Learning Svelte 5 with FlowWrite

A practical tutorial using a real-world codebase

Contents

1 Introduction

This tutorial teaches Svelte 5 using FlowWrite as our learning material. FlowWrite is a visual workflow editor for AI-assisted text processing — think "ComfyUI for text." By studying real production code, you'll learn patterns that actually work in practice.

1.1 What You'll Learn

- Svelte 5's new Runes API (`$state` , `$derived` , `$effect` , `$props` , `$bindable`)
- Component composition and communication
- Reactive state management
- Event handling and DOM interactions
- Styling with scoped CSS
- Integration with external libraries (@xyflow/svelte)
- TypeScript with Svelte

1.2 Project Structure

FlowWrite follows a clean architecture:

```
flow-write/src/
├── main.ts              # Entry point
├── App.svelte           # Root component
├── app.css              # Global styles
└── lib/
    ├── core/            # Business logic (pure TypeScript)
    ├── db/              # Database layer (IndexedDB)
    ├── api/             # LLM API client
    ├── components/      # Reusable UI components
    ├── nodes/           # Custom node types
    ├── edges/           # Custom edge types
    └── utils/           # Utility functions
```

2 Svelte 5 Runes: The New Reactivity System

Svelte 5 introduces "Runes" — special compiler instructions prefixed with `$`. Let's explore each one using FlowWrite examples.

2.1 `$state` — Reactive State

The `$state` rune creates reactive variables that trigger UI updates when changed.

From `FlowEditor.svelte`:

```
<script lang="ts">
  // Simple reactive state
  let darkMode = $state(false);
  let layoutDirection = $state<'TB' | 'LR'>('TB');

  // Object state with type annotation
  let contextMenu = $state({
    x: 0,
    y: 0,
    visible: false,
    type: 'node' as 'node' | 'edge',
    id: ''
  });

  // Array state with $state.raw for better performance
  // Use $state.raw when you replace the entire array rather than mutate it
  let nodes = $state.raw<Node[]>([
    {
      id: 'input-1',
      type: 'input',
      position: { x: 100, y: 200 },
      data: { label: 'User Input', inputType: 'text', isInput: true }
    }
  ]);
</script>
```

Key points:

- `$state(initialValue)` makes a variable reactive
- Use TypeScript generics for type safety: `$state<Type>(value)`
- `$state.raw()` skips deep reactivity — use when replacing whole objects/arrays

2.2 `$props` — Component Properties

The `$props` rune declares what data a component receives from its parent.

From `NavBar.svelte`:

```
<script lang="ts">
  // Declare props with $bindable for two-way binding
  let { activePage = $bindable('flow') } = $props();
</script>

<nav class="nav-bar">
  <button
    class:active={activePage === 'flow'}
    onclick={() => activePage = 'flow'}
  >
```

```
    Flow Editor
  </button>
</nav>
```

From `Toolbar.svelte` — Props with types:

```
<script lang="ts">
  let {
    selectedNodesCount = 0,
    selectedEdgesCount = 0,
    darkMode = false,
    layoutDirection = 'TB'
  }: {
    selectedNodesCount: number;
    selectedEdgesCount: number;
    darkMode: boolean;
    layoutDirection: 'TB' | 'LR';
  } = $props();
</script>
```

From `FlowCanvas.svelte` — Complex props:

```
<script lang="ts">
  let {
    nodes = $bindable(),      // Two-way binding
    edges = $bindable(),      // Two-way binding
    nodeTypes,                // One-way (read-only)
    edgeTypes,                // One-way (read-only)
    handleContextMenu         // Function prop
  }: {
    nodes: Node[];
    edges: Edge[];
    nodeTypes: Record<string, any>;
    edgeTypes: Record<string, any>;
    handleContextMenu: (event: MouseEvent, item: { type: 'node' | 'edge'; id: string })
=> void;
  } = $props();
</script>
```

2.3 `$bindable` — Two-Way Binding

Use `$bindable` when a child component needs to modify a parent's state.

