Microservices Architecture Conversion Plan

Project: NodeAngularFullStack
Date: 2025-10-23
Author: System Architecture Analysis
Status: Proposal - Awaiting Decision

Table of Contents

Executive Summary

This document proposes converting the current monolithic NodeAngularFullStack application into a microservices architecture with:

- **3 Core Tool Services**: Form Builder, SVG Drawing, Short Link
- **1 Core Platform Service**: Dashboard, Authentication, User Management
- **4 Frontend Applications**: Main Dashboard + 3 Tool-Specific Angular Apps
- **4 Dedicated Databases**: PostgreSQL per service for data isolation
- **Shared Component Library**: Reusable Angular components across frontend services

Key Benefits:
- Independent deployment and scaling of each tool
- Team autonomy per service
- Technology flexibility per service
- Fault isolation (one service failure doesn't crash entire system)

Key Challenges:
- Increased operational complexity
- Distributed data management
- Network latency between services
- More complex local development setup

Current Architecture Analysis

Current Monolithic Structure

```mermaid
graph TB
    subgraph "Current Monolith"
        Client[Angular Frontend
Port 4200]
        API[Express Backend
Port 3000]
        DB[(PostgreSQL
Single Database)]

        Client -->|HTTP/REST| API
        API -->|SQL Queries| DB
    end

    subgraph "Features in Monolith"
        Forms[Form Builder]
        SVG[SVG Drawing]
        Links[Short Links]
        Auth[Authentication]
        Admin[Admin Panel]
    end

    API -.-> Forms
    API -.-> SVG
    API -.-> Links
    API -.-> Auth
    API -.-> Admin
```

Current Database Schema

Single PostgreSQL Database with tables:

- `users`, `tenants`, `roles` (Authentication)
- `forms`, `form_schemas`, `form_submissions`, `form_themes` (Form Builder)
- `short_links` (Short Link Service)
- SVG Drawing data (if stored)

Current Frontend Structure

Single Angular Application at apps/web/:

- `features/tools/components/form-builder/` - Form Builder UI
- `features/tools/components/svg-drawing/` - SVG Drawing UI
- `features/admin/` - Admin Dashboard
- `features/auth/` - Authentication UI
- Shared components in `shared/`

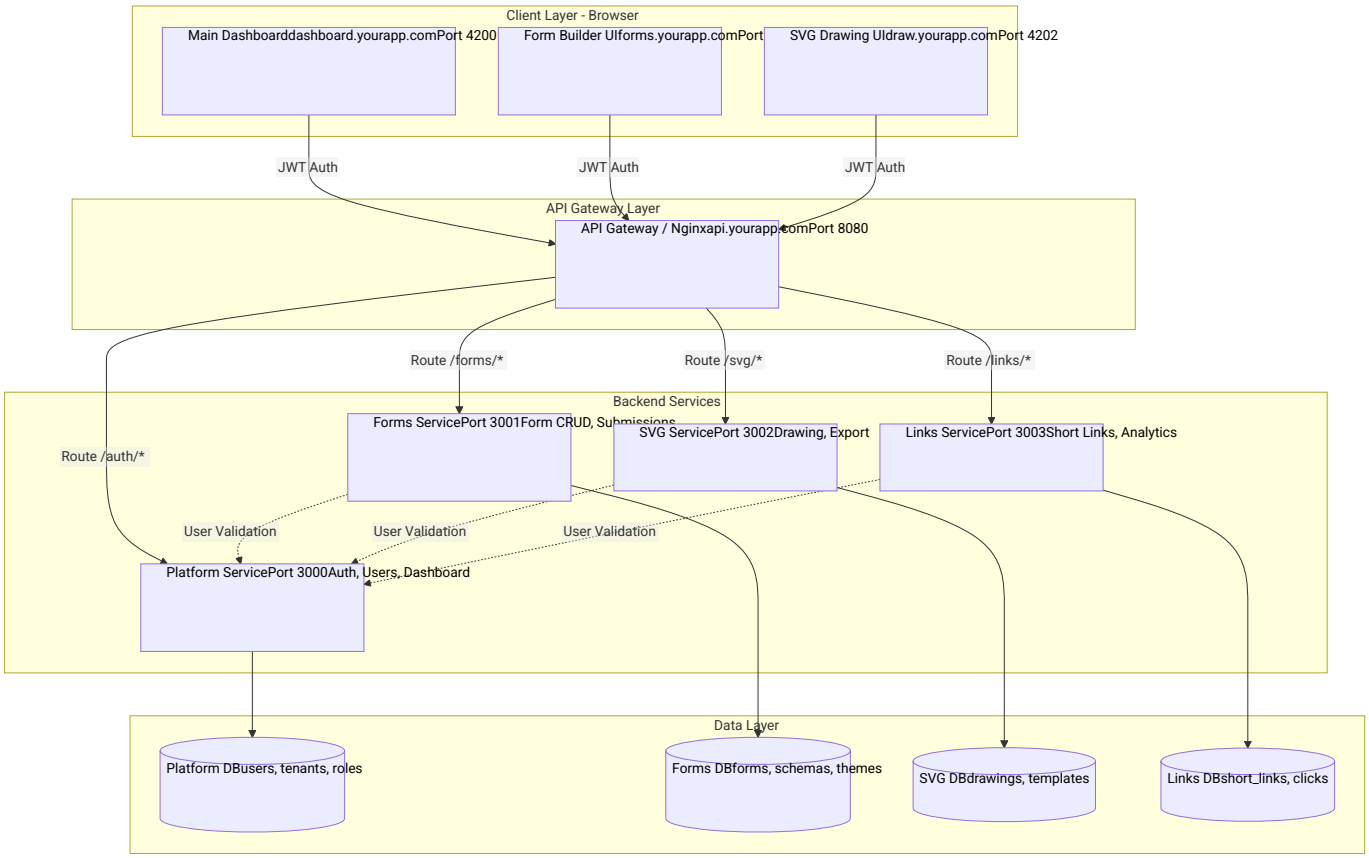
Current Pain Points Suitable for Microservices

1. **Form Builder Complexity**: 50+ components, heavy feature set, independent lifecycle
2. **SVG Drawing Independence**: Completely isolated feature with no dependencies on forms
3. **Short Link Service**: Stateless service, perfect candidate for separation
4. **Deployment Coupling**: Cannot deploy form builder updates without full system deployment
5. **Scaling Challenges**: Cannot scale form submission processing independently

