USSD Flow Editor - Architecture Documentation

Table of Contents

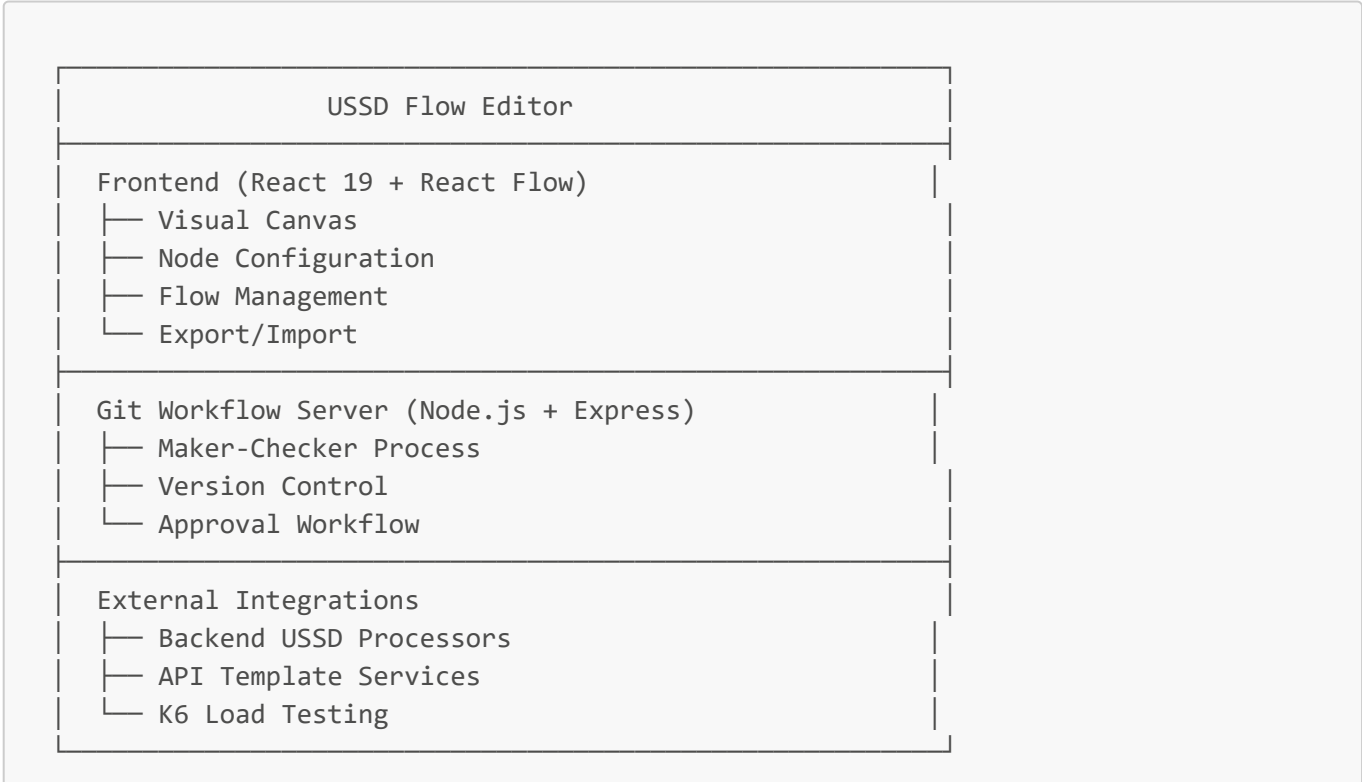
System Overview

The USSD Flow Editor is a sophisticated single-page application (SPA) built on modern React ecosystem, providing a visual flow design environment with real-time editing, export capabilities, and integration with backend systems.

