / 100 + ' ' + sizes[i];
}
}

define(StorageQuota);

```

14.6 File Type Filtering and Search

Let's add powerful filtering and search capabilities:

```

// services/file-search.ts
import { fileSystem, FileInfo } from './file-system';

interface SearchOptions {
  query?: string;
  types?: string[];
  minSize?: number;
  maxSize?: number;
  fromDate?: Date;
  toDate?: Date;
  path?: string;
  recursive?: boolean;
}

interface SearchResult {
  file: FileInfo;
  path: string;
  score: number;
}

class FileSearchService {
  async search(options: SearchOptions): Promise<SearchResult[]> {
    const results: SearchResult[] = [];
    await this.searchDirectory(options.path || '/', options, results);

    // Sort by relevance score
    results.sort((a, b) => b.score - a.score);

    return results;
  }
}

```

```

private async searchDirectory(
  path: string,
  options: SearchOptions,
  results: SearchResult[]
): Promise<void> {
  const files = await fileSystem.listFiles(path);

  for (const file of files) {
    const score = this.calculateScore(file, options);

    if (score > 0) {
      results.push({ file, path, score });
    }
  }

  // Search subdirectories if recursive
  if (options.recursive) {
    const directories = await fileSystem.listDirectories(path);

    for (const dir of directories) {
      const subPath = path === '/' ? `/${dir.name}` : `${path}/${dir.name}`;
      await this.searchDirectory(subPath, options, results);
    }
  }
}

private calculateScore(file: FileInfo, options: SearchOptions): number {
  let score = 0;

  // Check query match
  if (options.query) {
    const query = options.query.toLowerCase();
    const name = file.name.toLowerCase();

    if (name === query) {
      score += 100; // Exact match
    } else if (name.startsWith(query)) {
      score += 50; // Starts with
    } else if (name.includes(query)) {
      score += 25; // Contains
    } else {
      return 0; // No match
    }
  }

  // Check type filter
  if (options.types && options.types.length > 0) {

```

```

    const matchesType = options.types.some(type => {
      if (type.endsWith('/*')) {
        const category = type.split('/')[0];
        return file.type.startsWith(category + '/');
      }
      return file.type === type;
    });

    if (!matchesType) return 0;
    score += 10;
  }

  // Check size filters
  if (options.minSize !== undefined && file.size < options.minSize) {
    return 0;
  }

  if (options.maxSize !== undefined && file.size > options.maxSize) {
    return 0;
  }

  // Check date filters
  if (options.fromDate && file.lastModified < options.fromDate.getTime()) {
    return 0;
  }

  if (options.toDate && file.lastModified > options.toDate.getTime()) {
    return 0;
  }

  return score;
}

export const fileSearch = new FileSearchService();

```

14.7 Putting It All Together

Let's create a complete file manager component that combines everything:

```

// components/file-manager.ts
import { html, define, Component } from '@larc/lib';
import './file-browser';
import './file-upload';
import './storage-quota';

interface FileManagerState {

```

```

    currentView: 'browser' | 'upload' | 'settings';
  }

class FileManager extends Component {
  static tagName = 'file-manager';

  state: FileManagerState = {
    currentView: 'browser'
  };

  render() {
    const { currentView } = this.state;

    return html`
      <div class="file-manager">
        <div class="sidebar">
          <h3>File Manager</h3>

          <nav class="nav">
            <button
              class="nav-item ${currentView === 'browser' ? 'active' : ''}"
              onclick="${() => this.setState({ currentView: 'browser' })}"
            >
              [folder] Browse Files
            </button>

            <button
              class="nav-item ${currentView === 'upload' ? 'active' : ''}"
              onclick="${() => this.setState({ currentView: 'upload' })}"
            >
              [upload] Upload
            </button>

            <button
              class="nav-item ${currentView === 'settings' ? 'active' : ''}"
              onclick="${() => this.setState({ currentView: 'settings' })}"
            >
              [gear] Storage
            </button>
          </nav>
        </div>

        <div class="main-content">
          ${currentView === 'browser' ? html`
            <file-browser></file-browser>
          ` : ''}
        </div>
      </div>
    `;
  }
}

```

```

    ${currentView === 'upload' ? html`
      <div class="upload-view">
        <h2>Upload Files</h2>
        <file-upload
          path="/"
          maxSize="${50 * 1024 * 1024}"
          multiple="${true}"
        ></file-upload>
      </div>
    ` : ''}

    ${currentView === 'settings' ? html`
      <div class="settings-view">
        <h2>Storage Settings</h2>
        <storage-quota></storage-quota>
      </div>
    ` : ''}
  </div>
</div>

<style>
  .file-manager {
    display: grid;
    grid-template-columns: 250px 1fr;
    height: 100vh;
    overflow: hidden;
  }

  .sidebar {
    padding: 1.5rem;
    background: #f8f9fa;
    border-right: 1px solid #ddd;
  }

  .sidebar h3 {
    margin: 0 0 1.5rem 0;
  }

  .nav {
    display: flex;
    flex-direction: column;
    gap: 0.5rem;
  }

  .nav-item {
    display: flex;
    align-items: center;
  }

```

```

        gap: 0.5rem;
        padding: 0.75rem 1rem;
        border: none;
        background: transparent;
        border-radius: 4px;
        cursor: pointer;
        text-align: left;
        font-size: 1rem;
    }

    .nav-item:hover {
        background: rgba(0, 0, 0, 0.05);
    }

    .nav-item.active {
        background: #007bff;
        color: white;
    }

    .main-content {
        padding: 1.5rem;
        overflow-y: auto;
    }

    .upload-view, .settings-view {
        max-width: 800px;
    }

    .upload-view h2, .settings-view h2 {
        margin-top: 0;
    }
</style>
`;
}
}

define(FileManager);

```

14.8 What We've Learned

In this chapter, we've built a comprehensive file management system for LARC applications:

- **OPFS integration** with a clean, promise-based API for file operations
- **File browser component** with list and grid views, sorting, and navigation
- **Upload handling** with drag-and-drop, progress tracking, and validation
- **Storage quota management** with monitoring and persistence requests
- **File search and filtering** capabilities for finding files quickly

- **Complete file manager** that combines all features into a cohesive interface

File management in the browser has evolved from “upload to server” to “the browser IS the file system.” With OPFS, your LARC applications can provide native-app-like file handling with offline support, fast access, and the web’s inherent advantages of zero-install deployment.

You now have the tools to build applications that handle files like a pro—whether you’re building a photo editor, document manager, or any app that needs to work with files locally. Just remember to be mindful of storage quotas, request persistence when appropriate, and always have a backup strategy for important user data.

The browser’s file system is powerful, but it’s not Fort Knox—treat it as temporary storage that happens to persist, and you’ll build resilient applications that users can trust.

Chapter 15

Theming and Styling

“CSS is the only language where two plus two equals five, sometimes three, and occasionally ‘align-items: center’ doesn’t actually center things.”

— Every web developer who’s ever tried to center a div

Theming is the art of making your application look consistently beautiful (or at least consistently mediocre) across all components, all pages, and all user preferences. It’s the difference between an application that feels like a cohesive product and one that looks like it was assembled by a committee that never met.

In this chapter, we’ll build a robust theming system for LARC applications using CSS custom properties, explore light and dark mode implementations, create a theme provider component that broadcasts theme changes through the PAN bus, handle dynamic theme switching without page reloads, and implement responsive design patterns that adapt to any screen size.

Fair warning: we’re going to spend quality time with CSS. If you thought JavaScript was weird, wait until you meet `:host-context()`, CSS cascade layers, and the eternal mystery of specificity. But by the end of this chapter, you’ll have a theming system that’s maintainable, performant, and doesn’t require a PhD in CSS archaeology.

15.1 CSS Custom Properties: Variables That Actually Work

CSS custom properties (often called CSS variables) are the foundation of modern theming. Unlike Sass variables that compile away at build time, CSS custom properties are live—change them at runtime, and everything updates instantly.

Here’s the basic syntax:

```
/* Define custom properties */
:root {
  --primary-color: #667eea;
  --secondary-color: #764ba2;
  --text-color: #333;
  --background-color: #fff;
}
```

```

/* Use custom properties */
button {
  background: var(--primary-color);
  color: var(--background-color);
}

```

Change `--primary-color` anywhere in your code, and all buttons update automatically. It's like magic, except it actually works consistently across browsers.

15.1.1 Defining a Theme System

Let's build a comprehensive theme system with semantic tokens:

```

/* theme/base.css */

/* Color primitives - raw colors */
:root {
  /* Blues */
  --blue-50: #eff6ff;
  --blue-100: #dbeafe;
  --blue-500: #3b82f6;
  --blue-600: #2563eb;
  --blue-900: #1e3a8a;

  /* Grays */
  --gray-50: #f9fafb;
  --gray-100: #f3f4f6;
  --gray-200: #e5e7eb;
  --gray-600: #4b5563;
  --gray-800: #1f2937;
  --gray-900: #111827;

  /* Status colors */
  --green-500: #10b981;
  --red-500: #ef4444;
  --yellow-500: #f59e0b;
}

/* Semantic tokens - what colors mean */
:root {
  /* Brand colors */
  --color-primary: var(--blue-600);
  --color-primary-hover: var(--blue-500);
  --color-primary-active: var(--blue-900);

  /* Text colors */
  --color-text-primary: var(--gray-900);
  --color-text-secondary: var(--gray-600);
}

```

```
--color-text-inverse: var(--gray-50);

/* Background colors */
--color-bg-primary: #ffffff;
--color-bg-secondary: var(--gray-50);
--color-bg-tertiary: var(--gray-100);

/* Border colors */
--color-border: var(--gray-200);
--color-border-focus: var(--blue-500);

/* Status colors */
--color-success: var(--green-500);
--color-error: var(--red-500);
--color-warning: var(--yellow-500);

/* Typography */
--font-family-base: system-ui, -apple-system, 'Segoe UI', Roboto, sans-serif;
--font-family-mono: 'Courier New', Courier, monospace;

--font-size-xs: 0.75rem; /* 12px */
--font-size-sm: 0.875rem; /* 14px */
--font-size-base: 1rem; /* 16px */
--font-size-lg: 1.125rem; /* 18px */
--font-size-xl: 1.25rem; /* 20px */
--font-size-2xl: 1.5rem; /* 24px */
--font-size-3xl: 1.875rem; /* 30px */

--font-weight-normal: 400;
--font-weight-medium: 500;
--font-weight-semibold: 600;
--font-weight-bold: 700;

--line-height-tight: 1.25;
--line-height-normal: 1.5;
--line-height-relaxed: 1.75;

/* Spacing */
--space-xs: 0.25rem; /* 4px */
--space-sm: 0.5rem; /* 8px */
--space-md: 1rem; /* 16px */
--space-lg: 1.5rem; /* 24px */
--space-xl: 2rem; /* 32px */
--space-2xl: 3rem; /* 48px */
--space-3xl: 4rem; /* 64px */

/* Borders */
```

```

--border-width: 1px;
--border-radius-sm: 0.25rem; /* 4px */
--border-radius-md: 0.5rem; /* 8px */
--border-radius-lg: 1rem; /* 16px */
--border-radius-full: 9999px;

/* Shadows */
--shadow-sm: 0 1px 2px 0 rgba(0, 0, 0, 0.05);
--shadow-md: 0 4px 6px -1px rgba(0, 0, 0, 0.1);
--shadow-lg: 0 10px 15px -3px rgba(0, 0, 0, 0.1);
--shadow-xl: 0 20px 25px -5px rgba(0, 0, 0, 0.1);

/* Transitions */
--transition-fast: 150ms ease-in-out;
--transition-base: 250ms ease-in-out;
--transition-slow: 350ms ease-in-out;

/* Z-index layers */
--z-dropdown: 1000;
--z-sticky: 1020;
--z-fixed: 1030;
--z-modal-backdrop: 1040;
--z-modal: 1050;
--z-popover: 1060;
--z-tooltip: 1070;
}

```

This gives us a two-tier system: primitives (the actual colors) and semantic tokens (what the colors mean). Components use semantic tokens, never primitives directly.

15.1.2 Dark Mode: Inverting Without Inverting

Dark mode isn't just "make everything dark." Good dark mode is subtle, uses slightly muted colors, and maintains contrast ratios for accessibility.

```

/* theme/dark.css */

/* Dark mode overrides */
[data-theme="dark"] {
  /* Text colors */
  --color-text-primary: var(--gray-50);
  --color-text-secondary: var(--gray-200);
  --color-text-inverse: var(--gray-900);

  /* Background colors */
  --color-bg-primary: var(--gray-900);
  --color-bg-secondary: var(--gray-800);
  --color-bg-tertiary: var(--gray-600);
}

```

```

/* Border colors */
--color-border: var(--gray-600);

/* Adjust shadows for dark backgrounds */
--shadow-sm: 0 1px 2px 0 rgba(0, 0, 0, 0.3);
--shadow-md: 0 4px 6px -1px rgba(0, 0, 0, 0.4);
--shadow-lg: 0 10px 15px -3px rgba(0, 0, 0, 0.5);
--shadow-xl: 0 20px 25px -5px rgba(0, 0, 0, 0.6);
}

```

We override only semantic tokens, never primitives. This keeps dark mode maintainable—add a new component, and it automatically works in both themes.

15.1.3 System Preference Detection

Respect the user's OS preference:

```

/* theme/system.css */

/* Automatically use dark mode if system preference is dark */
@media (prefers-color-scheme: dark) {
  :root:not([data-theme="light"]) {
    --color-text-primary: var(--gray-50);
    --color-text-secondary: var(--gray-200);
    --color-text-inverse: var(--gray-900);
    --color-bg-primary: var(--gray-900);
    --color-bg-secondary: var(--gray-800);
    --color-bg-tertiary: var(--gray-600);
    --color-border: var(--gray-600);
    --shadow-sm: 0 1px 2px 0 rgba(0, 0, 0, 0.3);
    --shadow-md: 0 4px 6px -1px rgba(0, 0, 0, 0.4);
    --shadow-lg: 0 10px 15px -3px rgba(0, 0, 0, 0.5);
    --shadow-xl: 0 20px 25px -5px rgba(0, 0, 0, 0.6);
  }
}

```

This respects the system preference unless the user explicitly chooses a theme (via `data-theme` attribute).

15.2 Theme Provider Component

Now let's build a component that manages theme state and publishes changes through the PAN bus:

```

// components/theme-provider.mjs

import { publish, subscribe } from '../pan.js';

class ThemeProvider extends HTMLElement {

```

```

constructor() {
  super();
  this.currentTheme = this.getInitialTheme();
}

connectedCallback() {
  // Apply initial theme
  this.applyTheme(this.currentTheme);

  // Subscribe to theme change requests
  this.subscriptions = [
    subscribe('theme.change', (msg) => {
      this.setTheme(msg.data.theme);
    }),

    subscribe('theme.toggle', () => {
      this.toggleTheme();
    }),

    subscribe('theme.query', () => {
      this.publishCurrentTheme();
    })
  ];

  // Listen for system preference changes
  this.mediaQuery = window.matchMedia('(prefers-color-scheme: dark)');
  this.handleMediaChange = () => {
    if (this.currentTheme === 'system') {
      this.applyTheme('system');
      this.publishCurrentTheme();
    }
  };
  this.mediaQuery.addEventListener('change', this.handleMediaChange);

  // Publish initial theme
  this.publishCurrentTheme();
}

disconnectedCallback() {
  this.subscriptions.forEach(unsub => unsub());
  this.mediaQuery?.removeEventListener('change', this.handleMediaChange);
}

/**
 * Get initial theme from localStorage or system preference
 */
getInitialTheme() {

```

```

    const stored = localStorage.getItem('theme');
    if (stored && ['light', 'dark', 'system'].includes(stored)) {
      return stored;
    }
    return 'system';
  }

  /**
   * Set theme and persist preference
   */
  setTheme(theme) {
    if (!['light', 'dark', 'system'].includes(theme)) {
      console.error(`Invalid theme: ${theme}`);
      return;
    }

    this.currentTheme = theme;
    this.applyTheme(theme);
    localStorage.setItem('theme', theme);
    this.publishCurrentTheme();
  }

  /**
   * Toggle between light and dark (ignoring system preference)
   */
  toggleTheme() {
    const resolvedTheme = this.getResolvedTheme();
    const newTheme = resolvedTheme === 'light' ? 'dark' : 'light';
    this.setTheme(newTheme);
  }

  /**
   * Apply theme to document
   */
  applyTheme(theme) {
    const resolvedTheme = this.resolveTheme(theme);
    document.documentElement.setAttribute('data-theme', resolvedTheme);
  }

  /**
   * Resolve 'system' theme to actual theme
   */
  resolveTheme(theme) {
    if (theme === 'system') {
      return window.matchMedia('(prefers-color-scheme: dark)').matches
        ? 'dark'
        : 'light';
    }
  }

```

```

    }
    return theme;
  }

  /**
   * Get the currently resolved theme (not 'system')
   */
  getResolvedTheme() {
    return document.documentElement.getAttribute('data-theme') || 'light';
  }

  /**
   * Publish current theme state
   */
  publishCurrentTheme() {
    const resolvedTheme = this.getResolvedTheme();
    publish('theme.current', {
      theme: this.currentTheme,
      resolvedTheme,
      isSystemPreference: this.currentTheme === 'system'
    });
  }
}

customElements.define('theme-provider', ThemeProvider);

```

Add it to your HTML:

```

<!DOCTYPE html>
<html>
<head>
  <title>My App</title>
  <link rel="stylesheet" href="./theme/base.css">
  <link rel="stylesheet" href="./theme/dark.css">
  <script type="module" src="./src/pan.js"></script>
</head>
<body>
  <theme-provider></theme-provider>

  <!-- Your app content -->
  <main>
    <h1>Hello, World!</h1>
  </main>
</body>
</html>

```

Now any component can change the theme:

```
import { publish } from '../pan.js';

// Set theme explicitly
publish('theme.change', { theme: 'dark' });

// Toggle theme
publish('theme.toggle', {});

// Query current theme
publish('theme.query', {});
```

15.3 Theme-Aware Components

Components should respond to theme changes by subscribing to `theme.current`:

```
// components/theme-display.mjs

import { subscribe } from '../pan.js';

class ThemeDisplay extends HTMLElement {
  connectedCallback() {
    this.unsubscribe = subscribe('theme.current', (msg) => {
      this.render(msg.data);
    });
  }

  render({ theme, resolvedTheme, isSystemPreference }) {
    this.innerHTML = `
      <div class="theme-display">
        <p>Current theme: <strong>${theme}</strong></p>
        ${isSystemPreference ? `
          <p>Resolved from system: <strong>${resolvedTheme}</strong></p>
        ` : ''}
        <p>Active theme: <strong>${resolvedTheme}</strong></p>
      </div>
    `;
  }

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

customElements.define('theme-display', ThemeDisplay);
```

15.4 Theme Switcher Component

Let's build a polished theme switcher with three options: light, dark, and system:

```
// components/theme-switcher.mjs

import { publish, subscribe } from '../pan.js';

class ThemeSwitcher extends HTMLElement {
  constructor() {
    super();
    this.currentTheme = 'system';
  }

  connectedCallback() {
    this.className = 'theme-switcher';

    // Subscribe to theme updates
    this.unsubscribe = subscribe('theme.current', (msg) => {
      this.currentTheme = msg.data.theme;
      this.render();
    });

    // Request current theme
    publish('theme.query', {});
  }

  render() {
    this.innerHTML = `
      <div class="theme-switcher__container">
        <button
          class="theme-switcher__button ${this.currentTheme === 'light' ? 'active' : ''}"
          data-theme="light"
          aria-label="Light theme"
        >
          [sun] Light
        </button>
        <button
          class="theme-switcher__button ${this.currentTheme === 'dark' ? 'active' : ''}"
          data-theme="dark"
          aria-label="Dark theme"
        >
          [moon] Dark
        </button>
        <button
          class="theme-switcher__button ${this.currentTheme === 'system' ? 'active' : ''}"
          data-theme="system"
          aria-label="System theme"
        >
          [system] System
        </button>
      </div>
    `;
  }
}
```

```

    >
      [laptop] System
    </button>
  </div>
`
;

// Attach event listeners
this.querySelectorAll('[data-theme]').forEach(button => {
  button.addEventListener('click', () => {
    const theme = button.dataset.theme;
    publish('theme.change', { theme });
  });
});
}

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

customElements.define('theme-switcher', ThemeSwitcher);

```

And the styles:

```

/* components/theme-switcher.css */

.theme-switcher__container {
  display: flex;
  gap: var(--space-xs);
  padding: var(--space-xs);
  background: var(--color-bg-secondary);
  border-radius: var(--border-radius-lg);
}

.theme-switcher__button {
  padding: var(--space-sm) var(--space-md);
  border: var(--border-width) solid transparent;
  border-radius: var(--border-radius-md);
  background: transparent;
  color: var(--color-text-secondary);
  font-size: var(--font-size-sm);
  font-weight: var(--font-weight-medium);
  cursor: pointer;
  transition: all var(--transition-fast);
}

```

```
.theme-switcher__button:hover {
  background: var(--color-bg-tertiary);
  color: var(--color-text-primary);
}

.theme-switcher__button.active {
  background: var(--color-primary);
  color: var(--color-text-inverse);
  border-color: var(--color-primary);
}

.theme-switcher__button:focus-visible {
  outline: 2px solid var(--color-border-focus);
  outline-offset: 2px;
}
```

15.5 Smooth Transitions Between Themes

Switching themes can be jarring. Let's add smooth transitions:

```
/* theme/transitions.css */

/* Transition all themed properties */

* {
  transition:
    background-color var(--transition-base),
    border-color var(--transition-base),
    color var(--transition-base),
    box-shadow var(--transition-base);
}

/* Disable transitions during page load */
.no-transitions * {
  transition: none !important;
}

/* Respect user preference for reduced motion */
@media (prefers-reduced-motion: reduce) {
  * {
    transition: none !important;
  }
}
```

Update the theme provider to disable transitions during initial load:

```

class ThemeProvider extends HTMLElement {
  connectedCallback() {
    // Disable transitions during initial load
    document.documentElement.classList.add('no-transitions');

    this.applyTheme(this.currentTheme);

    // Re-enable transitions after a frame
    requestAnimationFrame(() => {
      document.documentElement.classList.remove('no-transitions');
    });

    // ... rest of connectedCallback
  }
}

```

15.6 Scoped Themes for Components

Sometimes a component needs its own theme, independent of the global theme:

```

// components/branded-card.mjs

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

    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;

          /* Define local theme */
          --card-bg: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
          --card-text: #ffffff;
          --card-border-radius: 1rem;
          --card-padding: 2rem;
        }

        .card {
          background: var(--card-bg);
          color: var(--card-text);
          border-radius: var(--card-border-radius);
          padding: var(--card-padding);
          box-shadow: var(--shadow-xl);
        }

        /* Allow customization via CSS custom properties */

```

```

        :host([variant="flat"]) {
            --card-bg: var(--color-bg-secondary);
            --card-text: var(--color-text-primary);
        }
    </style>

    <div class="card">
        <slot></slot>
    </div>
    `;
}
}

customElements.define('branded-card', BrandedCard);

```

Use it:

```

<!-- Uses gradient background -->
<branded-card>
    <h2>Premium Feature</h2>
    <p>This card has its own theme!</p>
</branded-card>

<!-- Uses flat background -->
<branded-card variant="flat">
    <h2>Standard Feature</h2>
    <p>This one inherits from global theme.</p>
</branded-card>

```

15.7 Responsive Design with Custom Properties

CSS custom properties can adapt to viewport size:

```

/* theme/responsive.css */

:root {
    /* Base spacing */
    --space-page-horizontal: var(--space-md);
    --space-page-vertical: var(--space-lg);

    /* Base font sizes */
    --font-size-display: var(--font-size-2xl);
}

/* Tablet and up */
@media (min-width: 768px) {
    :root {
        --space-page-horizontal: var(--space-xl);
    }
}

```

```

    --space-page-vertical: var(--space-2xl);
    --font-size-display: var(--font-size-3xl);
  }
}

/* Desktop and up */
@media (min-width: 1024px) {
  :root {
    --space-page-horizontal: var(--space-2xl);
    --space-page-vertical: var(--space-3xl);
    --font-size-display: 2.5rem;
  }
}

/* Container */
.container {
  max-width: 1280px;
  margin: 0 auto;
  padding: var(--space-page-vertical) var(--space-page-horizontal);
}

/* Display text */
.display {
  font-size: var(--font-size-display);
  line-height: var(--line-height-tight);
  font-weight: var(--font-weight-bold);
}

```

Components automatically adapt to these responsive tokens.

15.8 Multiple Brand Themes

Support multiple brands with theme switching:

```

/* theme/brands.css */

/* Default brand (Acme Corp) */
:root {
  --brand-primary: #667eea;
  --brand-secondary: #764ba2;
  --brand-logo-url: url('/logos/acme.svg');
}

/* Brand: TechStart */
[data-brand="techstart"] {
  --brand-primary: #10b981;
  --brand-secondary: #059669;
  --brand-logo-url: url('/logos/techstart.svg');
}

```

```

}

/* Brand: Creative Co */
[data-brand="creative"] {
  --brand-primary: #f59e0b;
  --brand-secondary: #d97706;
  --brand-logo-url: url('/logos/creative.svg');
}

/* Apply brand colors to components */
.button-primary {
  background: var(--brand-primary);
}

.button-secondary {
  background: var(--brand-secondary);
}

.logo {
  content: var(--brand-logo-url);
}

```

Switch brands dynamically:

```

// Set brand via data attribute
document.documentElement.setAttribute('data-brand', 'techstart');

```

Or extend the theme provider to manage brands:

```

class ThemeProvider extends HTMLElement {
  setBrand(brand) {
    document.documentElement.setAttribute('data-brand', brand);
    publish('theme.brand.changed', { brand });
  }
}

```

15.9 Performance Considerations

CSS custom properties are fast, but here are tips to keep themes performant:

15.9.1 1. Minimize Transitions

Don't transition everything:

```

/* Bad: transitions on every property */

* {
  transition: all var(--transition-base);
}

```

```
/* Good: transition only themed properties */

* {
  transition:
    background-color var(--transition-base),
    color var(--transition-base);
}
```

15.9.2 2. Use CSS Containment

Help browsers optimize rendering:

```
.card {
  contain: layout style;
}
```

15.9.3 3. Avoid Deep Custom Property Lookups

Custom properties have inheritance cost. Don't nest too deeply:

```
/* Bad: deep nesting */
:root {
  --color-1: #333;
  --color-2: var(--color-1);
  --color-3: var(--color-2);
  --color-4: var(--color-3);
}

/* Good: direct references */
:root {
  --base-gray: #333;
  --color-text: var(--base-gray);
  --color-border: var(--base-gray);
}
```

15.9.4 4. Batch Theme Changes

If changing multiple properties, use a data attribute rather than individual properties:

```
// Bad: multiple property changes
document.documentElement.style.setProperty('--color-primary', '#fff');
document.documentElement.style.setProperty('--color-secondary', '#000');
document.documentElement.style.setProperty('--color-text', '#333');

// Good: single attribute change
document.documentElement.setAttribute('data-theme', 'dark');
```

15.10 Accessibility in Theming

Ensure your themes are accessible:

15.10.1 1. Maintain Contrast Ratios

Use tools like WebAIM's contrast checker. Aim for:

- **4.5:1** for normal text (WCAG AA)
- **7:1** for normal text (WCAG AAA)
- **3:1** for large text (WCAG AA)

```
/* Good contrast */
:root {
  --color-text-primary: #111827; /* Dark on light */
  --color-bg-primary: #ffffff;
}

[data-theme="dark"] {
  --color-text-primary: #f9fafb; /* Light on dark */
  --color-bg-primary: #111827;
}
```

15.10.2 2. Respect Reduced Motion

```
@media (prefers-reduced-motion: reduce) {

  * {
    animation-duration: 0.01ms !important;
    animation-iteration-count: 1 !important;
    transition-duration: 0.01ms !important;
  }
}
```

15.10.3 3. Provide High Contrast Mode

```
@media (prefers-contrast: high) {
  :root {
    --color-text-primary: #000000;
    --color-bg-primary: #ffffff;
    --color-border: #000000;
    --border-width: 2px;
  }
}
```

15.10.4 4. Test with Screen Readers

Ensure theme changes are announced:

```

class ThemeProvider extends HTMLElement {
  applyTheme(theme) {
    const resolvedTheme = this.resolveTheme(theme);
    document.documentElement.setAttribute('data-theme', resolvedTheme);

    // Announce theme change to screen readers
    this.announceThemeChange(resolvedTheme);
  }

  announceThemeChange(theme) {
    const announcement = document.createElement('div');
    announcement.setAttribute('role', 'status');
    announcement.setAttribute('aria-live', 'polite');
    announcement.className = 'sr-only';
    announcement.textContent = `Theme changed to ${theme} mode`;
    document.body.appendChild(announcement);

    setTimeout(() => announcement.remove(), 1000);
  }
}

```

With screen-reader-only CSS:

```

.sr-only {
  position: absolute;
  width: 1px;
  height: 1px;
  padding: 0;
  margin: -1px;
  overflow: hidden;
  clip: rect(0, 0, 0, 0);
  white-space: nowrap;
  border-width: 0;
}

```

15.11 Wrapping Up

You now have a robust theming system for LARC applications. You understand CSS custom properties, how to implement light and dark modes that respect system preferences, how to build a theme provider that broadcasts changes through the PAN bus, and how to create accessible, performant themes.

The key insights:

- Use two-tier token system: primitives and semantic tokens
- Components use semantic tokens, never raw colors
- Theme provider manages state and publishes to PAN bus
- Respect system preferences but allow explicit overrides

- Smooth transitions make theme switching delightful
- Accessibility isn't optional—contrast, reduced motion, and screen readers matter
- CSS custom properties are fast; use them liberally

In the next chapter, we'll tackle performance optimization—making LARC applications fast through message filtering, lazy loading, virtual scrolling, and careful memory management. Because a beautiful theme doesn't matter if your app takes ten seconds to load.

Now go forth and theme your application. And remember: if users complain about your color choices, you can always blame it on “brand guidelines.”

Chapter 16

Performance Optimization

“Premature optimization is the root of all evil. But shipping a slow application is the root of losing all your users.”

— Donald Knuth (paraphrased by someone who’s watched users abandon slow apps)

Performance isn’t about making your application fast—it’s about making it feel fast. Users don’t care if your message bus can handle 10,000 messages per second if clicking a button takes three seconds to respond. They don’t care if your virtual DOM is optimized if the initial page load shows a blank screen for five seconds.

In this chapter, we’ll explore performance optimization strategies specific to LARC applications: efficient message filtering and routing, component lazy loading, virtual scrolling for massive lists, debouncing and throttling high-frequency events, memory management to prevent leaks, and bundle size optimization. By the end, you’ll know how to build LARC applications that are not just correct, but fast.

16.1 Message Filtering and Routing Efficiency

The PAN bus is central to LARC applications. Every publish triggers subscriptions, and inefficient patterns can create performance bottlenecks.

16.1.1 Pattern: Specific Topic Subscriptions

Subscribe to specific topics, not wildcards, when possible:

```
// Bad: too broad
subscribe('*', (msg) => {
  if (msg.topic.startsWith('user.')) {
    // Handle user messages
  }
});
```

```
// Good: specific subscription
subscribe('user.*', (msg) => {
  // Only receives user messages
});
```

```
});

// Better: most specific possible
subscribe('user.profile.updated', (msg) => {
  // Only receives profile updates
});
```

Specific subscriptions reduce unnecessary function calls.

16.1.2 Pattern: Early Returns

Return early from subscription handlers when the message isn't relevant:

```
subscribe('user.data', (msg) => {
  // Early return if not our user
  if (msg.data.userId !== this.currentUserId) {
    return;
  }

  // Expensive processing only for relevant messages
  this.processUserData(msg.data);
});
```

16.1.3 Pattern: Unsubscribe Aggressively

Unsubscribe as soon as you no longer need messages:

```
class TemporaryComponent extends HTMLElement {
  connectedCallback() {
    // Subscribe to one-time event
    this.unsubscribe = subscribe('data.loaded', (msg) => {
      this.render(msg.data);

      // Unsubscribe immediately after first message
      this.unsubscribe();
      this.unsubscribe = null;
    });
  }

  disconnectedCallback() {
    // Clean up if component removed before message received
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }
}
```

16.1.4 Implementing Message Throttling

Throttle high-frequency messages at the source:

```
class MouseTracker extends HTMLElement {
  constructor() {
    super();
    this.lastPublishTime = 0;
    this.publishInterval = 50; // Publish at most every 50ms (20 FPS)
  }

  connectedCallback() {
    this.addEventListener('mousemove', this.handleMouseMove.bind(this));
  }

  handleMouseMove(event) {
    const now = Date.now();

    // Throttle: only publish if enough time has passed
    if (now - this.lastPublishTime < this.publishInterval) {
      return;
    }

    this.lastPublishTime = now;

    publish('mouse.position', {
      x: event.clientX,
      y: event.clientY,
      timestamp: now
    });
  }
}

customElements.define('mouse-tracker', MouseTracker);
```

16.1.5 Debouncing Message Publishers

For user input, debounce to reduce message frequency:

```
class SearchInput extends HTMLElement {
  constructor() {
    super();
    this.debounceTimer = null;
    this.debounceDelay = 300; // Wait 300ms after last keystroke
  }

  connectedCallback() {
    this.innerHTML = `
      <input type="text" placeholder="Search..." />
    `;
  }
}
```

```

    `;

    this.querySelector('input').addEventListener('input', (event) => {
        this.handleInput(event.target.value);
    });
}

handleInput(value) {
    // Clear previous timer
    clearTimeout(this.debounceTimer);

    // Set new timer
    this.debounceTimer = setTimeout(() => {
        publish('search.query', { query: value });
    }, this.debounceDelay);
}

customElements.define('search-input', SearchInput);

```

16.1.6 Batching Messages

When publishing multiple related messages, batch them:

```

class BulkUpdater extends HTMLElement {
    updateMultipleItems(items) {
        // Bad: publish once per item
        // items.forEach(item => {
        //     publish('item.updated', item);
        // });

        // Good: batch into single message
        publish('items.updated', { items });
    }
}

```

Subscribers process the batch:

```

class ItemList extends HTMLElement {
    connectedCallback() {
        this.unsubscribe = subscribe('items.updated', (msg) => {
            // Process entire batch at once
            this.updateItems(msg.data.items);
        });
    }

    updateItems(items) {
        // Batch DOM updates
        const fragment = document.createDocumentFragment();
    }
}

```

```

    items.forEach(item => {
      const li = document.createElement('li');
      li.textContent = item.name;
      fragment.appendChild(li);
    });

    this.querySelector('ul').innerHTML = '';
    this.querySelector('ul').appendChild(fragment);
  }
}

```

16.2 Component Lazy Loading

Load components only when needed. LARC's autoloader already does this for components near the viewport, but you can optimize further.

16.2.1 Lazy Loading Off-Screen Components

Use IntersectionObserver to load components when they approach the viewport:

```

// components/lazy-loader.mjs

class LazyLoader extends HTMLElement {
  constructor() {
    super();
    this.loaded = false;
  }

  connectedCallback() {
    const componentName = this.getAttribute('component');
    const loadDistance = parseInt(this.getAttribute('load-distance') || '600');

    if (!componentName) {
      console.error('LazyLoader: component attribute required');
      return;
    }

    // Observe element
    this.observer = new IntersectionObserver(
      (entries) => {
        entries.forEach(entry => {
          if (entry.isIntersecting && !this.loaded) {
            this.loadComponent(componentName);
          }
        });
      },
      { rootMargin: `${loadDistance}px` }
    );
  }
}

```

```

    );

    this.observer.observe(this);
  }

  async loadComponent(componentName) {
    this.loaded = true;
    this.observer.disconnect();

    try {
      // Show loading state
      this.innerHTML = '<div class="loading">Loading...</div>';

      // Dynamically import component
      await import(`./components/${componentName}.mjs`);

      // Replace loader with actual component
      const component = document.createElement(componentName);

      // Copy attributes to component
      for (const attr of this.attributes) {
        if (attr.name !== 'component' && attr.name !== 'load-distance') {
          component.setAttribute(attr.name, attr.value);
        }
      }

      this.innerHTML = '';
      this.appendChild(component);
    } catch (error) {
      console.error(`Failed to load component ${componentName}:`, error);
      this.innerHTML = '<div class="error">Failed to load component</div>';
    }
  }

  disconnectedCallback() {
    if (this.observer) {
      this.observer.disconnect();
    }
  }
}

customElements.define('lazy-loader', LazyLoader);

```

Use it:

```

<!-- Component loads when it approaches viewport -->
<lazy-loader component="heavy-chart" load-distance="400"></lazy-loader>

```

```

<!-- Multiple lazy components -->
<lazy-loader component="user-profile"></lazy-loader>
<lazy-loader component="activity-feed"></lazy-loader>
<lazy-loader component="notifications-panel"></lazy-loader>

```

16.2.2 Code Splitting Routes

Split application by routes:

```

// components/app-router.mjs

import { subscribe } from '../pan.js';

class AppRouter extends HTMLElement {
  constructor() {
    super();
    this.currentRoute = null;
    this.loadedComponents = new Set();
  }

  connectedCallback() {
    this.unsubscribe = subscribe('route.change', async (msg) => {
      await this.loadRoute(msg.data.route);
    });
  }

  async loadRoute(route) {
    if (this.currentRoute === route) {
      return;
    }

    this.currentRoute = route;

    // Show loading state
    this.innerHTML = '<div class="route-loading">Loading page...</div>';

    try {
      // Lazy load route component
      const componentName = this.getComponentForRoute(route);

      if (!this.loadedComponents.has(componentName)) {
        await import(`../pages/${componentName}.mjs`);
        this.loadedComponents.add(componentName);
      }

      // Render route component
      this.innerHTML = `<${componentName}></${componentName}>`;
    } catch (error) {

```

```

        console.error(`Failed to load route ${route}:`, error);
        this.innerHTML = '<div class="error">Page not found</div>';
    }
}

getComponentForRoute(route) {
    const routeMap = {
        '/': 'home-page',
        '/profile': 'profile-page',
        '/settings': 'settings-page',
        '/dashboard': 'dashboard-page'
    };

    return routeMap[route] || 'not-found-page';
}

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

customElements.define('app-router', AppRouter);

```

16.3 Virtual Scrolling for Large Lists

Rendering thousands of DOM elements is slow. Virtual scrolling renders only visible items.

Here's a robust virtual list implementation:

```

// components/virtual-list.mjs

import { subscribe } from '../pan.js';

class VirtualList extends HTMLElement {
    constructor() {
        super();
        this.items = [];
        this.itemHeight = 50; // Default height
        this.visibleCount = 20;
        this.scrollTop = 0;
        this.startIndex = 0;
        this.endIndex = 20;
        this.containerHeight = 800;
    }
}

```

```

static get observedAttributes() {
  return ['item-height', 'container-height'];
}

attributeChangedCallback(name, oldValue, newValue) {
  if (name === 'item-height') {
    this.itemHeight = parseInt(newValue);
  } else if (name === 'container-height') {
    this.containerHeight = parseInt(newValue);
  }

  if (oldValue !== newValue) {
    this.render();
  }
}

connectedCallback() {
  const topic = this.getAttribute('topic') || 'list.items';

  this.unsubscribe = subscribe(topic, (msg) => {
    this.items = msg.data.items || [];
    this.render();
  });

  this.render();
}

render() {
  // Calculate visible range
  this.visibleCount = Math.ceil(this.containerHeight / this.itemHeight) + 2; // Buffer
  this.startIndex = Math.max(0, Math.floor(this.scrollTop / this.itemHeight) - 1);
  this.endIndex = Math.min(this.items.length, this.startIndex + this.visibleCount);

  const visibleItems = this.items.slice(this.startIndex, this.endIndex);
  const totalHeight = this.items.length * this.itemHeight;
  const offsetY = this.startIndex * this.itemHeight;

  this.innerHTML = `
    <div class="virtual-list-container" style="height: ${this.containerHeight}px; overflow-y:
      <div class="virtual-list-spacer" style="height: ${totalHeight}px; position: relative;"
        <div class="virtual-list-content" style="position: absolute; top: ${offsetY}px; width:
          ${this.renderItems(visibleItems)}
        </div>
      </div>
    </div>
  `;

```

```

    // Attach scroll handler
    const container = this.querySelector('.virtual-list-container');
    container.addEventListener('scroll', this.handleScroll.bind(this));

    // Restore scroll position
    container.scrollTop = this.scrollTop;
  }

  renderItems(items) {
    return items.map((item, index) => {
      const globalIndex = this.startIndex + index;
      return `
        <div class="virtual-list-item" style="height: ${this.itemHeight}px;" data-index="${globalIndex}">
          ${this.renderItem(item, globalIndex)}
        </div>
      `;
    }).join('');
  }

  renderItem(item, index) {
    // Override this method to customize item rendering
    return `
      <div style="padding: 12px; border-bottom: 1px solid #ddd;">
        <strong>#${index + 1}</strong>: ${item.name || item.title || JSON.stringify(item)}
      </div>
    `;
  }

  handleScroll(event) {
    const newScrollTop = event.target.scrollTop;

    // Only re-render if we've scrolled enough
    if (Math.abs(newScrollTop - this.scrollTop) > this.itemHeight) {
      this.scrollTop = newScrollTop;
      this.render();
    } else {
      this.scrollTop = newScrollTop;
    }
  }

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

```

```
customElements.define('virtual-list', VirtualList);
```

Use it:

```
<virtual-list
  topic="users.list"
  item-height="60"
  container-height="600"
></virtual-list>
```

Publish items:

```
import { publish } from './pan.js';

// Generate 10,000 items
const items = Array.from({ length: 10000 }, (_, i) => ({
  id: i,
  name: `User ${i}`,
  email: `user${i}@example.com`
}));

publish('users.list', { items });
```

The virtual list renders only ~22 items at a time, regardless of whether there are 100 or 100,000 items.

16.3.1 Dynamic Item Heights

For variable-height items, maintain a height cache:

```
class DynamicVirtualList extends VirtualList {
  constructor() {
    super();
    this.itemHeights = new Map(); // Cache of measured heights
    this.averageHeight = 50;
  }

  render() {
    // Calculate positions using cached heights
    let offsetY = 0;
    let startIndex = 0;

    for (let i = 0; i < this.items.length; i++) {
      const height = this.itemHeights.get(i) || this.averageHeight;

      if (offsetY + height < this.scrollTop) {
        offsetY += height;
        startIndex = i + 1;
      } else if (offsetY > this.scrollTop + this.containerHeight) {
        break;
      }
    }
  }
}
```

```

    }
  }

  this.startIndex = startIndex;
  this.endIndex = Math.min(this.items.length, startIndex + this.visibleCount);

  // Rest of rendering...
  // After rendering, measure actual heights and cache them
  this.measureItemHeights();
}

measureItemHeights() {
  requestAnimationFrame(() => {
    const items = this.querySelectorAll('.virtual-list-item');
    items.forEach((item, index) => {
      const globalIndex = this.startIndex + index;
      const height = item.offsetHeight;
      this.itemHeights.set(globalIndex, height);
    });
  });
}
}

customElements.define('dynamic-virtual-list', DynamicVirtualList);

```

16.4 Debouncing and Throttling

We've seen throttling and debouncing briefly. Let's explore them deeply.

16.4.1 Debounce Utility

Create a reusable debounce utility:

```

// utils/debounce.js

/**
 * Debounce function - waits for delay after last call
 * @param {Function} fn - Function to debounce
 * @param {number} delay - Delay in milliseconds
 * @returns {Function} Debounced function
 */
export function debounce(fn, delay) {
  let timeoutId = null;

  const debounced = function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {

```

```

    fn.apply(this, args);
  }, delay);
};

// Add cancel method
debounced.cancel = function() {
  clearTimeout(timeoutId);
};

return debounced;
}

```

Use it:

```

import { debounce } from '../utils/debounce.js';

class SearchBox extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `<input type="text" placeholder="Search..." />`;

    const input = this.querySelector('input');

    // Debounce search
    const debouncedSearch = debounce((value) => {
      publish('search.query', { query: value });
    }, 300);

    input.addEventListener('input', (e) => {
      debouncedSearch(e.target.value);
    });
  }
}

```

16.4.2 Throttle Utility

Create a reusable throttle utility:

```

// utils/throttle.js

/**
 * Throttle function - ensures function runs at most once per interval
 * @param {Function} fn - Function to throttle
 * @param {number} interval - Minimum interval between calls
 * @returns {Function} Throttled function
 */
export function throttle(fn, interval) {
  let lastCall = 0;
  let timeoutId = null;

```

```

const throttled = function(...args) {
  const now = Date.now();
  const timeSinceLastCall = now - lastCall;

  if (timeSinceLastCall >= interval) {
    lastCall = now;
    fn.apply(this, args);
  } else {
    // Schedule call for end of interval
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      lastCall = Date.now();
      fn.apply(this, args);
    }, interval - timeSinceLastCall);
  }
};

// Add cancel method
throttled.cancel = function() {
  clearTimeout(timeoutId);
};

return throttled;
}

```

Use it:

```

import { throttle } from '../utils/throttle.js';

class ScrollTracker extends HTMLElement {
  connectedCallback() {
    // Throttle scroll events to 100ms (10 FPS)
    const throttledScroll = throttle(() => {
      publish('scroll.position', {
        x: window.scrollX,
        y: window.scrollY
      });
    }, 100);

    window.addEventListener('scroll', throttledScroll);

    this.cleanup = () => {
      window.removeEventListener('scroll', throttledScroll);
    };
  }

  disconnectedCallback() {
    if (this.cleanup) {

```

```

        this.cleanup();
    }
}

```

16.4.3 RequestAnimationFrame Throttling

For animation-related events, use requestAnimationFrame:

```

class RAFThrottle {
  constructor(callback) {
    this.callback = callback;
    this.rafId = null;
    this.lastArgs = null;
  }

  trigger(...args) {
    this.lastArgs = args;

    if (this.rafId === null) {
      this.rafId = requestAnimationFrame(() => {
        this.callback.apply(null, this.lastArgs);
        this.rafId = null;
      });
    }
  }

  cancel() {
    if (this.rafId !== null) {
      cancelAnimationFrame(this.rafId);
      this.rafId = null;
    }
  }
}

```

Use it:

```

class SmoothScroller extends HTMLElement {
  connectedCallback() {
    this.rafThrottle = new RAFThrottle(() => {
      publish('scroll.position', {
        x: window.scrollX,
        y: window.scrollY
      });
    });

    window.addEventListener('scroll', () => {
      this.rafThrottle.trigger();
    });
  }
}

```

```
}

disconnectedCallback() {
  if (this.rafThrottle) {
    this.rafThrottle.cancel();
  }
}
}
```

16.5 Memory Management

JavaScript has garbage collection, but you can still leak memory. Here's how to avoid it.

16.5.1 Pattern: Clean Up Subscriptions

Always unsubscribe in `disconnectedCallback`:

```
class LeakFreeComponent extends HTMLElement {
  connectedCallback() {
    this.subscriptions = [
      subscribe('topic.one', this.handleOne.bind(this)),
      subscribe('topic.two', this.handleTwo.bind(this)),
      subscribe('topic.three', this.handleThree.bind(this))
    ];
  }

  disconnectedCallback() {
    // Clean up all subscriptions
    this.subscriptions.forEach(unsub => unsub());
    this.subscriptions = [];
  }
}
```

16.5.2 Pattern: Remove Event Listeners

Always remove event listeners:

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

  handleClick(event) {
    publish('click.tracked', { x: event.clientX, y: event.clientY });
  }

  disconnectedCallback() {
    this.removeEventListener('click', this.handleClick);
  }
}
```

```
    this.removeEventListener('click', this.handleClick);
  }
}
```

16.5.3 Pattern: Clear Timers

Clear all timers and intervals:

```
class PeriodicUpdater extends HTMLElement {
  connectedCallback() {
    this.intervalId = setInterval(() => {
      this.update();
    }, 5000);
  }

  disconnectedCallback() {
    clearInterval(this.intervalId);
  }
}
```

16.5.4 Pattern: Cancel Pending Promises

Track and cancel pending async operations:

```
class DataFetcher extends HTMLElement {
  constructor() {
    super();
    this.abortController = null;
  }

  async connectedCallback() {
    await this.fetchData();
  }

  async fetchData() {
    // Cancel previous request if still pending
    if (this.abortController) {
      this.abortController.abort();
    }

    this.abortController = new AbortController();

    try {
      const response = await fetch('/api/data', {
        signal: this.abortController.signal
      });

      const data = await response.json();
    }
  }
}
```

```
    this.render(data);
  } catch (error) {
    if (error.name === 'AbortError') {
      console.log('Fetch cancelled');
    } else {
      console.error('Fetch failed:', error);
    }
  }
}

disconnectedCallback() {
  if (this.abortController) {
    this.abortController.abort();
  }
}
}
```

16.5.5 Pattern: Weak References for Caches

Use WeakMap for caches tied to object lifetimes:

```
class ComponentCache {
  constructor() {
    this.cache = new WeakMap();
  }

  get(element) {
    return this.cache.get(element);
  }

  set(element, data) {
    this.cache.set(element, data);
  }

  // No need for cleanup - garbage collected automatically
}

const componentCache = new ComponentCache();
```

16.6 Bundle Size Optimization

Smaller bundles load faster. Here's how to minimize size.

16.6.1 1. Tree Shaking

Ensure your modules are tree-shakeable by using ES6 imports/exports:

```
// Good: named exports (tree-shakeable)
export function used() { /* ... */ }
export function unused() { /* ... */ }

// Consumer imports only what they need
import { used } from './utils.js';
```

16.6.2 2. Dynamic Imports

Load code on demand:

```
class FeatureToggle extends HTMLElement {
  async enableFeature() {
    // Load feature code only when enabled
    const { AdvancedFeature } = await import('./advanced-feature.js');

    const feature = new AdvancedFeature();
    feature.activate();
  }
}
```

16.6.3 3. Avoid Large Dependencies

Check dependency sizes before adding them:

```
# Use bundlephobia to check size
npm install -g bundle-phobia-cli
bundle-phobia moment # Shows: 231 kB minified
```

Consider alternatives:

```
// Heavy: moment.js (231 kB)
import moment from 'moment';
const date = moment().format('YYYY-MM-DD');

// Light: native Intl API (0 kB)
const date = new Intl.DateTimeFormat('en-CA').format(new Date());
```

16.6.4 4. Code Splitting by Route

We saw this earlier—split by route:

```
// pages/index.js - loads only home page code
export { default as HomePage } from './home-page.js';

// pages/dashboard.js - loads only dashboard code
export { default as DashboardPage } from './dashboard-page.js';
```

16.6.5 5. Minification

Use a minifier for production:

```
{
  "scripts": {
    "build": "esbuild src/app.js --bundle --minify --outfile=dist/app.js"
  }
}
```

16.6.6 6. Compression

Enable gzip or brotli compression on your server:

```
// server.js (Express example)
import compression from 'compression';
import express from 'express';

const app = express();

// Enable compression
app.use(compression());

app.use(express.static('dist'));
```

16.7 Performance Monitoring

Measure performance to know what to optimize:

```
// components/performance-monitor.mjs

class PerformanceMonitor extends HTMLElement {
  connectedCallback() {
    // Monitor navigation timing
    this.reportNavigationTiming();

    // Monitor long tasks
    this.observeLongTasks();

    // Monitor message bus performance
    this.monitorMessageBus();
  }

  reportNavigationTiming() {
    window.addEventListener('load', () => {
      const timing = performance.getEntriesByType('navigation')[0];

      console.log('Performance Metrics:', {
        'DNS Lookup': `${timing.domainLookupEnd - timing.domainLookupStart}ms`,

```

```

    'TCP Connection': `${timing.connectEnd - timing.connectStart}ms`,
    'Request': `${timing.responseStart - timing.requestStart}ms`,
    'Response': `${timing.responseEnd - timing.responseStart}ms`,
    'DOM Processing': `${timing.domComplete - timing.domLoading}ms`,
    'Total Load Time': `${timing.loadEventEnd - timing.fetchStart}ms`
  });

  publish('performance.navigation', {
    dnsLookup: timing.domainLookupEnd - timing.domainLookupStart,
    connection: timing.connectEnd - timing.connectStart,
    request: timing.responseStart - timing.requestStart,
    response: timing.responseEnd - timing.responseStart,
    domProcessing: timing.domComplete - timing.domLoading,
    totalLoadTime: timing.loadEventEnd - timing.fetchStart
  });
});
}

observeLongTasks() {
  if ('PerformanceObserver' in window) {
    const observer = new PerformanceObserver((list) => {
      for (const entry of list.getEntries()) {
        console.warn('Long Task Detected:', {
          duration: `${entry.duration}ms`,
          startTime: entry.startTime
        });

        publish('performance.long-task', {
          duration: entry.duration,
          startTime: entry.startTime
        });
      }
    });

    observer.observe({ entryTypes: ['longtask'] });
  }
}

monitorMessageBus() {
  // Wrap publish to measure performance
  const originalPublish = window.publish;

  window.publish = function(topic, data) {
    const start = performance.now();
    const result = originalPublish.call(this, topic, data);
    const duration = performance.now() - start;
  }
}

```

```
    if (duration > 16) { // More than one frame
      console.warn(`Slow message publish: ${topic} took ${duration.toFixed(2)}ms`);
    }

    return result;
  };
}
}

customElements.define('performance-monitor', PerformanceMonitor);
```

16.8 Wrapping Up

Performance optimization in LARC applications comes down to a few key principles:

1. **Message efficiency:** Subscribe specifically, unsubscribe aggressively, throttle/debounce high-frequency events
2. **Lazy loading:** Load components and routes only when needed
3. **Virtual scrolling:** Render only visible items for large lists
4. **Memory management:** Clean up subscriptions, listeners, timers, and pending operations
5. **Bundle optimization:** Tree shake, dynamic import, minimize dependencies, compress output

Performance isn't a one-time task—it's ongoing. Profile regularly, measure what matters (user-perceived performance), and optimize the bottlenecks, not the code you think might be slow.

In the next chapter, we'll tackle testing strategies—unit tests, integration tests, E2E tests, and how to test message-driven architectures without losing your mind. Because fast code that doesn't work is still useless.

Now go forth and optimize. And remember: the fastest code is code that never runs. But users expect your app to do something, so optimize the code that does run.

Testing Strategies

— Edsger W. Dijkstra (with career advice added)

In this chapter, we'll explore testing strategies for LARC applications: unit testing web components, integration testing message flows, end-to-end testing with complete PAN applications, mocking the message bus, testing async operations, and building test utilities that make testing a joy rather than a chore.

17.1 The Testing Pyramid for LARC

The diagram is a pyramid divided into three horizontal sections. The top section is the smallest, the middle section is medium-sized, and the bottom section is the largest. To the right of each section is a label, and to the left of the bottom section is a list of characteristics.

Test Type	Quantity	Characteristics
E2E Tests	Few	- Full application - Real browser - Slow, brittle
Integration Tests	Some	- Multiple components - Message flows - Medium speed
Unit Tests	Many	- Single components

- Pure functions
- Fast, focused

Most tests should be unit tests. Fewer integration tests. Even fewer E2E tests.

17.2 Unit Testing Components

Unit tests verify individual components in isolation. Let's use Vitest (modern, fast) or Mocha (classic, reliable).

17.2.1 Setting Up Vitest

```
npm install -D vitest happy-dom
```

Create a test configuration:

```
// vitest.config.js

import { defineConfig } from 'vitest/config';

export default defineConfig({
  test: {
    environment: 'happy-dom',
    globals: true,
    setupFiles: ['./tests/setup.js']
  }
});
```

Create test setup:

```
// tests/setup.js

import { beforeEach, afterEach } from 'vitest';

// Clean up DOM after each test
afterEach(() => {
  document.body.innerHTML = '';
});
```

17.2.2 Testing a Simple Component

Let's test a counter component:

```
// components/counter-button.mjs

class CounterButton extends HTMLElement {
  constructor() {
    super();
    this.count = 0;
  }
}
```

```

connectedCallback() {
  this.render();
}

increment() {
  this.count++;
  this.render();
}

render() {
  this.innerHTML = `
    <button id="increment">
      Count: ${this.count}
    </button>
  `;

  this.querySelector('#increment').addEventListener('click', () => {
    this.increment();
  });
}
}

customElements.define('counter-button', CounterButton);

export { CounterButton };

```

Test it:

```

// tests/counter-button.test.js

import { describe, it, expect, beforeEach } from 'vitest';
import { CounterButton } from '../components/counter-button.mjs';

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

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

  it('should render with initial count of 0', () => {
    expect(element.count).toBe(0);
    expect(element.textContent).toContain('Count: 0');
  });

  it('should increment count when button is clicked', () => {

```

```

    const button = element.querySelector('button');

    button.click();
    expect(element.count).toBe(1);
    expect(element.textContent).toContain('Count: 1');

    button.click();
    expect(element.count).toBe(2);
    expect(element.textContent).toContain('Count: 2');
  });

  it('should call increment method when clicked', () => {
    const incrementSpy = vi.spyOn(element, 'increment');
    const button = element.querySelector('button');

    button.click();

    expect(incrementSpy).toHaveBeenCalledTimes(1);
  });
});

```

Run tests:

```
npm test
```

17.2.3 Testing Components with Attributes

```

// components/user-badge.mjs

class UserBadge extends HTMLElement {
  static get observedAttributes() {
    return ['username', 'role'];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (oldValue !== newValue) {
      this.render();
    }
  }

  connectedCallback() {
    this.render();
  }

  render() {
    const username = this.getAttribute('username') || 'Anonymous';
    const role = this.getAttribute('role') || 'User';
  }
}

```

```

    this.innerHTML = `
      <div class="user-badge">
        <span class="username">${username}</span>
        <span class="role">${role}</span>
      </div>
    `;
  }
}

customElements.define('user-badge', UserBadge);
export { UserBadge };

```

Test it:

```

// tests/user-badge.test.js

import { describe, it, expect, beforeEach } from 'vitest';
import { UserBadge } from '../components/user-badge.mjs';

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

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

  it('should render with default values', () => {
    expect(element.textContent).toContain('Anonymous');
    expect(element.textContent).toContain('User');
  });

  it('should render with provided attributes', () => {
    element.setAttribute('username', 'Alice');
    element.setAttribute('role', 'Admin');

    expect(element.textContent).toContain('Alice');
    expect(element.textContent).toContain('Admin');
  });

  it('should update when attributes change', () => {
    element.setAttribute('username', 'Bob');
    expect(element.textContent).toContain('Bob');

    element.setAttribute('username', 'Charlie');
    expect(element.textContent).toContain('Charlie');
    expect(element.textContent).not.toContain('Bob');
  });
});

```

```
it('should have correct CSS classes', () => {
  element.setAttribute('username', 'Alice');

  const badge = element.querySelector('.user-badge');
  const username = element.querySelector('.username');
  const role = element.querySelector('.role');

  expect(badge).not.toBeNull();
  expect(username).not.toBeNull();
  expect(role).not.toBeNull();
});
});
```

17.3 Mocking the PAN Bus

Testing message-driven components requires mocking the message bus.

