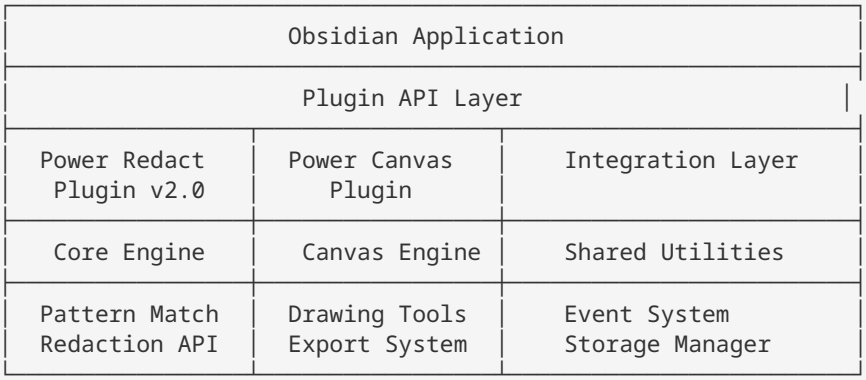
Architecture Overview - Power Components

Comprehensive system architecture documentation for the Power Components suite.

🏗️ System Architecture

High-Level Overview



🔒 Power Redact Plugin Architecture

Component Structure

```
Power Redact Plugin v2.0
├── Core Engine
│   ├── Pattern Detection System
│   ├── Redaction Processing Engine
│   └── Undo/Redo Manager
├── UI Components
│   ├── Settings Panel
│   ├── Redaction Modal
│   └── Status Indicators
├── API Layer
│   ├── Plugin API Interface
│   ├── Command Registration
│   └── Event Handlers
└── Storage System
    ├── Pattern Storage
    ├── Settings Storage
    └── Cache Management
```

Core Components

1. Pattern Detection System

```
interface PatternDetector {
  patterns: Map<string, RegExp>;
  detectPatterns(text: string): DetectionResult[];
  addCustomPattern(name: string, pattern: RegExp): void;
  removePattern(name: string): void;
}

class PatternDetector implements IPatternDetector {
  private builtInPatterns = {
    ssn: /\b\d{3}-\d{2}-\d{4}\b/g,
    email: /\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b/g,
    phone: /\b\d{3}-\d{3}-\d{4}\b/g,
    creditCard: /\b\d{4}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}\b/g
  };
}
```

2. Redaction Processing Engine

```
interface RedactionEngine {
  redactText(text: string, patterns: string[]): RedactionResult;
  applyStyle(match: string, style: RedactionStyle): string;
  batchProcess(files: TFile[]): Promise<BatchResult>;
}

enum RedactionStyle {
  BLACKOUT = 'blackout',
  BLUR = 'blur',
  HASH = 'hash',
  CUSTOM = 'custom'
}
```

3. Undo/Redo Manager

```
interface UndoRedoManager {
  pushState(state: EditorState): void;
  undo(): EditorState | null;
  redo(): EditorState | null;
  clearHistory(): void;
}
```

Data Flow

```
User Input → Pattern Detection → Redaction Engine → UI Update
     ↓              ↓                    ↓              ↓
Settings ←→ Pattern Storage ←→ Redaction Cache ←→ Undo Stack
```

🎨 Power Canvas Plugin Architecture

Component Structure

```
Power Canvas Plugin
├── Canvas Engine
│   ├── Rendering System
│   ├── Drawing Tools Manager
│   └── Layer Management
├── Interactive Elements
│   ├── Shape Tools
│   ├── Annotation System
│   └── Selection Manager
├── Export System
│   ├── Format Handlers (PNG, SVG, PDF)
│   ├── Quality Settings
│   └── Batch Export
└── Collaboration Layer
    ├── Real-time Sync
    ├── Conflict Resolution
    └── Version Control
```

Core Components

1. Canvas Engine

```typescript
interface CanvasEngine {
  canvas: HTMLCanvasElement;
  context: CanvasRenderingContext2D;
  layers: Layer[];

  render(): void;
  addLayer(layer: Layer): void;
  removeLayer(id: string): void;
  exportCanvas(format: ExportFormat): Promise<Blob>;
}

class CanvasEngine implements ICanvasEngine {
  private renderLoop(): void {
    requestAnimationFrame(() => {
      this.clearCanvas();
      this.renderLayers();
      this.renderLoop();
    });
  }
}
```

