FlowAPI: Visual API Builder Platform

Amr Badran
Asem Diab

Supervisor: Dr. Emad Nastsheh

2026 Jan, 20

Contents

List of Figures

Abstract

FlowAPI is a visual API development platform that enables users to create, configure, and deploy RESTful APIs without writing code. The platform provides a drag-and-drop workflow builder that allows developers to construct complex API endpoints by combining pre-built functions for database operations, data manipulation, validation, and business logic.

The platform is a multi-tenant architecture where each application has its own isolated database, ensuring complete data separation and security. A visual workflow builder enables users to create endpoint logic through an intuitive drag-and-drop interface, while there is database page allow users to design and manage their database without direct database access. Also, An AI-powered assistant helps users generate workflows through natural language descriptions, making the platform accessible to non-technical users.

FlowAPI also includes a marketplace for sharing and buying APIs, team collaboration features with role-based access control, and analytics for monitoring API usage. Built on a microservices architecture with ASP.NET Core, Node.js/Express, Python/FastAPI, and React, FlowAPI demonstrates how visual development tools can enhance API creation, making it accessible to both technical and non-technical users while maintaining the flexibility needed for complex enterprise applications.

The platform addresses critical gaps in the API development landscape by offering a unified solution that combines visual workflow building, database management, AI assistance, and collaborative features. By bridging the divide between complex development frameworks and limited low-code solutions, FlowAPI has the potential to accelerate API development, reduce costs, and enable organizations to respond more quickly to changing business requirements.

Chapter 1

Introduction

1.1 Background

API development has traditionally been a complex process required deep programming knowledge, long setup, and large time investment. , the fundamental requirement of writing code still a barrier for many users, . This limitation becomes particularly evident when organizations need to rapidly prototype APIs, integrate systems, or enable non-technical team members to contribute to API development.

The current landscape of API development tools is fragmented. Solutions range from low-code platforms that offer limited flexibility to development frameworks that require huge technical expertise. Many existing tools focus on specific aspects such as API documentation, testing, or gateway management—but few provide a comprehensive solution that combines visual development, database management, workflow execution, and deployment in a single platform.

Furthermore, the growing demand for APIs in modern software development, combined with the reduced number of skilled developers, creates a need for tools that can make API creation easy. Organizations increasingly require APIs for internal integrations, third-party services, mobile applications, and web platforms, yet the development process remains time-consuming and resource-intensive. This gap between demand and accessibility is where FlowAPI aims to make a significant impact.

1.2 Terminology

To ensure clarity throughout this report, key terms and concepts are defined below:

1.2.1 Core Concepts

An **API Endpoint** is a specific URL path and HTTP method combination that defines a point of interaction in an API. For example, `GET /users` retrieves a list of users, while `POST /orders` creates a new order. Endpoints serve as the building blocks of RESTful APIs, each handling a specific operation.

A **Function Stack** is a sequence of pre-built functions that execute sequentially to implement the business logic of an API endpoint. Each function performs a specific op-

eration such as database queries, data validation, or variable manipulation. Functions share a common execution context, allowing data to flow from one function to the next through variables and placeholders.

A **Visual Workflow Builder** is a drag-and-drop interface that allows users to create API endpoint logic by combining functions visually, without writing code. Users can add functions, configure their parameters, reorder them, and see the workflow take shape visually, similar to flowchart creation tools but specifically designed for API logic.

Multi-Tenancy is an architectural pattern where a single application instance serves multiple customers (tenants), with each tenant's data completely isolated from others. In FlowAPI, each application (project) acts as a tenant, with its own isolated database and resources, ensuring that data from one application cannot be accessed by another.

Placeholder Resolution is a system that dynamically replaces template variables (e.g., `{{user.id}}`) with actual values at runtime, enabling dynamic data flow between functions. This system allows functions to reference data from previous functions, request parameters, or computed values, creating a flexible data flow mechanism.

1.2.2 Technology Stack

Microservices Architecture is an architectural style where an application is composed of small, independent services that communicate over well-defined APIs. Each service handles a specific concern, such as user management, database operations, or workflow execution. This approach enables independent development, deployment, and scaling of services.

Clean Architecture is a software design philosophy that emphasizes separation of concerns, with clear boundaries between business logic, data access, and presentation layers. The architecture ensures that business logic is independent of frameworks and external concerns, making the system more maintainable and testable.

CQRS (Command Query Responsibility Segregation) is a pattern that separates read and write operations, optimizing each for their specific use case. Commands handle mutations and return results, while queries handle data retrieval without side effects. This separation allows for independent optimization of read and write paths.

NoSQL Database refers to database management systems that store data in formats other than relational tables. FlowAPI uses MongoDB, which stores data as documents (JSON-like structures), provide flexibility for dynamic schemas and nested data structures.

1.2.3 Development Concepts

No-Code/Low-Code Development is an approach that enables users to create applications through visual interfaces and configuration rather than traditional programming. This paradigm has gained large interest as organizations hope to reduce development time

and enable non-technical users to contribute to application development.

Drag-and-Drop Interface is a user interface paradigm, where users can manipulate elements by clicking, dragging, and dropping them to different locations or configurations. This interaction model is intuitive and reduces the learning curve for new users.

Visual Programming is a programming paradigm that uses visual elements (blocks, nodes, diagrams) instead of text-based code to represent program logic. While visual programming has been explored for decades, modern implementations leverage advanced UI frameworks for provide smooth, responsive interfaces.

Workflow Engine is a system that executes a sequence of operations (workflow) defined by users, handling state management, error handling, and data flow between steps. Workflow engines abstract away the complexity of orchestrating multiple operations, allowing users to focus on defining what should happen rather than how to implement it.

1.3 Objectives

The primary objectives of this work are to analyze existing API development tools and identify gaps in their ability to provide comprehensive, user-friendly API creation capabilities that combine visual development, database management, and workflow execution. Through this analysis, we aim to understand the limitations of current solutions and design a platform that addresses these gaps in good manner.

The main objective is to design and develop FlowAPI, a visual API builder platform that enables users to create, configure, and deploy RESTful APIs through an intuitive drag-and-drop interface, eliminating the need for traditional coding. The platform should provide good database management tools that allow users to create tables, define schemas, manage records, and establish relationships between data entities without direct database access.

Another key objective is to implement a flexible workflow execution engine that supports sequential function execution, conditional logic, loops, and dynamic data manipulation, enabling complex business logic without code. This engine should handle request processing, function execution, placeholder resolution, and response generation seamlessly.

The project also aims to create an AI-powered assistant that helps users generate API workflows through natural language descriptions, making the platform accessible to non-technical users. This assistant should understand user requirements, query available functions and database schema, and build appropriate function stacks automatically.

Furthermore, the system should enable multi-tenant architecture with complete data isolation, allowing multiple users and organizations to use the platform simultaneously while maintaining security and privacy. Each application should have its own isolated database and resources, ensuring that data from one application cannot be accessed by another.

The platform should provide marketplace functionality for sharing, discovering, and buying APIs, introduce a collaborative ecosystem around API development. Users should be able to list their APIs, set pricing, and enable others to purchase and use their APIs.

Finally, the system should implement team collaboration features that enable multiple developers to work together on API projects, with role-based access control and invitation system. This will allow teams to collaborate effectively, while maintaining proper access control and security.

1.4 Summary

FlowAPI addresses critical gaps in the API development landscape by offering a unified platform that combines visual workflow building, database management, and AI assistance. Its emphasis on no-code development, multi-tenancy, and collaborative features makes it accessible to both technical and non-technical users, democratizing the creation of RESTful APIs.

The platform's visual workflow builder enables users to create complex API logic through an intuitive drag-and-drop interface, while the comprehensive database management tools allow users to design and manage their data structures without direct database access. The AI-powered assistant further lowers the barrier to entry by generating workflows from natural language descriptions.

By bridging the divide between complex development frameworks and limited low-code solutions, FlowAPI has the potential to accelerate API development, reduce costs, and enable organizations to respond more quickly to changing business requirements. The platform's multi-tenant architecture ensures scalability and security, while the marketplace and collaboration features foster a vibrant ecosystem around API development.

The system's applicability spans across industries such as e-commerce, healthcare, education, and enterprise software, where quick API development and integration are crucial for business success. FlowAPI is a significant step forward in making API development more accessible, efficient, and collaborative.

Chapter 2

Literature Review

2.1 Introduction

From straightforward service endpoints to complex, distributed systems that drive now applications, API development has progressed. Nevertheless, the process of developing APIs is still primarily code-centric and necessitates a high level of programming proficiency. In order to determine the gaps that FlowAPI seeks to fill, this chapter examines current low-code solutions, workflow automation tools, and API development platforms.

2.2 Theoretical Foundations

2.2.1 Visual Programming and No-Code Development

Visual programming paradigms have been explored for decades, with the goal of making software development more accessible [1]. The idea is to use visual components, like blocks, nodes, or flowcharts, to represent program logic instead of text-based code. Visual programming can lower the learning curve for non-programmers while preserving the expressiveness required for complex applications, according to research.

The increasing use of low-code and no-code platforms in enterprise software development has been noted in recent surveys [2]. Users can concentrate on business logic and user experience instead of technical implementation thanks to these platforms, which abstract away implementation details.

This idea is expanded by no-code and low-code platforms, which offer pre-made parts and templates that users can mix and match to build applications. Users can concentrate on business logic and user experience instead of technical implementation thanks to these platforms, which abstract away implementation details. The effectiveness of this strategy for some kinds of applications is shown by the success of platforms like Microsoft Power Automate [3] and Zapier [4].

2.2.2 Process Automation and Workflow Engines

Workflow engines handle data flow, error handling, and state management by carrying out user-defined sequences of operations. Conventional workflow engines, like those found, concentrate on managing processes and coordinating services. In order to facilitate data transformation, API orchestration, and service integration, modern workflow engines have developed.

The main difficulty in workflow design is making a balance between usability and flexibility. While too little flexibility can restrict the platform's applicability to a variety of use cases, too much flexibility can overwhelm users. In order to overcome this difficulty, FlowAPI offers an extensive function library that covers typical use cases while retaining the adaptability to manage complicated situations.

2.2.3 Multi-Tenant Architecture

A single application instance serves several clients under the multi-tenancy architectural pattern, with each client's data kept separate from the others [5]. Because it allows for effective resource utilization while preserving security and privacy, this pattern is crucial for Software-as-a-Service (SaaS) platforms.

Multi-tenancy can be approached in a number of ways, each with unique trade-offs. The strongest isolation is achieved with database-level isolation, which requires more resources because each tenant has its own database. Efficiency and isolation are balanced by schema-level isolation, in which tenants share a database but have different schemas. Although it requires careful access control, row-level isolation, in which all tenants share the same schema and are identified by a tenant ID column, offers the most effective resource usage. For the best security and isolation guarantees, FlowAPI employs database-level isolation.

2.3 Existing Solutions and Their Approaches

2.3.1 API Gateway and Management Platforms

Platforms like Kong [6], Apigee [7], and AWS API Gateway [8] provide powerful API management capabilities, including routing, authentication, rate limiting, and analytics. However, these platforms focus on managing and securing existing APIs rather than creating new ones. They assume that APIs are already developed and deployed, requiring users to write code separately.

While these platforms excel at API lifecycle management, they do not address the initial API development process, leaving users to rely on traditional development tools and frameworks. This gap creates an opportunity for platforms like FlowAPI that focus on API creation rather than just management.

2.3.2 Low-Code API Development Platforms

Platforms such as Backendless [9], and Firebase [10] offer low-code approaches to API development. These platforms provide visual interfaces for creating APIs, often with integrated database management and authentication features. However, these solutions often have limitations in flexibility, vendor lock-in with proprietary data formats, restricted ability to handle complex workflows, and limited support for advanced data manipulation and transformation.

2.3.3 Workflow Automation Platforms

Tools like Zapier [4], n8n [11], and Microsoft Power Automate [3] excel at connecting services and automating workflows. They provide visual interfaces for creating automation flows, with extensive integrations to third-party services. However, these platforms are primarily designed for automation and integration rather than API development.

These platforms focus on actions triggered by events. They do not serve HTTP requests, which limits their ability to create RESTful APIs for other applications. They often need external services for data storage and management. This makes them less suitable for building custom business logic that requires sequential execution with shared context.

2.3.4 Backend-as-a-Service (BaaS) Platforms

BaaS platforms like Parse, Supabase and Appwrite give us backend infrastructure and APIs that're ready to use. These include things like authentication and database access and file storage. This really helps because it saves us a lot of time when we are setting up backend services.

Baas platforms have some problems. For example the API structures are already. That may not work for every situation. We also cannot customize the business logic much as we want because we are limited to what the platform gives us. Sometimes we have to write our code to do complicated things.. We do not have as much control over how the APIs are designed and how they behave. BaaS platforms, like Parse, Supabase and Appwrite are still really useful. They have these limitations.

2.3.5 API Design and Documentation Tools

Tools like Postman, Swagger or OpenAPI and Insomnia are really helpful for developers. They use these tools to design and test and document Application Programming Interfaces. These tools are very useful for people who make APIs. They make it easier for developers to do their job.. People who use these tools still have to write code to make the API work. This is a problem for people who do not know how to code. These tools are great. They do not get rid of the need to write code. So people who are not good at coding still have a time making APIs. Tools like Postman and Swagger or OpenAPI and

Insomnia are good, at helping with some parts of making an API.

2.4 Identified Gaps in Existing Solutions

The review of existing solutions reveals several key gaps in the current landscape of API development tools. While some tools offer visual interfaces, they often lack the flexibility to handle complex workflows, conditional logic, loops, and advanced data manipulation. FlowAPI addresses this by providing a well-designed function stack system that supports sequential execution, nested functions, and dynamic data flow.

Different platforms usually require you to juggle between database management and API building, sending you to a different tool or interface, when developing an API. Well-known platform FlowAPI has now streamlined this process by merging these two functions, so that users can create tables, update records, and build APIs all in one place. Coming rushing between different systems is a thing of the past, and the result is a much smoother development experience.

While AI is rapidly being integrated into code editors, very few platforms offer AI-driven workflow generation for API building, and FlowAPI is one of the rare ones that does. Users can now just tell the AI assistant what they want their API to do, and it generates the workflow for them, making it accessible to anyone who can write in natural language, even non-technical people.

When it comes to databases, a lot of low-code platforms use shared databases and row-level isolation which can raise serious security concerns. FlowAPI addresses these problems by implementing database-level isolation, where each application gets its own MongoDB database, giving complete data separation and a huge boost in security. This setup also uses connection pooling and caching to keep the system running efficiently.

As for collaboration and sharing, some platforms are pretty good, but don't really offer anywhere to buy and sell APIs. FlowAPI's got this sorted out with a marketplace feature that lets users list, find and purchase APIs. This creates a sharing economy, where people can monetize their work and get access to pre-built solutions

Many platforms only offer a small set of built-in functions. FlowAPI is different because it provides a wide range of functions for database operations, data handling, validation, control flow, and security. Users can also extend these features using a function stack system. Because of this, users can build complex APIs without writing any code.

2.5 Proposed Approach: FlowAPI

FlowAPI solves these limitations by introducing a new way to build APIs. It combines visual workflow design, built-in database management, AI assistance, and collaboration tools in one platform. Users create API logic by connecting pre-built functions in a visual editor. Each function performs a single task, and the functions run step by step while

sharing the same execution data. This approach gives users the flexibility of coding while keeping the process simple and visual.

FlowAPI also includes full database management features. Users can create tables, define data structures, manage records, and set relationships between tables directly inside the platform. It supports different types of primary keys, reference fields, and CSV import and export. This removes the need to switch between multiple tools and makes development faster and easier.

The platform includes an AI assistant that can build workflows from natural language descriptions. Users can describe what they want, and the assistant selects the right functions and database elements to create the workflow. This makes API development possible even for users with little or no technical background.

FlowAPI uses a multi-tenant architecture with complete database separation. Each application has its own isolated MongoDB database. This design improves security and privacy while still allowing efficient use of system resources through connection pooling and caching.

FlowAPI also offers a marketplace where users can share, discover, and sell APIs. This encourages collaboration and reuse, allowing users to customize APIs using existing components or publish their own. The marketplace also supports motivating users to build high-quality APIs.

The platform includes a placeholder system that allows data to flow dynamically between functions. It supports variables, request data, nested fields, and expressions. This makes it possible to handle complex logic while keeping the visual workflow easy to understand.

Overall, FlowAPI provides a strong library of functions for database operations, data processing, control flow, validation, and security. This allows users to build powerful APIs without writing code, while still supporting a wide range of use cases.

2.6 Conclusion

This review show existing API tools, workflow platforms, and low-code solutions. While many tools offer useful features, none combine visual workflows, database management, AI support, multi-tenant design, and team collabration in a single system. FlowAPI fill this gap by bringing these features together, making API development simpler, more flexible, and accessible to a wider range of users.

Chapter 3

Methodology

3.1 System Design and Architecture

3.1.1 System Overview

FlowAPI is designed using microservices architecture. This means the system is divided into several small, independent services, and each service is responsible for a specefic task. This design makes the system easier to develop, maintain, and scale. Each service can be deployed independently, and different technologies can be used where appropriate.

