API Integration Utility - Design Document

Project Overview

The API Integration Utility is a full-stack web application designed to manage, configure, and execute chained API integrations. The system allows users to create workflows that execute multiple API endpoints in sequence, with data flowing between them based on predefined mappings.

Architecture Overview

Technology Stack

Backend:

- .NET Core 9.0 (ASP.NET Core Web API)
- Entity Framework Core (ORM)
- PostgreSQL (Database)
- JWT Authentication
- Docker containerization

Frontend:

- React 18 with TypeScript
- Material-UI (MUI) for UI components
- Axios for HTTP client
- React Router for navigation
- Tailwind CSS for styling

Infrastructure:

- Docker Compose for orchestration
- PostgreSQL database
- Nginx for frontend serving

Backend Design

1. API Architecture

The backend follows a clean architecture pattern with clear separation of concerns:

- **Controllers**: Handle HTTP requests and responses
- **Models**: Entity models representing database tables
- **DTOs**: Data Transfer Objects for API communication
- **Data Layer**: Entity Framework context and migrations
- **Services**: Business logic (implicit in controllers for this project)

2. Key Design Patterns

- **Repository Pattern**: Implemented through Entity Framework Core DbContext
- **DTO Pattern**: Separate objects for API input/output to decouple internal models from external contracts
- **Dependency Injection**: Used throughout the application for loose coupling

3. Authentication & Authorization

- JWT-based authentication system
- Token-based session management
- Password validation with complexity requirements
- Protected API endpoints requiring authentication

4. API Chaining Logic

Mapping System:

- `api_endpoint_mapping.json`: Maps HTTP method + URL combinations to short names
- `chain_mapping.json`: Defines input requirements and output mappings for each endpoint

Execution Flow:

1. Load integration configuration from database
2. Execute endpoints in sequence order
3. Extract data from responses using JSON path mappings
4. Pass extracted data to subsequent endpoints as parameters
5. Maintain execution context throughout the chain

Context Management:

- Dictionary-based context that persists across API calls
- Parameter substitution in URLs using `{parameter}` syntax
- Automatic data extraction from JSON responses

Database Design

1. Database Schema

Users Table:

```
diff
CopyEdit
- Id (Primary Key, Auto-increment)
- Username (Unique, Required, Max 100 chars)
- PasswordHash (Required)
- CreatedAt (Default: CURRENT_TIMESTAMP)
```

ApiEndpoints Table:

```
diff
CopyEdit
- Id (Primary Key, Auto-increment)
- Name (Required, Max 100 chars)
```

```
- Url (Required, Max 500 chars)
- Method (Required, Max 10 chars)
- Description (Required)
- Category (Max 50 chars)
- CreatedAt (Default: CURRENT_TIMESTAMP)
- UpdatedAt (Nullable)
```

ApiIntegrations Table:

```
pgsql
CopyEdit
- Id (Primary Key, Auto-increment)
- Name (Required, Max 100 chars, Unique)
- CreatedAt (Default: CURRENT_TIMESTAMP)
- LastModifiedAt (Nullable)
```

ApiIntegrationConnections Table:

```
vbnet
CopyEdit
- Id (Primary Key, Auto-increment)
- ApiIntegrationId (Foreign Key to ApiIntegrations)
- ApiEndpointId (Foreign Key to ApiEndpoints)
- SequenceNumber (Required, Unique within integration)
```

People Table:

```
diff
CopyEdit
- Id (Primary Key, Auto-increment)
- Name (Required, Max 100 chars)
- Email (Required, Max 255 chars)
- CreatedAt (Default: CURRENT_TIMESTAMP)
```

Products Table:

```
pgsql
CopyEdit
- Id (Primary Key, Auto-increment)
- Name (Required, Max 200 chars)
- Price (Decimal 18,2)
- PersonId (Foreign Key to People)
- CreatedAt (Default: CURRENT_TIMESTAMP)
```

2. Relationships

- **One-to-Many**: ApiIntegration → ApiIntegrationConnections
- **Many-to-One**: ApiIntegrationConnections → ApiEndpoint
- **One-to-Many**: Person → Products (Cascade Delete)
- **Many-to-One**: Products → Person