Parent (App.svelte):

```
<script>
  let activePage = $state('flow');
</script>

<NavBar bind:activePage />
```

Child (NavBar.svelte):

```
<script lang="ts">
  let { activePage = $bindable('flow') } = $props();
  // Child can now modify activePage, and parent sees the change
</script>

<button onclick={() => activePage = 'api-test'}>
  LLM API Test
</button>
```

2.4 `$effect` — Side Effects

The `$effect` rune runs code when reactive dependencies change.

From `App.svelte` — Persisting state:

```
<script>
  import { onMount } from 'svelte';
  import { saveSetting, loadSetting, SETTINGS_KEYS } from './lib/db/settings';

  let activePage = $state('flow');
  let isLoaded = $state(false);

  // Load saved state on mount
  onMount(async () => {
    activePage = await loadSetting(SETTINGS_KEYS.PREFERENCES_ACTIVE_PAGE, 'flow');
    isLoaded = true;
  });

  // Save state whenever activePage changes
  $effect(() => {
    if (!isLoaded) return;  // Guard: don't save before initial load
    saveSetting(SETTINGS_KEYS.PREFERENCES_ACTIVE_PAGE, activePage);
  });
</script>
```

From `FlowEditor.svelte` — Setting up event listeners:

```
<script lang="ts">
  let contextMenu = $state({ x: 0, y: 0, visible: false, type: 'node', id: '' });

  $effect(() => {
    // This effect sets up a global click listener
    document.addEventListener('contextmenu', (e) => {
      if (!(e.target as HTMLElement).closest('.svelte-flow__node') &&
          !(e.target as HTMLElement).closest('.svelte-flow__edge')) {
        contextMenu.visible = false;
      }
    });

    // Run validation on mount
    validateAndAnalyze();
  });
</script>
```

2.5 `$derived` — Computed Values

Use `$derived` for values that automatically update when their dependencies change.

Conceptual example (pattern used in nodes):

```
<script lang="ts">
  let nodes = $state<Node[]>([]);
  let edges = $state<Edge[]>([]);

  // Automatically updates when nodes or edges change
  let nodeCount = $derived(nodes.length);
  let edgeCount = $derived(edges.length);
  let isEmpty = $derived(nodes.length === 0 && edges.length === 0);
</script>

<p>Graph has {nodeCount} nodes and {edgeCount} edges</p>
{#if isEmpty}
  <p>Drag nodes from the sidebar to get started!</p>
{/if}
```

3 Component Communication

Svelte offers multiple patterns for components to communicate.

3.1 Props Down, Events Up

The classic pattern: parents pass data down via props, children notify parents via events.

From `Toolbar.svelte`:

```
<script lang="ts">
  import { createEventDispatcher } from 'svelte';

  const dispatch = createEventDispatcher();

  function handleAction(action: string) {
    dispatch('action', action);  // Emit custom event
  }
</script>

<button onclick={() => handleAction('layout')}>
  <span>📐</span>
  <span>Layout</span>
</button>

<button onclick={() => handleAction('execute')}>
  <span>▶</span>
  <span>Run</span>
</button>
```

Parent (FlowEditor.svelte):

```
<script lang="ts">
  function handleToolbarAction(action: string) {
    switch (action) {
      case 'layout':
        autoLayout({ algorithm: 'dagre', direction: layoutDirection });
        break;
      case 'execute':
        executeWorkflow();
        break;
      case 'validate':
        validateAndAnalyze();
        alert(validationResult?.valid ? 'Valid!' : 'Issues found');
        break;
    }
  }
</script>

<Toolbar
  selectedNodesCount={selectedNodes.length}
  selectedEdgesCount={selectedEdges.length}
  {darkMode}
  {layoutDirection}
  onaction={(e: CustomEvent<string>) => handleToolbarAction(e.detail)}
/>
```

3.2 Two-Way Binding with `bind:`

For parent-child state synchronization, use `bind:`.