17.3.1 Creating a Mock Bus

```
// tests/mocks/mock-bus.js

class MockBus {
  constructor() {
    this.subscriptions = new Map();
    this.published = [];
  }

  // Mock publish function
  publish(topic, data) {
    this.published.push({ topic, data, timestamp: Date.now() });

    // Trigger subscriptions
    const handlers = this.subscriptions.get(topic) || [];
    handlers.forEach(handler => {
      handler({ topic, data });
    });

    // Trigger wildcard subscriptions
    const wildcardHandlers = this.getWildcardHandlers(topic);
    wildcardHandlers.forEach(handler => {
      handler({ topic, data });
    });
  }

  // Mock subscribe function
  subscribe(pattern, handler) {
```

```

    if (!this.subscriptions.has(pattern)) {
      this.subscriptions.set(pattern, []);
    }

    this.subscriptions.get(pattern).push(handler);

    // Return unsubscribe function
    return () => {
      const handlers = this.subscriptions.get(pattern);
      const index = handlers.indexOf(handler);
      if (index > -1) {
        handlers.splice(index, 1);
      }
    };
  }

  // Get handlers for wildcard patterns
  getWildcardHandlers(topic) {
    const handlers = [];

    for (const [pattern, patternHandlers] of this.subscriptions) {
      if (this.matchesPattern(topic, pattern)) {
        handlers.push(...patternHandlers);
      }
    }

    return handlers;
  }

  // Simple wildcard matching
  matchesPattern(topic, pattern) {
    if (pattern === '*') return true;
    if (pattern === topic) return false; // Exact match handled separately

    const patternParts = pattern.split('.');
    const topicParts = topic.split('.');

    if (patternParts.length !== topicParts.length) {
      return false;
    }

    return patternParts.every((part, i) => {
      return part === '*' || part === topicParts[i];
    });
  }

  // Reset the bus

```

```
reset() {
  this.subscriptions.clear();
  this.published = [];
}

// Test helpers
getPublished(topic) {
  return this.published.filter(msg => msg.topic === topic);
}

getLastPublished(topic) {
  const messages = this.getPublished(topic);
  return messages[messages.length - 1];
}

wasPublished(topic, data) {
  return this.published.some(msg =>
    msg.topic === topic &&
    JSON.stringify(msg.data) === JSON.stringify(data)
  );
}
}

export { MockBus };
```

17.3.2 Using the Mock Bus

```
// tests/setup.js

import { beforeEach, afterEach } from 'vitest';
import { MockBus } from '../mocks/mock-bus.js';

let mockBus;

beforeEach(() => {
  mockBus = new MockBus();

  // Replace global publish and subscribe
  global.publish = mockBus.publish.bind(mockBus);
  global.subscribe = mockBus.subscribe.bind(mockBus);
});

afterEach(() => {
  mockBus.reset();
  document.body.innerHTML = '';
});
```

```
// Export for use in tests
export function getMockBus() {
  return mockBus;
}
```

17.3.3 Testing Message-Driven Components

```
// components/notification-display.mjs

import { subscribe } from '../pan.js';

class NotificationDisplay extends HTMLElement {
  constructor() {
    super();
    this.notifications = [];
  }

  connectedCallback() {
    this.unsubscribe = subscribe('notification.show', (msg) => {
      this.addNotification(msg.data);
    });

    this.render();
  }

  addNotification(notification) {
    this.notifications.push(notification);
    this.render();
  }

  render() {
    this.innerHTML = `
      <div class="notifications">
        ${this.notifications.map(n => `
          <div class="notification notification-${n.type}">
            ${n.message}
          </div>
        `).join('')}
      </div>
    `;
  }

  disconnectedCallback() {
    if (this.unsubscribe) {
      this.unsubscribe();
    }
  }
}
```

```
}  
  
customElements.define('notification-display', NotificationDisplay);  
export { NotificationDisplay };
```

Test it:

```
// tests/notification-display.test.js  
  
import { describe, it, expect, beforeEach } from 'vitest';  
import { NotificationDisplay } from '../components/notification-display.mjs';  
import { getMockBus } from './setup.js';  
  
describe('NotificationDisplay', () => {  
  let element;  
  let mockBus;  
  
  beforeEach(() => {  
    mockBus = getMockBus();  
    element = document.createElement('notification-display');  
    document.body.appendChild(element);  
  });  
  
  it('should start with no notifications', () => {  
    expect(element.notifications).toHaveLength(0);  
    expect(element.querySelector('.notification')).toBeNull();  
  });  
  
  it('should display notification when message is published', () => {  
    publish('notification.show', {  
      type: 'info',  
      message: 'Hello, World!'  
    });  
  
    expect(element.notifications).toHaveLength(1);  
    expect(element.textContent).toContain('Hello, World!');  
    expect(element.querySelector('.notification-info')).not.toBeNull();  
  });  
  
  it('should display multiple notifications', () => {  
    publish('notification.show', { type: 'info', message: 'First' });  
    publish('notification.show', { type: 'warning', message: 'Second' });  
    publish('notification.show', { type: 'error', message: 'Third' });  
  
    expect(element.notifications).toHaveLength(3);  
    expect(element.textContent).toContain('First');  
    expect(element.textContent).toContain('Second');  
    expect(element.textContent).toContain('Third');
```

```
});

it('should unsubscribe when disconnected', () => {
  element.remove();

  // Publish after removal
  publish('notification.show', { type: 'info', message: 'After removal' });

  // Should not have been added
  expect(element.notifications).toHaveLength(0);
});
});
```

17.4 Integration Testing Message Flows

Integration tests verify multiple components working together through message flows.

```
// tests/integration/shopping-cart.test.js

import { describe, it, expect, beforeEach } from 'vitest';
import { ProductCatalog } from '../../components/product-catalog.mjs';
import { ShoppingCart } from '../../components/shopping-cart.mjs';
import { CartBadge } from '../../components/cart-badge.mjs';
import { getMockBus } from '../setup.js';

describe('Shopping Cart Integration', () => {
  let catalog;
  let cart;
  let badge;
  let mockBus;

  beforeEach(() => {
    mockBus = getMockBus();

    // Create components
    catalog = document.createElement('product-catalog');
    cart = document.createElement('shopping-cart');
    badge = document.createElement('cart-badge');

    // Add to DOM
    document.body.appendChild(catalog);
    document.body.appendChild(cart);
    document.body.appendChild(badge);
  });

  it('should update cart and badge when product is added', () => {
    // Simulate adding product
```

```
publish('cart.item.added', {
  productId: 1,
  name: 'Widget',
  price: 10,
  quantity: 1
});

// Cart should contain the item
expect(cart.items).toHaveLength(1);
expect(cart.items[0].name).toBe('Widget');

// Badge should show count
expect(badge.itemCount).toBe(1);
expect(badge.textContent).toContain('1');
});

it('should publish cart.updated when item is added', () => {
  publish('cart.item.added', {
    productId: 1,
    name: 'Widget',
    price: 10,
    quantity: 1
  });

  // Verify cart.updated was published
  const updated = mockBus.getLastPublished('cart.updated');
  expect(updated).not.toBeUndefined();
  expect(updated.data.items).toHaveLength(1);
  expect(updated.data.total).toBe(10);
});

it('should handle multiple items', () => {
  publish('cart.item.added', {
    productId: 1,
    name: 'Widget',
    price: 10,
    quantity: 2
  });

  publish('cart.item.added', {
    productId: 2,
    name: 'Gadget',
    price: 20,
    quantity: 1
  });

  expect(cart.items).toHaveLength(2);
});
```

```

    expect(badge.itemCount).toBe(3); // 2 widgets + 1 gadget
  });

  it('should update quantities for duplicate items', () => {
    publish('cart.item.added', {
      productId: 1,
      name: 'Widget',
      price: 10,
      quantity: 1
    });

    publish('cart.item.added', {
      productId: 1,
      name: 'Widget',
      price: 10,
      quantity: 1
    });

    // Should have one item with quantity 2
    expect(cart.items).toHaveLength(1);
    expect(cart.items[0].quantity).toBe(2);
  });
});

```

17.5 Testing Async Operations

Many LARC operations are async. Test them properly.

17.5.1 Testing Promises

```

// components/data-loader.mjs

import { publish } from '../pan.js';

class DataLoader extends HTMLElement {
  async connectedCallback() {
    try {
      publish('data.loading', { loading: true });

      const response = await fetch('/api/data');
      const data = await response.json();

      publish('data.loaded', { data });
    } catch (error) {
      publish('data.error', { error: error.message });
    } finally {

```

```

    publish('data.loading', { loading: false });
  }
}
}

customElements.define('data-loader', DataLoader);
export { DataLoader };

```

Test it:

```

// tests/data-loader.test.js

import { describe, it, expect, beforeEach, vi } from 'vitest';
import { DataLoader } from '../components/data-loader.mjs';
import { getMockBus } from './setup.js';

describe('DataLoader', () => {
  let mockBus;

  beforeEach(() => {
    mockBus = getMockBus();

    // Mock fetch
    global.fetch = vi.fn();
  });

  it('should publish loading state', async () => {
    fetch.mockResolvedValueOnce({
      json: async () => ({ items: [] })
    });

    const element = document.createElement('data-loader');
    document.body.appendChild(element);

    // Wait for async operations
    await new Promise(resolve => setTimeout(resolve, 0));

    // Check loading messages
    const loadingMessages = mockBus.getPublished('data.loading');
    expect(loadingMessages).toHaveLength(2);
    expect(loadingMessages[0].data.loading).toBe(true);
    expect(loadingMessages[1].data.loading).toBe(false);
  });

  it('should publish data when loaded successfully', async () => {
    const mockData = { items: [1, 2, 3] };

    fetch.mockResolvedValueOnce({

```

```

    json: async () => mockData
  });

  const element = document.createElement('data-loader');
  document.body.appendChild(element);

  // Wait for async operations
  await new Promise(resolve => setTimeout(resolve, 0));

  const loaded = mockBus.getLastPublished('data.loaded');
  expect(loaded).not.toBeUndefined();
  expect(loaded.data.data).toEqual(mockData);
});

it('should publish error when fetch fails', async () => {
  fetch.mockRejectedValueOnce(new Error('Network error'));

  const element = document.createElement('data-loader');
  document.body.appendChild(element);

  // Wait for async operations
  await new Promise(resolve => setTimeout(resolve, 0));

  const error = mockBus.getLastPublished('data.error');
  expect(error).not.toBeUndefined();
  expect(error.data.error).toBe('Network error');
});
});

```

17.5.2 Testing with Async/Await

Use async/await in tests:

```

it('should load data', async () => {
  fetch.mockResolvedValueOnce({
    json: async () => ({ data: 'test' })
  });

  const element = document.createElement('data-loader');
  document.body.appendChild(element);

  // Wait for component to finish loading
  await vi.waitFor(() => {
    expect(mockBus.wasPublished('data.loaded', { data: { data: 'test' } })).toBe(true);
  });
});

```

17.5.3 Testing Timeouts and Intervals

```
// components/auto-saver.mjs

import { subscribe } from '../pan.js';

class AutoSaver extends HTMLElement {
  constructor() {
    super();
    this.saveInterval = 5000; // 5 seconds
    this.intervalId = null;
  }

  connectedCallback() {
    this.intervalId = setInterval(() => {
      publish('data.save', { timestamp: Date.now() });
    }, this.saveInterval);
  }

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

customElements.define('auto-saver', AutoSaver);
export { AutoSaver };
```

Test it:

```
// tests/auto-saver.test.js

import { describe, it, expect, beforeEach, vi } from 'vitest';
import { AutoSaver } from '../components/auto-saver.mjs';
import { getMockBus } from './setup.js';

describe('AutoSaver', () => {
  let mockBus;

  beforeEach(() => {
    mockBus = getMockBus();
    vi.useFakeTimers();
  });

  afterEach(() => {
    vi.useRealTimers();
  });

  it('should save at regular intervals', () => {
```

```

const element = document.createElement('auto-saver');
document.body.appendChild(element);

// Fast-forward 5 seconds
vi.advanceTimersByTime(5000);
expect(mockBus.getPublished('data.save')).toHaveLength(1);

// Fast-forward another 5 seconds
vi.advanceTimersByTime(5000);
expect(mockBus.getPublished('data.save')).toHaveLength(2);

// Fast-forward another 5 seconds
vi.advanceTimersByTime(5000);
expect(mockBus.getPublished('data.save')).toHaveLength(3);
});

it('should stop saving when disconnected', () => {
  const element = document.createElement('auto-saver');
  document.body.appendChild(element);

  vi.advanceTimersByTime(5000);
  expect(mockBus.getPublished('data.save')).toHaveLength(1);

  element.remove();

  // Should not save after removal
  vi.advanceTimersByTime(5000);
  expect(mockBus.getPublished('data.save')).toHaveLength(1);
});
});

```

17.6 End-to-End Testing

E2E tests verify the entire application in a real browser. Use Playwright or Cypress.

17.6.1 Setting Up Playwright

```

npm install -D @playwright/test
npx playwright install

```

Create a test:

```

// tests/e2e/shopping-cart.spec.js

import { test, expect } from '@playwright/test';

test.describe('Shopping Cart', () => {

```

```
test.beforeEach(async ({ page }) => {
  await page.goto('http://localhost:3000');
});

test('should add item to cart', async ({ page }) => {
  // Click add to cart button
  await page.click('button:has-text("Add to Cart")');

  // Verify cart badge updates
  const badge = page.locator('cart-badge');
  await expect(badge).toContainText('1');

  // Verify cart displays item
  const cart = page.locator('shopping-cart');
  await expect(cart).toContainText('Widget');
  await expect(cart).toContainText('$10');
});

test('should calculate total correctly', async ({ page }) => {
  // Add multiple items
  await page.click('button:has-text("Add to Cart")').first();
  await page.click('button:has-text("Add to Cart")').nth(1);

  // Verify total
  const cart = page.locator('shopping-cart');
  await expect(cart).toContainText('Total: $30');
});

test('should persist cart across page reloads', async ({ page }) => {
  // Add item to cart
  await page.click('button:has-text("Add to Cart")');

  // Reload page
  await page.reload();

  // Verify cart still has item
  const cart = page.locator('shopping-cart');
  await expect(cart).toContainText('Widget');
});
});
```

Run E2E tests:

```
npx playwright test
```

17.6.2 Testing Theme Switching

```
// tests/e2e/theme.spec.js

import { test, expect } from '@playwright/test';

test.describe('Theme Switching', () => {
  test('should toggle between light and dark mode', async ({ page }) => {
    await page.goto('http://localhost:3000');

    // Check initial theme
    const html = page.locator('html');
    await expect(html).toHaveAttribute('data-theme', 'light');

    // Click dark mode button
    await page.click('button:has-text("Dark")');

    // Verify theme changed
    await expect(html).toHaveAttribute('data-theme', 'dark');

    // Verify styles applied
    const body = page.locator('body');
    const bgColor = await body.evaluate(el =>
      getComputedStyle(el).backgroundColor
    );
    expect(bgColor).toBe('rgb(17, 24, 39)'); // Dark background
  });

  test('should persist theme preference', async ({ page }) => {
    await page.goto('http://localhost:3000');

    // Switch to dark mode
    await page.click('button:has-text("Dark")');

    // Reload page
    await page.reload();

    // Verify theme persisted
    const html = page.locator('html');
    await expect(html).toHaveAttribute('data-theme', 'dark');
  });
});
```

17.7 Test Utilities and Helpers

Build reusable utilities to make testing easier.

17.7.1 Component Test Harness

```
// tests/utils/component-harness.js

class ComponentHarness {
  constructor(tagName, attributes = {}) {
    this.element = document.createElement(tagName);

    // Set attributes
    for (const [key, value] of Object.entries(attributes)) {
      this.element.setAttribute(key, value);
    }

    document.body.appendChild(this.element);
  }

  // Query within component
  query(selector) {
    return this.element.querySelector(selector);
  }

  queryAll(selector) {
    return this.element.querySelectorAll(selector);
  }

  // Get text content
  text() {
    return this.element.textContent.trim();
  }

  // Click element
  click(selector) {
    const el = selector ? this.query(selector) : this.element;
    el.click();
    return this;
  }

  // Type into input
  type(selector, value) {
    const input = this.query(selector);
    input.value = value;
    input.dispatchEvent(new Event('input', { bubbles: true }));
    return this;
  }

  // Wait for condition
  async waitFor(condition, timeout = 1000) {
```

```

    const start = Date.now();

    while (Date.now() - start < timeout) {
      if (condition(this.element)) {
        return;
      }
      await new Promise(resolve => setTimeout(resolve, 50));
    }

    throw new Error('Timeout waiting for condition');
  }

  // Clean up
  destroy() {
    this.element.remove();
  }
}

export { ComponentHarness };

```

Use it:

```

// tests/user-profile.test.js

import { describe, it, expect } from 'vitest';
import { ComponentHarness } from '../utils/component-harness.js';
import { UserProfile } from '../components/user-profile.mjs';

describe('UserProfile', () => {
  it('should display user information', async () => {
    const harness = new ComponentHarness('user-profile', {
      'user-id': '123'
    });

    // Publish user data
    publish('user.data', {
      userId: '123',
      name: 'Alice',
      email: 'alice@example.com'
    });

    // Wait for render
    await harness.waitFor(el => el.textContent.includes('Alice'));

    expect(harness.text()).toContain('Alice');
    expect(harness.text()).toContain('alice@example.com');

    harness.destroy();
  });
});

```

```
});
});
```

17.7.2 Message Bus Test Helper

```
// tests/utils/message-helper.js

import { getMockBus } from '../setup.js';

class MessageHelper {
  constructor() {
    this.bus = getMockBus();
  }

  // Publish and wait for response
  async publishAndWait(publishTopic, publishData, waitTopic, timeout = 1000) {
    return new Promise((resolve, reject) => {
      const timeoutId = setTimeout(() => {
        unsubscribe();
        reject(new Error(`Timeout waiting for ${waitTopic}`));
      }, timeout);

      const unsubscribe = subscribe(waitTopic, (msg) => {
        clearTimeout(timeoutId);
        unsubscribe();
        resolve(msg.data);
      });

      publish(publishTopic, publishData);
    });
  }

  // Wait for specific message
  async waitForMessage(topic, predicate = null, timeout = 1000) {
    return new Promise((resolve, reject) => {
      const timeoutId = setTimeout(() => {
        unsubscribe();
        reject(new Error(`Timeout waiting for ${topic}`));
      }, timeout);

      const unsubscribe = subscribe(topic, (msg) => {
        if (!predicate || predicate(msg.data)) {
          clearTimeout(timeoutId);
          unsubscribe();
          resolve(msg.data);
        }
      });
    });
  }
}
```

```

    });
  }

  // Assert message was published
  assertPublished(topic, data = null) {
    const messages = this.bus.getPublished(topic);

    if (messages.length === 0) {
      throw new Error(`Expected message on topic "${topic}" but none were published`);
    }

    if (data !== null) {
      const match = messages.some(msg =>
        JSON.stringify(msg.data) === JSON.stringify(data)
      );

      if (!match) {
        throw new Error(
          `Expected message on topic "${topic}" with data ${JSON.stringify(data)} ` +
          `but received: ${JSON.stringify(messages.map(m => m.data))}`
        );
      }
    }
  }

  // Assert message was NOT published
  assertNotPublished(topic) {
    const messages = this.bus.getPublished(topic);

    if (messages.length > 0) {
      throw new Error(
        `Expected no messages on topic "${topic}" but ${messages.length} were published`
      );
    }
  }
}

export { MessageHelper };

```

Use it:

```

import { MessageHelper } from './utils/message-helper.js';

it('should respond to data request', async () => {
  const helper = new MessageHelper();

  const element = document.createElement('data-provider');
  document.body.appendChild(element);

```

```
// Publish request and wait for response
const response = await helper.publishAndWait(
  'data.request',
  { id: 123 },
  'data.response'
);

expect(response.id).toBe(123);
expect(response.data).toBeDefined();
});
```

17.8 Test Coverage

Measure test coverage to identify untested code:

```
npm install -D @vitest/coverage-v8
```

Run with coverage:

```
npx vitest --coverage
```

Aim for:

- **80%+ overall coverage**
- **100% coverage for critical paths** (auth, payments, data loss scenarios)
- **Lower coverage for UI glue code** (it's okay)

17.9 Wrapping Up

Testing LARC applications is straightforward once you understand the patterns:

1. **Unit tests:** Test components in isolation, mock the message bus
2. **Integration tests:** Test message flows between components
3. **E2E tests:** Test the full application in a real browser
4. **Mock the bus:** Use MockBus for predictable, fast tests
5. **Test async operations:** Use async/await and fake timers
6. **Build utilities:** Create harnesses and helpers to simplify testing

The key insight: message-driven architecture makes testing easier, not harder. Components are decoupled, dependencies are explicit (subscriptions), and side effects are observable (publications).

Write tests. Run them often. Trust them completely. And when a test fails, thank it for catching a bug before your users did.

You've now completed a comprehensive tour of LARC development: theming and styling for beautiful UIs, performance optimization for fast applications, and testing strategies for reliable software. You have all the tools to build production-ready LARC applications.

Now go forth and build something amazing. And remember: untested code is legacy code the moment you write it.

Chapter 18

Error Handling and Debugging

“The only thing more satisfying than writing code that works is understanding why code that doesn’t work... doesn’t work.” — Ancient Developer Proverb

Error handling in LARC is like being a detective in a noir film—except instead of following a femme fatale through shadowy streets, you’re following messages through a bus topology. The good news? LARC’s message-passing architecture makes debugging surprisingly tractable. The bad news? You still have to actually do the debugging.

18.1 Error Boundaries: Containing the Chaos

In traditional frameworks, errors cascade like dominoes in a Rube Goldberg machine. One component explodes, and suddenly your entire application is showing a white screen of death. LARC’s component isolation means errors are naturally contained—but you still need to handle them gracefully.

18.1.1 Understanding Error Propagation

When a LARC component throws an error during message handling, the error is caught at the component boundary. The component’s state remains unchanged, and the error is logged. Other components continue merrily on their way, blissfully unaware of their sibling’s existential crisis.

```
// A component with a bug
class BuggyCounter extends Component {
  init() {
    this.state = { count: 0 };
  }

  receive(type, data) {
    if (type === 'increment') {
      // Oops, typo in property name
      this.state.count++; // This will throw
    }
  }

  render() {
```

```

    return html`<div>Count: ${this.state.count}</div>`;
  }
}

```

When this component receives an ‘increment’ message, it’ll throw a `TypeError`. But here’s the beautiful part: the error doesn’t take down your app. The counter just stays at zero, looking sheepish.

18.1.2 Implementing Error Handlers

You can catch and handle errors within your components explicitly:

```

class ResilientCounter extends Component {
  init() {
    this.state = { count: 0, error: null };
  }

  receive(type, data) {
    try {
      if (type === 'increment') {
        // Intentionally buggy operation
        if (data.shouldFail) {
          throw new Error('Increment failed: cosmic rays detected');
        }
        this.state.count++;
        this.state.error = null;
      } else if (type === 'reset-error') {
        this.state.error = null;
      }
    } catch (error) {
      this.state.error = error.message;
      // Emit error to the bus for centralized handling
      this.emit('app-error', {
        component: this.constructor.name,
        error: error.message,
        timestamp: Date.now()
      });
    }
  }

  render() {
    if (this.state.error) {
      return html`
        <div class="error-state">
          <p>[!] Error: ${this.state.error}</p>
          <button onclick=${() => this.receive('reset-error')}>
            Try Again
          </button>
        </div>
      `;
    }
  }
}

```

```

        </div>
      `;
    }

    return html`
      <div>
        <p>Count: ${this.state.count}</p>
        <button onclick=${() => this.receive('increment')}>+1</button>
      </div>
    `;
  }
}

```

This component catches errors, stores them in state, and emits an ‘app-error’ message to the bus. This pattern gives you three levels of defense:

1. **Local recovery:** The component can display an error state
2. **Global awareness:** Other components can react to the error
3. **Continued operation:** The app keeps running

18.1.3 Global Error Handler Component

Create a dedicated error handler that listens to all error messages:

```

class ErrorMonitor extends Component {
  init() {
    this.state = {
      errors: [],
      maxErrors: 50 // Keep last 50 errors
    };
    this.on('app-error', this.logError);
    this.on('*', this.catchUnhandledErrors);
  }

  logError(data) {
    const errorEntry = {
      ...data,
      id: crypto.randomUUID()
    };

    this.state.errors.unshift(errorEntry);

    // Trim to max size
    if (this.state.errors.length > this.state.maxErrors) {
      this.state.errors.length = this.state.maxErrors;
    }

    // Send to external error tracking service
    this.reportToErrorService(errorEntry);
  }
}

```

```

}

catchUnhandledErrors(type, data) {
  // Wrap all message handlers to catch uncaught errors
  // This is more advanced - see the DevTools section
}

reportToErrorService(error) {
  // Integration with Sentry, LogRocket, etc.
  if (window.errorTracker) {
    window.errorTracker.captureMessage(error);
  }
}

render() {
  if (this.state.errors.length === 0) {
    return html`<div class="error-monitor">No errors [*]</div>`;
  }

  return html`
    <div class="error-monitor">
      <h3>Error Log (${this.state.errors.length})</h3>
      <ul>
        ${this.state.errors.map(err => html`
          <li key=${err.id}>
            <strong>${err.component}</strong>: ${err.error}
            <span class="timestamp">
              ${new Date(err.timestamp).toLocaleTimeString()}
            </span>
          </li>
        `)}
      </ul>
    </div>
  `;
}
}

```

18.2 Message Tracing: Following the Breadcrumbs

The hardest bugs to debug are the ones where you *know* you sent a message, but nothing happened. Did it get lost in the mail? Did the recipient get it and ignore you? Is this a metaphor for dating?

18.2.1 Built-in Message Tracing

LARC's bus system can be configured to trace all messages:

```
import { createBus } from 'larc';

const bus = createBus({
  debug: true, // Enable debug mode
  traceMessages: true // Log all messages
});

// Now every message will be logged
bus.emit('user-login', { username: 'detective' });
// Console: [LARC] user-login -> { username: 'detective' }
```

But debug mode in production is like wearing a tuxedo to a demolition derby—technically impressive, but not practical. Instead, implement selective tracing:

```
class MessageTracer extends Component {
  init() {
    this.state = {
      trace: false,
      messageLog: [],
      tracedTypes: new Set(['user-action', 'api-error', 'navigation'])
    };

    // Listen to ALL messages
    this.on('*', this.traceMessage);

    // Control tracing
    this.on('enable-trace', () => this.state.trace = true);
    this.on('disable-trace', () => this.state.trace = false);
    this.on('clear-trace', () => this.state.messageLog = []);
  }

  traceMessage(type, data) {
    if (!this.state.trace && !this.state.tracedTypes.has(type)) {
      return; // Skip if tracing is off and not a traced type
    }

    const entry = {
      timestamp: performance.now(),
      type,
      data: JSON.parse(JSON.stringify(data)), // Deep clone
      stack: new Error().stack // Capture call stack
    };

    this.state.messageLog.push(entry);

    // Log to console with styling
    console.log(
      `%c[LARC] ${type}`,
```

```

    'color: #00f; font-weight: bold',
    data
  );
}

render() {
  return html`
    <div class="message-tracer">
      <button onclick=${() => this.receive('enable-trace')}>
        Start Tracing
      </button>
      <button onclick=${() => this.receive('disable-trace')}>
        Stop Tracing
      </button>
      <button onclick=${() => this.receive('clear-trace')}>
        Clear Log
      </button>
      <div class="trace-log">
        ${this.state.messageLog.map((entry, i) => html`
          <div key=${i} class="trace-entry">
            <span class="time">${entry.timestamp.toFixed(2)}ms</span>
            <span class="type">${entry.type}</span>
            <pre>${JSON.stringify(entry.data, null, 2)}</pre>
          </div>
        `)}
      </div>
    </div>
  `;
}
}

```

18.2.2 Message Flow Visualization

Sometimes you need to see the big picture. Build a message flow diagram:

```

class MessageFlowVisualizer extends Component {
  init() {
    this.state = {
      nodes: new Map(), // component name -> position
      edges: [], // { from, to, type, timestamp }
      recording: false
    };

    this.on('*', this.recordMessage);
  }

  recordMessage(type, data) {
    if (!this.state.recording) return;
  }
}

```

```

// Track which component sent this
const sourceComponent = this.identifySource();

// Track which components might handle this
const targetComponents = this.identifyTargets(type);

targetComponents.forEach(target => {
  this.state.edges.push({
    from: sourceComponent,
    to: target,
    type,
    timestamp: Date.now()
  });
});

// Auto-prune old edges after 10 seconds
const cutoff = Date.now() - 10000;
this.state.edges = this.state.edges.filter(e => e.timestamp > cutoff);
}

identifySource() {
  // Analyze call stack to identify sending component
  const stack = new Error().stack;
  // Parse stack frames to find component name
  // (Implementation details depend on your naming conventions)
  return 'UnknownSource';
}

identifyTargets(type) {
  // This would require introspection of registered handlers
  // For now, return placeholder
  return ['ComponentA', 'ComponentB'];
}

render() {
  // Render as a force-directed graph using D3.js or similar
  // For brevity, showing simplified version
  return html`
    <div class="flow-visualizer">
      <button onclick=${() => this.state.recording = !this.state.recording}>
        ${this.state.recording ? 'Stop' : 'Start'} Recording
      </button>
      <svg width="800" height="600">
        ${this.state.edges.map((edge, i) => html`
          <line key=${i}
            x1=${this.getNodeX(edge.from)}
            y1=${this.getNodeY(edge.from)}

```

```

        x2=${this.getNodeX(edge.to)}
        y2=${this.getNodeY(edge.to)}
        stroke="#888"
        stroke-width="2" />
    `)}
</svg>
</div>
`;
}

getNodeX(nodeName) {
    // Calculate position for node
    return 100; // Placeholder
}

getNodeY(nodeName) {
    return 100; // Placeholder
}
}

```

18.3 DevTools Integration: Professional Debugging

Browser DevTools are your best friend, but they're even better when your framework plays nice with them.

18.3.1 Custom Console Formatters

Make LARC messages beautiful in the console:

```

// Add custom formatter for LARC messages
if (window.devtoolsFormatters) {
    window.devtoolsFormatters.push({
        header(obj) {
            if (!obj || !obj.__larcMessage) return null;

            return ['div', { style: 'color: #0066cc; font-weight: bold' },
                ['span', {}, `[msg] LARC Message: ${obj.type}`]
            ];
        },
        hasBody(obj) {
            return obj && obj.__larcMessage;
        },
        body(obj) {
            return ['div', {},
                ['div', {}, `Type: ${obj.type}`],
                ['div', {}, `Data: `, ['object', { object: obj.data }]],
                ['div', {}, `Timestamp: ${new Date(obj.timestamp).toISOString()}`]
            ];
        }
    });
}

```

```

    ];
  }
});
}

// Wrap bus.emit to add metadata
const originalEmit = bus.emit;
bus.emit = function(type, data) {
  const message = {
    __larcMessage: true,
    type,
    data,
    timestamp: Date.now()
  };
  console.log(message);
  return originalEmit.call(this, type, data);
};

```

18.3.2 Source Maps and Stack Traces

When errors occur, you want meaningful stack traces:

```

class ErrorReporter extends Component {
  init() {
    // Catch global errors
    window.addEventListener('error', (event) => {
      this.handleError({
        message: event.message,
        source: event.filename,
        line: event.lineno,
        column: event.colno,
        stack: event.error?.stack
      });
    });

    // Catch promise rejections
    window.addEventListener('unhandledrejection', (event) => {
      this.handleError({
        message: event.reason?.message || 'Unhandled Promise Rejection',
        stack: event.reason?.stack
      });
    });
  }

  handleError(error) {
    // Parse stack trace to extract meaningful info
    const frames = this.parseStackTrace(error.stack);
  }
}

```

```

    this.emit('fatal-error', {
      ...error,
      frames,
      userAgent: navigator.userAgent,
      url: window.location.href,
      timestamp: new Date().toISOString()
    });
  }

  parseStackTrace(stack) {
    if (!stack) return [];

    return stack.split('\n')
      .slice(1) // Skip first line (error message)
      .map(line => {
        // Parse format: "at functionName (file:line:col)"
        const match = line.match(/at\s+(.+) \s+\((.+):(\d+):(\d+)\)/);
        if (match) {
          return {
            function: match[1],
            file: match[2],
            line: parseInt(match[3]),
            column: parseInt(match[4])
          };
        }
        return null;
      })
      .filter(Boolean);
  }

  render() {
    return null; // Invisible component
  }
}

```

18.3.3 Performance Profiling

Use the Performance API to identify bottlenecks:

```

class PerformanceMonitor extends Component {
  init() {
    this.state = {
      measurements: []
    };

    this.on('*', this.measureMessageHandling);
  }
}

```

```

measureMessageHandling(type, data) {
  const markName = `message-${type}-${Date.now()}`;
  performance.mark(markName);

  // Measure next tick (after handlers complete)
  setTimeout(() => {
    performance.measure(type, markName);

    const entries = performance.getEntriesByType('measure');
    const latest = entries[entries.length - 1];

    if (latest.duration > 16) { // Slower than 60fps
      console.warn(`Slow message handler: ${type} took ${latest.duration}ms`);
      this.emit('performance-warning', {
        type,
        duration: latest.duration
      });
    }

    performance.clearMarks(markName);
    performance.clearMeasures(type);
  }, 0);
}

render() {
  return null;
}
}

```

18.4 Logging Strategies: Write Once, Debug Forever

Good logging is like leaving a trail of breadcrumbs, except the breadcrumbs are actually useful and don't get eaten by birds.

18.4.1 Structured Logging

Don't just log strings. Log objects with context:

```

class Logger extends Component {
  init() {
    this.state = {
      level: 'info', // debug, info, warn, error
      transports: [this.consoleTransport, this.remoteTransport]
    };

    this.on('log', this.handleLog);
  }
}

```

```
handleLog({ level, message, context }) {
  if (!this.shouldLog(level)) return;

  const entry = {
    timestamp: new Date().toISOString(),
    level,
    message,
    context,
    sessionId: this.getSessionId(),
    userId: this.getUserId()
  };

  this.state.transports.forEach(transport => {
    transport(entry);
  });
}

shouldLog(level) {
  const levels = ['debug', 'info', 'warn', 'error'];
  const currentIndex = levels.indexOf(this.state.level);
  const requestedIndex = levels.indexOf(level);
  return requestedIndex >= currentIndex;
}

consoleTransport(entry) {
  const styles = {
    debug: 'color: gray',
    info: 'color: blue',
    warn: 'color: orange',
    error: 'color: red; font-weight: bold'
  };

  console.log(
    `%c[${entry.level.toUpperCase()}] ${entry.message}`,
    styles[entry.level],
    entry.context
  );
}

remoteTransport(entry) {
  // Send to logging service
  if (entry.level === 'error' || entry.level === 'warn') {
    navigator.sendBeacon('/api/logs', JSON.stringify(entry));
  }
}

getSessionId() {
```

```

    return sessionStorage.getItem('sessionId') || 'unknown';
  }

  getUserId() {
    return localStorage.getItem('userId') || 'anonymous';
  }

  render() {
    return null;
  }
}

// Usage in other components
class UserProfile extends Component {
  async loadUserData(userId) {
    this.emit('log', {
      level: 'info',
      message: 'Loading user profile',
      context: { userId }
    });

    try {
      const data = await fetch(`/api/users/${userId}`).then(r => r.json());
      this.state.user = data;

      this.emit('log', {
        level: 'info',
        message: 'User profile loaded',
        context: { userId, username: data.username }
      });
    } catch (error) {
      this.emit('log', {
        level: 'error',
        message: 'Failed to load user profile',
        context: { userId, error: error.message }
      });
    }
  }
}

```

18.4.2 Log Aggregation and Search

Build a searchable log viewer:

```

class LogViewer extends Component {
  init() {
    this.state = {
      logs: [],

```

```

    filter: '',
    levelFilter: 'all'
  };

  this.on('log', (data) => {
    this.state.logs.push(data);
    // Keep only last 1000 logs
    if (this.state.logs.length > 1000) {
      this.state.logs.shift();
    }
  });
}

get filteredLogs() {
  return this.state.logs.filter(log => {
    const matchesLevel = this.state.levelFilter === 'all' ||
      log.level === this.state.levelFilter;
    const matchesText = !this.state.filter ||
      JSON.stringify(log).toLowerCase()
        .includes(this.state.filter.toLowerCase());
    return matchesLevel && matchesText;
  });
}

exportLogs() {
  const blob = new Blob(
    [JSON.stringify(this.state.logs, null, 2)],
    { type: 'application/json' }
  );
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a');
  a.href = url;
  a.download = `logs-${Date.now()}.json`;
  a.click();
}

render() {
  return html`
    <div class="log-viewer">
      <div class="controls">
        <input
          type="text"
          placeholder="Filter logs..."
          value=${this.state.filter}
          oninput=${(e) => this.state.filter = e.target.value}
        />
        <select

```

```

        onchange=${(e) => this.state.levelFilter = e.target.value}
      >
      <option value="all">All Levels</option>
      <option value="debug">Debug</option>
      <option value="info">Info</option>
      <option value="warn">Warn</option>
      <option value="error">Error</option>
    </select>
    <button onclick=${() => this.exportLogs()}>Export</button>
    <button onclick=${() => this.state.logs = []}>Clear</button>
  </div>
  <div class="log-entries">
    ${this.filteredLogs.map((log, i) => html`
      <div key=${i} class="log-entry level-${log.level}">
        <span class="timestamp">${log.timestamp}</span>
        <span class="level">${log.level}</span>
        <span class="message">${log.message}</span>
        <details>
          <summary>Context</summary>
          <pre>${JSON.stringify(log.context, null, 2)}</pre>
        </details>
      </div>
    `)}
  </div>
</div>
`
;
}
}

```

18.5 Common Pitfalls and Solutions

Let's address the bugs that keep you up at night (or at least keep you Googling until 2 AM).

18.5.1 Pitfall #1: Message Type Typos

```

// Component A emits
this.emit('user-logged-in', { userId: 123 }); // Typo!

// Component B listens
this.on('user-logged-in', (data) => { /* never called */ });

```

Solution: Use constants for message types:

```

// messages.js
export const Messages = {
  USER_LOGGED_IN: 'user-logged-in',
  USER_LOGGED_OUT: 'user-logged-out',

```

```

    CART_UPDATED: 'cart-updated'
  };

  // Usage
  import { Messages } from './messages.js';

  this.emit(Messages.USER_LOGGED_IN, { userId: 123 });
  this.on(Messages.USER_LOGGED_IN, (data) => { /* works! */ });

```

18.5.2 Pitfall #2: Infinite Message Loops

```

class BadCounter extends Component {
  init() {
    this.state = { count: 0 };
    this.on('increment', () => {
      this.state.count++;
      this.emit('count-changed', { count: this.state.count });
    });
    this.on('count-changed', () => {
      this.emit('increment'); // INFINITE LOOP!
    });
  }
}

```

Solution: Add loop detection:

```

class LoopDetector extends Component {
  init() {
    this.messageStack = [];
    this.on('*', this.detectLoop);
  }

  detectLoop(type) {
    this.messageStack.push(type);

    // Check for cycles
    const lastFive = this.messageStack.slice(-5);
    if (this.hasCycle(lastFive)) {
      console.error('Message loop detected:', lastFive);
      this.emit('message-loop-detected', { sequence: lastFive });
    }

    // Clean up old messages
    setTimeout(() => this.messageStack.shift(), 1000);
  }

  hasCycle(sequence) {
    // Simple cycle detection: same message repeated 3+ times

```

```

    const counts = {};
    sequence.forEach(type => counts[type] = (counts[type] || 0) + 1);
    return Object.values(counts).some(count => count >= 3);
  }

  render() {
    return null;
  }
}

```

18.5.3 Pitfall #3: Stale Closures in Handlers

```

class StaleCounter extends Component {
  init() {
    this.state = { count: 0 };

    // This handler captures the initial value of count
    this.on('log-count', () => {
      console.log(this.state.count); // Always logs 0!
    });

    this.on('increment', () => {
      this.state.count++;
    });
  }
}

```

Solution: Always access `this.state` directly, never capture it:

```

class FreshCounter extends Component {
  init() {
    this.state = { count: 0 };

    // Access this.state.count at call time
    this.on('log-count', () => {
      console.log(this.state.count); // Always current!
    });
  }
}

```

18.5.4 Pitfall #4: Async Race Conditions

```

class RacyLoader extends Component {
  async loadData(id) {
    const data = await fetch(`/api/items/${id}`).then(r => r.json());
    this.state.currentItem = data; // May be stale if user clicked again!
  }
}

```

```
}

```

Solution: Track request IDs:

```
class SafeLoader extends Component {
  init() {
    this.state = {
      currentItem: null,
      loading: false
    };
    this.currentRequestId = 0;
  }

  async loadData(id) {
    const requestId = ++this.currentRequestId;
    this.state.loading = true;

    try {
      const data = await fetch(`/api/items/${id}`).then(r => r.json());

      // Only update if this is still the latest request
      if (requestId === this.currentRequestId) {
        this.state.currentItem = data;
        this.state.loading = false;
      }
    } catch (error) {
      if (requestId === this.currentRequestId) {
        this.state.loading = false;
        this.emit('load-error', { error: error.message });
      }
    }
  }
}
```

18.6 Debugging Checklist

When something goes wrong, work through this checklist:

1. **Is the message being sent?**
 - Add a `console.log` right before `emit()`
 - Check the `MessageTracer` component
2. **Is the message type spelled correctly?**
 - Use message constants
 - Enable strict type checking
3. **Is the handler registered?**
 - Check that `this.on()` is called in `init()`
 - Verify the component is instantiated
4. **Is the handler being called?**

- Add `console.log` at the start of the handler
 - Use the browser debugger with breakpoints
5. **Is the state updating?**
 - Log state before and after updates
 - Check for typos in property names
 6. **Is the render being triggered?**
 - LARC should re-render after state changes
 - Check for errors in render method
 7. **Are there async issues?**
 - Use request IDs for async operations
 - Check Promise rejection handling
 8. **Is there a message loop?**
 - Use the `LoopDetector` component
 - Review message flow diagram

18.7 Conclusion

Debugging LARC applications is like detective work with better tooling. The message-passing architecture gives you clear boundaries and audit trails. Components fail independently. Errors are contained. And with the right monitoring in place, you'll know about problems before your users do.

Remember: the best debugging session is the one you don't have to do because you wrote good error handling in the first place. But when things do go wrong (and they will), you're now armed with the tools to track down bugs faster than a caffeinated squirrel.

In the next chapter, we'll explore advanced patterns that will make your LARC applications more powerful—and hopefully won't introduce too many new bugs to debug.

Chapter 19

Advanced Patterns

“Any sufficiently advanced technology is indistinguishable from magic. Any sufficiently advanced LARC pattern is indistinguishable from over-engineering.” — Clarke’s Third Law, Revised

You’ve mastered the basics of LARC. Your components communicate gracefully. Your state management is pristine. Your error handling would make a DevOps engineer weep tears of joy. But now you’re ready for the advanced stuff—the patterns that separate the “just building apps” developers from the “architect a scalable micro-frontend ecosystem” developers.

Fair warning: some of these patterns are powerful. Some are clever. Some might be too clever. Use your judgment, and remember that the best code is the code your teammates can understand at 9 AM on a Monday.

19.1 Message Forwarding and Bridging

Sometimes you need messages from one bus to appear on another. Maybe you’re integrating a third-party widget. Maybe you’re building a multi-window application. Maybe you just like making things complicated (no judgment).

19.1.1 Basic Message Forwarding

Forward messages from one bus to another:

```
class MessageBridge extends Component {
  constructor(sourceBus, targetBus, messageTypes) {
    super();
    this.sourceBus = sourceBus;
    this.targetBus = targetBus;
    this.messageTypes = messageTypes || ['*']; // Forward all by default
  }

  init() {
    this.messageTypes.forEach(type => {
      this.sourceBus.on(type, (data) => {
```

```

        this.targetBus.emit(type, data);
    });
});
}

render() {
    return null; // Bridges don't render
}
}

// Usage
const mainBus = createBus();
const widgetBus = createBus();

// Forward user actions from widget to main app
const bridge = new MessageBridge(
    widgetBus,
    mainBus,
    ['user-click', 'user-input']
);

```

19.1.2 Bidirectional Bridging

When you need messages flowing both ways:

```

class BidirectionalBridge extends Component {
    constructor(busA, busB, config = {}) {
        super();
        this.busA = busA;
        this.busB = busB;
        this.config = {
            aToB: config.aToB || ['*'], // Types to forward A -> B
            bToA: config.bToA || ['*'], // Types to forward B -> A
            transform: config.transform || ((data) => data), // Transform data
            filter: config.filter || (() => true) // Filter messages
        };
    }

    init() {
        // Forward A -> B
        this.config.aToB.forEach(type => {
            this.busA.on(type, (data) => {
                if (this.config.filter(type, data, 'aToB')) {
                    const transformed = this.config.transform(data, 'aToB');
                    this.busB.emit(type, transformed);
                }
            });
        });
    }
};

```

```

    // Forward B -> A
    this.config.bToA.forEach(type => {
      this.busB.on(type, (data) => {
        if (this.config.filter(type, data, 'bToA')) {
          const transformed = this.config.transform(data, 'bToA');
          this.busA.emit(type, transformed);
        }
      });
    });
  });
}

render() {
  return null;
}
}

// Usage with transformation
const bridge = new BidirectionalBridge(mainBus, widgetBus, {
  aToB: ['theme-changed', 'user-logged-in'],
  bToA: ['widget-action'],
  transform: (data, direction) => {
    // Add metadata for tracking
    return {
      ...data,
      bridged: true,
      direction,
      timestamp: Date.now()
    };
  },
  filter: (type, data, direction) => {
    // Don't forward internal messages
    return !type.startsWith('internal-');
  }
});

```

19.1.3 Message Translation

When buses speak different dialects:

```

class MessageTranslator extends Component {
  constructor(sourceBus, targetBus, translations) {
    super();
    this.sourceBus = sourceBus;
    this.targetBus = targetBus;
    this.translations = translations;
  }
}

```

```

init() {
  Object.entries(this.translations).forEach(([sourceType, config]) => {
    this.sourceBus.on(sourceType, (data) => {
      const targetType = config.type || sourceType;
      const targetData = config.transform
        ? config.transform(data)
        : data;

      this.targetBus.emit(targetType, targetData);
    });
  });
}

render() {
  return null;
}
}

// Usage: Translate between LARC app and legacy jQuery plugin
const translator = new MessageTranslator(larcBus, jqueryBus, {
  'user-logged-in': {
    type: 'userLogin', // Different naming convention
    transform: (data) => ({
      userId: data.id, // Different property names
      userName: data.username,
      timestamp: new Date().toISOString()
    })
  },
  'cart-updated': {
    type: 'cartChange',
    transform: (data) => ({
      items: data.cartItems.map(item => ({
        id: item.productId,
        qty: item.quantity,
        price: item.unitPrice
      }))
    })
  }
});

```

19.2 Multi-Bus Architectures

One bus is good. Multiple buses? That's when things get interesting (and complicated).

19.2.1 Domain-Segregated Buses

Separate concerns by domain:

```

class MultiDomainApp {
  constructor() {
    // Separate buses for different domains
    this.buses = {
      auth: createBus({ namespace: 'auth' }),
      cart: createBus({ namespace: 'cart' }),
      ui: createBus({ namespace: 'ui' }),
      analytics: createBus({ namespace: 'analytics' })
    };

    // Create cross-domain bridges
    this.setupBridges();
  }

  setupBridges() {
    // Auth events trigger analytics
    new MessageBridge(
      this.buses.auth,
      this.buses.analytics,
      ['user-logged-in', 'user-logged-out']
    );

    // Cart events trigger UI updates
    new MessageBridge(
      this.buses.cart,
      this.buses.ui,
      ['cart-updated']
    );

    // Auth changes affect cart
    this.buses.auth.on('user-logged-out', () => {
      this.buses.cart.emit('clear-cart');
    });
  }

  getComponentProps(domain) {
    return {
      bus: this.buses[domain],
      globalBus: this.buses.ui // Some components need global access
    };
  }
}

// Usage
const app = new MultiDomainApp();

class LoginForm extends Component {

```

```

constructor(props) {
  super(props);
  this.authBus = props.bus; // Domain-specific bus
}

async handleLogin(username, password) {
  // Emit on auth bus
  this.authBus.emit('login-attempt', { username });

  const result = await this.authenticate(username, password);

  if (result.success) {
    this.authBus.emit('user-logged-in', {
      userId: result.userId,
      username: username
    });
  }
}
}

const loginForm = new LoginForm(app.getComponentProps('auth'));

```

19.2.2 Hierarchical Bus Structure

Create parent-child bus relationships:

```

class HierarchicalBus {
  constructor(parent = null) {
    this.parent = parent;
    this.children = new Set();
    this.handlers = new Map();

    if (parent) {
      parent.children.add(this);
    }
  }

  emit(type, data, options = {}) {
    const { bubble = false, propagate = false } = options;

    // Handle locally
    this._emitLocal(type, data);

    // Bubble up to parent
    if (bubble && this.parent) {
      this.parent.emit(type, data, { bubble: true });
    }
  }
}

```

```

    // Propagate down to children
    if (propagate) {
      this.children.forEach(child => {
        child.emit(type, data, { propagate: true });
      });
    }
  }

  _emitLocal(type, data) {
    const handlers = this.handlers.get(type) || [];
    handlers.forEach(handler => handler(data));

    const wildcardHandlers = this.handlers.get('*') || [];
    wildcardHandlers.forEach(handler => handler(type, data));
  }

  on(type, handler) {
    if (!this.handlers.has(type)) {
      this.handlers.set(type, []);
    }
    this.handlers.get(type).push(handler);
  }

  destroy() {
    if (this.parent) {
      this.parent.children.delete(this);
    }
    this.children.clear();
    this.handlers.clear();
  }
}

// Usage: App with nested modules
const appBus = new HierarchicalBus();
const moduleBus = new HierarchicalBus(appBus);
const subModuleBus = new HierarchicalBus(moduleBus);

// Local event
subModuleBus.emit('button-clicked', { id: 123 });

// Bubble up to parent
subModuleBus.emit('critical-error', { error: 'Oh no!' }, { bubble: true });

// Propagate down to all children
appBus.emit('theme-changed', { theme: 'dark' }, { propagate: true });

```

19.3 Backend Integration Strategies

LARC runs in the browser, but your data lives on a server. Let's build bridges between these two worlds.

19.3.1 API Gateway Component

Centralize all API calls in one component:

```
class APIGateway extends Component {
  init() {
    this.state = {
      baseUrl: '/api',
      token: localStorage.getItem('authToken'),
      requestQueue: [],
      online: navigator.onLine
    };

    // Listen for API requests
    this.on('api-request', this.handleRequest);
    this.on('auth-token-updated', (data) => {
      this.state.token = data.token;
    });

    // Handle online/offline
    window.addEventListener('online', () => {
      this.state.online = true;
      this.flushQueue();
    });
    window.addEventListener('offline', () => {
      this.state.online = false;
    });
  }

  async handleRequest({ method, endpoint, data, requestId }) {
    if (!this.state.online) {
      this.state.requestQueue.push({ method, endpoint, data, requestId });
      this.emit('api-offline', { requestId });
      return;
    }

    try {
      const response = await fetch(`${this.state.baseUrl}${endpoint}`, {
        method,
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Bearer ${this.state.token}`
        },
        body: data ? JSON.stringify(data) : undefined
      });
    }
  }
}
```

```

    });

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

    const result = await response.json();

    this.emit('api-success', {
      requestId,
      endpoint,
      result
    });

  } catch (error) {
    this.emit('api-error', {
      requestId,
      endpoint,
      error: error.message
    });
  }
}

async flushQueue() {
  const queue = [...this.state.requestQueue];
  this.state.requestQueue = [];

  for (const request of queue) {
    await this.handleRequest(request);
  }
}

render() {
  return null;
}
}

// Usage in other components
class UserProfile extends Component {
  loadUserData(userId) {
    const requestId = crypto.randomUUID();

    this.emit('api-request', {
      method: 'GET',
      endpoint: `/users/${userId}`,
      requestId
    });
  }
}

```

```

    this.once(`api-success`, (data) => {
      if (data.requestId === requestId) {
        this.state.user = data.result;
      }
    });

    this.once(`api-error`, (data) => {
      if (data.requestId === requestId) {
        this.state.error = data.error;
      }
    });
  }
}

```

19.3.2 WebSocket Integration

Real-time bidirectional communication:

```

class WebSocketBridge extends Component {
  init() {
    this.state = {
      connected: false,
      reconnectAttempts: 0,
      maxReconnectAttempts: 5
    };

    this.ws = null;
    this.connect();

    // Listen for outgoing messages
    this.on('ws-send', this.sendMessage);
    this.on('ws-disconnect', () => this.disconnect());
  }

  connect() {
    const protocol = window.location.protocol === 'https:' ? 'wss:' : 'ws:';
    const wsURL = `${protocol}://${window.location.host}/ws`;

    this.ws = new WebSocket(wsURL);

    this.ws.onopen = () => {
      this.state.connected = true;
      this.state.reconnectAttempts = 0;
      this.emit('ws-connected');
    };

    this.ws.onmessage = (event) => {

```

```
    try {
      const message = JSON.parse(event.data);
      // Emit as LARC message
      this.emit(message.type, message.data);
    } catch (error) {
      console.error('Invalid WebSocket message:', event.data);
    }
  };

  this.ws.onclose = () => {
    this.state.connected = false;
    this.emit('ws-disconnected');
    this.attemptReconnect();
  };

  this.ws.onerror = (error) => {
    this.emit('ws-error', { error });
  };
}

sendMessage({ type, data }) {
  if (this.ws && this.state.connected) {
    this.ws.send(JSON.stringify({ type, data }));
  } else {
    console.warn('WebSocket not connected, message queued');
    // Could implement a queue here
  }
}

attemptReconnect() {
  if (this.state.reconnectAttempts >= this.state.maxReconnectAttempts) {
    this.emit('ws-reconnect-failed');
    return;
  }

  this.state.reconnectAttempts++;
  const delay = Math.min(1000 * Math.pow(2, this.state.reconnectAttempts), 30000);

  setTimeout(() => {
    this.connect();
  }, delay);
}

disconnect() {
  if (this.ws) {
    this.ws.close();
    this.ws = null;
  }
}
```

```

    }
  }

  render() {
    return null;
  }
}

// Usage
class ChatComponent extends Component {
  init() {
    this.state = { messages: [] };

    // Receive messages from WebSocket
    this.on('chat-message', (data) => {
      this.state.messages.push(data);
    });
  }

  sendMessage(text) {
    // Send via WebSocket
    this.emit('ws-send', {
      type: 'chat-message',
      data: {
        text,
        userId: this.getCurrentUserId(),
        timestamp: Date.now()
      }
    });
  }
}

```

19.3.3 GraphQL Integration

For those who prefer structured queries:

```

class GraphQLClient extends Component {
  init() {
    this.state = {
      endpoint: '/graphql',
      cache: new Map()
    };

    this.on('graphql-query', this.executeQuery);
    this.on('graphql-mutation', this.executeMutation);
  }

  async executeQuery({ query, variables, requestId, cache = true }) {

```

```
// Check cache
const cacheKey = JSON.stringify({ query, variables });
if (cache && this.state.cache.has(cacheKey)) {
  this.emit('graphql-result', {
    requestId,
    data: this.state.cache.get(cacheKey),
    cached: true
  });
  return;
}

try {
  const response = await fetch(this.state.endpoint, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ query, variables })
  });

  const result = await response.json();

  if (result.errors) {
    throw new Error(result.errors[0].message);
  }

  // Cache result
  if (cache) {
    this.state.cache.set(cacheKey, result.data);
  }

  this.emit('graphql-result', {
    requestId,
    data: result.data
  });
} catch (error) {
  this.emit('graphql-error', {
    requestId,
    error: error.message
  });
}

}

async executeMutation({ mutation, variables, requestId }) {
  try {
    const response = await fetch(this.state.endpoint, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
```

```

        body: JSON.stringify({ query: mutation, variables })
    });

    const result = await response.json();

    if (result.errors) {
        throw new Error(result.errors[0].message);
    }

    // Invalidate cache on mutation
    this.state.cache.clear();

    this.emit('graphql-result', {
        requestId,
        data: result.data
    });

} catch (error) {
    this.emit('graphql-error', {
        requestId,
        error: error.message
    });
}

render() {
    return null;
}
}

// Usage
class UserList extends Component {
    loadUsers() {
        const requestId = crypto.randomUUID();

        this.emit('graphql-query', {
            query: `
                query GetUsers($limit: Int) {
                    users(limit: $limit) {
                        id
                        username
                        email
                    }
                }
            `,
            variables: { limit: 10 },
            requestId

```

```

});

this.once('graphql-result', (data) => {
  if (data.requestId === requestId) {
    this.state.users = data.data.users;
  }
});
}
}

```

19.4 Micro-Frontends with LARC

Split your monolith into independently deployable micro-frontends. It's like microservices, but with more JavaScript!

19.4.1 Module Federation Pattern

Load remote LARC modules dynamically:

```

class MicroFrontendLoader extends Component {
  init() {
    this.state = {
      modules: new Map(),
      loading: new Set()
    };

    this.on('load-module', this.loadModule);
    this.on('unload-module', this.unloadModule);
  }

  async loadModule({ name, url, props }) {
    if (this.state.modules.has(name)) {
      console.warn(`Module ${name} already loaded`);
      return;
    }

    if (this.state.loading.has(name)) {
      console.warn(`Module ${name} is already loading`);
      return;
    }

    this.state.loading.add(name);
    this.emit('module-loading', { name });

    try {
      // Dynamic import
      const module = await import(/* webpackIgnore: true */ url);

```

```

    // Initialize module with props
    const instance = new module.default(props);

    this.state.modules.set(name, instance);
    this.state.loading.delete(name);

    this.emit('module-loaded', { name });
  } catch (error) {
    this.state.loading.delete(name);
    this.emit('module-load-error', {
      name,
      error: error.message
    });
  }
}

unloadModule({ name }) {
  const module = this.state.modules.get(name);
  if (module && module.destroy) {
    module.destroy();
  }
  this.state.modules.delete(name);
  this.emit('module-unloaded', { name });
}

render() {
  return html`
    <div class="micro-frontend-container">
      ${Array.from(this.state.modules.entries()).map(([name, module]) => html`
        <div key=${name} class="module-wrapper" data-module=${name}>
          ${module.render ? module.render() : ''}
        </div>
      `)}
    </div>
  `;
}
}

// Usage: Load shopping cart from different server
loader.receive('load-module', {
  name: 'shopping-cart',
  url: 'https://cdn.example.com/modules/cart.js',
  props: {
    bus: sharedBus,
    apiEndpoint: '/api/cart'
  }
})

```

```
});
```

19.4.2 Shell Application Pattern

Create a shell that hosts multiple micro-frontends:

```
class MicroFrontendShell extends Component {
  init() {
    this.state = {
      activeModule: null,
      modules: {
        'dashboard': {
          url: '/modules/dashboard.js',
          title: 'Dashboard'
        },
        'products': {
          url: '/modules/products.js',
          title: 'Products'
        },
        'checkout': {
          url: '/modules/checkout.js',
          title: 'Checkout'
        }
      }
    }
  }

  this.loader = new MicroFrontendLoader({ bus: this.bus });
  this.on('navigate-to-module', this.navigateToModule);
}

async navigateToModule({ module }) {
  // Unload previous module
  if (this.state.activeModule) {
    this.emit('unload-module', { name: this.state.activeModule });
  }

  // Load new module
  const config = this.state.modules[module];
  if (config) {
    await this.emit('load-module', {
      name: module,
      url: config.url,
      props: {
        bus: this.bus,
        navigate: (to) => this.navigateToModule({ module: to })
      }
    });
  }
}
```

```

        this.state.activeModule = module;
    }
}

render() {
    return html`
        <div class="shell">
            <nav class="shell-nav">
                ${Object.entries(this.state.modules).map(([key, config]) => html`
                    <button
                        key=${key}
                        class=${this.state.activeModule === key ? 'active' : ''}
                        onclick=${() => this.navigateToModule({ module: key })}
                    >
                        ${config.title}
                    </button>
                `)}
            </nav>
            <main class="shell-content">
                ${this.loader.render()}
            </main>
        </div>
    `;
}
}

```

19.5 Plugin Systems

Let users extend your application with their own components.

19.5.1 Plugin Registry

```

class PluginRegistry extends Component {
    init() {
        this.state = {
            plugins: new Map(),
            hooks: new Map()
        };

        this.on('register-plugin', this.registerPlugin);
        this.on('unregister-plugin', this.unregisterPlugin);
        this.on('execute-hook', this.executeHook);
    }

    registerPlugin({ id, plugin }) {
        if (this.state.plugins.has(id)) {
            throw new Error(`Plugin ${id} already registered`);
        }
    }
}

```

```

}

// Validate plugin interface
if (!plugin.init || typeof plugin.init !== 'function') {
  throw new Error('Plugin must have an init() method');
}

this.state.plugins.set(id, plugin);

// Register plugin hooks
if (plugin.hooks) {
  Object.entries(plugin.hooks).forEach(([hookName, handler]) => {
    if (!this.state.hooks.has(hookName)) {
      this.state.hooks.set(hookName, []);
    }
    this.state.hooks.get(hookName).push({ id, handler });
  });
}

// Initialize plugin
plugin.init({
  bus: this.bus,
  emit: (type, data) => this.emit(type, data)
});

this.emit('plugin-registered', { id });
}

unregisterPlugin({ id }) {
  const plugin = this.state.plugins.get(id);
  if (!plugin) return;

  // Remove hooks
  this.state.hooks.forEach((handlers, hookName) => {
    this.state.hooks.set(
      hookName,
      handlers.filter(h => h.id !== id)
    );
  });

  // Cleanup plugin
  if (plugin.destroy) {
    plugin.destroy();
  }

  this.state.plugins.delete(id);
  this.emit('plugin-unregistered', { id });
}

```

```

}

async executeHook({ hook, data }) {
  const handlers = this.state.hooks.get(hook) || [];

  let result = data;

  for (const { id, handler } of handlers) {
    try {
      result = await handler(result);
    } catch (error) {
      console.error(`Plugin ${id} hook ${hook} failed:`, error);
    }
  }

  return result;
}

render() {
  return null;
}
}

// Example plugin
const analyticsPlugin = {
  init({ bus, emit }) {
    this.bus = bus;
    this.emit = emit;

    // Listen to all messages
    bus.on('*', (type, data) => {
      this.trackEvent(type, data);
    });
  },

  hooks: {
    'before-submit': async (formData) => {
      // Validate or transform data
      console.log('Analytics: Form submission', formData);
      return formData;
    },

    'after-navigation': async (route) => {
      // Track page view
      console.log('Analytics: Page view', route);
      return route;
    }
  }
}

```

```

},

trackEvent(type, data) {
  // Send to analytics service
  if (window.gtag) {
    window.gtag('event', type, data);
  }
},

destroy() {
  console.log('Analytics plugin destroyed');
}
};

// Register plugin
registry.receive('register-plugin', {
  id: 'analytics',
  plugin: analyticsPlugin
});

// Use hooks
const formData = { name: 'Alice', email: 'alice@example.com' };
registry.receive('execute-hook', {
  hook: 'before-submit',
  data: formData
}).then(result => {
  console.log('After hook:', result);
});

```

19.6 Middleware Patterns

Intercept and transform messages as they flow through your application.