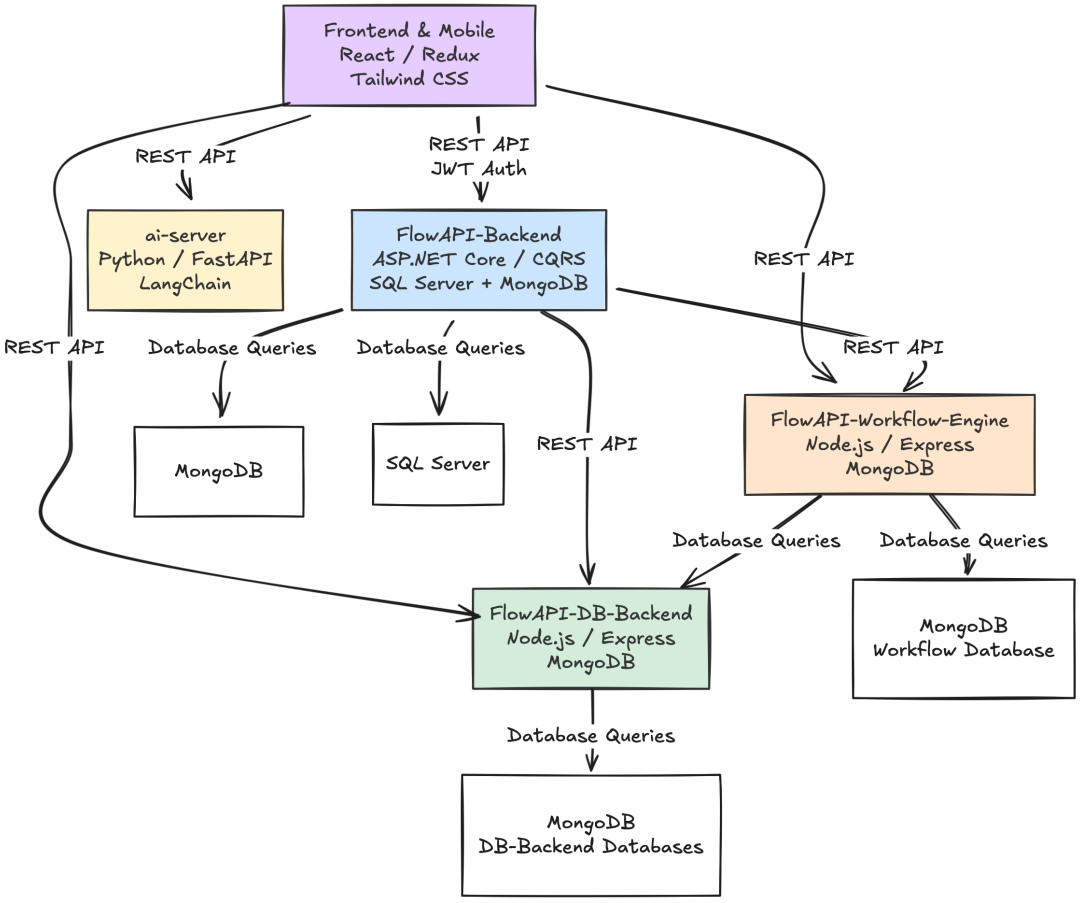


Figure 3.1: Microservices architecture of FlowAPI

The system consists of five main components. The **FlowAPI-Backend** is the core service was built using ASP.NET Core. It handles user authentication, application creation, API endpoint configuration, marketplace features, and team collaboration. it adhers to Clean Architecture principles and uses the CQRS pattern with MediatR, which helps keep the code organized and easy to maintain.

The **FlowAPI-DB-Backend** is a Node.js service that handles database operations. It manages table creation, schema definitions, and CRUD operations. Each application has its own MongoDB database for complete data separation.

The **FlowAPI-Workflow-Engine** is another Node.js service that executes all API requests in the system. When a request is received, the engine finds the matching endpoint, runs its function stack one after another, and returns a response. It provide features like placeholders, conditions, loops, and database access, allowing users to build complex logic without writing code.

The **FlowAPI-Frontend** is a React-based single-page application that shows the user interface. It includes ui for building workflows, managing databases, browsing the marketplace, and make collabration with team members. Redux Toolkit is used for state management, and Tailwind CSS is used for styling.

The **AI-Server** is built using Python. provides AI-powered workflow generation by converting natural language descriptions into function stack. The service uses LangChain to understand user requests and follows a multi-step process to select functions and construct workflows.

3.1.2 Communication Patterns

The microservices communicate using RESTful APIs and store data in MongoDB and SQL-Server. This design keeps services loosely connected while allowing them to work together efficiently. The frontend communicates with backend services through REST APIs and uses JWT tokens for user authentication and API keys for executing workflows.

During workflow execution, the Workflow Engine communicates with the DB-Backend to perform database operations. This separation ensures that database access is handled in abstracted way.

The frontend also sends natural language requests to the AI-Server. The AI-Server processes these requests, generates workflows, and returns json document which is an endpoint. Before executing any workflow, the Workflow Engine verifies API keys and retrieves endpoint information from the main backend to ensure proper secuirty.

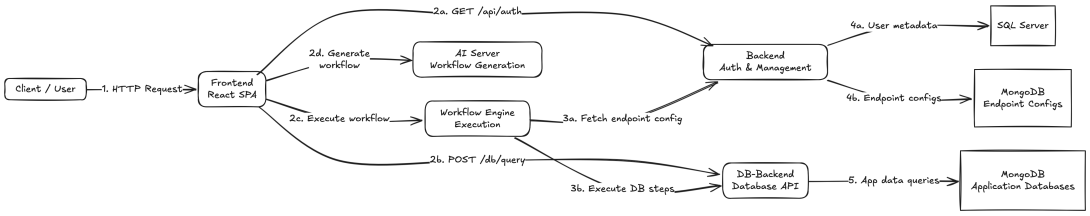


Figure 3.2: Communication flow between microservices

3.1.3 Data Flow

Data flows through the system in a clear way. When a user registers, the frontend sends the request to the Backend service, which stores user information and generates JWT tokens.

When a user creates a new application, the Backend service stores application information in a SQL Server database and creates a dedicated MongoDB database for that application.

Database tables are created through the frontend and handled by the DB-Backend service. Endpoint configurations are defined using the workflow builder. Endpoint metadata is stored in SQL Server, while function stacks and dynamic configurations are stored in MongoDB.

When an API request is received, the Workflow Engine matches it to the correct endpoint, executes the function stack, performs database operations through the DB-Backend service, and returns the final response.

The AI feature allows users to describe their API needs in their language. The AI-Server analyzes the request, checks available functions and database schemas, and generates a workflow that is applied to the endpoint.

3.1.4 Multi-Tenancy Architecture

FlowAPI uses a multi-tenant architecture with full database isolation. Each application has its own MongoDB database, its name is unique application ID. This ensures that data from one application cannot be accessed by another.

Each application is assigned a unique API key that is required during workflow execution. The Workflow Engine validates this key before processing requests, ensuring that only authorized applications can access their APIs.

Team collaboration is supported through role-based access control. The system defines two roles: Owner and Member. Owners have full control over the application, while Members have limited permissions.

3.1.5 Security Architecture

Security is implemented at multiple levels throughout the system. User authentication is handled using JWT tokens, which include user information and authorization claims. These tokens are checked on every protected request, making the system to remain stateless.

Workflow execution requires valid API keys, which are verified before any logic is executed. Endpoints can also be configured to require JWT authentication, allowing both public and protected APIs. The Workflow Engine validates tokens using application-specific secret keys and adds user data to the execution context when required.

3.1.6 Deployment Architecture

FlowAPI is deployed using Docker containers, which ensure consistent behavior across development, testing, and production environments. Each service includes a Dockerfile that defines its runtime environment and dependencies.

Docker Compose is used during development to run all services together on a local machine. It make it simple setup by defining services, networks, and configurations in a single file.

The Backend service supports automatic database migrations on startup. This ensures that the database schema remains consistent and up to date.

3.2 FlowAPI Backend

3.2.1 Overview

The FlowAPI Backend is the main API and was built with ASP.NET Core, being the central seervice for the platform. It handles user authentication, application management, endpoint configuration, marketplace functionality, and team collaboration. The service adhers to Clean Architecture principles with CQRS (Command Query Responsibility Segregation) pattern and uses MediatR for request handling.

The service acts as the primary interface for the frontend, providing RESTful APIs for all platform operations. It manages user accounts, applications, endpoints, API groups, marketplace listings, and team collaborations.

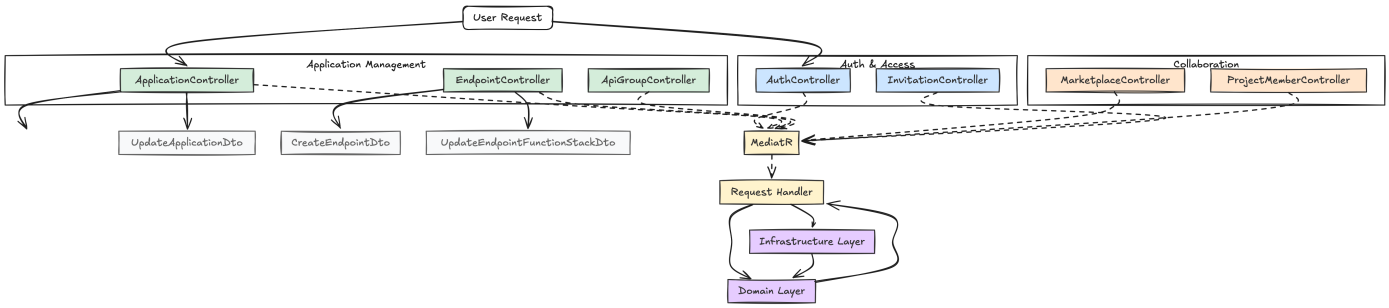


Figure 3.3: API Schema for FlowAPI Backend showing endpoints and relationships

3.2.2 Architecture Pattern: Clean Architecture

The Backend service is designed using the Clean Architecture pattern. This approach separates the system into clear layers, each with a responsibility. The main goal is to keep the core business logic independent from frameworks and external tools.

The architecture is divided into four main layers.

The **Domain Layer** contains the core entities such as User, Application, Endpoint, ApiGroup, MarketplaceApplication, ProjectMember, and Invitation. These entities represent the main business concepts and include the rules and behaviors related to them. This layer also defines interfaces for repositories and services, but it does not depend on any external frameworks. Because of this, the domain layer remains independent.

The **Application Layer** follows the CQRS pattern. Commands are used for operations that modify data, such as creating applications, updating endpoints, or sending invitations. Queries are used for reading data, such as retrieving application details or listing marketplace items. MediatR is used to handle requests and responses, which helps separate business logic from controllers and supports features such validation and logging.

The **Infrastructure Layer** provides implementations for the interfaces defined in the Domain Layer. It handles data storage using Entity Framework Core with SQL Server and MongoDB. This layer handles external services such as JWT token generation, email delivery, and file storage.

The **Presentation Layer** includes REST API controllers, Data Transfer Objects (DTOs), FluentValidation validators, and middleware. Controllers receive HTTP requests and convert them into Commands or Queries, which are then sent through MediatR. DTOs define a clear contract between the API and its clients, while validators ensure that incoming data is correct before processing.

3.2.3 Domain Model

The Backend service uses a well-defined domain model focused on applications and API endpoints. This model shows the key concepts of the system and the relationships between them.

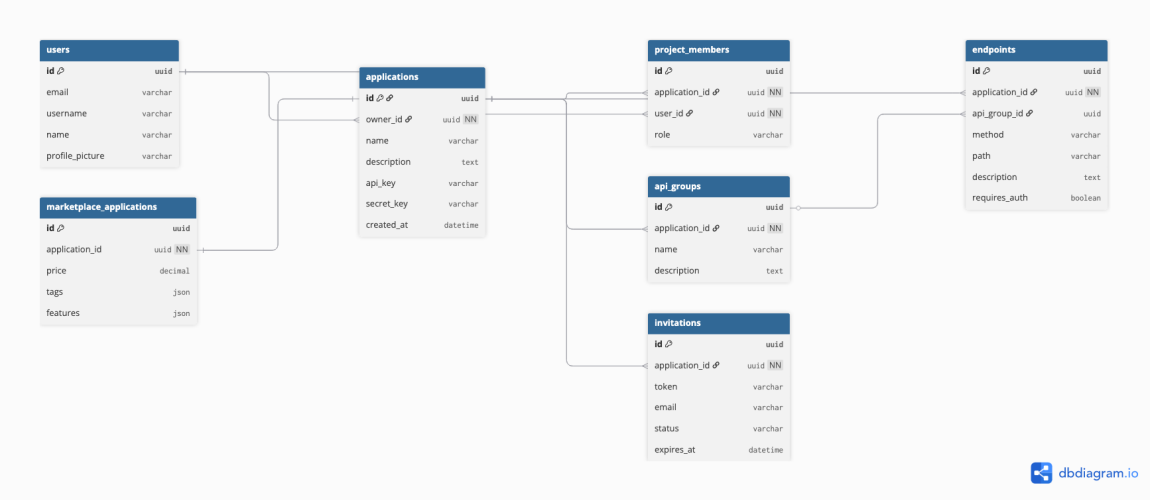


Figure 3.4: Core Domain Model Class Diagram

The **User** entity manages user accounts, authentication, and profile information. It supports secure login using ASP.NET Core Identity, including password hashing and email verification. Users can own applications and make collabration with others. The entity stores basic profile details such as name, email, username, and profile picture.

The **Application** entity represents a project created by a user. Each application has a unique API key and secret key used during workflow execution. Every application is assigned its own MongoDB database, making sure for data isolation. Applications can also be listed in the marketplace. Metadata such as name, description, creation date, and owner information is stored with the application.

The **ApiGroup** entity is used to organize endpoints into logical groups. Each group belongs to an application and can consist of multiple endpoints.

The **Endpoint** entity represents an API endpoint defined by the user. Basic information such as HTTP method, URL path, description, and authentication requirements is stored in SQL Server. More flexible data, such as function stacks and config, is stored in MongoDB. This hybrid approach provides both structure and flexibility.

The **MarketplaceApplication** entity represents applications listed in the marketplace. It stores pricing details, tags, features, and purchase information. Social features such as likes and favorites help users discover popular APIs. After purchasing an application, buyers receive an API key that allows them to use the API.

The **ProjectMember** entity supports team collaboration by linking users to applications with role member. The system supports two roles: Owner and Member. Owners have full permissions, while Members have limited.

The **Invitation** entity manages team invitations. It handles token creation, email sending, and invitation status. When an invitation is accepted, the system creates a ProjectMember record and grants access to the application.

3.2.4 CQRS Pattern Implementation

The Backend service uses the CQRS pattern to separate read and write operations. Commands are responsible for modifiying data, while Queries are used only for retrieving information.

Examples of Commands include:

- `CreateApplicationCommand`: Creates a new application and generates API keys
- `UpdateApplicationCommand`: Updates application information
- `CreateEndpointCommand`: Creates a new API endpoint
- `UpdateEndpointFunctionStackCommand`: Updates an endpoint's function stack
- `SendInvitationCommand`: Sends collaboration invitations

Queries are used for reading data, such as:

- `GetApplicationByIdQuery`: Retrieves application details
- `GetUserApplicationsQuery`: Lists applications owned by a user
- `GetEndpointsByApiGroupQuery`: Retrieves endpoints in a group
- `GetMarketplaceListingsQuery`: Lists marketplace applications

.

3.2.5 MediatR Integration

MediatR is used as a central communication way inside the Backend service. Controllers send Commands or Queries through MediatR, which routes them to the correct handlers. This make controllers simple and focused on handling HTTP requests.

MediatR also supports pipeline that handle common tasks such as logging, validation, and error handling. Because handlers are independent of HTTP context, they are easy to test in isolation. This design reduces boilerplate code and improves maintainability.

3.2.6 Database Strategy

The Backend service uses both SQL Server and MongoDB to take advantage of their strengths.

SQL Server is used for structured data that requires strong consistency, such as:

- User accounts and authentication data
- Application metadata
- API groups
- Marketplace listings

- Team members and invitations

MongoDB is used for flexible data structures, including:

- Endpoint configurations and function stacks
- Request and response templates
- Dynamic endpoint data
- Application-specific databases for isolation

This hybrid approach allows the system to remain both structured and flexible.

3.2.7 Authentication and Authorization

The system uses JWT for authentication to control access to the platform. Tokens are generated during login and included in requests to protected endpoints.

3.2.8 API Versioning

The Backend service uses URL versioning to manage API changes. Each request must include the version number in the URL, such as `/api/v1.0/`. This make multiple versions of the API to exist at the same time and supports backward compatibility. Swagger documentation is generated for each version.

3.2.9 Marketplace System

The marketplace makes users to publish, discover, and sell APIs. Users can manage listings with pricing, descriptions, tags, and features. Payments are made through Stripe, which processes transactions and delivers API keys to buyers.

Search, filtering, likes, and favorites help users find relevant APIs. Purchased APIs can be used immediately using the provided API key.

3.2.10 Team Collaboration

Team collaboration is supported via email invitations. Owners can invite users to collaborate, and invited users can accept the invitation through a secure link. Owners have full control, while Members can collaborate without access to critical actions like deleting the project.

3.2.11 Error Handling

The Backend service includes a global error handling system to make sure for consistent responses. All unhandled exceptions are logged and converted into user-friendly error messages. Validation errors return clear field-level messages using FluentValidation Library.

Also logging with Serilog helps with monitoring and debugging, while custom domain exceptions clearly represent business rule violations.

3.3 FlowAPI DB Backend

3.3.1 Overview

The FlowAPI DB Backend is a Node.js service which is responsible for handling all database operations in the platform. It provides REST APIs for creating and managing tables and records inside MongoDB. Each application is given its own MongoDB database, which makes sure complete data separation between different projects inside same cluster.

