Finance Flow - Software Design Specification

1. Executive Summary

1.1 Project Overview

Finance Flow is a comprehensive personal finance management web application designed for local deployment. The application provides centralized tracking and management of income, expenses, bills, debts, assets, insurance policies, and financial accounts with data visualization and financial insights.

1.2 Target Architecture

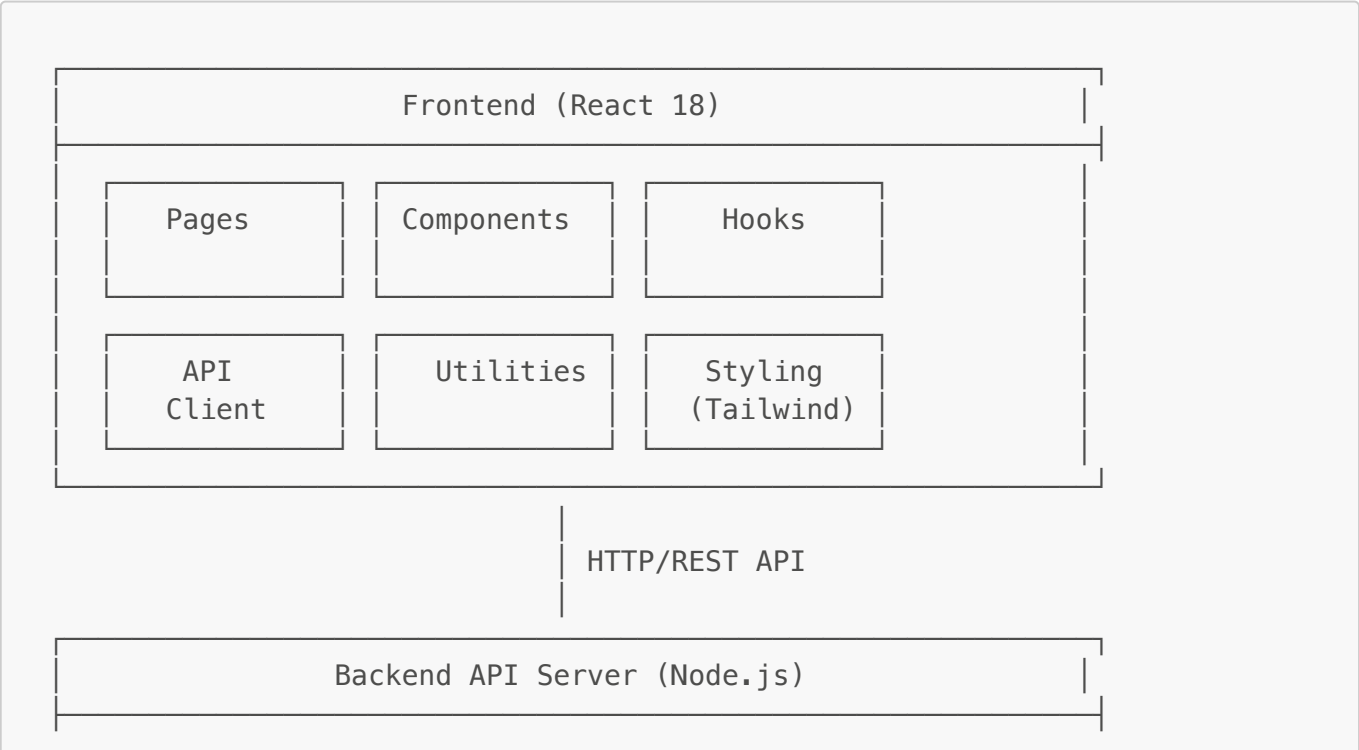
- **Frontend**: React 18 with modern hooks and functional components
- **Backend**: Node.js/Express API server (to be implemented)
- **Database**: PostgreSQL for local data storage
- **Deployment**: Local hosting with optional Docker containerization
- **Authentication**: Local user management (no cloud dependencies)