Architecture Principles

- **Component-Based**: Modular React components with single responsibility
- **State-Driven**: Centralized state management with React hooks
- **Event-Driven**: React Flow events for canvas interactions
- **Utility-First**: Shared utilities for common operations
- **Integration-Ready**: Designed for backend system integration

System Boundaries



Technology Stack

Frontend Core

- **React 19**: Latest React with concurrent features and enhanced hooks
- **React Flow 12**: Advanced graph visualization and interaction library
- **Vite 7.1.2**: Next-generation build tool with fast HMR and optimized bundling
- **ESLint**: Code quality and consistency enforcement

Styling & UI

- **CSS Modules**: Scoped styling for component isolation
- **React Flow CSS**: Built-in canvas and node styling
- **Responsive Design**: Mobile-friendly interface design
- **Custom Icons**: Unicode and emoji-based iconography

Backend Services

- **Node.js 18+**: Server runtime for git workflow
- **Express.js**: Web framework for API endpoints
- **Git Integration**: Version control for maker-checker workflow
- **Process Management**: Child process execution for git operations

Development Tools

- **Vite Dev Server**: Development environment with HMR
- **ESLint Config**: Modern JavaScript/React linting rules

Testing & Validation

- **K6**: Load testing script generation
- **JOLT**: JSON transformation and validation
- **Apache Calcite SQL**: Conditional logic evaluation
- **Custom Validators**: Flow integrity checking

Component Architecture

Core Application Structure

```
src/
├── App.jsx                 # Root component with React Flow provider
├── components/             # Reusable UI components
│   ├── FlowControls/      # Canvas control components
│   ├── NodeTypes/         # Custom node implementations
│   ├── TemplateForms/     # Template configuration forms
│   └── Shared/            # Common UI components
├── utils/                 # Utility functions and helpers
```

```
├── styles/                # Global and component styles
└── assets/                # Static resources
```

Component Hierarchy

```
App.jsx
├── ReactFlowProvider
│   ├── ReactFlow
│   │   ├── Background
│   │   ├── Controls
│   │   ├── MiniMap
│   │   └── CustomNodes
│   │       ├── StartNode
│   │       ├── MenuNode
│   │       ├── DynamicMenuNode
│   │       ├── InputNode
│   │       ├── ActionNode
│   │       └── EndNode
│   ├── NodePalette
│   ├── FlowControls
│   │   ├── ExportControls
│   │   ├── ImportControls
│   │   ├── ValidationPanel
│   │   └── LayoutControls
│   ├── NodeConfigPanel
│   │   ├── StartConfig
│   │   ├── MenuConfig
│   │   ├── DynamicMenuConfig
│   │   ├── InputConfig
│   │   ├── ActionConfig
│   │   └── EndConfig
│   └── TemplateCreator
│       ├── TemplateForm
│       ├── JoltEditor
│       ├── TestRunner
│       └── AIGenerator
```

Custom Node Components

Node Base Structure

All custom nodes follow a consistent pattern:

```jsx
// NodeTypes/MenuNode.jsx
import { memo } from 'react';
import { Handle, Position } from 'reactflow';

const MenuNode = ({ data, selected }) => {
  const { config, label } = data;
```

```
  return (
    <div className={`menu-node ${selected ? 'selected' : ''}`}>
      {/* Input Handle */}
      <Handle
        type="target"
        position={Position.Left}
        id="input"
        className="node-handle"
      />

      {/* Node Content */}
      <div className="node-header">
        <span className="node-icon">📋</span>
        <span className="node-title">{label}</span>
      </div>

      <div className="node-content">
        {/* Display menu options */}
        <div className="menu-options">
          {config.prompts?.en?.split('\\n').map((option, idx) => (
            <div key={idx} className="menu-option">{option}</div>
          ))}
        </div>

        {/* Composite Code Display */}
        {config.compositCode && (
          <div className="composite-code">
            <strong>{config.compositCode}</strong>
          </div>
        )}
      </div>

      {/* Dynamic Output Handles */}
      {Object.keys(config.transitions || {}).map((key, index) => (
        <Handle
          key={key}
          type="source"
          position={Position.Right}
          id={`option-${key}`}
          className="node-handle"
          style={{ top: 60 + index * 25 }}
        />
      ))}
    </div>
  );
};

export default memo(MenuNode);
```

Node Configuration System

Each node type has dedicated configuration logic:

```
// components/NodeConfigPanel.jsx
const NodeConfigPanel = ({ selectedNode, onNodeConfigChange }) => {
  const [config, setConfig] = useState(selectedNode?.data?.config || {});

  const renderConfigForm = () => {
    switch (selectedNode?.data?.type) {
      case 'START':
        return <StartConfig config={config} onChange={setConfig} />;
      case 'MENU':
        return <MenuConfig config={config} onChange={setConfig} />;
      case 'DYNAMIC-MENU':
        return <DynamicMenuConfig config={config} onChange={setConfig} />;
      case 'INPUT':
        return <InputConfig config={config} onChange={setConfig} />;
      case 'ACTION':
        return <ActionConfig config={config} onChange={setConfig} />;
      case 'END':
        return <EndConfig config={config} onChange={setConfig} />;
      default:
        return <div>Select a node to configure</div>;
    }
  };

  // ... configuration persistence logic
};
```