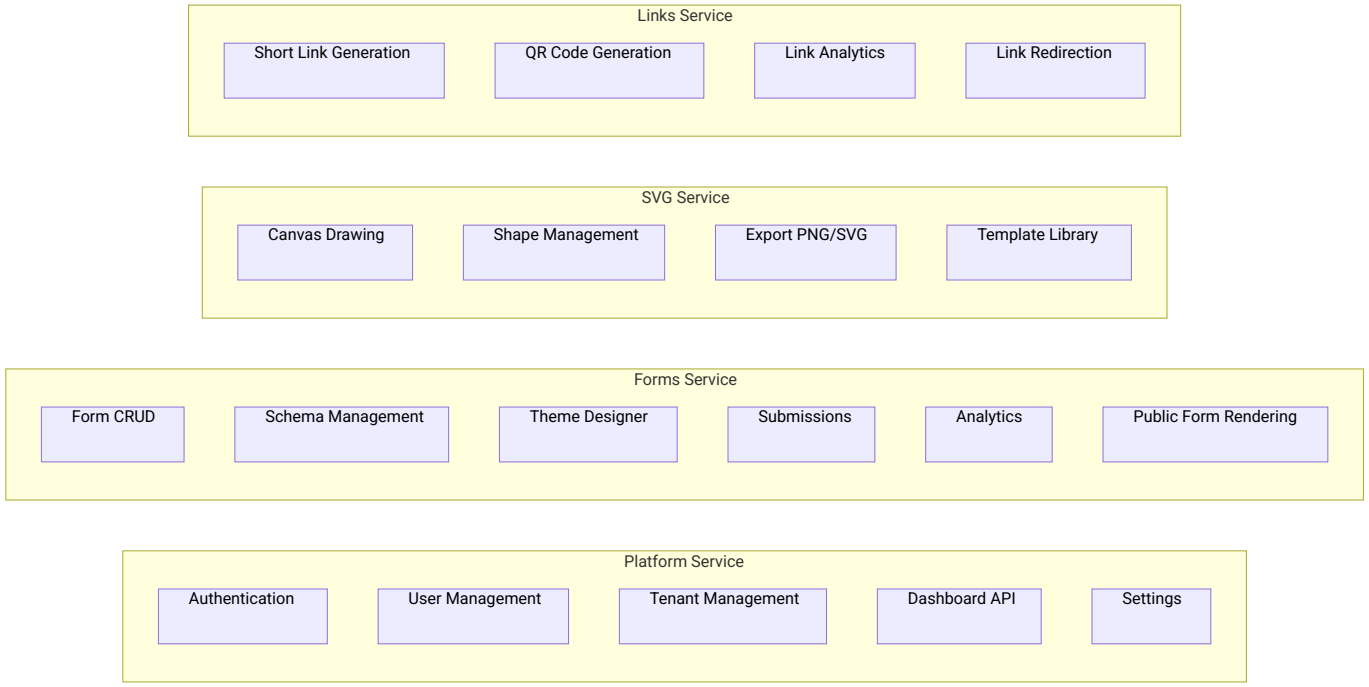
---

Proposed Microservices Architecture

High-Level System Architecture



Service Boundaries



Service Decomposition Strategy

1. Platform Service (Core)

Responsibility: Authentication, user management, dashboard orchestration

Port: 3000 (local) | api.yourapp.com (production)

Database: platform_db

- Tables: `users`, `tenants`, `roles`, `sessions`, `user_settings`

API Endpoints:

POST /api/auth/register
POST /api/auth/login
POST /api/auth/logout
GET /api/auth/profile
PUT /api/auth/profile
GET /api/users
POST /api/users
GET /api/dashboard/stats

Technology Stack:

- Backend: Express.js + TypeScript

- Database: PostgreSQL
- Auth: JWT + Passport.js
- Caching: Redis (sessions)

---

2. Forms Service

Responsibility: Complete form builder lifecycle, submissions, analytics

Port: 3001 (local) | forms-api.yourapp.com (production)

Database: forms_db

- Tables: `forms`, `form_schemas`, `form_submissions`, `form_themes`, `short_links` (only form-related)

API Endpoints:

```
GET    /api/forms
POST   /api/forms
GET    /api/forms/:id
PUT    /api/forms/:id
DELETE /api/forms/:id
POST   /api/forms/:id/publish
GET    /api/forms/:id/submissions
POST   /api/forms/:id/analytics
GET    /api/public/forms/:shortCode
POST   /api/public/forms/:shortCode/submit
GET    /api/themes
POST   /api/themes
```

Technology Stack:

- Backend: Express.js + TypeScript
- Database: PostgreSQL
- Validation: express-validator
- Sanitization: DOMPurify
- Storage: DO Spaces (file uploads)

Inter-Service Communication:

- Calls Platform Service to validate JWT tokens
- Calls Links Service to generate short links

---

3. SVG Drawing Service

Responsibility: Canvas-based drawing, shape management, export

Port: 3002 (local) | svg-api.yourapp.com (production)

Database: svg_db

- Tables: `drawings`, `shapes`, `templates`, `exports`

API Endpoints:

```
GET    /api/drawings
POST   /api/drawings
GET    /api/drawings/:id
PUT    /api/drawings/:id
DELETE /api/drawings/:id
POST   /api/drawings/:id/export
GET    /api/templates
POST   /api/templates
```

Technology Stack:

- Backend: Express.js + TypeScript
- Database: PostgreSQL
- Image Processing: Sharp.js (for PNG export)
- Storage: DO Spaces (exported images)

Inter-Service Communication:

- Calls Platform Service to validate JWT tokens

---

4. Short Links Service

Responsibility: Short link generation, QR codes, analytics, redirection

Port: 3003 (local) | links-api.yourapp.com (production)

Database: links_db

- Tables: `short_links`, `link_clicks`, `qr_codes`

API Endpoints:

```
POST   /api/links/generate
GET    /api/links/:shortCode
GET    /api/links/:shortCode/analytics
GET    /api/links/:shortCode/qr
GET    /:shortCode (public redirect)
```

Technology Stack:

- Backend: Express.js + TypeScript
- Database: PostgreSQL
- QR Generation: qrcode library
- Caching: Redis (hot links)

Inter-Service Communication:

- Calls Platform Service to validate JWT tokens
- Standalone for public redirects (no auth required)

Frontend Microservices Strategy

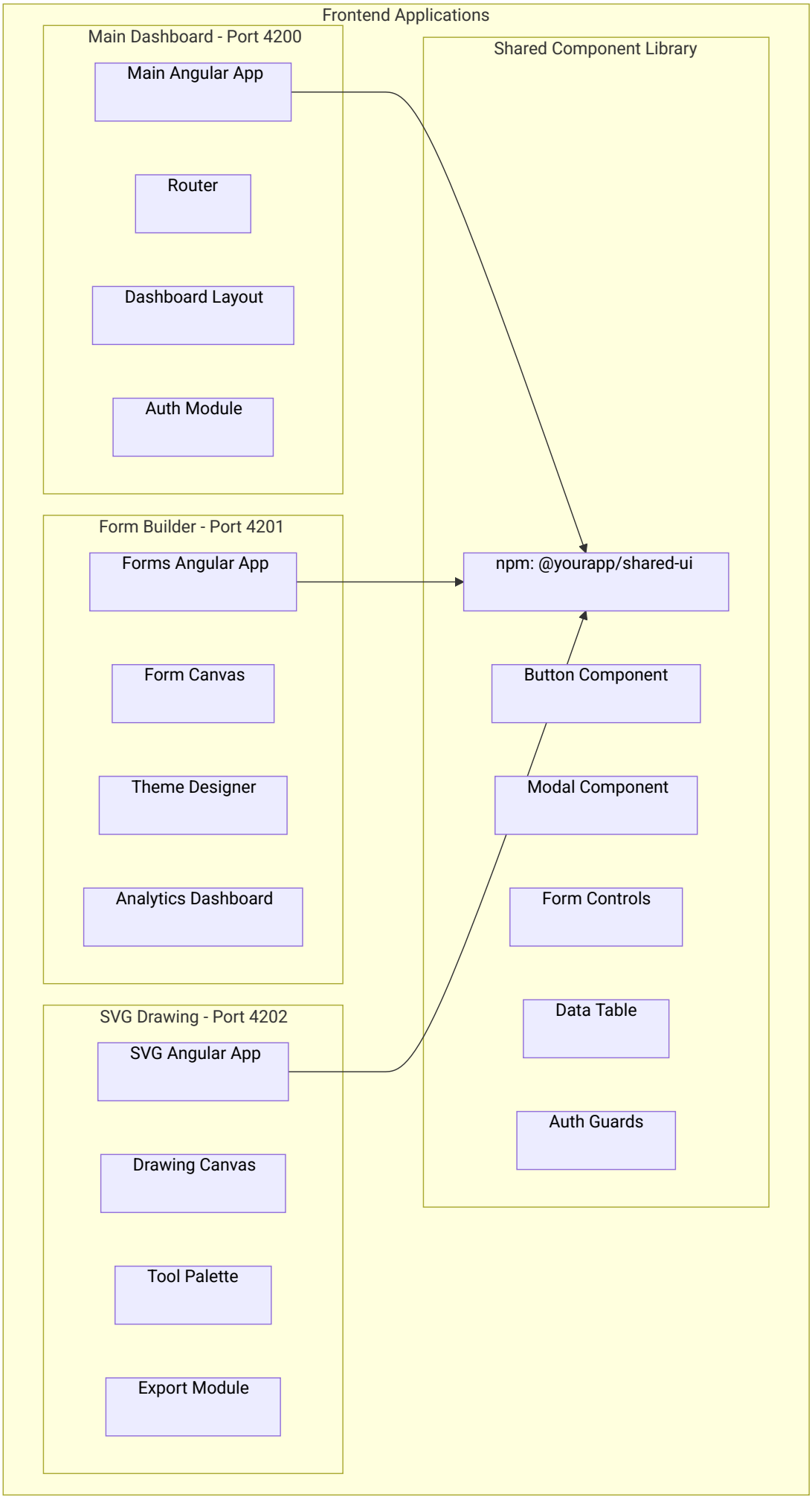
Angular Module Federation vs Separate Projects