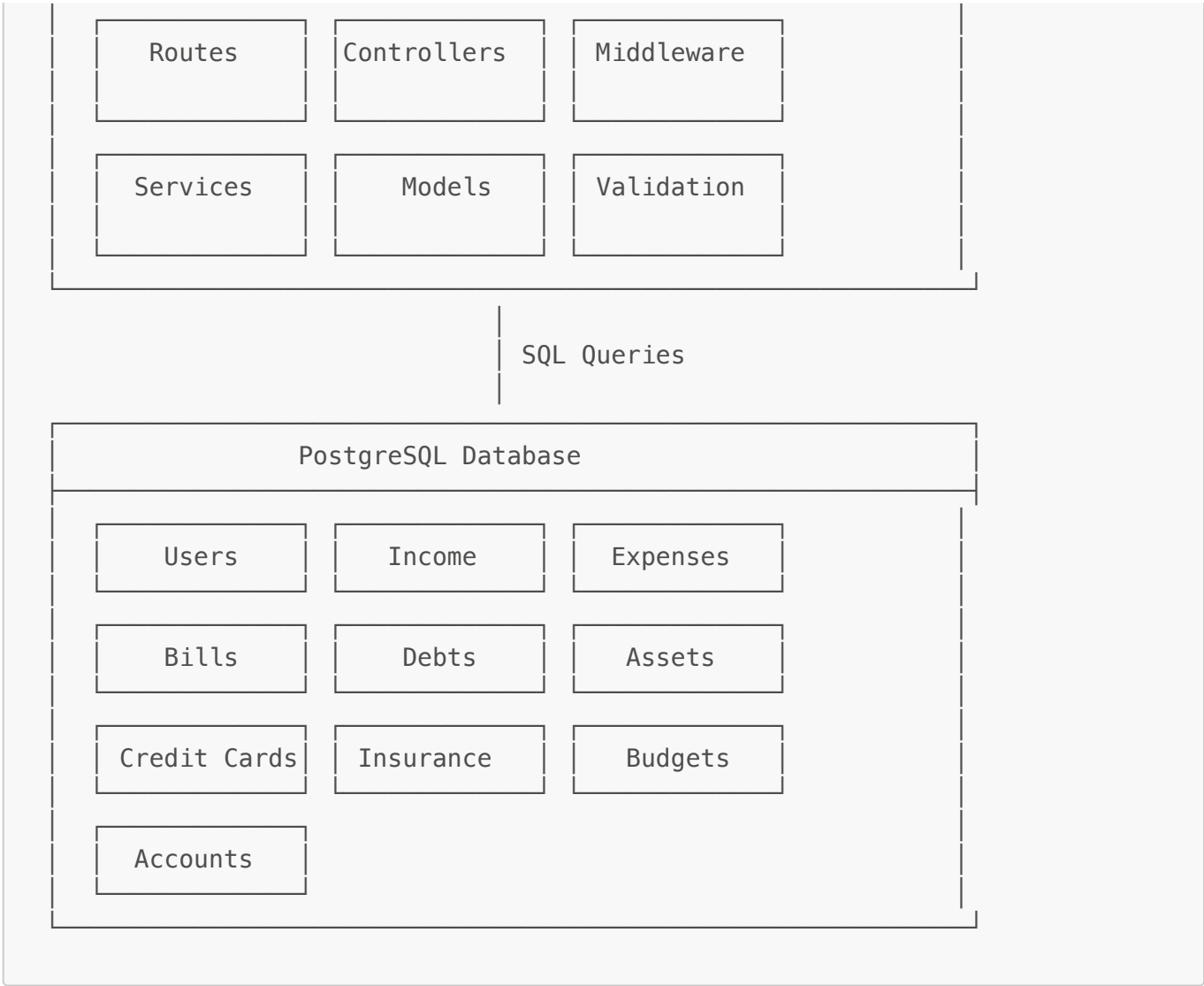
1.3 Design Objectives

- **Local-First**: Complete functionality without internet dependencies
- **Data Privacy**: All financial data stored locally
- **User Experience**: Intuitive interface inspired by spreadsheet layouts
- **Performance**: Fast loading and responsive UI
- **Maintainability**: Clean architecture with separation of concerns

2. System Architecture

2.1 High-Level Architecture





2.2 Technology Stack

Frontend

- **React 18.2.0**: Modern React with concurrent features
- **React Router DOM 7**: Client-side routing and navigation
- **Vite 5**: Build tool and development server
- **Tailwind CSS 3**: Utility-first CSS framework
- **Shadcn/ui**: Component library built on Radix UI primitives
- **React Hook Form**: Form management with validation
- **Zod**: TypeScript-first schema validation
- **Recharts**: Data visualization library
- **Date-fns**: Date manipulation library
- **Lucide React**: Icon library