FlowEditor.svelte:

```
<FlowCanvas
  bind:nodes={nodes}
  bind:edges={edges}
  {nodeTypes}
  {edgeTypes}
  {handleContextMenu}
/>
```

FlowCanvas.svelte:

```
<script lang="ts">
  let {
    nodes = $bindable(),
    edges = $bindable(),
    // ...
  } = $props();

  function onconnect(connection: any) {
    // Modifying edges here updates the parent's state
    const newEdge: Edge = {
      id: `e-${connection.source}-${connection.target}`,
      source: connection.source,
      target: connection.target,
      type: 'smoothstep'
    };
    edges = [...edges, newEdge];
  }
</script>
```

3.3 Function Props

Pass functions as props for flexible communication.

FlowEditor.svelte:

```
<script lang="ts">
  function handleContextMenu(event: MouseEvent, item: { type: 'node' | 'edge'; id:
string }) {
    event.preventDefault();
    contextMenu = {
      x: event.clientX,
      y: event.clientY,
      visible: true,
      type: item.type,
      id: item.id
    };
  }
</script>

<FlowCanvas {handleContextMenu} />
```

FlowCanvas.svelte:

```
<script lang="ts">
  let { handleContextMenu } = $props();

  function handleNodeContextMenu(event: any) {
    handleContextMenu(event.originalEvent, { type: 'node', id: event.detail.id });
  }
</script>
```

4 Template Syntax

4.1 Conditional Rendering with `{#if}`

From `Toolbar.svelte`:

```
{#if selectedNodesCount > 0 || selectedEdgesCount > 0}
  <div class="toolbar-divider"></div>
  <div class="selection-info">
    <span>{selectedNodesCount} node{selectedNodesCount !== 1 ? 's' : ''}</span>
    <span>, </span>
    <span>{selectedEdgesCount} edge{selectedEdgesCount !== 1 ? 's' : ''}</span>
  </div>
{/if}
```

From `LLMNode.svelte`:

```
{#if data.model}
  <div class="node-field">
    <span class="field-label">Model:</span>
    <span class="field-value">{data.model}</span>
  </div>
{/if}
```

From `App.svelte` — if/else:

```
{#if activePage === 'flow'}
  <FlowEditor />
{:else}
  <ApiTest />
{/if}
```

4.2 Iteration with `{#each}`

From `NodeSidebar.svelte`:

```
<script lang="ts">
  const nodeTypes = [
    { type: 'input', label: 'Input', icon: '📥', color: '#f59e0b', description: 'Start
workflow' },
    { type: 'llm', label: 'LLM Node', icon: '🤖', color: '#0ea5e9', description: 'Call
LLM API' },
    { type: 'text', label: 'Text', icon: '📝', color: '#22c55e', description: 'Process
text' },
    { type: 'output', label: 'Output', icon: '📤', color: '#ec4899', description:
'Final output' }
  ];
</script>

<div class="node-list">
  {#each nodeTypes as nodeType}
    <div
      class="node-sidebar-item"
      draggable="true"
      ondragstart={(e) => handleDragStart(e, nodeType)}
```

```
    >
      <div class="node-item-header">
        <span class="node-icon" style="background: {nodeType.color}20; color:
{nodeType.color};">
          {nodeType.icon}
        </span>
        <span class="node-label">{nodeType.label}</span>
      </div>
      <p class="node-description">{nodeType.description}</p>
    </div>
  {/each}
</div>
```

4.3 Dynamic Classes with `class:`

From `LLMNode.svelte`:

```
<!-- Add 'selected' class when selected is true -->
<div class="custom-node" class:selected>
  <span class="custom-node-status {status}">{status}</span>
</div>
```

From `Toolbar.svelte`:

```
<button
  class="toolbar-button"
  class:active={layoutDirection === 'TB'}
  onclick={() => handleAction('layout-tb')}
>
  <span>⬇</span>
</button>
```

4.4 Dynamic Styles

From `NodeSidebar.svelte`:

```
<span
  class="node-icon"
  style="background: {nodeType.color}20; color: {nodeType.color};"
>
  {nodeType.icon}
</span>
```

5 Event Handling

5.1 DOM Events

Svelte uses `on` prefix for event handlers in Svelte 5.