Decision: Separate Angular Projects with Shared Library

Rationale:

- Complete independence per tool
- Different deployment cycles
- Different teams can own each tool
- Easier to scale development teams
- Clearer service boundaries

Frontend Architecture


Frontend Applications
Main Dashboard - Port 4200
Main Angular App
Router
Dashboard Layout
Auth Module
Form Builder - Port 4201
Forms Angular App
Form Canvas
Theme Designer
Analytics Dashboard
SVG Drawing - Port 4202
SVG Angular App
Drawing Canvas
Tool Palette
Export Module
Shared Component Library
npm: @yourapp/shared-ui
Button Component
Modal Component
Form Controls
Data Table
Auth Guards

Project Structure

```
/
├── apps/
│   ├── dashboard/              # Main dashboard (Port 4200)
│   │   ├── src/
│   │   │   ├── app/
│   │   │   │   ├── core/      # Core services, guards
│   │   │   │   ├── features/  # Dashboard features
│   │   │   │   │   ├── home/
│   │   │   │   │   ├── settings/
│   │   │   │   │   └── user-profile/
│   │   │   │   └── layouts/   # Dashboard layout
│   │   │   └── environments/
│   │   ├── angular.json
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
│   ├── forms/                 # Form Builder (Port 4201)
│   │   ├── src/
│   │   │   ├── app/
│   │   │   │   ├── features/
│   │   │   │   │   ├── form-builder/
│   │   │   │   │   ├── theme-designer/
│   │   │   │   │   ├── submissions/
│   │   │   │   │   └── analytics/
│   │   │   │   └── layouts/   # Forms-specific layout
│   │   │   └── environments/
│   │   ├── angular.json
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
│   ├── svg-drawing/           # SVG Drawing (Port 4202)
│   │   ├── src/
│   │   │   ├── app/
│   │   │   │   ├── features/
│   │   │   │   │   ├── canvas/
│   │   │   │   │   ├── tools/
│   │   │   │   │   ├── templates/
│   │   │   │   │   └── export/
│   │   │   │   └── layouts/   # SVG-specific layout
│   │   │   └── environments/
│   │   ├── angular.json
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
│   └── api/                   # Keep existing or split further
│
├── packages/
│   ├── shared-ui/             # Shared Angular components
│   │   ├── src/
│   │   │   └── lib/
│   │   │       └── components/
```

```
│   │   │   │   │   ├── button/
│   │   │   │   │   ├── modal/
│   │   │   │   │   ├── form-controls/
│   │   │   │   │   ├── data-table/
│   │   │   │   ├── directives/
│   │   │   │   ├── pipes/
│   │   │   │   ├── guards/
│   │   │   │   ├── interceptors/
│   │   │   │   ├── services/
│   │   │   │   │   ├── auth.service.ts
│   │   │   │   │   ├── api-client.service.ts
│   │   │   ├── public-api.ts
│   │   ├── package.json
│   │   ├── tsconfig.json
│   │
│   ├── shared-types/          # Shared TypeScript types
│   │   ├── src/
│   │   │   ├── auth.types.ts
│   │   │   ├── forms.types.ts
│   │   │   ├── svg.types.ts
│   │   │   ├── index.ts
│   │   ├── package.json
│   │   ├── tsconfig.json
│   │
│   ├── shared-config/         # Shared configuration
│   │   ├── src/
│   │   │   ├── api-endpoints.ts
│   │   │   ├── constants.ts
│   │   │   ├── index.ts
│   │   ├── package.json
│
├── infrastructure/
│   ├── docker/
│   │   ├── dashboard.Dockerfile
│   │   ├── forms.Dockerfile
│   │   ├── svg.Dockerfile
│   │   ├── platform-api.Dockerfile
│   │   ├── forms-api.Dockerfile
│   │   ├── svg-api.Dockerfile
│   │   ├── links-api.Dockerfile
│   ├── nginx/
│   │   ├── api-gateway.conf
│   ├── docker-compose.yml
│
├── package.json (root workspace)
```

Shared Components Strategy

1. Shared UI Component Library (@yourapp/shared-ui)

Purpose: Reusable Angular components, services, guards used across all frontends

Components to Extract:

```
// From apps/web/src/app/shared/components/
export { ButtonComponent } from './components/button/button.component';
export { ModalComponent } from './components/modal/modal.component';
export { CardComponent } from './components/card/card.component';
export { ToastComponent } from './components/toast/toast.component';
export { LoadingSpinnerComponent } from './components/loading-spinner/loading-spinner.component';
export { DataTableComponent } from './components/data-table/data-table.component';
export { FormFieldComponent } from './components/form-field/form-field.component';
export { ToolCardComponent } from './components/tool-card/tool-card.component';
```

Services to Extract:

```
// From apps/web/src/app/core/
export { AuthService } from './services/auth.service';
export { ApiClientService } from './services/api-client.service';
export { AuthGuard } from './guards/auth.guard';
export { AuthInterceptor } from './interceptors/auth.interceptor';
```

Installation in Each App:

```
{
  "dependencies": {
    "@yourapp/shared-ui": "workspace:*",
    "@yourapp/shared-types": "workspace:*",
    "@yourapp/shared-config": "workspace:*"
  }
}
```

Usage Example:

```
// In apps/forms/src/app/app.config.ts
import { AuthService, AuthGuard } from '@yourapp/shared-ui';
import { FormSchema } from '@yourapp/shared-types';
import { API_ENDPOINTS } from '@yourapp/shared-config';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    AuthService,
    AuthGuard,
    { provide: 'API_BASE_URL', useValue: API_ENDPOINTS.FORMS_API }
  ]
};
```

2. Shared Type Definitions (@yourapp/shared-types)

Purpose: TypeScript interfaces shared between frontend and backend services

Type Files:

```
// auth.types.ts
export interface User {
  id: string;
  email: string;
  role: 'admin' | 'user' | 'readonly';
  tenantId?: string;
}

export interface AuthResponse {
  user: User;
  accessToken: string;
  refreshToken: string;
}

// forms.types.ts
export interface FormSchema {
  id: string;
  title: string;
  userId: string;
  fields: FormField[];
  settings: FormSettings;
  themeId?: string;
}

// svg.types.ts
export interface Drawing {
  id: string;
  userId: string;
  shapes: Shape[];
  canvasSize: { width: number; height: number };
}
```