Backend (To Be Implemented)

- **Node.js 18+**: Runtime environment
- **Express.js**: Web application framework
- **PostgreSQL 14+**: Relational database
- **Prisma/TypeORM**: Database ORM

- **bcrypt**: Password hashing
- **jsonwebtoken**: JWT authentication
- **helmet**: Security middleware
- **cors**: Cross-origin resource sharing

Development & Testing

- **ESLint**: Code linting
- **Prettier**: Code formatting
- **Jest**: Testing framework
- **React Testing Library**: Component testing
- **Supertest**: API testing
- **Docker**: Containerization (optional)

3. Frontend Design

3.1 Component Architecture

3.1.1 Page Components (11 Main Pages)

```
src/pages/
├── Dashboard.jsx          # Financial overview with charts and metrics
├── Income.jsx            # Income tracking and categorization
├── Expenses.jsx          # Expense management with categories
├── Bills.jsx             # Bill tracking with due dates
├── Debts.jsx             # Debt portfolio management
├── Assets.jsx            # Asset tracking and valuation
├── Budget.jsx            # Budget planning and comparison
├── DebtReduction.jsx     # Debt payoff strategies
├── CreditLoans.jsx       # Credit card and loan management
├── Insurance.jsx         # Insurance policy tracking
├── Accounts.jsx          # Secure credential storage
└── Layout.jsx            # Sidebar navigation and layout
```

3.1.2 UI Component Library (Shadcn/ui)

```
src/components/ui/
├── Form Components       # Input, Select, Textarea, Checkbox, etc.
├── Display Components    # Card, Table, Badge, Avatar, etc.
├── Navigation           # Button, Tabs, Breadcrumb, Pagination
├── Feedback            # Alert, Toast, Dialog, Progress
├── Data Visualization  # Charts (via Recharts integration)
└── Layout              # Container, Grid, Flex utilities
```

3.1.3 Custom Hooks

```
src/hooks/
├── useAuth.js           # Authentication state management
├── useApi.js            # API call wrapper with error handling
├── useLocalStorage.js   # Local storage persistence
├── useDebounce.js       # Input debouncing for search/filter
└── useMobile.js         # Mobile device detection
```

3.2 State Management Strategy

3.2.1 Local State (React Hooks)

- Component-specific state using `useState`
- Complex state logic using `useReducer`
- Side effects and data fetching using `useEffect`

3.2.2 Global State (Context API)

```
// AuthContext - User authentication state
const AuthContext = createContext({
  user: null,
  login: () => {},
  logout: () => {},
  isAuthenticated: false
});

// ThemeContext - UI theme management
const ThemeContext = createContext({
  theme: 'light',
  toggleTheme: () => {}
});
```

3.2.3 Server State (React Query - Future Enhancement)

- Caching of API responses
- Background refetching
- Optimistic updates
- Error handling and retries

3.3 Routing Structure

```
// src/pages/index.jsx
const routes = [
  { path: '/', element: <Dashboard /> },
  { path: '/income', element: <Income /> },
  { path: '/expenses', element: <Expenses /> },
  { path: '/bills', element: <Bills /> },
```

```
  { path: '/debts', element: <Debts /> },
  { path: '/assets', element: <Assets /> },
  { path: '/budget', element: <Budget /> },
  { path: '/debt-reduction', element: <DebtReduction /> },
  { path: '/credit-loans', element: <CreditLoans /> },
  { path: '/insurance', element: <Insurance /> },
  { path: '/accounts', element: <Accounts /> }
];
```

3.4 Styling System

3.4.1 Tailwind Configuration

```
// tailwind.config.js
module.exports = {
  content: ['./src/**/*.{js,jsx}'],
  theme: {
    extend: {
      colors: {
        primary: { /* Custom color palette */ },
        secondary: { /* Secondary colors */ }
      },
      fontFamily: {
        sans: ['Inter', 'system-ui', 'sans-serif']
      }
    }
  },
  plugins: [require('@tailwindcss/forms')]
};
```

3.4.2 Component Styling Approach

- **Utility Classes**: Primary styling method using Tailwind
- **Component Variants**: Shadcn/ui component variants
- **Custom CSS**: Minimal custom CSS for complex animations
- **Responsive Design**: Mobile-first responsive classes