This service acts as a dedicated data layer for FlowAPI. It hides the complexity of multi-tenant database management and exposes simple APIs for working with tables and records.

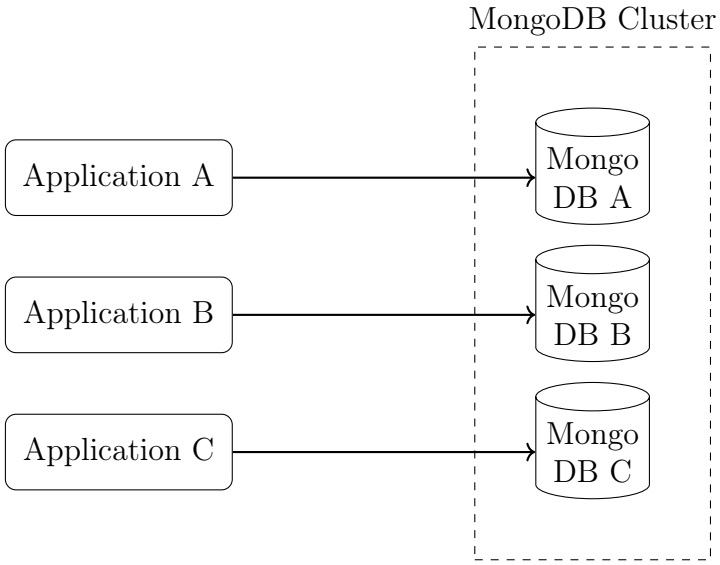


Figure 3.5: Multi-tenant database architecture with isolated application databases

3.3.2 Multi-Tenant Architecture

The DB Backend uses a strong multi-tenant design based on database-level isolation. Each application has its own MongoDB database, identified by the application ID. This make sure that data from one application cannot be accessed by another.

The service manages database connections using connection cache. When a request arrives, the service checks whether a connection for that application already exists. If it

does, the connection is reused to improve performance. If not, a new connection is created and stored for future use. This approach balances performance and resource usage while keeping data isolated.

Mongoose connection pooling is used to manage multiple requests efficiently. Instead of opening a new connection for each request, multiple requests can share a pool of existing connections.

Databases are created only when needed. When a user creates the first table for an application, the system automatically creates the database and the required collections. This lazy creation approach avoids unnecessary setup and keeps the system efficient.

3.3.3 Database Schema Design

The DB Backend uses a flexible schema designed for multi-tenant and dynamic data. It relies on two main collections: `Tables` and `Data`. The Tables collection stores table definitions and metadata, while the Data collection stores the actual records.

The Tables collection contains information such as table name, primary key type, and column definitions. Table names must be unique within an application. Column definitionns include data types and properties (meta data), which are used to validate records before they are stored.

Each record is stored as a plain JSON object. The primary key is included in every record, and its type depends on the table configuration. Records are validated against the table schema before being saved.

3.3.4 Primary Key Types

The system supports three types of primary keys to meet different needs.

The **ObjectId** type uses MongoDB built-in ObjectId, which is generated automatically and ensures uniqueness. This is the default and recommended option.

The **Sequential** type uses increasing numeric values such as 1, 2, and 3. This is useful when ordered IDs are required. The system tracks the last used value and increments it for new records.

The **UUID** type uses UUID version 4 strings. This type allows unique IDs to be generated independently and works well in distributed systems.

3.3.5 Column Type System

The DB Backend includes a two-level type system consisting of base types and subtypes. Base types define how data is stored, while subtypes add semantic meaning.

Base types include:

- `String`
- `Number`
- `Boolean`
- `Timestamp`
- `Date`

Subtypes add meaning to the data:

- `ID`
- `Text`
- `Reference`
- `Reference List`
- `UUID`
- `Sequential`

3.3.6 Reference Column System

The DB Backend provides relationships between tables using the reference columns. A reference column stores a single ObjectId pointing to a record in another table, supporting one-to-many relationships. Reference list columns store arrays of ObjectIds, enable many-to-many relationships.

The system validate reference formats and ensures that referenced tables exist. However, joining related data is handled by the Workflow Engine rather than the DB Backend.

3.3.7 API Endpoints

The DB Backend provides RESTful APIs for managing tables and records. All operations require `projectId` and `ownerID` to enforce access control and tenant isolation.

Table management endpoints include:

- `POST /DB/tables`
- `GET /DB/tables`
- `GET /DB/tables/:tableId`
- `PUT /DB/tables/:tableId`
- `DELETE /DB/tables/:tableId`

Record management endpoints support full CRUD operations, filtering, sorting, pagination, and CSV imports.

3.3.8 Data Validation and Type Conversion

All incoming data is being validated in front of table schemas. The system check required fields, data types, and reference formats. Type conversion automatically converts values such as string numbers, booleans, and dates into their correct formats.

Conversion functions handle transformations between display values used in APIs and stored values used in the database. This makes the API easier to use while maintaining data integrity.

3.3.9 CSV Import Feature

The DB Backend support bulk data import from CSV files. The system read CSV files, map columns, converts values, validates records, and inserts them in batches. Errors are reported in clear way so users can fix and retry failed rows.

3.3.10 Security Features

Security is being enforced on multiple levels. Owner and project make sure that users only access their own data. Each application uses a separate database, preventing wrong access.

3.3.11 Limitations and Considerations

The current design is storing records in arrays, which are limited by MongoDB's 16MB document size limit. This approach works well for small and medium tables but may not scale for huge datasets, Future improvements should include storing records as separate documents.

3.4 FlowAPI Workflow Engine

3.4.1 Overview

The FlowAPI Workflow Engine is a Node.js microservice that runs API endpoints created by users through visual workflows. Instead of writing code, users define a sequence of functions, called a function stack, which the engine executes at runtime. Each incoming HTTP request is matched to a saved endpoint configuration in MongoDB, and the associated workflow is executed step by step.

This engine acts as the core execution service of the platform. It turns user-defined workflows into real API behavior by handling request matching, function execution, and response generation. By managing placeholders, variables, conditions, loops, and database actions, the engine allow users to build advanced APIs without writing custom code.

3.4.2 Request Processing Flow

When a request comes to the Workflow Engine, it follow a clear process. First, the engine checks for a valid API key in the request headers or query parameters. This key identifies the application and make sure that only authorized requests are processed.

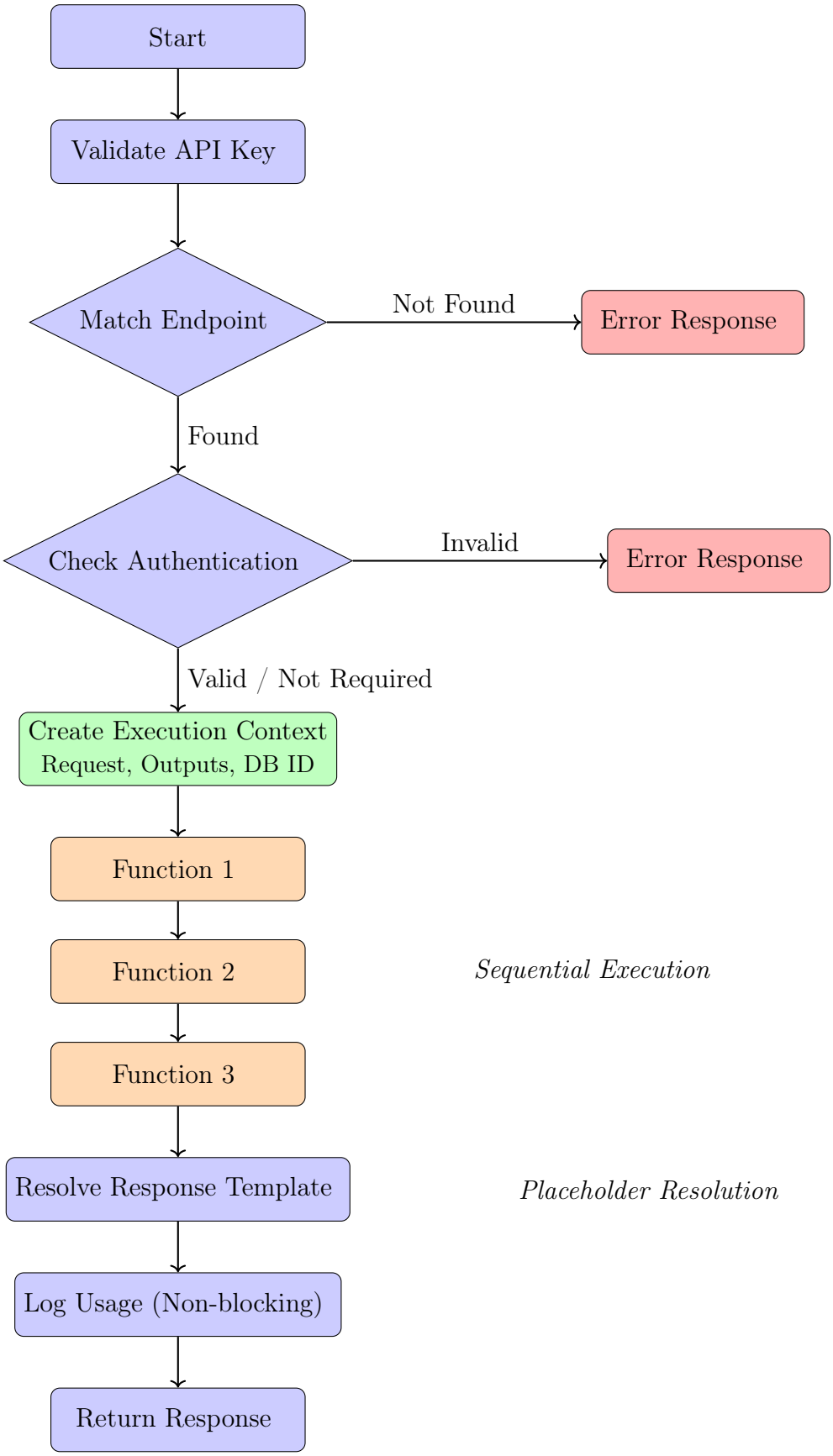
Next, the engine searches in MongoDB for an endpoint configuration that matches the request method and path. It supports both exact paths and dynamic paths with parameters. For example, the pattern `/users/:userId` matches requests like `/users/123` and extracts the `userId` value easily.

If the endpoint needs authentication, the engine validate the provided JWT token. This includes checking the token signature, expiration time, and extracting the user data. Once validated, the user information is added to the execution context so it can be used by functions later in the workflow.

After validation, the engine make a shared execution context. This context consist of request details, a shared output object, and the application's database identifier. All functions in the function stack use this same context, allowing data to flow from one function to the another.

Functions are executed one after another. Each function read from the context, performs its task, and stores results back into the context. If any function fails, execution stops immediately and an error response is returned. Once all functions finish successfully, the engine builds the final response by resolving placeholders and returns it to the client.

Figure 3.6: Workflow execution process from request to response



3.4.3 Function Stack System

function stack system is the heart of out system. This system allow users to build complex logic by combining small, reusable builiding blocks (functions) . Each function follows a standard structure and implements an `execute(config, context)` method.

All functions are registered in a registry in memory. while execution, the engine looks up each function by name and runs it in sequence. This design makes the system easy to extend, as new functions can be added without changing the core engine.

Every function in the stack get the same context object. This allows functions to share data easily, such as using the result of a database query in a later validation or response step. Some functions can also run nested workflows, enabling conditions and loops to be defined visually.

3.4.4 Function Categories

The Workflow Engine support a rich set of built-in functions grouped by purpose.

Database functions allow users to read and write data. Functions such as `getItem` and `getItems` retrieve records, while `insertItem`, `editItem`, and `deleteItem` manage data changes.

Data manipulation functions helping in managing variables and arrays. Users can set variables, create arrays, add items, or remove items without writing any code. These functions make it easy to transform and organize data during execution.

Control flow functions handle logic. The `ifElse` function runs different workflows based on conditions, while `forEach` repeats a workflow for each item in a list. These features allow users to build dynamic and flexible logic.

Validation functions check inputs and enforces rules. The `simpleValidation` function make sure required fields are present and correctly formatted, while `throwIf` stops execution when a condition is met.

Security functions support authentication and authorization features. The `createJwtToken` function generates signed JWT tokens, allowing APIs to implement login and authorization flows.

3.4.5 Placeholder Resolution System

The engine uses a very powerful placeholder system to move data between functions. Placeholders use the format `{{...}}` and are being resolved at runtime.

Placeholders can reference variables created by earlier functions from shared context object, request parameters, query values, or request body fields. For example, `{{request.params.userId}}` accesses a URL parameter, while `{{myResult}}` accesses a stored output value.

The system also provies nested access and simple expressions. This allows users to reference deeply nested data or perform basic calculations directly inside configurations. Placeholders are resolved recursively, ensuring all dynamic values are being replaced before execution continue.

3.4.6 Endpoint Matching

The Workflow Engine matches requests from the HTTP method and path. Exact matches are checked first. If no exact match is found, the engine attempts pattern matching using path parameters.

When a pattern matches, parameter values are extracted and added to the request object. These values are then available throughout the workflow, making endpoints flexible and reusable

3.4.7 Authentication System

Authentication can be enabled on endpoint. When required, the engine extracts the JWT token from the `Authorization` header and validates it using the application's secret key.

If validation succeeds, user data from the token is added to the execution context. This allows workflows to use user information for access control. Endpoints that don't require authentication can still fully public.

3.4.8 Response Generation

Responses are made using templates that might contain placeholders. These placeholders are resolved via values from the execution context

3.4.9 Error Handling

The engine provides clear and consistent error handling. Missing endpoints return a 404 error, while authentication problems return a 401 error with helpful messages.

If a function failed, the error message include the function name and reason for failure. Validation errors return detailed feedback about incorrect or missing fields. Errors follow a structured format, making them easy for clients to process.

3.4.10 Multi-Tenancy Support

The Workflow Engine fully support multi-tenant execution. Each request is linked to a specific application, and only that application endpoints and its database were used.

Endpoints are filtered by application during matching, and all database operations use the correct application database. This ensures strong isolation between applications and prevents cross-tenant data access at every stage of execution.

3.5 FlowAPI Frontend

3.5.1 Overview and Architecture

The FlowAPI Frontend is a React-based Single Page Application (SPA) that serves as the visual interface for the FlowAPI platform. It enables users to create, configure, and manage API endpoints, database tables, workflows, and marketplace listings through a visual workflow builder. The frontend is built using React 19, Redux Toolkit for state management, and Tailwind CSS for styling, providing a consistent and structured user interface.

The application follows a component-based architecture with clear separation of concerns. The presentation layer consist of React components and pages responsible for rendering UI elements and handling user interactions. The state management layer uses a centralized Redux store with feature-based slices. The service layer contains API client services used for backend communication, abstracting HTTP logic. The utils layer provides shared helper functions used across the frontend.

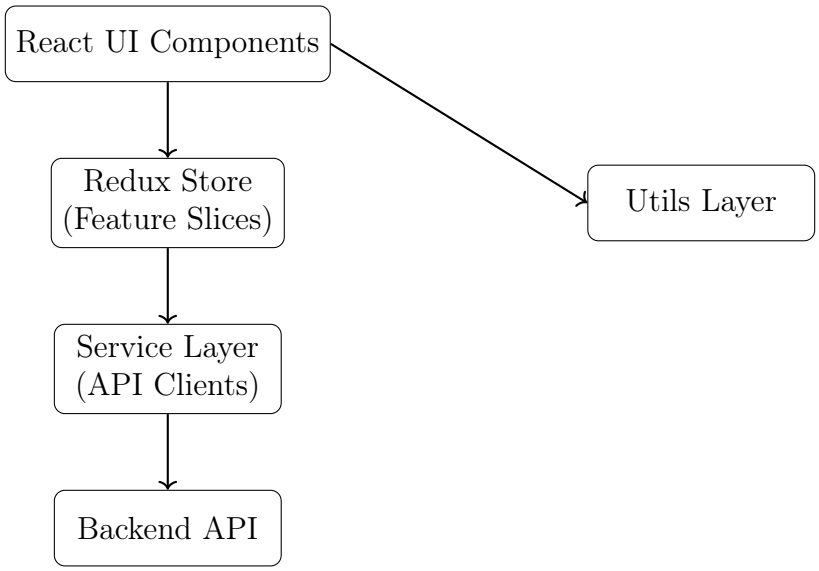


Figure 3.7: Frontend logical architecture showing component, state, and service layers

3.5.2 Visual Workflow Builder

The core feature of the FlowAPI Frontend is the visual workflow builder, which allows users to define API endpoint logic using a drag-and-drop interface. The function stack editor represents each function as a visual block that can be reordered and nested. Each block displays its function type and key configuration details.

Function configuration is handled through modal dialogs that present inputs specific to the selected function type. When a function block is selected, a configuration modal opens with context-aware fields. The interface include variable autocomplete, which suggests variables generated by previous functions, reducing configuration errors.

The workflow builder supports nested execution through ifElse and forEach blocks. Users can attach child function stacks to conditional branches or iteration blocks. Nested function placement is managed using a path-based structure that tracks function hierarchy within the workflow.

Real-time validation checks function configurations as users edit them. Validation errors are displayed immediately, allowing users to correct invalid inputs before saving.

Workflows can be exported and imported as JSON files. This allows workflows to be backed up, shared, or reused across projects.