Data Flow

State Architecture

The application uses a hybrid state management approach:

1. **React Flow State**: Canvas state (nodes, edges, viewport)
2. **Local Component State**: Configuration panels, forms
3. **Derived State**: Export formats, validation results
4. **Session State**: Templates, git workflow status

Data Flow Patterns

Node Creation Flow

```
User Drag from Palette
    ↓
onDragEnd → createNode()
    ↓
Generate unique ID
    ↓
```

```
Create node object with defaults
    ↓
Add to React Flow nodes array
    ↓
Canvas re-renders with new node
```

Configuration Update Flow

```
User modifies config panel
    ↓
onChange → setConfig()
    ↓
Save to localStorage
    ↓
Update node data
    ↓
setNodes() with updated data
    ↓
Node re-renders with new content
```

Export Generation Flow

```
User clicks Export
    ↓
Collect current nodes & edges
    ↓
Transform to export format
    ↓
Apply business logic transformations
    ↓
Generate JSON output
    ↓
Copy to clipboard or download
```

State Synchronization

Multiple state synchronization mechanisms ensure consistency:

```jsx
// State synchronization in App.jsx
const [nodes, setNodes, onNodesChange] = useNodesState(initialNodes);
const [edges, setEdges, onEdgesChange] = useEdgesState(initialEdges);
const [selectedNode, setSelectedNode] = useState(null);

// Synchronize selected node with canvas selection
const onSelectionChange = useCallback(({ nodes }) => {
  setSelectedNode(nodes[0] || null);
```

```
  }, []);

  // Update node configuration
  const onNodeConfigChange = useCallback((nodeId, newConfig) => {
    setNodes(nodes =>
      nodes.map(node =>
        node.id === nodeId
          ? { ...node, data: { ...node.data, config: newConfig }}
          : node
      )
    );
  }, [setNodes]);
```

State Management

React Flow State Management

React Flow provides built-in state management for:

- **Nodes**: Position, dimensions, data, selection state
- **Edges**: Connections, styling, animation
- **Viewport**: Zoom level, pan position, boundaries

Configuration State Patterns

Persistent Configuration

Node configurations are automatically persisted:

```
// Auto-save configuration changes
useEffect(() => {
  if (selectedNode && config) {
    // Save to localStorage
    const configKey = `node_config_${selectedNode.id}`;
    localStorage.setItem(configKey, JSON.stringify(config));

    // Update node data
    onNodeConfigChange(selectedNode.id, config);
  }
}, [config, selectedNode, onNodeConfigChange]);
```

Template State Management

Template system maintains separate state:

```
// TemplateCreator state management
const [templates, setTemplates] = useState([]);
const [selectedTemplate, setSelectedTemplate] = useState(null);
```

```
const [isEditing, setIsEditing] = useState(false);

// Load templates from various sources
useEffect(() => {
  const loadTemplates = async () => {
    const stored = JSON.parse(localStorage.getItem('api_templates') || '[]');
    const imported = await loadImportedTemplates();
    setTemplates([...stored, ...imported]);
  };
  loadTemplates();
}, []);
```

Validation State

Real-time validation maintains error state:

```
// Flow validation state
const [validationErrors, setValidationErrors] = useState([]);
const [validationWarnings, setValidationWarnings] = useState([]);

// Validate flow on changes
useEffect(() => {
  const errors = validateFlow(nodes, edges);
  setValidationErrors(errors.filter(e => e.severity === 'error'));
  setValidationWarnings(errors.filter(e => e.severity === 'warning'));
}, [nodes, edges]);
```

Utility Systems

Core Utilities

Flow Utilities (`utils/flowUtils.js`)