4. Backend Design (To Be Implemented)

4.1 API Architecture

4.1.1 RESTful API Endpoints

```
Authentication:
POST   /api/auth/register     # User registration
POST   /api/auth/login        # User login
POST   /api/auth/logout       # User logout
```

```
GET    /api/auth/me          # Get current user

Income Management:
GET    /api/income           # List user's income records
POST   /api/income           # Create income record
PUT    /api/income/:id       # Update income record
DELETE /api/income/:id       # Delete income record

Expense Management:
GET    /api/expenses         # List user's expenses
POST   /api/expenses         # Create expense record
PUT    /api/expenses/:id     # Update expense record
DELETE /api/expenses/:id     # Delete expense record

[Similar patterns for Bills, Debts, Assets, etc.]
```

4.1.2 Middleware Stack

```
// Express middleware configuration
app.use(helmet());                    // Security headers
app.use(cors(corsOptions));           // CORS configuration
app.use(express.json({ limit: '10mb' })); // JSON parsing
app.use(express.urlencoded({ extended: true }));
app.use(authMiddleware);              // JWT authentication
app.use(validationMiddleware);        // Request validation
app.use(errorHandlerMiddleware);      // Error handling
```

4.2 Database Layer

4.2.1 Database Connection

```
// Database configuration with connection pooling
const pool = new Pool({
  host: process.env.DB_HOST || 'localhost',
  port: process.env.DB_PORT || 5432,
  database: process.env.DB_NAME || 'finance_flow',
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  max: 20,
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000,
});
```

4.2.2 Data Access Layer

```
// Example service layer for Income entity
class IncomeService {
  async findByUserId(userId, filters = {}) {
    const query = `
      SELECT * FROM income
      WHERE user_id = $1
      ORDER BY date_received DESC
    `;
    return await db.query(query, [userId]);
  }

  async create(userId, incomeData) {
    const query = `
      INSERT INTO income (user_id, source, amount, category,
date_received)
      VALUES ($1, $2, $3, $4, $5)
      RETURNING *
    `;
    return await db.query(query, [userId, ...Object.values(incomeData)]);
  }
}
```

4.3 Security Implementation

4.3.1 Authentication Strategy

- **JWT Tokens**: Stateless authentication
- **Password Hashing**: bcrypt with salt rounds
- **Session Management**: Token refresh mechanism
- **Rate Limiting**: Prevent brute force attacks

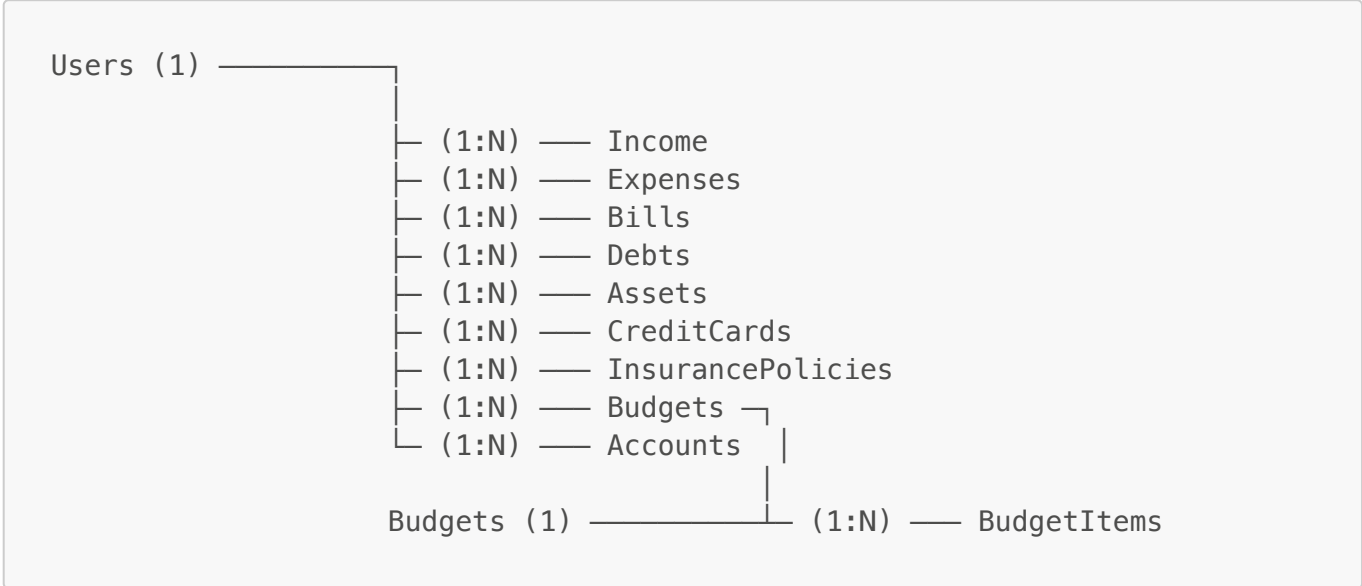
4.3.2 Data Protection

```
// Encryption for sensitive data
const crypto = require('crypto');

class EncryptionService {
  encrypt(text) {
    const cipher = crypto.createCipher('aes-256-cbc',
process.env.ENCRYPTION_KEY);
    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');
    return encrypted;
  }

  decrypt(encryptedText) {
    const decipher = crypto.createDecipher('aes-256-cbc',
process.env.ENCRYPTION_KEY);
    let decrypted = decipher.update(encryptedText, 'hex', 'utf8');
```

```
    decrypted += decipher.final('utf8');
    return decrypted;
  }
}
```