19.6.1 Message Middleware

```

class MessageMiddleware extends Component {
  init() {
    this.state = {
      middlewares: []
    };

    this.on('register-middleware', this.registerMiddleware);

    // Intercept all messages
    this.interceptBus();
  }
}

```

```
registerMiddleware({ middleware, priority = 0 }) {
  this.state.middlewares.push({ middleware, priority });

  // Sort by priority (higher first)
  this.state.middlewares.sort((a, b) => b.priority - a.priority);
}

interceptBus() {
  const originalEmit = this.bus.emit.bind(this.bus);

  this.bus.emit = async (type, data) => {
    let context = {
      type,
      data,
      timestamp: Date.now(),
      stopped: false
    };

    // Run through middleware chain
    for (const { middleware } of this.state.middlewares) {
      context = await middleware(context);

      if (context.stopped) {
        return; // Stop propagation
      }
    }

    // Emit transformed message
    originalEmit(context.type, context.data);
  };
}

render() {
  return null;
}

// Example middleware: Logging
const loggingMiddleware = async (context) => {
  console.log(`[Middleware] ${context.type}`, context.data);
  return context;
};

// Example middleware: Rate limiting
const rateLimitMiddleware = (() => {
  const limits = new Map();
```

```

return async (context) => {
  const key = context.type;
  const now = Date.now();
  const limit = limits.get(key) || { count: 0, resetAt: now + 1000 };

  if (now > limit.resetAt) {
    limit.count = 0;
    limit.resetAt = now + 1000;
  }

  limit.count++;

  if (limit.count > 10) {
    console.warn(`Rate limit exceeded for ${key}`);
    context.stopped = true;
  }

  limits.set(key, limit);
  return context;
};
})();

// Example middleware: Transform
const transformMiddleware = async (context) => {
  // Add metadata to all messages
  context.data = {
    ...context.data,
    _meta: {
      timestamp: context.timestamp,
      version: '1.0'
    }
  };
  return context;
};

// Register middleware
middleware.receive('register-middleware', {
  middleware: loggingMiddleware,
  priority: 100
});

middleware.receive('register-middleware', {
  middleware: rateLimitMiddleware,
  priority: 90
});

middleware.receive('register-middleware', {

```

```

    middleware: transformMiddleware,
    priority: 80
  });

```

19.6.2 Async Middleware with Error Handling

```

class AsyncMiddleware extends Component {
  init() {
    this.state = {
      middlewares: []
    };
  }

  async runMiddleware(context) {
    try {
      for (const middleware of this.state.middlewares) {
        context = await middleware(context);

        if (context.stopped) {
          break;
        }
      }

      return context;
    } catch (error) {
      console.error('Middleware error:', error);

      // Emit error event
      this.emit('middleware-error', {
        error: error.message,
        context
      });

      // Stop propagation on error
      context.stopped = true;
      return context;
    }
  }
}

// Example: Authentication middleware
const authMiddleware = async (context) => {
  const protectedMessages = ['api-request', 'user-action'];

  if (protectedMessages.includes(context.type)) {
    const token = localStorage.getItem('authToken');

```

```
if (!token) {
  console.warn('Authentication required');
  context.stopped = true;

  // Redirect to login
  setTimeout(() => {
    bus.emit('navigate', { route: '/login' });
  }, 0);
}

return context;
};
```

19.7 Conclusion

These advanced patterns are powerful tools in your LARC toolkit. Use them judiciously. Not every application needs a multi-bus architecture or a plugin system. But when you do need them, you'll be glad you have them.

Remember: the goal is to build maintainable, scalable applications—not to use every pattern just because you can. Choose patterns that solve real problems in your codebase, and your future self (and your teammates) will thank you.

In the next chapter, we'll take your LARC application from development to production, covering deployment strategies, performance optimization, and how to sleep soundly knowing your app is running smoothly in the wild.

Chapter 20

Deployment and Production

“In development, everything works. In production, nothing works. In between is where your career is made.” — Murphy’s Law of Software Development

You’ve built your LARC application. It’s beautiful. It’s tested. It works perfectly on your machine. Now comes the moment of truth: deploying it to production, where real users with real problems will find real bugs you never knew existed.

The good news? LARC’s simplicity makes deployment straightforward. The better news? We’re about to make it even easier.

20.1 Build Considerations (Or Lack Thereof)

One of LARC’s most delightful features is that it doesn’t require a build step. No webpack. No babel. No spending three days configuring bundlers. You can literally serve your `.js` files directly to browsers.

20.1.1 The No-Build Approach

For small to medium applications, skip the build entirely:

```
my-app/
|-- index.html
|-- app.js
|-- components/
|   |-- header.js
|   |-- sidebar.js
|   `-- footer.js
`-- lib/
    `-- larc.js

<!DOCTYPE html>
<html>
<head>
  <title>My LARC App</title>
  <script type="module" src="app.js"></script>
```

```
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

```
// app.js
import { createBus, Component, html } from './lib/larc.js';
import { Header } from './components/header.js';
import { Sidebar } from './components/sidebar.js';
import { Footer } from './components/footer.js';

const bus = createBus();

// Initialize components
new Header({ bus, target: document.querySelector('#header') });
new Sidebar({ bus, target: document.querySelector('#sidebar') });
new Footer({ bus, target: document.querySelector('#footer') });
```

Deploy this to any static file server. Done. Seriously. That's it.

20.1.2 When You Actually Need a Build Step

Sometimes you want to optimize. Fair enough. Here's when a build makes sense:

1. **Minification:** Reduce file size for faster loading
2. **Code splitting:** Load only what's needed for each page
3. **Tree shaking:** Remove unused code
4. **Transpilation:** Support older browsers (if you must)
5. **Asset optimization:** Compress images, inline critical CSS

20.1.3 Minimal Build with esbuild

esbuild is fast enough that you'll think it's broken:

```
// build.js
import * as esbuild from 'esbuild';

await esbuild.build({
  entryPoints: ['src/app.js'],
  bundle: true,
  minify: true,
  sourcemap: true,
  target: ['es2020'],
  outfile: 'dist/app.js',
  format: 'esm'
});

console.log('Build complete!');
```

Run it:

```
node build.js
```

That's your entire build process. Add it to `package.json`:

```
{
  "scripts": {
    "build": "node build.js",
    "dev": "node build.js --watch"
  }
}
```

20.1.4 Code Splitting for Larger Apps

Split your code by route or feature:

```
// build.js
import * as esbuild from 'esbuild';

await esbuild.build({
  entryPoints: [
    'src/app.js',
    'src/pages/home.js',
    'src/pages/about.js',
    'src/pages/contact.js'
  ],
  bundle: true,
  minify: true,
  splitting: true,
  format: 'esm',
  outdir: 'dist',
  chunkNames: 'chunks/[name]-[hash]'
});
```

Then lazy load pages:

```
class Router extends Component {
  async loadPage(pageName) {
    this.state.loading = true;

    try {
      // Dynamic import
      const module = await import(`./pages/${pageName}.js`);
      const PageComponent = module.default;

      this.state.currentPage = new PageComponent({
        bus: this.bus,
        target: this.pageContainer
      });
    }
  }
}
```

```

        this.state.loading = false;

    } catch (error) {
        console.error('Failed to load page:', error);
        this.state.error = error.message;
    }
}

render() {
    if (this.state.loading) {
        return html`<div class="loading">Loading...</div>`;
    }

    if (this.state.error) {
        return html`<div class="error">Error: ${this.state.error}</div>`;
    }

    return html`<div ref=${el => this.pageContainer = el}></div>`;
}
}

```

20.1.5 TypeScript Integration (Optional)

If you're into type safety:

```

// tsconfig.json
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ES2020",
    "moduleResolution": "node",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "outDir": "dist"
  },
  "include": ["src/**/*"]
}

```

```

# Build
tsc && node build.js

```

Or use esbuild to handle TypeScript directly:

```

await esbuild.build({
  entryPoints: ['src/app.ts'],
  bundle: true,
  minify: true,

```

```
loader: { '.ts': 'ts' },  
outfile: 'dist/app.js'  
});
```

20.2 CDN Deployment

Content Delivery Networks make your app fast worldwide. Users in Tokyo load from Tokyo. Users in Paris load from Paris. Everyone's happy.

20.2.1 Static File Hosting

Deploy to any CDN that serves static files:

Cloudflare Pages:

```
# Install Wrangler CLI  
npm install -g wrangler  
  
# Deploy  
wrangler pages publish dist
```

Netlify:

```
# Install Netlify CLI  
npm install -g netlify-cli  
  
# Deploy  
netlify deploy --dir=dist --prod
```

Vercel:

```
# Install Vercel CLI  
npm install -g vercel  
  
# Deploy  
vercel --prod
```

20.2.2 Configuration Files

Most CDN providers want a config file:

Cloudflare Pages (_headers):

```
/*  
  Cache-Control: public, max-age=31536000, immutable  
  
/index.html  
  Cache-Control: no-cache  
  
/app.js  
  Cache-Control: public, max-age=31536000, immutable
```

```
/service-worker.js
  Cache-Control: no-cache
```

Netlify (netlify.toml):

```
[build]
  publish = "dist"
  command = "npm run build"

[[headers]]
  for = "/*.js"
  [headers.values]
    Cache-Control = "public, max-age=31536000, immutable"

[[headers]]
  for = "/index.html"
  [headers.values]
    Cache-Control = "no-cache"

[[redirects]]
  from = "/*"
  to = "/index.html"
  status = 200
```

Vercel (vercel.json):

```
{
  "buildCommand": "npm run build",
  "outputDirectory": "dist",
  "routes": [
    {
      "src": "/(.*\\.js)",
      "headers": {
        "Cache-Control": "public, max-age=31536000, immutable"
      }
    },
    {
      "src": "/index.html",
      "headers": {
        "Cache-Control": "no-cache"
      }
    },
    {
      "handle": "filesystem"
    },
    {
      "src": "/(.*)",
      "dest": "/index.html"
    }
  ]
}
```

```

    }
  ]
}

```

20.2.3 Asset Fingerprinting

Add content hashes to filenames for cache busting:

```

// build.js
import * as esbuild from 'esbuild';
import { createHash } from 'crypto';
import { readFileSync, writeFileSync } from 'fs';

// Build
await esbuild.build({
  entryPoints: ['src/app.js'],
  bundle: true,
  minify: true,
  metafile: true,
  outfile: 'dist/app.js'
});

// Add hash to filename
const content = readFileSync('dist/app.js');
const hash = createHash('sha256').update(content).digest('hex').slice(0, 8);
const hashedFilename = `app.${hash}.js`;

// Rename file
renameSync('dist/app.js', `dist/${hashedFilename}`);

// Update index.html
let html = readFileSync('src/index.html', 'utf-8');
html = html.replace('app.js', hashedFilename);
writeFileSync('dist/index.html', html);

console.log(`Built: ${hashedFilename}`);

```

20.3 Caching Strategies

Caching is the art of remembering things so you don't have to fetch them again. Get it right, and your app is lightning fast. Get it wrong, and users see stale content for months.

20.3.1 Browser Cache Headers

Set appropriate cache headers for different file types:

```

// Edge function (Cloudflare Workers example)
export default {

```

```

async fetch(request) {
  const url = new URL(request.url);
  const response = await fetch(request);

  // Clone response so we can modify headers
  const newResponse = new Response(response.body, response);

  if (url.pathname.endsWith('.js') || url.pathname.endsWith('.css')) {
    // Cache JavaScript and CSS for 1 year
    newResponse.headers.set(
      'Cache-Control',
      'public, max-age=31536000, immutable'
    );
  } else if (url.pathname.endsWith('.html')) {
    // Don't cache HTML
    newResponse.headers.set(
      'Cache-Control',
      'no-cache, must-revalidate'
    );
  } else if (url.pathname.match(/\.(png|jpg|jpeg|gif|webp|svg)$/)) {
    // Cache images for 30 days
    newResponse.headers.set(
      'Cache-Control',
      'public, max-age=2592000'
    );
  }

  return newResponse;
}
};

```

20.3.2 Service Worker Caching

Implement offline support and faster loads:

```

// service-worker.js
const CACHE_NAME = 'larc-app-v1';
const URLS_TO_CACHE = [
  '/',
  '/index.html',
  '/app.js',
  '/styles.css',
  '/lib/larc.js'
];

self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)

```

```

        .then(cache => cache.addAll(URLS_TO_CACHE))
    );
});

self.addEventListener('fetch', (event) => {
    event.respondWith(
        caches.match(event.request)
            .then(response => {
                // Return cached version or fetch new
                if (response) {
                    return response;
                }

                return fetch(event.request).then(response => {
                    // Cache new responses
                    if (!response || response.status !== 200 || response.type !== 'basic') {
                        return response;
                    }

                    const responseToCache = response.clone();

                    caches.open(CACHE_NAME)
                        .then(cache => {
                            cache.put(event.request, responseToCache);
                        });

                    return response;
                });
            })
    );
});

self.addEventListener('activate', (event) => {
    // Clean up old caches
    event.waitUntil(
        caches.keys().then(cacheNames => {
            return Promise.all(
                cacheNames.map(cacheName => {
                    if (cacheName !== CACHE_NAME) {
                        return caches.delete(cacheName);
                    }
                })
            );
        })
    );
});
});

```

Register the service worker:

```
// app.js
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/service-worker.js')
      .then(registration => {
        console.log('Service Worker registered:', registration.scope);
      })
      .catch(error => {
        console.error('Service Worker registration failed:', error);
      });
  });
}
```

20.3.3 API Response Caching

Cache API responses intelligently:

```
class CachedAPIClient extends Component {
  init() {
    this.state = {
      cache: new Map(),
      cacheDurations: {
        'GET': 5 * 60 * 1000, // 5 minutes
        'POST': 0, // Don't cache
        'PUT': 0,
        'DELETE': 0
      }
    };

    this.on('api-request', this.handleRequest);
  }

  async handleRequest({ method, endpoint, data, requestId, bypassCache }) {
    const cacheKey = `${method}:${endpoint}:${JSON.stringify(data || {})}`;

    // Check cache for GET requests
    if (method === 'GET' && !bypassCache) {
      const cached = this.state.cache.get(cacheKey);

      if (cached && Date.now() < cached.expiresAt) {
        this.emit('api-success', {
          requestId,
          result: cached.data,
          cached: true
        });
        return;
      }
    }
  }
}
```

```
}

try {
  const response = await fetch(endpoint, {
    method,
    headers: { 'Content-Type': 'application/json' },
    body: data ? JSON.stringify(data) : undefined
  });

  const result = await response.json();

  // Cache GET responses
  if (method === 'GET') {
    this.state.cache.set(cacheKey, {
      data: result,
      expiresAt: Date.now() + this.state.cacheDurations[method]
    });
  } else {
    // Invalidate cache on mutations
    this.invalidateCache(endpoint);
  }

  this.emit('api-success', {
    requestId,
    result,
    cached: false
  });
} catch (error) {
  this.emit('api-error', {
    requestId,
    error: error.message
  });
}

}

invalidateCache(pattern) {
  // Remove cache entries matching pattern
  for (const key of this.state.cache.keys()) {
    if (key.includes(pattern)) {
      this.state.cache.delete(key);
    }
  }
}

render() {
  return null;
}
```

```

    }
  }
}

```

20.4 Performance Monitoring

You can't improve what you don't measure. Let's measure everything.

20.4.1 Real User Monitoring (RUM)

Track actual user experience:

```

class PerformanceMonitor extends Component {
  init() {
    this.state = {
      metrics: {}
    };

    // Capture Core Web Vitals
    this.measureWebVitals();

    // Monitor component render times
    this.monitorComponents();

    // Track custom metrics
    this.on('track-metric', this.trackMetric);
  }

  measureWebVitals() {
    // Largest Contentful Paint (LCP)
    new PerformanceObserver((list) => {
      const entries = list.getEntries();
      const lastEntry = entries[entries.length - 1];

      this.state.metrics.lcp = lastEntry.renderTime || lastEntry.loadTime;
      this.sendMetric('lcp', this.state.metrics.lcp);
    }).observe({ entryTypes: ['largest-contentful-paint'] });

    // First Input Delay (FID)
    new PerformanceObserver((list) => {
      const entries = list.getEntries();
      entries.forEach(entry => {
        this.state.metrics.fid = entry.processingStart - entry.startTime;
        this.sendMetric('fid', this.state.metrics.fid);
      });
    }).observe({ entryTypes: ['first-input'] });

    // Cumulative Layout Shift (CLS)

```

```

let clsScore = 0;
new PerformanceObserver((list) => {
  for (const entry of list.getEntries()) {
    if (!entry.hadRecentInput) {
      clsScore += entry.value;
    }
  }
  this.state.metrics.cls = clsScore;
  this.sendMetric('cls', clsScore);
}).observe({ entryTypes: ['layout-shift'] });

// Time to First Byte (TTFB)
const navigationEntry = performance.getEntriesByType('navigation')[0];
if (navigationEntry) {
  this.state.metrics.ttfb = navigationEntry.responseStart - navigationEntry.requestStart;
  this.sendMetric('ttfb', this.state.metrics.ttfb);
}
}

monitorComponents() {
  // Wrap component render methods to track timing
  const originalRender = Component.prototype.render;

  Component.prototype.render = function(...args) {
    const start = performance.now();
    const result = originalRender.apply(this, args);
    const duration = performance.now() - start;

    if (duration > 16) { // Slower than 60fps
      this.bus.emit('slow-render', {
        component: this.constructor.name,
        duration
      });
    }

    return result;
  };

  this.on('slow-render', (data) => {
    this.sendMetric('slow-render', data);
  });
}

trackMetric({ name, value, tags }) {
  this.state.metrics[name] = value;
  this.sendMetric(name, value, tags);
}

```

```

sendMetric(name, value, tags = {}) {
  // Send to analytics service
  const payload = {
    metric: name,
    value,
    tags: {
      ...tags,
      url: window.location.pathname,
      userAgent: navigator.userAgent,
      timestamp: Date.now()
    }
  };

  // Use sendBeacon for reliability
  navigator.sendBeacon('/api/metrics', JSON.stringify(payload));
}

render() {
  // Optional: Display metrics in dev mode
  if (process.env.NODE_ENV === 'development') {
    return html`
      <div class="perf-monitor">
        <h4>Performance Metrics</h4>
        <dl>
          <dt>LCP</dt>
          <dd>${this.state.metrics.lcp?.toFixed(2)}ms</dd>
          <dt>FID</dt>
          <dd>${this.state.metrics.fid?.toFixed(2)}ms</dd>
          <dt>CLS</dt>
          <dd>${this.state.metrics.cls?.toFixed(3)}</dd>
          <dt>TTFB</dt>
          <dd>${this.state.metrics.ttfb?.toFixed(2)}ms</dd>
        </dl>
      </div>
    `;
  }

  return null;
}
}

```

20.4.2 Custom Performance Marks

Track specific operations:

```

class DataLoader extends Component {
  async loadUserData(userId) {

```

```

performance.mark('load-user-start');

try {
  const response = await fetch(`/api/users/${userId}`);
  const data = await response.json();

  performance.mark('load-user-end');
  performance.measure('load-user', 'load-user-start', 'load-user-end');

  const measurement = performance.getEntriesByName('load-user')[0];

  this.emit('track-metric', {
    name: 'user-load-time',
    value: measurement.duration,
    tags: { userId }
  });

  this.state.user = data;
} catch (error) {
  performance.mark('load-user-error');
  this.emit('track-metric', {
    name: 'user-load-error',
    value: 1,
    tags: { userId, error: error.message }
  });
}
}
}

```

20.4.3 Bundle Size Monitoring

Track your bundle size over time:

```

// build.js
import * as esbuild from 'esbuild';
import { statSync, writeFileSync } from 'fs';

const result = await esbuild.build({
  entryPoints: ['src/app.js'],
  bundle: true,
  minify: true,
  metafile: true,
  outfile: 'dist/app.js'
});

// Analyze bundle
const stats = statSync('dist/app.js');

```

```

const bundleSize = stats.size;
const bundleSizeKB = (bundleSize / 1024).toFixed(2);

console.log(`Bundle size: ${bundleSizeKB} KB`);

// Save to history
const history = {
  timestamp: new Date().toISOString(),
  size: bundleSize,
  sizeKB: bundleSizeKB
};

writeFileSync('build-stats.json', JSON.stringify(history, null, 2));

// Fail build if bundle is too large
const MAX_SIZE_KB = 500;
if (parseFloat(bundleSizeKB) > MAX_SIZE_KB) {
  throw new Error(`Bundle size ${bundleSizeKB} KB exceeds limit of ${MAX_SIZE_KB} KB`);
}

```

20.5 Production Debugging

Debugging production is like debugging with one hand tied behind your back and the lights off. Here's how to see in the dark.

20.5.1 Source Maps

Always deploy source maps (but protect them):

```

// build.js
await esbuild.build({
  entryPoints: ['src/app.js'],
  bundle: true,
  minify: true,
  sourcemap: 'external', // Creates separate .map file
  outfile: 'dist/app.js'
});

```

Serve source maps only to authenticated users:

```

// Edge function
export default {
  async fetch(request) {
    const url = new URL(request.url);

    // Protect source maps
    if (url.pathname.endsWith('.map')) {
      const authToken = request.headers.get('Authorization');

```

```

    if (!isValidDevToken(authToken)) {
      return new Response('Unauthorized', { status: 401 });
    }
  }

  return fetch(request);
}
};

```

20.5.2 Remote Error Tracking

Integrate with error tracking services:

```

class ErrorTracker extends Component {
  init() {
    // Initialize error tracking (e.g., Sentry)
    if (window.Sentry) {
      window.Sentry.init({
        dsn: 'YOUR_SENTRY_DSN',
        environment: process.env.NODE_ENV,
        release: process.env.APP_VERSION,
        beforeSend(event, hint) {
          // Add custom context
          event.contexts = {
            ...event.contexts,
            app: {
              userId: localStorage.getItem('userId'),
              sessionId: sessionStorage.getItem('sessionId')
            }
          };
          return event;
        }
      });
    }

    // Catch global errors
    window.addEventListener('error', (event) => {
      this.trackError({
        message: event.message,
        stack: event.error?.stack,
        source: event.filename,
        line: event.lineno,
        column: event.colno
      });
    });

    // Catch promise rejections

```

```

window.addEventListener('unhandledrejection', (event) => {
  this.trackError({
    message: event.reason?.message || 'Unhandled Promise Rejection',
    stack: event.reason?.stack
  });
});

// Listen for application errors
this.on('app-error', this.trackError);
}

trackError(error) {
  if (window.Sentry) {
    window.Sentry.captureException(error);
  }

  // Also log to our own service
  navigator.sendBeacon('/api/errors', JSON.stringify({
    ...error,
    timestamp: new Date().toISOString(),
    url: window.location.href,
    userAgent: navigator.userAgent
  }));
}

render() {
  return null;
}
}

```

20.5.3 Feature Flags

Control features in production without deploying:

```

class FeatureFlags extends Component {
  init() {
    this.state = {
      flags: {},
      loading: true
    };

    this.loadFlags();
    this.on('check-flag', this.checkFlag);
  }

  async loadFlags() {
    try {
      const response = await fetch('/api/feature-flags');
    }
  }
}

```

```

        this.state.flags = await response.json();
        this.state.loading = false;
        this.emit('flags-loaded');
    } catch (error) {
        console.error('Failed to load feature flags:', error);
        this.state.loading = false;
    }
}

checkFlag({ flag, defaultValue = false }) {
    if (this.state.loading) {
        return defaultValue;
    }

    return this.state.flags[flag] ?? defaultValue;
}

render() {
    return null;
}
}

// Usage
class NewFeature extends Component {
    init() {
        this.state = { enabled: false };

        this.on('flags-loaded', () => {
            this.emit('check-flag', { flag: 'new-feature-enabled' });
        });

        this.on('flag-result', ({ flag, value }) => {
            if (flag === 'new-feature-enabled') {
                this.state.enabled = value;
            }
        });
    }

    render() {
        if (!this.state.enabled) {
            return html`<div>Coming soon!</div>`;
        }

        return html`<div class="new-feature">New feature content</div>`;
    }
}

```

20.6 Versioning and Upgrades

Manage versions without breaking production.

20.6.1 Semantic Versioning

Track your app version:

```
// version.js
export const VERSION = '1.2.3';
export const BUILD_DATE = '2025-12-04T10:30:00Z';
```

Display in your app:

```
class AppFooter extends Component {
  render() {
    return html`
      <footer>
        <span>v${VERSION}</span>
        <span>Built: ${new Date(BUILD_DATE).toLocaleString()}</span>
      </footer>
    `;
  }
}
```

20.6.2 Update Notifications

Notify users when a new version is available:

```
class UpdateChecker extends Component {
  init() {
    this.state = {
      currentVersion: VERSION,
      latestVersion: VERSION,
      updateAvailable: false
    };

    this.checkForUpdates();

    // Check every 30 minutes
    setInterval(() => this.checkForUpdates(), 30 * 60 * 1000);
  }

  async checkForUpdates() {
    try {
      const response = await fetch('/version.json', {
        cache: 'no-cache'
      });
      const data = await response.json();
```

```

    if (data.version !== this.state.currentVersion) {
      this.state.latestVersion = data.version;
      this.state.updateAvailable = true;
      this.emit('update-available', {
        current: this.state.currentVersion,
        latest: data.version
      });
    }
  } catch (error) {
    console.error('Failed to check for updates:', error);
  }
}

render() {
  if (!this.state.updateAvailable) {
    return null;
  }

  return html`
    <div class="update-banner">
      <p>A new version (${this.state.latestVersion}) is available!</p>
      <button onclick=${() => window.location.reload()}>
        Refresh Now
      </button>
      <button onclick=${() => this.state.updateAvailable = false}>
        Later
      </button>
    </div>
  `;
}
}

```

20.6.3 Rolling Deployments

Deploy gradually to minimize risk:

```

// Edge function for gradual rollout
export default {
  async fetch(request) {
    const url = new URL(request.url);

    // Determine which version to serve
    const userId = getUserIdFromRequest(request);
    const rolloutPercent = 10; // Serve v2 to 10% of users

    const hash = hashString(userId);
    const bucket = hash % 100;
  }
}

```

```
if (bucket < rolloutPercent) {  
  // Serve new version  
  return fetch(`${url.origin}/v2${url.pathname}`);  
} else {  
  // Serve current version  
  return fetch(request);  
}  
}  
};  
  
function hashString(str) {  
  let hash = 0;  
  for (let i = 0; i < str.length; i++) {  
    hash = ((hash << 5) - hash) + str.charCodeAt(i);  
    hash |= 0;  
  }  
  return Math.abs(hash);  
}
```

20.7 Deployment Checklist

Before you deploy to production, verify:

- ☐ All tests pass
- ☐ Bundle is minified and gzipped
- ☐ Source maps are generated (and protected)
- ☐ Cache headers are configured correctly
- ☐ Service worker is registered (if using)
- ☐ Error tracking is enabled
- ☐ Performance monitoring is active
- ☐ Feature flags are configured
- ☐ API endpoints point to production
- ☐ Environment variables are set
- ☐ Database migrations are applied (if applicable)
- ☐ SSL certificate is valid
- ☐ CDN is configured with correct origins
- ☐ Monitoring alerts are configured
- ☐ Rollback plan is documented
- ☐ Team is notified of deployment

20.8 Monitoring Production Health

Set up health checks and dashboards:

```
class HealthCheck extends Component {  
  init() {  
    this.state = {
```

```
    status: 'unknown',
    checks: {}
  };

  this.runHealthChecks();

  // Run checks every 60 seconds
  setInterval(() => this.runHealthChecks(), 60000);
}

async runHealthChecks() {
  const checks = {
    api: await this.checkAPI(),
    websocket: await this.checkWebSocket(),
    localStorage: this.checkLocalStorage(),
    serviceWorker: await this.checkServiceWorker()
  };

  this.state.checks = checks;

  const allHealthy = Object.values(checks).every(c => c.status === 'ok');
  this.state.status = allHealthy ? 'healthy' : 'degraded';

  if (!allHealthy) {
    this.emit('health-check-failed', { checks });
  }
}

async checkAPI() {
  try {
    const response = await fetch('/api/health', { timeout: 5000 });
    return { status: response.ok ? 'ok' : 'error' };
  } catch (error) {
    return { status: 'error', error: error.message };
  }
}

async checkWebSocket() {
  // Check if WebSocket connection is alive
  return { status: 'ok' }; // Simplified
}

checkLocalStorage() {
  try {
    localStorage.setItem('test', 'test');
    localStorage.removeItem('test');
    return { status: 'ok' };
  }
}
```

```
    } catch (error) {  
      return { status: 'error', error: error.message };  
    }  
  }  
  
  async checkServiceWorker() {  
    if ('serviceWorker' in navigator) {  
      const registration = await navigator.serviceWorker.getRegistration();  
      return { status: registration ? 'ok' : 'not-registered' };  
    }  
    return { status: 'not-supported' };  
  }  
  
  render() {  
    return null;  
  }  
}
```

20.9 Conclusion

Deploying a LARC application is refreshingly simple. No complex build pipelines. No Docker orchestration. No Kubernetes manifests that would make a Vogon poet proud. Just clean, modern JavaScript that runs anywhere.

But simplicity doesn't mean carelessness. Monitor your app. Cache intelligently. Track errors. Use feature flags. Version carefully. And always have a rollback plan.

Your LARC application is now live, serving real users, solving real problems. You've built something with vanilla JavaScript that's faster, simpler, and more maintainable than most framework-heavy applications. That's worth celebrating.

Now go forth and deploy. And when something breaks (it will), you'll have the tools to fix it quickly. That's the LARC way.

Congratulations—you've completed Building with LARC: A Reference Manual!

Chapter 21

Core Components Reference

“Good API documentation is like a lighthouse: it doesn’t just show you where you are, it shows you where you can go.”

— Developer wisdom, hard-won

This chapter provides comprehensive API documentation for LARC’s core components—the foundational building blocks that power every LARC application. Unlike the narrative chapters that teach concepts through examples, this is reference material designed for repeated consultation during development.

Think of this chapter as your field guide. When you need to know exactly which attributes **pan-bus** accepts, what events **pan-theme-provider** emits, or how to programmatically control **pan-routes**, you’ll find your answers here.

We’ll cover four essential components:

- **pan-bus**: The message bus that enables component communication
- **pan-theme-provider**: Centralized theme management with system preference detection
- **pan-theme-toggle**: UI component for theme switching
- **pan-routes**: Runtime-configurable message routing system

Each component section follows the same structure: overview, usage guidance, installation, complete attribute/method/event reference, working examples, and troubleshooting.

21.1 pan-bus

21.1.1 Overview

pan-bus is the central message bus for LARC applications. It implements a publish-subscribe pattern that enables decoupled communication between components. The enhanced version includes memory management, rate limiting, message validation, routing capabilities, and comprehensive debugging tools.

Every LARC application needs exactly one **pan-bus** instance, typically placed in the document’s `<head>` or at the root of the `<body>`.

21.1.2 When to Use

Use **pan-bus** when:

- Building any LARC application (it's foundational)
- You need decoupled component communication
- You want components to react to application state changes without direct coupling
- You're implementing request-response patterns between components

Don't use **pan-bus** when:

- Building a pure static site with no interactivity
- All your components can communicate through direct DOM manipulation (though PAN is usually better)

21.1.3 Installation and Setup

The simplest setup requires no configuration:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/core/pan-bus.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>
  <!-- Your application -->
</body>
</html>
```

For production applications, you'll typically add configuration:

```
<pan-bus
  max-retained="2000"
  max-message-size="2097152"
  debug="false"
  enable-routing="true"
  allow-global-wildcard="false">
</pan-bus>
```

The bus automatically announces readiness by:

1. Setting `window.__panReady = true`
2. Dispatching a `pan:sys.ready` event
3. Exposing global APIs at `window.pan.bus`, `window.pan.routes`, and `window.pan.debug`

21.1.4 Attributes

Attribute	Type	Default	Description
<code>max-retained</code>	Integer	1000	Maximum number of retained messages. When exceeded, oldest messages are evicted using LRU strategy.
<code>max-message-size</code>	Integer	1048576 (1MB)	Maximum total message size in bytes, including metadata.
<code>max-payload-size</code>	Integer	524288 (512KB)	Maximum payload (data field) size in bytes.
<code>cleanup-interval</code>	Integer	30000 (30s)	Milliseconds between automatic cleanup of dead subscriptions and stale rate limit data.
<code>rate-limit</code>	Integer	1000	Maximum messages per client per second.
<code>allow-global-wildcard</code>	Boolean	<code>true</code>	Whether to allow <code>*</code> wildcard subscriptions (subscribe to all messages). Set to <code>false</code> for security in production.
<code>debug</code>	Boolean	<code>false</code>	Enable verbose console logging for all bus operations.
<code>enable-routing</code>	Boolean	<code>false</code>	Enable the declarative routing system (see <code>pan-routes</code> section).
<code>enable-tracing</code>	Boolean	<code>false</code>	Enable message tracing for debugging (captures full message history).

Example with configuration:

```
<!-- Production configuration -->
<pan-bus
  max-retained="5000"
  max-message-size="2097152"
  rate-limit="2000"
  allow-global-wildcard="false"
  enable-routing="true">
</pan-bus>
```

21.1.5 Methods

All methods are available on the `pan-bus` element instance and through the global `window.pan.bus` reference.

21.1.5.1 `publish(topic, data, options)`

Publishes a message to the bus.

Parameters: - **topic** (String, required): Message topic/identifier - **data** (Any, required): Message payload (must be JSON-serializable) - **options** (Object, optional): Additional message options - **retain** (Boolean): Store message for late subscribers - **clientId** (String): Publisher identifier for rate limiting - Any other fields are included in the message

Returns: undefined

Example:

```
const bus = document.querySelector('pan-bus');

// Simple publish
bus.publish('user.login', { userId: '123', name: 'Alice' });

// Publish with retention
bus.publish('app.config', { theme: 'dark' }, { retain: true });

// Publish with metadata
bus.publish('sensor.update',
  { temperature: 22.5, humidity: 45 },
  {
    retain: true,
    source: 'sensor-01',
    priority: 'high'
  }
);
```

21.1.5.2 subscribe(topics, handler)

Subscribes to one or more topic patterns.

Parameters: - **topics** (String or Array, required): Topic pattern(s) to subscribe to. Supports wild-cards: `user.*` matches `user.login`, `user.logout`, etc. - **handler** (Function, required): Callback function receiving (**message**) when matching messages arrive

Returns: Function - Unsubscribe function to call when done

Example:

```
const bus = document.querySelector('pan-bus');

// Subscribe to single topic
const unsub1 = bus.subscribe('user.login', (msg) => {
  console.log('User logged in:', msg.data);
});

// Subscribe to multiple topics
const unsub2 = bus.subscribe(['cart.add', 'cart.remove'], (msg) => {
  console.log('Cart changed:', msg.topic, msg.data);
});
```

```
// Subscribe with wildcard
const unsub3 = bus.subscribe('sensor.*', (msg) => {
  console.log('Sensor update:', msg.topic, msg.data);
});

// Unsubscribe when done
unsub1();
unsub2();
unsub3();
```

21.1.5.3 PanBusEnhanced.matches(topic, pattern)

Static method to test if a topic matches a pattern.

Parameters: - **topic** (String, required): Topic to test - **pattern** (String, required): Pattern to match against (supports wildcards)

Returns: Boolean - True if topic matches pattern

Example:

```
const Bus = customElements.get('pan-bus');

Bus.matches('user.login', 'user.*'); // true
Bus.matches('user.login', 'user.login'); // true
Bus.matches('user.login', 'cart.*'); // false
Bus.matches('user.login', '*'); // true
Bus.matches('sensor.temperature', 'sensor.temp*'); // true (wildcard in pattern)
```

21.1.6 Events

The bus listens for these custom events (dispatched by components):

21.1.6.1 pan:hello

Registers a client with the bus.

Detail payload:

```
{
  id: String, // Unique client identifier
  caps: Array<String> // Optional client capabilities
}
```

Example:

```
document.dispatchEvent(new CustomEvent('pan:hello', {
  bubbles: true,
  detail: {
    id: 'my-component-123',
    caps: ['request-response', 'streaming']
  }
}));
```

```
    }  
  }));
```

21.1.6.2 pan:subscribe

Subscribes to topics.

Detail payload:

```
{  
  topics: Array<String>, // Topic patterns to subscribe to  
  clientId: String,      // Optional client identifier  
  options: {  
    retained: Boolean    // Request retained messages on subscription  
  }  
}
```

Example:

```
document.dispatchEvent(new CustomEvent('pan:subscribe', {  
  bubbles: true,  
  detail: {  
    topics: ['user.*', 'app.config'],  
    clientId: 'dashboard-widget',  
    options: { retained: true }  
  }  
}));
```

21.1.6.3 pan:unsubscribe

Unsubscribes from topics.

Detail payload:

```
{  
  topics: Array<String>, // Topic patterns to unsubscribe from  
  clientId: String        // Optional client identifier  
}
```

21.1.6.4 pan:publish

Publishes a message.

Detail payload:

```
{  
  topic: String,          // Message topic  
  data: Any,              // Message payload (JSON-serializable)  
  retain: Boolean,        // Store for late subscribers  
  clientId: String,       // Publisher identifier  
  // Any additional fields  
}
```

21.1.6.5 pan:request

Publishes a request message (same as `pan:publish` but semantic distinction).

21.1.6.6 pan:reply

Delivers a reply message (bypasses normal routing).

21.1.6.7 pan:sys.stats

Requests bus statistics.

Response via `pan:deliver`:

```
{
  topic: 'pan:sys.stats',
  data: {
    published: Number,      // Total messages published
    delivered: Number,      // Total messages delivered
    dropped: Number,        // Messages dropped (rate limit)
    retainedEvicted: Number, // Retained messages evicted
    subsCleanedUp: Number,  // Dead subscriptions cleaned
    errors: Number,         // Total errors
    subscriptions: Number,  // Current subscription count
    clients: Number,        // Registered clients
    retained: Number,       // Current retained messages
    config: Object          // Current configuration
  }
}
```

21.1.6.8 pan:sys.clear-retained

Clears retained messages.

Detail payload:

```
{
  pattern: String // Optional: only clear topics matching pattern
}
```

Example:

```
// Clear all retained messages
document.dispatchEvent(new CustomEvent('pan:sys.clear-retained', {
  bubbles: true,
  detail: {}
}));

// Clear specific pattern
document.dispatchEvent(new CustomEvent('pan:sys.clear-retained', {
  bubbles: true,
```

```
    detail: { pattern: 'sensor.*' }
  }));
```

The bus dispatches these events:

21.1.6.9 pan:sys.ready

Dispatched when bus is ready.

Detail payload:

```
{
  enhanced: true,
  routing: Boolean,    // Whether routing is enabled
  tracing: Boolean,    // Whether tracing is enabled
  config: Object       // Full configuration
}
```

21.1.6.10 pan:sys.error

Dispatched when errors occur.

Detail payload:

```
{
  code: String,        // Error code (e.g., 'RATE_LIMIT_EXCEEDED')
  message: String,     // Human-readable error message
  details: Object      // Additional error context
}
```

21.1.6.11 pan:deliver

Dispatched to deliver messages to subscribers.

Detail payload: The full message object with these guaranteed fields:

```
{
  topic: String,       // Message topic
  data: Any,           // Message payload
  id: String,          // Unique message ID (UUID)
  ts: Number           // Timestamp (milliseconds since epoch)
  // Plus any additional fields from publish
}
```

21.1.7 Working Examples

21.1.7.1 Basic Publish-Subscribe

```
// Component A: Subscribe
class DashboardWidget extends HTMLElement {
  connectedCallback() {
```

```

    const bus = document.querySelector('pan-bus');

    this.unsubscribe = bus.subscribe('user.login', (msg) => {
      this.innerHTML = `Welcome, ${msg.data.name}!`;
    });
  }

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

// Component B: Publish
class LoginForm extends HTMLElement {
  handleLogin(userId, name) {
    const bus = document.querySelector('pan-bus');
    bus.publish('user.login', { userId, name });
  }
}

```

21.1.7.2 Using Retained Messages

```

// Publish configuration once
const bus = document.querySelector('pan-bus');
bus.publish('app.config',
  {
    apiUrl: 'https://api.example.com',
    theme: 'dark',
    language: 'en'
  },
  { retain: true }
);

// Late subscriber gets retained message
class SettingsPanel extends HTMLElement {
  connectedCallback() {
    const bus = document.querySelector('pan-bus');

    // Request retained messages
    document.dispatchEvent(new CustomEvent('pan:subscribe', {
      bubbles: true,
      detail: {
        topics: ['app.config'],
        options: { retained: true } // Get retained message immediately
      }
    }));
  }
}

```

```

    // Handler receives retained message
    this.unsubscribe = bus.subscribe('app.config', (msg) => {
      this.applyConfig(msg.data);
    });
  }
}

```

21.1.7.3 Wildcard Subscriptions

```

// Subscribe to all sensor events
const bus = document.querySelector('pan-bus');

const unsub = bus.subscribe('sensor.*', (msg) => {
  console.log(`Sensor ${msg.topic}:`, msg.data);
});

// These all match
bus.publish('sensor.temperature', { value: 22.5 });
bus.publish('sensor.humidity', { value: 45 });
bus.publish('sensor.pressure', { value: 1013 });

```

21.1.7.4 Request-Response Pattern

```

// Requester
class DataFetcher extends HTMLElement {
  async fetchData(userId) {
    const requestId = crypto.randomUUID();
    const bus = document.querySelector('pan-bus');

    return new Promise((resolve) => {
      // Subscribe to response
      const unsub = bus.subscribe(`response.${requestId}`, (msg) => {
        unsub(); // Unsubscribe after first response
        resolve(msg.data);
      });

      // Publish request
      bus.publish('data.fetch',
        { userId },
        { requestId, responseChannel: `response.${requestId}` }
      );

      // Timeout after 5 seconds
      setTimeout(() => {
        unsub();
        resolve(null);
      }, 5000);
    });
  }
}

```

```

    }, 5000);
  });
}
}

// Responder
class DataProvider extends HTMLElement {
  connectedCallback() {
    const bus = document.querySelector('pan-bus');

    this.unsubscribe = bus.subscribe('data.fetch', async (msg) => {
      const data = await this.fetchUserData(msg.data.userId);

      // Publish response
      bus.publish(msg.responseChannel, data);
    });
  }
}

```

21.1.7.5 Monitoring Bus Health

```

// Get statistics
document.dispatchEvent(new CustomEvent('pan:sys.stats', {
  bubbles: true
}));

document.addEventListener('pan:deliver', (e) => {
  if (e.detail.topic === 'pan:sys.stats') {
    console.log('Bus stats:', e.detail.data);
    // {
    //   published: 1523,
    //   delivered: 3046,
    //   dropped: 5,
    //   subscriptions: 12,
    //   retained: 8,
    //   ...
    // }
  }
});

// Monitor errors
document.addEventListener('pan:sys.error', (e) => {
  console.error('Bus error:', e.detail.code, e.detail.message);
});

```

21.1.8 Common Issues and Solutions

Issue: Messages not being delivered

Check these common causes:

1. Bus not initialized: Wait for `pan:sys.ready` event
2. Subscription pattern doesn't match topic: Use `PanBusEnhanced.matches()` to test
3. Component disconnected: Subscribe in `connectedCallback()`, unsubscribe in `disconnectedCallback()`
4. Rate limit exceeded: Check console for errors, increase `rate-limit` attribute

```
// Wait for bus ready
document.addEventListener('pan:sys.ready', () => {
  // Now safe to subscribe/publish
});

// Or check programmatically
if (window.__panReady) {
  // Bus is ready
}
```

Issue: Memory leaks from subscriptions

Always unsubscribe in `disconnectedCallback()`:

```
class MyComponent extends HTMLElement {
  connectedCallback() {
    const bus = document.querySelector('pan-bus');
    this.unsubscribe = bus.subscribe('my.topic', this.handler);
  }

  disconnectedCallback() {
    if (this.unsubscribe) {
      this.unsubscribe();
      this.unsubscribe = null;
    }
  }
}
```

Issue: “Data must be JSON-serializable” error

Message payloads cannot contain functions, DOM nodes, or circular references:

```
// Bad
bus.publish('user.data', {
  element: document.querySelector('#foo'), // DOM node
  callback: () => {} // Function
});

// Good
bus.publish('user.data', {
  elementId: 'foo', // String reference
});
```

```
    shouldCallback: true // Boolean flag
  });
```

Issue: Rate limiting in production

Adjust `rate-limit` based on your application's needs:

```
<!-- For high-frequency updates (sensor data, etc.) -->
<pan-bus rate-limit="5000"></pan-bus>

<!-- For typical applications -->
<pan-bus rate-limit="1000"></pan-bus>
```

21.2 pan-theme-provider

21.2.1 Overview

`pan-theme-provider` manages application theme state and automatically responds to system light/dark mode preferences. It broadcasts theme changes via the PAN bus, enabling all components to update their appearance in a coordinated fashion.

The provider supports three theme modes: `light`, `dark`, and `auto`. In `auto` mode, it tracks system preferences and updates automatically when users change their OS theme settings.

21.2.2 When to Use

Use `pan-theme-provider` when:

- Your application supports light and dark themes
- You want to respect user system preferences
- You need coordinated theme switching across multiple components
- You're building a theme-aware component library

Don't use `pan-theme-provider` when:

- Your application has only one fixed theme
- You're implementing a custom theme system that goes beyond light/dark

21.2.3 Installation and Setup

Include the provider once per application, typically near the bus:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/core/pan-bus.mjs"></script>
  <script type="module" src="/ui/pan-theme-provider.mjs"></script>
</head>
<body>
```

```

<pan-bus></pan-bus>
<pan-theme-provider theme="auto"></pan-theme-provider>
<!-- Your application -->
</body>
</html>

```

The provider automatically:

1. Detects current system theme preference
2. Applies theme to `document.documentElement` via `data-theme` attribute
3. Sets `color-scheme` CSS property for native UI elements
4. Broadcasts `theme.changed` messages via PAN bus
5. Monitors system preference changes

21.2.4 Attributes

Attribute	Type	Default	Description
theme	String	"auto"	Theme mode: "light", "dark", or "auto". Auto mode follows system preferences.

Example:

```

<!-- Use system preference (recommended) -->
<pan-theme-provider theme="auto"></pan-theme-provider>

<!-- Force light theme -->
<pan-theme-provider theme="light"></pan-theme-provider>

<!-- Force dark theme -->
<pan-theme-provider theme="dark"></pan-theme-provider>

```

21.2.5 Methods

All methods are available on the provider element instance.

21.2.5.1 setTheme(theme)

Sets the theme mode.

Parameters: - theme (String, required): One of "light", "dark", or "auto"

Returns: undefined

Example:

```

const provider = document.querySelector('pan-theme-provider');

```

```
provider.setTheme('dark'); // Switch to dark theme
provider.setTheme('auto'); // Switch to auto mode
```

21.2.5.2 getTheme()

Gets the current theme mode (not the effective theme).

Returns: String - Current theme mode ("light", "dark", or "auto")

Example:

```
const provider = document.querySelector('pan-theme-provider');
console.log(provider.getTheme()); // "auto"
```

21.2.5.3 getEffectiveTheme()

Gets the actual theme being applied (resolves auto mode to light or dark).

Returns: String - Effective theme ("light" or "dark")

Example:

```
const provider = document.querySelector('pan-theme-provider');
console.log(provider.getEffectiveTheme()); // "dark" (if system is dark)
```

21.2.5.4 getSystemTheme()

Gets the current system theme preference.

Returns: String - System theme ("light" or "dark")

Example:

```
const provider = document.querySelector('pan-theme-provider');
console.log(provider.getSystemTheme()); // "dark"
```

21.2.6 Events

The provider dispatches these events:

21.2.6.1 theme-change (DOM event)

Dispatched whenever the theme changes.

Detail payload:

```
{
  theme: String,      // Current theme mode ("light", "dark", or "auto")
  effective: String   // Effective theme ("light" or "dark")
}
```

Example:

```
const provider = document.querySelector('pan-theme-provider');

provider.addEventListener('theme-change', (e) => {
  console.log('Theme changed:', e.detail.theme, '->', e.detail.effective);
});
```

21.2.6.2 theme.changed (PAN message)

Published via PAN bus when theme changes.

Message payload:

```
{
  theme: String,      // Current theme mode
  effective: String   // Effective theme
}
```

Example:

```
const bus = document.querySelector('pan-bus');

bus.subscribe('theme.changed', (msg) => {
  console.log('Theme changed to:', msg.data.effective);
  this.updateStyles(msg.data.effective);
});
```

21.2.6.3 theme.system-changed (PAN message)

Published when system theme preference changes.

Message payload:

```
{
  theme: String // New system theme ("light" or "dark")
}
```

21.2.7 Working Examples

21.2.7.1 Basic Theme Setup

```
<!DOCTYPE html>
<html>
<head>
  <style>
    /* Define theme variables */
    :root[data-theme="light"] {
      --bg: #ffffff;
      --text: #1e293b;
      --border: #e2e8f0;
    }
  </style>
</head>
```

```

:root[data-theme="dark"] {
  --bg: #1e293b;
  --text: #f1f5f9;
  --border: #334155;
}

body {
  background: var(--bg);
  color: var(--text);
  border-color: var(--border);
}
</style>
</head>
<body>
  <pan-bus></pan-bus>
  <pan-theme-provider theme="auto"></pan-theme-provider>

  <h1>Theme-aware content</h1>
</body>
</html>

```

21.2.7.2 Component Responding to Theme Changes

```

class ThemedCard extends HTMLElement {
  connectedCallback() {
    const bus = document.querySelector('pan-bus');

    this.unsubscribe = bus.subscribe('theme.changed', (msg) => {
      this.updateTheme(msg.data.effective);
    });

    // Get initial theme
    const provider = document.querySelector('pan-theme-provider');
    if (provider) {
      this.updateTheme(provider.getEffectiveTheme());
    }
  }

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

  updateTheme(theme) {
    this.className = `card theme-${theme}`;
    // Update component appearance
  }
}

```

```
customElements.define('themed-card', ThemedCard);
```

21.2.7.3 Programmatic Theme Control

```
// Toggle between light and dark
function toggleTheme() {
  const provider = document.querySelector('pan-theme-provider');
  const current = provider.getEffectiveTheme();
  provider.setTheme(current === 'light' ? 'dark' : 'light');
}

// Cycle through all modes
function cycleTheme() {
  const provider = document.querySelector('pan-theme-provider');
  const current = provider.getTheme();
  const cycle = { auto: 'light', light: 'dark', dark: 'auto' };
  provider.setTheme(cycle[current]);
}

// Reset to auto
function resetTheme() {
  const provider = document.querySelector('pan-theme-provider');
  provider.setTheme('auto');
}
```

21.2.7.4 Persisting Theme Preference

```
class ThemeManager extends HTMLElement {
  connectedCallback() {
    const bus = document.querySelector('pan-bus');
    const provider = document.querySelector('pan-theme-provider');

    // Load saved preference
    const saved = localStorage.getItem('theme-preference');
    if (saved && provider) {
      provider.setTheme(saved);
    }

    // Save when theme changes
    this.unsubscribe = bus.subscribe('theme.changed', (msg) => {
      localStorage.setItem('theme-preference', msg.data.theme);
    });
  }

  disconnectedCallback() {
    if (this.unsubscribe) this.unsubscribe();
  }
}
```

```

    }
  }

  customElements.define('theme-manager', ThemeManager);

```

21.2.8 Common Issues and Solutions

Issue: Theme not applying

Ensure CSS variables are defined for both themes:

```

/* Must define for both themes */
:root[data-theme="light"] {
  --color: #000;
}

:root[data-theme="dark"] {
  --color: #fff;
}

/* Then use in components */
.my-element {
  color: var(--color);
}

```

Issue: Flash of wrong theme on page load

Set theme before page renders to prevent flash:

```

<head>
  <!-- Inline script before any content -->
  <script>
    // Apply saved theme immediately
    const saved = localStorage.getItem('theme-preference');
    if (saved && saved !== 'auto') {
      document.documentElement.setAttribute('data-theme', saved);
    } else {
      // Detect system preference
      const dark = window.matchMedia('(prefers-color-scheme: dark)').matches;
      document.documentElement.setAttribute('data-theme', dark ? 'dark' : 'light');
    }
  </script>

  <!-- Then load components -->
  <script type="module" src="components.js"></script>
</head>

```

Issue: Components not updating when theme changes

Subscribe to `theme.changed` in `connectedCallback()`:

```
connectedCallback() {
  const bus = document.querySelector('pan-bus');
  this.unsubscribe = bus.subscribe('theme.changed', (msg) => {
    this.render(); // Re-render with new theme
  });
}
```

21.3 pan-theme-toggle

21.3.1 Overview

`pan-theme-toggle` is a UI component for switching themes. It displays the current theme and allows users to cycle through light, dark, and auto modes. The component integrates with `pan-theme-provider` via the PAN bus.

The toggle supports three visual variants: icon-only, button with label, and dropdown menu with all theme options.

21.3.2 When to Use

Use `pan-theme-toggle` when:

- You want to provide users control over theme preferences
- You need a quick, accessible way to switch themes
- You're building a settings panel or toolbar

Don't use `pan-theme-toggle` when:

- You want themes to be automatic only (use only `pan-theme-provider`)
- You're implementing custom theme controls with different UX

21.3.3 Installation and Setup

Include the toggle component in your UI:

```
<pan-bus></pan-bus>
<pan-theme-provider theme="auto"></pan-theme-provider>

<!-- Icon-only toggle (default) -->
<pan-theme-toggle></pan-theme-toggle>

<!-- Button with label -->
<pan-theme-toggle label="Theme"></pan-theme-toggle>

<!-- Dropdown menu -->
<pan-theme-toggle variant="dropdown"></pan-theme-toggle>
```

The toggle automatically:

1. Queries the theme provider for current theme

2. Subscribes to `theme.changed` messages
3. Updates its icon to reflect current theme
4. Communicates theme changes to the provider

21.3.4 Attributes

Attribute	Type	Default	Description
<code>label</code>	String	<code>""</code>	Optional text label to display next to icon. Only used with <code>button</code> variant.
<code>variant</code>	String	<code>"icon"</code>	Visual style: <code>"icon"</code> (icon only), <code>"button"</code> (icon + label), or <code>"dropdown"</code> (menu with all options).

Examples:

```
<!-- Icon only (minimal) -->
<pan-theme-toggle></pan-theme-toggle>

<!-- Button with label -->
<pan-theme-toggle variant="button" label="Theme"></pan-theme-toggle>

<!-- Dropdown menu -->
<pan-theme-toggle variant="dropdown"></pan-theme-toggle>
```

21.3.5 Methods

The toggle component has no public methods. All interaction happens through the UI or via the theme provider.

21.3.6 Events

The toggle doesn't emit custom events. Theme changes are communicated through `pan-theme-provider`.

21.3.7 Working Examples

21.3.7.1 Navigation Bar with Theme Toggle

```
<nav class="main-nav">
  <div class="nav-brand">
    <h1>My App</h1>
  </div>

  <div class="nav-actions">
    <button>Settings</button>
    <pan-theme-toggle variant="icon"></pan-theme-toggle>
  </div>
</nav>
```

```

    </div>
  </nav>

  <style>
    .main-nav {
      display: flex;
      justify-content: space-between;
      align-items: center;
      padding: 1rem 2rem;
      background: var(--color-surface);
      border-bottom: 1px solid var(--color-border);
    }

    .nav-actions {
      display: flex;
      gap: 1rem;
      align-items: center;
    }
  </style>

```

21.3.7.2 Settings Panel with Dropdown

```

<div class="settings-panel">
  <h2>Preferences</h2>

  <div class="setting-row">
    <label>Theme</label>
    <pan-theme-toggle variant="dropdown"></pan-theme-toggle>
  </div>

  <div class="setting-row">
    <label>Language</label>
    <select>
      <option>English</option>
      <option>Espanol</option>
    </select>
  </div>
</div>

<style>
  .setting-row {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 1rem 0;
    border-bottom: 1px solid var(--color-border);
  }

```

```
</style>
```

21.3.7.3 Responsive Theme Toggle

```
<!-- Show dropdown on mobile, icon on desktop -->
<style>
  .theme-toggle-mobile {
    display: block;
  }

  .theme-toggle-desktop {
    display: none;
  }

  @media (min-width: 768px) {
    .theme-toggle-mobile {
      display: none;
    }

    .theme-toggle-desktop {
      display: block;
    }
  }
</style>

<div class="theme-toggle-mobile">
  <pan-theme-toggle variant="dropdown"></pan-theme-toggle>
</div>

<div class="theme-toggle-desktop">
  <pan-theme-toggle variant="icon"></pan-theme-toggle>
</div>
```

21.3.7.4 Custom Styled Toggle

```
<pan-theme-toggle variant="button" label="Appearance"></pan-theme-toggle>

<style>
  pan-theme-toggle {
    /* Override CSS custom properties */
    --color-surface: #f8fafc;
    --color-border: #cbd5e1;
    --color-text: #1e293b;
    --font-sans: 'Inter', system-ui, sans-serif;
  }

  pan-theme-toggle::part(button) {
```

```

    /* Style shadow DOM parts if exposed */
    border-radius: 0.75rem;
    padding: 0.75rem 1.5rem;
  }
</style>

```

21.3.8 Common Issues and Solutions

Issue: Toggle not working

Ensure `pan-theme-provider` is present:

```

<!-- Provider must exist -->
<pan-theme-provider theme="auto"></pan-theme-provider>

<!-- Then toggle will work -->
<pan-theme-toggle></pan-theme-toggle>

```

Issue: Toggle shows wrong icon

The toggle subscribes to theme changes on connect. If added dynamically, wait for PAN bus:

```

// Wait for bus ready before adding toggle
document.addEventListener('pan:sys.ready', () => {
  const toggle = document.createElement('pan-theme-toggle');
  document.body.appendChild(toggle);
});

```

Issue: Dropdown menu positioning

The dropdown uses `position: absolute` and may need container constraints:

```

<div style="position: relative;">
  <pan-theme-toggle variant="dropdown"></pan-theme-toggle>
</div>

```

21.4 pan-routes

21.4.1 Overview

`pan-routes` provides runtime-configurable message routing for the PAN bus. It enables declarative routing rules that match messages based on topic, content, or metadata, then perform actions like emitting new messages, forwarding to different topics, logging, or calling handler functions.

Routes are defined programmatically and can be enabled, disabled, or updated at runtime. This makes `pan-routes` ideal for complex message flows, cross-cutting concerns (logging, monitoring), and workflow orchestration.

Note that `pan-routes` is not a URL router (for that, see Chapter 9). It's a message router for the PAN bus.

21.4.2 When to Use

Use **pan-routes** when:

- You need to transform or redirect messages based on content
- You're implementing cross-cutting concerns (logging, analytics, monitoring)
- You want to decouple message producers from consumers
- You're building workflow automation or state machines
- You need conditional message routing based on complex predicates

Don't use **pan-routes** when:

- Simple direct pub-sub suffices for your needs
- You need URL/navigation routing (use client-side router instead)
- The added complexity isn't justified by your use case

21.4.3 Installation and Setup

Enable routing in the bus configuration:

```
<pan-bus enable-routing="true"></pan-bus>
```

Once enabled, access the routing manager via the global API:

```
const routes = window.pan.routes;

// Or from the bus element
const bus = document.querySelector('pan-bus');
const routes = bus.routingManager;
```

21.4.4 Methods

All methods are available on the **PanRoutesManager** instance.

21.4.4.1 add(route)

Adds a new route to the routing system.

Parameters: - route (Object, required): Route configuration object

Route configuration:

```
{
  id: String,           // Optional: unique ID (generated if omitted)
  name: String,         // Required: human-readable name
  enabled: Boolean,     // Optional: whether route is active (default: true)
  order: Number,       // Optional: execution order (default: 0)
  match: {              // Required: matching criteria
    type: String|Array, // Match message type
    topic: String|Array, // Match message topic
    source: String|Array, // Match message source
    tagsAny: Array,      // Match any of these tags
    tagsAll: Array,     // Match all of these tags
  }
}
```

```

    where: Object           // Predicate for complex matching
  },
  transform: Object,       // Optional: transform matched message
  actions: Array,          // Required: actions to perform
  meta: {                  // Optional: metadata
    createdBy: String,
    tags: Array
  }
}

```

Returns: Object - The created route with generated ID

Example:

```

const routes = window.pan.routes;