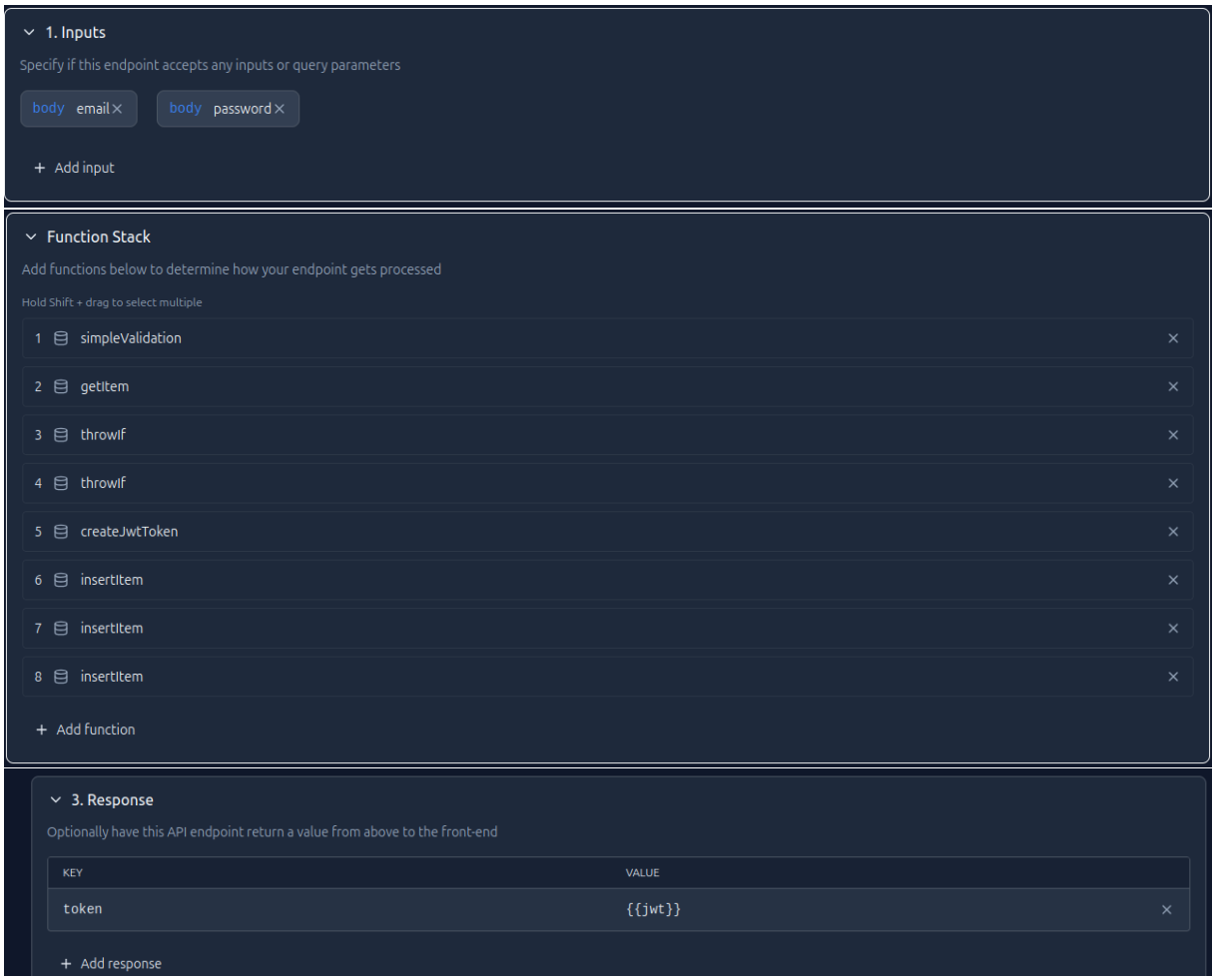


Figure 3.8: Workflow Section

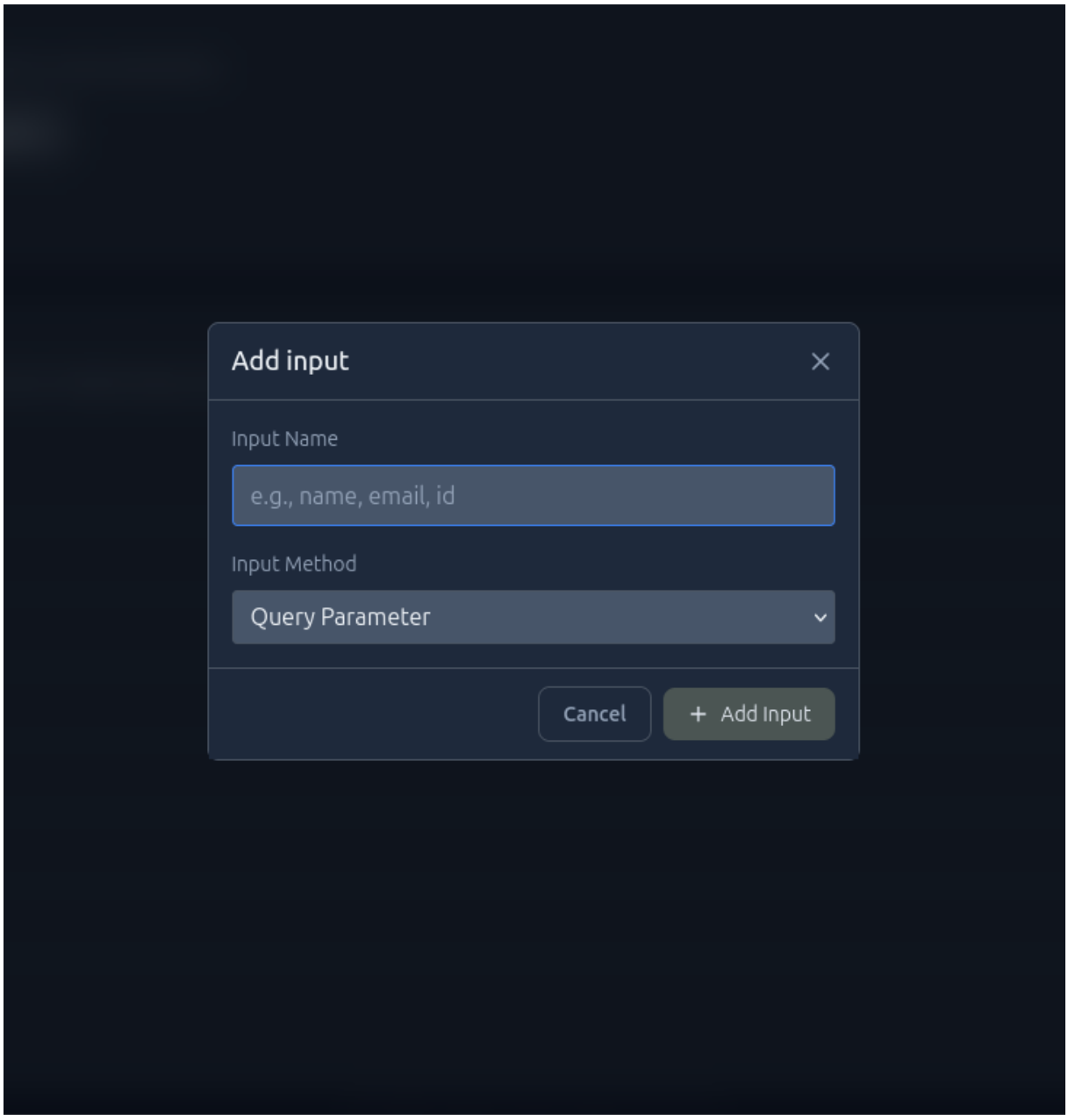


Figure 3.9: input modal

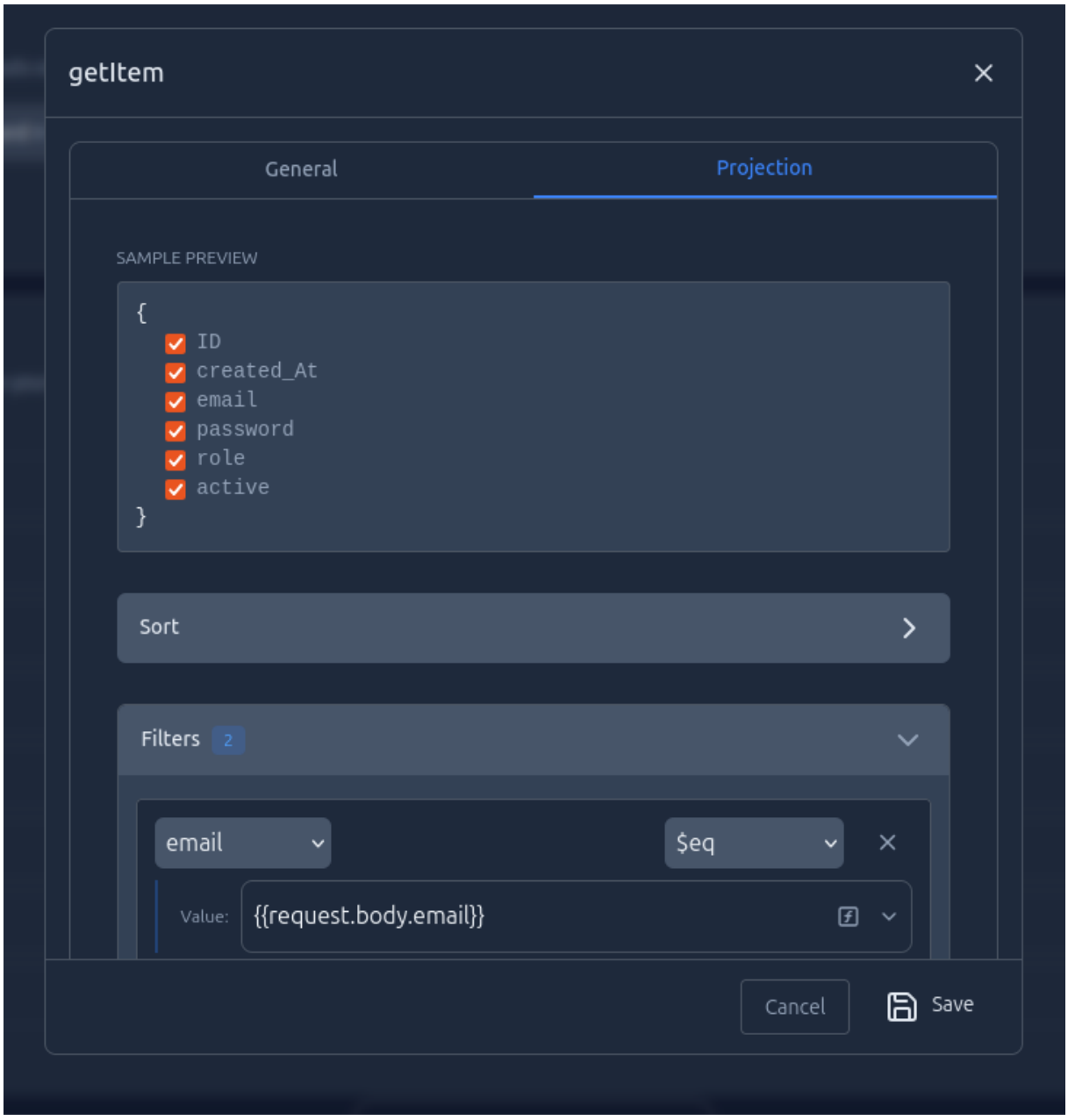


Figure 3.10: AddFunctionModal

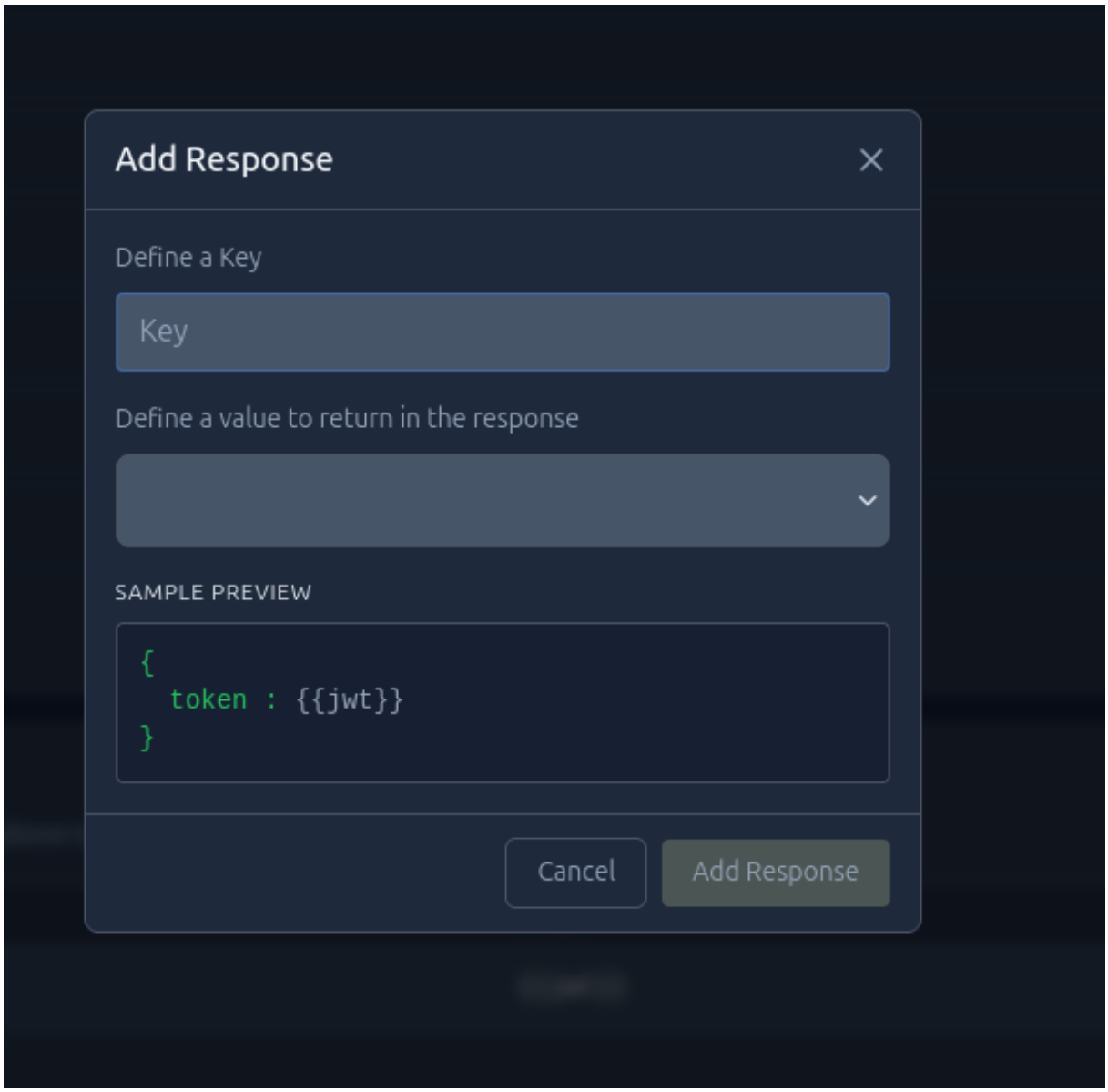


Figure 3.11: Response Modal

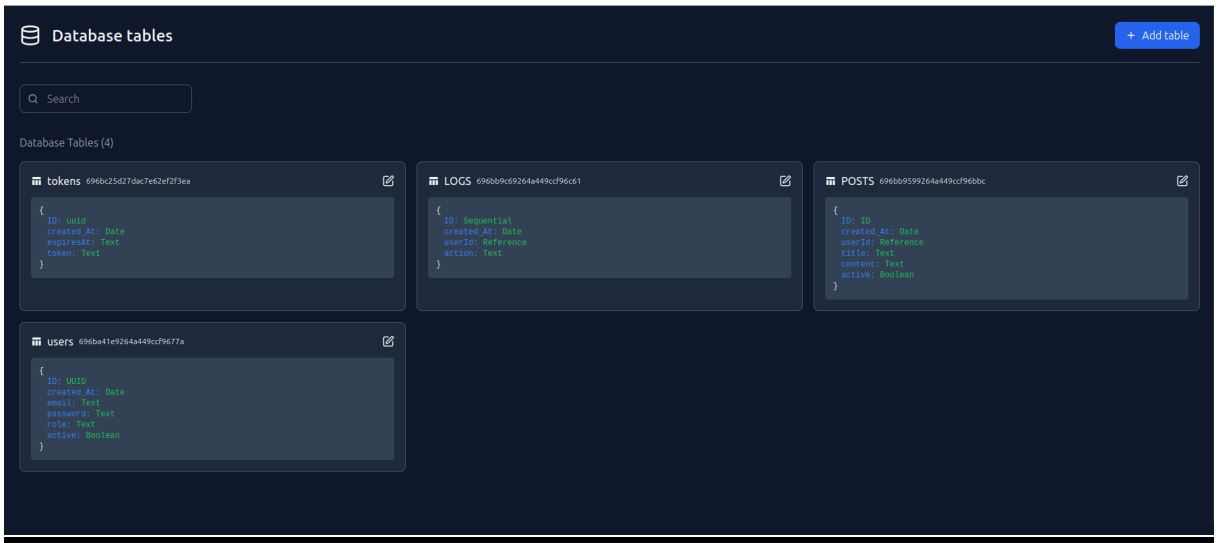
3.5.3 Database Management Interface

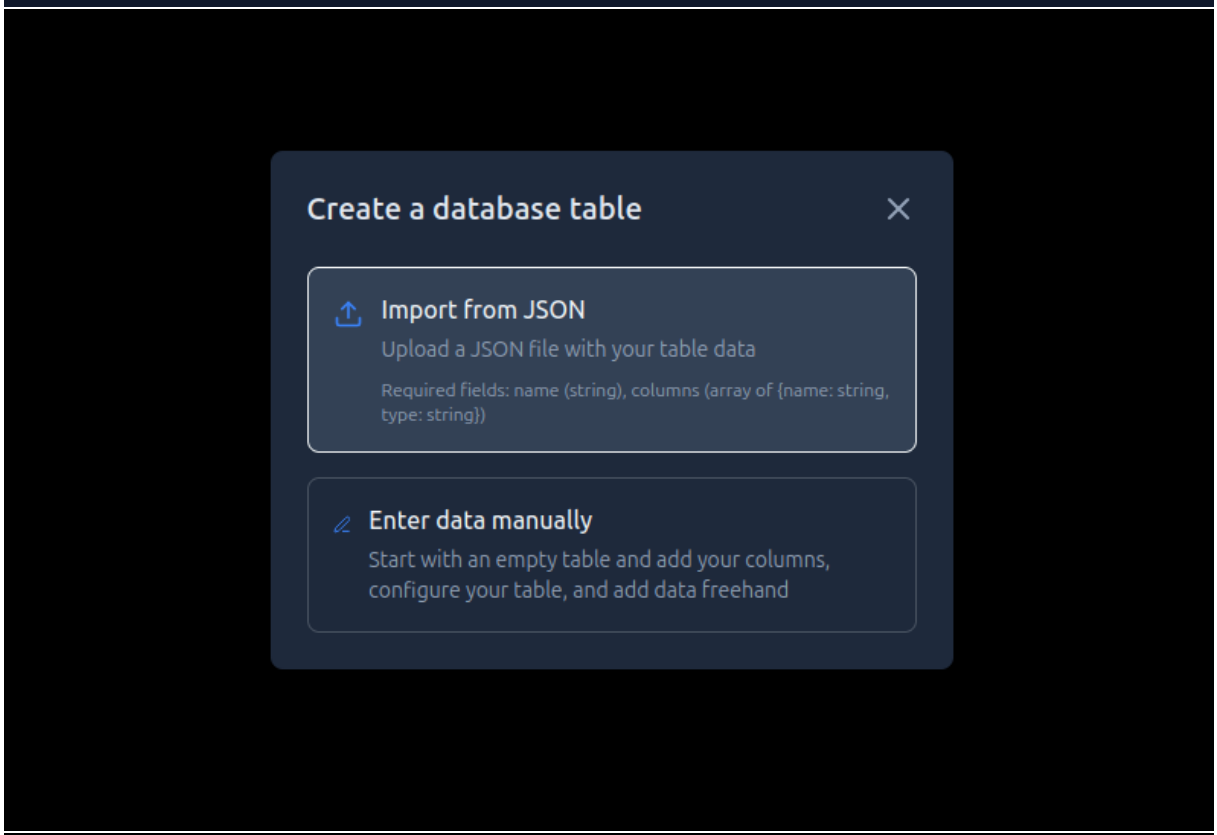
The FlowAPI Frontend provides database management interfaces that allow users to visually define and manage table schema. Users can create tables by specifying column definitions, primary key types, and column properties. Reference columns can be configured to represent relationships between tables.

The schema editor allows modification of existing tables. Users can add, remove, or update columns and adjust column types. Reference columns are managed through a visual selector for choosing related tables.

Records are displayed using a data grid that supports sorting and filtering. Columns can be shown or hidden based on user preference. Inline editing is prepared at the UI level but reserved for future backend support.

Record creation and editing are handled through dynamically generated forms based on the table schema. Reference fields use selection components to link records correctly.


Database tables
+ Add table
Search
Database Tables (4)
tokens 696bc25d27dac7e62ef2f3ea
{
ID: uuid
created_At: Date
expiresAt: Text
token: Text
}
LOGS 696bb9c69264a449ccf96c61
{
ID: Sequential
created_At: Date
userId: Reference
action: Text
}
POSTS 696bb9599264a449ccf96bbc
{
ID: ID
created_At: Date
userId: Reference
title: Text
content: Text
active: Boolean
}
users 696ba41e9264a449ccf9677a
{
ID: UUID
created_At: Date
email: Text
password: Text
role: Text
active: Boolean
}


Create a database table
Import from JSON
Upload a JSON file with your table data
Required fields: name (string), columns (array of {name: string, type: string})
Enter data manually
Start with an empty table and add your columns, configure your table, and add data freehand

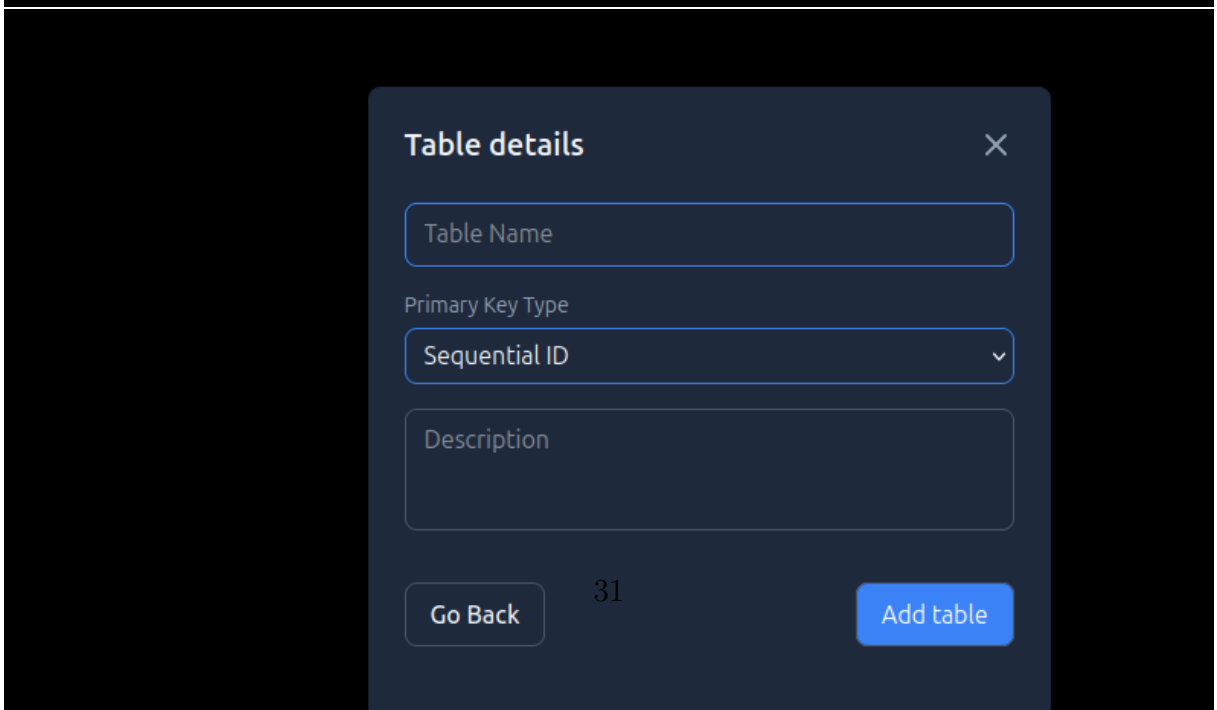


Table details
Table Name
Primary Key Type
Sequential ID
Description
Go Back
Add table

3.5.4 State Management with Redux

The FlowAPI Frontend uses Redux Toolkit to manage application state. The Redux store is separated into feature-based slices. The WorkFlowSlice manages workflow-related state, including function stacks, selected functions, and detected variables. Workflow changes are reflected immediately in the store.

The TableSlice stores table schemas and metadata, enabling consistent access across the application. Table identifiers and names are mapped for efficient reference.

The APISlice manages API groupings and endpoint organization, allowing endpoints to be structured logically within the interface.

State updates occur through dispatched actions processed by reducers. Components subscribed to state changes automatically re-render when relevant data updates.

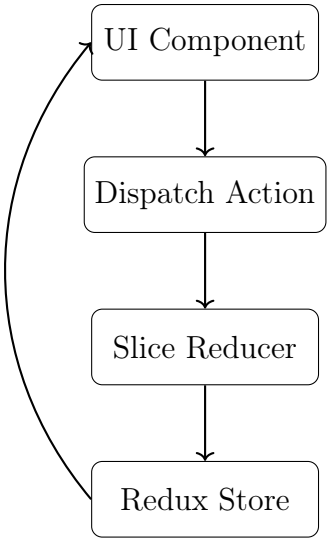


Figure 3.13: Redux Toolkit state update flow between components and store

3.5.5 Component Architecture

The FlowAPI Frontend follows a hierarchical component structure. Page components represent route-level views and manage data fetching and layout composition. Examples include DashboardPage, DatabasePage, WorkFlowPage, EndPointsPage, MarketplacePage, and SettingsPage.

Feature components encapsulate reusable logic for specific functionality. These include the FunctionStackSection, FunctionModal, and DataTableGrid components.

Shared UI components provide consistent interaction patterns across the frontend:

- **Modals**
- **Forms**
- **Buttons**
- **Inputs**
- **Cards**

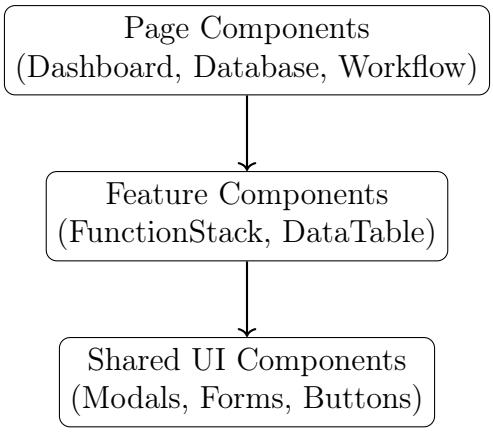


Figure 3.14: Hierarchical organization of frontend components

3.5.6 Function Block System

The FlowAPI Frontend implements a function block system that maps function types to dedicated React components. A factory pattern is used to return the correct block component based on function type.

Each block component receives standardized properties, including function data and update handlers. Configuration changes propagate back to the Redux store.

Each function type has its own configuration UI. For example, database blocks expose query and filter inputs, while control flow blocks expose condition builders and nested function stacks.

Supported function categories include:

- **Database Blocks**
- **Data Manipulation Blocks**
- **Control Flow Blocks**
- **Validation Blocks**
- **Security Blocks**

3.5.7 Variable Management System

The FlowAPI Frontend automatically tracks variables defined within workflows. Variables are extracted from function outputs and iteration aliases.

Variable scope is enforced based on workflow hierarchy. Parent-level variables are available to nested blocks, while nested variables are restricted to their local scope.