5. Data Model Design

5.1 Entity Relationships



5.2 Data Validation Rules

5.2.1 Common Validation Patterns

```
// Zod schemas for data validation
const IncomeSchema = z.object({
  source: z.string().min(1).max(255),
  amount: z.number().positive(),
  category: z.enum(['salary', 'freelance', 'investment', 'rental',
'business', 'other']),
  date_received: z.date(),
  frequency: z.enum(['weekly', 'bi-weekly', 'monthly', 'quarterly',
'yearly']),
  is_recurring: z.boolean().default(false)
});
```

5.2.2 Business Logic Constraints

- **Amounts**: Must be positive numbers with 2 decimal places
- **Dates**: Cannot be future dates for historical records
- **Categories**: Must match predefined category lists
- **Frequencies**: Limited to supported frequency options
- **User Isolation**: All records must belong to authenticated user

6. User Interface Design

6.1 Design System

6.1.1 Color Palette

```
:root {
  --primary-50: #eff6ff;
  --primary-500: #3b82f6;
  --primary-900: #1e3a8a;

  --secondary-50: #f8fafc;
  --secondary-500: #64748b;
  --secondary-900: #0f172a;

  --success: #10b981;
  --warning: #f59e0b;
  --error: #ef4444;
}
```

6.1.2 Typography Scale

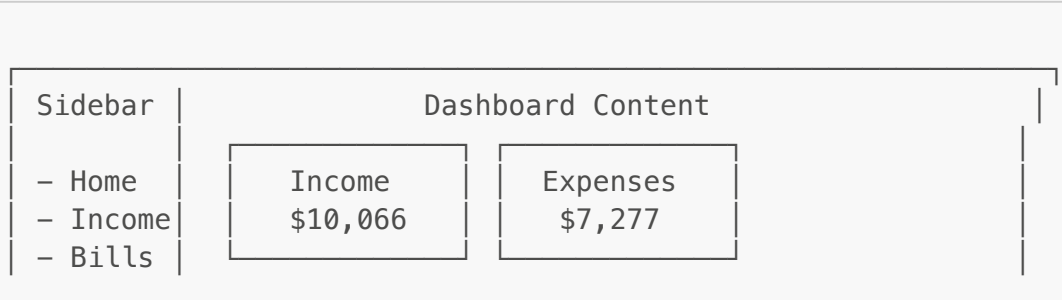
```
.text-xs { font-size: 0.75rem; }    /* 12px */
.text-sm { font-size: 0.875rem; }   /* 14px */
.text-base { font-size: 1rem; }     /* 16px */
.text-lg { font-size: 1.125rem; }   /* 18px */
.text-xl { font-size: 1.25rem; }    /* 20px */
.text-2xl { font-size: 1.5rem; }    /* 24px */
.text-3xl { font-size: 1.875rem; }  /* 30px */
```