// Simple route
routes.add({
  name: 'Login -> Dashboard',
  match: { type: 'user.login.success' },
  actions: [
    { type: 'EMIT', message: { topic: 'ui.navigate', data: { to: '/dashboard' } } }
  ]
});

// Complex route with filtering
routes.add({
  name: 'High temp alert',
  match: {
    type: 'sensor.update',
    where: {
      op: 'gt',
      path: 'payload.temperature',
      value: 30
    }
  },
  actions: [
    {
      type: 'EMIT',
      message: { topic: 'alert.high-temp' },
      inherit: ['payload', 'meta']
    },
    {
      type: 'LOG',
      level: 'warn',
      template: 'High temp: {{payload.temperature}} degreesC'
    }
  ]
}

```

```
});
```

21.4.4.2 update(id, patch)

Updates an existing route.

Parameters: - id (String, required): Route ID - patch (Object, required): Fields to update

Returns: Object - Updated route

Example:

```
routes.update('route-123', {  
  enabled: false,      // Disable route  
  order: 10            // Change execution order  
});
```

21.4.4.3 remove(id)

Removes a route.

Parameters: - id (String, required): Route ID

Returns: Boolean - True if route existed and was removed

21.4.4.4 enable(id) / disable(id)

Enables or disables a route without removing it.

Parameters: - id (String, required): Route ID

Example:

```
routes.disable('route-123'); // Temporarily disable  
// ... later ...  
routes.enable('route-123');  // Re-enable
```

21.4.4.5 get(id)

Retrieves a route by ID.

Parameters: - id (String, required): Route ID

Returns: Object|undefined - The route or undefined if not found

21.4.4.6 list(filter)

Lists all routes, optionally filtered.

Parameters: - filter (Object, optional): Filter criteria - enabled (Boolean): Only return enabled/disabled routes

Returns: Array<Object> - Array of routes sorted by order

Example:

```
// Get all routes
const allRoutes = routes.list();

// Get only enabled routes
const activeRoutes = routes.list({ enabled: true });
```

21.4.4.7 clear()

Removes all routes.

21.4.4.8 registerTransformFn(fnId, fn)

Registers a transform function for use in routes.

Parameters: - `fnId` (String, required): Unique function identifier - `fn` (Function, required): Transform function

Example:

```
// Register transform
routes.registerTransformFn('toUpperCase', (value) => {
  return typeof value === 'string' ? value.toUpperCase() : value;
});

// Use in route
routes.add({
  name: 'Uppercase messages',
  match: { type: 'message.send' },
  transform: {
    op: 'map',
    path: 'payload.text',
    fnId: 'toUpperCase'
  },
  actions: [
    { type: 'FORWARD', topic: 'message.send.processed' }
  ]
});
```

21.4.4.9 registerHandler(handlerId, fn)

Registers a handler function for CALL actions.

Parameters: - `handlerId` (String, required): Unique handler identifier - `fn` (Function, required): Handler function receiving message

Example:

```
// Register handler
routes.registerHandler('logToServer', async (message) => {
  await fetch('/api/logs', {
    method: 'POST',
```

```

    body: JSON.stringify(message)
  });
});

// Use in route
routes.add({
  name: 'Log errors to server',
  match: { type: 'error.*' },
  actions: [
    { type: 'CALL', handlerId: 'logToServer' }
  ]
});

```

21.4.4.10 getStats()

Returns routing statistics.

Returns: Object - Statistics object

```

{
  routesEvaluated: Number,    // Total routes evaluated
  routesMatched: Number,      // Total routes matched
  actionsExecuted: Number,     // Total actions executed
  errors: Number,             // Total errors
  routeCount: Number,         // Current route count
  enabledRouteCount: Number,   // Enabled route count
  transformFnCount: Number,    // Registered transforms
  handlerCount: Number        // Registered handlers
}

```

21.4.4.11 resetStats()

Resets all statistics to zero.

21.4.4.12 setEnabled(enabled)

Enables or disables the entire routing system.

Parameters: - enabled (Boolean, required): Whether routing should be active

21.4.4.13 onRoutesChanged(listener)

Subscribes to route configuration changes.

Parameters: - listener (Function, required): Callback receiving updated route list

Returns: Function - Unsubscribe function

Example:

```

const unsubscribe = routes.onRoutesChanged((routeList) => {
  console.log('Routes updated:', routeList.length);
});

```

```
});  
  
// Later  
unsubscribe();
```

21.4.4.14 onError(listener)

Subscribes to routing errors.

Parameters: - listener (Function, required): Callback receiving error details

Returns: Function - Unsubscribe function

21.4.5 Route Configuration

21.4.5.1 Match Criteria

Match by type:

```
match: {  
  type: 'user.login'           // Single type  
  // OR  
  type: ['user.login', 'user.register'] // Multiple types  
}
```

Match by topic:

```
match: {  
  topic: 'sensor.temp'  
  // OR  
  topic: ['sensor.temp', 'sensor.humidity']  
}
```

Match by source:

```
match: {  
  source: 'dashboard-widget'  
  // OR  
  source: ['widget-1', 'widget-2']  
}
```

Match by tags:

```
match: {  
  tagsAny: ['urgent', 'high-priority'], // Has any of these tags  
  tagsAll: ['verified', 'logged']      // Has all of these tags  
}
```

Match with predicates:

Predicates support: eq, neq, gt, gte, lt, lte, in, regex, and, or, not

```

// Greater than
match: {
  where: {
    op: 'gt',
    path: 'payload.value',
    value: 100
  }
}

// Regular expression
match: {
  where: {
    op: 'regex',
    path: 'payload.email',
    value: '^[\\w-]+@[\\w-]+\\. [a-z]{2,}$'
  }
}

// Combined predicates
match: {
  where: {
    op: 'and',
    children: [
      { op: 'eq', path: 'payload.status', value: 'active' },
      { op: 'gt', path: 'payload.score', value: 75 }
    ]
  }
}

```

21.4.5.2 Transform Operations

Identity (no transformation):

```
transform: { op: 'identity' }
```

Pick fields:

```

transform: {
  op: 'pick',
  paths: ['payload.userId', 'payload.email', 'meta.timestamp']
}

```

Map with function:

```

// First register function
routes.registerTransformFn('double', (x) => x * 2);

// Then use in route
transform: {

```

```

    op: 'map',
    path: 'payload.value',
    fnId: 'double'
  }

```

Custom transformation:

```

// Register custom transform
routes.registerTransformFn('summarize', (message) => {
  return {
    ...message,
    payload: {
      summary: `${message.type}: ${message.payload.count} items`
    }
  };
});

// Use in route
transform: {
  op: 'custom',
  fnId: 'summarize'
}

```

21.4.5.3 Actions

EMIT action - Publishes a new message:

```

{
  type: 'EMIT',
  message: {
    topic: 'new.topic',
    data: { /* payload */ }
  },
  inherit: ['payload', 'meta'] // Inherit fields from original message
}

```

FORWARD action - Forwards message to different topic:

```

{
  type: 'FORWARD',
  topic: 'new.topic',           // Required
  typeOverride: 'new.type'     // Optional
}

```

LOG action - Logs message:

```

{
  type: 'LOG',
  level: 'info',                // 'log', 'info', 'warn', 'error'
  template: 'User {{payload.userId}} logged in at {{meta.timestamp}}'
}

```

```
}
```

CALL action - Calls registered handler:

```
{  
  type: 'CALL',  
  handlerId: 'my-handler'  
}
```

21.4.6 Working Examples

21.4.6.1 Workflow Orchestration

```
const routes = window.pan.routes;  
  
// Step 1: User registers -> validate email  
routes.add({  
  name: 'Registration -> Email validation',  
  match: { type: 'user.register' },  
  actions: [  
    {  
      type: 'EMIT',  
      message: { topic: 'email.validate' },  
      inherit: ['payload']  
    }  
  ]  
});  
  
// Step 2: Email validated -> send welcome message  
routes.add({  
  name: 'Email validated -> Welcome',  
  match: { type: 'email.validated' },  
  actions: [  
    {  
      type: 'EMIT',  
      message: { topic: 'email.send', data: { template: 'welcome' } },  
      inherit: ['payload']  
    }  
  ]  
});  
  
// Step 3: All done -> show dashboard  
routes.add({  
  name: 'Welcome sent -> Dashboard',  
  match: { type: 'email.sent', where: { op: 'eq', path: 'payload.template', value: 'welcome' } },  
  actions: [  
    {  
      type: 'EMIT',
```

```

        message: { topic: 'ui.navigate', data: { to: '/dashboard' } }
    }
}
});

```

21.4.6.2 Cross-cutting Logging

```

// Log all error messages
routes.add({
  name: 'Error logger',
  match: { topic: 'error.*' },
  actions: [
    {
      type: 'LOG',
      level: 'error',
      template: '{{topic}} {{payload.message}}'
    }
  ]
});

// Log high-value transactions
routes.add({
  name: 'High-value transaction logger',
  match: {
    type: 'transaction.complete',
    where: { op: 'gt', path: 'payload.amount', value: 1000 }
  },
  actions: [
    {
      type: 'LOG',
      level: 'info',
      template: 'High-value transaction: ${payload.amount}'
    },
    {
      type: 'CALL',
      handlerId: 'notifyFinance'
    }
  ]
});

```

21.4.6.3 Message Filtering and Transformation

```

// Filter and forward sensor data
routes.add({
  name: 'Filter valid sensor readings',
  match: {

```

```

    type: 'sensor.reading',
    where: {
      op: 'and',
      children: [
        { op: 'gte', path: 'payload.temperature', value: -40 },
        { op: 'lte', path: 'payload.temperature', value: 85 }
      ]
    },
    transform: {
      op: 'pick',
      paths: ['payload.temperature', 'payload.humidity', 'meta.sensorId']
    },
    actions: [
      { type: 'FORWARD', topic: 'sensor.valid' }
    ]
  });

```

21.4.6.4 Analytics and Monitoring

```

// Count messages by type
const messageCounts = new Map();

routes.registerHandler('countMessages', (msg) => {
  const count = messageCounts.get(msg.type) || 0;
  messageCounts.set(msg.type, count + 1);
});

routes.add({
  name: 'Message counter',
  match: { topic: '*' }, // Match all messages
  actions: [
    { type: 'CALL', handlerId: 'countMessages' }
  ]
});

// Report stats periodically
setInterval(() => {
  console.log('Message counts:', Object.fromEntries(messageCounts));
}, 60000);

```

21.4.7 Common Issues and Solutions

Issue: Routes not firing

Ensure routing is enabled:

```
<pan-bus enable-routing="true"></pan-bus>
```

Check route is enabled:

```
const route = routes.get('route-id');
console.log('Enabled:', route.enabled);
```

Issue: Route matching wrong messages

Test match criteria:

```
// Debug what routes match
const bus = document.querySelector('pan-bus');
bus.setAttribute('debug', 'true');

// Or use route stats
console.log(routes.getStats());
```

Issue: Transform function not found

Register before adding routes:

```
// Register first
routes.registerTransformFn('myTransform', (msg) => msg);

// Then use
routes.add({
  name: 'My route',
  transform: { op: 'custom', fnId: 'myTransform' },
  actions: [...]
});
```

Issue: Routes executing in wrong order

Set explicit order:

```
routes.add({
  name: 'First route',
  order: 0, // Executes first
  // ...
});

routes.add({
  name: 'Second route',
  order: 10, // Executes after order 0
  // ...
});
```

21.5 Summary

This chapter provided comprehensive API reference documentation for LARC's four core components:

- **pan-bus**: The foundational message bus enabling decoupled component communication with memory management, rate limiting, and routing
- **pan-theme-provider**: Centralized theme management that respects system preferences and coordinates theme changes across the application
- **pan-theme-toggle**: User-facing theme switching controls with multiple visual variants
- **pan-routes**: Runtime-configurable message routing for complex workflows and cross-cutting concerns

These components form the backbone of every LARC application. The bus provides communication infrastructure, the theme components enable appearance customization, and routes add sophisticated message routing capabilities.

As you build with LARC, you'll reference this chapter frequently for attribute names, method signatures, event payloads, and troubleshooting guidance. The examples demonstrate real-world patterns you can adapt to your specific needs.

In the next chapter, we'll explore advanced component patterns that build on these foundations, showing you how to create sophisticated, composable components that leverage the full power of the LARC architecture.

Chapter 22

Data Components

In which we explore state management and persistent storage without losing track of what's true

Data is the lifeblood of any application, but managing that data—keeping it consistent, synchronized, and available—is where complexity breeds. An application without proper data management is like a library where books randomly teleport between shelves. Eventually, nobody trusts anything they find.

This chapter covers LARC's data components: tools designed to manage state and persistent storage in ways that feel predictable and maintainable. We'll explore **pan-store**, a reactive state management solution built on JavaScript Proxies and EventTarget, and **pan-idb**, a component that bridges IndexedDB with LARC's message bus. By the end, you'll understand how to build applications that handle data with discipline and grace.

22.1 Overview

LARC provides two core components for data management:

- **pan-store**: Reactive state management for in-memory application state
- **pan-idb**: IndexedDB integration for persistent client-side storage

These components operate independently but complement each other. Use **pan-store** for reactive application state that needs to be synchronized across components. Use **pan-idb** when you need data to persist across sessions or when working with large datasets that exceed reasonable memory limits.

Both components communicate via the PAN bus, making them first-class participants in LARC's message-based architecture. State changes become messages. Database operations become requests. Everything flows through topics, maintaining the architectural consistency that makes LARC applications comprehensible.

22.2 pan-store: Reactive State Management

22.2.1 Purpose

pan-store provides reactive state management using JavaScript Proxies and the EventTarget API. It's designed for shared application state that needs to be observed by multiple components without tight coupling.

Think of it as a specialized key-value store that automatically notifies subscribers when values change. Set a property, and any component listening for that change receives a message. No manual event dispatching, no brittle observer patterns, just reactive updates that work.

22.2.2 When to Use

Use **pan-store** when you need:

- **Shared state across components:** User preferences, authentication status, shopping cart contents
- **Reactive updates:** Components that need to re-render when specific values change
- **Middleware hooks:** Logging, validation, or side effects on state changes
- **Derived values:** Computed properties that depend on other state
- **Undo/redo functionality:** State snapshots make time-travel debugging possible

22.2.3 When Not to Use

Avoid **pan-store** for:

- **Local component state:** Use plain JavaScript properties instead
- **Large datasets:** IndexedDB or OPFS are better suited for bulk data
- **Transient UI state:** Dropdown open/closed, hover states, animation frames
- **High-frequency updates:** Thousands of changes per second may cause performance issues

22.2.4 Installation

```
import { createStore, bind } from './pan-store.mjs';
```

The module exports two functions:

- **createStore(initial):** Creates a new reactive store
- **bind(element, store, mapping, options):** Binds form inputs to store properties

22.2.5 API Reference

22.2.5.1 createStore(initial)

Creates a reactive store with optional initial state.

Parameters: - **initial** (Object, optional): Initial state object. Defaults to {}

Returns: Store instance with the following methods and properties

Example:

```
const store = createStore({
  count: 0,
  user: { name: 'Ada', role: 'admin' }
});
```

22.2.5.2 Store Properties

state (Proxy)

The reactive state object. Access and modify properties directly:

```
store.state.count = 5;
console.log(store.state.count); // 5
```

Any assignment triggers change events and notifies subscribers.

22.2.5.3 Store Methods

subscribe(callback)

Subscribes to state changes.

Parameters: - **callback** (Function): Called when state changes. Receives event object with **detail** containing:

- **key** (String): Changed property name
- **value** (Any): New value
- **oldValue** (Any): Previous value
- **state** (Proxy): Current state object

Returns: Unsubscribe function

Example:

```
const unsub = store.subscribe(({ detail }) => {
  console.log(`${detail.key} changed from ${detail.oldValue} to ${detail.value}`);
});

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

set(key, value)

Sets a single property.

Parameters: - **key** (String): Property name - **value** (Any): New value

Example:

```
store.set('theme', 'dark');
```

patch(object)

Merges multiple properties at once.

Parameters: - **object** (Object): Properties to merge

Example:

```
store.patch({
  theme: 'dark',
  fontSize: 16
});
```

update(fn)

Updates state using a function.

Parameters: - **fn** (Function): Receives current state snapshot, returns new state (or mutates and returns undefined)

Example:

```
store.update(state => {
  state.count += 1;
  return state;
});
```

select(path)

Retrieves nested value by dot-notation path.

Parameters: - **path** (String): Dot-separated property path

Returns: Value at path, or undefined if not found

Example:

```
store.state.user = { profile: { name: 'Ada' } };
const name = store.select('user.profile.name'); // 'Ada'
```

derive(key, deps, computeFn)

Creates a computed/derived value.

Parameters: - **key** (String): Name for derived property - **deps** (Array|Function): Dependency property names, or compute function if omitted - **computeFn** (Function): Computation function receiving dependency values

Returns: Unsubscribe function

Example:

```
store.state.firstName = 'Ada';
store.state.lastName = 'Lovelace';

store.derive('fullName', ['firstName', 'lastName'], (first, last) => {
  return `${first} ${last}`;
});

console.log(store.state.fullName); // 'Ada Lovelace'
```

batch(fn)

Batches multiple updates into single change event.

Parameters: - `fn` (Function): Receives object with `set(key, value)` method and `state` proxy

Example:

```
store.batch(({ set }) => {  
  set('loading', true);  
  set('error', null);  
  set('data', null);  
});
```

use(middleware)

Adds middleware function called on every state change.

Parameters: - `middleware` (Function): Receives object with `key`, `value`, `oldValue`, `state`

Returns: Unsubscribe function

Example:

```
const unuse = store.use(({ key, value }) => {  
  console.log(`[Middleware] ${key} = ${value}`);  
});
```

snapshot()

Creates deep clone of current state.

Returns: Plain object with current state

Example:

```
const current = store.snapshot();  
console.log(current); // { count: 5, theme: 'dark' }
```

reset()

Resets state to initial values.

Example:

```
store.reset();
```

has(key)

Checks if property exists (including derived properties).

Parameters: - `key` (String): Property name

Returns: Boolean

Example:

```
store.has('count'); // true  
store.has('nonexistent'); // false
```

delete(key)

Removes property from state.

Parameters: - `key` (String): Property name

Returns: Boolean (true if deleted, false if didn't exist)

Example:

```
store.delete('temporaryFlag');
```

`keys()`

Returns all property names, including derived properties.

Returns: Array of strings

Example:

```
const allKeys = store.keys(); // ['count', 'theme', 'fullName']
```

22.2.5.4 Store Events

`state`

Emitted when state changes.

Event Detail: - `key` (String): Changed property name - `value` (Any): New value - `oldValue` (Any): Previous value - `state` (Proxy): Current state - `batch` (Boolean, optional): True if part of batch update - `changes` (Array, optional): Array of changes in batch - `deleted` (Boolean, optional): True if property was deleted

`derived`

Emitted when derived value updates.

Event Detail: - `key` (String): Derived property name - `value` (Any): New computed value - `state` (Proxy): Current state

22.2.6 `bind(element, store, mapping, options)`

Binds form inputs to store properties, creating two-way data binding.

Parameters: - `element` (HTMLElement): Container element - `store` (Store): Store instance - `mapping` (Object): Map of CSS selectors to property names - `options` (Object, optional):

- `events` (Array): Events to listen for (default: ['input', 'change'])

Returns: Unbind function

Example:

```
const store = createStore({ username: '', email: '' });

const form = document.querySelector('#user-form');
const unbind = bind(form, store, {
  'input[name="username"]': 'username',
  'input[name="email"]': 'email'
});
```

```
});  
  
// Input changes update store  
// Store changes update inputs
```

22.2.7 Complete Working Examples

22.2.7.1 Basic Counter

```
<!DOCTYPE html>  
<html>  
<head>  
  <title>Counter with pan-store</title>  
</head>  
<body>  
  <div id="app">  
    <h1>Count: <span id="count">0</span></h1>  
    <button id="increment">+</button>  
    <button id="decrement">-</button>  
    <button id="reset">Reset</button>  
  </div>  
  
  <script type="module">  
    import { createStore } from './pan-store.mjs';  
  
    const store = createStore({ count: 0 });  
  
    // Subscribe to changes  
    store.subscribe(({ detail }) => {  
      if (detail.key === 'count') {  
        document.getElementById('count').textContent = detail.value;  
      }  
    });  
  
    // Bind buttons  
    document.getElementById('increment').addEventListener('click', () => {  
      store.state.count++;  
    });  
  
    document.getElementById('decrement').addEventListener('click', () => {  
      store.state.count--;  
    });  
  
    document.getElementById('reset').addEventListener('click', () => {  
      store.reset();  
    });  
  </script>
```

```
</body>
</html>
```

22.2.7.2 Form Binding

```
<!DOCTYPE html>
<html>
<head>
  <title>Form Binding</title>
</head>
<body>
  <form id="settings">
    <label>
      Theme:
      <select name="theme">
        <option value="light">Light</option>
        <option value="dark">Dark</option>
      </select>
    </label>

    <label>
      <input type="checkbox" name="notifications">
      Enable notifications
    </label>

    <label>
      Font size:
      <input type="range" name="fontSize" min="12" max="24">
    </label>
  </form>

  <pre id="output"></pre>

  <script type="module">
    import { createStore, bind } from './pan-store.mjs';

    const store = createStore({
      theme: 'light',
      notifications: false,
      fontSize: 16
    });

    // Bind form inputs
    const form = document.getElementById('settings');
    bind(form, store, {
      'select[name="theme"]': 'theme',
      'input[name="notifications"]': 'notifications',
```

```
    'input[name="fontSize"]': 'fontSize'
  });

  // Display current state
  const output = document.getElementById('output');
  store.subscribe(() => {
    output.textContent = JSON.stringify(store.snapshot(), null, 2);
  });

  // Initial render
  output.textContent = JSON.stringify(store.snapshot(), null, 2);
</script>
</body>
</html>
```

22.2.7.3 Derived Values and Middleware

```
import { createStore } from './pan-store.mjs';

// Create store with cart items
const store = createStore({
  items: [
    { id: 1, name: 'Widget', price: 10, quantity: 2 },
    { id: 2, name: 'Gadget', price: 25, quantity: 1 }
  ],
  taxRate: 0.08
});

// Derive subtotal
store.derive('subtotal', ['items'], (items) => {
  return items.reduce((sum, item) => sum + (item.price * item.quantity), 0);
});

// Derive tax
store.derive('tax', ['subtotal', 'taxRate'], (subtotal, rate) => {
  return subtotal * rate;
});

// Derive total
store.derive('total', ['subtotal', 'tax'], (subtotal, tax) => {
  return subtotal + tax;
});

// Add logging middleware
store.use(({ key, value, oldValue }) => {
  console.log(`State changed: ${key}`, { oldValue, newValue: value });
});
```

```

// Add validation middleware
store.use(({ key, value }) => {
  if (key === 'taxRate' && (value < 0 || value > 1)) {
    console.error('Invalid tax rate:', value);
  }
});

// Subscribe to total changes
store.subscribe(({ detail }) => {
  if (detail.key === 'total') {
    console.log(`Cart total: ${detail.value.toFixed(2)}`);
  }
});

// Access computed values
console.log(store.state.subtotal); // 45
console.log(store.state.tax); // 3.6
console.log(store.state.total); // 48.6

```

22.2.7.4 Time-Travel Debugging

```

import { createStore } from './pan-store.mjs';

const store = createStore({ position: { x: 0, y: 0 } });

// History tracking
const history = [store.snapshot()];
let historyIndex = 0;

store.subscribe(() => {
  // Save snapshot after each change
  const snapshot = store.snapshot();
  history.splice(historyIndex + 1);
  history.push(snapshot);
  historyIndex = history.length - 1;
});

function undo() {
  if (historyIndex > 0) {
    historyIndex--;
    store.patch(history[historyIndex]);
  }
}

function redo() {
  if (historyIndex < history.length - 1) {

```

```

    historyIndex++;
    store.patch(history[historyIndex]);
  }
}

// Make changes
store.state.position = { x: 10, y: 20 };
store.state.position = { x: 15, y: 25 };

console.log(store.state.position); // { x: 15, y: 25 }

undo();
console.log(store.state.position); // { x: 10, y: 20 }

undo();
console.log(store.state.position); // { x: 0, y: 0 }

redo();
console.log(store.state.position); // { x: 10, y: 20 }

```

22.2.8 Common Issues and Solutions

Issue: Nested object changes not detected

```

// Problem: Direct mutation doesn't trigger updates
store.state.user.name = 'Ada'; // No event fired

// Solution: Reassign the parent object
store.state.user = { ...store.state.user, name: 'Ada' };

// Or use update()
store.update(state => {
  state.user.name = 'Ada';
  return state;
});

```

Issue: Circular references causing errors

```

// Problem: snapshot() fails with circular structures
const store = createStore({});
store.state.self = store.state; // Circular reference

// Solution: Avoid circular references, or use custom serialization
store.use(({ key, value }) => {
  // Custom handling for specific keys
  if (key === 'self') {
    return; // Skip serialization
  }
});

```

Issue: Performance with frequent updates

```
// Problem: Hundreds of updates firing individual events
for (let i = 0; i < 1000; i++) {
  store.state.count = i; // 1000 events
}

// Solution: Use batch()
store.batch(({ set }) => {
  for (let i = 0; i < 1000; i++) {
    set('count', i);
  }
}); // Single event
```

Issue: Memory leaks from uncanceled subscriptions

```
// Problem: Subscriptions outlive components
class MyComponent extends HTMLElement {
  connectedCallback() {
    store.subscribe(this.handleChange); // Never unsubscribed
  }
}

// Solution: Store unsub function and call in disconnectedCallback
class MyComponent extends HTMLElement {
  connectedCallback() {
    this.unsub = store.subscribe(this.handleChange);
  }

  disconnectedCallback() {
    if (this.unsub) this.unsub();
  }
}
```

22.3 pan-idb: IndexedDB Integration

22.3.1 Purpose

pan-idb provides a declarative interface to IndexedDB through LARC's message bus. It handles database initialization, schema upgrades, and CRUD operations via PAN topics, abstracting away IndexedDB's verbose API.

Think of it as a database component that speaks the language of your application. Instead of managing transactions, cursors, and error handlers manually, you publish messages and receive results.

22.3.2 When to Use

Use `pan-idb` when you need:

- **Persistent client-side storage:** Data that survives page reloads and browser restarts
- **Offline-first applications:** Local storage for sync later
- **Large datasets:** Gigabytes of data that won't fit in memory
- **Structured queries:** Indexed lookups by multiple fields
- **File-like data:** Blobs, images, or binary data

22.3.3 When Not to Use

Avoid `pan-idb` for:

- **Simple key-value storage:** Use `localStorage` or `sessionStorage`
- **Transient state:** Use `pan-store` for in-memory state
- **Small data:** Overhead isn't worth it for tiny datasets
- **Server-authoritative data:** If server is source of truth, cache in memory instead

22.3.4 Installation

`pan-idb` is a custom element. Include it in your HTML or create it programmatically:

```
<pan-idb
  database="myapp"
  store="documents"
  key-path="id"
  auto-increment
  indexes='[{"name":"byTitle","keyPath":"title"}, {"name":"byDate","keyPath":"created"}]' '>
</pan-idb>
```

22.3.5 Attributes Reference

database (required)

Database name.

Type: String **Default:** None **Example:** `database="myapp"`

version

Database version number. Increment to trigger schema upgrade.

Type: Number **Default:** 1 **Example:** `version="2"`

store (required)

Object store name (similar to table name).

Type: String **Default:** None **Example:** `store="documents"`

key-path

Property name to use as primary key.

Type: String **Default:** "id" **Example:** `key-path="documentId"`

auto-increment

Use auto-incrementing keys. Presence of attribute enables it.

Type: Boolean **Default:** false **Example:** auto-increment (no value needed)

indexes

JSON array of index configurations.

Type: JSON String **Default:** [] **Format:**

```
[
  {
    "name": "byTitle",
    "keyPath": "title",
    "unique": false,
    "multiEntry": false
  }
]
```

22.3.6 PAN Topics

All topics follow the pattern `{store}.idb.{operation}`. For a store named `documents`, topics are:

22.3.6.1 Subscribe Topics (Commands)**`{store}.idb.get`**

Retrieve item by key.

Message Data: - key (Any): Item key

Response: `{store}.idb.result`

Example:

```
pc.publish({
  topic: 'documents.idb.get',
  data: { key: 123 }
});
```

`{store}.idb.put`

Insert or update item.

Message Data: - item (Object): Item to store

Response: `{store}.idb.result`

Example:

```
pc.publish({
  topic: 'documents.idb.put',
  data: {
    item: { id: 123, title: 'Report', content: '...' }
  }
});
```

```
}  
});
```

{store}.idb.add

Insert item (fails if key exists).

Message Data: - item (Object): Item to add

Response: {store}.idb.result

Example:

```
pc.publish({  
  topic: 'documents.idb.add',  
  data: {  
    item: { id: 456, title: 'New Doc' }  
  }  
});
```

{store}.idb.delete

Delete item by key.

Message Data: - key (Any): Item key

Response: {store}.idb.result

Example:

```
pc.publish({  
  topic: 'documents.idb.delete',  
  data: { key: 123 }  
});
```

{store}.idb.clear

Delete all items.

Message Data: Empty object {}

Response: {store}.idb.result

Example:

```
pc.publish({  
  topic: 'documents.idb.clear',  
  data: {}  
});
```

{store}.idb.list

List items with optional filtering.

Message Data: - index (String, optional): Index name to use - range (IDBKeyRange, optional): Key range for filtering - direction (String, optional): 'next', 'prev', 'nextunique', 'prevunique' - limit (Number, optional): Maximum results

Response: {store}.idb.result

Example:

```
pc.publish({
  topic: 'documents.idb.list',
  data: {
    index: 'byDate',
    direction: 'prev',
    limit: 10
  }
});
```

{store}.idb.query

Query by index.

Message Data: - index (String): Index name - value (Any): Value to match

Response: {store}.idb.result

Example:

```
pc.publish({
  topic: 'documents.idb.query',
  data: {
    index: 'byTitle',
    value: 'Report'
  }
});
```

{store}.idb.count

Count items.

Message Data: - index (String, optional): Index name

Response: {store}.idb.result

Example:

```
pc.publish({
  topic: 'documents.idb.count',
  data: {}
});
```

22.3.6.2 Publish Topics (Results)

{store}.idb.ready

Published when database is initialized and ready.

Event Data: - database (String): Database name - store (String): Store name

{store}.idb.result

Published after successful operation.

Event Data: - **operation** (String): Operation name ('get', 'put', etc.) - **success** (Boolean): Always **true** - **requestId** (String, optional): Original request ID - Additional fields depend on operation:

- **get:** **item** (Object)
- **put/add:** **key** (Any)
- **list/query:** **items** (Array)
- **count:** **count** (Number)

{store}.idb.error

Published after failed operation.

Event Data: - **operation** (String): Operation name - **success** (Boolean): Always **false** - **error** (String): Error message - **requestId** (String, optional): Original request ID

22.3.7 Methods Reference

The component also exposes JavaScript methods for direct usage:

async get(key)

Retrieve item by key.

Returns: Promise resolving to item or **undefined**

Example:

```
const idb = document.querySelector('pan-idb');
const doc = await idb.get(123);
```

async put(item)

Insert or update item.

Returns: Promise resolving to key

Example:

```
const key = await idb.put({ id: 123, title: 'Updated' });
```

async add(item)

Insert item (throws if exists).

Returns: Promise resolving to key

async delete(key)

Delete item.

Returns: Promise resolving to **undefined**

async clear()

Delete all items.

Returns: Promise resolving to undefined

async list(options)

List items.

Parameters: - options (Object): Same as message data

Returns: Promise resolving to array of items

async query(index, value)

Query by index.

Returns: Promise resolving to array of items

async count(index)

Count items.

Returns: Promise resolving to number

22.3.8 Complete Working Examples

22.3.8.1 Document Storage

```
<!DOCTYPE html>
<html>
<head>
  <title>Document Manager</title>
</head>
<body>
  <pan-idb
    database="docapp"
    store="documents"
    key-path="id"
    auto-increment
    indexes=' [
      {"name": "byTitle", "keyPath": "title"},
      {"name": "byCreated", "keyPath": "created"}
    ]'>
  </pan-idb>

  <form id="doc-form">
    <input name="title" placeholder="Title" required>
    <textarea name="content" placeholder="Content"></textarea>
    <button type="submit">Save</button>
  </form>

  <ul id="doc-list"></ul>

  <script type="module">
    import { PanClient } from './pan-client.mjs';
```

```
const pc = new PanClient();
const form = document.getElementById('doc-form');
const list = document.getElementById('doc-list');

// Wait for database ready
pc.subscribe('documents.idb.ready', loadDocuments);

// Save document
form.addEventListener('submit', async (e) => {
  e.preventDefault();
  const formData = new FormData(form);

  pc.publish({
    topic: 'documents.idb.add',
    data: {
      item: {
        title: formData.get('title'),
        content: formData.get('content'),
        created: Date.now()
      }
    }
  });

  form.reset();
});

// Listen for save results
pc.subscribe('documents.idb.result', (msg) => {
  if (msg.data.operation === 'add') {
    loadDocuments();
  }
});

// Load and display documents
function loadDocuments() {
  pc.publish({
    topic: 'documents.idb.list',
    data: {
      index: 'byCreated',
      direction: 'prev',
      limit: 20
    }
  });
}

pc.subscribe('documents.idb.result', (msg) => {
  if (msg.data.operation === 'list') {
```

```

        renderDocuments(msg.data.items);
    }
});

function renderDocuments(docs) {
    list.innerHTML = docs.map(doc => `
        <li>
            <strong>${doc.title}</strong>
            <p>${doc.content}</p>
            <small>${new Date(doc.created).toLocaleString()}</small>
            <button onclick="deleteDoc(${doc.id})">Delete</button>
        </li>
    `).join('');
}

window.deleteDoc = (id) => {
    pc.publish({
        topic: 'documents.idb.delete',
        data: { key: id }
    });
};

pc.subscribe('documents.idb.result', (msg) => {
    if (msg.data.operation === 'delete') {
        loadDocuments();
    }
});
</script>
</body>
</html>

```

22.3.8.2 Direct API Usage

```

// Get reference to component
const idb = document.querySelector('pan-idb');

// Wait for ready
await customElements.whenDefined('pan-idb');
await idb.initPromise;

// CRUD operations
const id = await idb.add({
    title: 'Report Q4',
    status: 'draft',
    created: Date.now()
});

```

```

const doc = await idb.get(id);
console.log(doc);

doc.status = 'published';
await idb.put(doc);

// Query by index
const drafts = await idb.query('byStatus', 'draft');
console.log(`Found ${drafts.length} drafts`);

// List all with limit
const recent = await idb.list({
  index: 'byCreated',
  direction: 'prev',
  limit: 5
});

// Count items
const total = await idb.count();
console.log(`Total documents: ${total}`);

// Delete
await idb.delete(id);

```

22.3.8.3 Offline Task Queue

```

import { PanClient } from './pan-client.mjs';

class OfflineQueue {
  constructor() {
    this.pc = new PanClient();
    this.setupDatabase();
    this.setupListeners();
  }

  setupDatabase() {
    const idb = document.createElement('pan-idb');
    idb.setAttribute('database', 'offline-queue');
    idb.setAttribute('store', 'tasks');
    idb.setAttribute('key-path', 'id');
    idb.setAttribute('auto-increment', '');
    idb.setAttribute('indexes', JSON.stringify([
      { name: 'byStatus', keyPath: 'status' },
      { name: 'byTimestamp', keyPath: 'timestamp' }
    ]));
    document.body.appendChild(idb);
    this.idb = idb;
  }
}

```

```
}

setupListeners() {
  // Process queue when online
  window.addEventListener('online', () => this.processQueue());

  // Listen for new tasks
  this.pc.subscribe('queue.add', (msg) => {
    this.enqueue(msg.data.task);
  });
}

async enqueue(task) {
  await this.idb.add({
    ...task,
    status: 'pending',
    timestamp: Date.now()
  });

  if (navigator.onLine) {
    this.processQueue();
  }
}

async processQueue() {
  const pending = await this.idb.query('byStatus', 'pending');

  for (const task of pending) {
    try {
      await this.executeTask(task);
      await this.idb.delete(task.id);
    } catch (error) {
      console.error('Task failed:', error);
      // Update task status
      task.status = 'failed';
      task.error = error.message;
      await this.idb.put(task);
    }
  }
}

async executeTask(task) {
  // Execute actual task (e.g., API call)
  const response = await fetch(task.url, {
    method: task.method,
    body: JSON.stringify(task.data)
  });
}
```

```

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

    return response.json();
  }
}

// Usage
const queue = new OfflineQueue();

// Enqueue tasks
queue.pc.publish({
  topic: 'queue.add',
  data: {
    task: {
      url: '/api/items',
      method: 'POST',
      data: { name: 'New Item' }
    }
  }
});

```

22.3.8.4 Syncing with pan-store

```

import { createStore } from './pan-store.mjs';
import { PanClient } from './pan-client.mjs';

class PersistentStore {
  constructor(storeName, initialState = {}) {
    this.storeName = storeName;
    this.store = createStore(initialState);
    this.pc = new PanClient();
    this.setupPersistence();
    this.loadPersistedState();
  }

  setupPersistence() {
    // Create IndexedDB component
    const idb = document.createElement('pan-idb');
    idb.setAttribute('database', 'persistent-stores');
    idb.setAttribute('store', 'states');
    idb.setAttribute('key-path', 'name');
    document.body.appendChild(idb);
    this.idb = idb;
  }
}

```

```

    // Save on every change
    this.store.subscribe(({ detail }) => {
      this.persist();
    });
  }

  async loadPersistedState() {
    await customElements.whenDefined('pan-idb');
    await this.idb.initPromise;

    const saved = await this.idb.get(this.storeName);
    if (saved && saved.state) {
      this.store.patch(saved.state);
    }
  }

  async persist() {
    const snapshot = this.store.snapshot();
    await this.idb.put({
      name: this.storeName,
      state: snapshot,
      updated: Date.now()
    });
  }

  get state() {
    return this.store.state;
  }
}

// Usage
const appStore = new PersistentStore('app', {
  theme: 'light',
  sidebarOpen: true,
  fontSize: 14
});

// Changes automatically persist
appStore.state.theme = 'dark';

// State restored on page reload

```

22.3.9 Common Issues and Solutions

Issue: Database version conflicts

```

// Problem: Different tabs have different versions
// Tab 1 opens v1, Tab 2 tries v2, Tab 1 blocks upgrade

```

```
// Solution: Handle versionchange event
const idb = document.querySelector('pan-idb');
idb.db.addEventListener('versionchange', () => {
  idb.db.close();
  alert('Database upgraded. Please reload page.');
```

Issue: Quota exceeded errors

```
// Problem: Storing too much data
// Error: QuotaExceededError

// Solution: Check available storage
if (navigator.storage && navigator.storage.estimate) {
  const estimate = await navigator.storage.estimate();
  const percent = (estimate.usage / estimate.quota) * 100;

  if (percent > 90) {
    console.warn('Storage nearly full:', percent.toFixed(1) + '%');
```

Issue: Index not working after changes

```
// Problem: Modified keyPath but index still references old path

// Solution: Increment version and recreate indexes
// Change version="1" to version="2" in HTML
// onupgradeneeded handler will recreate indexes
```

Issue: Transactions timing out

```
// Problem: Long-running operation causes transaction timeout

// Solution: Break into smaller transactions
async function bulkInsert(items) {
  const BATCH_SIZE = 100;

  for (let i = 0; i < items.length; i += BATCH_SIZE) {
    const batch = items.slice(i, i + BATCH_SIZE);

    for (const item of batch) {
      await idb.add(item);
    }

    // Allow other operations between batches
    await new Promise(resolve => setTimeout(resolve, 0));
  }
}
```

```
}
```

22.4 Combining pan-store and pan-idb

The real power emerges when combining reactive state with persistent storage:

```
import { createStore } from './pan-store.mjs';

class HybridStore {
  constructor(name, initial = {}) {
    this.name = name;
    this.memory = createStore(initial);
    this.setupPersistence();
    this.setupSync();
  }

  setupPersistence() {
    const idb = document.createElement('pan-idb');
    idb.setAttribute('database', 'hybrid-stores');
    idb.setAttribute('store', 'data');
    idb.setAttribute('key-path', 'key');
    document.body.appendChild(idb);
    this.idb = idb;
  }

  async setupSync() {
    await customElements.whenDefined('pan-idb');
    await this.idb.initPromise;

    // Load persisted data
    const items = await this.idb.list();
    for (const item of items) {
      if (item.store === this.name) {
        this.memory.state[item.key] = item.value;
      }
    }

    // Sync changes to IndexedDB
    this.memory.subscribe(async ({ detail }) => {
      if (detail.deleted) {
        await this.idb.delete(`${this.name}.${detail.key}`);
      } else {
        await this.idb.put({
          key: `${this.name}.${detail.key}`,
          store: this.name,
          value: detail.value,
          updated: Date.now()
        });
      }
    });
  }
}
```

```

    });
  }
});
}

get state() {
  return this.memory.state;
}

subscribe(fn) {
  return this.memory.subscribe(fn);
}
}

// Usage: reactive AND persistent
const userPrefs = new HybridStore('preferences', {
  theme: 'light',
  language: 'en'
});

// Reactive updates
userPrefs.subscribe(({ detail }) => {
  console.log('Preference changed:', detail.key);
});

// Changes persist automatically
userPrefs.state.theme = 'dark';

```

22.5 Related Components

- **pan-client**: Underlying message bus for PAN communication
- **pan-persistence-strategy**: Advanced persistence patterns
- **pan-offline-sync**: Synchronization with remote servers
- **pan-event**: Event delegation and routing

22.6 Best Practices

1. **Choose the right tool**: Use pan-store for reactive state, pan-idb for persistence
2. **Avoid excessive persistence**: Don't save every keystroke to IndexedDB
3. **Version your schemas**: Plan for database migrations
4. **Handle errors gracefully**: Storage operations can fail
5. **Test offline scenarios**: Ensure app works without network
6. **Clean up subscriptions**: Prevent memory leaks
7. **Use indexes wisely**: Every index adds storage overhead
8. **Batch operations**: Group related changes when possible
9. **Monitor storage quota**: Don't assume unlimited space

10. **Document your state shape:** Make data structures explicit

22.7 Conclusion

Data management doesn't have to be chaotic. With `pan-store` and `pan-idb`, you have tools that handle state and persistence in ways that feel natural within LARC's architecture. Changes flow through messages, operations return predictable results, and components stay loosely coupled.

The key is choosing the right abstraction for your data. Ephemeral UI state stays in component properties. Shared reactive state lives in `pan-store`. Persistent data goes in `pan-idb`. Everything communicates via the PAN bus.

When you structure your data management this way, applications become comprehensible again. You know where state lives, how it changes, and when it persists. That clarity—knowing what's true about your application—is worth more than any clever framework feature.

Chapter 23

UI Components

“The best components are like good appliances: they do one thing well, fit perfectly into your existing setup, and you never have to think about how they work—until you need to, and then the manual is actually helpful.”

— A Developer Who’s Read Too Many Component APIs

If LARC’s PAN bus is the nervous system and custom components are the organs, then LARC’s built-in UI components are the power tools in your workshop. They’re purpose-built solutions for common UI patterns: file management, markdown editing, and content rendering. Each component is designed to work seamlessly with the PAN bus while remaining usable as a standalone web component.

This chapter provides comprehensive API documentation for three essential UI components: **pan-files**, **pan-markdown-editor**, and **pan-markdown-renderer**. These aren’t just reference docs—you’ll learn when to use each component, how to integrate them into your applications, and how to troubleshoot common issues.

23.1 pan-files: File System Browser

The **pan-files** component provides a complete file browser interface backed by the browser’s Origin Private File System (OPFS). It’s designed for applications that need client-side file management without server storage.

23.1.1 Overview and Purpose

pan-files is a file system manager that combines a visual file browser UI with programmatic file operations. It gives users a familiar folder-and-file interface while providing developers with a clean API for reading, writing, and managing files entirely in the browser.

Under the hood, **pan-files** uses OPFS, a browser-native storage API that provides fast, private file system access. Files stored in OPFS persist across sessions, survive page reloads, and remain sandboxed to your origin—they’re never sent to a server unless you explicitly choose to do so.

Key features include:

- Visual file browser with icons and metadata

- File and folder creation, renaming, and deletion
- Real-time search and filtering
- Drag-and-drop support (UI ready, extensible)
- PAN bus integration for file events
- Programmatic API for file operations

23.1.2 When to Use pan-files

Use pan-files when:

- Building offline-first applications that need local file storage
- Creating note-taking apps, code editors, or document managers
- You need client-side file persistence without backend infrastructure
- Building progressive web apps (PWAs) with file management features
- You want users to manage their own files privately in their browser

Don't use pan-files when:

- You need server-side file storage or sharing between users
- Files must be accessible from multiple devices (OPFS is device-specific)
- You need file system operations outside the browser (use Node.js fs module)
- Your files are large enough to approach storage quota limits
- You just need simple key-value storage (use localStorage or IndexedDB)

23.1.3 Installation and Setup

Add the component to your HTML:

```
<!DOCTYPE html>
<html>
<head>
  <script type="module">
    import '/path/to/pan-files.mjs';
  </script>
</head>
<body>
  <pan-files></pan-files>
</body>
</html>
```

The component automatically initializes OPFS when connected to the DOM. No additional configuration is required for basic usage.

23.1.4 Attributes Reference

23.1.4.1 path

- **Type:** String
- **Default:** '/'
- **Description:** The current directory path to display. Currently, the component primarily operates at the root level, but this attribute is designed for future nested directory navigation.

```
<pan-files path="/"></pan-files>
```

23.1.4.2 filter

- **Type:** String
- **Default:** '' (empty string, no filtering)
- **Description:** Comma-separated list of file extensions to display. Only files matching these extensions will be shown. Directories are always visible.

```
<!-- Show only markdown and text files -->
<pan-files filter=".md,.txt"></pan-files>
```

23.1.4.3 show-hidden

- **Type:** Boolean (as string)
- **Default:** 'false'
- **Description:** Whether to display hidden files (files starting with a dot).

```
<!-- Show hidden files -->
<pan-files show-hidden="true"></pan-files>
```

23.1.5 Methods Reference

All methods are asynchronous and return Promises. Methods that manipulate files automatically trigger UI refreshes and publish PAN bus events.

23.1.5.1 writeFile(path, content)

Writes content to a file, creating it if it doesn't exist or overwriting it if it does.

- **Parameters:**
 - path (String): File path (e.g., '/notes.txt')
 - content (String): Content to write
- **Returns:** Promise<void>
- **Throws:** Error if write operation fails

```
const files = document.querySelector('pan-files');
await files.writeFile('/hello.txt', 'Hello, World!');
```

23.1.5.2 readFile(path)

Reads the contents of a file as text.

- **Parameters:**
 - path (String): File path to read
- **Returns:** Promise<String> - File contents
- **Throws:** Error if file doesn't exist or read fails

```
const files = document.querySelector('pan-files');
const content = await files.readFile('/hello.txt');
console.log(content); // "Hello, World!"
```

23.1.5.3 deleteFile(path)

Deletes a file and publishes a deletion event.

- **Parameters:**
 - path (String): File path to delete
- **Returns:** Promise<void>
- **Throws:** Error if deletion fails
- **Side Effects:** Publishes file.deleted event to PAN bus

```
const files = document.querySelector('pan-files');
await files.deleteFile('/old-notes.txt');
```

23.1.5.4 listFiles()

Returns an array of all files in the current directory.

- **Parameters:** None
- **Returns:** Promise<Array<FileInfo>> - Array of file objects

Each FileInfo object contains:

```
{
  name: 'example.txt',      // File name
  path: '/example.txt',    // Full path
  isDirectory: false,      // Whether it's a directory
  size: 1024,               // File size in bytes
  entry: FileSystemHandle   // Native OPFS handle
}
```

```
const files = document.querySelector('pan-files');
const allFiles = await files.listFiles();
console.log(`Found ${allFiles.length} files`);
```

23.1.5.5 refresh()

Reloads the file list from OPFS and updates the UI.

- **Parameters:** None
- **Returns:** Promise<void>

```
const files = document.querySelector('pan-files');
await files.refresh();
```

23.1.6 Events Reference

pan-files publishes and subscribes to PAN bus events for integration with other components.

23.1.6.1 Published Events

file.selected Published when a user clicks on a file in the browser.

Payload:

```
{
  path: '/example.txt',    // Full file path
  name: 'example.txt',    // File name
  isDirectory: false      // Whether it's a directory
}
```

file.created Published when a file or folder is created.

Payload:

```
{
  path: '/new-file.txt',
  name: 'new-file.txt',
  isDirectory: false      // true for folders
}
```

file.deleted Published when a file is deleted.

Payload:

```
{
  path: '/deleted.txt'
}
```

file.renamed Published when a file is renamed.

Payload:

```
{
  oldPath: '/old-name.txt',
  newPath: '/new-name.txt'
}
```

file.content-loaded Published in response to a **file.load** event.

Payload:

```
{
  path: '/example.txt',
  content: 'File contents here...'
}
```

23.1.6.2 Subscribed Events

file.save Saves a file with the provided content.

Payload:

```
{
  path: '/save-me.txt',
  content: 'Content to save'
}
```

file.load Loads a file and publishes its content via **file.content-loaded**.

Payload:

```
{
  path: '/load-me.txt'
}
```

file.delete Deletes a file.

Payload:

```
{
  path: '/delete-me.txt'
}
```

file.create Creates a file with optional content.

Payload:

```
{
  path: '/new.txt',
  content: 'Optional initial content' // Defaults to empty string
}
```

23.1.7 Complete Working Example

Here's a complete example showing **pan-files** integrated with the PAN bus:

```
<!DOCTYPE html>
<html>
<head>
  <script type="module">
    import './pan-bus.mjs';
    import './pan-files.mjs';

    // Wait for components to load
    customElements.whenDefined('pan-bus').then(() => {
      const bus = document.querySelector('pan-bus');

      // Listen for file selections
      bus.subscribe('file.selected', (msg) => {
        console.log('File selected:', msg.data.path);

        // Load the file's contents
        bus.publish('file.load', { path: msg.data.path });
      });
    });
  </script>
</head>
</html>
```

```

});

// Listen for file content
bus.subscribe('file.content-loaded', (msg) => {
  console.log('File content:', msg.data.content);
  document.getElementById('preview').textContent = msg.data.content;
});

// Programmatic file operations
window.createQuickNote = async () => {
  const files = document.querySelector('pan-files');
  const timestamp = new Date().toISOString();
  await files.writeFile(`/note-${Date.now()}.txt`,
    `Note created at ${timestamp}`);
  await files.refresh();
};
});
</script>
<style>
  body {
    display: grid;
    grid-template-columns: 300px 1fr;
    gap: 1rem;
    height: 100vh;
    margin: 0;
    padding: 1rem;
  }
  pan-files {
    border: 1px solid #ccc;
    border-radius: 4px;
  }
  #preview {
    padding: 1rem;
    border: 1px solid #ccc;
    border-radius: 4px;
    white-space: pre-wrap;
    font-family: monospace;
  }
</style>
</head>
<body>
  <pan-bus></pan-bus>

  <pan-files filter=".txt,.md"></pan-files>

  <div>
    <button onclick="createQuickNote()">Create Quick Note</button>

```

```
<pre id="preview">Select a file to preview...</pre>
</div>
</body>
</html>
```

23.1.8 Related Components

- **pan-markdown-editor**: Use alongside **pan-files** for editing markdown files
- **pan-bus**: Required for event-based file operations
- **pan-markdown-renderer**: Display rendered markdown from files

23.1.9 Common Issues and Solutions

Issue: Files don't persist after closing the browser

OPFS storage persists by default, but it can be cleared if:

- User clears browser data
- Browser is in private/incognito mode
- Storage quota is exceeded

Solution: Inform users about persistence limitations and provide export functionality for critical data.

Issue: “Failed to initialize OPFS” error

Cause: OPFS requires a secure context (HTTPS) and isn't available in all browsers.

Solution:

- Ensure your site runs on HTTPS (or localhost for development)
- Check browser support: OPFS works in Chrome 102+, Edge 102+, Opera 88+
- Provide fallback storage (IndexedDB) for unsupported browsers

Issue: File list doesn't update after programmatic changes

Cause: The UI doesn't automatically refresh after calling `writeFile()` or `deleteFile()`.

Solution: Call `refresh()` after file operations:

```
await files.writeFile('/new.txt', 'content');
await files.refresh();
```

Issue: Can't access files from network requests or other origins

Cause: OPFS is origin-private—files aren't accessible via URLs or from other domains.

Solution: This is by design for security. To share files, explicitly read content and send via `fetch`/`WebSocket`.

23.2 pan-markdown-editor: Rich Markdown Editor

The `pan-markdown-editor` component is a full-featured markdown editor with formatting toolbar, live preview, keyboard shortcuts, and auto-save capabilities.

23.2.1 Overview and Purpose

`pan-markdown-editor` transforms a simple textarea into a powerful markdown editing environment. It's designed for content-heavy applications like note-taking apps, documentation tools, blogs, and content management systems.

The editor provides a rich formatting toolbar with common markdown operations, supports keyboard shortcuts for power users, and includes a live preview pane that renders markdown as you type. It integrates seamlessly with the PAN bus, broadcasting changes and responding to external commands.

Key features include:

- Rich formatting toolbar (bold, italic, headings, lists, etc.)
- Keyboard shortcuts (Ctrl+B, Ctrl+I, Ctrl+K, etc.)
- Optional live preview with split-pane view
- Auto-indent and list continuation
- Word and character count
- Optional auto-save with debouncing
- Tab key support for indentation

23.2.2 When to Use pan-markdown-editor

Use `pan-markdown-editor` when:

- Building markdown-based content applications
- You need a WYSIWYG-lite editing experience
- Users benefit from a formatting toolbar
- You want keyboard shortcuts for common formatting
- Live preview helps users understand markdown rendering

Don't use `pan-markdown-editor` when:

- You need WYSIWYG rich text editing (use a rich text editor)
- Users don't know markdown (provide alternatives or training)
- You're editing non-markdown content (use plain textarea)
- You need mobile-optimized input (the toolbar can be cramped)

23.2.3 Installation and Setup

```
<!DOCTYPE html>
<html>
<head>
  <script type="module">
    import './pan-markdown-editor.mjs';
    import './pan-markdown-renderer.mjs'; // Required for preview mode
  </script>
```

```
</head>
<body>
  <pan-markdown-editor
    value="# Hello World"
    preview="true">
  </pan-markdown-editor>
</body>
</html>
```

23.2.4 Attributes Reference

23.2.4.1 value

- **Type:** String
- **Default:** ''
- **Description:** Initial markdown content for the editor.

```
<pan-markdown-editor value="# My Document
Start writing here..."></pan-markdown-editor>
```

23.2.4.2 placeholder

- **Type:** String
- **Default:** 'Start writing...'
- **Description:** Placeholder text shown when the editor is empty.

```
<pan-markdown-editor placeholder="Enter your markdown here..."></pan-markdown-editor>
```

23.2.4.3 preview

- **Type:** Boolean (as string)
- **Default:** 'false'
- **Description:** Whether to show the live preview pane alongside the editor.

```
<pan-markdown-editor preview="true"></pan-markdown-editor>
```

23.2.4.4 autosave

- **Type:** Boolean (as string)
- **Default:** 'false'
- **Description:** Whether to enable auto-save with 1-second debouncing. When enabled, publishes `markdown.saved` events automatically.

```
<pan-markdown-editor autosave="true"></pan-markdown-editor>
```

23.2.5 Methods Reference

23.2.5.1 setValue(value)

Sets the editor content programmatically.

- **Parameters:**
 - value (String): New markdown content
- **Returns:** void
- **Side Effects:** Updates stats, preview, and triggers change events

```
const editor = document.querySelector('pan-markdown-editor');
editor.setValue('# New Content\n\nThis replaces all existing content.');
```

23.2.5.2 getValue()

Returns the current editor content.

- **Parameters:** None
- **Returns:** String - Current markdown content

```
const editor = document.querySelector('pan-markdown-editor');
const markdown = editor.getValue();
console.log(markdown);
```

23.2.5.3 insertText(text)

Inserts text at the current cursor position.

- **Parameters:**
 - text (String): Text to insert
- **Returns:** void

```
const editor = document.querySelector('pan-markdown-editor');
editor.insertText('\n\n---\n\n'); // Insert horizontal rule
```

23.2.5.4 focus()

Focuses the editor textarea.

- **Parameters:** None
- **Returns:** void

```
const editor = document.querySelector('pan-markdown-editor');
editor.focus();
```

23.2.6 Toolbar Actions

The toolbar provides buttons for common markdown operations. Each button has a corresponding keyboard shortcut.

Button	Action	Keyboard	Result
B	Bold	Ctrl+B	**text**
I	Italic	Ctrl+I	<i>*text*</i>
S	Strikethrough	-	~~text~~

Button	Action	Keyboard	Result
H1-H3	Headings	-	# text
* List	Bullet list	-	* item
1. List	Numbered list	-	1. item
v Task	Task list	-	- [] task
[link] Link	Insert link	Ctrl+K	[text] (url)
[image] Image	Insert image	-	![alt] (url)
{ }	Inline code	-	`code`
</>	Code block	-	```lang\ncode\n```
" Quote	Blockquote	-	> quote
-	Horizontal rule	-	---
[+] Table	Insert table	-	Markdown table template
[eye] Preview	Toggle preview	-	Shows/hides preview pane

23.2.7 Keyboard Shortcuts

- **Ctrl+B**: Bold selection
- **Ctrl+I**: Italic selection
- **Ctrl+K**: Insert link
- **Ctrl+S**: Save (publishes `markdown.saved` event)
- **Tab**: Insert two spaces (for indentation)
- **Enter**: Auto-continue lists (bullets, numbers, tasks)

23.2.8 Events Reference

23.2.8.1 Published Events

markdown.changed Published whenever the content changes.

Payload:

```
{
  content: '# Markdown content',
  wordCount: 42,
  charCount: 256
}
```

markdown.saved Published when Ctrl+S is pressed or auto-save triggers.

Payload:

```
{
  content: '# Current markdown content'
}
```

23.2.8.2 Subscribed Events

markdown.set-content Sets the editor content externally.

Payload:

```
{
  content: '# New content to set'
}
```

`markdown.get-content` Requests current content. The editor responds by publishing `markdown.content-response`.

Payload: None (empty object)

Response via `markdown.content-response`:

```
{
  content: '# Current content'
}
```

23.2.9 Complete Working Example

Here's a markdown editor integrated with file storage:

```
<!DOCTYPE html>
<html>
<head>
  <script type="module">
    import './pan-bus.mjs';
    import './pan-files.mjs';
    import './pan-markdown-editor.mjs';
    import './pan-markdown-renderer.mjs';

    customElements.whenDefined('pan-bus').then(() => {
      const bus = document.querySelector('pan-bus');
      let currentFile = null;

      // Load file content into editor when selected
      bus.subscribe('file.selected', (msg) => {
        if (msg.data.path.endsWith('.md')) {
          currentFile = msg.data.path;
          bus.publish('file.load', { path: msg.data.path });
        }
      });

      bus.subscribe('file.content-loaded', (msg) => {
        const editor = document.querySelector('pan-markdown-editor');
        editor.setValue(msg.data.content);
        document.getElementById('filename').textContent =
          msg.data.path.split('/').pop();
      });

      // Save editor content back to file
      bus.subscribe('markdown.saved', async (msg) => {
        if (currentFile) {
```

```

        bus.publish('file.save', {
            path: currentFile,
            content: msg.data.content
        });
        showNotification('Saved!');
    }
});

window.createNewNote = () => {
    currentFile = `/note-${Date.now()}.md`;
    document.getElementById('filename').textContent =
        currentFile.split('/').pop();
    const editor = document.querySelector('pan-markdown-editor');
    editor.setValue('# New Note\n\n');
    editor.focus();
};

function showNotification(message) {
    const notif = document.getElementById('notification');
    notif.textContent = message;
    notif.style.display = 'block';
    setTimeout(() => notif.style.display = 'none', 2000);
}
});
</script>
<style>
body {
    margin: 0;
    padding: 0;
    display: grid;
    grid-template-rows: auto 1fr;
    height: 100vh;
}
.toolbar {
    padding: 1rem;
    border-bottom: 1px solid #ccc;
    display: flex;
    gap: 1rem;
    align-items: center;
}
#filename {
    font-weight: bold;
    flex: 1;
}
#notification {
    display: none;
    background: #4caf50;

```

```

        color: white;
        padding: 0.5rem 1rem;
        border-radius: 4px;
    }
    .content {
        display: grid;
        grid-template-columns: 250px 1fr;
        gap: 1rem;
        padding: 1rem;
        overflow: hidden;
    }
    pan-files, pan-markdown-editor {
        border: 1px solid #ccc;
        border-radius: 4px;
        overflow: auto;
    }
</style>
</head>
<body>
    <pan-bus></pan-bus>

    <div class="toolbar">
        <span id="filename">No file selected</span>
        <button onclick="createNewNote()">New Note</button>
        <div id="notification"></div>
    </div>

    <div class="content">
        <pan-files filter=".md"></pan-files>
        <pan-markdown-editor
            preview="true"
            autosave="true"
            placeholder="Select a file or create a new note...">
        </pan-markdown-editor>
    </div>
</body>
</html>

```

23.2.10 Related Components

- **pan-files**: Store and retrieve markdown files
- **pan-markdown-renderer**: Display rendered output
- **pan-bus**: Coordinate between editor and other components

23.2.11 Common Issues and Solutions

Issue: Toolbar buttons don't work on mobile

Cause: Mobile browsers handle focus and selection differently.