3. Shared Configuration (@yourapp/shared-config)

Purpose: Centralized configuration constants

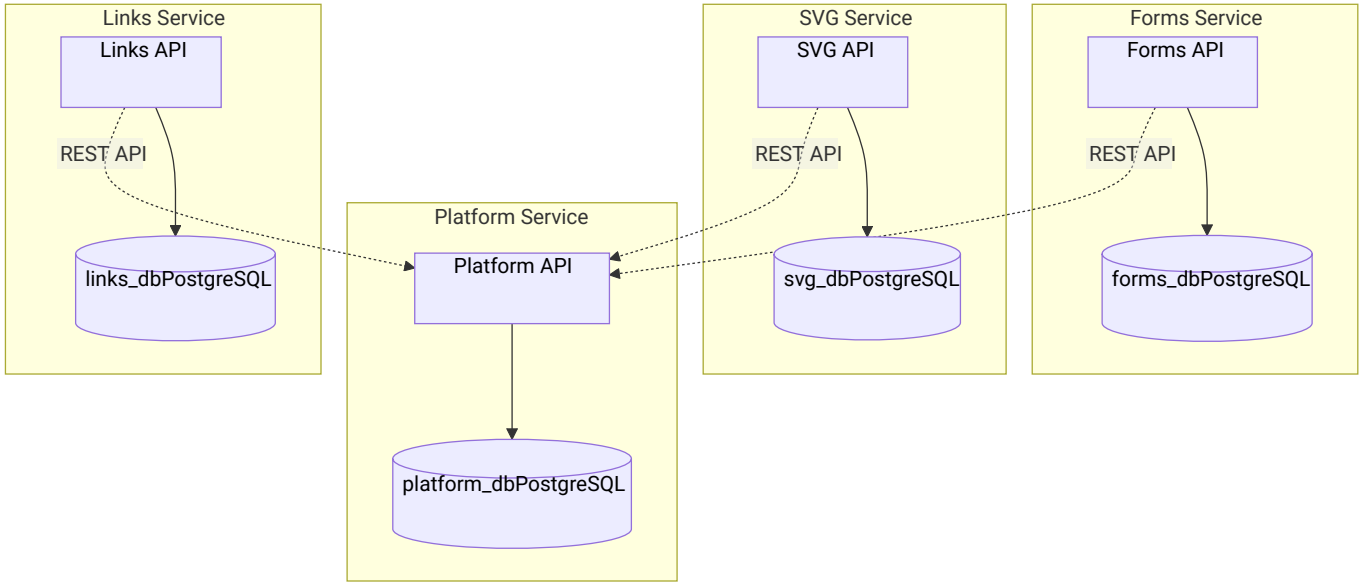
API Endpoints:

```
// api-endpoints.ts
export const API_ENDPOINTS = {
  PLATFORM_API: process.env['PLATFORM_API_URL'] || 'http://localhost:3000',
  FORMS_API: process.env['FORMS_API_URL'] || 'http://localhost:3001',
  SVG_API: process.env['SVG_API_URL'] || 'http://localhost:3002',
  LINKS_API: process.env['LINKS_API_URL'] || 'http://localhost:3003',
};
```

```
export const FRONTEND_URLS = {
  DASHBOARD: process.env['DASHBOARD_URL'] || 'http://localhost:4200',
  FORMS: process.env['FORMS_URL'] || 'http://localhost:4201',
  SVG: process.env['SVG_URL'] || 'http://localhost:4202',
};
```

Database Strategy

Database Per Service Pattern



Database Migration Strategy

Current Single Database:

nodeangularfullstack_db
- users
- tenants
- roles
- forms
- form_schemas
- form_submissions
- form_themes
- short_links
- (svg data if exists)

Target Multi-Database:

1. platform_db:

```
CREATE DATABASE platform_db;
```

-- Tables
- users (id, email, password_hash, role, tenant_id, created_at, updated_at)
- tenants (id, name, plan, settings, created_at)
- roles (id, name, permissions)
- sessions (id, user_id, token, expires_at)
- user_settings (user_id, key, value)

2. forms_db:

```
CREATE DATABASE forms_db;

-- Tables
- forms (id, user_id, title, description, created_at, updated_at)
- form_schemas (id, form_id, schema_json, version, created_at)
- form_submissions (id, form_schema_id, data_json, submitted_at, ip_address)
- form_themes (id, user_id, name, properties_json, created_at)
- form_short_links (id, form_schema_id, short_code, token, expires_at)
```

3. svg_db:

```
CREATE DATABASE svg_db;

-- Tables
- drawings (id, user_id, title, canvas_json, created_at, updated_at)
- shapes (id, drawing_id, type, properties_json, order)
- templates (id, user_id, name, canvas_json, created_at)
- exports (id, drawing_id, format, file_url, created_at)
```

4. links_db:

```
CREATE DATABASE links_db;

-- Tables
- short_links (id, user_id, original_url, short_code, expires_at, created_at)
- link_clicks (id, short_link_id, clicked_at, ip_address, user_agent, referrer)
- qr_codes (id, short_link_id, image_url, format, created_at)
```

Data Consistency Challenges

Challenge 1: User ID References Across Services

Problem: Forms service stores user_id, but user data is in platform_db

Solution: User Context Token Pattern

```
// Platform service includes user context in JWT
interface JWTPayload {
  userId: string;
  email: string;
  role: string;
```

```
  tenantId?: string;
  iat: number;
  exp: number;
}

// Forms service validates token and uses userId directly
// No need to query platform_db for every request
```

Challenge 2: Form Short Links Reference Forms

Problem: Short links in links_db reference forms in forms_db

Solution: Resource ID Pattern

```
// Short link stores resource metadata
interface ShortLink {
  id: string;
  userId: string;
  resourceType: 'form' | 'svg' | 'external';
  resourceId: string; // Form ID or SVG ID
  shortCode: string;
  originalUrl: string;
  expiresAt: Date;
}

// Links service calls Forms service to validate form exists
// No foreign key constraint across databases
```

Database Migration Script

```
// migration/split-databases.ts
import { Pool } from 'pg';

async function splitDatabases() {
  const sourceDb = new Pool({ database: 'nodeangularfullstack_db' });

  // 1. Create new databases
  await createDatabase('platform_db');
  await createDatabase('forms_db');
  await createDatabase('svg_db');
  await createDatabase('links_db');

  // 2. Migrate users table to platform_db
  const platformDb = new Pool({ database: 'platform_db' });
  await platformDb.query(`
    CREATE TABLE users AS
    SELECT * FROM dblink('dbname=nodeangularfullstack_db',
                         'SELECT * FROM users')
    AS users(id uuid, email text, ...);
  `);
```

```
  // 3. Migrate forms tables to forms_db
  const formsDb = new Pool({ database: 'forms_db' });
  await formsDb.query(`
    CREATE TABLE forms AS
    SELECT * FROM dblink('dbname=nodeangularfullstack_db',
                         'SELECT * FROM forms')
    AS forms(id uuid, user_id uuid, ...);
  `);

  // 4. Repeat for other tables...

  // 5. Validate data integrity
  await validateMigration();
}
```

Communication Patterns

1. Synchronous Communication (REST)

Use Cases:

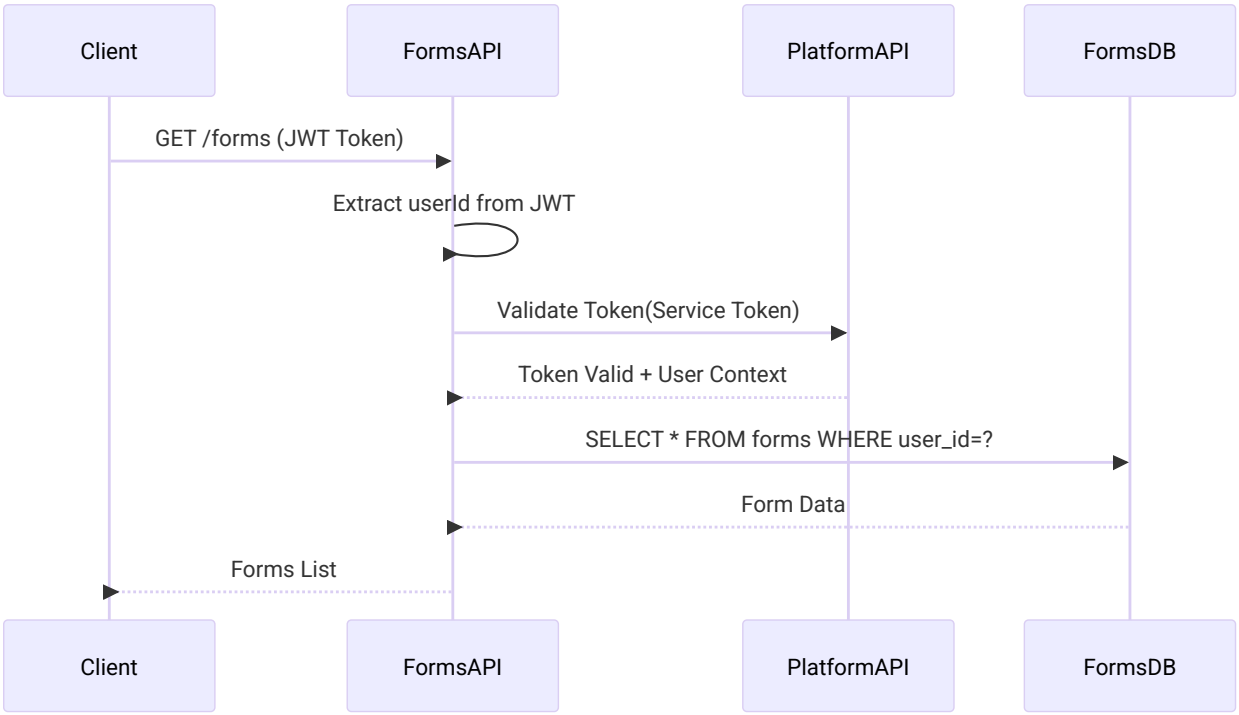
- User authentication validation
- Fetching user details
- Form existence checks

Example: Forms Service → Platform Service

```
// apps/forms-api/src/middleware/auth.middleware.ts
async function validateUser(userId: string): Promise<User> {
  const response = await axios.get(
    `${PLATFORM_API_URL}/api/users/${userId}`,
    {
      headers: { 'X-Service-Token': SERVICE_AUTH_TOKEN }
    }
  );
  return response.data;
}
```

2. Service-to-Service Authentication

Pattern: Service Token + User Context



Implementation:

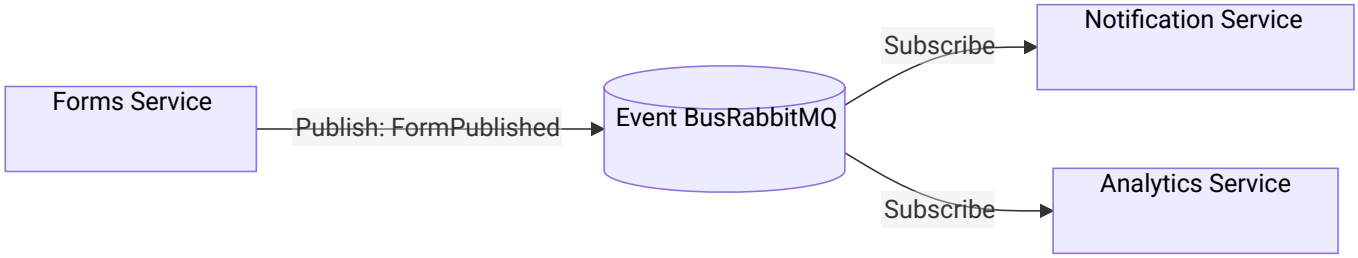
```
// Shared middleware for all services
// packages/shared-backend/auth-middleware.ts

import axios from 'axios';

export async function validateServiceRequest(req: Request, res: Response, next: NextFunction) {
  const userToken = req.headers.authorization?.replace('Bearer ', '');

  if (!userToken) {
    return res.status(401).json({ error: 'Unauthorized' });
  }

  try {
    // Call Platform Service to validate token
    const response = await axios.post(
      `${PLATFORM_API_URL}/api/auth/validate`,
      { token: userToken },
      {
        headers: {
          'X-Service-Token': process.env['SERVICE_AUTH_TOKEN']
        }
      }
    );

    // Attach user context to request
    req.user = response.data.user;
    next();
  } catch (error) {
    return res.status(401).json({ error: 'Invalid token' });
  }
}
```

3. Event-Driven Communication (Future Enhancement)

Use Cases:

- Form published → Send notification
- User registered → Create default theme
- Form submitted → Update analytics

Technology: RabbitMQ or Redis Pub/Sub



Local Development Setup

Docker Compose Configuration

File: infrastructure/docker-compose.yml

```
version: '3.9'

services:
  # ==================== Databases ====================
  platform-db:
    image: postgres:15
    container_name: platform-db
    environment:
      POSTGRES_DB: platform_db
      POSTGRES_USER: dbuser
      POSTGRES_PASSWORD: dbpassword
    ports:
      - "5432:5432"
    volumes:
      - platform-db-data:/var/lib/postgresql/data

  forms-db:
    image: postgres:15
    container_name: forms-db
    environment:
      POSTGRES_DB: forms_db
```

```
    POSTGRES_USER: dbuser
    POSTGRES_PASSWORD: dbpassword
  ports:
    - "5433:5432"
  volumes:
    - forms-db-data:/var/lib/postgresql/data

svg-db:
  image: postgres:15
  container_name: svg-db
  environment:
    POSTGRES_DB: svg_db
    POSTGRES_USER: dbuser
    POSTGRES_PASSWORD: dbpassword
  ports:
    - "5434:5432"
  volumes:
    - svg-db-data:/var/lib/postgresql/data

links-db:
  image: postgres:15
  container_name: links-db
  environment:
    POSTGRES_DB: links_db
    POSTGRES_USER: dbuser
    POSTGRES_PASSWORD: dbpassword
  ports:
    - "5435:5432"
  volumes:
    - links-db-data:/var/lib/postgresql/data

# ==================== Backend Services ====================
platform-api:
  build:
    context: .
    dockerfile: infrastructure/docker/platform-api.Dockerfile
  container_name: platform-api
  ports:
    - "3000:3000"
  environment:
    NODE_ENV: development
    DATABASE_URL: postgresql://dbuser:dbpassword@platform-db:5432/platform_db
    JWT_SECRET: your-secret-key
    SERVICE_AUTH_TOKEN: shared-service-secret
  depends_on:
    - platform-db

forms-api:
  build:
    context: .
    dockerfile: infrastructure/docker/forms-api.Dockerfile
  container_name: forms-api
  ports:
    - "3001:3001"
```

```
    environment:
      NODE_ENV: development
      DATABASE_URL: postgresql://dbuser:dbpassword@forms-db:5432/forms_db
      PLATFORM_API_URL: http://platform-api:3000
      SERVICE_AUTH_TOKEN: shared-service-secret
    depends_on:
      - forms-db
      - platform-api

  svg-api:
    build:
      context: .
      dockerfile: infrastructure/docker/svg-api.Dockerfile
    container_name: svg-api
    ports:
      - "3002:3002"
    environment:
      NODE_ENV: development
      DATABASE_URL: postgresql://dbuser:dbpassword@svg-db:5432/svg_db
      PLATFORM_API_URL: http://platform-api:3000
      SERVICE_AUTH_TOKEN: shared-service-secret
    depends_on:
      - svg-db
      - platform-api

  links-api:
    build:
      context: .
      dockerfile: infrastructure/docker/links-api.Dockerfile
    container_name: links-api
    ports:
      - "3003:3003"
    environment:
      NODE_ENV: development
      DATABASE_URL: postgresql://dbuser:dbpassword@links-db:5432/links_db
      PLATFORM_API_URL: http://platform-api:3000
      SERVICE_AUTH_TOKEN: shared-service-secret
    depends_on:
      - links-db
      - platform-api

  # ==================== Frontend Services ====================
  dashboard:
    build:
      context: .
      dockerfile: infrastructure/docker/dashboard.Dockerfile
    container_name: dashboard
    ports:
      - "4200:4200"
    environment:
      PLATFORM_API_URL: http://localhost:3000
    volumes:
      - ./apps/dashboard:/app
      - /app/node_modules
```

```
  forms-ui:
    build:
      context: .
      dockerfile: infrastructure/docker/forms.Dockerfile
    container_name: forms-ui
    ports:
      - "4201:4201"
    environment:
      FORMS_API_URL: http://localhost:3001
      PLATFORM_API_URL: http://localhost:3000
    volumes:
      - ./apps/forms:/app
      - /app/node_modules

  svg-ui:
    build:
      context: .
      dockerfile: infrastructure/docker/svg.Dockerfile
    container_name: svg-ui
    ports:
      - "4202:4202"
    environment:
      SVG_API_URL: http://localhost:3002
      PLATFORM_API_URL: http://localhost:3000
    volumes:
      - ./apps/svg-drawing:/app
      - /app/node_modules

  # ==================== API Gateway ====================
  nginx:
    image: nginx:alpine
    container_name: api-gateway
    ports:
      - "8080:80"
    volumes:
      - ./infrastructure/nginx/api-gateway.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - platform-api
      - forms-api
      - svg-api
      - links-api

volumes:
  platform-db-data:
  forms-db-data:
  svg-db-data:
  links-db-data:
```