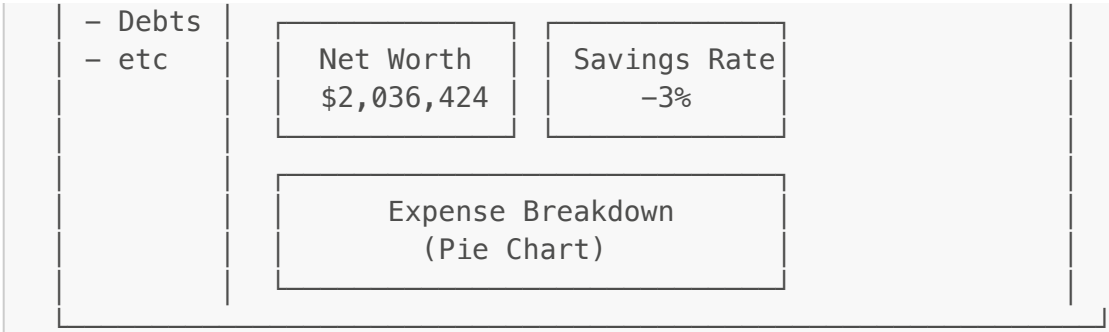
6.1.3 Spacing System

- **Base Unit**: 4px (0.25rem)
- **Common Spacing**: 4px, 8px, 12px, 16px, 24px, 32px, 48px, 64px
- **Container Widths**: sm(640px), md(768px), lg(1024px), xl(1280px)

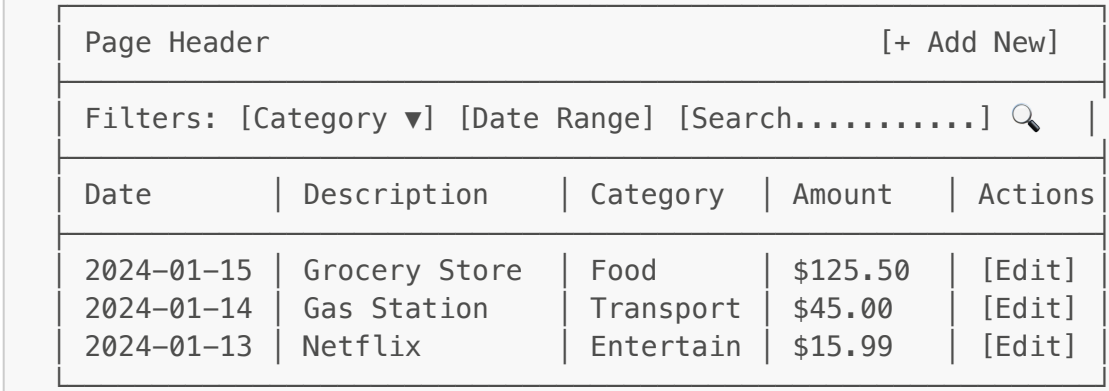
6.2 Layout Patterns

6.2.1 Dashboard Layout





6.2.2 Data Table Layout



6.3 Responsive Design Strategy

6.3.1 Breakpoint Strategy

```
/* Mobile First Approach */
.container { @apply px-4; }                    /* Mobile: < 640px */
@screen sm { .container { @apply px-6; } }     /* Small: >= 640px */
@screen md { .container { @apply px-8; } }     /* Medium: >= 768px */
@screen lg { .container { @apply px-12; } }    /* Large: >= 1024px */
@screen xl { .container { @apply px-16; } }    /* XLarge: >= 1280px */
```

6.3.2 Sidebar Behavior

- **Desktop**: Fixed sidebar with full navigation
- **Tablet**: Collapsible sidebar with hamburger menu
- **Mobile**: Bottom navigation or slide-out drawer

7. Performance Considerations

7.1 Frontend Optimization

7.1.1 Code Splitting

```
// Lazy loading of page components
const Dashboard = lazy(() => import('./pages/Dashboard'));
const Income = lazy(() => import('./pages/Income'));
const Expenses = lazy(() => import('./pages/Expenses'));

// Route-based code splitting
<Routes>
  <Route path="/" element={
    <Suspense fallback={<Loading />}>
      <Dashboard />
    </Suspense>
  } />
</Routes>
```

7.1.2 Data Fetching Optimization

```
// Debounced search to reduce API calls
const useDebounce = (value, delay) => {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => clearTimeout(handler);
  }, [value, delay]);

  return debouncedValue;
};
```

7.1.3 Memoization Strategy

```
// Memoize expensive calculations
const financialMetrics = useMemo(() => {
  return calculateFinancialMetrics(income, expenses, debts, assets);
}, [income, expenses, debts, assets]);

// Memoize callback functions
const handleFilter = useCallback((filters) => {
  setCurrentFilters(filters);
}, []);
```