Central utility for flow operations:

```
// Export format transformation
export const exportToFlowFormat = (nodes, edges) => {
  return nodes.map(node => {
    const exportNode = {
      id: node.id,
      type: node.data.type,
      transitions: {},
      ...extractNodeSpecificData(node)
    };

    // Add edge-based transitions
    const nodeEdges = edges.filter(edge => edge.source === node.id);
    nodeEdges.forEach(edge => {
      const sourceHandle = edge.sourceHandle || '*';
```

```
      exportNode.transitions[sourceHandle] = edge.target;
    });

    return exportNode;
  });
};

// Composite code handling
export const extractCompositeCode = (node) => {
  if (node.data.type === 'MENU' || node.data.type === 'END') {
    return node.data.config?.compositCode || null;
  }
  return null;
};
```

JOLT Generator (`utils/JoltGeneratorEnhanced.js`)

Sophisticated transformation generation:

```
// Generate JOLT specifications from templates
export const generateJoltSpec = (template, sessionVariables) => {
  const spec = [];

  // Shift operation for field mapping
  if (template.fieldMappings) {
    spec.push({
      operation: "shift",
      spec: generateShiftSpec(template.fieldMappings, sessionVariables)
    });
  }

  // Default operation for static values
  if (template.defaultValues) {
    spec.push({
      operation: "default",
      spec: template.defaultValues
    });
  }

  return spec;
};
```

Template Manager (`utils/TemplateManager.js`)

Template lifecycle management:

```
// Template CRUD operations
export class TemplateManager {
  static save(template) {
```

```
    const templates = this.getAll();
    const index = templates.findIndex(t => t._id === template._id);

    if (index >= 0) {
      templates[index] = template;
    } else {
      template._id = generateId();
      templates.push(template);
    }

    localStorage.setItem('api_templates', JSON.stringify(templates));
    return template;
  }

  static getAll() {
    return JSON.parse(localStorage.getItem('api_templates') || '[]');
  }

  static validate(template) {
    const errors = [];
    if (!template.name) errors.push('Template name required');
    if (!template.requestTemplate) errors.push('Request template required');
    return errors;
  }
}
```

Validation Systems

Flow Validation

Comprehensive flow integrity checking:

```
// Flow validation rules
export const validateFlow = (nodes, edges) => {
  const errors = [];
  const warnings = [];

  // Rule: Must have exactly one START node
  const startNodes = nodes.filter(n => n.data.type === 'START');
  if (startNodes.length === 0) {
    errors.push({ type: 'missing_start', message: 'Flow must have a START node'
});
  } else if (startNodes.length > 1) {
    warnings.push({ type: 'multiple_start', message: 'Multiple START nodes found'
});
  }

  // Rule: All nodes must be connected
  const connectedNodes = new Set();
  edges.forEach(edge => {
    connectedNodes.add(edge.source);
```

```
    connectedNodes.add(edge.target);
  });

  nodes.forEach(node => {
    if (!connectedNodes.has(node.id) && node.data.type !== 'START') {
      warnings.push({
        type: 'orphaned_node',
        message: `Node ${node.data.label} is not connected`,
        nodeId: node.id
      });
    }
  });

  return [...errors, ...warnings];
};
```

Template Validation

API template validation:

```
// Template validation system
export const validateTemplate = (template) => {
  const validation = {
    valid: true,
    errors: [],
    warnings: []
  };

  // Validate request structure
  if (!template.requestTemplate?.url) {
    validation.errors.push('Request URL is required');
    validation.valid = false;
  }

  // Validate JOLT specification
  if (template.joltSpec) {
    try {
      JSON.stringify(template.joltSpec);
    } catch (e) {
      validation.errors.push('Invalid JOLT specification format');
      validation.valid = false;
    }
  }

  return validation;
};
```

Integration Points

Backend Integration

USSD Processing Integration

Flow export format designed for backend processors:

```
// Backend integration format
{
  "flowMetadata": {
    "id": "ussd_flow_v1.2",
    "version": "1.2.0",
    "created": "2024-01-15T10:00:00Z",
    "language": "multi"
  },
  "nodes": [
    {
      "id": "start_123",
      "type": "START",
      "ussdCode": "*123#",
      "transitions": { "*123#": "menu_main_456" },
      "prompts": { "en": "Welcome", "es": "Bienvenido" }
    },
    {
      "id": "menu_main_456",
      "type": "MENU",
      "compositCode": "7634",
      "transitions": { "1": "action_balance", "2": "input_amount" },
      "prompts": { "en": "1. Balance\\n2. Transfer" }
    }
  ]
}
```