Autocomplete suggestions are provided in configuration inputs, filtered based on available scope and variable name matching.

3.5.8 AI Chat Assistant Integration

The FlowAPI Frontend includes an AI chat assistant that allows users to generate workflows using natural language. The chat interface sends user input along with contextual data such as database schema and existing workflow definitions.

The AI service returns generated workflow structures, including function stacks and request/response schema, which are applied directly to the endpoint state.

Generated workflows can be reviewed and modified manually after insertion.

Figure 3.15: AI-assisted workflow generation interaction sequence

3.5.9 Performance Optimizations

The FlowAPI Frontend applies selective performance optimizations using useMemo and useCallback. These hooks are used to avoid unnecessary recalculation of derived values and recreation of callback functions.

At this stage, the frontend doesn't implement route-level code splitting or React.memo-based component memoization. Performance improvements are primarily achieved through memoization of derived state and event handlers.

3.5.10 User Experience Features

The FlowAPI Frontend supports drag-and-drop interactions for workflow editing. Changes are reflected immediately through synchronized state updates.

Auto-save mechanisms persist changes to the backend with denounced requests. Toast notifications provide feedback for successful actions and errors.

Loading indicators and error messages communicate operation status clearly to users.

3.5.11 Responsive Design

The FlowAPI Frontend is designed to operate across different screen sizes. Layouts adapt based on available space using CSS Grid and Flexbox.

Certain components adjust behavior depending on screen size, such as modal presentation and navigation layout. Touch interactions are supported for compatible devices.

3.6 AI Server

3.6.1 Overview

The AI Server is a Python/FastAPI microservice that provides AI-powered workflow generation for FlowAPI. enables users to create API endpoint workflows by describing their requirements in natural language. The service uses LangChain to orchestrate Large Language Models (LLMs) that understand user requests, query available functions blocks and database schema, and generate suitable function stacks.

The AI Server represents a significant innovation in making API development accessible to non-technical users. By allowing users to describe their API requirements in plain English, the system eliminates the barrier to understanding function stacks and workflow construction. Users can simply describe what they want their API to do, and the AI automatically generates the appropriate workflow.

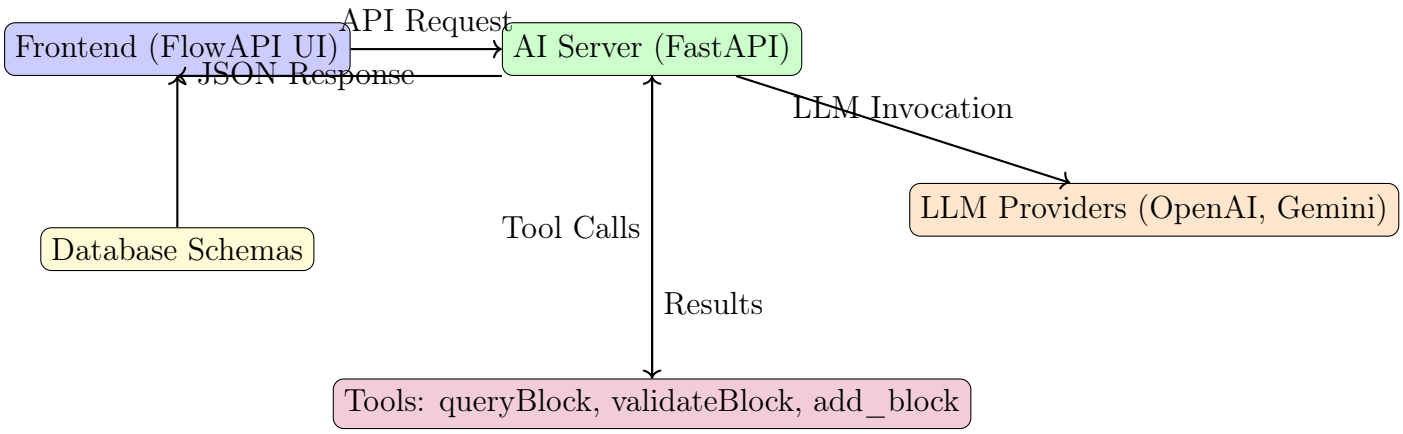


Figure 3.16: High-Level Architecture of the AI Server

3.6.2 LangChain Integration

The AI Server uses LangChain 1.2.4 for LLM orchestration, provide a framework for building applications powered by language models. LangChain abstracts away the complexity of working with different LLM providers, enabling the system to use OpenAI backend.

The system represent ReAct (Reasoning and Acting) agent pattern, which enables the AI to reason about tasks and take actions. The agent can think about what need to be done, decide which actions to take, execute those actions, and observe the results. This pattern is particularly well-suited for workflow generation, as the agent needs to understand requirements, query available functions, validate configurations, and construct workflows.

The agent has access to three main tools that enable it to interact with the system. The `queryBlock` tool retrieves documentation for workflow blocks, allowing the agent to understand what functions are available and how to use them. The `validateBlock` tool validates a block configuration against its schema, ensuring that generate blocks are correct before adding them to the workflow. The `add_block` tool adds a validated block to the workflow, enabling the agent to build workflows incrementally.

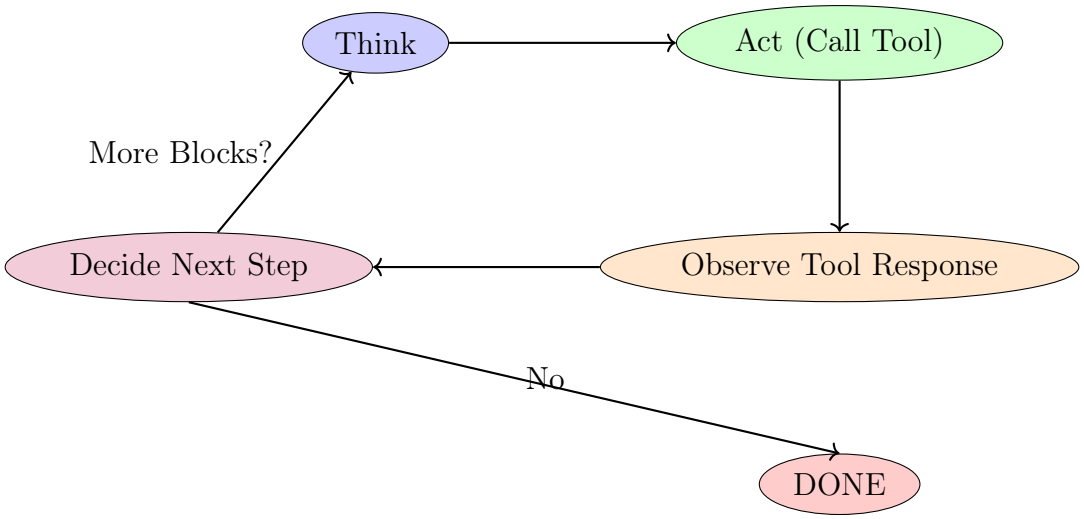


Figure 3.17: LangChain ReAct Agent Flow

3.6.3 Function Stack Generation Process

The core functionality of the AI Server is generating function stacks from natural language descriptions. This process involves several steps that transform a user's natural language request into a working API workflow. The process begins with prompt construction, where the system builds a comprehensive prompt that includes the user's natural language request, available database tables and their schema, the current function stack (if editing an existing endpoint), the request schema (path params, query params, body), and system instructions for workflow construction.