From `NodeSidebar.svelte` — Drag and Drop:

```
<script lang="ts">
  function handleDragStart(event: DragEvent, nodeType: typeof nodeTypes[0]) {
    if (event.dataTransfer) {
      event.dataTransfer.setData('application/node-type', nodeType.type);
      event.dataTransfer.effectAllowed = 'copy';
    }
  }

  function handleDragEnd(event: DragEvent) {
    if (event.dataTransfer) {
      event.dataTransfer.clearData();
    }
  }
</script>

<div
  class="node-sidebar-item"
  draggable="true"
  ondragstart={(e) => handleDragStart(e, nodeType)}
  ondragend={handleDragEnd}
  role="button"
  tabindex="0"
>
```

From `FlowEditor.svelte` — Drop zone:

```
<script lang="ts">
  async function handleDrop(event: DragEvent) {
    event.preventDefault();

    const nodeType = event.dataTransfer?.getData('application/node-type');
    if (!nodeType) return;

    const reactFlowBounds = (event.currentTarget as
HTMLElement).getBoundingClientRect();
    const position = {
      x: event.clientX - reactFlowBounds.left,
      y: event.clientY - reactFlowBounds.top
    };

    const newNode: Node = {
      id: `${nodeType}-${Date.now()}`,
      type: nodeType,
      position,
      data: getDefaultNodeData(nodeType)
    };

    nodes = [...nodes, newNode];
  }
```

```
</script>

<div
  class="flow-canvas"
  ondrop={handleDrop}
  ondragover={(e: DragEvent) => e.preventDefault()}
  role="application"
>
```

5.2 Custom Events with `createEventDispatcher`

From `ContextMenu.svelte` (pattern):

```
<script lang="ts">
  import { createEventDispatcher } from 'svelte';

  const dispatch = createEventDispatcher<{
    action: { action: string; type: 'node' | 'edge'; id: string }
  }>();

  function handleAction(action: string) {
    dispatch('action', { action, type, id });
  }
</script>

<button onclick={() => handleAction('delete')}>Delete</button>
<button onclick={() => handleAction('duplicate')}>Duplicate</button>
```

Handling in parent:

```
<ContextMenu
  x={contextMenu.x}
  y={contextMenu.y}
  visible={contextMenu.visible}
  type={contextMenu.type}
  id={contextMenu.id}
  onaction={handleContextMenuAction}
/>

<script>
  function handleContextMenuAction(event: CustomEvent<{ action: string; type: 'node' |
'edge'; id: string }>) {
    const { action, type, id } = event.detail;

    switch (action) {
      case 'delete':
        if (type === 'node') {
          nodes = nodes.filter(n => n.id !== id);
          edges = edges.filter(e => e.source !== id && e.target !== id);
        } else {
          edges = edges.filter(e => e.id !== id);
        }
        break;
      case 'duplicate':
        // ... duplicate logic
        break;
```

```
    }
  }
</script>
```

6 Styling

6.1 Scoped Styles

Styles in `<style>` blocks are automatically scoped to the component.

From `NavBar.svelte`:

```
<nav class="nav-bar">
  <div class="nav-brand">FlowWrite</div>
  <div class="nav-links">
    <button class:active={activePage === 'flow'}>Flow Editor</button>
  </div>
</nav>

<style>
  /* These styles ONLY apply to this component */
  .nav-bar {
    display: flex;
    align-items: center;
    justify-content: space-between;
    height: 56px;
    padding: 0 24px;
    background: #1a1a2e;
  }

  .nav-brand {
    font-size: 18px;
    font-weight: 600;
    color: #e0e0e0;
  }

  button {
    padding: 8px 16px;
    background: transparent;
    border: none;
    border-radius: 6px;
    color: #a0a0a0;
    cursor: pointer;
    transition: all 0.2s;
  }

  button:hover {
    color: #e0e0e0;
    background: #2d2d44;
  }

  button.active {
    color: #fff;
    background: #3d3d5c;
  }
</style>
```

6.2 Global Styles with `:global()`

Use `:global()` to escape scoping when needed.