Solution: The component is optimized for desktop. For mobile, consider hiding the toolbar and relying on keyboard shortcuts or providing a simplified mobile UI.

Issue: Preview pane doesn't update

Cause: `pan-markdown-renderer` component isn't loaded.

Solution: Ensure you import both components:

```
import './pan-markdown-editor.mjs';
import './pan-markdown-renderer.mjs';
```

Issue: Large documents cause lag

Cause: Real-time rendering of very large documents can be slow.

Solution:

- Disable preview for large documents
- Add debouncing to preview updates
- Split large documents into smaller sections

Issue: Keyboard shortcuts conflict with browser shortcuts

Cause: Some browsers intercept Ctrl+S, Ctrl+K, etc.

Solution: The component calls `preventDefault()` for most shortcuts, but browser behavior varies. Consider documenting known conflicts.

23.3 pan-markdown-renderer: Markdown Display

The `pan-markdown-renderer` component takes markdown text and renders it as formatted HTML with syntax highlighting, tables, and GitHub-flavored markdown support.

23.3.1 Overview and Purpose

`pan-markdown-renderer` is a read-only component that displays markdown content as formatted HTML. It's the display counterpart to `pan-markdown-editor`—while the editor lets users write markdown, the renderer shows them what it looks like.

The renderer implements a custom markdown parser that supports standard markdown syntax plus GitHub-flavored extensions like task lists and tables. It's designed to be lightweight (no external dependencies), secure (HTML sanitization), and styleable (CSS custom properties).

Key features include:

- Complete markdown syntax support
- GitHub-flavored markdown (tables, task lists)
- Safe HTML rendering (sanitized by default)
- Syntax highlighting structure (CSS-based)
- Responsive design

- Custom styling via CSS variables

23.3.2 When to Use pan-markdown-renderer

Use `pan-markdown-renderer` when:

- Displaying markdown content to users
- Building preview panes for editors
- Rendering blog posts, documentation, or comments
- You need sanitized HTML output from markdown
- You want consistent markdown rendering across your app

Don't use `pan-markdown-renderer` when:

- You need a full-featured markdown parser (use `marked.js`, `markdown-it`)
- You require advanced syntax highlighting (use `Prism` or `highlight.js`)
- You're rendering non-markdown content
- You need to edit content (use `pan-markdown-editor`)

23.3.3 Installation and Setup

```
<!DOCTYPE html>
<html>
<head>
  <script type="module">
    import './pan-markdown-renderer.mjs';
  </script>
</head>
<body>
  <pan-markdown-renderer content="# Hello World

This is bold and this is italic."></pan-markdown-renderer>
</body>
</html>
```

23.3.4 Attributes Reference

23.3.4.1 content

- **Type:** String
- **Default:** ''
- **Description:** Markdown content to render. Changes to this attribute trigger re-rendering.

```
<pan-markdown-renderer content="# Title

Paragraph text here."></pan-markdown-renderer>
```

23.3.4.2 sanitize

- **Type:** Boolean (as string)
- **Default:** 'true'

- **Description:** Whether to sanitize HTML in markdown content. When true, raw HTML is escaped. Set to false only if you trust the content source.

```
<!-- Allow raw HTML (use with caution) -->
<pan-markdown-renderer
  content="# Title

<div class='custom'>Raw HTML</div>"
  sanitize="false">
</pan-markdown-renderer>
```

23.3.5 Methods Reference

23.3.5.1 setContent(content)

Sets the markdown content programmatically and triggers rendering.

- **Parameters:**
 - content (String): Markdown content to render
- **Returns:** void

```
const renderer = document.querySelector('pan-markdown-renderer');
renderer.setContent('# Dynamic Content\n\nUpdated at runtime.');
```

23.3.5.2 getContent()

Returns the current markdown content (not the rendered HTML).

- **Parameters:** None
- **Returns:** String - Current markdown content

```
const renderer = document.querySelector('pan-markdown-renderer');
console.log(renderer.getContent());
```

23.3.5.3 getHtml()

Returns the rendered HTML output.

- **Parameters:** None
- **Returns:** String - Rendered HTML

```
const renderer = document.querySelector('pan-markdown-renderer');
const html = renderer.getHtml();
console.log(html); // "<h1>Title</h1><p>Content...</p>"
```

23.3.6 Supported Markdown Syntax

The renderer supports the following markdown features:

Headings

```
# H1
## H2
### H3
#### H4
##### H5
##### H6
```

Emphasis

```
**bold** or __bold__
*italic* or _italic_
~~strikethrough~~
```

Lists

```
* Bullet item
* Another item

1. Numbered item
2. Another item

- [ ] Unchecked task
- [x] Checked task
```

Links and Images

```
[Link text](https://example.com)
![Alt text](https://example.com/image.png)
```

Code

```
Inline `code`

```
```javascript
// Code block with language
function hello() {
  console.log('Hello');
}
```


```

```
Blockquotes

```
```markdown
> This is a quote
> with multiple lines
```


```

Horizontal Rules

```
---
***
---
```

Tables

```
Header 1	Header 2
Cell 1	Cell 2
```

23.3.7 Styling with CSS Variables

The renderer uses CSS custom properties for easy theming:

```
pan-markdown-renderer {
  /* Text colors */
  --color-text: #1e293b;
  --color-text-muted: #64748b;

  /* Backgrounds */
  --color-bg-alt: #f8fafc;
  --color-border: #e2e8f0;

  /* Code blocks */
  --color-code-bg: #1e293b;
  --color-code-text: #e2e8f0;

  /* Links */
  --color-primary: #006699;

  /* Fonts */
  --font-mono: 'Courier New', monospace;
}
```

Example: Dark mode theme

```
pan-markdown-renderer.dark-mode {
  --color-text: #e2e8f0;
  --color-text-muted: #94a3b8;
  --color-bg-alt: #1e293b;
  --color-border: #334155;
  --color-code-bg: #0f172a;
  --color-code-text: #e2e8f0;
  --color-primary: #38bdf8;
}
```

23.3.8 Events Reference

23.3.8.1 Subscribed Events

markdown.render Triggers rendering with new content.

Payload:

```
{
  content: '# Content to render'
```

```
}

```

Example:

```
const bus = document.querySelector('pan-bus');
bus.publish('markdown.render', {
  content: '# Hello from PAN bus'
});

```

23.3.9 Complete Working Example

Here's a complete example showing a markdown documentation viewer:

```
<!DOCTYPE html>
<html>
<head>
  <script type="module">
    import './pan-bus.mjs';
    import './pan-markdown-renderer.mjs';

    customElements.whenDefined('pan-bus').then(() => {
      const bus = document.querySelector('pan-bus');
      const renderer = document.querySelector('pan-markdown-renderer');

      // Sample documentation sections
      const docs = {
        intro: `# Getting Started

Welcome to our documentation! This guide will help you understand the basics.

## Prerequisites

Before you begin, make sure you have:

- A modern web browser
- Basic knowledge of HTML and JavaScript
- 15 minutes of free time

## Installation

1. Download the package
2. Extract to your project
3. Import the components

\\`\\`\\`javascript
import './components/app.mjs';
\\`\\`\\`,

        api: `# API Reference

```

```

## Core Methods

### ``initialize(config)``

Initializes the application with the provided configuration.

**Parameters:**
- ``config`` (Object): Configuration object
  - ``debug`` (Boolean): Enable debug mode
  - ``theme`` (String): Theme name

**Returns:** Promise<void>

**Example:**
```javascript
await initialize({
 debug: true,
 theme: 'dark'
});
```

    examples: `# Examples

## Hello World

The simplest example:

```html
<hello-world></hello-world>
```

## Task List

- [x] Create component
- [x] Write documentation
- [ ] Deploy to production

## Data Table

Feature	Supported	Version
Import	[check]	1.0
Export	[check]	1.0
Sync	[hourglass]	2.0
    };

    // Navigation

```

```
window.showDoc = (section) => {
  renderer.setContent(docs[section]);

  // Update active state
  document.querySelectorAll('nav button').forEach(btn => {
    btn.classList.toggle('active', btn.dataset.section === section);
  });
};

// Show initial doc
showDoc('intro');
});
</script>
<style>
body {
  margin: 0;
  font-family: system-ui, sans-serif;
  display: grid;
  grid-template-columns: 200px 1fr;
  height: 100vh;
}
nav {
  background: #f8fafc;
  padding: 1rem;
  border-right: 1px solid #e2e8f0;
}
nav h2 {
  margin-top: 0;
  font-size: 1rem;
  color: #64748b;
}
nav button {
  display: block;
  width: 100%;
  padding: 0.5rem;
  margin: 0.25rem 0;
  border: none;
  background: transparent;
  text-align: left;
  cursor: pointer;
  border-radius: 4px;
  transition: background 0.15s;
}
nav button:hover {
  background: #e2e8f0;
}
nav button.active {
```

```

    background: #006699;
    color: white;
  }
  main {
    padding: 2rem;
    overflow-y: auto;
  }
  pan-markdown-renderer {
    display: block;
    max-width: 800px;
  }
</style>
</head>
<body>
  <pan-bus></pan-bus>

  <nav>
    <h2>Documentation</h2>
    <button data-section="intro" onclick="showDoc('intro')" class="active">
      Getting Started
    </button>
    <button data-section="api" onclick="showDoc('api')">
      API Reference
    </button>
    <button data-section="examples" onclick="showDoc('examples')">
      Examples
    </button>
  </nav>

  <main>
    <pan-markdown-renderer></pan-markdown-renderer>
  </main>
</body>
</html>

```

23.3.10 Related Components

- **pan-markdown-editor**: Edit markdown content
- **pan-files**: Store markdown files
- **pan-bus**: Coordinate rendering with other components

23.3.11 Common Issues and Solutions

Issue: Code blocks don't have syntax highlighting

Cause: The renderer provides structure but not syntax coloring.

Solution: Add a CSS-based syntax highlighter or a library like Prism.js:

```
<link rel="stylesheet" href="prism.css">
<script src="prism.js"></script>
```

Issue: Tables render incorrectly

Cause: Table markdown must follow strict formatting with alignment rows.

Solution: Ensure tables have a header separator row:

```
| Header 1 | Header 2 |
|-----|-----|  <- Required separator
| Cell 1   | Cell 2   |
```

Issue: Raw HTML appears in output

Cause: HTML sanitization is enabled by default.

Solution: Only disable sanitization if you trust the content source:

```
<pan-markdown-renderer sanitize="false" content="<div>Raw HTML</div>">
</pan-markdown-renderer>
```

Issue: Custom markdown extensions not supported

Cause: The built-in parser implements standard markdown only.

Solution: For advanced features, consider replacing the internal parser with markdown-it or marked.js and subclassing the component.

Issue: Markdown doesn't wrap in mobile views

Cause: Long code blocks or wide tables can overflow.

Solution: Add responsive styling:

```
pan-markdown-renderer {
  overflow-x: auto;
}
pan-markdown-renderer pre {
  max-width: 100%;
  overflow-x: auto;
}
```

23.4 Component Integration Patterns

The three components work best when integrated together. Here are common patterns:

23.4.1 Pattern 1: Markdown Note-Taking App

```
// Connect file selection -> editor -> auto-save -> file storage
bus.subscribe('file.selected', (msg) => {
  bus.publish('file.load', { path: msg.data.path });
```

```
});

bus.subscribe('file.content-loaded', (msg) => {
  const editor = document.querySelector('pan-markdown-editor');
  editor.setValue(msg.data.content);
});

bus.subscribe('markdown.saved', (msg) => {
  bus.publish('file.save', {
    path: currentFile,
    content: msg.data.content
  });
});
```

23.4.2 Pattern 2: Documentation Viewer

```
// Render markdown files as formatted documentation
bus.subscribe('file.selected', async (msg) => {
  if (msg.data.path.endsWith('.md')) {
    const files = document.querySelector('pan-files');
    const content = await files.readFile(msg.data.path);

    const renderer = document.querySelector('pan-markdown-renderer');
    renderer.setContent(content);
  }
});
```

23.4.3 Pattern 3: Split-View Editor

```
<div class="split-view">
  <pan-markdown-editor></pan-markdown-editor>
  <pan-markdown-renderer></pan-markdown-renderer>
</div>

<script type="module">
  // Sync editor to renderer
  bus.subscribe('markdown.changed', (msg) => {
    const renderer = document.querySelector('pan-markdown-renderer');
    renderer.setContent(msg.data.content);
  });
</script>
```

23.5 Conclusion

LARC's UI components demonstrate the power of web components: they're self-contained, reusable, and work with or without a framework. `pan-files` handles storage, `pan-markdown-editor` handles input, and `pan-markdown-renderer` handles display—each focused on doing one thing well.

The PAN bus ties them together, allowing components to communicate without tight coupling. You can use these components individually or compose them into full applications. And because they're built on web standards, they'll work in browsers long after today's frameworks fade into obscurity.

In the next chapter, we'll explore advanced component patterns: building custom UI components that follow these same principles, creating reusable component libraries, and designing component APIs that stand the test of time.

Chapter 24

Integration Components

In which we bridge the gap between LARC applications and the outside world—REST APIs, GraphQL servers, WebSocket streams, and Server-Sent Events—without losing our composure or our data

Every modern web application is, at heart, an integration problem. You’re not building a standalone fortress; you’re building a trading post that speaks multiple languages, accepts multiple currencies, and somehow keeps track of what goes in and what goes out. Your frontend needs to talk to REST APIs, subscribe to real-time WebSocket feeds, execute GraphQL queries, and listen to Server-Sent Event streams—often simultaneously.

LARC’s integration components solve this problem by providing declarative, PAN-bus-connected adapters for external data sources. They transform HTTP requests, WebSocket events, and SSE streams into PAN messages, and PAN messages back into network requests. The result is a clean architectural boundary: your application components remain blissfully unaware of whether their data comes from REST, GraphQL, or a carrier pigeon.

This chapter provides comprehensive API documentation for four integration components:

- **pan-data-connector**: REST API integration with full CRUD support
- **pan-graphql-connector**: GraphQL query and mutation bridge
- **pan-websocket**: Bidirectional WebSocket communication
- **pan-sse**: Server-Sent Events streaming

Each section follows the same structure: overview, usage guidance, installation, attribute/method/event reference, complete examples, and troubleshooting. Think of this chapter as your field guide to connecting LARC applications to the wider internet ecosystem.

24.1 pan-data-connector

24.1.1 Overview

pan-data-connector is a declarative REST API bridge that maps PAN bus topics to HTTP endpoints. It implements the standard CRUD pattern—list, get, create, update, delete—using `fetch()` and publishes responses as retained PAN messages. This allows components to request data via topics without knowing anything about HTTP methods, URL construction, or response handling.

The connector listens for request topics like `${resource}.list.get` and `${resource}.item.save`, performs the appropriate HTTP request, and publishes state updates to `${resource}.list.state` and `${resource}.item.state.${id}`. All state messages are retained, so late-subscribing components receive the most recent data immediately.

24.1.2 When to Use

Use `pan-data-connector` when:

- Working with RESTful APIs that follow standard CRUD patterns
- You want declarative data fetching without writing `fetch()` calls in every component
- You need automatic state synchronization across multiple components
- You're building admin interfaces, CRUD applications, or data management tools
- Your API uses predictable URL patterns (e.g., `/api/users`, `/api/users/:id`)

Don't use `pan-data-connector` when:

- Your API doesn't follow REST conventions (use custom `fetch()` or build a specialized connector)
- You need fine-grained control over request timing and caching
- Your endpoints use non-standard HTTP methods or complex request patterns
- You're working with GraphQL (use `pan-graphql-connector` instead)

24.1.3 Installation and Setup

Include the component module and add it to your HTML:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/ui/pan-bus.mjs"></script>
  <script type="module" src="/ui/pan-data-connector.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <!-- Simple configuration -->
  <pan-data-connector
    resource="users"
    base-url="https://api.example.com">
  </pan-data-connector>

  <!-- Your application -->
</body>
</html>
```

For APIs requiring authentication headers:

```
<pan-data-connector
  resource="users"
  base-url="https://api.example.com"
```

```

credentials="include">
<script type="application/json">
{
  "headers": {
    "Authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "X-API-Version": "2023-01"
  }
}
</script>
</pan-data-connector>

```

24.1.4 Attributes

Attribute	Type	Default	Description
resource	String	"items"	Logical resource name. Used as the topic prefix (e.g., <code>users</code> creates topics like <code>users.list.get</code>).
base-url	String	""	Base URL for API endpoints. Trailing slashes are automatically removed.
key	String	"id"	The field name used as the unique identifier for items.
list-path	String	"/\${resource}"	URL path template for list operations. Override for non-standard endpoints.
item-path	String	"/\${resource}/\${id}"	URL path template for single-item operations. The <code>:id</code> placeholder is replaced with the actual ID.
update-method	String	"PUT"	HTTP method for updates. Use "PATCH" for partial updates.
credentials	String	""	Fetch credentials mode: "include", "same-origin", or "omit".

Example configurations:

```

<!-- Non-standard paths -->
<pan-data-connector
  resource="products"
  base-url="https://shop.example.com"
  list-path="/v2/catalog/products"
  item-path="/v2/catalog/products/:id">
</pan-data-connector>

```

```

<!-- UUID-based API -->
<pan-data-connector
  resource="orders"
  base-url="/api"
  key="uuid"
  update-method="PATCH">
</pan-data-connector>

<!-- Complex authentication -->
<pan-data-connector resource="documents" base-url="/api/v1">
  <script type="application/json">
    {
      "headers": {
        "Authorization": "Bearer ${TOKEN}",
        "X-Tenant-ID": "acme-corp",
        "Accept": "application/vnd.api+json"
      }
    }
  </script>
</pan-data-connector>

```

24.1.5 Topics

The connector listens to and publishes messages on the following topics:

24.1.5.1 Subscribed Topics (Requests)

`${resource}.list.get`

Fetches the list of items. Query parameters can be passed in the message data.

Request payload:

```

{
  // Optional: any query parameters
  page: 1,
  limit: 20,
  filter: 'active'
}

```

`${resource}.item.get`

Fetches a single item by ID.

Request payload:

```

{
  id: 123
}
// Or simply: 123

```

`${resource}.item.save`

Creates a new item (if no ID) or updates an existing item.

Request payload:

```
{
  item: {
    id: 123, // Optional; omit for creation
    name: "New Product",
    price: 29.99
  }
}
// Or simply: { id: 123, name: "...", price: 29.99 }
```

`${resource}.item.delete`

Deletes an item by ID.

Request payload:

```
{
  id: 123
}
// Or simply: 123
```

24.1.5.2 Published Topics (Responses)**`${resource}.list.state`** (retained)

Published after successful list fetch. Contains the current list of items.

Payload:

```
{
  items: [
    { id: 1, name: "Product A", price: 19.99 },
    { id: 2, name: "Product B", price: 29.99 }
  ]
}
```

`${resource}.item.state.${id}` (retained)

Published after successful item fetch or save. Contains the current item state.

Payload:

```
{
  item: {
    id: 123,
    name: "Product C",
    price: 39.99,
    updatedAt: "2024-01-15T10:30:00Z"
  }
}
```

```

    }
  }

```

For deletions, a non-retained deletion notification is published:

```

{
  id: 123,
  deleted: true
}

```

24.1.5.3 Reply Topics

If the request includes `replyTo` and `correlationId` fields, the connector publishes a response to the reply topic:

Success response:

```

{
  ok: true,
  items: [...], // For list operations
  item: {...}   // For item operations
}

```

Error response:

```

{
  ok: false,
  error: {
    status: 404,
    statusText: "Not Found",
    body: { message: "Item not found" }
  }
}

```

24.1.6 Authentication Integration

`pan-data-connector` automatically integrates with LARC's authentication system. It subscribes to `auth.internal.state` (retained) and automatically injects `Authorization: Bearer ${token}` headers when a token is available.

This means you can configure authentication once in `pan-auth-provider`, and all connectors automatically include credentials:

```

<pan-auth-provider
  storage="local"
  token-key="app_token">
</pan-auth-provider>

<!-- This connector will automatically use the auth token -->
<pan-data-connector
  resource="users"

```

```
base-url="https://api.example.com">
</pan-data-connector>
```

24.1.7 Complete Examples

24.1.7.1 Basic CRUD Application

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/ui/pan-bus.mjs"></script>
  <script type="module" src="/ui/pan-data-connector.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <pan-data-connector
    resource="todos"
    base-url="/api">
  </pan-data-connector>

  <div id="app"></div>

  <script type="module">
    const bus = document.querySelector('pan-bus');

    // Subscribe to list state
    bus.subscribe('todos.list.state', (msg) => {
      const todos = msg.data.items;
      renderTodoList(todos);
    });

    // Fetch initial list
    bus.publish('todos.list.get', {});

    function renderTodoList(todos) {
      const app = document.getElementById('app');
      app.innerHTML = `
        <h1>Todo List</h1>
        <ul>
          ${todos.map(todo => `
            <li>
              ${todo.title}
              <button onclick="completeTodo(${todo.id})">Done</button>
              <button onclick="deleteTodo(${todo.id})">Delete</button>
            </li>`
          )}
        </ul>
      `;
    }
  </script>
```

```

        `).join('')}
    </ul>
    <form onsubmit="addTodo(event)">
      <input type="text" id="newTodo" placeholder="New todo...">
      <button type="submit">Add</button>
    </form>
  `;
}

window.addTodo = (event) => {
  event.preventDefault();
  const input = document.getElementById('newTodo');
  const title = input.value.trim();

  if (!title) return;

  bus.publish('todos.item.save', {
    item: { title, completed: false }
  });

  input.value = '';
};

window.completeTodo = (id) => {
  // Fetch current state, update, and save
  const unsub = bus.subscribe(`todos.item.state.${id}`, (msg) => {
    const todo = msg.data.item;
    bus.publish('todos.item.save', {
      item: { ...todo, completed: true }
    });
    unsub();
  }, { retained: true });

  bus.publish('todos.item.get', { id });
};

window.deleteTodo = (id) => {
  if (confirm('Delete this todo?')) {
    bus.publish('todos.item.delete', { id });
  }
};
</script>
</body>
</html>

```

24.1.7.2 Request-Response Pattern

For operations that need explicit confirmation:

```
const bus = document.querySelector('pan-bus');

async function saveUser(userData) {
  return new Promise((resolve, reject) => {
    const correlationId = `save-${Date.now()}`;
    const replyTo = `app.reply.${correlationId}`;

    // Subscribe to reply
    const unsub = bus.subscribe(replyTo, (msg) => {
      unsub();
      if (msg.data.ok) {
        resolve(msg.data.item);
      } else {
        reject(new Error(msg.data.error.body?.message || 'Save failed'));
      }
    });

    // Send request with reply routing
    bus.publish('users.item.save', {
      item: userData,
      replyTo,
      correlationId
    });
  });
}

// Usage
try {
  const savedUser = await saveUser({ name: 'Alice', email: 'alice@example.com' });
  console.log('User saved:', savedUser);
} catch (error) {
  console.error('Failed to save user:', error);
}
```

24.1.7.3 Query Parameters and Filtering

```
// Paginated list with filters
bus.publish('products.list.get', {
  page: 2,
  limit: 20,
  category: 'electronics',
  minPrice: 100,
  maxPrice: 1000,
  sort: 'price:asc'
```

```
});
```

```
// The connector converts this to:
```

```
// GET /api/products?page=2&limit=20&category=electronics&minPrice=100&maxPrice=1000&sort=price
```

24.1.8 Related Components

- **pan-bus:** Required for message routing
- **pan-auth-provider:** Automatic authentication header injection
- **pan-store:** Can be used to cache connector state in memory
- **pan-idb:** Can persist connector state to IndexedDB for offline support

24.1.9 Common Issues and Solutions

24.1.9.1 Issue: CORS Errors

Symptom: Browser console shows “Access-Control-Allow-Origin” errors.

Solution: Configure your server to include proper CORS headers, or use a proxy during development:

```
// Development proxy in Vite config
export default {
  server: {
    proxy: {
      '/api': {
        target: 'https://api.example.com',
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, '')
      }
    }
  }
}
```

24.1.9.2 Issue: Stale Data After Updates

Symptom: List doesn’t reflect changes after creating/updating items.

Solution: The connector automatically refreshes the list after save/delete operations. If you need manual refresh:

```
bus.publish('users.list.get', {});
```

24.1.9.3 Issue: 401 Unauthorized Errors

Symptom: Requests fail with 401 status after initial success.

Solution: Ensure your auth token is being refreshed. The connector automatically picks up new tokens from `auth.internal.state`:

```
// When token is refreshed
bus.publish('auth.internal.state', {
```

```

    authenticated: true,
    token: newToken,
    user: { id: 123, name: 'Alice' }
  }, { retain: true });

```

24.1.9.4 Issue: Slow Performance with Large Lists

Symptom: UI freezes when loading large datasets.

Solution: Implement pagination and avoid loading all items at once:

```

// Load in pages
const PAGE_SIZE = 50;
let currentPage = 1;

function loadNextPage() {
  bus.publish('items.list.get', {
    page: currentPage,
    limit: PAGE_SIZE
  });
  currentPage++;
}

```

24.2 pan-graphql-connector

24.2.1 Overview

pan-graphql-connector bridges LARC's PAN bus to GraphQL APIs. It maps the same CRUD topic patterns as pan-data-connector but executes GraphQL queries and mutations instead of REST calls. You define your GraphQL operations as child `<script>` elements, and the connector handles execution, response parsing, and state publication.

This component is ideal for applications that interact with GraphQL APIs while maintaining architectural consistency with REST-based LARC applications.

24.2.2 When to Use

Use pan-graphql-connector when:

- Your backend uses GraphQL instead of REST
- You want to leverage GraphQL's flexible query structure
- You need to fetch nested or related data in a single request
- Your API benefits from GraphQL's type system and introspection
- You're building against existing GraphQL services (GitHub, Shopify, etc.)

Don't use pan-graphql-connector when:

- Your backend uses REST (use pan-data-connector)
- You need real-time subscriptions (use pan-websocket with GraphQL subscription protocol)

- Your queries are so dynamic that templating won't work (write custom GraphQL clients)

24.2.3 Installation and Setup

```
<!DOCTYPE html>
<html>
<head>
  <script type="module" src="/ui/pan-bus.mjs"></script>
  <script type="module" src="/ui/pan-graphql-connector.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <pan-graphql-connector
    resource="users"
    endpoint="https://api.example.com/graphql"
    key="id">

    <!-- List query -->
    <script type="application/graphql" data-op="list">
      query GetUsers($limit: Int, $offset: Int) {
        users(limit: $limit, offset: $offset) {
          id
          name
          email
          createdAt
        }
      }
    </script>

    <!-- Single item query -->
    <script type="application/graphql" data-op="item">
      query GetUser($id: ID!) {
        user(id: $id) {
          id
          name
          email
          createdAt
          posts {
            id
            title
          }
        }
      }
    </script>

    <!-- Save mutation -->
```

```

<script type="application/graphql" data-op="save">
  mutation SaveUser($id: ID, $item: UserInput!) {
    saveUser(id: $id, input: $item) {
      id
      name
      email
      createdAt
    }
  }
</script>

<!-- Delete mutation -->
<script type="application/graphql" data-op="delete">
  mutation DeleteUser($id: ID!) {
    deleteUser(id: $id)
  }
</script>

<!-- Response path mapping -->
<script type="application/json" data-paths>
  {
    "list": "data.users",
    "item": "data.user",
    "save": "data.saveUser",
    "delete": "data.deleteUser"
  }
</script>
</pan-graphql-connector>
</body>
</html>

```

24.2.4 Attributes

Attribute	Type	Default	Description
resource	String	"items"	Logical resource name for topic prefixes.
endpoint	String	Required	GraphQL HTTP endpoint URL.
key	String	"id"	Field name used as the unique identifier.

24.2.5 GraphQL Operation Scripts

Define GraphQL operations as child `<script type="application/graphql">` elements:

`data-op="list"`

Executed when `${resource}.list.get` is published. Variables from the message data are passed to the query.

data-op="item"

Executed when `${resource}.item.get` is published. Receives `{ id }` as a variable.

data-op="save"

Executed when `${resource}.item.save` is published. Receives `{ id, item }` as variables (id is null for creation).

data-op="delete"

Executed when `${resource}.item.delete` is published. Receives `{ id }` as a variable.

24.2.6 Response Path Mapping

The `<script type="application/json" data-paths>` element maps GraphQL response paths to data:

```
{
  "list": "data.users",          // Path to array in list response
  "item": "data.user",          // Path to object in item response
  "save": "data.saveUser",      // Path to object in save response
  "delete": "data.deleteUser"   // Path to boolean/success indicator in delete response
}
```

Without path mapping, the connector attempts to extract data from the top-level `data` field.

24.2.7 Topics

The topic structure is identical to `pan-data-connector`:

- **Listens:** `${resource}.list.get`, `${resource}.item.get`, `${resource}.item.save`, `${resource}.item.delete`
- **Publishes:** `${resource}.list.state`, `${resource}.item.state.${id}`
- **Reply support:** Same as `pan-data-connector`

24.2.8 Authentication Integration

Like `pan-data-connector`, this component subscribes to `auth.internal.state` and automatically injects `Authorization: Bearer ${token}` headers.

24.2.9 Complete Examples

24.2.9.1 GitHub API Integration

```
<pan-graphql-connector
  resource="repos"
  endpoint="https://api.github.com/graphql"
  key="id">
```

```

<script type="application/graphql" data-op="list">
  query GetRepositories($login: String!) {
    user(login: $login) {
      repositories(first: 20, orderBy: {field: UPDATED_AT, direction: DESC}) {
        nodes {
          id
          name
          description
          url
          stargazerCount
          updatedAt
        }
      }
    }
  }
</script>

<script type="application/json" data-paths>
  {
    "list": "data.user.repositories.nodes"
  }
</script>
</pan-graphql-connector>

<script type="module">
  const bus = document.querySelector('pan-bus');

  // Fetch repositories for a user
  bus.publish('repos.list.get', { login: 'torvalds' });

  bus.subscribe('repos.list.state', (msg) => {
    console.log('Repositories:', msg.data.items);
  });
</script>

```

24.2.9.2 Nested Data Fetching

```

<pan-graphql-connector
  resource="posts"
  endpoint="/graphql"
  key="id">

  <script type="application/graphql" data-op="item">
    query GetPost($id: ID!) {
      post(id: $id) {
        id
        title
      }
    }
  </script>

```

```

        content
        author {
            id
            name
            avatar
        }
        comments {
            id
            text
            author {
                name
            }
            createdAt
        }
        tags
    }
}
</script>

<script type="application/json" data-paths>
  { "item": "data.post" }
</script>
</pan-graphql-connector>

```

24.2.10 Related Components

- **pan-bus**: Required for message routing
- **pan-data-connector**: REST equivalent
- **pan-auth-provider**: Automatic token injection
- **pan-websocket**: For GraphQL subscriptions over WebSocket

24.2.11 Common Issues and Solutions

24.2.11.1 Issue: GraphQL Errors Not Surfaced

Symptom: Requests fail silently without clear error messages.

Solution: GraphQL returns errors in the `errors` array. The connector concatenates error messages. Check browser console for details:

```

bus.subscribe('users.list.state', (msg) => {
  if (msg.data.items.length === 0) {
    console.warn('Empty result-check console for GraphQL errors');
  }
});

```

24.2.11.2 Issue: Response Path Incorrect

Symptom: Published state is empty even though GraphQL response contains data.

Solution: Verify your path mapping matches the response structure. Use browser DevTools Network tab to inspect the actual GraphQL response:

```
// Response:
{
  "data": {
    "viewer": {
      "repositories": [...]
    }
  }
}

// Correct path:
{
  "list": "data.viewer.repositories"
}
```

24.2.11.3 Issue: Variables Not Passed Correctly

Symptom: GraphQL complains about missing required variables.

Solution: Ensure the query variable names match what you're passing in the PAN message:

```
// Query expects $limit
query GetItems($limit: Int) { ... }

// Pass correct variable name
bus.publish('items.list.get', { limit: 50 });
```

24.3 pan-websocket

24.3.1 Overview

`pan-websocket` creates a bidirectional bridge between LARC's PAN bus and WebSocket servers. It forwards PAN messages to the WebSocket connection and publishes incoming WebSocket messages to the PAN bus. The component handles connection lifecycle, automatic reconnection with exponential backoff, heartbeat pings, and topic-based message filtering.

This enables real-time, full-duplex communication patterns: chat applications, live collaboration, gaming, IoT dashboards, and any scenario where both client and server need to push messages at will.

24.3.2 When to Use

Use `pan-websocket` when:

- You need bidirectional real-time communication
- Both client and server need to initiate messages
- You're building chat, collaboration, or multiplayer features

- You need lower latency than HTTP polling or SSE
- Your server supports WebSocket protocol

Don't use `pan-websocket` when:

- You only need server-to-client updates (use `pan-sse` for simplicity)
- Your infrastructure doesn't support WebSocket (some proxies block them)
- You're working with simple request-response patterns (use `pan-data-connector`)
- You need guaranteed message delivery and ordering (WebSocket doesn't guarantee these; consider adding application-level acknowledgment)

24.3.3 Installation and Setup

```
<!DOCTYPE html>
<html>
<head>
  <script type="module" src="/ui/pan-bus.mjs"></script>
  <script type="module" src="/ui/pan-websocket.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <pan-websocket
    url="wss://api.example.com/ws"
    outbound-topics="chat.* user.typing"
    inbound-topics="chat.* user.* system.*"
    auto-reconnect="true"
    reconnect-delay="1000,15000"
    heartbeat="30"
    heartbeat-topic="sys.ping">
  </pan-websocket>
</body>
</html>
```

24.3.4 Attributes

Attribute	Type	Default	Description
<code>url</code>	String	Required	WebSocket server URL (must start with <code>ws://</code> or <code>wss://</code>).
<code>protocols</code>	String	""	Comma-separated list of WebSocket subprotocols.
<code>outbound-topics</code>	String	""	Space-separated topic patterns to forward from PAN bus to WebSocket. Empty means no topics are forwarded.

Attribute	Type	Default	Description
<code>inbound-topics</code>	String	<code>"*"</code>	Space-separated topic patterns to publish from WebSocket to PAN bus. Default <code>"*"</code> publishes all.
<code>auto-reconnect</code>	Boolean	<code>true</code>	Enable automatic reconnection on disconnect.
<code>reconnect-delay</code>	String	<code>"1000,15000"</code>	Min and max reconnection delay in milliseconds (exponential backoff).
<code>heartbeat</code>	Number	<code>30</code>	Seconds between heartbeat ping messages. Set to 0 to disable.
<code>heartbeat-topic</code>	String	<code>"sys.ping"</code>	Topic used for heartbeat messages.

Example configurations:

```

<!-- Chat application -->
<pan-websocket
  url="wss://chat.example.com"
  outbound-topics="chat.send user.typing"
  inbound-topics="chat.* presence.*">
</pan-websocket>

<!-- IoT dashboard -->
<pan-websocket
  url="wss://iot.example.com/devices"
  outbound-topics="device.command.*"
  inbound-topics="sensor.* device.status.*"
  heartbeat="10">
</pan-websocket>

<!-- Authentication with token -->
<pan-websocket
  url="wss://api.example.com/ws?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
  outbound-topics="*"
  inbound-topics="*">
</pan-websocket>

```

24.3.5 Topics

The connector publishes system lifecycle events:

`ws.connected`

Published when WebSocket connection is established.

Payload:

```
{
  url: "wss://api.example.com/ws",
  timestamp: 1704444000000
}
```

ws.disconnected

Published when connection closes.

Payload:

```
{
  code: 1000,           // WebSocket close code
  reason: "Normal closure",
  wasClean: true,       // Whether close was clean
  timestamp: 1704444100000
}
```

ws.error

Published when connection error occurs.

Payload:

```
{
  error: "Connection refused",
  timestamp: 1704444050000
}
```

ws.message

Published for every incoming WebSocket message (before topic-specific publishing).

Payload:

```
{
  message: { topic: "chat.message", data: {...} },
  timestamp: 1704444075000
}
```

Or for non-JSON messages:

```
{
  raw: "plain text message",
  timestamp: 1704444075000
}
```

24.3.6 Methods

Access the element to call methods programmatically:

```
const ws = document.querySelector('pan-websocket');
```

send(data)

Sends data directly through the WebSocket connection.

Parameters:

- `data` (String | Object): Data to send. Objects are JSON-stringified automatically.

```
ws.send({ topic: 'custom.event', data: { foo: 'bar' } });
ws.send('plain text message');
```

`close()`

Closes the WebSocket connection and disables auto-reconnect.

```
ws.close();
```

`reconnect()`

Manually triggers reconnection (closes current connection and establishes new one).

```
ws.reconnect();
```

24.3.7 Message Format

Messages sent over WebSocket should follow this JSON structure:

```
{
  topic: "event.name",
  data: { /* payload */ },
  ts: 1704444000000,    // Optional timestamp
  id: "msg-123"         // Optional message ID
}
```

The connector:

1. Checks if `message.topic` matches any `inbound-topics` patterns
2. Publishes to PAN bus as: `{ topic: message.topic, data: message.data, retain: message.retain }`

Outbound messages are forwarded in the same format.

24.3.8 Complete Examples

24.3.8.1 Real-time Chat Application

```
<!DOCTYPE html>
<html>
<head>
  <script type="module" src="/ui/pan-bus.mjs"></script>
  <script type="module" src="/ui/pan-websocket.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <pan-websocket
```

```
url="wss://chat.example.com/room/general"
outbound-topics="chat.send user.typing"
inbound-topics="chat.* user.* presence.*">
</pan-websocket>

<div id="chat">
  <div id="messages"></div>
  <div id="typing"></div>
  <form id="chatForm">
    <input type="text" id="messageInput" placeholder="Type a message...">
    <button type="submit">Send</button>
  </form>
</div>

<script type="module">
  const bus = document.querySelector('pan-bus');
  const messagesDiv = document.getElementById('messages');
  const typingDiv = document.getElementById('typing');
  const form = document.getElementById('chatForm');
  const input = document.getElementById('messageInput');

  // Subscribe to incoming messages
  bus.subscribe('chat.message', (msg) => {
    const { user, text, timestamp } = msg.data;
    appendMessage(user, text, timestamp);
  });

  // Subscribe to typing indicators
  bus.subscribe('user.typing', (msg) => {
    const { user, isTyping } = msg.data;
    updateTypingIndicator(user, isTyping);
  });

  // Subscribe to connection status
  bus.subscribe('ws.connected', () => {
    console.log('Chat connected');
  });

  bus.subscribe('ws.disconnected', () => {
    console.warn('Chat disconnected');
  });

  // Send message on form submit
  form.addEventListener('submit', (e) => {
    e.preventDefault();
    const text = input.value.trim();
```

```

    if (!text) return;

    bus.publish('chat.send', {
      text,
      user: getCurrentUser(),
      timestamp: Date.now()
    });

    input.value = '';
  });

  // Send typing indicator
  let typingTimeout;
  input.addEventListener('input', () => {
    bus.publish('user.typing', {
      user: getCurrentUser(),
      isTyping: true
    });

    clearTimeout(typingTimeout);
    typingTimeout = setTimeout(() => {
      bus.publish('user.typing', {
        user: getCurrentUser(),
        isTyping: false
      });
    }, 2000);
  });

  function appendMessage(user, text, timestamp) {
    const time = new Date(timestamp).toLocaleTimeString();
    messagesDiv.innerHTML += `
      <div class="message">
        <strong>${user}</strong>
        <span class="time">${time}</span>
        <p>${text}</p>
      </div>
    `;
    messagesDiv.scrollTop = messagesDiv.scrollHeight;
  }

  function updateTypingIndicator(user, isTyping) {
    if (isTyping) {
      typingDiv.textContent = `${user} is typing...`;
    } else {
      typingDiv.textContent = '';
    }
  }
}

```

```

    function getCurrentUser() {
        return localStorage.getItem('username') || 'Anonymous';
    }
</script>
</body>
</html>

```

24.3.8.2 IoT Sensor Dashboard

```

<pan-websocket
  url="wss://iot.example.com/stream"
  outbound-topics="device.control.*"
  inbound-topics="sensor.* device.status.*"
  heartbeat="10">
</pan-websocket>

<script type="module">
  const bus = document.querySelector('pan-bus');

  // Subscribe to all sensor updates
  bus.subscribe('sensor.*', (msg) => {
    const { deviceId, sensorType, value, unit } = msg.data;
    updateSensorDisplay(deviceId, sensorType, value, unit);
  });

  // Subscribe to device status
  bus.subscribe('device.status.*', (msg) => {
    const { deviceId, online, battery } = msg.data;
    updateDeviceStatus(deviceId, online, battery);
  });

  // Control device
  function controlDevice(deviceId, action) {
    bus.publish(`device.control.${deviceId}`, {
      action,
      timestamp: Date.now()
    });
  }

  // Example: Turn on device
  controlDevice('device-001', 'power:on');
</script>

```

24.3.9 Related Components

- **pan-bus**: Required for message routing
- **pan-sse**: For unidirectional server-to-client streaming

- **pan-data-connector:** For request-response HTTP patterns
- **pan-graphql-connector:** For GraphQL over WebSocket subscriptions

24.3.10 Common Issues and Solutions

24.3.10.1 Issue: Connection Keeps Dropping

Symptom: `ws.disconnected` events happen frequently.

Solution: 1. Check server-side WebSocket timeout configuration 2. Reduce `heartbeat` interval to keep connection alive 3. Verify firewall/proxy doesn't block WebSocket

```
<pan-websocket
  url="wss://api.example.com/ws"
  heartbeat="15"
  reconnect-delay="500,5000">
</pan-websocket>
```

24.3.10.2 Issue: Messages Not Being Forwarded

Symptom: Messages published to PAN don't appear on WebSocket.

Solution: Ensure topics match `outbound-topics` patterns:

```
<!-- Only forwards topics starting with "app." -->
<pan-websocket
  url="wss://api.example.com/ws"
  outbound-topics="app.*">
</pan-websocket>
```

```
// This WILL be forwarded
bus.publish('app.user.update', { id: 123 });

// This will NOT be forwarded
bus.publish('other.event', { data: 'ignored' });
```

24.3.10.3 Issue: Reconnection Storms

Symptom: Many reconnection attempts happen too quickly, overwhelming server.

Solution: Increase minimum reconnection delay and maximum backoff:

```
<pan-websocket
  url="wss://api.example.com/ws"
  reconnect-delay="5000,60000">
</pan-websocket>
```

This uses exponential backoff from 5 seconds to 60 seconds maximum.

24.4 pan-sse

24.4.1 Overview

`pan-sse` bridges Server-Sent Events (SSE) streams to LARC's PAN bus. It opens an `EventSource` connection, listens for server events, and publishes them as PAN messages. Unlike `WebSocket`, SSE is unidirectional (server to client) but simpler to implement, works over standard HTTP, and automatically reconnects on failure.

SSE is ideal for live feeds, notification streams, real-time dashboards, and any scenario where the server pushes updates but the client only sends occasional HTTP requests.

24.4.2 When to Use

Use `pan-sse` when:

- You only need server-to-client real-time updates
- Your infrastructure doesn't support `WebSocket`
- You want automatic reconnection built into the browser
- Your server can stream `text/event-stream` responses
- You're building live feeds, notifications, or monitoring dashboards

Don't use `pan-sse` when:

- You need bidirectional communication (use `pan-websocket`)
- You need to send frequent client-to-server messages (SSE doesn't support that)
- Your server doesn't support persistent HTTP connections
- You need binary data streaming (SSE is text-only)

24.4.3 Installation and Setup

```
<!DOCTYPE html>
<html>
<head>
  <script type="module" src="/ui/pan-bus.mjs"></script>
  <script type="module" src="/ui/pan-sse.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <pan-sse
    src="/api/events"
    topics="user.* system.notification"
    with-credentials="true"
    persist-last-event="app-events"
    backoff="1000,10000">
  </pan-sse>
</body>
</html>
```

24.4.4 Attributes

Attribute	Type	Default	Description
<code>src</code>	String	Required	SSE endpoint URL. Must return <code>Content-Type: text/event-stream</code> .
<code>topics</code>	String	""	Space-separated list of topics to subscribe to. Appended as <code>?topics=topic1,topic2</code> .
<code>with-credentials</code>	Boolean	<code>true</code>	Include credentials (cookies) with EventSource request.
<code>persist-last-event</code>	String	""	localStorage key to persist last event ID. On reconnect, sends <code>?lastEventId=...</code> to resume stream.
<code>backoff</code>	String	<code>"1000,15000"</code>	Min and max reconnection delay in milliseconds. Uses jittered exponential backoff.

Example configurations:

```

<!-- Live notification feed -->
<pan-sse
  src="https://api.example.com/notifications"
  topics="notification.new"
  persist-last-event="notifications">
</pan-sse>

<!-- Stock price updates -->
<pan-sse
  src="/api/stocks/stream"
  topics="stock.price stock.trade"
  with-credentials="false">
</pan-sse>

<!-- Server monitoring -->
<pan-sse
  src="https://monitor.example.com/events"
  topics="server.* alert.*"
  persist-last-event="monitoring"
  backoff="2000,30000">
</pan-sse>

```

24.4.5 Server-Side SSE Format

Your server should send events in this format:

`Content-Type: text/event-stream`

Cache-Control: no-cache

Connection: keep-alive

id: 123

event: notification.new

data: {"userId":456,"message":"You have a new message"}

id: 124

event: user.login

data: {"userId":789,"timestamp":1704444000000}

id: 125

data: {"topic":"system.status","status":"ok"}

- **id**: Optional event ID for resumption (used with `persist-last-event`)
- **event**: Optional event type (becomes PAN topic if provided)
- **data**: Event payload (JSON parsed automatically)

If no `event` field is provided, the connector looks for `topic` in the JSON data.

24.4.6 Topics

The connector doesn't publish system lifecycle events by default. It simply forwards server events to PAN bus topics.

Events are published as:

```
{
  topic: eventType || data.topic,
  data: data.data || data.payload || data,
  retain: data.retain || false
}
```

24.4.7 Complete Examples

24.4.7.1 Live Notification Feed

Server (Node.js Express):

```
app.get('/api/notifications', (req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  // Send initial event
  res.write(`id: ${Date.now()}\n`);
  res.write(`event: notification.init\n`);
  res.write(`data: {"status":"connected"}\n\n`);

  // Subscribe to notification system
  const unsubscribe = notificationService.subscribe(notification) => {
```

```

    res.write(`id: ${notification.id}\n`);
    res.write(`event: notification.new\n`);
    res.write(`data: ${JSON.stringify(notification)}\n\n`);
  });

  // Cleanup on disconnect
  req.on('close', () => {
    unsubscribe();
    res.end();
  });
});

```

Client:

```

<pan-sse
  src="/api/notifications"
  persist-last-event="notifications">
</pan-sse>

<div id="notifications"></div>

<script type="module">
  const bus = document.querySelector('pan-bus');
  const container = document.getElementById('notifications');

  bus.subscribe('notification.new', (msg) => {
    const { userId, message, timestamp } = msg.data;

    const div = document.createElement('div');
    div.className = 'notification';
    div.innerHTML = `
      <span class="time">${new Date(timestamp).toLocaleTimeString()}</span>
      <p>${message}</p>
    `;

    container.prepend(div);

    // Auto-remove after 10 seconds
    setTimeout(() => div.remove(), 10000);
  });
</script>

```

24.4.7.2 Real-time Analytics Dashboard

Server:

```

app.get('/api/analytics/stream', (req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');

```

```

res.setHeader('Connection', 'keep-alive');

// Send metrics every 5 seconds
const interval = setInterval(() => {
  const metrics = {
    activeUsers: getActiveUserCount(),
    requestsPerSecond: getRequestRate(),
    errorRate: getErrorRate(),
    timestamp: Date.now()
  };

  res.write(`event: analytics.metrics\n`);
  res.write(`data: ${JSON.stringify(metrics)}\n\n`);
}, 5000);

req.on('close', () => {
  clearInterval(interval);
  res.end();
});
});

```

Client:

```

<pan-sse src="/api/analytics/stream"></pan-sse>

<script type="module">
  const bus = document.querySelector('pan-bus');

  bus.subscribe('analytics.metrics', (msg) => {
    const { activeUsers, requestsPerSecond, errorRate } = msg.data;

    updateChart('users', activeUsers);
    updateChart('requests', requestsPerSecond);
    updateChart('errors', errorRate);
  });
</script>

```

24.4.7.3 Multi-Topic Subscription

```

<pan-sse
  src="/api/events"
  topics="user.login user.logout order.created order.shipped"
  persist-last-event="app-events">
</pan-sse>

<script type="module">
  const bus = document.querySelector('pan-bus');

```

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

// Subscribe to order events
bus.subscribe('order.*', (msg) => {
  console.log('Order event:', msg.topic, msg.data);
});
</script>
```

The topics attribute sends `?topics=user.login,user.logout,order.created,order.shipped` to the server, allowing it to filter events before streaming.

24.4.8 Related Components

- **pan-bus**: Required for message routing
- **pan-websocket**: For bidirectional communication
- **pan-data-connector**: For request-response patterns
- **pan-store**: Can cache streamed data in memory

24.4.9 Common Issues and Solutions

24.4.9.1 Issue: EventSource Connection Fails Silently

Symptom: No events received, no error messages.

Solution: Check server CORS headers and Content-Type:

```
// Server must include:
res.setHeader('Content-Type', 'text/event-stream');
res.setHeader('Cache-Control', 'no-cache');
res.setHeader('Connection', 'keep-alive');
res.setHeader('Access-Control-Allow-Origin', 'https://your-client.com');
res.setHeader('Access-Control-Allow-Credentials', 'true');
```

24.4.9.2 Issue: Lost Events After Reconnect

Symptom: Gaps in data stream after network interruption.

Solution: Use `persist-last-event` and implement server-side event replay:

```
<pan-sse
  src="/api/events"
  persist-last-event="events">
</pan-sse>
```

Server checks `?lastEventId` query parameter:

```
app.get('/api/events', (req, res) => {
  const lastEventId = req.query.lastEventId;
```

```

if (lastEventId) {
  // Replay missed events since lastEventId
  const missedEvents = getEventsSince(lastEventId);
  missedEvents.forEach(event => {
    res.write(`id: ${event.id}\n`);
    res.write(`data: ${JSON.stringify(event)}\n\n`);
  });
}

// Continue with live stream
// ...
});

```

24.4.9.3 Issue: Memory Leak from Long-Running Streams

Symptom: Browser memory usage grows over time.

Solution: The component automatically handles cleanup on disconnect. Ensure you're not accumulating DOM nodes in your subscription handlers:

```

bus.subscribe('analytics.metrics', (msg) => {
  // BAD: Keeps appending without limit
  container.innerHTML += `<div>${msg.data.value}</div>`;

  // GOOD: Limit number of displayed items
  const div = document.createElement('div');
  div.textContent = msg.data.value;
  container.prepend(div);

  // Keep only last 100 items
  while (container.children.length > 100) {
    container.lastChild.remove();
  }
});

```

24.5 Architectural Patterns

24.5.1 Combining Multiple Connectors

Real-world applications often use multiple integration patterns simultaneously. Here's how to orchestrate them effectively:

```

<!DOCTYPE html>
<html>
<head>
  <script type="module" src="/ui/pan-bus.mjs"></script>
  <script type="module" src="/ui/pan-auth-provider.mjs"></script>

```

```

<script type="module" src="/ui/pan-data-connector.mjs"></script>
<script type="module" src="/ui/pan-websocket.mjs"></script>
<script type="module" src="/ui/pan-sse.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <!-- Authentication -->
  <pan-auth-provider
    storage="local"
    token-key="app_token">
  </pan-auth-provider>

  <!-- REST API for CRUD operations -->
  <pan-data-connector
    resource="documents"
    base-url="https://api.example.com">
  </pan-data-connector>

  <!-- WebSocket for real-time collaboration -->
  <pan-websocket
    url="wss://collab.example.com/ws"
    outbound-topics="document.edit.*"
    inbound-topics="document.edit.* user.cursor.*">
  </pan-websocket>

  <!-- SSE for notifications -->
  <pan-sse
    src="https://api.example.com/notifications"
    topics="notification.*"
    persist-last-event="notifications">
  </pan-sse>

  <div id="app"></div>

  <script type="module">
    const bus = document.querySelector('pan-bus');

    // Load initial document list via REST
    bus.publish('documents.list.get', { limit: 50 });

    // Subscribe to document state
    bus.subscribe('documents.list.state', (msg) => {
      renderDocumentList(msg.data.items);
    });

    // When user opens a document, subscribe to real-time edits

```

```

function openDocument(id) {
  bus.publish('documents.item.get', { id });

  bus.subscribe(`document.edit.${id}`, (msg) => {
    applyRemoteEdit(msg.data);
  });

  bus.subscribe(`user.cursor.${id}`, (msg) => {
    updateCursorPosition(msg.data.userId, msg.data.position);
  });
}

// When user edits locally, broadcast via WebSocket
function handleLocalEdit(documentId, edit) {
  bus.publish(`document.edit.${documentId}`, {
    userId: getCurrentUserId(),
    edit,
    timestamp: Date.now()
  });
}

// Subscribe to notifications via SSE
bus.subscribe('notification.*', (msg) => {
  showNotification(msg.data);
});

// Save changes via REST when user stops editing
async function saveDocument(documentId, content) {
  return new Promise((resolve, reject) => {
    const correlationId = `save-${Date.now()}`;
    const replyTo = `app.reply.${correlationId}`;

    const unsub = bus.subscribe(replyTo, (msg) => {
      unsub();
      msg.data.ok ? resolve(msg.data.item) : reject(msg.data.error);
    });

    bus.publish('documents.item.save', {
      item: { id: documentId, content },
      replyTo,
      correlationId
    });
  });
}
</script>
</body>
</html>

```

This architecture uses:

- **REST** for durable state (document CRUD)
- **WebSocket** for ephemeral real-time updates (cursor positions, live edits)
- **SSE** for server-initiated notifications (comments, mentions)

Each protocol handles what it does best, unified through PAN topics.

24.5.2 Optimistic Updates with Rollback

When combining REST APIs with real-time updates, implement optimistic updates for better perceived performance:

```
class DocumentEditor {
  constructor() {
    this.bus = document.querySelector('pan-bus');
    this.pendingUpdates = new Map();
  }

  async updateDocument(id, changes) {
    const updateId = `update-${Date.now()}-${Math.random()}`;

    // Apply changes optimistically
    this.applyChangesLocally(id, changes);

    // Track pending update
    this.pendingUpdates.set(updateId, { id, changes });

    try {
      // Send to server
      await this.saveToServer(id, changes);

      // Success: remove from pending
      this.pendingUpdates.delete(updateId);
    } catch (error) {
      // Failure: rollback
      console.error('Update failed, rolling back:', error);
      this.rollbackChanges(id, changes);
      this.pendingUpdates.delete(updateId);

      throw error;
    }
  }

  applyChangesLocally(id, changes) {
    // Update local state immediately
    const event = new CustomEvent('document-updated', {
      detail: { id, changes }
    });
  }
}
```

```

    window.dispatchEvent(event);
  }

  rollbackChanges(id, changes) {
    // Revert local state
    const event = new CustomEvent('document-rollback', {
      detail: { id, changes }
    });
    window.dispatchEvent(event);
  }

  saveToServer(id, changes) {
    return new Promise((resolve, reject) => {
      const correlationId = `save-${Date.now()}`;
      const replyTo = `app.reply.${correlationId}`;

      const timeout = setTimeout(() => {
        unsub();
        reject(new Error('Save timeout'));
      }, 10000);

      const unsub = this.bus.subscribe(replyTo, (msg) => {
        clearTimeout(timeout);
        unsub();

        if (msg.data.ok) {
          resolve(msg.data.item);
        } else {
          reject(new Error(msg.data.error?.body?.message || 'Save failed'));
        }
      });

      this.bus.publish('documents.item.save', {
        item: { id, ...changes },
        replyTo,
        correlationId
      });
    });
  }
}

```

24.5.3 Conflict Resolution

When multiple users edit the same document simultaneously, conflicts arise. Here's a simple last-write-wins strategy with vector clocks:

```

class ConflictResolver {
  constructor(bus) {

```

```

    this.bus = bus;
    this.vectorClock = new Map();
  }

  handleRemoteEdit(documentId, edit) {
    const localVersion = this.vectorClock.get(documentId) || 0;
    const remoteVersion = edit.version || 0;

    if (remoteVersion > localVersion) {
      // Remote is newer: apply
      this.applyEdit(documentId, edit);
      this.vectorClock.set(documentId, remoteVersion);
    } else if (remoteVersion < localVersion) {
      // Local is newer: ignore
      console.log('Ignoring stale remote edit');
    } else {
      // Same version: conflict
      this.resolveConflict(documentId, edit);
    }
  }

  resolveConflict(documentId, remoteEdit) {
    // Strategy 1: Last-write-wins by timestamp
    const localTimestamp = this.getLocalTimestamp(documentId);
    if (remoteEdit.timestamp > localTimestamp) {
      this.applyEdit(documentId, remoteEdit);
    }

    // Strategy 2: Operational Transform (more complex)
    // Strategy 3: CRDT (Conflict-free Replicated Data Types)
    // Strategy 4: User-initiated merge
  }

  applyEdit(documentId, edit) {
    // Apply the edit to local state
    this.bus.publish(`document.local-update.${documentId}`, {
      edit,
      source: 'remote'
    });
  }

  getLocalTimestamp(documentId) {
    // Retrieve from local state
    return Date.now();
  }
}

```

24.5.4 Offline Support

Combine connectors with service workers and IndexedDB for offline-first applications:

```
// Service worker for offline caching
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('app-v1').then((cache) => {
      return cache.addAll([
        '/',
        '/ui/pan-bus.mjs',
        '/ui/pan-data-connector.mjs',
        '/app.js',
        '/styles.css'
      ]);
    })
  );
});

self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});

// Application code with offline queue
class OfflineQueue {
  constructor(bus) {
    this.bus = bus;
    this.queue = [];
    this.loadQueue();

    // Monitor connection status
    window.addEventListener('online', () => this.processQueue());
    window.addEventListener('offline', () => console.log('Offline mode'));

    // Intercept save operations when offline
    this.bus.subscribe('documents.item.save', (msg) => {
      if (!navigator.onLine) {
        this.queueOperation('save', msg.data);
        // Publish immediate optimistic success
        if (msg.replyTo) {
          this.bus.publish(msg.replyTo, {
            correlationId: msg.correlationId,
            data: { ok: true, item: msg.data.item, queued: true }
          });
        }
      }
    });
  }
}
```

```

    }
  }
});
}

queueOperation(type, data) {
  this.queue.push({ type, data, timestamp: Date.now() });
  this.saveQueue();
}

async processQueue() {
  console.log(`Processing ${this.queue.length} queued operations...`);

  while (this.queue.length > 0 && navigator.onLine) {
    const operation = this.queue[0];

    try {
      await this.executeOperation(operation);
      this.queue.shift();
      this.saveQueue();
    } catch (error) {
      console.error('Failed to process queued operation:', error);
      break;
    }
  }
}

async executeOperation(operation) {
  switch (operation.type) {
    case 'save':
      this.bus.publish('documents.item.save', operation.data);
      break;
    case 'delete':
      this.bus.publish('documents.item.delete', operation.data);
      break;
  }
}

saveQueue() {
  localStorage.setItem('offline-queue', JSON.stringify(this.queue));
}

loadQueue() {
  try {
    const data = localStorage.getItem('offline-queue');
    this.queue = data ? JSON.parse(data) : [];
  } catch {

```

```

    this.queue = [];
  }
}
}

```

24.5.5 Rate Limiting and Backpressure

When dealing with high-frequency WebSocket or SSE streams, implement rate limiting to prevent UI overload:

```

class RateLimitedSubscriber {
  constructor(bus, topic, handler, options = {}) {
    this.bus = bus;
    this.handler = handler;
    this.buffer = [];
    this.lastFlush = Date.now();
    this.flushInterval = options.interval || 100; // ms
    this.maxBatch = options.maxBatch || 50;

    this.unsub = bus.subscribe(topic, (msg) => {
      this.buffer.push(msg);

      // Flush if buffer is full
      if (this.buffer.length >= this.maxBatch) {
        this.flush();
      }
    });

    // Periodic flush
    this.timer = setInterval(() => this.flush(), this.flushInterval);
  }

  flush() {
    if (this.buffer.length === 0) return;

    const batch = this.buffer.splice(0, this.maxBatch);
    this.handler(batch);
    this.lastFlush = Date.now();
  }

  destroy() {
    clearInterval(this.timer);
    this.unsub();
    this.flush(); // Flush remaining
  }
}