2. Drawing Tools Manager

```
interface DrawingTool {
  name: string;
  icon: string;
  cursor: string;

  onMouseDown(event: MouseEvent): void;
  onMouseMove(event: MouseEvent): void;
  onMouseUp(event: MouseEvent): void;
}

class PenTool implements DrawingTool {
  private path: Point[] = [];
  private isDrawing = false;
}
```

3. Layer Management

```
interface Layer {
  id: string;
  name: string;
  visible: boolean;
  opacity: number;
  blendMode: BlendMode;
  elements: CanvasElement[];
}

class LayerManager {
  private layers: Layer[] = [];
  private activeLayer: Layer;

  addLayer(layer: Layer): void;
  removeLayer(id: string): void;
  reorderLayers(fromIndex: number, toIndex: number): void;
}
```

Rendering Pipeline

```
User Input → Tool Handler → Canvas Update → Layer Render → Display
     ↓            ↓              ↓              ↓            ↓
Tool State → Drawing Buffer → Layer Buffer → Composite → Screen
```

🔗 Integration Layer Architecture

Shared Components

1. Event System

```
interface EventBus {
  subscribe<T>(event: string, handler: (data: T) => void): void;
  unsubscribe(event: string, handler: Function): void;
  emit<T>(event: string, data: T): void;
}

// Cross-plugin communication
class PowerComponentsEventBus implements EventBus {
  private events = new Map<string, Function[]>();

  // Enable communication between plugins
  bridgePlugins(redactPlugin: PowerRedactPlugin, canvasPlugin: PowerCanvasPlugin):
void;
}
```

2. Storage Manager

```
interface StorageManager {
  saveSettings(pluginId: string, settings: any): Promise<void>;
  loadSettings(pluginId: string): Promise<any>;
  clearSettings(pluginId: string): Promise<void>;
}

class UnifiedStorageManager implements StorageManager {
  private obsidianAdapter: ObsidianStorageAdapter;
  private cache: Map<string, any> = new Map();
}
```

3. Utility Functions

```
namespace PowerComponentsUtils {
  export function sanitizeText(text: string): string;
  export function validateRegex(pattern: string): boolean;
  export function formatFileSize(bytes: number): string;
  export function debounce<T extends Function>(func: T, delay: number): T;
  export function throttle<T extends Function>(func: T, limit: number): T;
}
```

Plugin Communication

```typescript
// Cross-plugin integration example
interface PluginBridge {
  redactAndCanvas(content: string): Promise<CanvasData>;
  canvasWithRedaction(canvas: CanvasData, patterns: string[]): Promise<CanvasData>;
}

class PowerComponentsBridge implements PluginBridge {
  constructor(
    private redactPlugin: PowerRedactPlugin,
    private canvasPlugin: PowerCanvasPlugin
  ) {}

  async redactAndCanvas(content: string): Promise<CanvasData> {
    const redacted = await this.redactPlugin.processText(content);
    return await this.canvasPlugin.createFromText(redacted);
  }
}
```

📊 Data Models

Power Redact Data Models

```typescript
interface RedactionPattern {
  id: string;
  name: string;
  pattern: string;
  flags: string;
  enabled: boolean;
  style: RedactionStyle;
  replacement?: string;
}

interface RedactionResult {
  originalText: string;
  redactedText: string;
  matches: RedactionMatch[];
  timestamp: number;
}

interface RedactionMatch {
  text: string;
  start: number;
  end: number;
  pattern: string;
  style: RedactionStyle;
}
```

Power Canvas Data Models

```
interface CanvasData {
  id: string;
  name: string;
  width: number;
  height: number;
  layers: Layer[];
  metadata: CanvasMetadata;
}

interface CanvasElement {
  id: string;
  type: ElementType;
  position: Point;
  properties: ElementProperties;
  style: ElementStyle;
}

interface Point {
  x: number;
  y: number;
}

interface ElementStyle {
  color: string;
  strokeWidth: number;
  opacity: number;
  blendMode: BlendMode;
}
```

🔄 State Management

Plugin State Architecture

```
interface PluginState {
  settings: PluginSettings;
  runtime: RuntimeState;
  cache: CacheState;
}

class StateManager<T extends PluginState> {
  private state: T;
  private subscribers: StateSubscriber<T>[] = [];

  setState(newState: Partial<T>): void {
    this.state = { ...this.state, ...newState };
    this.notifySubscribers();
  }

  getState(): T {
    return { ...this.state };
  }
}
```

State Synchronization

```
Plugin A State ←→ Event Bus ←→ Plugin B State
       ↓                              ↓
   Local Storage ←→ Sync Manager ←→ Local Storage
```