From `FlowCanvas.svelte`:

```
<style>
  /* This targets the .svelte-flow class globally */
  :global(.svelte-flow) {
    width: 100%;
    height: 100%;
  }
</style>
```

From `App.svelte`:

```
<style>
  :global(html) {
    margin: 0;
    padding: 0;
  }

  :global(body) {
    margin: 0;
    padding: 0;
    overflow: hidden;
  }
</style>
```

6.3 CSS Variables for Theming

From `app.css` — Define variables:

```
:root {
  --flow-bg: #ffffff;
  --flow-text: #1e293b;
  --flow-border: #e2e8f0;
  --flow-primary: #0ea5e9;
  --flow-success: #22c55e;
  --flow-warning: #f59e0b;
  --flow-error: #ef4444;
}

.dark {
  --flow-bg: #1a1a2e;
  --flow-text: #e2e8f0;
  --flow-border: #334155;
  --flow-primary: #38bdf8;
}
```

From `LLMNode.svelte` — Use variables:

```
<style>
  .custom-node {
    background: var(--flow-bg);
    border: 1px solid var(--flow-border);
  }

  .custom-node.selected {
    border-color: var(--flow-primary);
    box-shadow: 0 0 0 2px rgba(14, 165, 233, 0.3);
```

```
  }

  .custom-node-title {
    color: var(--flow-text);
  }
</style>
```

6.4 Conditional Class Styling

From `FlowEditor.svelte`:

```
<div class="flow-container" class:darkMode>
  <!-- content -->
</div>

<style>
  .flow-container {
    background: var(--flow-bg);
    transition: background-color 0.3s;
  }

  .flow-container.darkMode {
    --flow-bg: #1a1a2e;
    --flow-text: #e2e8f0;
    --flow-border: #334155;
  }
</style>
```

7 Working with External Libraries

FlowWrite uses @xyflow/svelte for the node-based editor. Here's how to integrate external libraries.

7.1 Importing and Using Library Components

From `FlowCanvas.svelte`:

```
<script lang="ts">
  import {
    SvelteFlow,
    Background,
    Controls,
    MiniMap,
    Panel,
    useSvelteFlow,
    BackgroundVariant,
    type Node,
    type Edge
  } from '@xyflow/svelte';

  // Import library CSS
  import '@xyflow/svelte/dist/style.css';

  // Use library hooks
  const { fitView } = useSvelteFlow();
</script>

<SvelteFlow
  bind:nodes={nodes}
  bind:edges={edges}
  {nodeTypes}
  {edgeTypes}
  {onconnect}
  fitView
  snapToGrid
  snapGrid={[15, 15]}
  defaultEdgeOptions={{ type: 'smoothstep', animated: true }}
>
  <Background variant={BackgroundVariant.Lines} gap={20} />
  <Controls />
  <MiniMap />
  <Panel position="top-right">
    <div class="info-panel">
      <span>{nodes.length} nodes</span>
    </div>
  </Panel>
</SvelteFlow>
```

7.2 Registering Custom Components

From `FlowEditor.svelte`:

```
<script lang="ts">
  import { LLMNode, TextNode, InputNode, OutputNode } from './nodes';
  import { LabeledEdge, CustomEdge } from './edges';

  // Register custom node types
```

```
  const nodeTypes = {
    llm: LLMNode,
    text: TextNode,
    input: InputNode,
    output: OutputNode
  };

  // Register custom edge types
  const edgeTypes = {
    labeled: LabeledEdge,
    custom: CustomEdge
  };
</script>

<SvelteFlowProvider>
  <FlowCanvas
    {nodeTypes}
    {edgeTypes}
    bind:nodes={nodes}
    bind:edges={edges}
  />
</SvelteFlowProvider>
```

7.3 Creating Custom Node Components

From `LLMNode.svelte`:

```
<script lang="ts">
  import { Handle, Position, type NodeProps } from '@xyflow/svelte';

  // Svelte 4 style props (library compatibility)
  type $$Props = NodeProps;

  export let data: $$Props['data'];
  export let selected: $$Props['selected'] = false;

  // Reactive declarations (Svelte 4 style for library compat)
  $: status = data.status || 'idle';
  $: label = data.label || 'LLM Node';
  $: provider = data.provider || 'OpenAI';
</script>

<div class="custom-node" class:selected>
  <div class="custom-node-header">
    <div class="custom-node-icon">🤖</div>
    <span class="custom-node-title">{label}</span>
    <span class="custom-node-status {status}">{status}</span>
  </div>

  <!-- Connection handles -->
  <Handle type="target" position={Position.Left} />
  <Handle type="source" position={Position.Right} />
</div>
```