The prompt is carefully engineered to provide the agent with all the information it needs to generate an appropriate workflow. The system instructions explain how to construct workflows, what functions are available, and how to use them. The database schema information give the agent ability to understand what data is available and how to query it. The current workflow (if any) provides context for editing existing endpoints.

Agent invocation happen with tools for querying blocks, validating, and adding blocks. The agent is configured with a recursion limit to prevent infinite loops, as the agent might get stuck querying functions without making progress. A timeout protection mechanism ensures that long-running operations don't block the system indefinitely. These safety measures are crucial for ensuring that the AI system remains responsive and doesn't consume excessive resources.

The agent then iteratively constructs blocks by querying block documentation to understand available functions, constructing block configurations based on user requirements, validating blocks before adding them, and adding validated blocks to the workflow. This iterative process continues until the workflow is complete or the agent determines that no more blocks are needed.

Result extraction happens after the agent completes its work. The system extracts the generated function stack from the agent's response, which may be in various formats depending on how the agent returns results. The extraction process handles different

response formats and ensures that a valid function stack is returned to the front-end.

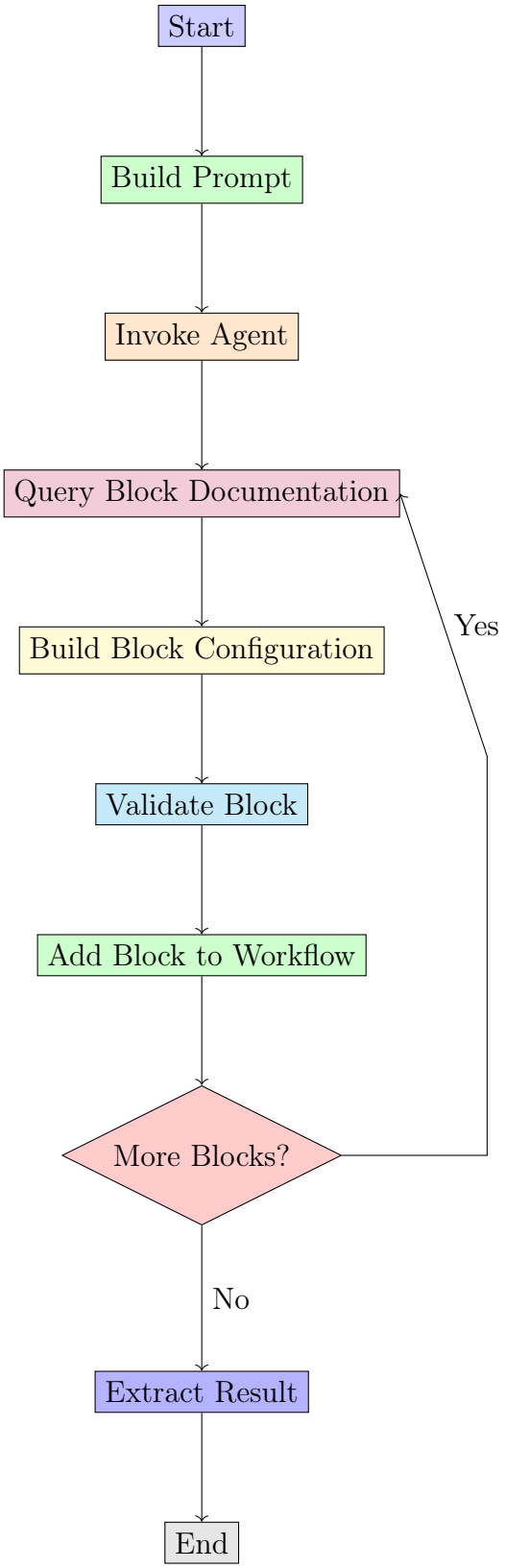


Figure 3.18: Function Stack Generation Process

3.6.4 Tool Implementations

The AI Server provides three main tools for the agent, each serving a specific purpose in the workflow generation process. The `queryBlock` tool retrieves official documentation for workflow blocks. When the agent needs to understand how to use a function, it calls this tool with the block name. The tool returns a human-readable description that includes the block's purpose, configuration schema with all fields, field types and requirements, and usage examples. This documentation enables the agent to understand what functions are available and how to configure them correctly.

The `validateBlock` tool validates a block configuration against its schema. Before adding a block to the workflow, the agent calls this tool to ensure that the block configuration is correct. The tool returns a validation result that includes a boolean indicating if validation passed, an array of error messages if validation failed, and the validated block object. This validation step is crucial for ensuring that generated workflows are correct and will execute successfully.

The `add_block` tool adds a validated block to the workflow. After validating a block, the agent calls this tool to add it to the workflow. The tool returns a success result that includes a boolean indicating if the block was added, a success or error message, the index of the added block, and the total number of blocks in the workflow. This tool enables the agent to build workflows incrementally, adding blocks one at a time until the workflow is complete.

3.6.5 Prompt Engineering

The AI Server uses a streamlined prompt system to guide the agent toward generating correct workflows. The system prompt defines the agent's role as a "workflow builder for a JSON-based backend (Xano-like)," setting the context for the agent's behavior. The prompt specifies a clear termination signal ("DONE") to ensure the agent knows when workflow generation is complete.

Workflow rules are clearly defined in the prompt, specifying a mandatory pipeline for block creation: query the block documentation, validate the block, and then add it to the workflow. This pipeline ensures that the agent follows a consistent process and generates correct workflows.

The prompt includes a list of available block names, enabling the agent to understand what functions are available. Basic placeholder syntax is explained, showing the agent how to use placeholders for dynamic values (request.body.email, user.id). Essential rules are emphasized, such as always querying schemas before use, validating before adding, and following exact formats. These rules help prevent the agent from making mistakes that would result in invalid workflows.

The user prompt includes the user's natural language request, which is the primary input for workflow generation. It also includes a summary of available database tables with their columns and types, enabling the agent to understand what data is available. The current function stack (if editing) provides context for modifications, and the request

schema (path parameters, query parameters, body structure) helps the agent understand the endpoint's input requirements.

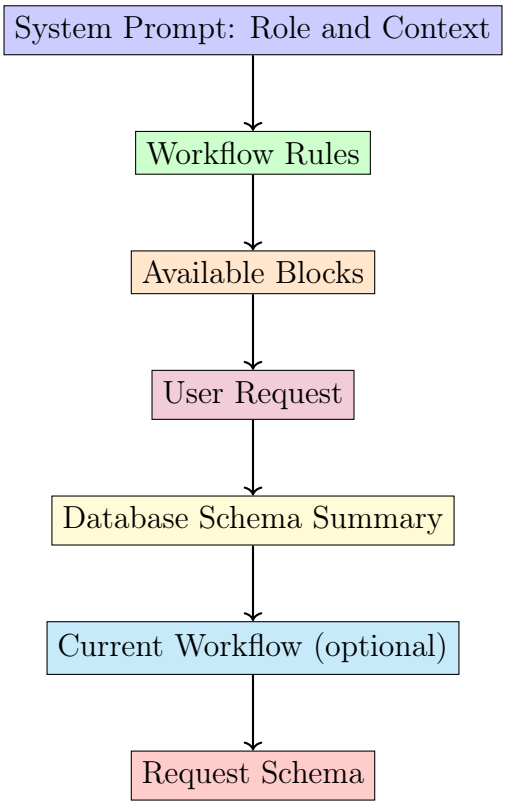


Figure 3.19: AI Server Prompt Structure

3.6.6 Logging and Monitoring

The AI Server includes basic logging for operational visibility. Agent execution logging records key events during agent execution, including start/completion status and basic error information. This provides essential visibility into agent operations.

Block addition logging records when blocks are successfully added to workflows through tool responses. The system logs success messages with block names and total block counts, providing a basic record of agent progress.

Error logging includes exception messages and basic context information to help identify issues. The logging system captures timeout errors, validation failing, and other operational exceptions with descriptive error messages.

Basic execution metrics are tracked, including iteration limits and timeout configurations. These settings help ensure safe operation and prevent infinite loops, though detailed performance analytics are not currently implemented.

3.6.7 Integration with Frontend

The AI Server integrates with the frontend through RESTful APIs, providing a clean interface for workflow generation. The API endpoint `POST /api/v1/agent/run` accepts requests with user request, database schema, current workflow, and request schema. The request format is JSON, making it easy for the frontend to construct requests.

The response format is also JSON, containing the generated function stack. This consistent format makes it easy for the frontend to process responses and apply generated workflows. Error handling returns structured error responses that the frontend can display to users, providing clear feedback when workflow generation fails.

The integration is designed to be seamless, with the frontend handling all the complexity of constructing requests and processing responses. The AI Server focuses on workflow generation, while the frontend handles user interaction and workflow application. This separation of concerns makes both systems easier to maintain and evolve.

3.6.8 Future Enhancements

The AI Server has significant potential for future enhancements that could further improve its capabilities. Workflow optimization could enable the AI to suggest improvements for existing workflows, such as combining functions or optimizing execution order. Error explanation could help users understand errors in workflows and suggest fixes, making the system more helpful.

Code documentation generation could automatically create documentation for generated workflows, helping users understand what their APIs do. Multi-model support could enable the system to use different LLM providers and models for different tasks, optimizing for cost and performance. Custom model fine-tuning could improve workflow generation quality by training models specifically on FlowAPI workflows.

These enhancements would make the AI Server even more powerful and useful, further lowering the barrier to API development and making the platform more accessible to non-technical users.

3.7 Infrastructure

3.7.1 Overview

FlowAPI was built to run inside containers using Docker. This approach allows the platform to be deployed in the same way was used in development, testing, and production environments. The system is made up of many microservices, and each service could be deployed on its own or together with others using Docker Compose tool

By using containerization, FlowAPI avoid many deployment problems caused by differences between environments. Services must behave the same way on a developer's

machine as they do in production, which makes deployment more reliable and scaling easier.

3.7.2 Containerization

All FlowAPI services are packaged as Docker containers. Each service includes its own Dockerfile that defines the environment and required dependencies.

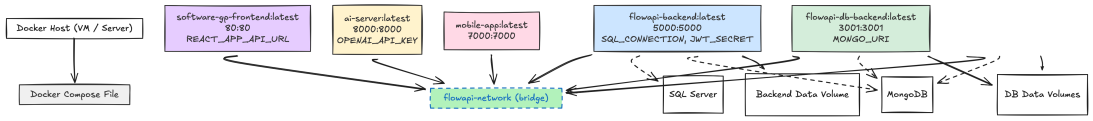


Figure 3.20: Docker container architecture for FlowAPI services

3.7.3 Docker Compose Configuration

Docker Compose is used in development environments to manage multiple services together. It allows all services, networks, and volumes to be defined in a single configuration file. With one command, developers can start or stop the entire system.

Docker Compose make sure that services start in the correct order, with the required dependencies running first. Services communicate using internal service names, which simplifies networking. volumes are used for databases and file storage, ensuring that data isn't lost when containers being restarted

Configuration for each environment is in the Compose files. This make it easier to manage service settings and dependencies and reduces setup complexity during development.

3.7.4 Database Infrastructure

FlowAPI use more than one database technology to match different data needs. SQL Server is used for structured and relational data that requires strong consistency and transactions. This includes user accounts, authentication data, application metadata, marketplace information, and team management data.

MongoDB is used for loosly and dynamic data. API endpoint definitions, function stacks, and request and response templates are stored as JSON documents in MongoDB. Each application have its own MongoDB database, which ensures complete data separation between different applications.

3.7.5 Environment Configuration

All services are configured using environment variables. This approach keeps sensitive information, such as database connection strings and secret keys, out of the source code. MongoDB and SQL Server connection details are provided through environment variables.

JWT settings, including secret keys and token expiration values, are also configured through environment variables. API keys for external services such as OpenAI and Stripe was handled the same way.

3.7.6 Future Infrastructure Enhancements

Future improvements may contain deploying services using Kubernetes for advanced orchestration, adding CI/CD pipelines for automated builds and deployments, and using Infrastructure as Code tools such as Terraform.

3.8 Mobile Application

3.8.1 Mobile Application Architecture

The mobile application is built using React Native and Expo, allowing a single codebase to run on Android, iOS, and the web. Expo Router is used for file-based routing, where application screens are organized inside the `app/` directory. Screens related to authentication are placed in a dedicated routing group, while the main application screens are grouped separately. The root layout is responsible for initializing navigation and registertion global provider such as theme handling.

The application follows a modular design approach. Individual screens focus on rendering the user interface and responding to user interactions, while data handling and backend communication are delegated to a separate service layer. This structure limits direct dependencies between UI components and API logic, that improves code maintenance and simplify testing.

3.8.2 Service-Oriented API Layer

Communication with the backend is centralized in a dedicated service layer implemented using Axios. Each functional domain is represented by its own service file, including authentication, applications, marketplace, dashboard analytics, settings, API groups, endpoints, and table management. This organization keeps network logic separate from screen components and reduces duplication.

Axios interceptors are configured to automatically attach the authentication token to outgoing requests. The interceptors also monitor responses and handle unauthorized access by performing session cleanup when app receives a 401 status code. This ensures uniform authentication handling across all API interactions without requiring additional logic at the screen level.

3.8.3 Authentication and Session Management

The application uses a JWT-based authentication mechanism. User login and registration are managed through the authentication service, which stores the issued token and decod2

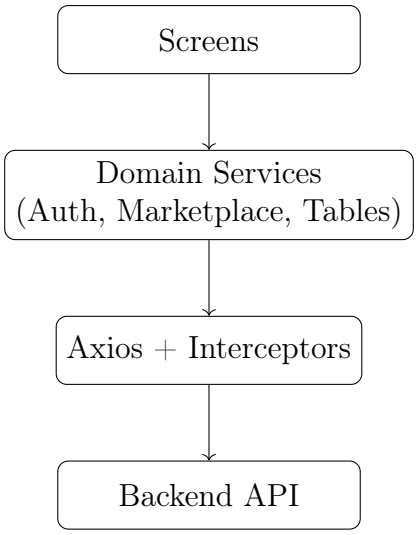


Figure 3.21: Centralized API communication through a service-oriented layer using Axios interceptors.

ed user information in AsyncStorage. Storing this data locally allows the application to preserve user sessions between restarts.

Token validity is checked during API communication. If a token is found to be expired or invalid, the stored session data is cleared and the user is redirected to the authentication flow. This approach ensures that access to protected screens is restricted to authenticated users only.

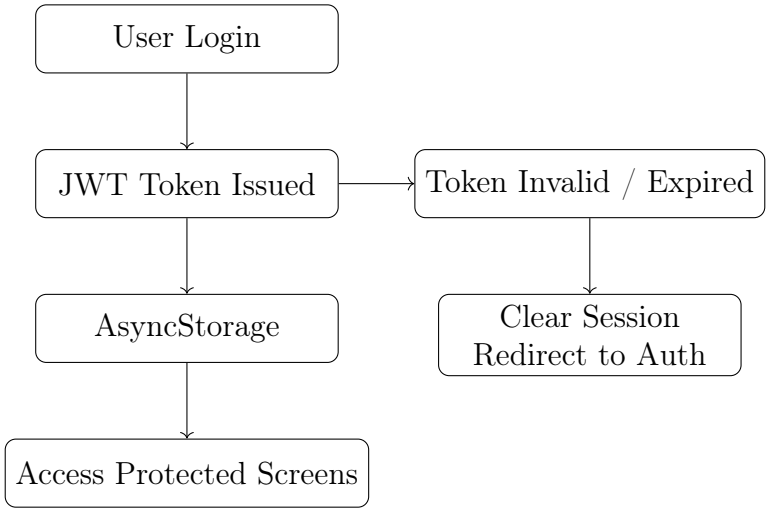


Figure 3.22: Authentication and session management flow using JWT tokens and persistent storage.

3.8.4 Navigation and Screen Organization

Navigation is implemented via Expo Router and combines stack navigation, tab navigation, and modal presentation. The bottom tab bar represents the entry points to the

main areas of the application, such as the dashboard, marketplace, and settings. Within each section, stack navigation is used to support hierarchical flows as detail pages and configuration screens.

Modal screens are used for forms and focused tasks, allowing users to complete actions without fully leaving their current screen. Safe Area Context is applied across the application to ensure that layouts adapt correctly to devices with notches and system interface elements.

3.8.5 State and Theme Management

Global theme management is handled through React Context. The ThemeContext supplies color values and style tokens that are consumed by screens and reusable components. This setup ensures consistent appearance throughout the application and support switching between light and dark themes.

When the theme is toggled, updates are applied globally and reflected immediately across all visible components. Using a centralized context prevents duplicated styling logic and maintains visual consistency across navigation layers.

3.8.6 Reusable Component System

The application includes a reusable component layer that encapsulate commonly used interface elements. These components are shared across multiple screens to maintain consistent behavior and styling. Icon rendering is handled using the Lucide and Feather icon libraries, which provide scalable vector icons across all supported platforms.