🚀 Performance Considerations

Optimization Strategies

1. Lazy Loading

```typescript
class LazyComponentLoader {
  private components = new Map<string, () => Promise<any>>();

  register(name: string, loader: () => Promise<any>): void {
    this.components.set(name, loader);
  }

  async load(name: string): Promise<any> {
    const loader = this.components.get(name);
    return loader ? await loader() : null;
  }
}
```

2. Memory Management

```typescript
class MemoryManager {
  private cache = new Map<string, CacheEntry>();
  private maxCacheSize = 100 * 1024 * 1024; // 100MB

  cleanup(): void {
    // LRU cache cleanup
    const entries = Array.from(this.cache.entries())
      .sort((a, b) => a[1].lastAccessed - b[1].lastAccessed);

    let currentSize = this.getCurrentCacheSize();
    while (currentSize > this.maxCacheSize && entries.length > 0) {
      const [key] = entries.shift()!;
      this.cache.delete(key);
      currentSize = this.getCurrentCacheSize();
    }
  }
}
```

3. Rendering Optimization

```
class CanvasOptimizer {
  private dirtyRegions: Rectangle[] = [];

  markDirty(region: Rectangle): void {
    this.dirtyRegions.push(region);
  }

  render(): void {
    if (this.dirtyRegions.length === 0) return;

    // Only render dirty regions
    for (const region of this.dirtyRegions) {
      this.renderRegion(region);
    }

    this.dirtyRegions = [];
  }
}
```

🔐 Security Architecture

Security Measures

1. Input Sanitization

```
class SecurityManager {
  sanitizeInput(input: string): string {
    return input
      .replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi, '')
      .replace(/javascript:/gi, '')
      .replace(/on\w+\s*=/gi, '');
  }

  validateRegex(pattern: string): boolean {
    try {
      new RegExp(pattern);
      return !this.containsDangerousPatterns(pattern);
    } catch {
      return false;
    }
  }
}
```

2. Data Encryption

```
class EncryptionManager {
  async encryptSensitiveData(data: string): Promise<string> {
    const key = await this.getEncryptionKey();
    return await this.encrypt(data, key);
  }

  async decryptSensitiveData(encryptedData: string): Promise<string> {
    const key = await this.getEncryptionKey();
    return await this.decrypt(encryptedData, key);
  }
}
```

📈 Scalability Design

Horizontal Scaling

```
interface ScalabilityManager {
  distributeLoad(tasks: Task[]): Promise<TaskResult[]>;
  balanceWorkers(workers: Worker[]): void;
  optimizeMemoryUsage(): void;
}

class WorkerPool {
  private workers: Worker[] = [];
  private taskQueue: Task[] = [];

  async executeTask(task: Task): Promise<TaskResult> {
    const worker = this.getAvailableWorker();
    return await worker.execute(task);
  }
}
```

🧪 Testing Architecture

Test Structure

```
interface TestSuite {
  unitTests: UnitTest[];
  integrationTests: IntegrationTest[];
  e2eTests: E2ETest[];
}

class PluginTestRunner {
  async runAllTests(): Promise<TestResults> {
    const results = {
      unit: await this.runUnitTests(),
      integration: await this.runIntegrationTests(),
      e2e: await this.runE2ETests()
    };

    return this.aggregateResults(results);
  }
}
```

📋 Deployment Architecture

Build Pipeline

```
Source Code → TypeScript Compilation → Bundle Creation → Minification → Distribution
     ↓               ↓                      ↓               ↓              ↓
Type Check → Linting → Testing → Asset Processing → Package Creation
```

Distribution Strategy

```typescript
interface DeploymentManager {
  buildProduction(): Promise<BuildResult>;
  createDistribution(): Promise<DistributionPackage>;
  validateBuild(): Promise<ValidationResult>;
}
```

🔍 Architecture Decisions

Key Design Decisions

1. **Plugin Separation:** Maintain independent plugins for focused functionality
2. **Shared Utilities:** Common utilities in integration layer for code reuse
3. **Event-Driven Communication:** Loose coupling between components
4. **Modular Architecture:** Easy to extend and maintain
5. **Performance First:** Optimized for large documents and complex canvases

Trade-offs

Decision	Pros	Cons
Separate Plugins	Independent development, focused features	Potential code duplication
Event Bus	Loose coupling, extensible	Debugging complexity
Canvas Rendering	High performance, flexible	Memory intensive
Pattern Caching	Fast redaction, responsive UI	Memory usage

This architecture supports the current feature set while providing a foundation for future enhancements and scalability.