3. Constraints and Indexes

- Unique constraint on `User.Username`
- Unique constraint on `ApiIntegration.Name`
- Unique constraint on `(ApiIntegrationId, SequenceNumber)` in `ApiIntegrationConnections`

- Foreign key constraints with appropriate delete behaviors
- Indexes on frequently queried fields

4. Data Seeding

The system includes seed data for:

- Sample API endpoints (People and Products APIs)
- Default admin user
- Sample people and products for testing

Frontend Design

1. Component Architecture

Page Components:

- `Dashboard.tsx`: Main application dashboard
- `Login.tsx`: User authentication
- `Register.tsx`: User registration

Feature Components:

- `ApiIntegrations.tsx`: Integration management and execution
- `ApiIntegrationBuilder.tsx`: Integration creation and editing

2. State Management

- React hooks for local state management
- Context API for global state (authentication)
- Axios interceptors for automatic token handling

3. UI/UX Design

Design System:

- Material-UI components for consistency
- Tailwind CSS for custom styling
- Responsive design for mobile compatibility
- Modern, clean interface with intuitive navigation

User Experience:

- Real-time API response display
- Execution time measurement
- Status code visualization
- Error handling with user-friendly messages
- Loading states and progress indicators

4. API Integration

Service Layer:

- Centralized API service functions
- Automatic token injection
- Error handling and retry logic
- Response transformation

Important Design Choices

1. API Chaining Architecture

Why JSON-based mapping files?

- Flexibility: Easy to modify without code changes
- Maintainability: Clear separation of configuration from logic
- Extensibility: New endpoints can be added without deployment

Why context-based parameter passing?

- Dynamic data flow: Responses from one API can feed into another
- Reusability: Same integration can work with different initial parameters
- Flexibility: Complex workflows with conditional logic

2. Database Design Decisions

Why separate `ApiIntegrationConnections` table?

- Many-to-many relationship between integrations and endpoints
- Sequence ordering support
- Future extensibility for additional connection metadata

Why unique constraint on integration name?

- Prevents confusion in UI
- Enables easy identification of integrations
- Maintains data integrity

3. Security Considerations

JWT Authentication:

- Stateless authentication suitable for microservices
- Automatic token refresh handling
- Secure password hashing

Input Validation:

- Server-side validation for all inputs
- SQL injection prevention through Entity Framework
- XSS prevention through proper output encoding

4. Performance Optimizations

Database:

- Proper indexing on frequently queried fields
- Lazy loading for navigation properties
- Efficient query patterns

API Execution:

- Asynchronous HTTP calls
- Connection pooling
- Response streaming for large payloads

5. Containerization Strategy

Multi-stage Docker builds:

- Optimized image sizes
- Security through minimal attack surface
- Consistent deployment across environments

Docker Compose orchestration:

- Easy local development setup
- Isolated network for services
- Persistent data volumes

6. Error Handling

Comprehensive error handling:

- HTTP status code mapping
- User-friendly error messages
- Detailed logging for debugging
- Graceful degradation

7. Scalability Considerations

Horizontal scaling ready:

- Stateless API design
- Database connection pooling
- Load balancer friendly

Future enhancements:

- Caching layer (Redis)
- Message queues for async processing
- Microservices architecture

Configuration Management

1. Environment-based Configuration

- Development vs Production settings
- Database connection strings
- JWT configuration
- API base URLs

2. Mapping File Management

- JSON configuration files in `Data` directory
- Runtime loading and validation
- Error handling for missing/invalid files

Testing Strategy

1. Backend Testing

- Unit tests for business logic
- Integration tests for API endpoints
- Database migration testing

2. Frontend Testing

- Component unit tests
- Integration tests for user flows
- API mocking for isolated testing

Deployment Considerations

1. Production Deployment

- Environment-specific configurations
- Database migration strategy
- Health check endpoints
- Monitoring and logging

2. CI/CD Pipeline

- Automated testing
- Docker image building
- Database migration automation