3.8.7 User Experience and Feedback

During API operations, the application displays loading indicators at the component or screen level to inform users that data is being fetched or processed. Toast notifications are used to communicate success and error messages without disrupting the current navigation flow. These messages are displayed briefly and dismissed automatically.

Error handling is implemented both in the service layer and within screens. Errors that affect user actions are presented in a clear and readable format, while authentication-related issues trigger session recovery and redirection when required.

3.8.8 Platform Support

The mobile application targets Android, iOS, and web platforms through Expo. Android and iOS builds are supported using Expo Go and development builds. The web version is primarily intended for development and testing purposes and share the same routing structure and codebase as the mobile platforms.

Chapter 4

Results and Discussion

4.1 System Implementation

The implemented FlowAPI platform successfully delivers a comprehensive visual API development environment. This section presents the key interfaces and functionality of the system, followed by an analysis of its performance and effectiveness. The platform demonstrates that visual API development can be powerful and accessible, enabling users to create complex APIs without writing code.

4.2 User Interface and Development Flow

4.2.1 Dashboard and Project Management

The FlowAPI platform provides an intuitive entry point for users through its dashboard interface. Users can view all their projects in a card-based layout, with each project showing key information such as name, description, and creation date. The dashboard displays statistics about API usage, endpoints, and database tables, giving users a quick overview of their platform activity.

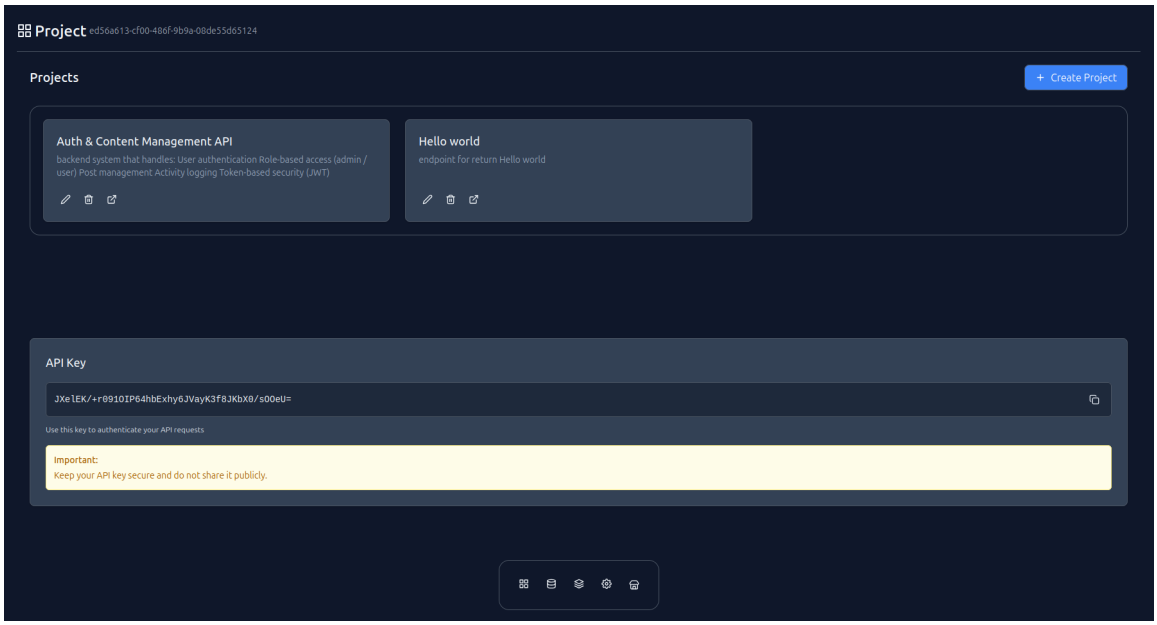


Figure 4.1: FlowAPI Platform Dashboard showing projects and statistics

The dashboard includes API key management, allowing users to view and copy their application API keys. This makes it easy for users to integrate their APIs with external applications. The dashboard also provides quick access to create new projects, switch between projects, and access key features such as the workflow builder and database management.

4.2.2 Database Management Interface

Users can create and manage database tables through an intuitive visual interface. The table creation process guides users through define table names, adding columns with suitable types, configuring primary keys, and making relationships through reference columns. The interface provides real-time validation feedback, helping users create correct schema.

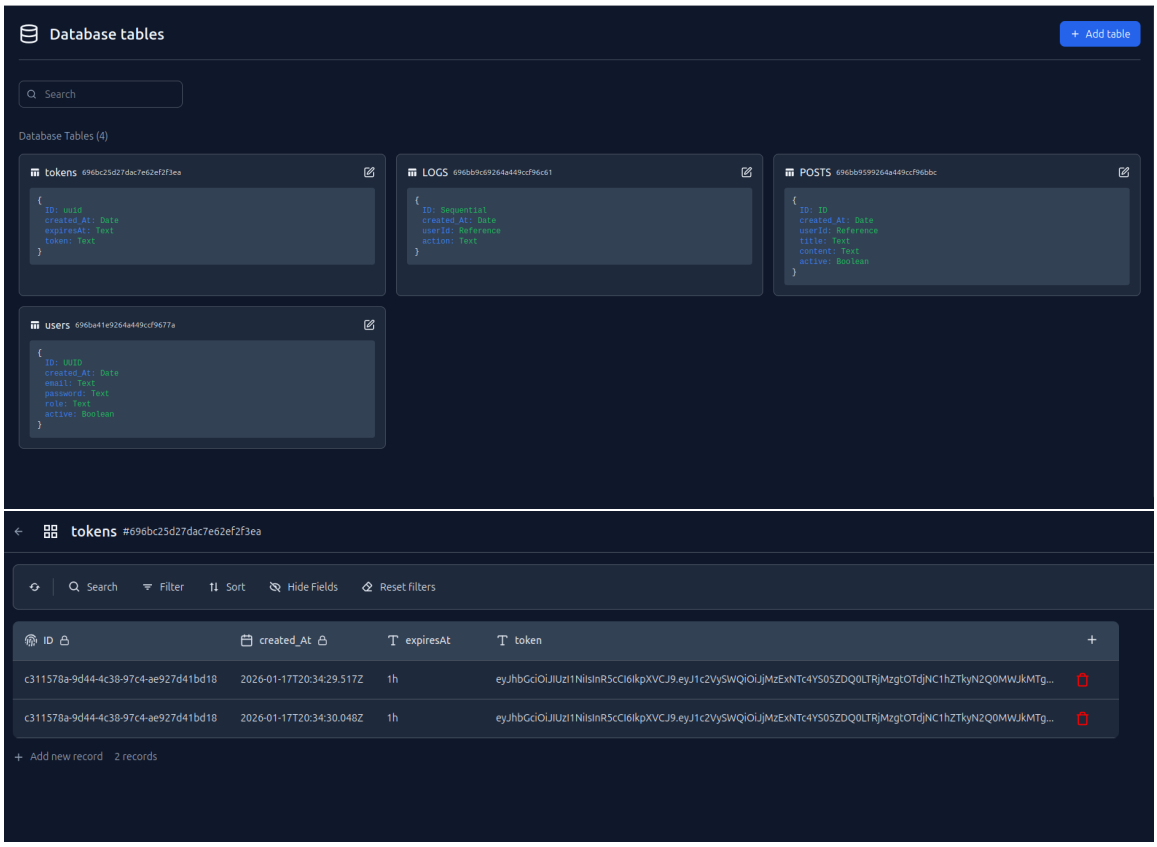


Figure 4.2: Database Table Management Interface

The data grid interface displays records in a sortable, filterable table format. Users can sort by any column, apply filters using various operators, and paginate through large datasets. The grid supports column visibility toggling and provides forms for adding, editing, and deleting records. Reference fields include visual interfaces for managing relationships, making it easy to work with related data.

4.2.3 Visual Workflow Builder

The core feature of FlowAPI is the visual workflow builder, which allows users to create API endpoint logic by combining functions in a drag-and-drop interface. The builder displays functions as visual blocks that can be dragged, dropped, and reordered. Each block shows its type and key information, making it easy to understand the workflow at a glance.

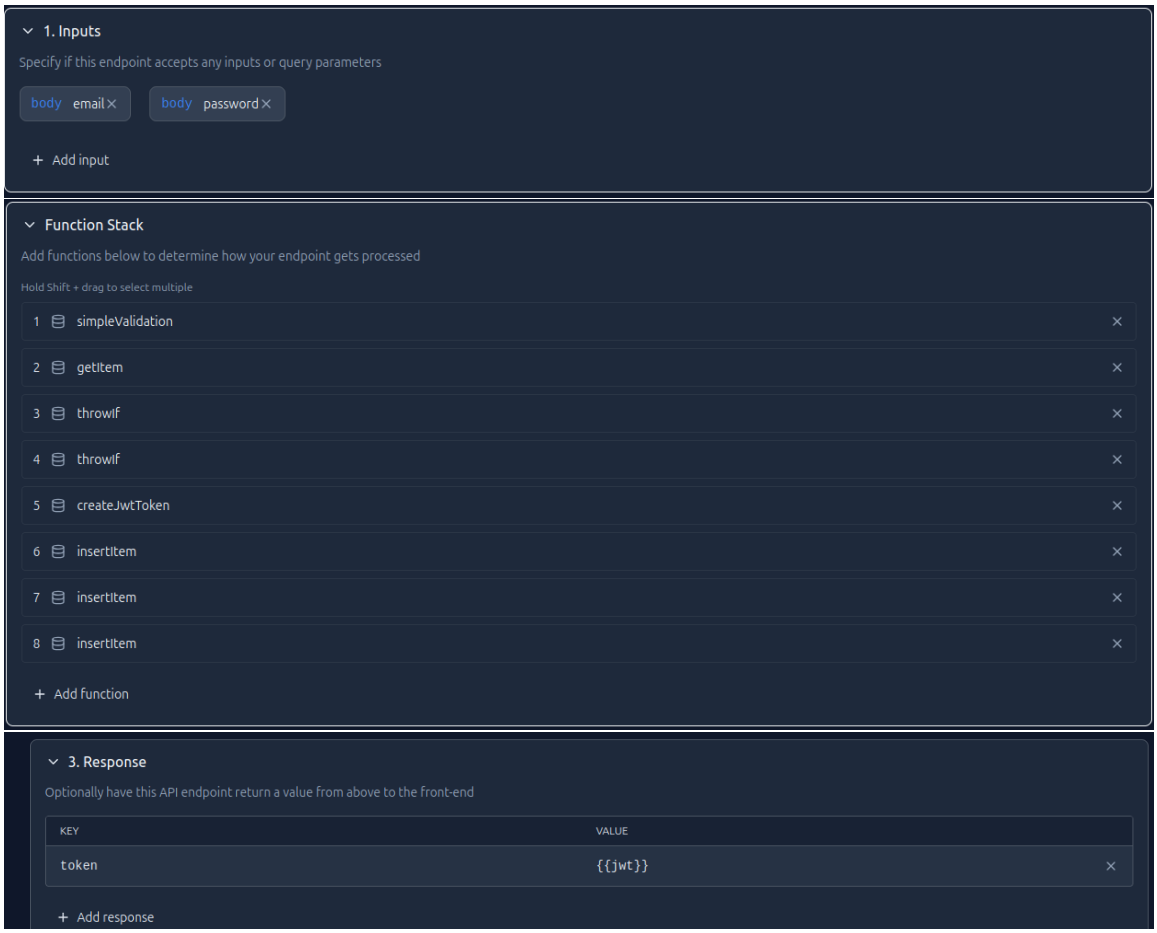
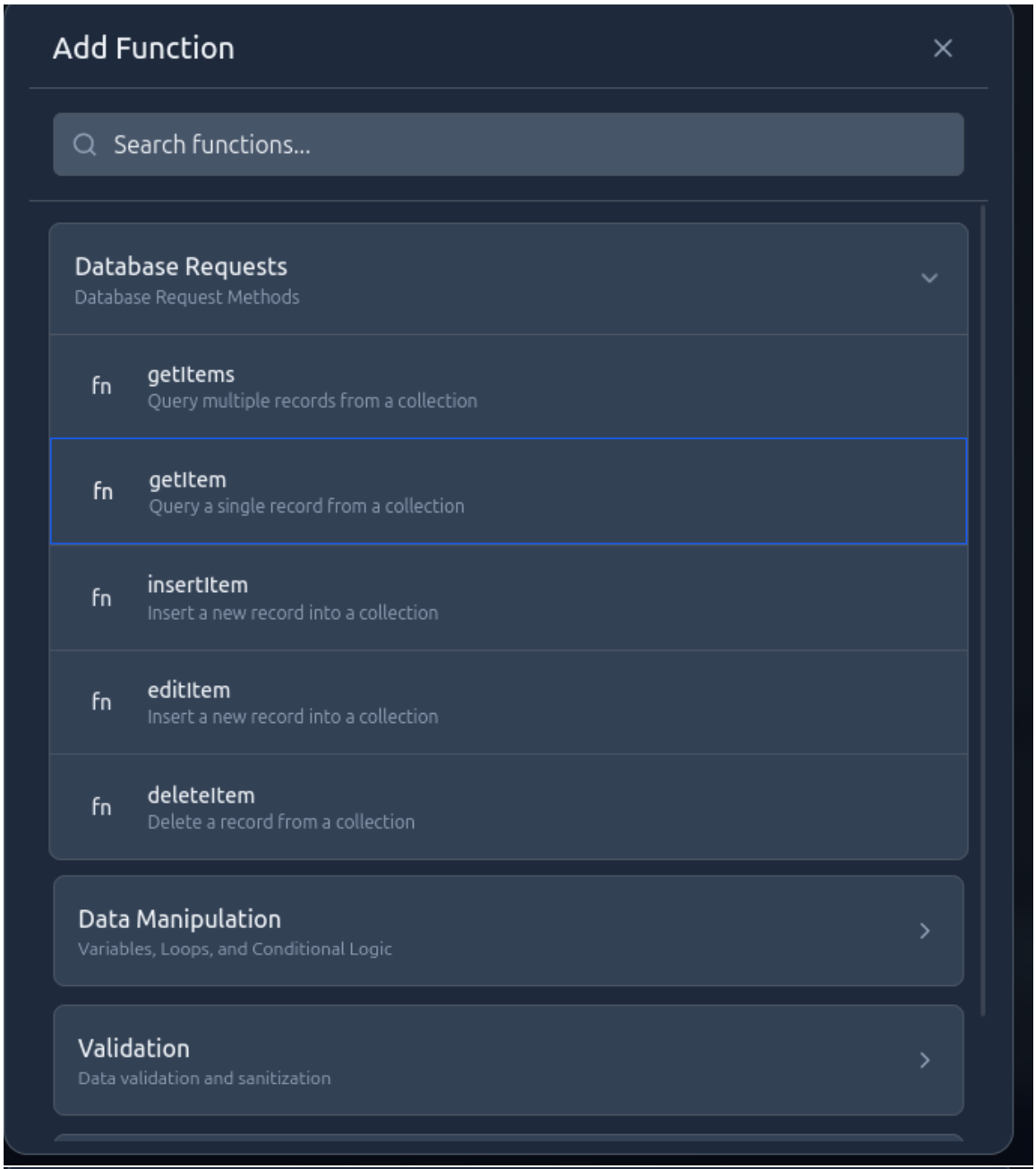
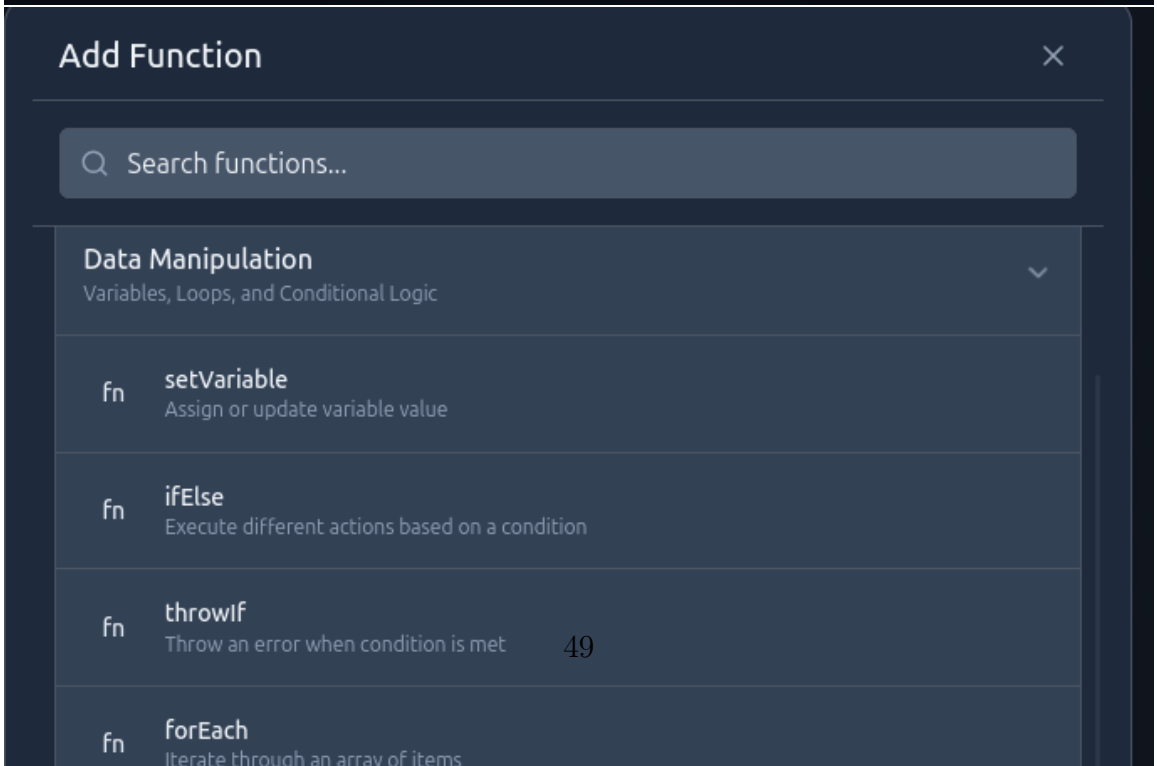


Figure 4.3: Visual Workflow Builder Interface


Add Function
Search functions...
Database Requests
Database Request Methods
fn
getItems
Query multiple records from a collection
fn
getItem
Query a single record from a collection
fn
insertItem
Insert a new record into a collection
fn
editItem
Insert a new record into a collection
fn
deleteItem
Delete a record from a collection
Data Manipulation
Variables, Loops, and Conditional Logic
Validation
Data validation and sanitization


Add Function
Search functions...
Data Manipulation
Variables, Loops, and Conditional Logic
fn
setVariable
Assign or update variable value
fn
ifElse
Execute different actions based on a condition
fn
throwIf
Throw an error when condition is met
fn
forEach
Iterate through an array of items

Function configuration happens through modal dialog that provide context-aware inputs. When users click on a function block, a modal opens with configuration options specific to that function type. The configuration interface includes variable auto-complete, which intelligently suggests available variables from previous functions. This auto-complete system helps users avoid typos and ensures correct variable references.