// Usage

```

```
const subscriber = new RateLimitedSubscriber(
  bus,
  'sensor.temperature',
  (messages) => {
    console.log(`Received ${messages.length} temperature readings`);
    updateChart(messages);
  },
  { interval: 200, maxBatch: 100 }
);
```

24.5.6 Security Considerations

24.5.6.1 Content Security Policy

When using WebSocket and SSE, configure CSP headers:

```
<meta http-equiv="Content-Security-Policy" content="
  default-src 'self';
  connect-src 'self' wss://api.example.com https://api.example.com;
  script-src 'self' 'unsafe-inline';
">
```

24.5.6.2 Token Expiration Handling

Automatically refresh auth tokens before they expire:

```
class TokenRefreshManager {
  constructor(bus) {
    this.bus = bus;
    this.refreshTimer = null;

    // Subscribe to auth state
    bus.subscribe('auth.internal.state', (msg) => {
      if (msg.data.authenticated) {
        this.scheduleRefresh(msg.data.expiresAt);
      }
    }, { retained: true });
  }

  scheduleRefresh(expiresAt) {
    clearTimeout(this.refreshTimer);

    const now = Date.now();
    const expiresIn = expiresAt - now;
    const refreshIn = Math.max(0, expiresIn - 60000); // Refresh 1 min before expiry

    this.refreshTimer = setTimeout(() => {
      this.refreshToken();
    }, refreshIn);
  }
}
```

```

}

async refreshToken() {
  try {
    const response = await fetch('/api/auth/refresh', {
      method: 'POST',
      credentials: 'include'
    });

    const { token, expiresAt } = await response.json();

    // Update auth state
    this.bus.publish('auth.internal.state', {
      authenticated: true,
      token,
      expiresAt
    }, { retain: true });
  } catch (error) {
    console.error('Token refresh failed:', error);
    // Redirect to login
    window.location.href = '/login';
  }
}
}

```

24.5.6.3 Input Validation

Always validate data from external sources:

```

function validateMessage(msg) {
  // Validate structure
  if (!msg || typeof msg !== 'object') {
    throw new Error('Invalid message structure');
  }

  if (typeof msg.topic !== 'string' || msg.topic.length === 0) {
    throw new Error('Missing or invalid topic');
  }

  // Validate data size
  const size = JSON.stringify(msg.data).length;
  if (size > 1048576) { // 1MB limit
    throw new Error('Message payload too large');
  }

  // Sanitize HTML if rendering user content
  if (msg.data.html) {

```

```

    msg.data.html = sanitizeHtml(msg.data.html);
  }

  return msg;
}

// Use in subscription handlers
bus.subscribe('chat.message', (msg) => {
  try {
    const validated = validateMessage(msg);
    displayChatMessage(validated.data);
  } catch (error) {
    console.error('Invalid message received:', error);
  }
});

```

24.6 Summary

LARC's integration components transform network protocols into PAN messages, maintaining architectural consistency across diverse data sources:

- **pan-data-connector** handles REST APIs with standard CRUD patterns
- **pan-graphql-connector** executes GraphQL queries and mutations
- **pan-websocket** enables bidirectional real-time communication
- **pan-sse** streams server-sent events for unidirectional updates

All four components:

- Operate declaratively via HTML attributes
- Publish retained state messages for late subscribers
- Support request-response patterns with `replyTo`
- Integrate with LARC's authentication system
- Handle connection lifecycle and error recovery

By combining these components strategically, you can build sophisticated applications that handle REST CRUD, real-time collaboration, server notifications, and offline support—all through a unified message bus architecture. The network complexity stays at the boundary, while your application logic remains focused on business concerns.

Choose the right connector for each data source, implement appropriate patterns for conflict resolution and offline support, and let LARC's integration layer handle the protocol details. Your components simply publish and subscribe to topics, blissfully unaware of whether their data travels over HTTP, WebSocket, or SSE.

Chapter 25

Utility Components

“The best debugging tool is still careful thought, coupled with judiciously placed print statements.”

— Brian Kernighan

Utility components are the unsung heroes of application development. They don’t render UI, manage state, or fetch data. Instead, they provide infrastructure: observability, message routing, cross-system integration. They’re the scaffolding that makes complex applications comprehensible and maintainable.

This chapter documents LARC’s utility components: **pan-debug** for message tracing and debugging, and **pan-forwarder** for HTTP message forwarding. These components help you understand what’s happening in your application and extend it beyond the browser.

25.1 Overview

LARC provides two utility components:

- **pan-debug**: Message tracing and debugging utilities for introspection
- **pan-forwarder**: HTTP message forwarding for server integration

These components operate at the infrastructure level. **pan-debug** observes message flows without altering them, while **pan-forwarder** bridges LARC’s in-browser message bus to external systems via HTTP. Together, they provide visibility into your application’s behavior and pathways to external integration.

25.2 pan-debug: Message Tracing and Debugging

25.2.1 Overview

pan-debug is not a custom element—it’s a JavaScript class (**PanDebugManager**) that provides introspection and debugging capabilities for PAN message flows. It tracks messages as they pass through the bus, records routing decisions, captures errors, and provides query capabilities for analysis.

Think of it as flight data recorder for your message bus. When something goes wrong—a message disappears, a route doesn’t fire, a handler throws an error—the trace buffer tells you exactly what happened.

25.2.2 When to Use

Use `pan-debug` when:

- Debugging routing issues or missing messages
- Profiling message throughput and patterns
- Investigating performance bottlenecks in message handling
- Building developer tools or admin dashboards
- Diagnosing production issues (with sampling to minimize overhead)

Don’t use `pan-debug` when:

- Building production applications (unless explicitly needed for diagnostics)
- Memory is constrained (trace buffer can grow large)
- You need real-time streaming of every message (sampling is more appropriate)

25.2.3 Installation and Setup

`pan-debug` is typically integrated into `pan-bus`. Access it via the global API after the bus is ready:

```
// Wait for bus readiness
await new Promise(resolve => {
  if (window.__panReady) resolve();
  else window.addEventListener('pan:sys.ready', resolve, { once: true });
});

// Access debug manager
const debug = window.pan.debug;
```

Alternatively, import the class directly:

```
import { PanDebugManager } from './pan-debug.mjs';

const debug = new PanDebugManager();
```

Enable tracing to start capturing messages:

```
// Enable with defaults (1000 message buffer, 100% sampling)
debug.enableTracing();

// Enable with custom configuration
debug.enableTracing({
  maxBuffer: 500,    // Keep last 500 messages
  sampleRate: 0.1   // Sample 10% of messages
});
```

25.2.4 API Reference

25.2.4.1 Constructor

```
new PanDebugManager()
```

Creates a new debug manager instance with default configuration:

- `enabled`: false
- `maxBuffer`: 1000
- `sampleRate`: 1.0
- `traceBuffer`: []
- `messageCount`: 0

25.2.4.2 `enableTracing(options)`

Enables message tracing with optional configuration.

Parameters: - `options` (Object, optional): Configuration object - `maxBuffer` (Number): Maximum messages to retain. Default: 1000 - `sampleRate` (Number): Sampling rate from 0.0 to 1.0. Default: 1.0

Returns: undefined

Example:

```
// Enable with full sampling
debug.enableTracing();

// Enable with limited buffer
debug.enableTracing({ maxBuffer: 100 });

// Enable with 25% sampling for production
debug.enableTracing({
  maxBuffer: 500,
  sampleRate: 0.25
});
```

25.2.4.3 `disableTracing()`

Disables message tracing. Logs the number of captured messages to the console.

Returns: undefined

Example:

```
debug.disableTracing();
// Console: "[PAN Debug] Tracing disabled (captured 347 messages)"
```

25.2.4.4 `trace(message, matchedRoutes)`

Records a message trace entry. Typically called by the bus routing system, not user code.

Parameters: - `message` (Object, required): Message object to trace - `matchedRoutes` (Array, optional): Array of route objects that matched this message. Default: []

Returns: undefined

Trace Entry Structure:

```
{
  message: {           // Sanitized message copy
    id: "msg-123",
    type: "user.login",
    topic: "user.login",
    ts: 1638360000000,
    data: { ... }
  },
  matchedRoutes: [ // Routes that matched
    {
      id: "route-1",
      name: "User Login Handler",
      actions: ["publish", "transform"],
      error: null
    }
  ],
  ts: 1638360000000, // Capture timestamp
  sequence: 347      // Message sequence number
}
```

25.2.4.5 `traceError(message, route, error)`

Records a routing error for a previously traced message.

Parameters: - `message` (Object, required): Message that caused the error - `route` (Object, required): Route that threw the error - `error` (Error, required): Error object

Returns: undefined

Example:

```
try {
  // Route action that throws
  executeRouteAction(message, route);
} catch (err) {
  debug.traceError(message, route, err);
  throw err;
}
```

25.2.4.6 `getTrace()`

Returns a copy of the trace buffer.

Returns: `Array<Object>` - Array of trace entries

Example:

```
const trace = debug.getTrace();
console.log(`Captured ${trace.length} messages`);

trace.forEach(entry => {
  console.log(`[${entry.sequence}] ${entry.message.topic}`, entry.message.data);
});
```

25.2.4.7 clearTrace()

Clears the trace buffer and resets message count.

Returns: undefined

Example:

```
debug.clearTrace();
// Console: "[PAN Debug] Trace buffer cleared"
```

25.2.4.8 getStats()

Returns statistics about the trace buffer.

Returns: Object with the following properties:

- **enabled** (Boolean): Whether tracing is enabled
- **messageCount** (Number): Total messages seen (including sampled)
- **bufferSize** (Number): Current number of entries in buffer
- **maxBuffer** (Number): Maximum buffer size
- **sampleRate** (Number): Current sampling rate
- **oldestMessage** (Number, optional): Timestamp of oldest message
- **newestMessage** (Number, optional): Timestamp of newest message
- **timespan** (Number, optional): Milliseconds between oldest and newest

Example:

```
const stats = debug.getStats();
console.log(`Tracing: ${stats.enabled ? 'ON' : 'OFF'}`);
console.log(`Buffer: ${stats.bufferSize} / ${stats.maxBuffer}`);
console.log(`Total processed: ${stats.messageCount}`);

if (stats.timespan) {
  console.log(`Timespan: ${((stats.timespan / 1000).toFixed(1))}s`);
}
```

25.2.4.9 query(filter)

Queries the trace buffer with filtering options.

Parameters: - **filter** (Object, optional): Filter criteria - **topic** (String): Match exact topic - **type** (String): Match exact message type - **hasRoutes** (Boolean): Filter by whether routes matched - **hasErrors** (Boolean): Filter by whether errors occurred - **startTs** (Number): Minimum timestamp

(milliseconds) - **endTs** (Number): Maximum timestamp (milliseconds) - **limit** (Number): Maximum results to return (returns most recent)

Returns: `Array<Object>` - Filtered trace entries

Example:

```
// Find all user.* messages
const userMessages = debug.query({ topic: 'user.login' });

// Find messages with no matched routes (dead letters)
const unrouted = debug.query({ hasRoutes: false });

// Find messages with errors
const errors = debug.query({ hasErrors: true });

// Find recent messages (last 10)
const recent = debug.query({ limit: 10 });

// Find messages in time range
const inRange = debug.query({
  startTs: Date.now() - 60000, // Last minute
  limit: 50
});

// Combine filters
const errorMessages = debug.query({
  hasErrors: true,
  startTs: Date.now() - 300000, // Last 5 minutes
  limit: 20
});
```

25.2.4.10 `export()`

Exports the trace buffer as formatted JSON.

Returns: `String` - JSON string with 2-space indentation

Example:

```
const json = debug.export();

// Save to file
const blob = new Blob([json], { type: 'application/json' });
const url = URL.createObjectURL(blob);
const a = document.createElement('a');
a.href = url;
a.download = `pan-trace-${Date.now()}.json`;
a.click();
```

25.2.4.11 import(json)

Imports a trace buffer from JSON. Replaces the current buffer.

Parameters: - `json` (String, required): JSON string from previous export

Returns: undefined

Throws: Error if JSON is invalid

Example:

```
// Load from file
const fileInput = document.querySelector('input[type="file"]');
fileInput.addEventListener('change', async (e) => {
  const file = e.target.files[0];
  const json = await file.text();

  try {
    debug.import(json);
    console.log('Trace imported successfully');
  } catch (err) {
    console.error('Failed to import trace:', err);
  }
});
```

25.2.5 Helper Functions

25.2.5.1 captureSnapshot(bus, routes, debug)

Creates a comprehensive snapshot of the current PAN state, including bus statistics, routing information, and debug status.

Parameters: - `bus` (Object, required): PAN bus instance - `routes` (Object, required): Routes manager instance - `debug` (Object, required): Debug manager instance

Returns: Object with the following structure:

```
{
  timestamp: 1638360000000,
  bus: {
    stats: {
      published: 1247,
      subscriptions: 23,
      // ... other bus stats
    },
    subscriptions: 23,
    retained: 15
  },
  routes: {
    routeCount: 8,
    activeRoutes: 6,
    // ... other route stats
  }
}
```

```

},
debug: {
  enabled: true,
  messageCount: 1247,
  bufferSize: 500,
  maxBuffer: 1000,
  sampleRate: 1.0
}
}

```

Example:

```

import { captureSnapshot } from './pan-debug.mjs';

// Capture current state
const snapshot = captureSnapshot(
  window.pan.bus,
  window.pan.routes,
  window.pan.debug
);

console.log('System snapshot:', snapshot);

// Store for later comparison
localStorage.setItem('pan-snapshot', JSON.stringify(snapshot));

```

25.2.6 Complete Working Examples

25.2.6.1 Example 1: Basic Debug Session

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module">
    import './pan-bus.mjs';

    // Wait for bus
    await new Promise(resolve => {
      window.addEventListener('pan:sys.ready', resolve, { once: true });
    });

    const debug = window.pan.debug;
    const bus = window.pan.bus;

    // Enable tracing
    debug.enableTracing({ maxBuffer: 50 });

```

```

    // Publish test messages
    bus.publish('user.login', { userId: '123' });
    bus.publish('user.profile', { name: 'Alice' });
    bus.publish('cart.add', { itemId: '456' });

    // Check trace
    console.log('Trace:', debug.getTrace());
    console.log('Stats:', debug.getStats());

    // Query for user messages
    const userMsgs = debug.query({
      topic: 'user.login'
    });
    console.log('User messages:', userMsgs);
  </script>
</head>
<body>
  <pan-bus></pan-bus>
</body>
</html>

```

25.2.6.2 Example 2: Production Sampling

```

// Enable lightweight tracing in production
class ProductionDebugger {
  constructor() {
    this.debug = window.pan.debug;

    // Sample 5% of messages, keep last 200
    this.debug.enableTracing({
      maxBuffer: 200,
      sampleRate: 0.05
    });

    // Periodically check for errors
    setInterval(() => this.checkForErrors(), 60000);
  }

  checkForErrors() {
    const errors = this.debug.query({
      hasErrors: true,
      startTs: Date.now() - 60000 // Last minute
    });

    if (errors.length > 0) {
      // Send to error tracking service
      this.reportErrors(errors);
    }
  }
}

```

```

    // Clear to prevent re-reporting
    this.debug.clearTrace();
  }
}

reportErrors(errors) {
  fetch('/api/errors', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      timestamp: Date.now(),
      errors: errors.map(e => ({
        message: e.message,
        route: e.matchedRoutes.find(r => r.error),
        error: e.matchedRoutes.find(r => r.error)?.error
      }))
    })
  });
}

// Initialize in production
if (window.location.hostname !== 'localhost') {
  new ProductionDebugger();
}

```

25.2.6.3 Example 3: Debug Dashboard

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <style>
    body { font: 14px/1.4 system-ui; margin: 20px; }
    .stats { display: grid; grid-template-columns: repeat(3, 1fr); gap: 16px; margin-bottom: 20px; }
    .card { border: 1px solid #ddd; border-radius: 8px; padding: 16px; }
    .trace-entry { border-left: 3px solid #4a90e2; padding: 8px; margin: 4px 0; background: #f9f9f9; }
    .error { border-left-color: #e74c3c; }
    button { padding: 8px 16px; margin: 4px; cursor: pointer; }
  </style>
</head>
<body>
  <h1>PAN Debug Dashboard</h1>

  <div class="stats">
    <div class="card">

```

```

    <h3>Status</h3>
    <div id="status"></div>
  </div>
  <div class="card">
    <h3>Buffer</h3>
    <div id="buffer"></div>
  </div>
  <div class="card">
    <h3>Timespan</h3>
    <div id="timespan"></div>
  </div>
</div>

<div>
  <button id="toggle">Enable Tracing</button>
  <button id="clear">Clear Buffer</button>
  <button id="export">Export JSON</button>
  <button id="errors">Show Errors</button>
  <button id="unrouted">Show Unrouted</button>
  <button id="all">Show All</button>
</div>

<div id="trace"></div>

<pan-bus></pan-bus>

<script type="module">
  await new Promise(resolve => {
    window.addEventListener('pan:sys.ready', resolve, { once: true });
  });

  const debug = window.pan.debug;
  const bus = window.pan.bus;

  // Update dashboard
  function updateStats() {
    const stats = debug.getStats();

    document.getElementById('status').innerHTML =
      `Enabled: ${stats.enabled ? '[v]' : '[x]'}<br>` +
      `Messages: ${stats.messageCount}`;

    document.getElementById('buffer').innerHTML =
      `Size: ${stats.bufferSize} / ${stats.maxBuffer}<br>` +
      `Sample: ${(stats.sampleRate * 100).toFixed(0)}%`;

    if (stats.timespan) {

```

```

    document.getElementById('timespan').innerHTML =
      `${(stats.timespan / 1000).toFixed(1)}s`;
  }
}

// Display trace entries
function displayTrace(entries) {
  const traceDiv = document.getElementById('trace');
  traceDiv.innerHTML = '';

  entries.forEach(entry => {
    const div = document.createElement('div');
    div.className = 'trace-entry';

    if (entry.matchedRoutes.some(r => r.error)) {
      div.classList.add('error');
    }

    div.innerHTML = `
      <strong>[${entry.sequence}] ${entry.message.topic}</strong><br>
      <pre>${JSON.stringify(entry.message.data, null, 2)}</pre>
      Routes: ${entry.matchedRoutes.length}
    `;

    traceDiv.appendChild(div);
  });

  updateStats();
}

// Controls
document.getElementById('toggle').onclick = () => {
  if (debug.getStats().enabled) {
    debug.disableTracing();
    document.getElementById('toggle').textContent = 'Enable Tracing';
  } else {
    debug.enableTracing({ maxBuffer: 100 });
    document.getElementById('toggle').textContent = 'Disable Tracing';
  }
  updateStats();
};

document.getElementById('clear').onclick = () => {
  debug.clearTrace();
  displayTrace([]);
};

```

```

document.getElementById('export').onclick = () => {
  const json = debug.export();
  const blob = new Blob([json], { type: 'application/json' });
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a');
  a.href = url;
  a.download = `pan-trace-${Date.now()}.json`;
  a.click();
};

document.getElementById('errors').onclick = () => {
  displayTrace(debug.query({ hasErrors: true }));
};

document.getElementById('unrouted').onclick = () => {
  displayTrace(debug.query({ hasRoutes: false }));
};

document.getElementById('all').onclick = () => {
  displayTrace(debug.getTrace());
};

// Enable by default
debug.enableTracing({ maxBuffer: 100 });
document.getElementById('toggle').textContent = 'Disable Tracing';

// Publish test messages
setInterval(() => {
  bus.publish('test.ping', { ts: Date.now() });
}, 2000);

// Update display every second
setInterval(updateStats, 1000);
updateStats();
</script>
</body>
</html>

```

25.2.7 Common Issues and Solutions

Issue: Trace buffer fills up quickly

Solution: Reduce maxBuffer or lower sampleRate:

```

debug.enableTracing({
  maxBuffer: 200,
  sampleRate: 0.1 // Only 10% of messages
});

```

Issue: Missing messages in trace

Cause: Sampling is active

Solution: Check the sample rate:

```
const stats = debug.getStats();
console.log(`Sample rate: ${stats.sampleRate}`);

// Increase to 100% for debugging
debug.enableTracing({ sampleRate: 1.0 });
```

Issue: Memory usage grows over time

Cause: Large trace buffer or high message volume

Solution: Use periodic cleanup:

```
// Clear trace every 5 minutes
setInterval(() => {
  debug.clearTrace();
}, 300000);
```

Issue: Cannot find specific message

Solution: Use query filters:

```
// Search by topic pattern
const results = debug.query({
  topic: 'user.login',
  startTs: Date.now() - 60000 // Last minute
});

if (results.length === 0) {
  console.log('No matching messages found');
  console.log('Total in buffer:', debug.getStats().bufferSize);
}
```

25.3 pan-forwarder: HTTP Message Forwarding

25.3.1 Overview

`pan-forwarder` is a custom element that forwards PAN messages to HTTP endpoints. It subscribes to topic patterns and POSTs each matching message to a configured destination, enabling integration with server-side systems, webhooks, and external APIs.

Think of it as a bridge between LARC's in-browser message bus and the wider world. Messages published to the PAN bus can trigger server-side actions, be logged to external systems, or forwarded to other clients via a hub.

25.3.2 When to Use

Use `pan-forwarder` when:

- Synchronizing local actions to a server (chat messages, collaborative editing)
- Logging client events to analytics or monitoring systems
- Triggering server-side workflows from browser actions
- Implementing server-sent events (SSE) with bidirectional flow
- Building multi-client synchronization

Don't use pan-forwarder when:

- You need request-response patterns (use `pan-xhr` or `fetch` directly)
- Messages contain sensitive data and your endpoint isn't secured
- You're forwarding high-frequency messages without throttling
- Creating message loops (forwarding + SSE on same topics)

25.3.3 Installation and Setup

```
<script type="module" src="/components/pan-forwarder.mjs"></script>

<pan-forwarder
  dest="https://api.example.com/events"
  topics="chat.* user.action.*"
  method="POST"
  headers='{ "Authorization": "Bearer token123" }'
  enabled="true">
</pan-forwarder>
```

25.3.4 Attributes

Attribute	Type	Default	Description
<code>dest</code>	String	<i>(required)</i>	Destination URL for HTTP requests. Must be a valid URL.
<code>topics</code>	String	<code>"*"</code>	Space-separated topic patterns to forward. Supports wildcards.
<code>method</code>	String	<code>"POST"</code>	HTTP method to use. Typically POST or PUT.
<code>headers</code>	String	<code>"{}"</code>	HTTP headers as JSON object or semicolon-separated key-value pairs.
<code>with-credentials</code>	Boolean	<code>true</code>	Whether to include credentials (cookies) in requests.
<code>enabled</code>	Boolean	<code>true</code>	Whether forwarding is active. Set to <code>"false"</code> or <code>"0"</code> to disable.

Attribute Details:

dest

Required. The HTTP endpoint that will receive forwarded messages.

```
<pan-forwarder dest="https://api.example.com/pan"></pan-forwarder>
```

topics

Space-separated list of topic patterns. Defaults to * (all messages).

```
<!-- Forward all user and admin messages -->
```

```
<pan-forwarder
  dest="/api/events"
  topics="user.* admin.*">
</pan-forwarder>
```

```
<!-- Forward specific topics -->
```

```
<pan-forwarder
  dest="/api/chat"
  topics="chat.message chat.typing">
</pan-forwarder>
```

method

HTTP method to use. Converted to uppercase.

```
<pan-forwarder dest="/api/events" method="PUT"></pan-forwarder>
```

headers

HTTP headers as JSON or semicolon-separated pairs:

```
<!-- JSON format -->
```

```
<pan-forwarder
  dest="/api/events"
  headers='{"Authorization": "Bearer abc123", "X-Client": "web"}'>
</pan-forwarder>
```

```
<!-- Semicolon-separated format -->
```

```
<pan-forwarder
  dest="/api/events"
  headers="Authorization: Bearer abc123; X-Client: web">
</pan-forwarder>
```

with-credentials

Controls whether cookies and authorization headers are included:

```
<!-- Include credentials (default) -->
```

```
<pan-forwarder dest="/api/events" with-credentials="true"></pan-forwarder>
```

```
<!-- Omit credentials for CORS requests -->
```

```
<pan-forwarder dest="https://other-domain.com/events" with-credentials="false"></pan-forwarder>
```

enabled

Controls whether forwarding is active:

```
<!-- Disabled -->
<pan-forwarder dest="/api/events" enabled="false"></pan-forwarder>

<!-- Enable/disable programmatically -->
<pan-forwarder id="fwd" dest="/api/events"></pan-forwarder>
<script>
  const fwd = document.getElementById('fwd');
  fwd.setAttribute('enabled', 'false'); // Disable
  fwd.setAttribute('enabled', 'true');  // Re-enable
</script>
```

25.3.5 Properties

Access configuration via JavaScript properties:

```
const forwarder = document.querySelector('pan-forwarder');

console.log(forwarder.dest);           // "https://api.example.com/events"
console.log(forwarder.topics);         // ["chat.*", "user.*"]
console.log(forwarder.method);         // "POST"
console.log(forwarder.headers);        // { Authorization: "Bearer ..." }
console.log(forwarder.withCredentials); // true
console.log(forwarder.enabled);        // true
```

25.3.6 Request Body Format

Each forwarded message is sent as JSON with the following structure:

```
{
  "topic": "chat.message",
  "data": {
    "user": "Alice",
    "text": "Hello world"
  },
  "retain": false,
  "id": "msg-123",
  "ts": 1638360000000
}
```

Fields: - **topic** (String): Message topic - **data** (Any): Message payload - **retain** (Boolean): Whether message was marked for retention - **id** (String, optional): Message ID if present - **ts** (Number, optional): Message timestamp if present

25.3.7 Deduplication

pan-forwarder implements best-effort deduplication using message IDs. If a message includes an id field, the forwarder tracks recently sent IDs to avoid duplicates.

The deduplication cache is cleared every 30 seconds to prevent unbounded growth.

25.3.8 Complete Working Examples

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/components/pan-bus.mjs"></script>
  <script type="module" src="/components/pan-forwarder.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <!-- Forward chat messages to server -->
  <pan-forwarder
    dest="/api/chat"
    topics="chat.message">
  </pan-forwarder>

  <input id="message" placeholder="Type a message">
  <button id="send">Send</button>

  <script type="module">
    import { PanClient } from '/components/pan-client.mjs';

    const pc = new PanClient();

    document.getElementById('send').onclick = () => {
      const text = document.getElementById('message').value;
      if (!text) return;

      // Publish locally (forwarder sends to server)
      pc.publish({
        topic: 'chat.message',
        data: {
          user: 'Alice',
          text: text,
          ts: Date.now()
        }
      });

      document.getElementById('message').value = '';
    };
  </script>
</body>
</html>
```

25.3.8.2 Example 2: Multi-Topic Forwarding with Headers

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/components/pan-bus.mjs"></script>
  <script type="module" src="/components/pan-forwarder.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <!-- Forward multiple topic patterns with auth -->
  <pan-forwarder
    dest="https://api.example.com/events"
    topics="user.* admin.* system.alert"
    headers='{ "Authorization": "Bearer abc123", "X-App": "dashboard" }'>
  </pan-forwarder>

  <script type="module">
    import { PanClient } from '/components/pan-client.mjs';

    const pc = new PanClient();

    // These will be forwarded
    pc.publish({ topic: 'user.login', data: { userId: '123' } });
    pc.publish({ topic: 'admin.action', data: { action: 'delete' } });
    pc.publish({ topic: 'system.alert', data: { level: 'critical' } });

    // This will NOT be forwarded (doesn't match patterns)
    pc.publish({ topic: 'ui.click', data: { button: 'submit' } });
  </script>
</body>
</html>

```

25.3.8.3 Example 3: SSE Integration (Bidirectional Sync)

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/components/pan-bus.mjs"></script>
  <script type="module" src="/components/pan-sse.mjs"></script>
  <script type="module" src="/components/pan-forwarder.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

```

```

<!-- Receive messages from server via SSE -->
<pan-sse
  src="/api/sse"
  topics="chat.message"
  persist-last-event="chat">
</pan-sse>

<!-- Forward local messages to server -->
<pan-forwarder
  dest="/api/chat"
  topics="chat.message">
</pan-forwarder>

<div id="messages"></div>
<input id="text" placeholder="Type a message">
<button id="send">Send</button>

<script type="module">
  import { PanClient } from '/components/pan-client.mjs';

  const pc = new PanClient();
  const messagesDiv = document.getElementById('messages');

  // Display incoming messages (from SSE or local)
  pc.subscribe('chat.message', (msg) => {
    const div = document.createElement('div');
    div.textContent = `${msg.data.user}: ${msg.data.text}`;
    messagesDiv.appendChild(div);
  });

  // Send message
  document.getElementById('send').onclick = () => {
    const text = document.getElementById('text').value;
    if (!text) return;

    // Publish locally
    // Forwarder sends to server
    // Server broadcasts via SSE to all clients
    pc.publish({
      topic: 'chat.message',
      data: {
        user: 'Current User',
        text: text,
        ts: Date.now()
      }
    });
  });

```

```

        document.getElementById('text').value = '';
    };
</script>
</body>
</html>

```

Important Note: When using `pan-forwarder` with `pan-sse`, be careful to avoid message loops. Forward write intents only (e.g., `chat.send`) rather than read events (e.g., `chat.message`), or ensure the server doesn't echo back messages from the same client.

25.3.8.4 Example 4: Conditional Forwarding

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/components/pan-bus.mjs"></script>
  <script type="module" src="/components/pan-forwarder.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <!-- Initially disabled -->
  <pan-forwarder
    id="forwarder"
    dest="/api/events"
    topics="user.*"
    enabled="false">
  </pan-forwarder>

  <label>
    <input type="checkbox" id="sync">
    Enable server sync
  </label>

  <script type="module">
    const forwarder = document.getElementById('forwarder');
    const checkbox = document.getElementById('sync');

    // Enable/disable based on checkbox
    checkbox.addEventListener('change', () => {
      forwarder.setAttribute('enabled', checkbox.checked);
    });
  </script>
</body>
</html>

```

25.3.8.5 Example 5: Analytics Forwarding

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script type="module" src="/components/pan-bus.mjs"></script>
  <script type="module" src="/components/pan-forwarder.mjs"></script>
</head>
<body>
  <pan-bus></pan-bus>

  <!-- Forward analytics events to tracking server -->
  <pan-forwarder
    dest="https://analytics.example.com/events"
    topics="analytics.*"
    headers='{ "X-API-Key": "your-api-key" }'>
  </pan-forwarder>

  <button id="btn1">Action 1</button>
  <button id="btn2">Action 2</button>

  <script type="module">
    import { PanClient } from '/components/pan-client.mjs';

    const pc = new PanClient();

    // Track button clicks
    document.getElementById('btn1').onclick = () => {
      pc.publish({
        topic: 'analytics.click',
        data: {
          button: 'action1',
          timestamp: Date.now(),
          page: window.location.pathname
        }
      });
    };

    document.getElementById('btn2').onclick = () => {
      pc.publish({
        topic: 'analytics.click',
        data: {
          button: 'action2',
          timestamp: Date.now(),
          page: window.location.pathname
        }
      });
    };
  </script>
</body>
</html>
```

```

    };

    // Track page views
    pc.publish({
      topic: 'analytics.pageview',
      data: {
        page: window.location.pathname,
        timestamp: Date.now(),
        referrer: document.referrer
      }
    });
  </script>
</body>
</html>

```

25.3.9 Server-Side Implementation

A typical server endpoint for receiving forwarded messages:

```

<?php
// /api/events endpoint (PHP example)
header('Content-Type: application/json');
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods: POST, OPTIONS');
header('Access-Control-Allow-Headers: Content-Type, Authorization');

if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
    exit(0);
}

$json = file_get_contents('php://input');
$message = json_decode($json, true);

if (!$message || !isset($message['topic'], $message['data'])) {
    http_response_code(400);
    echo json_encode(['error' => 'Invalid message format']);
    exit;
}

// Process message
$topic = $message['topic'];
$data = $message['data'];
$retain = $message['retain'] ?? false;
$id = $message['id'] ?? null;
$ts = $message['ts'] ?? time() * 1000;

// Log to database
$pdo = new PDO('sqlite:messages.db');

```

```

$stmt = $pdo->prepare('INSERT INTO messages (topic, data, ts) VALUES (?, ?, ?)');
$stmt->execute([$topic, json_encode($data), $ts]);

// Broadcast to other clients (if using SSE)
broadcast_message($topic, $data);

echo json_encode(['status' => 'ok']);

```

25.3.10 Related Components

pan-sse

Receives server-sent events and publishes them to the PAN bus. Often used in conjunction with `pan-forwarder` for bidirectional synchronization.

pan-xhr

Provides request-response HTTP patterns. Use for traditional API calls rather than one-way message forwarding.

pan-bus

The message bus that `pan-forwarder` subscribes to. Configure routing and message validation in `pan-bus`.

25.3.11 Common Issues and Solutions

Issue: Messages not being forwarded

Possible causes: 1. `enabled` attribute is false 2. Topic patterns don't match 3. No `dest` attribute 4. CORS errors (check browser console)

Solution:

```

const fwd = document.querySelector('pan-forwarder');

console.log('Enabled:', fwd.enabled);
console.log('Destination:', fwd.dest);
console.log('Topics:', fwd.topics);

// Check if topic matches
import { PanBusEnhanced } from '/components/pan-bus.mjs';
const matches = fwd.topics.some(pattern =>
  PanBusEnhanced.matches('your.topic', pattern)
);
console.log('Topic matches:', matches);

```

Issue: CORS errors when forwarding

Cause: Server doesn't allow cross-origin requests

Solution: Configure CORS headers on server:

```
// Server must respond with:
Access-Control-Allow-Origin: https://your-domain.com
Access-Control-Allow-Methods: POST, OPTIONS
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Allow-Credentials: true // If with-credentials="true"
```

Issue: Message loops with SSE

Cause: Forwarding the same topics that SSE publishes

Solution: Use different topic patterns:

```
<!-- BAD: Loop -->
<pan-sse src="/api/sse" topics="chat.message"></pan-sse>
<pan-forwarder dest="/api/chat" topics="chat.message"></pan-forwarder>

<!-- GOOD: Separate intent and state -->
<pan-sse src="/api/sse" topics="chat.message"></pan-sse>
<pan-forwarder dest="/api/chat" topics="chat.send"></pan-forwarder>

<!-- In your code, publish to chat.send instead of chat.message -->
<script>
  pc.publish({ topic: 'chat.send', data: { text: 'Hello' } });
</script>
```

Issue: High network traffic

Cause: Forwarding high-frequency messages without throttling

Solution: Throttle at the source:

```
class ThrottledPublisher {
  constructor(pc, interval = 100) {
    this.pc = pc;
    this.interval = interval;
    this.pending = null;
    this.timer = null;
  }

  publish(topic, data) {
    this.pending = { topic, data };

    if (!this.timer) {
      this.flush();
      this.timer = setInterval(() => this.flush(), this.interval);
    }
  }

  flush() {
    if (this.pending) {
      this.pc.publish(this.pending.topic, this.pending.data);
    }
  }
}
```

```

        this.pending = null;
    }
}

// Use throttled publisher for high-frequency events
const throttled = new ThrottledPublisher(pc, 100);

document.addEventListener('mousemove', (e) => {
    throttled.publish('mouse.move', { x: e.clientX, y: e.clientY });
});

```

Issue: Duplicate messages

Cause: Messages without IDs bypass deduplication

Solution: Include unique IDs in published messages:

```

import { generateId } from '/utils/id.mjs';

pc.publish({
    topic: 'chat.message',
    data: { text: 'Hello' },
    id: generateId() // Ensures deduplication
});

```

Issue: Authentication failures

Cause: Missing or incorrect headers

Solution: Verify headers are set correctly:

```

<pan-forwarder
  id="fwd"
  dest="/api/events"
  headers='{ "Authorization": "Bearer your-token" }'>
</pan-forwarder>

<script>
  const fwd = document.getElementById('fwd');
  console.log('Headers:', fwd.headers);
  // Should show: { Authorization: "Bearer your-token" }
</script>

```

Or set headers programmatically:

```

const fwd = document.querySelector('pan-forwarder');

// Get auth token
const token = localStorage.getItem('authToken');

// Update headers

```

```
fwd.setAttribute('headers', JSON.stringify({
  Authorization: `Bearer ${token}`
}));
```

25.4 Best Practices

25.4.1 Debug Component Best Practices

1. **Use sampling in production:** Full tracing has memory and performance overhead
2. **Clear buffers periodically:** Prevent unbounded memory growth
3. **Export traces before refresh:** Trace data is lost on page reload
4. **Query efficiently:** Use specific filters rather than scanning entire buffer
5. **Monitor error patterns:** Check for recurring issues with `query({ hasErrors: true })`

25.4.2 Forwarder Best Practices

1. **Avoid message loops:** Don't forward topics that SSE publishes back
2. **Use specific topic patterns:** Forward only what's needed
3. **Throttle high-frequency events:** Reduce network overhead
4. **Include message IDs:** Enable deduplication
5. **Secure endpoints:** Use HTTPS and authentication headers
6. **Handle server errors gracefully:** Forwarder silently ignores failures
7. **Test CORS configuration:** Ensure server allows cross-origin requests
8. **Use with-credentials carefully:** Required for cookies, but can cause CORS issues

25.5 Summary

Utility components provide infrastructure for observability and integration:

pan-debug gives you x-ray vision into message flows. Use it during development to understand routing behavior, diagnose issues, and profile performance. In production, enable sampling to capture errors without overwhelming resources.

pan-forwarder extends LARC beyond the browser. Use it to synchronize state with servers, log analytics events, trigger workflows, and build real-time collaborative features. Combined with **pan-sse**, it enables bidirectional message flow between client and server.

Both components embody LARC's philosophy: simple, focused tools that compose cleanly. They don't try to solve every problem—they solve specific problems well, and they integrate seamlessly with the rest of the ecosystem.

In the next chapter, we'll explore advanced integration patterns: building full-stack applications, implementing offline-first architectures, and connecting LARC to external systems and frameworks.

Chapter 26

Message Topics Reference

This appendix provides a comprehensive reference for LARC message topic conventions. Topics are the fundamental addressing mechanism in LARC applications—they determine how messages are routed, who receives them, and how components interact. Understanding topic patterns is essential for building scalable, maintainable LARC applications.

26.1 Topic Format and Structure

26.1.1 Standard Format

LARC topics follow a hierarchical dotted notation:

`resource.action.qualifier`

Components:

- **Resource:** The domain entity or system (users, posts, nav, ui, auth)
- **Action:** The operation or event type (list, item, get, save, updated)
- **Qualifier:** Additional context (state, request, reply, event)

This three-segment format provides clarity while remaining flexible enough for various use cases. More segments can be added when needed for specificity.

26.1.2 Topic Examples

Topic	Resource	Action	Qualifier	Purpose
<code>users.list.state</code>	users	list	state	Current user list (retained)
<code>users.item.get</code>	users	item	get	Fetch single user (request)
<code>posts.item.updated</code>	posts	item	updated	Post was modified (event)
<code>nav.goto</code>	nav	goto	—	Navigate to route (command)

Topic	Resource	Action	Qualifier	Purpose
<code>ui.modal.opened</code>	<code>ui</code>	<code>modal.opened</code>	—	Modal opened (event)
<code>auth.session.state</code>	<code>auth</code>	<code>session</code>	<code>state</code>	Current session (retained)

26.1.3 Naming Rules

Case and Separators:

- Use lowercase letters only
- Separate segments with dots (.)
- Use hyphens (-) within a segment if needed: `auth.two-factor.verify`

Character Set: - Alphanumeric characters: `a-z`, `0-9` - Dots for segment separation: `.` - Hyphens for compound words: `-` - No underscores, spaces, or special characters

Length: - Keep topics concise but descriptive - Typical range: 15-40 characters - Avoid abbreviations that sacrifice clarity

26.2 Topic Patterns and Matching

26.2.1 Exact Match

The simplest pattern matches a specific topic exactly:

```
client.subscribe('users.item.updated', (msg) => {
  console.log('User updated:', msg.data);
});
```

Receives only messages published to `users.item.updated`.

26.2.2 Single-Segment Wildcard

The asterisk (*) matches exactly one segment:

Pattern	Matches	Does Not Match
<code>users.*</code>	<code>users.list</code> , <code>users.item</code>	<code>users.list.state</code> , <code>users.item.updated</code>
<code>*.updated</code>	<code>users.updated</code> , <code>posts.updated</code>	<code>users.item.updated</code>
<code>users.*.state</code>	<code>users.list.state</code> , <code>users.item.state</code>	<code>users.state</code> , <code>users.list.item.state</code>

```
// Subscribe to all user-related events
client.subscribe('users.*', (msg) => {
  console.log('User event:', msg.topic);
});
```

26.2.3 Global Wildcard

The special pattern `*` matches all topics:

```
// Monitor all messages (use sparingly)
client.subscribe('*', (msg) => {
  console.log('[ALL]', msg.topic, msg.data);
});
```

Warning: Global wildcard subscriptions match every message in the system. Use them only for debugging, logging, or analytics. They can significantly impact performance in high-throughput applications.

26.2.4 Pattern Matching Examples

```
// Match all list operations
client.subscribe('*.list.*', (msg) => {
  // Matches: users.list.state, posts.list.get, comments.list.get
});

// Match all state topics
client.subscribe('*.*.state', (msg) => {
  // Matches: users.list.state, app.theme.state, nav.route.state
});

// Match all operations on items
client.subscribe('*.item.*', (msg) => {
  // Matches: users.item.get, posts.item.save, users.item.updated
});
```

26.3 Reserved Topic Namespaces

Certain topic namespaces are reserved for LARC internal use. Applications must not publish to these topics directly.

26.3.1 `pan:*` Namespace (System Internal)

The `pan:*` namespace is reserved for PAN bus internals:

Topic	Purpose	Usage
<code>pan:sys.ready</code>	Bus ready signal	Listen only
<code>pan:sys.stats</code>	Bus statistics	Request only
<code>pan:sys.error</code>	System errors	Listen only
<code>pan:sys.clear-retained</code>	Clear retained messages	Request only
<code>pan:publish</code>	Internal publish event	Internal only
<code>pan:subscribe</code>	Internal subscribe event	Internal only
<code>pan:unsubscribe</code>	Internal unsubscribe event	Internal only
<code>pan:deliver</code>	Internal message delivery	Internal only

Topic	Purpose	Usage
<code>pan:hello</code>	Client registration	Internal only
<code>pan:\$reply:*</code>	Auto-generated reply topics	Internal only

Never: - Publish to `pan:*` topics directly - Subscribe to internal topics like `pan:publish` or `pan:subscribe` - Manually create `pan:$reply:*` topics (these are auto-generated by `request()`)

Exception: You may subscribe to system notification topics like `pan:sys.ready` and `pan:sys.error`.

26.3.2 `sys:*` Namespace (System Reserved)

The `sys:*` namespace is reserved for future system-level functionality:

```
sys:error          # Future: System-wide errors
sys:perf           # Future: Performance monitoring
sys:debug          # Future: Debug information
sys:config         # Future: Configuration changes
```

Do not use `sys:*` topics in application code.

26.3.3 Application Namespaces

Your application should establish its own top-level namespaces:

Namespace	Purpose	Examples
<code>app.*</code>	Application-level concerns	<code>app.config.state</code> , <code>app.theme.state</code>
<code>auth.*</code>	Authentication/authorization	<code>auth.login</code> , <code>auth.session.state</code>
<code>session.*</code>	Session management	<code>session.started</code> , <code>session.expired</code>
<code>nav.*</code>	Navigation	<code>nav.goto</code> , <code>nav.route.state</code>
<code>ui.*</code>	UI components	<code>ui.modal.opened</code> , <code>ui.toast.show</code>
<code>analytics.*</code>	Analytics tracking	<code>analytics.event</code> , <code>analytics.page-view</code>

26.4 CRUD Topic Patterns

CRUD operations (Create, Read, Update, Delete) follow consistent topic patterns across all resources.

26.4.1 List Operations

26.4.1.1 List State (Retained)

Topic: `${resource}.list.state`

Purpose: Retained snapshot of current list data. New subscribers receive the most recent list immediately.

Message Format:

```
{
  topic: 'users.list.state',
  data: {
    items: [/* array of items */],
    total: 150,           // Total count (for pagination)
    page: 1,             // Current page
    filter: {},           // Active filters
    sort: 'name-asc'     // Active sort
  },
  retain: true
}
```

Usage:

```
// Publish list state
client.publish({
  topic: 'users.list.state',
  data: {
    items: users,
    total: users.length,
    page: 1
  },
  retain: true
});

// Subscribe to list state
client.subscribe('users.list.state', (msg) => {
  renderList(msg.data.items);
}, { retained: true });
```

26.4.1.2 List Get (Request)

Topic: \${resource}.list.get

Purpose: Request to fetch list data with optional parameters.

Request Format:

```
{
  topic: 'users.list.get',
  data: {
    page: 1,
    limit: 20,
    filter: { active: true },
    sort: 'name-asc'
  }
}
```

```
}
```

Response Format:

```
{
  ok: true,
  items: [/* array */],
  total: 150,
  page: 1
}
```

Usage:

```
// Request list
const response = await client.request('users.list.get', {
  page: 1,
  limit: 20,
  filter: { active: true }
});

if (response.data.ok) {
  renderList(response.data.items);
}
```

26.4.2 Item Operations

26.4.2.1 Item Get (Request)

Topic: \${resource}.item.get

Purpose: Fetch a single item by ID.

Request Format:

```
{
  topic: 'users.item.get',
  data: { id: 123 }
}
```

Response Format:

```
// Success
{ ok: true, item: { id: 123, name: 'Alice', email: '...' } }

// Not found
{ ok: false, error: 'Not found', code: 'NOT_FOUND' }
```

26.4.2.2 Item Save (Request)

Topic: \${resource}.item.save

Purpose: Create or update an item. If id is present, updates existing item. If id is omitted, creates new item.

Request Format:

```
{
  topic: 'users.item.save',
  data: {
    item: {
      id: 123,           // Omit for create
      name: 'Alice',
      email: 'alice@example.com'
    }
  }
}
```

Response Format:

```
// Success
{ ok: true, item: { id: 123, name: 'Alice', email: '...' } }

// Validation error
{ ok: false, error: 'Invalid email', code: 'VALIDATION_ERROR' }
```

26.4.2.3 Item Delete (Request)

Topic: \${resource}.item.delete

Purpose: Delete an item by ID.

Request Format:

```
{
  topic: 'users.item.delete',
  data: { id: 123 }
}
```

Response Format:

```
// Success
{ ok: true, id: 123 }

// Not found
{ ok: false, error: 'Not found', code: 'NOT_FOUND' }
```

26.4.2.4 Item Select (Event)

Topic: \${resource}.item.select

Purpose: User selected/focused an item (no reply expected).

Message Format:

```
{
  topic: 'users.item.select',
```

```
data: { id: 123 }
}
```

Usage:

```
// Publish selection
client.publish({
  topic: 'users.item.select',
  data: { id: userId }
});

// Handle selection
client.subscribe('users.item.select', (msg) => {
  highlightItem(msg.data.id);
  loadDetails(msg.data.id);
});
```

26.4.3 Item Events

Item events notify about completed operations:

Topic	Trigger	Data
<code>\${resource}.item.created</code>	Item created	<code>{ item: {...} }</code>
<code>\${resource}.item.updated</code>	Item updated	<code>{ item: {...} }</code>
<code>\${resource}.item.deleted</code>	Item deleted	<code>{ id: 123 }</code>

Example:

```
// After save operation completes
client.publish({
  topic: 'users.item.updated',
  data: { item: savedUser }
});
```

26.4.4 Per-Item State

For tracking state of individual items:

Topic: `${resource}.item.state.${id}`

Purpose: Retained state for a specific item (e.g., online status, typing indicator).

Example:

```
// Publish item state
client.publish({
  topic: `users.item.state.${userId}`,
  data: {
    id: userId,
    online: true,
  }
});
```

```

    typing: false,
    lastSeen: Date.now()
  },
  retain: true
});

// Subscribe to specific item
client.subscribe(`users.item.state.${userId}`, (msg) => {
  updatePresence(msg.data);
}, { retained: true });

// Subscribe to all item states
client.subscribe('users.item.state.*', (msg) => {
  updatePresence(msg.data);
});

```

26.5 State Management Topics

State topics use the `.state` qualifier and are always retained.

26.5.1 Global State

Pattern: `${domain}.state`

Examples:

<code>'app.config.state'</code>	# Application configuration
<code>'app.theme.state'</code>	# Current theme
<code>'app.language.state'</code>	# Current language
<code>'ui.sidebar.state'</code>	# Sidebar open/closed
<code>'ui.loading.state'</code>	# Loading indicator

Usage:

```

// Publish state
client.publish({
  topic: 'app.theme.state',
  data: { mode: 'dark', accent: '#007bff' },
  retain: true
});

// Subscribe (receives current state immediately)
client.subscribe('app.theme.state', (msg) => {
  applyTheme(msg.data);
}, { retained: true });

```

26.5.2 Scoped State

Pattern: `${domain}.${scope}.state`

Examples:

```
'users.list.state'      # User list
'auth.session.state'    # Current session
'nav.route.state'       # Current route
'search.query.state'    # Search query
'filters.active.state'  # Active filters
```

26.6 Events vs Commands

Distinguish between events (past tense) and commands (imperative).

26.6.1 Events

Events describe something that **already happened**. They use past tense.

Characteristics: - Past tense verbs: **created, updated, deleted, opened, closed** - Fire-and-forget (no reply expected) - Multiple subscribers allowed - Informational

Examples:

Topic	Description
users.item.created	A user was created
users.item.updated	A user was updated
ui.modal.opened	A modal was opened
ui.modal.closed	A modal was closed
session.started	Session started
session.expired	Session expired
auth.login.success	Login succeeded
auth.login.failed	Login failed
nav.navigated	Navigation completed

Usage:

```
// Publish event
client.publish({
  topic: 'users.item.created',
  data: { item: newUser }
});

// Multiple handlers can react
client.subscribe('users.item.created', logAnalytics);
client.subscribe('users.item.created', sendWelcomeEmail);
client.subscribe('users.item.created', updateDashboard);
```

26.6.2 Commands

Commands request something to **happen**. They use imperative/verb form.

Characteristics: - Imperative verbs: `save`, `delete`, `open`, `close`, `goto` - May expect reply (request/reply pattern) - Usually single handler - May fail

Examples:

Topic	Description
<code>users.item.save</code>	Save a user
<code>users.item.delete</code>	Delete a user
<code>ui.modal.open</code>	Open a modal
<code>ui.modal.close</code>	Close a modal
<code>nav.goto</code>	Navigate to route
<code>nav.back</code>	Go back in history
<code>auth.login</code>	Perform login
<code>auth.logout</code>	Perform logout

Usage:

```
// Fire-and-forget command
client.publish({
  topic: 'nav.goto',
  data: { route: '/users/123' }
});

// Request command (expect reply)
const response = await client.request('users.item.save', {
  item: { name: 'Alice' }
});
```

26.7 Domain-Specific Patterns

26.7.1 Authentication

```
// Commands
'auth.login'           # Login request
'auth.logout'          # Logout request
'auth.refresh'         # Refresh token
'auth.verify'          # Verify credentials

// Events
'auth.login.success'   # Login succeeded
'auth.login.failed'    # Login failed
'auth.logout'          # User logged out
'auth.token.expired'   # Token expired

// State
'auth.session.state'   # Current session (retained)
'auth.user.state'      # Current user info (retained)
```

26.7.2 Navigation

```
// Commands
'nav.goto'           # Navigate to route
'nav.back'           # Go back
'nav.forward'        # Go forward
'nav.replace'        # Replace current route

// Events
'nav.navigated'      # Navigation completed
'nav.error'          # Navigation error

// State
'nav.route.state'    # Current route (retained)
'nav.history.state'  # History stack (retained)
```

26.7.3 UI Components

```
// Modal
'ui.modal.open'      # Command: open modal
'ui.modal.close'     # Command: close modal
'ui.modal.opened'    # Event: modal opened
'ui.modal.closed'    # Event: modal closed
'ui.modal.state'     # State: current modal (retained)

// Sidebar
'ui.sidebar.toggle'  # Command: toggle sidebar
'ui.sidebar.open'    # Command: open sidebar
'ui.sidebar.close'   # Command: close sidebar
'ui.sidebar.state'   # State: open/closed (retained)

// Toast
'ui.toast.show'      # Command: show toast
'ui.toast.hide'      # Command: hide toast

// Loading
'ui.loading.start'   # Command: start loading
'ui.loading.stop'    # Command: stop loading
'ui.loading.state'   # State: loading status (retained)
```

26.7.4 Forms

```
// Validation
'form.validate'      # Command: validate form
'form.validated'     # Event: validation complete
'form.validation.state' # State: validation errors (retained)
```

```
// Submission
'form.submit'           # Command: submit form
'form.submitted'        # Event: form submitted
'form.submit.success'   # Event: submission succeeded
'form.submit.failed'    # Event: submission failed

// Field changes
'form.field.changed'    # Event: field value changed
'form.field.focused'    # Event: field focused
'form.field.blurred'    # Event: field blurred
```

26.7.5 Data Synchronization

```
// Sync commands
'sync.start'            # Start sync
'sync.stop'            # Stop sync
'sync.refresh'         # Force refresh

// Sync events
'sync.started'          # Sync started
'sync.completed'       # Sync completed
'sync.failed'          # Sync failed
'sync.conflict'        # Sync conflict detected

// Sync state
'sync.status.state'    # Current sync status (retained)
'sync.last-update.state' # Last update time (retained)
```

26.8 Best Practices

26.8.1 Topic Naming

DO: - Use lowercase letters - Use dots to separate segments - Use descriptive names - Be consistent across resources - Use `.state` for retained topics - Use past tense for events - Use imperative for commands

DON'T: - Use underscores or camelCase - Use abbreviations that sacrifice clarity - Mix naming conventions - Use verbs for events (`users.update X -> users.updated [check]`) - Overuse wildcards (`*` matches everything)

26.8.2 Topic Catalog

For larger applications, maintain a centralized topic catalog:

```
// topics.js
export const TOPICS = {
  USERS: {
    LIST: {
```

```

    STATE: 'users.list.state',
    GET: 'users.list.get'
  },
  ITEM: {
    GET: 'users.item.get',
    SAVE: 'users.item.save',
    DELETE: 'users.item.delete',
    SELECT: 'users.item.select',
    UPDATED: 'users.item.updated',
    DELETED: 'users.item.deleted',
    STATE: (id) => `users.item.state.${id}`
  }
},

NAV: {
  GOTO: 'nav.goto',
  BACK: 'nav.back',
  ROUTE_STATE: 'nav.route.state'
},

AUTH: {
  LOGIN: 'auth.login',
  LOGOUT: 'auth.logout',
  SESSION_STATE: 'auth.session.state'
}
};

// Usage
client.publish({
  topic: TOPICS.USERS.ITEM.UPDATED,
  data: { item: user }
});

```

26.8.3 Performance Considerations

Wildcard Usage: - Avoid global wildcard (*) in production code - Prefer specific patterns (**users.*** over *****) - Each wildcard increases matching overhead

Topic Depth: - Keep topics shallow (3-4 segments ideal) - Deeper hierarchies increase matching cost - Balance specificity with performance

State Retention: - Use retention sparingly (only for actual state) - Don't retain high-volume event streams - Clear retained messages when no longer needed

26.9 Summary

Key Principles:

1. **Standard Format:** `resource.action.qualifier`

2. **State Suffix:** Always use `.state` for retained topics
3. **Events vs Commands:** Past tense for events, imperative for commands
4. **Consistency:** Use same patterns across all resources
5. **Reserved Namespaces:** Never use `pan:*` or `sys:*`
6. **Wildcards:** Use judiciously; prefer specific patterns
7. **Documentation:** Maintain topic catalog for large apps

Quick Reference:

Pattern	Example	Use Case
<code>\${resource}.list.state</code>	<code>users.list.state</code>	List data (retained)
<code>\${resource}.list.get</code>	<code>users.list.get</code>	Request list
<code>\${resource}.item.get</code>	<code>users.item.get</code>	Request single item
<code>\${resource}.item.save</code>	<code>users.item.save</code>	Save item
<code>\${resource}.item.delete</code>	<code>users.item.delete</code>	Delete item
<code>\${resource}.item.updated</code>	<code>users.item.updated</code>	Item updated event
<code>\${domain}.state</code>	<code>app.theme.state</code>	Global state (retained)
<code>\${domain}.\${action}</code>	<code>nav.goto</code>	Command

For complete API documentation, see the main API Reference.

Chapter 27

Event Envelope Specification

This appendix provides the complete specification for LARC message envelopes—the data structures that wrap every message flowing through the PAN bus. Understanding the envelope format is critical for debugging, building tooling, and understanding how the system works at a fundamental level.

27.1 Overview

Every message in LARC is wrapped in an envelope that provides metadata, routing information, and payload data. The envelope follows a simple, predictable structure that balances flexibility with consistency.

Key Characteristics:

- Plain JavaScript objects (no classes or prototypes)
- JSON-serializable (can be logged, stored, transmitted)
- Immutable once published (bus may add fields, but won't modify existing)
- Extensible through headers and custom fields

27.2 Message Envelope Structure

27.2.1 Complete Format

```
interface PanMessage {  
  // Required fields (must be provided by publisher)  
  topic: string;  
  data: any;  
  
  // Auto-generated fields (added by bus if not provided)  
  id?: string;  
  ts?: number;  
  
  // Optional feature fields  
  retain?: boolean;
```

```

replyTo?: string;
correlationId?: string;
headers?: Record<string, string>;

// Internal/system fields (typically not used by applications)
clientId?: string;
}

```

27.2.2 Minimal Message

The absolute minimum required to publish a message:

```

{
  topic: 'users.updated',
  data: { id: 123, name: 'Alice' }
}

```

The bus will enhance this to:

```

{
  topic: 'users.updated',
  data: { id: 123, name: 'Alice' },
  id: '550e8400-e29b-41d4-a716-446655440000',
  ts: 1699564800000
}

```

27.3 Field Specifications

27.3.1 topic (required)

Type: string

Purpose: Identifies the message type and routing destination.

Format: Dotted notation, typically `resource.action.qualifier`

Constraints: - Must be non-empty string - Lowercase letters and dots recommended - Max length: 256 characters (practical limit) - Pattern: `/^[a-z0-9.-]+$/i`

Examples:

```

'users.list.state'
'users.item.get'
'nav.goto'
'ui.modal.opened'
'auth.session.state'

```

Validation:

```

// Valid topics
'users.updated'      [v]
'nav.goto'           [v]

```

```

'users.item.state.123'      [v]
'auth.two-factor.verify'    [v]

// Invalid topics
''                          [x] Empty string
'users updated'             [x] Contains space
'users_updated'            [x] Underscore (not recommended)
null                        [x] Not a string

```

Reserved Patterns:

- `pan:*` - Reserved for PAN bus internals
- `sys:*` - Reserved for system-level topics

27.3.2 data (required)

Type: any (must be JSON-serializable)

Purpose: Message payload—the actual information being communicated.

Constraints: - Must be JSON-serializable (no functions, circular refs, DOM nodes) - Recommended max size: 512KB (configurable via `max-payload-size`) - Can be any valid JSON type: object, array, string, number, boolean, null

Supported Types:

```

// Object
{ id: 123, name: 'Alice', active: true }

// Array
[ { id: 1 }, { id: 2 }, { id: 3 } ]

// String
"Hello, world"

// Number
42
3.14159

// Boolean
true
false

// Null
null

```

Invalid Data:

```

// Functions
data: () => console.log('hi')    [x]

```

```
// undefined (use null instead)
data: undefined           [x]

// Circular references
const obj = {};
obj.self = obj;
data: obj                  [x]

// DOM nodes
data: document.body       [x]
```

Best Practices:

```
// Good: structured data
{
  topic: 'users.item.updated',
  data: {
    id: 123,
    name: 'Alice',
    email: 'alice@example.com',
    updatedAt: Date.now()
  }
}

// Good: minimal data
{
  topic: 'users.item.select',
  data: { id: 123 }
}

// Good: null for no data
{
  topic: 'ui.modal.close',
  data: null
}

// Acceptable: primitive data
{
  topic: 'counter.value',
  data: 42
}

// Bad: empty object when null is better
{
  topic: 'ui.modal.close',
  data: {} // Use null instead
}
```

27.3.3 id (optional, auto-generated)

Type: string (UUID v4)

Purpose: Unique identifier for message deduplication, tracking, and correlation.

Auto-generation: If not provided, bus generates a UUID v4.