8 Lifecycle and Async Operations

8.1 `onMount` for Initialization

From `App.svelte`:

```
<script>
  import { onMount } from 'svelte';
  import { loadSetting, SETTINGS_KEYS } from './lib/db/settings';

  let activePage = $state('flow');
  let isLoaded = $state(false);

  onMount(async () => {
    // Load saved preferences from IndexedDB
    activePage = await loadSetting(SETTINGS_KEYS.PREFERENCES_ACTIVE_PAGE, 'flow');
    isLoaded = true;
  });
</script>
```

8.2 Async Functions in Components

From `FlowEditor.svelte`:

```
<script lang="ts">
  import { layoutGraph, type LayoutOptions } from './utils';

  // Async function for auto-layout
  async function autoLayout(options: LayoutOptions) {
    nodes = await layoutGraph(nodes, edges, options);
  }

  function handleToolbarAction(action: string) {
    switch (action) {
      case 'layout':
        autoLayout({ algorithm: 'dagre', direction: layoutDirection });
        break;
    }
  }
</script>
```

8.3 Combining `$effect` with Async

Pattern for reactive persistence:

```
<script>
  let activePage = $state('flow');
  let isLoaded = $state(false);

  // Load on mount
  onMount(async () => {
    activePage = await loadSetting('activePage', 'flow');
    isLoaded = true;
  });

  // Save when value changes (after initial load)
  $effect(() => {
```

```
    if (!isLoaded) return;  // Prevent saving before loading
    saveSetting('activePage', activePage);
  });
</script>
```

9 TypeScript Integration

9.1 Typing Component Props

From `FlowCanvas.svelte`:

```
<script lang="ts">
  import { type Node, type Edge } from '@xyflow/svelte';

  let {
    nodes = $bindable(),
    edges = $bindable(),
    nodeTypes,
    edgeTypes,
    handleContextMenu
  }: {
    nodes: Node[];
    edges: Edge[];
    nodeTypes: Record<string, any>;
    edgeTypes: Record<string, any>;
    handleContextMenu: (event: MouseEvent, item: { type: 'node' | 'edge'; id: string })
=> void;
  } = $props();
</script>
```

9.2 Typing State

From `FlowEditor.svelte`:

```
<script lang="ts">
  import type { Node, Edge } from '@xyflow/svelte';

  // Typed array state
  let nodes = $state.raw<Node[]>([]);
  let edges = $state.raw<Edge[]>([]);

  // Typed object state
  let contextMenu = $state<{
    x: number;
    y: number;
    visible: boolean;
    type: 'node' | 'edge';
    id: string;
  }>({ x: 0, y: 0, visible: false, type: 'node', id: '' });

  // Union types
  let layoutDirection = $state<'TB' | 'LR'>('TB');

  // Nullable types
  let validationResult = $state<{ valid: boolean; errors: string[] } | null>(null);
</script>
```

9.3 Typing Custom Events

From `Toolbar.svelte`:

```
<script lang="ts">
  import { createEventDispatcher } from 'svelte';

  // Type the event dispatcher
  const dispatch = createEventDispatcher<{
    action: string;
  }>();

  function handleAction(action: string) {
    dispatch('action', action);
  }
</script>
```

10 Patterns from FlowWrite

10.1 Immutable State Updates

FlowWrite uses immutable patterns for predictable state management.