API Template Integration

Templates integrate with external API systems:

```
// Template execution context
{
  "templateId": "SEND_MONEY_API",
  "sessionContext": {
    "userId": "user123",
    "sessionToken": "abc...xyz",
    "variables": {
      "AMOUNT": "100",
      "PHONE": "1234567890"
    }
  },
  "requestTemplate": {
    "method": "POST",
    "url": "https://api.bank.com/transfer",
    "headers": { "Authorization": "Bearer {{sessionToken}}" },
    "body": { "amount": "{{AMOUNT}}", "phone": "{{PHONE}}" }
```

```
  },
  "joltSpec": [ /* transformation rules */ ]
}
```

Git Workflow Server

Maker-Checker Process

Node.js server manages approval workflow:

```
// git-workflow-server.js
const express = require('express');
const { execSync } = require('child_process');

app.post('/api/submit-flow', (req, res) => {
  const { flowData, submitter } = req.body;

  try {
    // Create feature branch
    const branchName = `flow-${Date.now()}-${submitter}`;
    execSync(`git checkout -b ${branchName}`);

    // Save flow data
    const fileName = `flows/${flowData.id}.json`;
    fs.writeFileSync(fileName, JSON.stringify(flowData, null, 2));

    // Commit changes
    execSync(`git add ${fileName}`);
    execSync(`git commit -m "Submit flow: ${flowData.metadata.name}"`);

    // Push for review
    execSync(`git push origin ${branchName}`);

    res.json({
      success: true,
      branchName,
      reviewUrl: `http://git-server/review/${branchName}`
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

K6 Testing Integration

Load Test Generation

Generate K6 scripts from flow configuration:

```
// K6 test generation
export const generateK6Test = (flowData, testConfig) => {
  const testScript = `
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  scenarios: {
    ussd_flow_test: {
      executor: 'ramping-vus',
      startVUs: 0,
      stages: ${JSON.stringify(testConfig.stages)},
    },
  },
};

const BASE_URL = '${testConfig.baseUrl}';

export default function () {
  const sessionId = generateSessionId();
  const phoneNumber = generatePhoneNumber();

  ${generateTestSteps(flowData)}
}

${generateHelperFunctions()}
  `;

  return testScript;
};
```

Performance Considerations

Frontend Optimization

React Flow Performance

- **Memoization**: All custom nodes are memoized with `React.memo()`
- **Selective Updates**: Only update changed nodes/edges
- **Viewport Optimization**: Virtualization for large flows
- **Handle Optimization**: Dynamic handle creation only when needed

Bundle Optimization

```
// vite.config.js optimization
export default defineConfig({
  build: {
    rollupOptions: {
      output: {
```

```
          manualChunks: {
            vendor: ['react', 'react-dom'],
            reactflow: ['reactflow'],
            utils: ['./src/utils/flowUtils.js',
'./src/utils/JoltGeneratorEnhanced.js']
          }
        }
      },
      chunkSizeWarningLimit: 1000
    },
    optimizeDeps: {
      include: ['reactflow', 'react', 'react-dom']
    }
  });
```

Memory Management

- **Cleanup Effects**: Proper useEffect cleanup
- **Event Listener Management**: Remove listeners on unmount
- **State Optimization**: Minimal state updates

Scalability Patterns

Large Flow Handling

```
// Performance optimizations for large flows
const FlowCanvas = () => {
  // Debounced save to prevent excessive writes
  const debouncedSave = useMemo(
    () => debounce((nodes, edges) => {
      saveFlowToStorage(nodes, edges);
    }, 1000),
    []
  );

  // Virtualized node rendering for 100+ nodes
  const visibleNodes = useMemo(() => {
    return nodes.filter(node => isNodeInViewport(node, viewport));
  }, [nodes, viewport]);

  return (
    <ReactFlow
      nodes={visibleNodes}
      edges={edges}
      nodesDraggable={nodes.length < 50} // Disable for large flows
      maxZoom={nodes.length > 100 ? 2 : 4} // Limit zoom for performance
    />
  );
};
```

Data Loading Strategies

- **Lazy Loading**: Load templates and configurations on demand
- **Caching**: Aggressive caching of computed values
- **Background Processing**: Non-blocking validation and export

This architecture documentation provides a comprehensive technical overview of the USSD Flow Editor system, enabling developers to understand the design decisions, integration patterns, and performance considerations that drive the application.