Format: Standard UUID format: xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx

Example:

```
'550e8400-e29b-41d4-a716-446655440000'
'7c9e6679-7425-40de-944b-e07fc1f90ae7'
```

Usage:

```
// Let bus generate (recommended)
client.publish({
  topic: 'users.updated',
  data: { id: 123 }
  // id will be auto-generated
});

// Provide custom ID (rare)
client.publish({
  topic: 'users.updated',
  data: { id: 123 },
  id: 'user-update-123-2024-11-01'
});
```

Use Cases: - Deduplication (detect duplicate messages) - Message tracking in logs - Correlation across systems - Idempotency keys

27.3.4 ts (optional, auto-generated)

Type: number (Unix timestamp in milliseconds)

Purpose: Message creation timestamp for ordering and time-based filtering.

Auto-generation: If not provided, bus adds `Date.now()`.

Format: Milliseconds since Unix epoch (January 1, 1970 00:00:00 UTC)

Example:

```
1699564800000 // 2023-11-10 00:00:00 UTC
1699651200000 // 2023-11-11 00:00:00 UTC
```

Usage:

```
// Let bus generate (recommended)
client.publish({
  topic: 'users.updated',
  data: { id: 123 }
```

```
// ts will be auto-generated
});

// Provide custom timestamp (rare)
client.publish({
  topic: 'users.updated',
  data: { id: 123 },
  ts: Date.parse('2024-11-01T12:00:00Z')
});
```

Use Cases: - Message ordering - Time-based filtering - Analytics and logging - Determining message freshness

Working with Timestamps:

```
// Convert to Date object
const date = new Date(msg.ts);

// Format for display
const formatted = new Date(msg.ts).toISOString();
// "2024-11-01T12:00:00.000Z"

// Check message age
const ageMs = Date.now() - msg.ts;
const ageSeconds = ageMs / 1000;
```

27.3.5 retain (optional)

Type: boolean

Purpose: Indicates message should be retained by bus and replayed to new subscribers.

Default: false

Constraints: - Only one message retained per topic (last value wins) - Subject to LRU eviction if bus retention limit exceeded - Bus default max retained: 1000 messages (configurable via **max-retained**)

Usage:

```
// Publish retained state
client.publish({
  topic: 'app.theme.state',
  data: { mode: 'dark' },
  retain: true
});

// Later subscriber receives immediately
client.subscribe('app.theme.state', (msg) => {
  applyTheme(msg.data);
}, { retained: true });
```

When to Use: - Application state (theme, language, configuration) - List data (current user list, current items) - Session information (current user, authentication) - Last known values (device status, connection state)

When NOT to Use: - Events (one-time notifications) - High-frequency updates (mouse movements, scroll events) - Temporary notifications (toasts, alerts) - Bulk data (large lists, file contents)

27.3.6 replyTo (optional)

Type: string (topic name)

Purpose: Specifies topic where reply should be sent (request/reply pattern).

Auto-generation: Auto-generated by `client.request()` method.

Format: Typically `pan:$reply:${clientId}:${correlationId}`

Usage:

```
// Manually set replyTo
client.publish({
  topic: 'users.item.get',
  data: { id: 123 },
  replyTo: 'users.item.get.reply.abc123',
  correlationId: 'req-001'
});

// Subscribe to reply
client.subscribe('users.item.get.reply.abc123', (msg) => {
  console.log('Response:', msg.data);
});

// Better: use client.request() (auto-generates replyTo)
const response = await client.request('users.item.get', { id: 123 });
```

Responder Pattern:

```
client.subscribe('users.item.get', async (msg) => {
  // Check if reply expected
  if (!msg.replyTo) return;

  const user = await database.getUser(msg.data.id);

  // Send reply to specified topic
  client.publish({
    topic: msg.replyTo,
    data: user ? { ok: true, item: user } : { ok: false, error: 'Not found' },
    correlationId: msg.correlationId
  });
});
```

27.3.7 correlationId (optional)

Type: string (UUID or custom identifier)

Purpose: Correlates replies with original requests in request/reply pattern.

Auto-generation: Auto-generated by `client.request()` method.

Format: Typically UUID, but can be any string identifier.

Usage:

```
// Manual correlation (rare)
const corrId = crypto.randomUUID();

client.publish({
  topic: 'users.item.get',
  data: { id: 123 },
  replyTo: 'users.reply',
  correlationId: corrId
});

client.subscribe('users.reply', (msg) => {
  if (msg.correlationId === corrId) {
    console.log('Our reply:', msg.data);
  }
});

// Better: use client.request() (auto-correlates)
const response = await client.request('users.item.get', { id: 123 });
```

Use Cases: - Request/reply pattern - Tracking conversation threads - Matching async responses - Distributed tracing

27.3.8 headers (optional)

Type: Record<string, string> (string key-value pairs)

Purpose: Free-form metadata for custom application needs.

Constraints: - Keys and values must be strings - No reserved header names (yet) - Included in message size calculations

Common Use Cases:

- User context (userId, sessionId, tenantId)
- Distributed tracing (traceId, spanId, parentSpanId)
- Source tracking (source, version, environment)
- Custom metadata (priority, category, tags)

Usage:

```
// Add headers
client.publish({
```

```
topic: 'analytics.event',
data: { action: 'click', target: 'button' },
headers: {
  userId: '123',
  sessionId: 'abc-def-ghi',
  source: 'mobile-app',
  version: '2.1.0',
  environment: 'production'
}
});

// Access headers in subscriber
client.subscribe('analytics.*', (msg) => {
  const userId = msg.headers?.userId;
  const source = msg.headers?.source;

  logEvent(msg.topic, msg.data, { userId, source });
});
```

Distributed Tracing Example:

```
// Start trace
const traceId = crypto.randomUUID();
const spanId = crypto.randomUUID();

client.publish({
  topic: 'users.item.get',
  data: { id: 123 },
  headers: {
    traceId,
    spanId,
    parentSpanId: null
  }
});

// Continue trace in handler
client.subscribe('users.item.get', async (msg) => {
  const childSpanId = crypto.randomUUID();

  // Process with trace context
  await database.getUser(msg.data.id, {
    traceId: msg.headers.traceId,
    parentSpanId: msg.headers.spanId,
    spanId: childSpanId
  });
});
```

27.3.9 clientId (internal)

Type: string

Purpose: Internal identifier for client that published the message.

Auto-generation: Generated by PanClient constructor.

Format: \${elementTag}#\${uuid}

Example: pan-user-list#7c9e6679-7425-40de-944b

Usage: Primarily for internal bus operations. Applications typically don't need to use this field.

27.4 Message Examples

27.4.1 Simple Event

```
{
  topic: 'users.item.updated',
  data: {
    id: 123,
    name: 'Alice Cooper',
    email: 'alice@example.com',
    updatedAt: 1699564800000
  },
  id: '550e8400-e29b-41d4-a716-446655440000',
  ts: 1699564800000
}
```

27.4.2 Retained State

```
{
  topic: 'users.list.state',
  data: {
    items: [
      { id: 1, name: 'Alice' },
      { id: 2, name: 'Bob' },
      { id: 3, name: 'Charlie' }
    ],
    total: 3,
    page: 1
  },
  id: '7c9e6679-7425-40de-944b-e07fc1f90ae7',
  ts: 1699651200000,
  retain: true
}
```

27.4.3 Request Message

```
{
  topic: 'users.item.get',
  data: { id: 123 },
  id: 'a1b2c3d4-e5f6-4a5b-8c9d-0e1f2a3b4c5d',
  ts: 1699651200000,
  replyTo: 'pan:$reply:user-list#abc123:req-001',
  correlationId: 'req-001'
}
```

27.4.4 Reply Message

```
{
  topic: 'pan:$reply:user-list#abc123:req-001',
  data: {
    ok: true,
    item: {
      id: 123,
      name: 'Alice',
      email: 'alice@example.com'
    }
  },
  id: 'b2c3d4e5-f6a7-4b5c-9d0e-1f2a3b4c5d6e',
  ts: 1699651205000,
  correlationId: 'req-001'
}
```

27.4.5 Message with Headers

```
{
  topic: 'analytics.page-view',
  data: {
    page: '/users/123',
    duration: 5000,
    referrer: '/dashboard'
  },
  id: 'c3d4e5f6-a7b8-4c5d-0e1f-2a3b4c5d6e7f',
  ts: 1699651200000,
  headers: {
    userId: '456',
    sessionId: 'session-xyz',
    device: 'mobile',
    browser: 'Chrome',
    version: '119.0',
    traceId: 'trace-abc-def'
  }
}
```

```
}
```

27.4.6 Error Response

```
{
  topic: 'pan:$reply:user-form#xyz789:req-002',
  data: {
    ok: false,
    error: 'User not found',
    code: 'NOT_FOUND',
    details: {
      requestedId: 999,
      timestamp: 1699651200000
    }
  },
  id: 'd4e5f6a7-b8c9-4d5e-1f2a-3b4c5d6e7f8a',
  ts: 1699651210000,
  correlationId: 'req-002'
}
```

27.5 Response Payload Conventions

While `data` can be any JSON value, LARC follows conventions for response payloads in request/reply patterns:

27.5.1 Success Response

```
{
  ok: true,
  item: { /* single item */ }
}

// or for lists
{
  ok: true,
  items: [ /* array of items */ ],
  total: 100
}
```

27.5.2 Error Response

```
{
  ok: false,
  error: 'Human-readable error message',
  code: 'ERROR_CODE',           // Optional: machine-readable code
}
```

```
details: { /* context */ }    // Optional: additional context
}
```

27.5.3 Example Error Codes

```
'NOT_FOUND'           // Resource doesn't exist
'VALIDATION_ERROR'    // Invalid input
'UNAUTHORIZED'        // Not authenticated
'FORBIDDEN'           // Not authorized
'CONFLICT'            // Resource conflict (e.g., duplicate)
'RATE_LIMIT'          // Too many requests
'SERVER_ERROR'        // Internal error
'TIMEOUT'             // Operation timed out
```

27.6 Message Size Limits

The PAN bus enforces configurable size limits:

27.6.1 Default Limits

- **Max message size:** 1MB (1,048,576 bytes)
- **Max payload size:** 512KB (524,288 bytes)

27.6.2 Configuration

```
<pan-bus
  max-message-size="2097152"
  max-payload-size="1048576">
</pan-bus>
```

27.6.3 Size Estimation

The bus estimates message size by JSON-stringifying and measuring:

```
function estimateSize(msg) {
  return new Blob([JSON.stringify(msg)]).size;
}
```

27.6.4 Handling Size Limits

```
// Large data: paginate requests
const response = await client.request('users.list.get', {
  page: 1,
  limit: 50 // Smaller page size
});

// Large payloads: use chunking or external storage
```

```
// Instead of:
client.publish({
  topic: 'file.uploaded',
  data: { fileContents: hugeBase64String } // Too large!
});

// Do:
const fileId = await uploadToStorage(fileContents);
client.publish({
  topic: 'file.uploaded',
  data: { fileId, url: `/files/${fileId}` }
});
```

27.7 Validation

The PAN bus validates messages before processing:

27.7.1 Topic Validation

```
// Must be non-empty string
topic: '' [x]
topic: null [x]
topic: undefined [x]

// Must not be reserved
topic: 'pan:publish' [x] (reserved)
topic: 'pan:subscribe' [x] (reserved)

// Valid
topic: 'users.updated' [v]
```

27.7.2 Data Validation

```
// Must be JSON-serializable
data: () => {} [x] (function)
data: document.body [x] (DOM node)
data: undefined [x] (use null)

const obj = {};
obj.self = obj;
data: obj [x] (circular reference)

// Valid
data: null [v]
data: { id: 123 } [v]
data: [1, 2, 3] [v]
```

```
data: "string"           [v]
data: 42                  [v]
```

27.7.3 Size Validation

Messages exceeding size limits are rejected with error:

```
// Error emitted if message too large
{
  topic: 'pan:sys.error',
  data: {
    code: 'MESSAGE_INVALID',
    message: 'Message size (2000000 bytes) exceeds limit (1048576 bytes)',
    details: { topic: 'users.list.state' }
  }
}
```

27.8 Internal System Messages

Certain system messages are emitted by the bus:

27.8.1 pan:sys.ready

Emitted when bus initializes:

```
{
  topic: 'pan:sys.ready',
  data: {
    enhanced: true,
    routing: false,
    tracing: false,
    config: { /* bus configuration */ }
  }
}
```

27.8.2 pan:sys.error

Emitted for validation errors, rate limits, etc:

```
{
  topic: 'pan:sys.error',
  data: {
    code: 'RATE_LIMIT_EXCEEDED',
    message: 'Too many messages',
    details: { clientId: 'user-list#abc' }
  }
}
```

27.8.3 pan:sys.stats

Response to stats request:

```
{
  topic: 'pan:sys.stats',
  data: {
    published: 1234,
    delivered: 5678,
    dropped: 0,
    retainedEvicted: 5,
    subsCleanedUp: 2,
    errors: 1,
    subscriptions: 18,
    clients: 5,
    retained: 42,
    config: { /* current config */ }
  }
}
```

27.9 Summary

Required Fields: - **topic** (string) - Message routing address - **data** (any) - JSON-serializable payload

Auto-Generated Fields:

- **id** (string) - UUID for deduplication/tracking
- **ts** (number) - Unix timestamp in milliseconds

Feature Fields: - **retain** (boolean) - Retain for late subscribers - **replyTo** (string) - Reply destination topic - **correlationId** (string) - Request/reply correlation - **headers** (object) - Custom metadata

Constraints: - Topic: non-empty string, max 256 chars - Data: JSON-serializable, max 512KB (configurable) - Total message: max 1MB (configurable) - Headers: string key-value pairs only

Best Practices: - Let bus auto-generate **id** and **ts** - Use structured objects for **data** - Use **retain: true** only for actual state - Follow response conventions (**ok**, **error**, **code**) - Include relevant context in **headers** for tracing - Keep messages small; paginate or reference external storage

For topic naming conventions, see Appendix A. For configuration options, see Appendix C.

Chapter 28

Configuration Options

This appendix provides a comprehensive reference for all configuration options available in LARC, from PAN bus settings to component configuration, global defaults, and environment variables. These settings control performance characteristics, security policies, feature flags, and operational behavior.

28.1 PAN Bus Configuration

The `<pan-bus>` element accepts configuration through HTML attributes. These settings control the bus's behavior, resource limits, and enabled features.

28.1.1 Attribute Reference

28.1.1.1 max-retained

Type: Integer **Default:** 1000 **Range:** 1 to 100000 **Purpose:** Maximum number of retained messages stored by the bus.

When this limit is exceeded, the bus evicts the least recently accessed retained message (LRU eviction).

```
<pan-bus max-retained="5000"></pan-bus>
```

Use Cases: - Small apps with minimal state: 100–500 - Medium apps: 1000–2000 (default: 1000)
- Large apps with extensive state: 5000–10000

Performance Impact:

- Higher values: More memory usage, slower eviction checks
- Lower values: Less memory, faster eviction, more message loss

Monitoring:

```
const stats = await client.request('pan:sys.stats', {});
console.log('Retained:', stats.data.retained);
console.log('Evicted:', stats.data.retainedEvicted);
```

28.1.1.2 max-message-size

Type: Integer **Default:** 1048576 (1MB) **Range:** 1024 to 10485760 (1KB to 10MB) **Purpose:** Maximum total size of a message envelope in bytes.

Includes all fields: topic, data, headers, id, ts, etc.

```
<pan-bus max-message-size="2097152"></pan-bus>
```

Recommendations: - Default 1048576 (1MB) suitable for most apps - Increase for apps with large payloads (analytics, file metadata) - Decrease for memory-constrained environments (IoT, embedded)

Enforcement:

```
// Message rejected if too large
{
  topic: 'pan:sys.error',
  data: {
    code: 'MESSAGE_INVALID',
    message: 'Message size (2000000 bytes) exceeds limit (1048576 bytes)'
  }
}
```

28.1.1.3 max-payload-size

Type: Integer **Default:** 524288 (512KB) **Range:** 1024 to 5242880 (1KB to 5MB) **Purpose:** Maximum size of message data field in bytes.

Separate limit for payload to prevent large data objects from consuming resources.

```
<pan-bus max-payload-size="1048576"></pan-bus>
```

Relationship to max-message-size:

max-payload-size <= max-message-size

Best Practice: Set max-payload-size to 50-70% of max-message-size to leave room for headers and metadata.

28.1.1.4 cleanup-interval

Type: Integer **Default:** 30000 (30 seconds) **Range:** 1000 to 300000 (1s to 5min) **Purpose:** Interval in milliseconds between automatic cleanup cycles.

The bus periodically removes dead subscriptions (components removed from DOM) and stale rate limit data.

```
<pan-bus cleanup-interval="60000"></pan-bus>
```

Tuning: - Frequent cleanup (10-20s): Lower memory, higher CPU - Infrequent cleanup (60-120s): Higher memory, lower CPU - Default (30s): Balanced

Monitoring:

```
const stats = await client.request('pan:sys.stats', {});
console.log('Cleaned up:', stats.data.subsCleanedUp);
```

28.1.1.5 rate-limit

Type: Integer **Default:** 1000 **Range:** 1 to 100000 **Purpose:** Maximum messages per client per rate limit window.

Prevents a single client from overwhelming the bus.

```
<pan-bus rate-limit="5000"></pan-bus>
```

Calculation:

$$\text{messages_per_second} = \text{rate-limit} / (\text{rate-limit-window} / 1000)$$

Default: $1000 / (1000 / 1000) = 1000$ messages/second

Rate Limit Exceeded:

```
// Message rejected
{
  topic: 'pan:sys.error',
  data: {
    code: 'RATE_LIMIT_EXCEEDED',
    message: 'Too many messages',
    details: { clientId: 'user-list#abc123' }
  }
}
```

Recommendations: - Development: 1000-5000 (relaxed) - Production: 1000-2000 (default) - High-throughput apps: 5000-10000

28.1.1.6 rate-limit-window

Type: Integer **Default:** 1000 (1 second) **Range:** 100 to 60000 (100ms to 1min) **Purpose:** Time window in milliseconds for rate limiting.

Works together with `rate-limit` to determine messages per time window.

```
<pan-bus rate-limit="2000" rate-limit-window="2000"></pan-bus>
```

This allows 2000 messages per 2 seconds = 1000 messages/second.

28.1.1.7 allow-global-wildcard

Type: Boolean **Default:** true **Purpose:** Allow or disallow global wildcard (*) subscriptions.

Global wildcard subscriptions match every message in the system. Disabling can improve security and performance.

```
<pan-bus allow-global-wildcard="false"></pan-bus>
```

When Disabled:

```
// Subscription rejected
client.subscribe('*', handler);
// Error: Global wildcard (*) subscriptions are disabled for security
```

Recommendations: - Development: `true` (useful for debugging) - Production: `false` (security/performance)

28.1.1.8 debug

Type: Boolean **Default:** `false` **Purpose:** Enable detailed console logging of bus operations.

```
<pan-bus debug="true"></pan-bus>
```

Output Examples:

```
[PAN Bus] PAN Bus Enhanced ready { maxRetained: 1000, ... }
[PAN Bus] Client registered { id: 'user-list#abc', caps: ['client'] }
[PAN Bus] Subscription added { pattern: 'users.*', clientId: 'user-list#abc' }
[PAN Bus] Published { topic: 'users.updated', delivered: 3, routes: 0 }
[PAN Bus] Cleaned up 2 dead subscriptions
```

Performance Impact: Minimal (logging is cheap), but avoid in production.

28.1.1.9 enable-routing

Type: Boolean **Default:** `false` **Purpose:** Enable the advanced routing system for message transformation and filtering.

```
<pan-bus enable-routing="true"></pan-bus>
```

Features Enabled:

- Declarative routing rules
- Message transformation
- Message filtering
- Topic aliasing
- Conditional routing

Access Routing API:

```
// Routes available at window.pan.routes
window.pan.routes.add({
  name: 'user-events-to-analytics',
  match: 'users.item.*',
  transform: (msg) => ({
    topic: 'analytics.event',
    data: { entity: 'user', action: msg.topic, ...msg.data }
  })
});
```

Performance Impact: Slight overhead per message for route matching.

28.1.1.10 enable-tracing

Type: Boolean **Default:** false **Purpose:** Enable message tracing for debugging and monitoring.

```
<pan-bus enable-tracing="true"></pan-bus>
```

Features Enabled:

- Message flow visualization
- Delivery tracking
- Route matching logs
- Performance metrics

Access Tracing API:

```
// Debug manager available at window.pan.debug
const traces = window.pan.debug.getTraces();
console.log('Recent traces:', traces);
```

28.1.2 Complete Configuration Example

```
<!DOCTYPE html>
<html>
<head>
  <title>My LARC App</title>
</head>
<body>
  <pan-bus
    max-retained="2000"
    max-message-size="2097152"
    max-payload-size="1048576"
    cleanup-interval="60000"
    rate-limit="5000"
    rate-limit-window="1000"
    allow-global-wildcard="false"
    debug="false"
    enable-routing="false"
    enable-tracing="false">
  </pan-bus>

  <my-app></my-app>
</body>
</html>
```

28.2 PanClient Configuration

The `PanClient` class accepts configuration through constructor parameters and method options.

28.2.1 Constructor Options

```
new PanClient(host?, busSelector?)
```

28.2.1.1 host

Type: HTMLElement | Document **Default:** document **Purpose:** Element to dispatch/receive events from.

```
// Default: document-level client
const client = new PanClient();

// Component-scoped client
class MyComponent extends HTMLElement {
  connectedCallback() {
    this.client = new PanClient(this);
  }
}
```

Use Cases: - Document-level (document): Most common, global communication - Component-scoped (element): Isolated communication within subtree

28.2.1.2 busSelector

Type: string **Default:** 'pan-bus' **Purpose:** CSS selector for bus element.

```
// Default selector
const client = new PanClient(document, 'pan-bus');

// Custom selector
const client = new PanClient(document, '#my-custom-bus');
```

Rarely needed unless using multiple buses or custom naming.

28.2.2 Subscription Options

```
client.subscribe(topics, handler, options?)
```

28.2.2.1 retained

Type: boolean **Default:** false **Purpose:** Receive retained messages immediately upon subscription.

```
// Get current state immediately
client.subscribe('app.theme.state', (msg) => {
  applyTheme(msg.data);
}, { retained: true });
```

Behavior: - **true:** Receives retained message immediately (if exists), then future messages - **false:** Only receives messages published after subscription

28.2.2.2 signal

Type: AbortSignal **Default:** undefined **Purpose:** Automatically unsubscribe when signal is aborted.

```
const controller = new AbortController();

client.subscribe('users.*', handler, {
  signal: controller.signal
});

// Later: unsubscribe all at once
controller.abort();
```

Use Case: Clean up multiple subscriptions with single abort.

28.2.3 Request Options

```
client.request(topic, data, options?)
```

28.2.3.1 timeoutMs

Type: number **Default:** 5000 (5 seconds) **Range:** 100 to 300000 (100ms to 5min) **Purpose:** Maximum time to wait for reply before timeout error.

```
// Default timeout
const response = await client.request('users.item.get', { id: 123 });

// Custom timeout
const response = await client.request('slow.operation', { ... }, {
  timeoutMs: 30000 // 30 seconds
});
```

Timeout Error:

```
try {
  await client.request('users.item.get', { id: 123 }, { timeoutMs: 1000 });
} catch (err) {
  console.error(err.message); // "PAN request timeout"
}
```

Recommendations: - Fast queries: 1000–2000ms - Standard requests: 5000ms (default) - Slow operations: 10000–30000ms - Long-running tasks: Consider async pattern instead

28.3 Component Configuration

LARC components can be configured through attributes, properties, and data attributes.

28.3.1 Standard Attributes

Most LARC components follow these conventions:

28.3.1.1 data-* Attributes

Use for configuration and initial values:

```
<pan-user-list
  data-page-size="50"
  data-sort="name-asc"
  data-filter="active">
</pan-user-list>
```

28.3.1.2 Boolean Attributes

Use presence/absence for boolean flags:

```
<!-- Feature enabled -->
<pan-data-grid sortable filterable paginated></pan-data-grid>

<!-- Feature disabled -->
<pan-data-grid></pan-data-grid>
```

28.3.1.3 Value Attributes

Use for string/number values:

```
<pan-modal
  size="large"
  position="center"
  backdrop="true">
</pan-modal>
```

28.3.2 Component-Specific Configuration

Configuration varies by component. Refer to component documentation for specifics.

Example: pan-storage

```
<pan-storage
  key="my-app-state"
  storage="localStorage"
  sync="true"
  debounce="1000">
</pan-storage>
```

Example: pan-routes

```
<pan-routes
  base-path="/app"
  hash-routing="false"
  scroll-restoration="true">
</pan-routes>
```

28.4 Global Configuration

Global configuration affects all LARC components and clients.

28.4.1 Window Configuration

Configure LARC globally before bus initialization:

```
<script>
window.LARC_CONFIG = {
  bus: {
    maxRetained: 2000,
    debug: false
  },
  defaults: {
    requestTimeout: 10000,
    retainedSubscription: true
  }
};
</script>

<pan-bus></pan-bus>
```

Note: This is a proposed pattern. Current implementation uses attributes only.

28.4.2 Feature Flags

Control experimental or optional features:

```
window.LARC_FEATURES = {
  routing: false,
  tracing: false,
  devtools: true
};
```

28.5 Environment Variables

For build-time configuration, use environment variables:

28.5.1 NODE_ENV

Values: development | production | test **Purpose:** Affects default behavior and optimizations.

```
NODE_ENV=production npm run build
```

Impact: - development: Debug logging, dev warnings, relaxed limits - production: Optimized, no debug output, strict limits - test: Test-specific behavior, mocked services

28.5.2 LARC_DEBUG

Values: true | false **Purpose:** Override debug mode regardless of NODE_ENV.

```
LARC_DEBUG=true npm start
```

28.5.3 LARC_MAX_RETAINED

Values: Integer **Purpose:** Override default max retained messages.

```
LARC_MAX_RETAINED=5000 npm start
```

28.5.4 LARC_RATE_LIMIT

Values: Integer **Purpose:** Override default rate limit.

```
LARC_RATE_LIMIT=10000 npm start
```

28.6 Configuration Profiles

Recommended configuration profiles for different scenarios:

28.6.1 Development Profile

Focus: Developer experience, debugging, relaxed limits

```
<pan-bus
  max-retained="500"
  max-message-size="1048576"
  max-payload-size="524288"
  cleanup-interval="30000"
  rate-limit="10000"
  allow-global-wildcard="true"
  debug="true"
  enable-routing="false"
  enable-tracing="true">
</pan-bus>
```

28.6.2 Production Profile

Focus: Performance, security, resource limits

```
<pan-bus
  max-retained="2000"
  max-message-size="1048576"
  max-payload-size="524288"
  cleanup-interval="60000"
  rate-limit="2000"
  allow-global-wildcard="false"
  debug="false"
  enable-routing="false"
  enable-tracing="false">
</pan-bus>
```

28.6.3 High-Throughput Profile

Focus: Maximum message volume, relaxed limits

```
<pan-bus
  max-retained="5000"
  max-message-size="2097152"
  max-payload-size="1048576"
  cleanup-interval="120000"
  rate-limit="10000"
  rate-limit-window="1000"
  allow-global-wildcard="false"
  debug="false"
  enable-routing="true"
  enable-tracing="false">
</pan-bus>
```

28.6.4 Memory-Constrained Profile

Focus: Minimal memory footprint

```
<pan-bus
  max-retained="100"
  max-message-size="262144"
  max-payload-size="131072"
  cleanup-interval="15000"
  rate-limit="500"
  allow-global-wildcard="false"
  debug="false"
  enable-routing="false"
  enable-tracing="false">
</pan-bus>
```

28.6.5 Testing Profile

Focus: Predictable behavior, fast cleanup

```
<pan-bus
  max-retained="50"
  max-message-size="1048576"
  max-payload-size="524288"
  cleanup-interval="5000"
  rate-limit="1000"
  allow-global-wildcard="true"
  debug="true"
  enable-routing="false"
  enable-tracing="true">
</pan-bus>
```

28.7 Runtime Configuration

Some settings can be changed at runtime through the bus API.

28.7.1 Get Current Configuration

```
const stats = await client.request('pan:sys.stats', {});
console.log('Config:', stats.data.config);
```

28.7.2 Clear Retained Messages

```
// Clear all retained messages
client.publish({
  topic: 'pan:sys.clear-retained',
  data: {}
});

// Clear matching pattern
client.publish({
  topic: 'pan:sys.clear-retained',
  data: { pattern: 'users.*' }
});
```

28.7.3 Get Statistics

```
const stats = await client.request('pan:sys.stats', {});

console.log({
  published: stats.data.published,
  delivered: stats.data.delivered,
  dropped: stats.data.dropped,
  retained: stats.data.retained,
  retainedEvicted: stats.data.retainedEvicted,
  subsCleanedUp: stats.data.subsCleanedUp,
  errors: stats.data.errors,
  subscriptions: stats.data.subscriptions,
  clients: stats.data.clients
});
```

28.8 Configuration Best Practices

28.8.1 Start Conservative

Begin with default settings:

```
<pan-bus></pan-bus>
```

Monitor with stats, adjust only when needed.

28.8.2 Monitor and Tune

```
// Periodic monitoring
setInterval(async () => {
  const stats = await client.request('pan:sys.stats', {});

  console.log('Bus Health:', {
    retained: `${stats.data.retained} / ${stats.data.config.maxRetained}`,
    dropped: stats.data.dropped,
    errors: stats.data.errors
  });

  // Alert if approaching limits
  if (stats.data.retained > stats.data.config.maxRetained * 0.8) {
    console.warn('Retained messages approaching limit');
  }
}, 60000);
```

28.8.3 Environment-Specific Configuration

Use different profiles per environment:

```
// config.js
export function getBusConfig() {
  if (process.env.NODE_ENV === 'production') {
    return {
      maxRetained: 2000,
      debug: false,
      rateLimit: 2000
    };
  }

  return {
    maxRetained: 500,
    debug: true,
    rateLimit: 10000
  };
}
```

```
<script type="module">
import { getBusConfig } from './config.js';

const config = getBusConfig();

document.querySelector('pan-bus').setAttribute('max-retained', config.maxRetained);
document.querySelector('pan-bus').setAttribute('debug', config.debug);
document.querySelector('pan-bus').setAttribute('rate-limit', config.rateLimit);
</script>
```

28.8.4 Document Your Configuration

```

<!--
  PAN Bus Configuration

  Environment: Production
  Profile: High-availability

  max-retained: 2000
    - Expected state topics: ~500
    - Headroom: 4x expected

  rate-limit: 2000
    - Expected peak load: 1000 msg/s
    - Headroom: 2x peak

  Last tuned: 2024-11-01
  Next review: 2024-12-01
-->
<pan-bus
  max-retained="2000"
  rate-limit="2000"
  debug="false">
</pan-bus>

```

28.9 Configuration Checklist

Pre-Launch: - ☐ `debug="false"` in production - ☐ `allow-global-wildcard="false"` for security
 - ☐ `max-retained` appropriate for app state volume - ☐ `rate-limit` appropriate for expected load
 - ☐ Monitoring enabled for bus statistics - ☐ Configuration documented in code comments

Performance Tuning:

- ☐ Monitor `retained` approaching `maxRetained`
- ☐ Monitor `dropped` messages (rate limiting)
- ☐ Monitor `errors` (validation, size limits)
- ☐ Monitor `subsCleanedUp` (memory leaks?)
- ☐ Adjust limits based on observed usage

Security: - ☐ Global wildcard disabled in production - ☐ Rate limits prevent DoS - ☐ Message size limits prevent memory exhaustion - ☐ Debug mode disabled in production

28.10 Summary

PAN Bus Core Settings:

- `max-retained` - Retained message limit (default: 1000)
- `max-message-size` - Total message size limit (default: 1MB)
- `max-payload-size` - Payload size limit (default: 512KB)

- `cleanup-interval` - Cleanup cycle interval (default: 30s)
- `rate-limit` - Messages per window (default: 1000)
- `allow-global-wildcard` - Allow `*` subscriptions (default: true)
- `debug` - Enable debug logging (default: false)

PAN Bus Features:

- `enable-routing` - Enable routing system (default: false)
- `enable-tracing` - Enable message tracing (default: false)

PanClient Settings:

- `host` - Event dispatch element (default: document)
- `busSelector` - Bus element selector (default: 'pan-bus')
- `retained` - Receive retained messages (default: false)
- `timeoutMs` - Request timeout (default: 5000ms)

Recommended Profiles:

- Development: Debug enabled, relaxed limits
- Production: Debug disabled, strict limits
- High-throughput: Increased limits, routing enabled
- Memory-constrained: Minimal retained, frequent cleanup
- Testing: Fast cleanup, debug enabled

Monitoring: - Use `pan:sys.stats` to monitor bus health - Alert on approaching limits - Tune based on observed behavior - Document configuration decisions

For message envelope structure, see Appendix B. For topic conventions, see Appendix A.

Chapter 29

Migration Guide

This appendix helps you upgrade LARC applications across versions, navigate breaking changes, and adopt new features while maintaining stability. Whether you're moving from an early prototype to a production release or keeping pace with framework evolution, this guide provides version-specific migration paths and practical strategies.

29.1 General Migration Strategy

Before diving into version-specific changes, establish a methodical upgrade process:

- 1. Review the Changelog** Start with LARC's release notes. Note breaking changes, deprecations, and new features relevant to your application.
- 2. Update in Increments** Avoid jumping multiple major versions. Upgrade one major version at a time, testing thoroughly between steps.
- 3. Run Your Test Suite** Execute all tests before and after migration. Pay special attention to component integration tests that exercise PAN bus communication.
- 4. Check Dependencies** Ensure your LARC-compatible libraries (routing, state management) support the new version. Update these incrementally.
- 5. Use Feature Flags** When migrating large applications, use feature flags to toggle between old and new implementations during transition periods.

29.2 Version 0.x to 1.0 Migration

The move to LARC 1.0 established core APIs and stabilized component architecture. Key changes:

29.2.1 Component Registration Changes

Before (0.x):

```
LARC.register('my-widget', {  
  template: '<div>Content</div>',  
  props: ['data']  
});
```

After (1.0):

```
class MyWidget extends HTMLElement {
  static observedAttributes = ['data'];

  connectedCallback() {
    this.render();
  }

  render() {
    this.innerHTML = '<div>Content</div>';
  }
}

customElements.define('my-widget', MyWidget);
```

Migration Steps: 1. Convert registration objects to ES6 classes extending `HTMLElement` 2. Move lifecycle hooks to standard Web Components callbacks 3. Implement `observedAttributes` for reactive properties 4. Replace template strings with `render()` methods

29.2.2 PAN Bus API Refinement

Version 1.0 introduced explicit bus references rather than implicit global access.

Before (0.x):

```
this.emit('data-changed', { value: 42 });
this.on('user-action', handler);
```

After (1.0):

```
this.pan.dispatch('data-changed', { value: 42 });
this.pan.subscribe('user-action', handler);
```

Migration Steps: 1. Replace `emit()` with `pan.dispatch()` 2. Replace `on()` with `pan.subscribe()` 3. Update cleanup to use returned unsubscribe functions 4. Add explicit PAN bus initialization if using custom buses

29.2.3 Attribute Handling

Attribute parsing became more strict in 1.0.

Before (0.x):

```
// Automatic JSON parsing
this.getAttribute('config'); // returns object
```

After (1.0):

```
// Explicit parsing required
JSON.parse(this.getAttribute('config') || '{}');
```

Migration Steps: 1. Add explicit JSON parsing for complex attributes 2. Implement `attributeChangedCallback()` for reactive updates 3. Use `observedAttributes` to declare monitored attributes

29.3 Version 1.x to 2.0 Migration

LARC 2.0 introduced TypeScript support, improved developer experience, and performance optimizations.

29.3.1 TypeScript Integration

While JavaScript remains fully supported, TypeScript brings type safety.

Migration Steps: 1. Install TypeScript definitions: `npm install --save-dev @larc/types` 2. Rename `.js` files to `.ts` incrementally 3. Add type annotations to component properties 4. Define custom event payload types

Example:

```
import { PANEvent } from '@larc/core';

interface UserData {
  id: string;
  name: string;
}

class UserCard extends HTMLElement {
  private userData: UserData | null = null;

  connectedCallback() {
    this.pan.subscribe<UserData>('user-selected', (event: PANEvent<UserData>) => {
      this.userData = event.detail;
      this.render();
    });
  }
}
```

29.3.2 Shadow DOM Adoption

Version 2.0 encouraged Shadow DOM for style encapsulation.

Before (1.x):

```
connectedCallback() {
  this.innerHTML = '<div class="container">Content</div>';
}
```

After (2.0):

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

```

this.shadowRoot.innerHTML = `
  <style>
    .container { padding: 1rem; }
  </style>
  <div class="container">Content</div>
`;
}

```

Migration Considerations:

- Global styles won't penetrate Shadow DOM
- Use CSS custom properties for theming
- Update selectors in tests to query shadow roots
- Consider performance impact for large component trees

29.3.3 Async Component Initialization

Version 2.0 added first-class async support.

Before (1.x):

```

connectedCallback() {
  fetch('/api/data').then(data => {
    this.data = data;
    this.render();
  });
}

```

After (2.0):

```

async connectedCallback() {
  await this.initialize();
}

async initialize() {
  try {
    this.data = await fetch('/api/data').then(r => r.json());
    this.render();
  } catch (error) {
    this.renderError(error);
  }
}

```

29.4 Version 2.x to 3.0 Migration

LARC 3.0 focused on performance, introducing reactive primitives and optimized rendering.

29.4.1 Reactive State Management

The new reactive state system replaces manual `render()` calls.

Before (2.x):

```
class Counter extends HTMLElement {
  constructor() {
    super();
    this.count = 0;
  }

  increment() {
    this.count++;
    this.render();
  }
}
```

After (3.0):

```
import { reactive } from '@larc/core';

class Counter extends HTMLElement {
  constructor() {
    super();
    this.state = reactive({ count: 0 });
    this.state.$watch(() => this.render());
  }

  increment() {
    this.state.count++; // automatically triggers render
  }
}
```

Migration Steps: 1. Wrap component state in `reactive()` 2. Set up `$watch()` for automatic rendering 3. Remove manual `render()` calls after state changes 4. Use `$batch()` for multiple simultaneous updates

29.4.2 PAN Bus Namespacing

Version 3.0 introduced event namespacing for better organization.

Before (2.x):

```
this.pan.dispatch('data-loaded', data);
this.pan.dispatch('data-error', error);
this.pan.dispatch('data-cleared');
```

After (3.0):

```
this.pan.dispatch('data:loaded', data);
this.pan.dispatch('data:error', error);
this.pan.dispatch('data:cleared');
```

```
// Subscribe to namespace
```

```
this.pan.subscribe('data:*', (event) => {
  console.log(`Data event: ${event.type}`);
});
```

29.4.3 Performance Optimizations

Version 3.0 added batched updates and render scheduling.

Manual Batching:

```
import { batch } from '@larc/core';

batch(() => {
  this.state.count++;
  this.state.name = 'Updated';
  this.state.timestamp = Date.now();
}); // Single render after all changes
```

Render Scheduling:

```
class HeavyComponent extends HTMLElement {
  render() {
    requestIdleCallback(() => {
      // Expensive rendering during idle time
      this.updateComplexUI();
    });
  }
}
```

29.5 Deprecation Timeline

29.5.1 Currently Deprecated (Remove in 4.0)

Legacy Event Syntax:

```
// Deprecated
this.pan.on('event-name', handler);

// Use instead
this.pan.subscribe('event-name', handler);
```

Global Bus Access:

```
// Deprecated
window.PAN.dispatch('event');

// Use instead
this.pan.dispatch('event');
```

Synchronous connectedCallback with async operations:

```
// Deprecated pattern
connectedCallback() {
  fetch('/data').then(d => this.data = d);
  this.render(); // Renders before data loads
}

// Preferred
async connectedCallback() {
  this.data = await fetch('/data').then(r => r.json());
  this.render();
}
```

29.5.2 Planned Deprecations (4.0+)

- Direct innerHTML manipulation (prefer template literals or JSX)
- Imperative event listener registration (favor declarative templates)
- String-based event names (transition to strongly-typed event enums)

29.6 Breaking Changes Checklist

When upgrading major versions, verify these common breaking change areas:

API Surface: - [] Component registration method - [] PAN bus method names - [] Event payload structure - [] Lifecycle callback signatures

Behavior Changes:

- ☐ Attribute parsing (automatic vs. manual)
- ☐ Default Shadow DOM usage
- ☐ Event bubbling and cancellation
- ☐ Async initialization timing

Build Process: - [] Bundler configuration - [] TypeScript compiler options - [] Test framework compatibility - [] Development server setup

Dependencies: - [] Peer dependency versions - [] Polyfill requirements - [] Browser compatibility targets - [] Third-party library compatibility

29.7 Migration Tools

29.7.1 Automated Refactoring

Use these tools to accelerate migration:

AST-Based Transforms:

```
npx @larc/migrate --from 2.x --to 3.0 src/**/*.js
```

Codemod Scripts:

```
// Example: Convert emit to dispatch
module.exports = function(fileInfo, api) {
  const j = api.jscodeshift;
  return j(fileInfo.source)
    .find(j.CallExpression, {
      callee: {
        object: { type: 'ThisExpression' },
        property: { name: 'emit' }
      }
    })
    .forEach(path => {
      path.value.callee.property.name = 'dispatch';
    })
    .toSource();
};
```

29.7.2 Manual Review Points

Automated tools can't catch everything. Manually review:

1. **Business Logic:** Ensure state changes behave identically
2. **Edge Cases:** Test error handling and boundary conditions
3. **Performance:** Profile before/after for regressions
4. **User Experience:** Verify visual consistency and interactions

29.8 Rollback Strategy

If migration causes critical issues:

1. **Revert Version:** Use git to restore previous package.json
2. **Isolate Changes:** Create feature branches for incremental updates
3. **Dual Implementation:** Run old and new code side-by-side with feature flags
4. **Gradual Rollout:** Deploy to subset of users before full migration

29.9 Getting Help

When stuck during migration:

- **Documentation:** Check version-specific upgrade guides at larc.dev/migrate
- **Community:** Ask in GitHub Discussions or Discord
- **Issue Tracker:** Search for similar migration problems
- **Support Contracts:** Enterprise users can access dedicated migration assistance

Migration is an investment in your application's future. Take time to understand changes, test thoroughly, and leverage community resources. The LARC team strives for smooth upgrade paths while continuing to evolve the framework.

Chapter 30

Recipes and Patterns

This appendix provides practical, copy-paste-ready solutions for common LARC development scenarios. Each recipe demonstrates a specific technique or pattern you'll encounter when building real applications. Use these as starting points, adapting them to your specific requirements.

30.1 Recipe 1: Lazy-Loading Components

Defer component loading until needed, reducing initial bundle size.

```
class LazyLoader extends HTMLElement {
  async connectedCallback() {
    const componentName = this.getAttribute('component');
    const modulePath = this.getAttribute('module');

    try {
      await import(modulePath);
      const element = document.createElement(componentName);
      Array.from(this.attributes).forEach(attr => {
        if (attr.name !== 'component' && attr.name !== 'module') {
          element.setAttribute(attr.name, attr.value);
        }
      });
      this.replaceWith(element);
    } catch (error) {
      this.innerHTML = `<div class="error">Failed to load component</div>`;
      console.error('Lazy load failed:', error);
    }
  }
}

customElements.define('lazy-loader', LazyLoader);
```

Usage:

```
<lazy-loader
  component="data-table"
  module="/components/data-table.js"
  data-source="/api/users">
</lazy-loader>
```

When to Use: - Large components used infrequently - Route-based code splitting - Conditional feature loading based on user permissions

30.2 Recipe 2: Form Validation Component

Reusable form validation with real-time feedback.

```
class ValidatedForm extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });
    this.validators = new Map();
    this.errors = new Map();

    this.shadowRoot.innerHTML = `
      <style>
        .field { margin-bottom: 1rem; }
        .error { color: #d32f2f; font-size: 0.875rem; margin-top: 0.25rem; }
        .valid { border-color: #4caf50; }
        .invalid { border-color: #d32f2f; }
      </style>
      <form>
        <slot></slot>
        <div class="actions">
          <button type="submit">Submit</button>
        </div>
      </form>
    `;

    this.setupValidation();
  }

  setupValidation() {
    const form = this.shadowRoot.querySelector('form');
    const inputs = this.querySelectorAll('[data-validate]');

    inputs.forEach(input => {
      const rules = input.getAttribute('data-validate').split(',');
      this.validators.set(input, rules);

      input.addEventListener('blur', () => this.validateField(input));
      input.addEventListener('input', () => {
        if (this.errors.has(input)) {

```

```

        this.validateField(input);
    }
});
});

form.addEventListener('submit', (e) => {
    e.preventDefault();
    if (this.validateAll()) {
        this.handleSubmit();
    }
});
}

validateField(input) {
    const rules = this.validators.get(input);
    const value = input.value.trim();
    let error = null;

    for (const rule of rules) {
        if (rule === 'required' && !value) {
            error = 'This field is required';
            break;
        }
        if (rule === 'email' && !this.isValidEmail(value)) {
            error = 'Invalid email address';
            break;
        }
        if (rule.startsWith('min:')) {
            const min = parseInt(rule.split(':')[1]);
            if (value.length < min) {
                error = `Minimum ${min} characters required`;
                break;
            }
        }
        if (rule.startsWith('max:')) {
            const max = parseInt(rule.split(':')[1]);
            if (value.length > max) {
                error = `Maximum ${max} characters allowed`;
                break;
            }
        }
    }

    this.updateFieldError(input, error);
    return !error;
}

```

```

updateFieldError(input, error) {
  input.classList.toggle('invalid', !!error);
  input.classList.toggle('valid', !error);

  let errorDiv = input.nextElementSibling;
  if (errorDiv && errorDiv.classList.contains('error')) {
    errorDiv.remove();
  }

  if (error) {
    this.errors.set(input, error);
    errorDiv = document.createElement('div');
    errorDiv.className = 'error';
    errorDiv.textContent = error;
    input.after(errorDiv);
  } else {
    this.errors.delete(input);
  }
}

validateAll() {
  let isValid = true;
  this.validators.forEach((rules, input) => {
    if (!this.validateField(input)) {
      isValid = false;
    }
  });
  return isValid;
}

isValidEmail(email) {
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
}

handleSubmit() {
  const formData = new FormData(this.querySelector('form'));
  this.pan.dispatch('form:submitted', Object.fromEntries(formData));
}

customElements.define('validated-form', ValidatedForm);

```

Usage:

```

<validated-form>
  <div class="field">
    <label>Email</label>
    <input type="email" name="email" data-validate="required,email">

```

```

</div>
<div class="field">
  <label>Password</label>
  <input type="password" name="password" data-validate="required,min:8">
</div>
</validated-form>

```

30.3 Recipe 3: Infinite Scroll List

Load data progressively as user scrolls.

```

class InfiniteList extends HTMLElement {
  constructor() {
    super();
    this.page = 1;
    this.loading = false;
    this.hasMore = true;
  }

  connectedCallback() {
    this.apiEndpoint = this.getAttribute('api');
    this.setupIntersectionObserver();
    this.loadMore();
  }

  setupIntersectionObserver() {
    const sentinel = document.createElement('div');
    sentinel.className = 'scroll-sentinel';
    this.appendChild(sentinel);

    this.observer = new IntersectionObserver((entries) => {
      if (entries[0].isIntersecting && !this.loading && this.hasMore) {
        this.loadMore();
      }
    }, { threshold: 0.1 });

    this.observer.observe(sentinel);
  }

  async loadMore() {
    this.loading = true;
    this.showLoadingIndicator();

    try {
      const response = await fetch(`${this.apiEndpoint}?page=${this.page}`);
      const data = await response.json();
    }
  }
}

```

```

    if (data.items.length === 0) {
      this.hasMore = false;
      this.hideLoadingIndicator();
      return;
    }

    this.renderItems(data.items);
    this.page++;
  } catch (error) {
    console.error('Failed to load items:', error);
    this.pan.dispatch('error', { message: 'Failed to load items' });
  } finally {
    this.loading = false;
    this.hideLoadingIndicator();
  }
}

renderItems(items) {
  const sentinel = this.querySelector('.scroll-sentinel');
  items.forEach(item => {
    const element = this.createItemElement(item);
    this.insertBefore(element, sentinel);
  });
}

createItemElement(item) {
  const div = document.createElement('div');
  div.className = 'list-item';
  div.innerHTML = `
    <h3>${item.title}</h3>
    <p>${item.description}</p>
  `;
  return div;
}

showLoadingIndicator() {
  let loader = this.querySelector('.loader');
  if (!loader) {
    loader = document.createElement('div');
    loader.className = 'loader';
    loader.textContent = 'Loading...';
    this.appendChild(loader);
  }
}

hideLoadingIndicator() {
  const loader = this.querySelector('.loader');

```

```

    if (loader) loader.remove();
  }

  disconnectedCallback() {
    if (this.observer) {
      this.observer.disconnect();
    }
  }
}

customElements.define('infinite-list', InfiniteList);

```

30.4 Recipe 4: Toast Notification System

Display temporary user notifications.

```

class ToastContainer extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          position: fixed;
          top: 1rem;
          right: 1rem;
          z-index: 10000;
          display: flex;
          flex-direction: column;
          gap: 0.5rem;
          max-width: 400px;
        }
        .toast {
          padding: 1rem 1.5rem;
          border-radius: 0.5rem;
          box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
          display: flex;
          align-items: center;
          gap: 0.75rem;
          animation: slideIn 0.3s ease;
        }
        .toast.success { background: #4caf50; color: white; }
        .toast.error { background: #f44336; color: white; }
        .toast.info { background: #2196f3; color: white; }
        .toast.warning { background: #ff9800; color: white; }
        @keyframes slideIn {
          from {
            transform: translateX(100%);
          }
        }
      </style>
    `;
  }
}

```

```

        opacity: 0;
      }
      to {
        transform: translateX(0);
        opacity: 1;
      }
    }
    .close {
      margin-left: auto;
      cursor: pointer;
      font-size: 1.25rem;
      opacity: 0.8;
    }
    .close:hover { opacity: 1; }
  </style>
`;

this.pan.subscribe('toast:show', (event) => {
  this.showToast(event.detail);
});
}

showToast({ message, type = 'info', duration = 3000 }) {
  const toast = document.createElement('div');
  toast.className = `toast ${type}`;
  toast.innerHTML = `
    <span class="message">${message}</span>
    <span class="close">&times;</span>
  `;

  toast.querySelector('.close').addEventListener('click', () => {
    this.removeToast(toast);
  });

  this.shadowRoot.appendChild(toast);

  if (duration > 0) {
    setTimeout(() => this.removeToast(toast), duration);
  }
}

removeToast(toast) {
  toast.style.animation = 'slideIn 0.3s ease reverse';
  setTimeout(() => toast.remove(), 300);
}
}

```

```
customElements.define('toast-container', ToastContainer);
```

Usage:

```
// Anywhere in your app
this.pan.dispatch('toast:show', {
  message: 'Settings saved successfully',
  type: 'success',
  duration: 3000
});
```

30.5 Recipe 5: Debounced Search Input

Optimize API calls by debouncing user input.

```
class SearchInput extends HTMLElement {
  constructor() {
    super();
    this.debounceTimer = null;
    this.debounceDelay = parseInt(this.getAttribute('debounce')) || 300;
  }

  connectedCallback() {
    this.innerHTML = `
      <div class="search-wrapper">
        <input type="search" placeholder="Search...">
        <span class="spinner" style="display: none;">[hourglass]</span>
      </div>
      <div class="results"></div>
    `;

    this.input = this.querySelector('input');
    this.spinner = this.querySelector('.spinner');
    this.resultsContainer = this.querySelector('.results');

    this.input.addEventListener('input', (e) => {
      this.handleInput(e.target.value);
    });

    this.pan.subscribe('search:results', (event) => {
      this.displayResults(event.detail);
    });
  }

  handleInput(value) {
    clearTimeout(this.debounceTimer);

    if (!value.trim()) {

```

```

        this.resultsContainer.innerHTML = '';
        return;
    }

    this.showSpinner();

    this.debounceTimer = setTimeout(() => {
        this.performSearch(value);
    }, this.debounceDelay);
}

async performSearch(query) {
    try {
        const apiEndpoint = this.getAttribute('api');
        const response = await fetch(`${apiEndpoint}?q=${encodeURIComponent(query)}`);
        const results = await response.json();
        this.pan.dispatch('search:results', results);
    } catch (error) {
        console.error('Search failed:', error);
    } finally {
        this.hideSpinner();
    }
}

displayResults(results) {
    if (results.length === 0) {
        this.resultsContainer.innerHTML = '<div class="no-results">No results found</div>';
        return;
    }

    this.resultsContainer.innerHTML = results
        .map(result => `<div class="result-item">${result.title}</div>`)
        .join('');
}

showSpinner() {
    this.spinner.style.display = 'inline';
}

hideSpinner() {
    this.spinner.style.display = 'none';
}
}

customElements.define('search-input', SearchInput);

```

30.6 Recipe 6: Modal Dialog

Accessible modal with focus trapping.

```
class ModalDialog extends HTMLElement {
  connectedCallback() {
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: none;
          position: fixed;
          top: 0;
          left: 0;
          right: 0;
          bottom: 0;
          z-index: 9999;
        }
        :host([open]) { display: block; }
        .backdrop {
          position: absolute;
          top: 0;
          left: 0;
          right: 0;
          bottom: 0;
          background: rgba(0, 0, 0, 0.5);
        }
        .modal {
          position: absolute;
          top: 50%;
          left: 50%;
          transform: translate(-50%, -50%);
          background: white;
          border-radius: 0.5rem;
          padding: 2rem;
          max-width: 90vw;
          max-height: 90vh;
          overflow: auto;
          box-shadow: 0 10px 25px rgba(0, 0, 0, 0.2);
        }
        .close {
          position: absolute;
          top: 1rem;
          right: 1rem;
          background: none;
          border: none;
          font-size: 1.5rem;
          cursor: pointer;
        }
      </style>
    `;
  }
}
```

```

    }
  </style>
  <div class="backdrop"></div>
  <div class="modal" role="dialog" aria-modal="true">
    <button class="close" aria-label="Close">&times;</button>
    <slot></slot>
  </div>
`;

this.shadowRoot.querySelector('.backdrop').addEventListener('click', () => this.close());
this.shadowRoot.querySelector('.close').addEventListener('click', () => this.close());

this.pan.subscribe('modal:open', (event) => {
  if (event.detail.id === this.id) {
    this.open();
  }
});
}

open() {
  this.setAttribute('open', '');
  this.previousFocus = document.activeElement;
  this.trapFocus();
  document.body.style.overflow = 'hidden';
}

close() {
  this.removeAttribute('open');
  document.body.style.overflow = '';
  if (this.previousFocus) {
    this.previousFocus.focus();
  }
  this.pan.dispatch('modal:closed', { id: this.id });
}

trapFocus() {
  const focusableElements = this.shadowRoot.querySelectorAll(
    'button, [href], input, select, textarea, [tabindex]:not([tabindex="-1"])'
  );
  const firstElement = focusableElements[0];
  const lastElement = focusableElements[focusableElements.length - 1];

  this.keydownHandler = (e) => {
    if (e.key !== 'Tab') return;

    if (e.shiftKey && document.activeElement === firstElement) {
      e.preventDefault();
    }
  }
}

```

```

        lastElement.focus();
    } else if (!e.shiftKey && document.activeElement === lastElement) {
        e.preventDefault();
        firstElement.focus();
    }
};

this.addEventListener('keydown', this.keydownHandler);
firstElement?.focus();
}

disconnectedCallback() {
    if (this.keydownHandler) {
        this.removeEventListener('keydown', this.keydownHandler);
    }
}
}

customElements.define('modal-dialog', ModalDialog);

```

30.7 Recipe 7: State Persistence

Save and restore component state to localStorage.

```

class StatefulComponent extends HTMLElement {
    constructor() {
        super();
        this.storageKey = this.getAttribute('storage-key') || 'component-state';
        this.state = this.loadState();
    }

    loadState() {
        try {
            const saved = localStorage.getItem(this.storageKey);
            return saved ? JSON.parse(saved) : this.getDefaultState();
        } catch (error) {
            console.error('Failed to load state:', error);
            return this.getDefaultState();
        }
    }

    saveState() {
        try {
            localStorage.setItem(this.storageKey, JSON.stringify(this.state));
            this.pan.dispatch('state:saved', { key: this.storageKey });
        } catch (error) {
            console.error('Failed to save state:', error);
        }
    }
}

```

```

        this.pan.dispatch('state:error', { error: error.message });
    }
}

updateState(updates) {
    this.state = { ...this.state, ...updates };
    this.saveState();
    this.render();
}

getDefaultState() {
    return {};
}

clearState() {
    localStorage.removeItem(this.storageKey);
    this.state = this.getDefaultState();
    this.render();
}
}

```

30.8 Recipe 8: Drag and Drop

Reorderable list with drag-and-drop.

```

class DraggableList extends HTMLElement {
    connectedCallback() {
        this.addEventListener('dragstart', this.handleDragStart.bind(this));
        this.addEventListener('dragover', this.handleDragOver.bind(this));
        this.addEventListener('drop', this.handleDrop.bind(this));
        this.addEventListener('dragend', this.handleDragEnd.bind(this));

        this.makeItemsDraggable();
    }

    makeItemsDraggable() {
        this.querySelectorAll('.draggable-item').forEach(item => {
            item.setAttribute('draggable', 'true');
        });
    }

    handleDragStart(e) {
        if (!e.target.classList.contains('draggable-item')) return;
        e.target.classList.add('dragging');
        e.dataTransfer.effectAllowed = 'move';
        e.dataTransfer.setData('text/html', e.target.innerHTML);
    }
}

```

```

handleDragOver(e) {
  e.preventDefault();
  e.dataTransfer.dropEffect = 'move';

  const dragging = this.querySelector('.dragging');
  const afterElement = this.getDragAfterElement(e.clientY);

  if (afterElement == null) {
    this.appendChild(dragging);
  } else {
    this.insertBefore(dragging, afterElement);
  }
}

handleDrop(e) {
  e.stopPropagation();
  this.dispatchReorderEvent();
}

handleDragEnd(e) {
  e.target.classList.remove('dragging');
}

getDragAfterElement(y) {
  const draggableElements = [
    ...this.querySelectorAll('.draggable-item:not(.dragging)')
  ];

  return draggableElements.reduce((closest, child) => {
    const box = child.getBoundingClientRect();
    const offset = y - box.top - box.height / 2;

    if (offset < 0 && offset > closest.offset) {
      return { offset: offset, element: child };
    } else {
      return closest;
    }
  }, { offset: Number.NEGATIVE_INFINITY }).element;
}

dispatchReorderEvent() {
  const order = Array.from(this.querySelectorAll('.draggable-item'))
    .map((item, index) => ({ index, id: item.dataset.id }));
  this.pan.dispatch('list:reordered', order);
}
}

```

```
customElements.define('draggable-list', DraggableList);
```

30.9 Recipe 9: Responsive Image

Automatically load appropriate image sizes.

```
class ResponsiveImage extends HTMLElement {
  connectedCallback() {
    this.sources = JSON.parse(this.getAttribute('sources'));
    this.alt = this.getAttribute('alt') || '';

    this.render();
    window.addEventListener('resize', () => this.handleResize());
  }

  render() {
    const src = this.selectSource();
    this.innerHTML = ``;
  }

  selectSource() {
    const width = window.innerWidth;
    const sorted = Object.entries(this.sources)
      .sort(([, a], [, b]) => parseInt(a) - parseInt(b));

    for (const [breakpoint, url] of sorted) {
      if (width <= parseInt(breakpoint)) {
        return url;
      }
    }

    return sorted[sorted.length - 1][1];
  }

  handleResize() {
    clearTimeout(this.resizeTimer);
    this.resizeTimer = setTimeout(() => {
      const currentSrc = this.querySelector('img').src;
      const newSrc = this.selectSource();
      if (currentSrc !== newSrc) {
        this.render();
      }
    }, 250);
  }
}

customElements.define('responsive-image', ResponsiveImage);
```

Usage:

```
<responsive-image
  sources='{"480": "/img/small.jpg", "1024": "/img/medium.jpg", "1920": "/img/large.jpg"}'
  alt="Product photo">
</responsive-image>
```

30.10 Recipe 10: Event Bus Bridge

Bridge LARC PAN bus events to external systems.

```
class EventBridge extends HTMLElement {
  connectedCallback() {
    this.externalSystem = this.getAttribute('target');
    this.eventMap = JSON.parse(this.getAttribute('event-map') || '{}');

    Object.keys(this.eventMap).forEach(panEvent => {
      this.pan.subscribe(panEvent, (event) => {
        this.bridgeEvent(panEvent, event.detail);
      });
    });
  }

  bridgeEvent(panEvent, data) {
    const externalEvent = this.eventMap[panEvent];

    switch (this.externalSystem) {
      case 'analytics':
        this.sendToAnalytics(externalEvent, data);
        break;
      case 'websocket':
        this.sendToWebSocket(externalEvent, data);
        break;
      case 'postmessage':
        this.sendToParent(externalEvent, data);
        break;
    }
  }

  sendToAnalytics(event, data) {
    if (window.gtag) {
      window.gtag('event', event, data);
    }
  }

  sendToWebSocket(event, data) {
    if (this.websocket?.readyState === WebSocket.OPEN) {
      this.websocket.send(JSON.stringify({ type: event, payload: data }));
    }
  }
}
```

```

    }
  }

  sendToParent(event, data) {
    window.parent.postMessage({ type: event, payload: data }, '*');
  }
}

customElements.define('event-bridge', EventBridge);

```

30.11 Common Patterns

30.11.1 Pattern: Component Composition

Build complex components from simpler ones.

```

class UserProfile extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <user-avatar user-id="${this.getAttribute('user-id')}"></user-avatar>
      <user-details user-id="${this.getAttribute('user-id')}"></user-details>
      <user-actions user-id="${this.getAttribute('user-id')}"></user-actions>
    `;
  }
}

```

30.11.2 Pattern: Higher-Order Components

Wrap components with additional functionality.

```

function withLoading(ComponentClass) {
  return class extends ComponentClass {
    connectedCallback() {
      this.showLoader();
      super.connectedCallback();
    }

    showLoader() {
      this.innerHTML = '<div class="loader">Loading...</div>';
    }
  };
}

customElements.define('user-card', withLoading(UserCard));

```

30.11.3 Pattern: Singleton Services

Share a single instance across components.

```

class DataCache {
    static instance = null;

    static getInstance() {
        if (!DataCache.instance) {
            DataCache.instance = new DataCache();
        }
        return DataCache.instance;
    }

    constructor() {
        this.cache = new Map();
    }

    get(key) {
        return this.cache.get(key);
    }

    set(key, value) {
        this.cache.set(key, value);
    }
}

```

30.12 Anti-Patterns to Avoid

30.12.1 Anti-Pattern: Tight Coupling

Bad:

```

class ComponentA extends HTMLElement {
    connectedCallback() {
        document.querySelector('component-b').doSomething();
    }
}

```

Good:

```

class ComponentA extends HTMLElement {
    connectedCallback() {
        this.pan.dispatch('action:requested', { data });
    }
}

```

30.12.2 Anti-Pattern: Massive Components

Bad: 500-line components handling everything.