7.2 Backend Optimization

7.2.1 Database Query Optimization

```
-- Proper indexing for common queries
CREATE INDEX idx_expenses_user_date ON expenses(user_id, date DESC);
CREATE INDEX idx_income_user_category ON income(user_id, category);

-- Efficient pagination queries
SELECT * FROM expenses
WHERE user_id = $1
ORDER BY date DESC
LIMIT $2 OFFSET $3;
```

7.2.2 Caching Strategy

```
// Memory caching for frequently accessed data
const NodeCache = require('node-cache');
const cache = new NodeCache({ stdTTL: 600 }); // 10 minutes

const getCachedUserSummary = async (userId) => {
  const cacheKey = `user_summary_${userId}`;
  let summary = cache.get(cacheKey);

  if (!summary) {
    summary = await calculateUserSummary(userId);
    cache.set(cacheKey, summary);
  }

  return summary;
};
```

8. Security Architecture

8.1 Authentication & Authorization

8.1.1 JWT Implementation

```
// JWT token generation
const generateToken = (user) => {
  return jwt.sign(
    {
      userId: user.id,
      email: user.email
    },
    process.env.JWT_SECRET,
    {
      expiresIn: '24h',
      issuer: 'finance-flow',
      audience: 'finance-flow-users'
    }
```

```
  );
};
```

8.1.2 Authorization Middleware

```
const requireAuth = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];

  if (!token) {
    return res.status(401).json({ error: 'Access token required' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(401).json({ error: 'Invalid token' });
  }
};
```

8.2 Data Protection

8.2.1 Input Validation

```
// Request validation middleware
const validateRequest = (schema) => {
  return (req, res, next) => {
    try {
      const validatedData = schema.parse(req.body);
      req.body = validatedData;
      next();
    } catch (error) {
      return res.status(400).json({
        error: 'Validation failed',
        details: error.errors
      });
    }
  };
};
```

8.2.2 SQL Injection Prevention

```
// Parameterized queries
const getUserExpenses = async (userId, filters) => {
  const query = `
```

```
    SELECT * FROM expenses
    WHERE user_id = $1
    AND date >= $2
    AND date <= $3
    ORDER BY date DESC
  `;
  return await db.query(query, [userId, filters.startDate,
filters.endDate]);
};
```

9. Testing Strategy

9.1 Frontend Testing

9.1.1 Unit Testing

```
// Component testing example
import { render, screen, fireEvent } from '@testing-library/react';
import { IncomeForm } from '../components/IncomeForm';

test('should submit form with valid data', async () => {
  const mockSubmit = jest.fn();
  render(<IncomeForm onSubmit={mockSubmit} />);

  fireEvent.change(screen.getByLabelText(/source/i), {
    target: { value: 'Salary' }
  });
  fireEvent.change(screen.getByLabelText(/amount/i), {
    target: { value: '5000' }
  });

  fireEvent.click(screen.getByRole('button', { name: /save/i }));

  await waitFor(() => {
    expect(mockSubmit).toHaveBeenCalledWith({
      source: 'Salary',
      amount: 5000
    });
  });
});
```

9.1.2 Integration Testing

```
// API integration testing
test('should fetch and display income data', async () => {
  const mockIncomeData = [
    { id: 1, source: 'Salary', amount: 5000, category: 'salary' }
  ];
```

```
  jest.spyOn(api, 'getIncome').mockResolvedValue(mockIncomeData);

  render(<IncomePage />);

  await waitFor(() => {
    expect(screen.getByText('Salary')).toBeInTheDocument();
    expect(screen.getByText('$5,000')).toBeInTheDocument();
  });
});
```

9.2 Backend Testing

9.2.1 API Testing

```
// API endpoint testing
const request = require('supertest');
const app = require('../app');

describe('Income API', () => {
  test('POST /api/income should create income record', async () => {
    const incomeData = {
      source: 'Freelance',
      amount: 1500,
      category: 'freelance',
      date_received: '2024-01-15'
    };

    const response = await request(app)
      .post('/api/income')
      .set('Authorization', `Bearer ${authToken}`)
      .send(incomeData)
      .expect(201);

    expect(response.body.source).toBe('Freelance');
    expect(response.body.amount).toBe(1500);
  });
});
```