Nginx API Gateway Configuration

File: infrastructure/nginx/api-gateway.conf

```
events {
    worker_connections 1024;
}

http {
    upstream platform_api {
        server platform-api:3000;
    }

    upstream forms_api {
        server forms-api:3001;
    }

    upstream svg_api {
        server svg-api:3002;
    }

    upstream links_api {
        server links-api:3003;
    }

    server {
        listen 80;
        server_name localhost;

        # Platform API Routes
        location /api/auth {
            proxy_pass http://platform_api;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }

        location /api/users {
            proxy_pass http://platform_api;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        location /api/dashboard {
            proxy_pass http://platform_api;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        # Forms API Routes
        location /api/forms {
            proxy_pass http://forms_api;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        location /api/themes {
            proxy_pass http://forms_api;
```

```
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /api/public/forms {
        proxy_pass http://forms_api;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    # SVG API Routes
    location /api/drawings {
        proxy_pass http://svg_api;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /api/svg/templates {
        proxy_pass http://svg_api;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    # Links API Routes
    location /api/links {
        proxy_pass http://links_api;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    # Short link redirect (no /api prefix)
    location ~ ^/[a-zA-Z0-9]{6,10}$ {
        proxy_pass http://links_api;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
  }
}
```

Local Development Commands

```
# Start all services
docker-compose up -d

# View logs for specific service
docker-compose logs -f forms-api

# Stop all services
docker-compose down

# Rebuild specific service
docker-compose up -d --build forms-api

# Access database
```

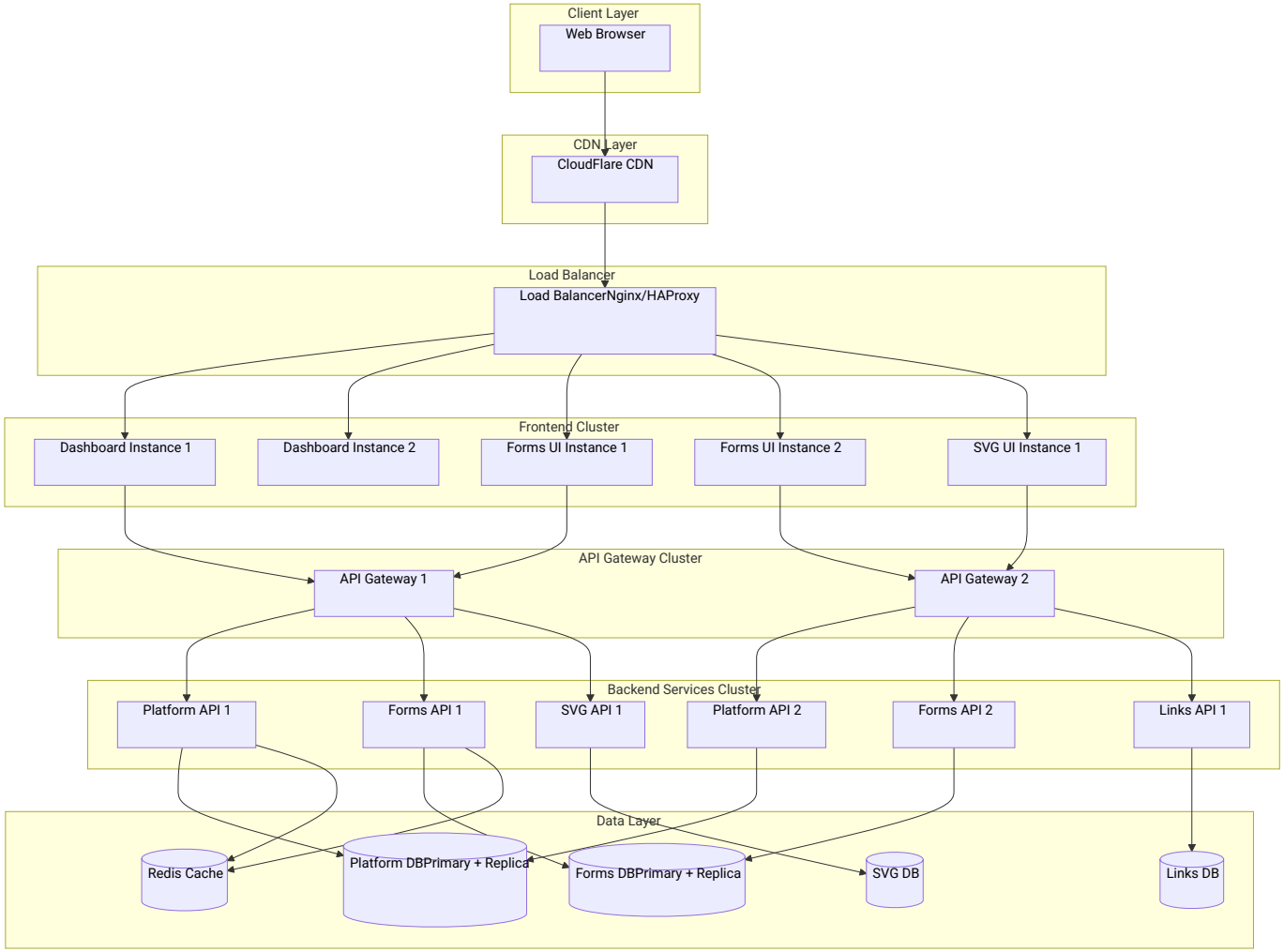
```
docker exec -it platform-db psql -U dbuser -d platform_db

# Run migrations
docker exec -it platform-api npm run db:migrate
docker exec -it forms-api npm run db:migrate
docker exec -it svg-api npm run db:migrate
docker exec -it links-api npm run db:migrate
```