Good: Break into focused, single-responsibility components.

30.12.3 Anti-Pattern: Ignoring Lifecycle

Bad:

```
class BadComponent extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = '<div>Content</div>'; // Too early!
  }
}
```

Good:

```
class GoodComponent extends HTMLElement {
  connectedCallback() {
    this.innerHTML = '<div>Content</div>';
  }
}
```

30.12.4 Anti-Pattern: Manual Memory Leaks

Bad:

```
connectedCallback() {
  this.pan.subscribe('event', handler);
  // Never unsubscribed!
}
```

Good:

```
connectedCallback() {
  this.unsubscribe = this.pan.subscribe('event', handler);
}

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

These recipes provide battle-tested solutions for common scenarios. Adapt them to your needs, understanding the principles behind each pattern. The best code is readable, maintainable, and solves the problem at hand without unnecessary complexity.

Chapter 31

Glossary

This glossary defines technical terms, LARC-specific concepts, and web standards references used throughout this manual. Terms are presented in alphabetical order with clear definitions and, where relevant, cross-references to related concepts.

31.1 A

Adapter A design pattern that converts one interface to another. In LARC contexts, adapters bridge LARC components with external libraries or non-standard APIs.

Attribute An HTML element property set via markup (e.g., `<my-component data-id="123">`). LARC components observe attributes through `observedAttributes` and respond to changes via `attributeChangedCallback()`.

Autonomous Custom Element A Web Component that extends `HTMLElement` directly rather than extending built-in HTML elements. LARC components are autonomous custom elements. Compare with *Customized Built-in Element*.

31.2 B

Batch Update An optimization technique that groups multiple state changes into a single render cycle, reducing unnecessary DOM operations and improving performance.

Binding The connection between a data source and its visual representation. LARC uses event-driven updates rather than automatic data binding, giving developers explicit control over rendering.

Browser Event Standard DOM events like `click`, `input`, or `submit`. LARC components listen to browser events and can translate them into PAN bus events for application-wide communication.

Bubble Event propagation through the DOM tree from child to parent elements. Browser events bubble by default; custom events must explicitly enable bubbling via `bubbles: true`.

31.3 C

Callback A function passed as an argument to another function, executed after a specific event or operation completes. LARC lifecycle methods (`connectedCallback`, `disconnectedCallback`) are

callbacks invoked by the browser at specific times.

Composed An event property that determines whether the event crosses Shadow DOM boundaries. Set via `composed: true` in event initialization. Essential for events that need to traverse shadow roots.

Component A self-contained, reusable user interface element. In LARC, components are Web Components registered via `customElements.define()` and implementing standard lifecycle callbacks.

Custom Element The Web Components standard for creating new HTML elements with custom behavior. LARC applications are built from custom elements. See also *Autonomous Custom Element*.

Customized Built-in Element A Web Component that extends an existing HTML element (e.g., `<button is="fancy-button">`). LARC primarily uses autonomous custom elements rather than customized built-ins.

31.4 D

Declarative A programming style that describes *what* should happen rather than *how*. HTML templates are declarative. Contrast with *Imperative*.

Dependency Injection A pattern where dependencies are provided to a component rather than created internally. LARC components receive the PAN bus reference rather than accessing a global singleton.

Dispatch Sending an event to the PAN bus for other components to receive. Called via `this.pan.dispatch(eventType, detail)`.

DOM (Document Object Model) The browser’s representation of an HTML document as a tree of objects. LARC components manipulate the DOM through standard APIs.

31.5 E

Element A node in the DOM tree representing an HTML tag. Custom elements are specialized elements with developer-defined behavior.

Emit Synonym for *dispatch*. Some frameworks use “emit” for event publication. LARC prefers “dispatch” to align with standard DOM terminology.

Encapsulation Hiding internal implementation details from external code. Shadow DOM provides style encapsulation; JavaScript class private fields provide data encapsulation.

Event A signal indicating something happened. Browser events (clicks, inputs) and custom events (PAN bus messages) both use the Event API.

Event Target Any object that can receive events and have listeners registered on it. All DOM nodes are event targets; the PAN bus is also an event target.

31.6 F

Fragment A `DocumentFragment` is a lightweight container for DOM nodes that can be manipulated off-screen and inserted into the document in one operation, reducing reflows.

Framework A comprehensive library providing structure and conventions for application development. LARC is a lightweight component architecture rather than a full framework, emphasizing web standards.

31.7 H

HTML Template The `<template>` element stores client-side content that won't render until explicitly instantiated. Useful for defining reusable markup structures.

Hydration The process of attaching event listeners and state to server-rendered HTML. LARC components hydrate automatically when defined via `customElements.define()`.

31.8 I

Imperative A programming style describing *how* to accomplish a task through explicit instructions. JavaScript is imperative. Contrast with *Declarative*.

Intersection Observer A browser API for efficiently detecting when elements enter or leave the viewport. Used for lazy loading, infinite scroll, and visibility tracking.

31.9 L

LARC (Lightweight Asynchronous Reactive Components) The component architecture described in this manual, emphasizing web standards, minimal abstraction, and explicit communication patterns.

Lifecycle The sequence of states a component passes through: creation, attachment to DOM, updates, and removal. LARC components implement standard Web Components lifecycle callbacks.

Lifecycle Callback Methods invoked by the browser at specific points in a component's lifecycle: `constructor()`, `connectedCallback()`, `disconnectedCallback()`, `attributeChangedCallback()`, `adoptedCallback()`.

Light DOM Regular DOM content, as opposed to Shadow DOM. Content placed inside a custom element's tags lives in the light DOM and can be redistributed via `<slot>`.

31.10 M

Microtask A JavaScript task scheduled via `Promise.then()` or `queueMicrotask()`. Microtasks run before the browser's next rendering cycle, useful for batching updates.

Module An ES6 module (`import/export`) that encapsulates code. LARC components are typically defined as modules, one component per file.

Mutation Observer A browser API for watching DOM changes. Less commonly needed in LARC since components manage their own rendering.

31.11 N

Namespace A prefix used to group related events or APIs. LARC encourages namespacing PAN bus events (e.g., `user:login`, `user:logout`) for better organization.

Node A basic DOM building block. Elements, text, and comments are all nodes. Components manipulate nodes through standard DOM APIs.

31.12 O

Observer Pattern A design pattern where objects (observers) subscribe to state changes in another object (subject). The PAN bus implements the observer pattern.

observedAttributes A static getter on custom element classes listing attributes the component wants to monitor. Changes trigger `attributeChangedCallback()`.

31.13 P

PAN Bus (Publish-and-subscribe Asynchronous Notification Bus) LARC’s event system for component communication. Components dispatch events to the bus and subscribe to events they care about, enabling loose coupling.

Polyfill JavaScript code that implements modern features in older browsers. Web Components polyfills enable LARC applications to run in browsers without native support.

Prop (Property) Short for “property,” data passed to a component. In LARC, complex data typically flows through PAN bus events rather than attributes, since attributes are limited to strings.

Publish-Subscribe A messaging pattern where publishers send messages to topics/channels, and subscribers receive messages from those topics. The PAN bus is a pub-sub system.

31.14 R

Reactive A programming model where the UI automatically updates in response to data changes. LARC components implement reactivity explicitly through PAN bus subscriptions rather than automatic binding.

Reconciliation The process of determining minimal DOM changes needed to reflect new state. LARC leaves reconciliation to developers or optional libraries rather than providing built-in virtual DOM diffing.

Render Convert data into visual representation. In LARC, rendering is explicit—components call their own `render()` methods when appropriate.

Reflow The browser’s process of recalculating element positions and dimensions. Excessive reflows hurt performance. LARC’s batch updates minimize reflows.

31.15 S

Scoped Styles CSS that applies only to a specific component without affecting other elements. Shadow DOM provides automatic style scoping.

Shadow DOM A web standard for attaching encapsulated DOM trees to elements. Shadow DOM provides style and markup encapsulation, preventing styles from leaking in or out.

Shadow Host The element to which a shadow root is attached. When you call `this.attachShadow()` on a custom element, that element becomes the shadow host.

Shadow Root The root of a shadow DOM tree, created via `element.attachShadow()`. Content inside the shadow root is isolated from the main document.

Slot A Shadow DOM feature for distributing light DOM content into shadow DOM. Defined with `<slot>` elements and allowing flexible content composition.

State Data that determines component appearance and behavior. LARC encourages explicit state management through component properties and PAN bus events.

Subscribe Registering a listener for events on the PAN bus. Called via `this.pan.subscribe(eventType, handler)`, returns an unsubscribe function.

31.16 T

Template Reusable markup structure. Can refer to HTML `<template>` elements or template literals (backtick strings) used for generating HTML.

Template Literal JavaScript's backtick string syntax supporting multiline strings and interpolation. Commonly used for component templates: ``<div>${value}</div>``.

Throttle Limiting function execution frequency. Unlike debouncing (which delays until activity stops), throttling ensures a function runs at most once per time interval.

31.17 U

Unsubscribe Removing a listener from the PAN bus. The function returned by `subscribe()` acts as an unsubscribe callback, essential for preventing memory leaks.

User Agent The browser or other software accessing a web application. User agent strings identify the browser type and version.

31.18 V

Virtual DOM An in-memory representation of the DOM used to calculate minimal changes before applying them. LARC doesn't include built-in virtual DOM, preferring explicit control or optional libraries.

31.19 W

Web Component An umbrella term for three standards: Custom Elements, Shadow DOM, and HTML Templates. LARC applications are built on Web Components.

Web Standards Specifications maintained by standards bodies (W3C, WHATWG) defining how web technologies work. LARC prioritizes web standards over proprietary abstractions.

31.20 LARC-Specific Terms

Component Bus A dedicated PAN bus instance for a specific component subtree. Allows isolated event scopes within larger applications. Most applications use a single global bus.

Component Tree The hierarchical structure of custom elements in an application. Events and data flow through this tree via the PAN bus.

Event Detail The `detail` property of a custom event, containing application-specific data. PAN bus events place their payload in the detail object.

Event Type A string identifying an event category (e.g., `'user:login'`, `'data:loaded'`). LARC encourages namespaced, descriptive event types.

Pan Property The `pan` property on custom elements, providing access to the PAN bus. Automatically injected by LARC's component initialization.

Reactive Primitive Basic reactive building blocks like reactive objects, computed values, and watchers. LARC's optional reactivity system provides these as lightweight utilities.

Unidirectional Data Flow An architecture where data flows in one direction through an application. LARC encourages this through PAN bus events: components dispatch actions upward and listen for state changes downward.

31.21 Web Standards References

CustomElementRegistry The browser's registry of defined custom elements, accessed via `window.customElements`. Provides `define()`, `get()`, `whenDefined()`, and `upgrade()` methods.

Event.prototype.composed Boolean property indicating whether an event crosses shadow DOM boundaries during event propagation.

Event.prototype.bubbles Boolean property indicating whether an event propagates up the DOM tree from its target.

HTMLElement The base interface for HTML elements. All LARC components extend `HTMLElement` or its subclasses.

MutationObserver API Interface for observing DOM mutations. Occasionally useful for LARC components that need to react to external DOM changes.

ShadowRoot Interface representing the root of a shadow DOM tree, providing methods like `querySelector()` that operate within the shadow scope.

shadowRoot.mode The encapsulation mode of a shadow root: `'open'` (accessible via `element.shadowRoot`) or `'closed'` (inaccessible from outside). LARC recommends open mode for testability.

31.22 Acronyms and Abbreviations

API - Application Programming Interface **CDN** - Content Delivery Network **CSS** - Cascading Style Sheets **DOM** - Document Object Model **ES6** - ECMAScript 2015 (JavaScript version) **HTML** - HyperText Markup Language **HTTP** - HyperText Transfer Protocol **JSX** - JavaScript XML (React's

template syntax, not part of LARC) **LARC** - Lightweight Asynchronous Reactive Components **MVC** - Model-View-Controller **NPM** - Node Package Manager **PAN** - Publish-and-subscribe Asynchronous Notification **REST** - Representational State Transfer **SPA** - Single-Page Application **SSR** - Server-Side Rendering **UI** - User Interface **URL** - Uniform Resource Locator **VDOM** - Virtual DOM **W3C** - World Wide Web Consortium **WHATWG** - Web Hypertext Application Technology Working Group

31.23 Related Concepts

Component Lifecycle See *Lifecycle* and *Lifecycle Callback*.

Custom Events Events created via `new CustomEvent()` rather than browser-generated events. PAN bus events are custom events.

Event-Driven Architecture An architectural pattern where components communicate through events rather than direct method calls. LARC's PAN bus enables event-driven architecture.

Loose Coupling Design principle where components depend on abstractions (event types) rather than concrete implementations (specific components), making systems more flexible and maintainable.

Separation of Concerns Design principle where different aspects of functionality are handled by different components. LARC components encapsulate specific UI concerns, communicating via well-defined events.

Single Responsibility Principle Each component should have one clear purpose. LARC encourages focused components that do one thing well.

31.24 Further Reading

MDN Web Docs (developer.mozilla.org) Comprehensive reference for Web APIs, including Web Components, DOM manipulation, and JavaScript features.

Web Components Specifications Official standards documents at w3.org and whatwg.org defining Custom Elements, Shadow DOM, and HTML Templates.

LARC Documentation Complete API reference and guides at larc.dev.

ECMAScript Specifications JavaScript language specifications at tc39.es.

This glossary covers core concepts needed to work effectively with LARC. For deeper exploration of specific topics, consult the main chapters of this manual and the reference materials listed above. Understanding these terms and their relationships helps you write clearer code, communicate more effectively with other developers, and leverage the full power of web standards in your applications.

Chapter 32

Resources

This appendix provides a curated collection of resources for learning, using, and extending LARC. Whether you're getting started, troubleshooting a problem, or contributing to the ecosystem, these links will help you find what you need.

32.1 Official Documentation

32.1.1 Primary Documentation

LARC Core Repository <https://github.com/larcjs/larc> The main LARC repository containing the core framework source code, examples, and technical documentation. This is the authoritative source for implementation details and includes the complete test suite.

LARC Components Library <https://github.com/larcjs/larc/tree/main/packages/components> Official component library with production-ready UI components, data components, integration components, and utilities. Each component includes comprehensive documentation and working examples.

API Reference <https://larcjs.com/api> Complete API documentation for all core classes, components, and utilities. Includes type definitions, method signatures, and interactive examples.

Getting Started Guide <https://larcjs.com/getting-started> Quick-start guide for new developers. Walks through installation, first application, and core concepts in 30 minutes.

32.1.2 Companion Books

Learning LARC The tutorial-focused companion to this reference manual. Organized around progressive learning with hands-on exercises, projects, and quizzes. Ideal for developers new to LARC or component-based architecture.

Building with LARC: A Reference Manual This book. Comprehensive reference covering all aspects of LARC development from core concepts to advanced patterns. Available online at <https://larcjs.com/reference>

32.2 Community Resources

32.2.1 Forums and Discussion

LARC Discussions (GitHub) <https://github.com/larcjs/larc/discussions> Official discussion forum for LARC developers. Ask questions, share projects, discuss patterns, and connect with other developers. Monitored by core maintainers.

Stack Overflow <https://stackoverflow.com/questions/tagged/larc> Tag: `larc` For technical troubleshooting and specific programming questions. Search existing questions before posting new ones.

Discord Community <https://discord.gg/zjUPsWTu> Real-time chat for LARC developers. Channels for beginners, advanced topics, component development, and off-topic discussion. Most active community hub.

Reddit `r/larcjs` <https://reddit.com/r/larcjs> Community-run subreddit for LARC news, showcases, and discussion. Good for project feedback and ecosystem updates.

32.2.2 Social Media

Twitter/X: `@larcjs` <https://twitter.com/larcjs> Official Twitter account for announcements, tips, and community highlights. Follow for news about releases, events, and ecosystem updates.

Mastodon: `@larcjs@fosstodon.org` <https://fosstodon.org/@larcjs> Official presence on the Fediverse for developers who prefer open platforms.

LinkedIn: LARC Developers Group <https://linkedin.com/groups/larcjs> Professional network for LARC developers. Good for job postings, industry discussion, and enterprise use cases.

32.3 Code Examples and Templates

32.3.1 Example Applications

Official Examples Repository <https://github.com/larcjs/larc/tree/main/packages/examples> Curated collection of example applications demonstrating LARC patterns and components. Each example is self-contained, documented, and includes setup instructions.

Notable examples include:

- Task Manager (state management, persistence)
- E-commerce Store (routing, forms, authentication)
- Real-time Chat (WebSocket integration, presence)
- File Manager (OPFS, drag-and-drop, uploads)
- Dashboard Builder (composable widgets, theming)

CodeSandbox Templates <https://codesandbox.io/search?refinementList%5Btags%5D=larc> Interactive online templates for rapid prototyping. Fork and experiment without local setup. Includes starter templates for common application types.

GitHub Topics: `#larcjs` <https://github.com/topics/larcjs> Community-contributed projects using LARC. Browse for inspiration, study real-world implementations, and discover reusable components.

32.3.2 Component Showcases

LARC Component Gallery <https://components.larcjs.com> Visual gallery of all official components with live demos, code samples, and customization tools. Essential reference when choosing components for your project.

Awesome LARC Components <https://github.com/larcjs/awesome-larc-components> Curated list of community-built components. Organized by category (UI, data, integration) with quality ratings and maintenance status.

32.4 Development Tools

32.4.1 Browser Extensions

LARC DevTools Chrome: <https://chrome.google.com/webstore/detail/larc-devtools> Firefox: <https://addons.mozilla.org/firefox/addon/larc-devtools> Browser extension providing visual PAN message inspection, component tree visualization, performance profiling, and state debugging.

Web Components DevTools General-purpose extension for debugging all Web Components, including LARC components. Useful for inspecting Shadow DOM and custom element lifecycles.

32.4.2 Editor Extensions

VS Code: LARC Extension <https://marketplace.visualstudio.com/items?itemName=larcjs.larc-vscode> Official VS Code extension providing:

- Component auto-completion
- PAN topic IntelliSense
- Snippet library for common patterns
- Integrated component browser
- Live component preview

JetBrains Plugin: LARC Support <https://plugins.jetbrains.com/plugin/larcjs-support> Support for WebStorm, IntelliJ IDEA, and other JetBrains IDEs. Provides code completion, navigation, and refactoring tools.

32.4.3 Command-Line Tools

LARC CLI <https://github.com/larcjs/larc/tree/main/cli>

```
$ npm install -g create-larc-app
```

Official command-line interface for:

- Project scaffolding (`larc create`)
- Component generation (`larc generate`)
- Development server with hot reload (`larc dev`)
- Production builds (`larc build`)
- Component registry integration (`larc add`)

create-larc-app <https://github.com/larcjs/larc/tree/main/cli>

```
$ npx create-larc-app my-app
```

Zero-configuration starter for new LARC projects. Includes pre-configured development environment, example components, and build tooling.

32.5 Learning Resources

32.5.1 Video Tutorials

LARC Fundamentals (YouTube) <https://youtube.com/playlist?list=PLarc-fundamentals> Official video series covering:

- Introduction to LARC (15 min)
- PAN Bus Messaging (22 min)
- Component Development (28 min)
- State Management Patterns (35 min)
- Building a Complete Application (1h 15min)

Egghead.io: Building with LARC <https://egghead.io/courses/building-with-larc> Professional screencast series (paid) with bite-sized lessons on specific topics. High production quality with accompanying code repositories.

Frontend Masters: LARC Workshop <https://frontendmasters.com/courses/larc> Full-day workshop covering LARC from fundamentals to advanced patterns. Includes exercises, quizzes, and downloadable resources.

32.5.2 Blog Posts and Articles

LARC Blog <https://blog.larcjs.com> Official blog with deep dives into architecture decisions, release notes, performance analysis, and best practices from core maintainers.

CSS-Tricks: LARC Guide Series <https://css-tricks.com/guides/larc> Multi-part guide covering LARC from a frontend developer's perspective. Excellent for understanding how LARC fits into modern web development.

Smashing Magazine: Component Architecture with LARC <https://smashingmagazine.com/larc-component-architecture> In-depth article comparing LARC's approach to other component frameworks. Good for understanding trade-offs and architectural decisions.

32.5.3 Podcasts

Syntax.fm: LARC Deep Dive <https://syntax.fm/show/larc-deep-dive> Popular web development podcast featuring LARC's creator discussing philosophy, implementation, and future direction.

ShopTalk Show: Building without Build Tools <https://shoptalkshow.com/larc-episode> Discussion of zero-build development philosophy and LARC's approach to modern web development.

32.6 Related Projects and Technologies

32.6.1 Web Components Standards

Web Components at MDN https://developer.mozilla.org/en-US/docs/Web/Web_Components Comprehensive documentation for Custom Elements, Shadow DOM, HTML Templates, and related browser APIs that LARC builds upon.

webcomponents.org <https://webcomponents.org> Community hub for Web Components with tutorials, best practices, and a component directory. Not LARC-specific but highly relevant.

Custom Elements Everywhere <https://custom-elements-everywhere.com> Test suite showing how different frameworks work with Web Components. Demonstrates LARC's excellent interoperability.

32.6.2 Message Bus Patterns

Enterprise Integration Patterns: Messaging <https://enterpriseintegrationpatterns.com/patterns/messaging> Classic reference for message-based architecture patterns. LARC implements many patterns described here adapted for browser environments.

Reactive Manifesto <https://reactivemanifesto.org> Principles of reactive system design that influenced LARC's architecture, particularly around message-driven communication.

32.6.3 Complementary Technologies

IndexedDB API https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API Browser database API used by LARC storage components for client-side persistence.

Origin Private File System (OPFS) https://developer.mozilla.org/en-US/docs/Web/API/File_System_Access_API Modern file system API supported by LARC file management components.

BroadcastChannel API https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API Cross-tab communication API used by LARC for multi-window synchronization.

32.7 Backend Integration

32.7.1 LARC-Compatible Backends

Node.js Backend Examples <https://github.com/larcjs/larc/tree/main/packages/examples/backends/nodejs> Reference implementations showing REST and WebSocket backends for LARC applications. Includes authentication, file uploads, and real-time features.

Python/Flask Backend Examples <https://github.com/larcjs/larc/tree/main/packages/examples/backends/python> Python backend examples demonstrating API design patterns that work well with LARC frontend applications.

Deno Backend Examples <https://github.com/larcjs/larc/tree/main/packages/examples/backends/deno> Modern JavaScript runtime examples showing how to build backends without Node.js dependencies.

32.7.2 API Design Guides

REST API Design for LARC Applications <https://larcjs.com/guides/rest-api-design> Best practices for designing REST APIs that integrate cleanly with LARC's data components and

message patterns.

WebSocket Integration Guide <https://larcjs.com/guides/websocket-integration> How to implement real-time features using WebSocket connections with LARC’s messaging system.

32.8 Testing and Quality

32.8.1 Testing Resources

LARC Testing Guide <https://larcjs.com/guides/testing> Official guide for testing LARC applications covering unit tests, integration tests, end-to-end tests, and visual regression testing.

Web Test Runner <https://modern-web.dev/docs/test-runner/overview> Recommended test runner for LARC applications. Fast, supports Web Components natively, and requires no browser driver.

Playwright <https://playwright.dev> End-to-end testing framework recommended for LARC application testing. Excellent Web Component support and debugging tools.

32.8.2 Performance Resources

Web Performance Working Group <https://w3c.github.io/web-performance> W3C standards for measuring and optimizing web performance. LARC follows these standards for component performance metrics.

web.dev Performance <https://web.dev/performance> Google’s comprehensive performance guide covering Core Web Vitals, optimization techniques, and measurement tools relevant to LARC applications.

32.9 Contributing and Extending

32.9.1 Contribution Guides

Contributing to LARC Core <https://github.com/larcjs/larc/blob/main/CONTRIBUTING.md> Guidelines for contributing to the LARC core framework. Includes coding standards, testing requirements, and pull request process.

Publishing Components <https://larcjs.com/guides/publishing-components> How to create, document, and publish reusable LARC components for the community. Covers naming conventions, versioning, and registry submission.

Component Development Guide <https://larcjs.com/guides/component-development> Best practices for building high-quality LARC components including accessibility, performance, and API design.

32.9.2 Governance and Roadmap

LARC Roadmap <https://github.com/larcjs/larc/blob/main/ROADMAP.md> Public roadmap showing planned features, architectural improvements, and long-term vision. Community feedback welcome.

RFC Process <https://github.com/larcjs/rfcs> Request for Comments process for proposing major changes to LARC. Review active RFCs and submit your own proposals.

Governance Model <https://github.com/larcjs/larc/blob/main/GOVERNANCE.md> How LARC is governed, who makes decisions, and how the community can participate in the project's direction.

32.10 Package Registries

32.10.1 NPM Packages

@larcjs/core <https://npmjs.com/package/@larcjs/core> Core framework package containing PAN bus, autoloader, and foundational components.

@larcjs/ui <https://npmjs.com/package/@larcjs/ui> Official component library with UI, data, and integration components.

@larcjs/core-types <https://npmjs.com/package/@larcjs/core-types> TypeScript type definitions for LARC core APIs and components.

@larcjs/testing-library <https://npmjs.com/package/@larcjs/testing-library> Testing utilities and helpers for LARC applications.

32.10.2 CDN Distributions

unpkg.com <https://unpkg.com/@larcjs/core@latest> Fast, global CDN for quick prototyping and development. Automatically serves latest versions.

jsDelivr <https://cdn.jsdelivr.net/npm/@larcjs/core@latest> Alternative CDN with excellent performance and reliability. Supports version pinning and package exploration.

LARC Official CDN <https://cdn.larcjs.com> Official CDN optimized for LARC with guaranteed uptime, geographic distribution, and versioned URLs.

32.11 Books and Long-Form Resources

32.11.1 Recommended Reading

Web Components: From Zero to Hero By Pascal Schilp Foundation knowledge for understanding the Web Components standards that LARC builds upon. Available free online.

Component-Based Development in JavaScript By Oliver Steele Explores component architecture patterns with examples in multiple frameworks including LARC. Good for understanding architectural trade-offs.

Event-Driven Architecture By Martin Fowler Classic software architecture text covering message-based patterns that inform LARC's design philosophy.

32.11.2 Academic Papers

Web Components: Standards, Patterns, and Best Practices Research paper analyzing Web Components adoption and patterns. Includes LARC case studies.

Message-Oriented Middleware for Browser Applications Academic treatment of message bus patterns in web applications with LARC as example implementation.

32.12 Deployment and Hosting

32.12.1 Hosting Platforms

Netlify <https://netlify.com> Recommended static hosting platform for LARC applications. Free tier suitable for most projects. Excellent CDN and deployment pipeline.

Vercel <https://vercel.com> Alternative hosting platform with Git integration, preview deployments, and serverless functions for backend features.

Cloudflare Pages <https://pages.cloudflare.com> Global edge network hosting with fast deployments and excellent performance. Good for international applications.

GitHub Pages <https://pages.github.com> Free hosting for open source projects. Simple deployment directly from GitHub repositories.

32.12.2 Deployment Guides

LARC Deployment Guide <https://larcjs.com/guides/deployment> Comprehensive guide covering deployment options, optimization strategies, caching configuration, and production best practices.

Performance Optimization Guide <https://larcjs.com/guides/performance-optimization> How to optimize LARC applications for production including code splitting, lazy loading, asset optimization, and CDN configuration.

32.13 Events and Training

32.13.1 Conferences

LARC Conf <https://conf.larcjs.com> Annual conference dedicated to LARC featuring talks, workshops, and networking. Recordings available online.

Web Components Summit <https://webcomponentssummit.com> General Web Components conference with LARC-specific tracks and presentations.

32.13.2 Workshops and Training

Official LARC Workshops <https://larcjs.com/workshops> In-person and virtual workshops taught by LARC experts. Topics range from fundamentals to advanced patterns.

Corporate Training <https://larcjs.com/training> Customized training programs for enterprise teams. Includes on-site workshops, consultation, and ongoing support.

32.14 Community Projects

32.14.1 Notable Applications Built with LARC

Browse <https://larcjs.com/showcase> for featured applications demonstrating LARC's capabilities in production environments.

32.14.2 Open Source Projects

LARC DevTools Browser extension and debugging toolkit (open source)

LARC Component Library Templates Starter templates for building your own component libraries

LARC Form Builder Visual form builder with code generation

LARC Dashboard Framework Composable dashboard system with widgets and layouts

32.15 Stay Updated

32.15.1 Newsletters

LARC Weekly <https://larcjs.com/newsletter> Weekly newsletter covering LARC news, tutorials, community projects, and ecosystem updates.

Web Components Weekly <https://webcomponents.dev/newsletter> General Web Components newsletter that frequently features LARC content.

32.15.2 Release Notes

LARC Changelog <https://github.com/larcjs/larc/blob/main/CHANGELOG.md> Detailed changelog for all LARC releases including breaking changes, new features, and bug fixes.

Security Advisories <https://github.com/larcjs/larc/security/advisories> Security announcements and vulnerability reports. Subscribe for critical updates.

32.16 Getting Help

When you need assistance:

1. **Search existing resources:** Check documentation, Stack Overflow, and GitHub Discussions first
2. **Prepare a minimal reproduction:** Create a CodeSandbox or GitHub repo demonstrating your issue
3. **Be specific:** Include LARC version, browser, error messages, and what you've already tried
4. **Choose the right channel:**
 - Technical questions -> Stack Overflow (tag: `larc`)
 - Bug reports -> GitHub Issues
 - General discussion -> GitHub Discussions or Discord
 - Real-time help -> Discord `#help` channel

32.16.1 Support Options

Community Support (Free) Discord, GitHub Discussions, Stack Overflow

Professional Support <https://larcjs.com/support> Commercial support plans available for enterprise users needing guaranteed response times and consulting.

This appendix is maintained by the LARC community. To suggest additions or corrections, submit a pull request to <https://github.com/larcjs/larc-docs> or open an issue describing the change.

Last updated: December 2025

Chapter 33

Index

33.1 A

addEventListener(), Chapter 4, Chapter 7 **Accessibility**, Chapter 15, Chapter 17, Chapter 23 **Acknowledgments**, Preface **Action topics**, Chapter 4, Appendix A **ActiveForm (todo status)**, Chapter 19 **adoptedStyleSheets**, Chapter 15 **Advanced patterns**, Chapter 19 **Alpine.js comparison**, Chapter 2 **Analytics tracking**, Chapter 4, Chapter 12 **Angular comparison**, Chapter 2 **Anti-patterns**, Chapter 4, Chapter 19 **Apache configuration**, Chapter 5, Chapter 20 **API design**, Chapter 11, Appendix G **API integration**, Chapter 11 **API reference**, Chapter 21-25, Appendix G **API topics**, Chapter 4, Chapter 11 **Application state**, Chapter 8 **Architectural decisions**, Chapter 2, Chapter 19 **Asynchronous patterns**, Chapter 6, Chapter 11 **Attributes (component)**, Chapter 4, Chapter 7, Chapter 21-25 **attributeChangedCallback()**, Chapter 7 **Authentication**, Chapter 12

- JWT tokens, Chapter 12
- OAuth integration, Chapter 12
- Session management, Chapter 12
- Token refresh, Chapter 12 **Authorization**, Chapter 12 **Auto-loading components**, Chapter 4, Chapter 5, Chapter 7 **Autoloader**, Chapter 4, Chapter 5, Chapter 7 **await**, Chapter 6, Chapter 11

33.2 B

Backend integration, Chapter 11, Appendix G

- Deno, Appendix G
- Node.js, Appendix G
- Python/Flask, Appendix G **Best practices**, Chapter 6-20 **Boolean attributes**, Chapter 7, Chapter 21-25 **Branching logic**, Chapter 19 **BroadcastChannel API**, Chapter 8, Chapter 13, Appendix G **Browser compatibility**, Chapter 5, Chapter 20
- Chrome, Chapter 5
- Edge, Chapter 5

- Firefox, Chapter 5
- Mobile browsers, Chapter 5
- Safari, Chapter 5 **Browser DevTools**, Chapter 5, Chapter 18
- Chrome DevTools, Chapter 5, Chapter 18
- Console panel, Chapter 5, Chapter 18
- Elements panel, Chapter 5
- Firefox Developer Tools, Chapter 5
- Network panel, Chapter 5, Chapter 18
- Safari Web Inspector, Chapter 5 **Bubbling (event)**, Chapter 4, Chapter 7 **Build tools**, Chapter 1, Chapter 2, Chapter 20
- Optional nature, Chapter 2
- Production optimization, Chapter 20
- Rollup, Chapter 20
- Vite, Chapter 5, Chapter 20
- Webpack, Chapter 20 **Bus statistics**, Chapter 4, Chapter 21

33.3 C

Caching, Chapter 11, Chapter 16, Chapter 20 **CAN bus (automotive)**, Chapter 1, Chapter 3
CDN deployment, Chapter 5, Chapter 20, Appendix G

- jsDelivr, Chapter 5, Appendix G
- unpkg, Chapter 5, Appendix G **Change detection**, Chapter 8 **Chrome DevTools Extension**, Chapter 5, Appendix G **Cleanup (subscription)**, Chapter 4, Chapter 7 **Client-side routing**, Chapter 9 **Cloudflare Pages**, Chapter 20, Appendix G **Code conventions**, Chapter 1 **Code examples**, Chapter 1, Appendix G **Code splitting**, Chapter 16, Chapter 20 **Command-line tools**, Chapter 5, Appendix G **Community resources**, Appendix G **Comparison with other frameworks**, Chapter 2 **Component API reference**, Chapter 21-25 **Component autoloading**, Chapter 4, Chapter 5, Chapter 7 **Component composition**, Chapter 4, Chapter 7 **Component development guide**, Appendix G **Component gallery**, Appendix G **Component lifecycle**, Chapter 7
- attributeChangedCallback, Chapter 7
- connectedCallback, Chapter 7
- disconnectedCallback, Chapter 7 **Component naming conventions**, Chapter 1, Chapter 7 **Component registration**, Chapter 7 **Component reusability**, Chapter 7, Chapter 19 **Component testing**, Chapter 17 **Composability**, Chapter 2, Chapter 4, Chapter 7 **Composition patterns**, Chapter 4, Chapter 7, Chapter 19 **Configuration**
- larc-config.mjs, Chapter 5

- pan-bus attributes, Chapter 4, Chapter 21
- Path configuration, Chapter 5 **connectedCallback()**, Chapter 7 **Console logging**, Chapter 4, Chapter 18 **Constructable Stylesheets**, Chapter 15 **Content Security Policy (CSP)**, Chapter 20 **Context API**, Chapter 8 **Contributing to LARC**, Appendix G **Convention over configuration**, Chapter 2, Chapter 5 **Core concepts**, Chapter 4 **Core Web Vitals**, Chapter 16, Appendix G **CORS errors**, Chapter 5 **correlationId**, Chapter 4, Chapter 6 **create-larc-app**, Chapter 5, Appendix G **Cross-origin images**, Chapter 14 **Cross-tab communication**, Chapter 8, Chapter 13 **CSS Custom Properties**, Chapter 15 **CSS encapsulation**, Chapter 7, Chapter 15 **Custom Elements**, Chapter 1, Chapter 4, Chapter 7
- v1 API, Chapter 7 **customElements.define()**, Chapter 7 **CustomEvent**, Chapter 4, Chapter 7

33.4 D

Dark mode, Chapter 15 **Dashboard applications**, Chapter 19 **Data components**, Chapter 22 **Data fetching**, Chapter 11

- Caching strategies, Chapter 11
- Error handling, Chapter 11
- Loading states, Chapter 11
- Pagination, Chapter 11 **Data validation**, Chapter 10 **Debugging**, Chapter 5, Chapter 18
- Browser DevTools, Chapter 5, Chapter 18
- Debug mode, Chapter 4, Chapter 21
- LARC DevTools extension, Appendix G
- Message tracing, Chapter 18, Chapter 21 **Decoupled architecture**, Chapter 2, Chapter 4 **Deduplication (messages)**, Chapter 4 **Deep linking**, Chapter 9 **Deployment**, Chapter 20
- CDN configuration, Chapter 20
- GitHub Pages, Chapter 20, Appendix G
- Netlify, Chapter 20, Appendix G
- Optimization, Chapter 20
- Static hosting, Chapter 20
- Vercel, Chapter 20, Appendix G **Design patterns**, Chapter 19 **Development environment setup**, Chapter 5 **Development server**, Chapter 5
- Live Server, Chapter 5
- PHP built-in, Chapter 5
- Python http.server, Chapter 5

- Vite, Chapter 5 **DevTools extension**, Chapter 5, Appendix G **Directory structure**, Chapter 5 **disconnectCallback()**, Chapter 7 **Discord community**, Appendix G **dispatchEvent()**, Chapter 4, Chapter 7 **Documentation resources**, Appendix G **DOM events**, Chapter 4, Chapter 7 **Drag and drop**, Chapter 14 **Dynamic imports**, Chapter 16, Chapter 20

33.5 E

E-commerce examples, Chapter 4, Appendix G **Editor configuration**, Chapter 5

- JetBrains, Appendix G
- Sublime Text, Chapter 5
- Vim, Chapter 5
- VS Code, Chapter 5, Appendix G **Emmet**, Chapter 5 **Encapsulation**, Chapter 7, Chapter 15 **End-to-end testing**, Chapter 17 **Enterprise Integration Patterns**, Appendix G **Error boundaries**, Chapter 18 **Error handling**, Chapter 18
- API errors, Chapter 11, Chapter 18
- Global handlers, Chapter 18
- User feedback, Chapter 18 **ES Modules**, Chapter 1, Chapter 5, Chapter 7 **ESLint**, Chapter 5 **Event delegation**, Chapter 7 **Event envelopes**, Chapter 4, Chapter 6, Appendix B **Event listeners**, Chapter 7 **Event topics**, Chapter 4, Appendix A **Event-driven architecture**, Chapter 2, Chapter 4 **Example applications**, Chapter 1, Chapter 5, Appendix G **Export/import**, Chapter 7

33.6 F

Feature detection, Chapter 5, Chapter 14 **Fetch API**, Chapter 11 **File management**, Chapter 14

- Downloads, Chapter 14
- Drag and drop, Chapter 14
- OPFS integration, Chapter 14
- Upload handling, Chapter 14 **File paths**, Chapter 1, Chapter 5 **Filtering (data)**, Chapter 22 **Firefox Developer Tools**, Chapter 5 **Form handling**, Chapter 10
- Accessibility, Chapter 10
- Custom validation, Chapter 10
- Multi-step forms, Chapter 10
- Submission, Chapter 10
- Validation, Chapter 10 **Frontend Masters**, Appendix G

33.7 G

Getting started, Chapter 5 **Git clone installation**, Chapter 5 **GitHub Discussions**, Appendix G **GitHub Pages**, Chapter 20, Appendix G **Global state**, Chapter 8 **Global wildcard subscriptions**, Chapter 4, Chapter 21 **Glossary**, Appendix F **Governance**, Appendix G

33.8 H

Hash routing, Chapter 9 **Headers (message)**, Chapter 4, Chapter 6 **Hello World example**, Chapter 5 **Hierarchical topics**, Chapter 4, Appendix A **History API**, Chapter 9 **Hot module replacement**, Chapter 2 **HTML Templates**, Chapter 7 **http-server**, Chapter 5

33.9 I

Icons, Chapter 23 **IDE support**, Chapter 5, Appendix G **Import maps**, Chapter 5, Chapter 20 **IndexedDB**, Chapter 8, Chapter 14, Appendix G **Infinite scroll**, Chapter 22 **Initial state loading**, Chapter 8 **Installation options**, Chapter 5

- **CDN**, Chapter 5
- **Git clone**, Chapter 5
- **NPM**, Chapter 5 **Integration components**, Chapter 24 **IntersectionObserver**, Chapter 4, Chapter 5, Chapter 16 **Introduction**, Chapter 1

33.10 J

JavaScript frameworks comparison, Chapter 2 **JetBrains plugin**, Appendix G **jsDelivr CDN**, Chapter 5, Appendix G **JSON serialization**, Chapter 4 **JWT authentication**, Chapter 12

33.11 K

Key concepts, Chapter 4

33.12 L

larc-config.mjs, Chapter 5 **LARC CLI**, Chapter 5, Appendix G **LARC philosophy**, Chapter 2 **LARC story**, Chapter 3 **Lazy loading**, Chapter 16, Chapter 20 **Learning path**, Chapter 1 **Learning LARC (book)**, Chapter 1, Appendix G **Lifecycle methods**, Chapter 7 **Lightweight architecture**, Chapter 1, Chapter 2 **Live Server**, Chapter 5 **Loading states**, Chapter 11, Chapter 22 **Local development**, Chapter 5 **Local state**, Chapter 8 **localStorage**, Chapter 5, Chapter 8 **Logging**, Chapter 18

33.13 M

Markdown editor, Chapter 23 **Memory management**, Chapter 4, Chapter 16, Chapter 21 **Message bus**, Chapter 4, Chapter 21

- Configuration, Chapter 4, Chapter 21
- Debug mode, Chapter 4
- Initialization, Chapter 5
- Rate limiting, Chapter 21
- Statistics, Chapter 4, Chapter 21 **Message envelope structure**, Chapter 4, Chapter 6, Appendix B **Message flow**, Chapter 6 **Message IDs**, Chapter 4 **Message lifecycle**, Chapter 4 **Message patterns**, Chapter 6, Chapter 19 **Message retention**, Chapter 4, Chapter 8 **Message routing**, Chapter 4, Chapter 21 **Message size limits**, Chapter 4, Chapter 21 **Message timestamps**, Chapter 4 **Message topics**, Chapter 1, Chapter 4, Appendix A **Message tracing**, Chapter 18, Chapter 21 **Message validation**, Chapter 4, Chapter 21 **Metadata (message)**, Chapter 4, Chapter 6 **Method signatures**, Chapter 1, Chapter 21-25 **Microservices pattern**, Chapter 19 **Migration guide**, Appendix D **MIME types**, Chapter 5 **Mobile browser support**, Chapter 5 **Modal dialogs**, Chapter 23 **Module loading**, Chapter 5, Chapter 7 **Multi-step forms**, Chapter 10 **Multi-tenant systems**, Chapter 4 **MutationObserver**, Chapter 16

33.14 N

Naming conventions

- Components, Chapter 1, Chapter 7
- Topics, Chapter 4, Appendix A **Navigation**, Chapter 9 **Netlify deployment**, Chapter 20, Appendix G **Network errors**, Chapter 11, Chapter 18 **Nginx configuration**, Chapter 5, Chapter 20 **Node.js backend**, Appendix G **Notifications**, Chapter 23 **NPM installation**, Chapter 5 **NPM packages**, Appendix G

33.15 O

OAuth integration, Chapter 12 **observedAttributes**, Chapter 7 **Offline support**, Chapter 14, Chapter 20 **Optimistic updates**, Chapter 11 **Optimization**, Chapter 16, Chapter 20

- Bundle size, Chapter 16, Chapter 20
- Code splitting, Chapter 16, Chapter 20
- Image optimization, Chapter 16
- Lazy loading, Chapter 16
- Performance, Chapter 16 **Origin Private File System (OPFS)**, Chapter 5, Chapter 14, Appendix G **O'Reilly conventions**, Chapter 1

33.16 P

Pagination, Chapter 11, Chapter 22 **PAN (Page Area Network)**, Chapter 1, Chapter 3, Chapter 4 **pan-bus component**, Chapter 4, Chapter 21

- Attributes, Chapter 21
- Configuration, Chapter 4, Chapter 21
- Events, Chapter 21

- Methods, Chapter 21 **pan-button component**, Chapter 23 **pan-card component**, Chapter 23 **pan-client API**, Chapter 4, Chapter 6 **pan-data-table component**, Chapter 22 **pan-form component**, Chapter 10 **pan-markdown-editor component**, Chapter 23 **pan-routes component**, Chapter 9, Chapter 21 **pan-storage component**, Chapter 8, Chapter 22 **pan-theme-provider component**, Chapter 15, Chapter 21 **pan-theme-toggle component**, Chapter 15, Chapter 21 **pan:deliver event**, Chapter 4, Chapter 6 **pan:publish event**, Chapter 4, Chapter 6 **pan:sys.ready event**, Chapter 5, Chapter 21 **pan:sys.stats**, Chapter 4, Chapter 21 **Pattern matching (topics)**, Chapter 4 **Performance optimization**, Chapter 16
- Benchmarking, Chapter 16
- Core Web Vitals, Chapter 16
- Lazy loading, Chapter 16
- Profiling, Chapter 16 **Persistence**, Chapter 8, Chapter 14 **Philosophy**, Chapter 2 **PHP server**, Chapter 5, Chapter 20 **Playwright testing**, Chapter 17, Appendix G **Plugin system**, Chapter 19 **Podcasts**, Appendix G **Polyfills**, Chapter 5 **Prerequisites**, Chapter 1, Chapter 5 **Production deployment**, Chapter 20 **Progressive enhancement**, Chapter 5 **Progressive Web Apps (PWA)**, Chapter 20 **Project structure**, Chapter 5 **Promises**, Chapter 6, Chapter 11 **Pub/sub pattern**, Chapter 4, Chapter 6 **publish() method**, Chapter 4, Chapter 6 **Publishing components**, Appendix G **Pull requests**, Appendix G **Python backend**, Appendix G **Python http.server**, Chapter 5

33.17 Q

Query parameters, Chapter 9 **QuotaExceededError**, Chapter 5, Chapter 8

33.18 R

Rate limiting, Chapter 21 **React comparison**, Chapter 2 **Reactive patterns**, Chapter 8, Appendix G **Real-time features**, Chapter 13

- Presence tracking, Chapter 13
- Server-Sent Events, Chapter 13
- WebSocket integration, Chapter 13 **Reddit community**, Appendix G **Redux comparison**, Chapter 2, Chapter 8 **References**, Appendix G **Refactoring**, Chapter 19 **Regex patterns**, Chapter 4 **Registration (component)**, Chapter 7 **Related projects**, Appendix G **Release notes**, Appendix G **Remote data**, Chapter 11 **Rendering optimization**, Chapter 16 **replyTo field**, Chapter 4, Chapter 6 **request() method**, Chapter 4, Chapter 6 **Request/reply pattern**, Chapter 4, Chapter 6 **ResizeObserver**, Chapter 16 **Resources**, Appendix G **REST API design**, Chapter 11, Appendix G **Retained messages**, Chapter 4, Chapter 8
- LRU eviction, Chapter 4
- Memory limits, Chapter 4

- State synchronization, Chapter 8 **Retry logic**, Chapter 11 **RFC process**, Appendix G **Roadmap**, Appendix G **Routing**, Chapter 9
- Client-side, Chapter 9
- Hash routing, Chapter 9
- History API, Chapter 9
- Message routing, Chapter 4
- Nested routes, Chapter 9
- Query parameters, Chapter 9

33.19 S

Safari Web Inspector, Chapter 5 **Sandbox mode**, Chapter 20 **Scaffolding tools**, Chapter 5, Appendix G **Scope (component)**, Chapter 7, Chapter 15 **Security**, Chapter 12, Chapter 20

- Authentication, Chapter 12
- Authorization, Chapter 12
- CSP, Chapter 20
- XSS prevention, Chapter 20 **Semantic routing**, Chapter 4 **Server-Sent Events (SSE)**, Chapter 13 **Service Workers**, Chapter 20 **Session management**, Chapter 12 **Setup**, Chapter 5 **Shadow DOM**, Chapter 1, Chapter 5, Chapter 7, Chapter 15
- CSS encapsulation, Chapter 15
- Debugging, Chapter 5
- Styling, Chapter 15 **Shopping cart example**, Chapter 4, Chapter 8 **Single Page Applications (SPA)**, Chapter 9 **Slot elements**, Chapter 7, Chapter 23 **Smashing Magazine**, Appendix G **Social media**, Appendix G **Software requirements**, Chapter 5 **Sorting (data)**, Chapter 22 **Stack Overflow**, Appendix G **State management**, Chapter 8
- Cross-tab sync, Chapter 8
- Local state, Chapter 8
- Persistent state, Chapter 8
- Shared state, Chapter 8
- State publisher pattern, Chapter 8 **State persistence**, Chapter 8 **State snapshots**, Chapter 4, Chapter 8 **State synchronization**, Chapter 4, Chapter 8 **Static hosting**, Chapter 20, Appendix G **Storage APIs**, Chapter 8, Chapter 14 **Story (LARC origin)**, Chapter 3 **Streaming data**, Chapter 13 **Style encapsulation**, Chapter 7, Chapter 15 **Styling components**, Chapter 15 **Sublime Text configuration**, Chapter 5 **subscribe() method**, Chapter 4, Chapter 6 **Subscription cleanup**, Chapter 4, Chapter 7 **Subscription patterns**, Chapter 4, Chapter 6 **Svelte comparison**, Chapter 2 **System themes**, Chapter 15

33.20 T

Tab synchronization, Chapter 8, Chapter 13 **Table components**, Chapter 22 **Task list example**, Chapter 4, Chapter 5 **Templates (HTML)**, Chapter 7 **Testing**, Chapter 17

- Component testing, Chapter 17
- E2E testing, Chapter 17
- Integration testing, Chapter 17
- Unit testing, Chapter 17
- Visual regression, Chapter 17 **Testing resources**, Appendix G **Theme switching**, Chapter 15 **Theming**, Chapter 15
- CSS Custom Properties, Chapter 15
- Dark mode, Chapter 15
- System preferences, Chapter 15 **Throttling**, Chapter 16 **Time-to-Interactive (TTI)**, Chapter 16 **Timestamps (message)**, Chapter 4 **Toast notifications**, Chapter 23 **TodoWrite patterns**, Chapter 19 **Topic conventions**, Chapter 1, Chapter 4, Appendix A **Topic hierarchies**, Chapter 4, Appendix A **Topic patterns**, Chapter 4 **Topic wildcards**, Chapter 4 **Tracing (message)**, Chapter 18, Chapter 21 **Training resources**, Appendix G **Tree shaking**, Chapter 20 **Troubleshooting**, Chapter 5, Chapter 18
- Component loading, Chapter 5
- CORS errors, Chapter 5
- Message delivery, Chapter 5
- Storage quota, Chapter 5
- Styling issues, Chapter 5 **TTL (Time To Live)**, Chapter 4 **Tutorial book**, Chapter 1, Appendix G **Twitter/X**, Appendix G **Type definitions**, Chapter 5, Appendix G **TypeScript support**, Chapter 1, Chapter 5 **Typographical conventions**, Chapter 1

33.21 U

UI components, Chapter 23 **Undo/redo**, Chapter 8, Chapter 19 **Unit testing**, Chapter 17 **unpkg CDN**, Chapter 5, Appendix G **Unsubscribe**, Chapter 4, Chapter 7 **Upload handling**, Chapter 14 **URL routing**, Chapter 9 **User authentication**, Chapter 12 **User input handling**, Chapter 10 **Utility components**, Chapter 25 **UUID generation**, Chapter 4

33.22 V

Validation - Form validation, Chapter 10 - Message validation, Chapter 4, Chapter 21 **Vercel deployment**, Chapter 20, Appendix G **Video tutorials**, Appendix G **Vim configuration**, Chapter 5 **Virtual DOM**, Chapter 2, Chapter 4 **Virtual scrolling**, Chapter 16, Chapter 22 **Visual regression testing**, Chapter 17 **Vite**, Chapter 5, Chapter 20 **Vue comparison**, Chapter 2 **VS Code extension**, Chapter 5, Appendix G

33.23 W

Web Components, Chapter 1, Chapter 4, Chapter 7, Appendix G

- Browser support, Chapter 5
- Custom Elements, Chapter 7
- HTML Templates, Chapter 7
- Shadow DOM, Chapter 7, Chapter 15
- Standards, Appendix G **Web Performance**, Chapter 16, Appendix G **Web Test Runner**, Chapter 17, Appendix G **web.dev**, Appendix G **webcomponents.org**, Appendix G **Web-Socket integration**, Chapter 13, Appendix G **WebStorm**, Chapter 5 **Webpack**, Chapter 1, Chapter 20 **Who should read this book**, Chapter 1 **Wildcard subscriptions**, Chapter 4, Chapter 21 **window.panClient**, Chapter 5, Chapter 6 ****window.__panReady**, **Chapter 5**, **Chapter 21 Workshops**, Appendix G

33.24 X

XSS prevention, Chapter 20

33.25 Y

YouTube tutorials, Appendix G

33.26 Z

Zero-build development, Chapter 1, Chapter 2, Chapter 5

- Philosophy, Chapter 2
- Workflow, Chapter 5

33.27 Appendices

Appendix A: Message Topic Conventions **Appendix B: Event Envelope Specification**
Appendix C: Configuration Reference **Appendix D: Migration Guides** **Appendix E:**
Code Recipes **Appendix F: Glossary** **Appendix G: Resources**

33.28 Components Quick Reference

Core Components (Chapter 21)

- pan-bus
- pan-theme-provider
- pan-theme-toggle
- pan-routes

Data Components (Chapter 22)

- pan-store
- pan-storage
- pan-data-table
- pan-list
- pan-filter
- pan-sort

UI Components (Chapter 23)

- pan-button
- pan-card
- pan-modal
- pan-toast
- pan-tabs
- pan-accordion
- pan-markdown-editor

Integration Components (Chapter 24)

- pan-http
- pan-websocket
- pan-sse
- pan-auth
- pan-analytics

Utility Components (Chapter 25)

- pan-logger
- pan-validator
- pan-debounce
- pan-throttle

This index references chapters and appendices by title. Page numbers would be added in print editions.

Chapter 34

Colophon

34.1 About the Cover Animal

The animal on the cover of *Building with LARC* is a **North American Beaver** (*Castor canadensis*), one of nature's most accomplished engineers and builders. Beavers are large, semi-aquatic rodents native to North America, known for their remarkable ability to modify their environment through the construction of dams, lodges, and canals.

Adult beavers typically weigh between 35-65 pounds and measure 3-4 feet in length, including their distinctive flat, paddle-shaped tail. Their dense, waterproof fur provides excellent insulation in cold water, while webbed hind feet make them powerful swimmers. The beaver's most recognizable features are its large, continuously growing incisors - sharp orange teeth that can fell trees up to 3 feet in diameter.

Beavers are legendary for their construction skills. Using branches, logs, mud, and stones, they build elaborate dams that create ponds and wetlands, fundamentally reshaping their ecosystem. These structures can span hundreds of feet and last for decades, sometimes even centuries with regular maintenance. The engineering is sophisticated: dams are typically built with a curved shape to better withstand water pressure, and beavers continuously monitor and repair their structures, plugging leaks and reinforcing weak points.

Their lodges - dome-shaped homes built from sticks and mud - feature underwater entrances for protection from predators, with living chambers positioned above the waterline. Inside, these chambers remain remarkably warm even during harsh winters, thanks to excellent insulation and the occupants' body heat.

Beavers work primarily at night and are highly industrious, with family groups collaborating on construction projects. They communicate through tail slaps on the water (warning signals), scent marking, and vocalizations. Their tree-felling technique is methodical: they gnaw around the trunk in an hourglass pattern until the tree falls, then section it into manageable pieces for transport and use.

We chose the Beaver for this book because, like these master builders, software architecture requires careful planning, solid engineering principles, and the ability to construct complex systems from simple components. The beaver's methodical approach to building - starting with a foundation, reinforcing structures, and creating systems that serve multiple purposes - mirrors the best practices

in software development. Just as beaver dams create thriving ecosystems, well-built applications create value for entire communities of users.

34.2 About the Cover Illustration

The cover illustration was hand-drawn by the author using pen and ink, photographed, and refined in Photoshop. The woodcut-style engraving technique echoes the natural history illustrations of 18th and 19th century field guides and the iconic animal covers of O'Reilly Media's technical books. The beaver is depicted sitting alertly on a log with wood chips nearby - a working builder surveying its domain - which seemed particularly fitting for a reference manual about building applications. The detailed cross-hatching captures the texture of the beaver's fur and the grain of the wood, reflecting both the precision of the craft and the artistry involved.

34.3 Fonts

The text typeface is Adobe Minion Pro, the display typeface is Myriad Pro, and the code typeface is Ubuntu Mono. The book was typeset using Pandoc and LaTeX.

34.4 Production Notes

This book was authored in Markdown using a modular structure with 25 chapters, 7 appendices, and a comprehensive index. The content was converted to multiple formats (HTML, PDF, and EPUB) using Pandoc. All code examples were tested against LARC version 3.x and validated for correctness.

The book follows O'Reilly's tradition of comprehensive technical documentation: starting with philosophy and context, moving through practical implementation, and concluding with exhaustive API reference material. The structure was inspired by classic programming references like *Programming Perl* and *Programming Python*, which balance narrative explanation with detailed technical specifications.

The build system itself was designed to embody LARC's philosophy: it works without complex toolchains, uses standard tools (Pandoc, LaTeX), and produces professional results with minimal configuration. In a meta sense, the build script is a small LARC application - taking structured inputs and producing multiple coordinated outputs.

34.5 Acknowledgments

Special thanks to the LARC community for their contributions, feedback, and real-world usage patterns that informed many sections of this book. Thanks also to Christopher Robison for his vision in creating LARC and his commitment to web standards and developer ergonomics.

If you find an error in this book, have suggestions for improvements, or want to contribute additional recipes and patterns, please visit our [GitHub repository](#) or contact the author. This book is a living document that grows with the LARC community.

Chapter 35

Learning LARC

35.1 The Web Has Grown Up. It's Time Our Apps Did Too.

Modern browsers aren't the brittle playgrounds they once were. They're fast, secure, richly capable application platforms — yet most of today's development stacks still treat them like dumb terminals that need layers of tooling, bundling, and framework magic just to function.

Learning LARC shows another path.

LARC embraces the browser as a mature runtime, using nothing but open standards — Web Components, modules, events, and message buses — to build complex, deeply interactive applications without build systems, without monoliths, and without ceremony. Through clear narrative examples and real architectural stories, this book teaches you how to design apps as ecosystems: small parts, clearly defined, communicating through a shared bus.

You'll learn how to structure large systems out of tiny cooperating modules, expose capabilities through message patterns instead of global state, keep your interfaces clean, and let the platform do the heavy lifting it was built for.

No bundlers. No scaffolding. No twenty-layer dependency stacks. Just the browser, finally treated like the grown-up it is.

Whether you're maintaining a legacy system or starting fresh, **Learning LARC** will help you rethink how modern web apps can — and should — be built.

35.2 About the Author

Christopher Robison is a veteran software engineer and architect with nearly three decades of experience building systems that range from biotech and online trading platforms to complex web applications and AI-driven tools. A lifelong maker with a deep appreciation for open standards, he has spent his career exploring the boundaries of what the web can do when you stop fighting the platform and start embracing it.

He is the creator of LARC.js and the PAN message bus, a browser-native architecture inspired by the elegant simplicity of the automotive CAN bus. His work blends engineering pragmatism with a

playful curiosity that has led him to design everything from 3D printers and robotics to interactive music systems and decentralized applications.

Christopher currently lives in San Francisco, where he continues to build things that bridge the digital and physical worlds — and occasionally sneaks off to play punk rock shows with his band.

Website: <https://larcjs.com>