9.2.2 Database Testing

```
// Database integration testing
describe('Income Service', () => {
  beforeEach(async () => {
    await db.query('TRUNCATE TABLE income RESTART IDENTITY CASCADE');
  });

  test('should create and retrieve income record', async () => {
```

```
    const incomeData = {
      source: 'Salary',
      amount: 5000,
      category: 'salary'
    };

    const created = await IncomeService.create(userId, incomeData);
    const retrieved = await IncomeService.findById(created.id);

    expect(retrieved.source).toBe('Salary');
    expect(retrieved.amount).toBe(5000);
  });
});
```

10. Deployment Architecture

10.1 Local Development Setup

10.1.1 Environment Configuration

```
# .env.development
NODE_ENV=development
DB_HOST=localhost
DB_PORT=5432
DB_NAME=finance_flow_dev
DB_USER=dev_user
DB_PASSWORD=dev_password
JWT_SECRET=dev_jwt_secret_key
ENCRYPTION_KEY=dev_encryption_key
```

10.1.2 Docker Configuration (Optional)

```
# Dockerfile
FROM node:18-alpine

WORKDIR /app

# Install dependencies
COPY package*.json ./
RUN npm ci --only=production

# Copy application code
COPY . .

# Build frontend
RUN npm run build

EXPOSE 3000
```

```
CMD ["npm", "start"]
```

```
# docker-compose.yml
version: '3.8'
services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - DB_HOST=postgres
    depends_on:
      - postgres

  postgres:
    image: postgres:14-alpine
    environment:
      POSTGRES_DB: finance_flow
      POSTGRES_USER: finance_user
      POSTGRES_PASSWORD: secure_password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  postgres_data:
```

10.2 Production Deployment

10.2.1 Build Process

```
{
  "scripts": {
    "build": "vite build",
    "build:server": "babel server -d dist",
    "start": "node dist/server.js",
    "migrate": "npm run migrate:run",
    "seed": "npm run seed:run"
  }
}
```

10.2.2 Database Migration

```
// migrations/001_initial_schema.js
exports.up = async (knex) => {
  // Create users table
  await knex.schema.createTable('users', (table) => {
    table.increments('id').primary();
    table.string('email').unique().notNullable();
    table.string('password_hash').notNullable();
    table.string('name');
    table.timestamps(true, true);
  });

  // Create income table
  await knex.schema.createTable('income', (table) => {
    table.increments('id').primary();
    table.integer('user_id').references('users.id').onDelete('CASCADE');
    table.string('source').notNullable();
    table.decimal('amount', 12, 2).notNullable();
    table.string('category').notNullable();
    table.date('date_received').notNullable();
    table.timestamps(true, true);
  });

  // Create indexes
  await knex.raw('CREATE INDEX idx_income_user_id ON income(user_id)');
};
```

11. Maintenance & Monitoring

11.1 Logging Strategy

11.1.1 Application Logging

```
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.errors({ stack: true }),
    winston.format.json()
  ),
  transports: [
    new winston.transports.File({ filename: 'logs/error.log', level:
'error' }),
    new winston.transports.File({ filename: 'logs/combined.log' }),
    new winston.transports.Console({
      format: winston.format.simple()
    })
  ]
});
```

11.1.2 Database Monitoring

```
-- Query performance monitoring
SELECT query, calls, total_time, mean_time
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;

-- Database size monitoring
SELECT
    pg_size_pretty(pg_database_size('finance_flow')) as database_size,
    pg_size_pretty(pg_total_relation_size('expenses')) as
expenses_table_size;
```

11.2 Backup & Recovery

11.2.1 Backup Strategy

```
#!/bin/bash
# Daily backup script
DATE=$(date +%Y%m%d)
pg_dump finance_flow > backups/finance_flow_$DATE.sql
gzip backups/finance_flow_$DATE.sql

# Keep last 30 days of backups
find backups/ -name "*.gz" -mtime +30 -delete
```

11.2.2 Recovery Procedures

```
# Database restoration
createdb finance_flow_restored
gunzip -c backups/finance_flow_20240115.sql.gz | psql
finance_flow_restored
```

This Software Design Specification provides a comprehensive blueprint for implementing the Finance Flow application with PostgreSQL integration while maintaining all existing functionality and ensuring robust, scalable, and secure local deployment.