Production Deployment Strategy

Domain/Subdomain Structure

```
Production URLs:
- yourapp.com                  → Main Dashboard (Port 4200)
- forms.yourapp.com            → Form Builder UI (Port 4201)
- draw.yourapp.com             → SVG Drawing UI (Port 4202)
- api.yourapp.com              → API Gateway (Nginx)
  ├── /auth                    → Platform API
  ├── /users                   → Platform API
  ├── /dashboard               → Platform API
  ├── /forms                   → Forms API
  ├── /themes                  → Forms API
  ├── /drawings                → SVG API
  └── /links                   → Links API
- link.yourapp.com/:shortCode  → Short Link Redirect
```

Deployment Architecture



Kubernetes Deployment (Recommended for Production)

File: infrastructure/k8s/forms-api-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: forms-api
  namespace: production
spec:
  replicas: 3
  selector:
    matchLabels:
      app: forms-api
  template:
    metadata:
      labels:
        app: forms-api
    spec:
      containers:
      - name: forms-api
        image: yourregistry/forms-api:latest
        ports:
        - containerPort: 3001
```

```
        env:
        - name: NODE_ENV
          value: "production"
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: forms-db-secret
              key: connection-string
        - name: PLATFORM_API_URL
          value: "http://platform-api-service:3000"
        - name: SERVICE_AUTH_TOKEN
          valueFrom:
            secretKeyRef:
              name: service-auth-secret
              key: token
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1000m"
        livenessProbe:
          httpGet:
            path: /health
            port: 3001
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health
            port: 3001
          initialDelaySeconds: 5
          periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: forms-api-service
  namespace: production
spec:
  selector:
    app: forms-api
  ports:
  - protocol: TCP
    port: 3001
    targetPort: 3001
  type: ClusterIP
```

Migration Roadmap

Phase 1: Preparation (2 weeks)

Goals:

- Set up new repository structure
- Create shared libraries
- Set up Docker development environment

Tasks:

1. ✅ Create monorepo structure with separate apps
2. ✅ Extract shared components to `@yourapp/shared-ui`
3. ✅ Extract shared types to `@yourapp/shared-types`
4. ✅ Set up Docker Compose for local development
5. ✅ Create database migration scripts
6. ✅ Set up CI/CD pipelines for each service

Deliverables:

- New repository structure
- Shared libraries published to npm workspace
- Docker Compose configuration
- Database migration plan

Phase 2: Backend Service Split (4 weeks)

Goals:

- Split monolithic API into 4 services
- Implement service-to-service authentication
- Set up API gateway

Week 1-2: Platform Service

1. ✅ Create `apps/platform-api/` with Express.js
2. ✅ Migrate authentication, users, tenants endpoints
3. ✅ Set up `platform_db` with user tables
4. ✅ Implement JWT validation endpoint for other services
5. ✅ Deploy and test in local Docker environment

Week 3: Forms Service

1. ✅ Create `apps/forms-api/` with Express.js
2. ✅ Migrate forms, schemas, submissions, themes endpoints
3. ✅ Set up `forms_db` with form tables
4. ✅ Implement Platform API integration for user validation
5. ✅ Deploy and test with Platform Service

Week 4: SVG + Links Services

1. ✅ Create `apps/svg-api/` and `apps/links-api/`
2. ✅ Migrate SVG drawing and short link endpoints
3. ✅ Set up `svg_db` and `links_db`
4. ✅ Deploy and test all services together

Deliverables:

- 4 working backend services
- All services communicating via REST
- API Gateway routing correctly
- All tests passing

Phase 3: Frontend Service Split (3 weeks)

Goals:

- Create 3 separate Angular applications
- Implement cross-app navigation
- Set up shared component library

Week 1: Dashboard App

1. ✅ Create `apps/dashboard/` Angular app
2. ✅ Migrate dashboard, auth, settings features
3. ✅ Install and use `@yourapp/shared-ui`
4. ✅ Set up routing and layout
5. ✅ Deploy to port 4200

Week 2: Forms App

1. ✅ Create `apps/forms/` Angular app
2. ✅ Migrate form-builder components
3. ✅ Migrate theme-designer components
4. ✅ Set up Forms API integration
5. ✅ Deploy to port 4201

Week 3: SVG App

1. ✅ Create `apps/svg-drawing/` Angular app
2. ✅ Migrate SVG canvas components
3. ✅ Set up SVG API integration
4. ✅ Deploy to port 4202
5. ✅ Test cross-app navigation

Deliverables:

- 3 working frontend applications
- Shared UI library in use
- Cross-app navigation working
- All features functional

Phase 4: Data Migration (1 week)

Goals:

- Migrate data from monolithic database to service-specific databases
- Validate data integrity

Tasks:

1. ✅ Run database split migration script
2. ✅ Validate user data in `platform_db`
3. ✅ Validate form data in `forms_db`
4. ✅ Validate SVG data in `svg_db`
5. ✅ Validate short link data in `links_db`
6. ✅ Run integration tests across all services
7. ✅ Set up database backups

Deliverables:

- All data migrated successfully
- Data integrity validated
- Backup strategy in place

Phase 5: Production Deployment (2 weeks)

Goals:

- Deploy to production environment
- Set up monitoring and logging
- Run performance tests

Week 1: Infrastructure Setup

1. ✅ Set up Kubernetes cluster (or VPS servers)
2. ✅ Configure DNS for subdomains
3. ✅ Set up SSL certificates
4. ✅ Deploy database clusters
5. ✅ Set up Redis cache

Week 2: Service Deployment

1. ✅ Deploy Platform API to production
2. ✅ Deploy Forms API to production
3. ✅ Deploy SVG API to production
4. ✅ Deploy Links API to production
5. ✅ Deploy frontend applications
6. ✅ Configure API Gateway with production URLs
7. ✅ Run smoke tests
8. ✅ Monitor for 24 hours

Deliverables:

- All services running in production
- Monitoring and alerting configured
- Performance validated
- Documentation updated

Phase 6: Optimization (Ongoing)

Goals:

- Optimize performance
- Implement caching strategies
- Add event-driven communication

Tasks:

1. Add Redis caching for hot data
2. Implement database read replicas
3. Set up CDN for static assets
4. Add RabbitMQ for async events
5. Implement rate limiting per service
6. Optimize Docker images
7. Set up auto-scaling rules

Pros and Cons Analysis

☑ Advantages of Microservices

1. Independent Deployment

- Deploy Form Builder updates without touching SVG Drawing
- Faster release cycles per feature
- Reduced risk of breaking unrelated features

2. Technology Flexibility

- Could rewrite SVG service in Go for better performance
- Could use different databases per service (e.g., MongoDB for SVG)
- Experiment with new technologies without full system risk

3. Scalability

- Scale Form Builder (high traffic) independently from SVG Drawing (low traffic)
- Optimize resource allocation per service
- Cost savings by not over-provisioning all services

4. Team Autonomy

- Separate teams can own complete services
- Clearer ownership and responsibility
- Parallel development without merge conflicts

5. Fault Isolation

- SVG service crash doesn't affect Form Builder
- Database issues in one service don't cascade
- Easier to debug and fix isolated failures

6. Better Code Organization

- Smaller codebases are easier to understand
- Clear service boundaries
- Reduced coupling between features

❌ Disadvantages of Microservices

1. Increased Complexity

- More moving parts to manage
- Inter-service communication adds latency
- Harder to debug distributed systems
- More complex local development setup

2. Operational Overhead

- Need to manage 4 databases instead of 1
- More deployment pipelines
- More monitoring and logging infrastructure
- Higher hosting costs (multiple servers/containers)

3. Data Consistency Challenges

- No foreign key constraints across services
- Eventual consistency patterns required
- Complex transactions spanning multiple services
- Data duplication across services

4. Network Latency

- Service-to-service calls add overhead
- Potential for cascade failures
- Need sophisticated retry/timeout logic
- More complex error handling

5. Testing Complexity

- Integration testing requires all services running
- End-to-end testing more complex
- Mocking inter-service calls in tests
- Harder to reproduce bugs

6. Higher Initial Investment

- More upfront development time
- Need Docker/Kubernetes expertise
- More infrastructure to set up
- Steeper learning curve for team

Cost and Complexity Assessment

Development Effort Estimate

Phase	Duration	Developer-Weeks	Complexity
Phase 1: Preparation	2 weeks	4 dev-weeks	Medium
Phase 2: Backend Split	4 weeks	12 dev-weeks	High
Phase 3: Frontend Split	3 weeks	9 dev-weeks	High
Phase 4: Data Migration	1 week	2 dev-weeks	Medium
Phase 5: Production Deployment	2 weeks	6 dev-weeks	High
Total	**12 weeks**	**33 dev-weeks**	**Very High**

Estimated Cost (2 developers):

- 12 weeks × 2 developers = 24 developer-weeks
- At $100/hour × 40 hours/week = $4,000/week
- **Total: ~$96,000 in labor costs**

Infrastructure Cost Comparison

Current Monolith (Monthly):

- 1 VPS Server (4 vCPU, 8GB RAM): $40
- 1 PostgreSQL Database: $25
- Total: **$65/month**

Microservices (Monthly):

- 4 VPS Servers (2 vCPU, 4GB RAM each): $20 × 4 = $80
- 4 PostgreSQL Databases: $15 × 4 = $60
- 1 Redis Instance: $20
- 1 Load Balancer: $20
- Total: **$180/month** (2.8x increase)

Kubernetes Cluster (Alternative):

- Managed Kubernetes Cluster: $75
- 6 Worker Nodes (2 vCPU, 4GB RAM): $15 × 6 = $90
- 4 Managed Databases: $15 × 4 = $60
- Total: **$225/month** (3.5x increase)

Operational Complexity Increase

Metric	Monolith	Microservices	Increase
Deployment Pipelines	1	7 (4 backend + 3 frontend)	7x
Databases to Manage	1	4	4x
Services to Monitor	2	8+	4x
Network Calls	0 (internal)	High (inter-service)	∞
Configuration Files	2	10+	5x
Docker Containers	2	8+	4x

Decision Framework

When Microservices Make Sense

✅ YES, proceed if:

1. You have **multiple teams** working on different features
2. Different tools have **vastly different scaling needs**
3. You need to **deploy features independently**
4. You have **DevOps expertise** on the team
5. **Budget allows** for 3x infrastructure cost increase
6. You're building for **long-term scalability** (1000+ users)

❌ NO, stay monolithic if:

1. You're a **solo developer** or small team (< 3 people)
2. All features have **similar traffic patterns**
3. You need **rapid feature development** without operational overhead
4. **Budget is constrained** (<$500/month infrastructure)
5. You're in **MVP/early stage** (< 100 users)
6. Team **lacks microservices experience**

Hybrid Approach (Recommended Starting Point)

Instead of full microservices, consider **Modular Monolith**:

1. **Keep single database** (easier data consistency)
2. **Keep single backend API** but organize as modules: ``` apps/api/ modules/ platform/ # Auth, users forms/ # Form builder svg/ # SVG drawing links/ # Short links ```
3. **Split frontend into separate apps** (easier than backend split): - Dashboard at `:4200` - Forms at `:4201` - SVG at `:4202`
4. **Use shared component library** as proposed
5. **Prepare for future split** by enforcing module boundaries

Benefits:

- ✅ Frontend independence (most of the UI benefits)
- ✅ Lower operational complexity
- ✅ Easier local development
- ✅ Single database (easier consistency)
- ✅ Can migrate to full microservices later if needed

Recommendations

For Small Team (1-3 developers)

Recommendation: Modular Monolith

Phase 1: Split Frontend Only (4 weeks)
Create 3 Angular apps
Shared component library
Keep single backend API
Keep single database

Phase 2: Optimize Backend Modules (2 weeks)
Organize API into modules
Enforce clear boundaries
Prepare for future split

Total: 6 weeks, ~$48,000 investment

For Medium Team (4-8 developers)

Recommendation: Gradual Microservices

Phase 1: Split Frontend (4 weeks)

Phase 2: Extract Links Service (2 weeks)
  Simplest service, good starting point
Phase 3: Extract Forms Service (3 weeks)
  Most complex, do last
Phase 4: Extract SVG Service (2 weeks)
  Medium complexity
Phase 5: Platform Service (1 week)
  Consolidate remaining

Total: 12 weeks, ~$96,000 investment

For Large Team (8+ developers)

Recommendation: Full Microservices

Follow complete roadmap in Phase 1-6
  Independent team per service
  Full DevOps automation
  Event-driven architecture
  Auto-scaling infrastructure

Total: 12 weeks, ~$120,000+ investment

Appendix: Example Service Implementation

Forms Service - Complete Structure