From `FlowEditor.svelte`:

```
<script lang="ts">
  function handleContextMenuAction(event: CustomEvent) {
    const { action, type, id } = event.detail;

    switch (action) {
      case 'delete':
        if (type === 'node') {
          // Create new arrays instead of mutating
          nodes = nodes.filter(n => n.id !== id);
          edges = edges.filter(e => e.source !== id && e.target !== id);
        } else {
          edges = edges.filter(e => e.id !== id);
        }
        break;

      case 'duplicate':
        if (type === 'node') {
          const node = nodes.find(n => n.id === id);
          if (node) {
            const newNode: Node = {
              ...node,  // Spread to copy
              id: `${node.id}-copy-${Date.now()}`,
              position: { x: node.position.x + 50, y: node.position.y + 50 },
              data: { ...node.data }  // Deep copy data
            };
            nodes = [...nodes, newNode];  // Create new array
          }
        }
        break;
    }
  }
</script>
```

10.2 Separation of Concerns

FlowWrite separates pure logic from UI components.

Core logic in `lib/core/textblock.ts`:

```
// Pure functions — no Svelte, no side effects
export function createTextBlock(content: string = ''): TextBlock {
  return {
    type: 'text',
    id: generateId(),
    content
  };
}

export function appendBlock(list: TextBlockList, block: AnyTextBlock): TextBlockList {
  return {
```

```
    ...list,
    blocks: [...list.blocks, block]
  };
}
```

Database layer in `lib/db/settings.ts`:

```
// Async operations isolated from components
export async function saveSetting<T>(key: string, value: T): Promise<void> {
  const record: SettingsRecord = {
    key,
    value: JSON.stringify(value),
    updatedAt: Date.now()
  };
  await db.settings.put(record);
}

export async function loadSetting<T>(key: string, defaultValue: T): Promise<T> {
  const record = await db.settings.get(key);
  if (!record) return defaultValue;
  try {
    return JSON.parse(record.value) as T;
  } catch {
    return defaultValue;
  }
}
```

10.3 Component Composition

App.svelte — Page routing:

```
<script>
  let activePage = $state('flow');
</script>

<div class="app">
  <div class="navbar">
    <NavBar bind:activePage />
  </div>
  <div class="content">
    {#if activePage === 'flow'}
      <FlowEditor />
    {:else}
      <ApiTest />
    {/if}
  </div>
</div>

<FloatingBall />
```

FlowEditor.svelte — Nested composition:

```
<div class="flow-container">
  <NodeSidebar />
```

```
  <div class="flow-main">
    <Toolbar
      selectedNodesCount={selectedNodes.length}
      {darkMode}
      {layoutDirection}
      onaction={handleToolbarAction}
    />

    <div class="flow-canvas">
      <SvelteFlowProvider>
        <FlowCanvas
          bind:nodes
          bind:edges
          {nodeTypes}
          {edgeTypes}
          {handleContextMenu}
        />
      </SvelteFlowProvider>
    </div>
  </div>

  <ContextMenu {contextMenu} onaction={handleContextMenuAction} />
</div>
```

11 Summary: Svelte 5 vs Svelte 4

Concept	Svelte 4	Svelte 5
Reactive variable	`let x = 0`	`let x = $state(0)`
Computed value	`$: doubled = x * 2`	`let doubled = $derived(x * 2)`
Side effect	`$: console.log(x)`	`$effect(() => console.log(x))`
Component props	`export let name`	`let { name } = $props()`
Two-way bindable	`export let value`	`let { value = $bindable() } = $props()`

11.1 Key Takeaways

1. **Use `$state` for reactive variables** — The foundation of Svelte 5 reactivity
2. **Use `$state.raw` for large arrays/objects** — Better performance when replacing wholesale
3. **Use `$derived` for computed values** — Automatically tracks dependencies
4. **Use `$effect` for side effects** — Runs after DOM updates
5. **Use `$props` and `$bindable` for component communication**
6. **Immutable updates trigger reactivity** — Create new objects/arrays instead of mutating
7. **Separate concerns** — Pure logic in `.ts` files, UI in `.svelte` files
8. **Scope styles by default** — Use `:global()` only when necessary
9. **Use CSS variables for theming** — Easy dark mode support

11.2 Next Steps

1. Explore the FlowWrite codebase to see these patterns in action
2. Try modifying components to understand reactivity
3. Add new node types following the `LLMNode.svelte` pattern
4. Experiment with the `$effect` cleanup function for subscriptions
5. Build your own workflow nodes!