```
apps/forms-api/
  src/
    controllers/
      forms.controller.ts
      themes.controller.ts
      submissions.controller.ts
      public-forms.controller.ts
    services/
      forms.service.ts
      themes.service.ts
      submissions.service.ts
      platform-integration.service.ts
    repositories/
      forms.repository.ts
      themes.repository.ts
      submissions.repository.ts
    middleware/
      auth.middleware.ts
      validation.middleware.ts
    validators/
      forms.validator.ts
      themes.validator.ts
```

```
│   │   ├── routes/
│   │   │   ├── forms.routes.ts
│   │   │   ├── themes.routes.ts
│   │   │   └── index.ts
│   │   ├── config/
│   │   │   ├── database.config.ts
│   │   │   └── app.config.ts
│   │   └── server.ts
│   ├── database/
│   │   ├── migrations/
│   │   │   ├── 001_create_forms_table.sql
│   │   │   ├── 002_create_schemas_table.sql
│   │   │   └── 003_create_themes_table.sql
│   │   └── seeds/
│   │       └── default_themes.sql
│   ├── tests/
│   │   ├── unit/
│   │   │   ├── forms.service.test.ts
│   │   │   └── themes.service.test.ts
│   │   └── integration/
│   │       ├── forms.api.test.ts
│   │       └── themes.api.test.ts
│   ├── Dockerfile
│   ├── package.json
│   ├── tsconfig.json
│   └── .env.example
```

Next Steps

1. **Review this document** with your team
2. **Decide on approach**: Full Microservices vs Modular Monolith vs Hybrid
3. **Estimate resources**: Developer time, budget, timeline
4. **Assess team skills**: Docker, Kubernetes, microservices experience
5. **Start with Phase 1** if proceeding: Set up repository structure
6. **Consider hiring** microservices consultant if team lacks expertise

---

Document Version: 1.0 **Last Updated:** 2025-10-23 **Status:** Awaiting Decision

---

Questions for Decision Making

1. **Team Size**: How many developers will work on this project?
2. **Timeline**: How urgently do you need to ship new features?
3. **Budget**: What's your monthly infrastructure budget?

4. **Expertise**: Does your team have microservices/DevOps experience?
5. **Scale**: How many concurrent users do you expect in 6 months? 1 year?
6. **Deployment Frequency**: How often do you want to deploy each tool?
7. **Business Need**: Do different tools require independent scaling?

Please answer these questions to determine the best path forward.