The system supports nested functions in ifElse and forEach blocks, enabling complex control flow. Users can add functions to both branches of an ifElse block, and define sequences of functions that execute for each item in a forEach loop. Real-time validation provides immediate feedback on function configuration validity, helping users fix issues before saving.

4.2.4 Function Configuration

Each function in the workflow can be configured through a dedicated modal dialog. The configuration interface provides inputs specific to each function type, with validation and helpful hints. For database functions, users can select collections, build filters, configure projections, and set sorting options. For data manipulation functions, users can set variable names and values, with support for placeholders and expressions.

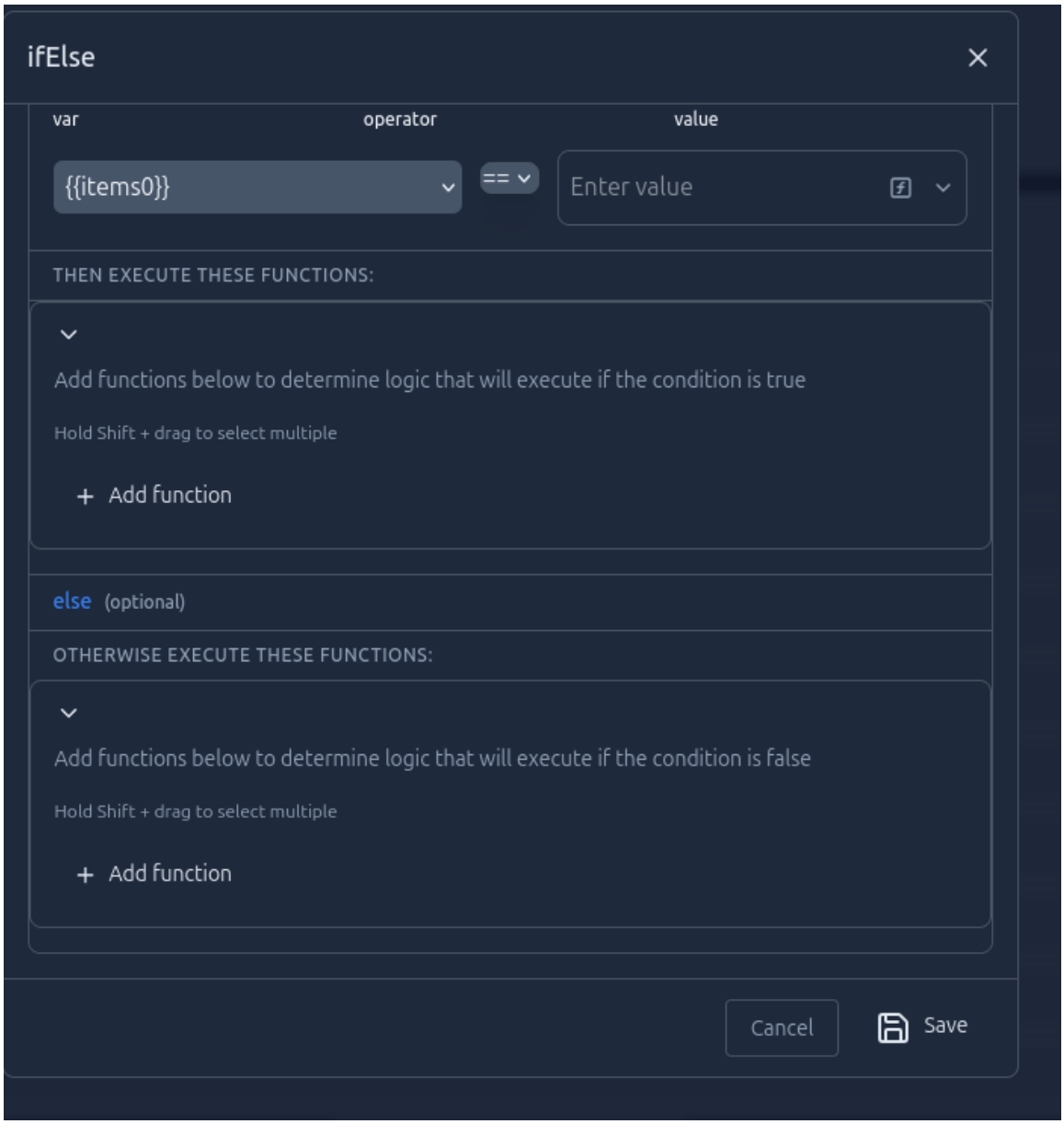


Figure 4.5: Ifelse Configuration Modal

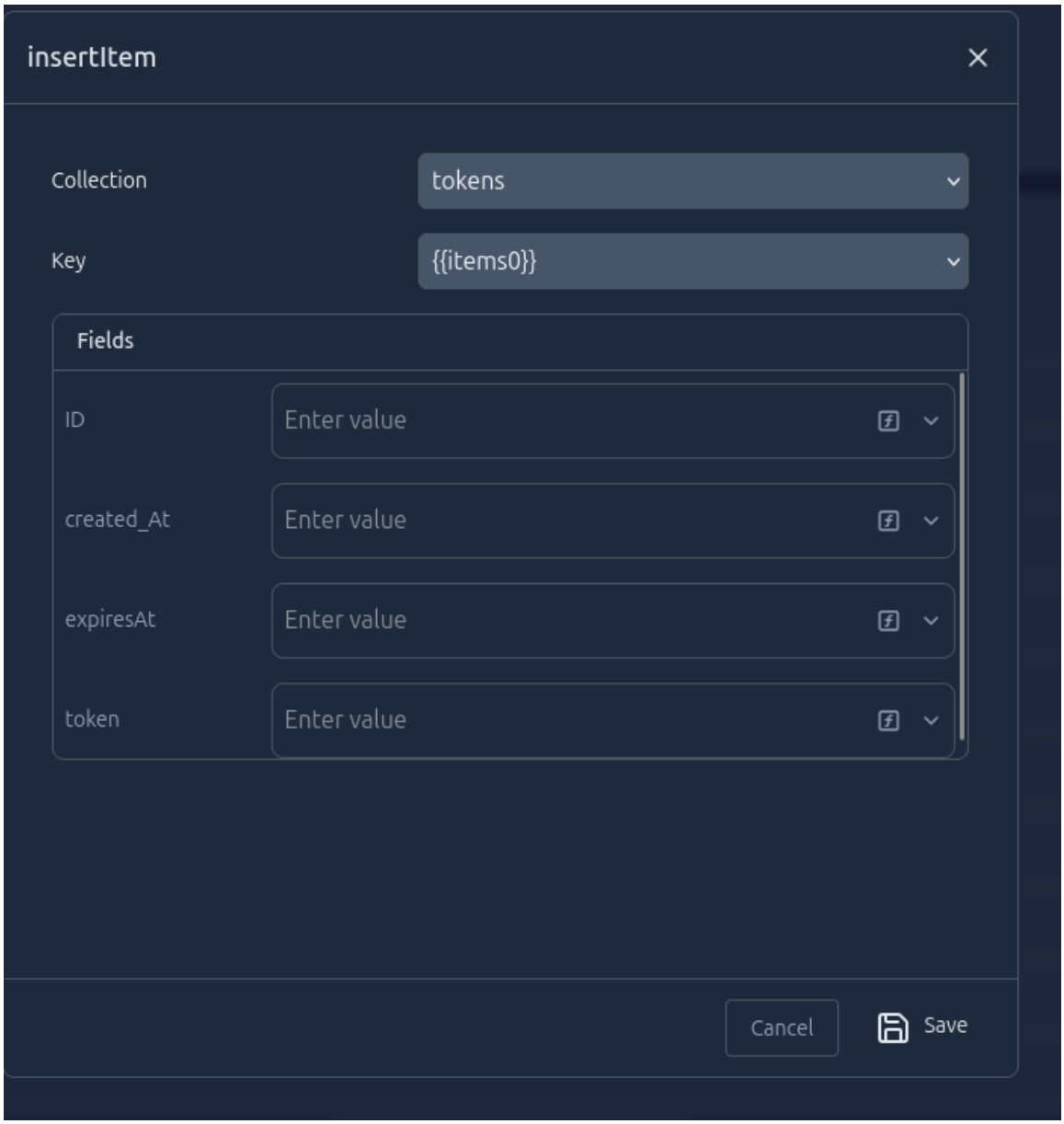


Figure 4.6: InsertItem Configuration Modal

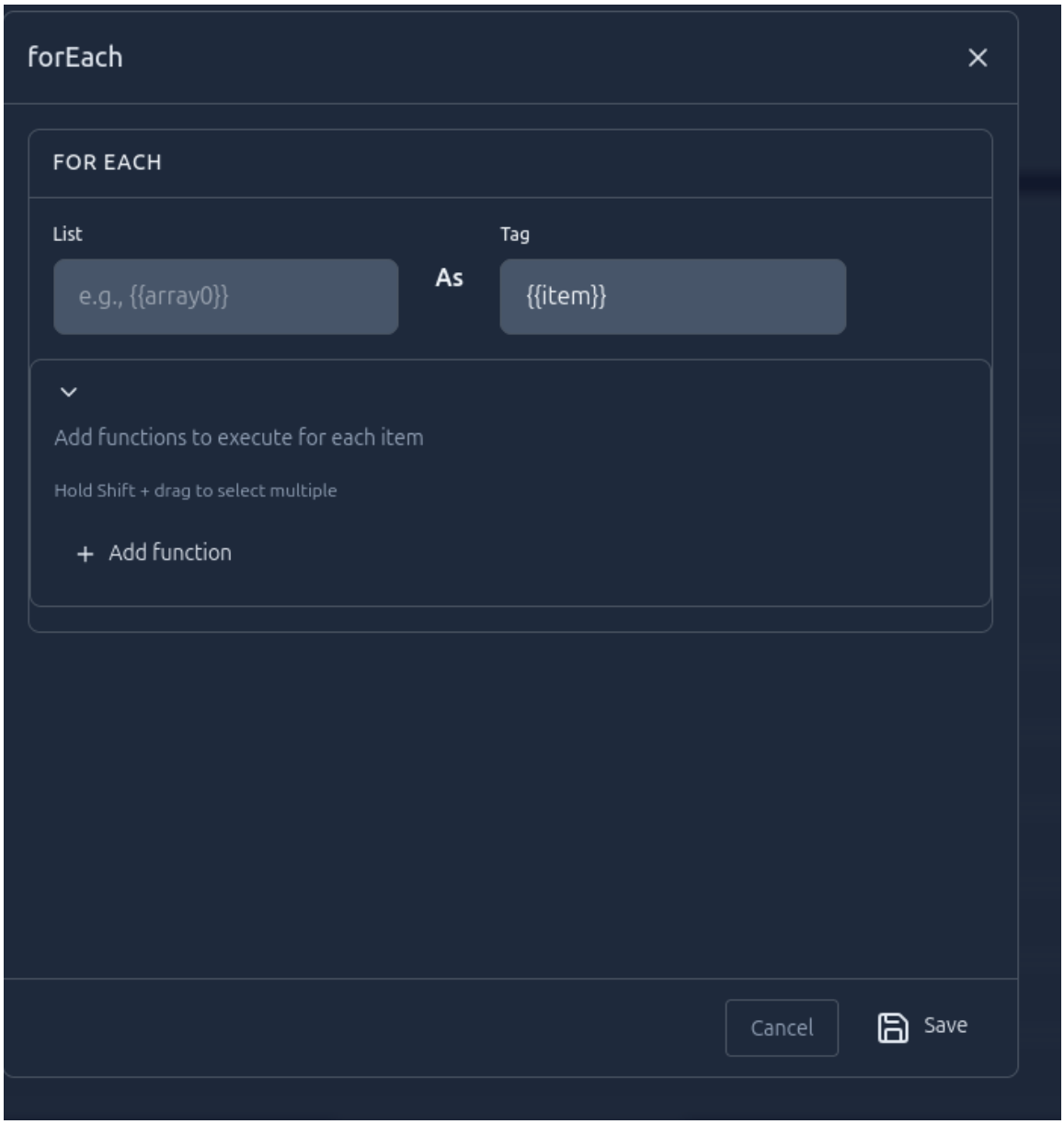


Figure 4.7: Foreach Configuration Modal

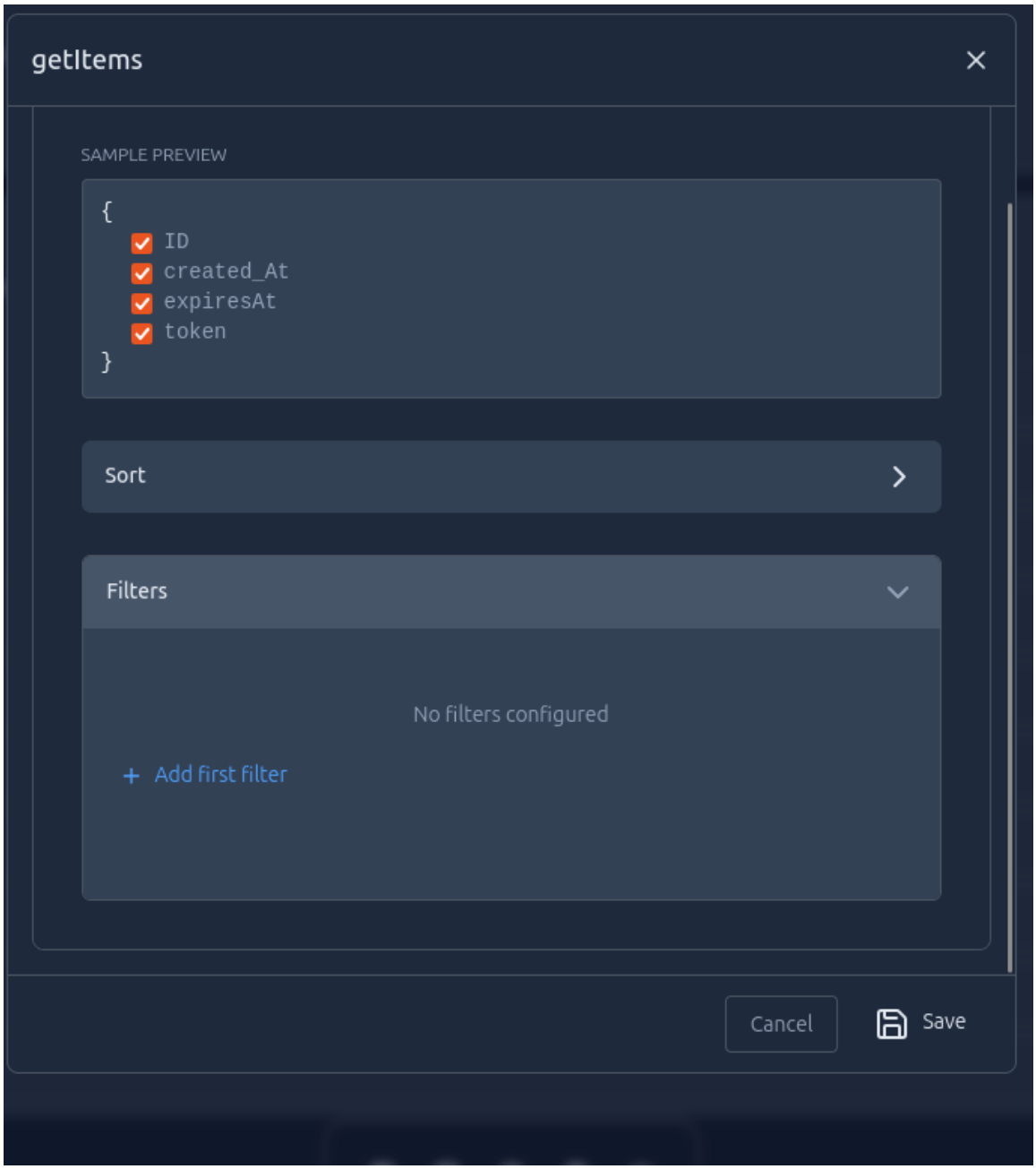


Figure 4.8: GetITem Configuration Modal

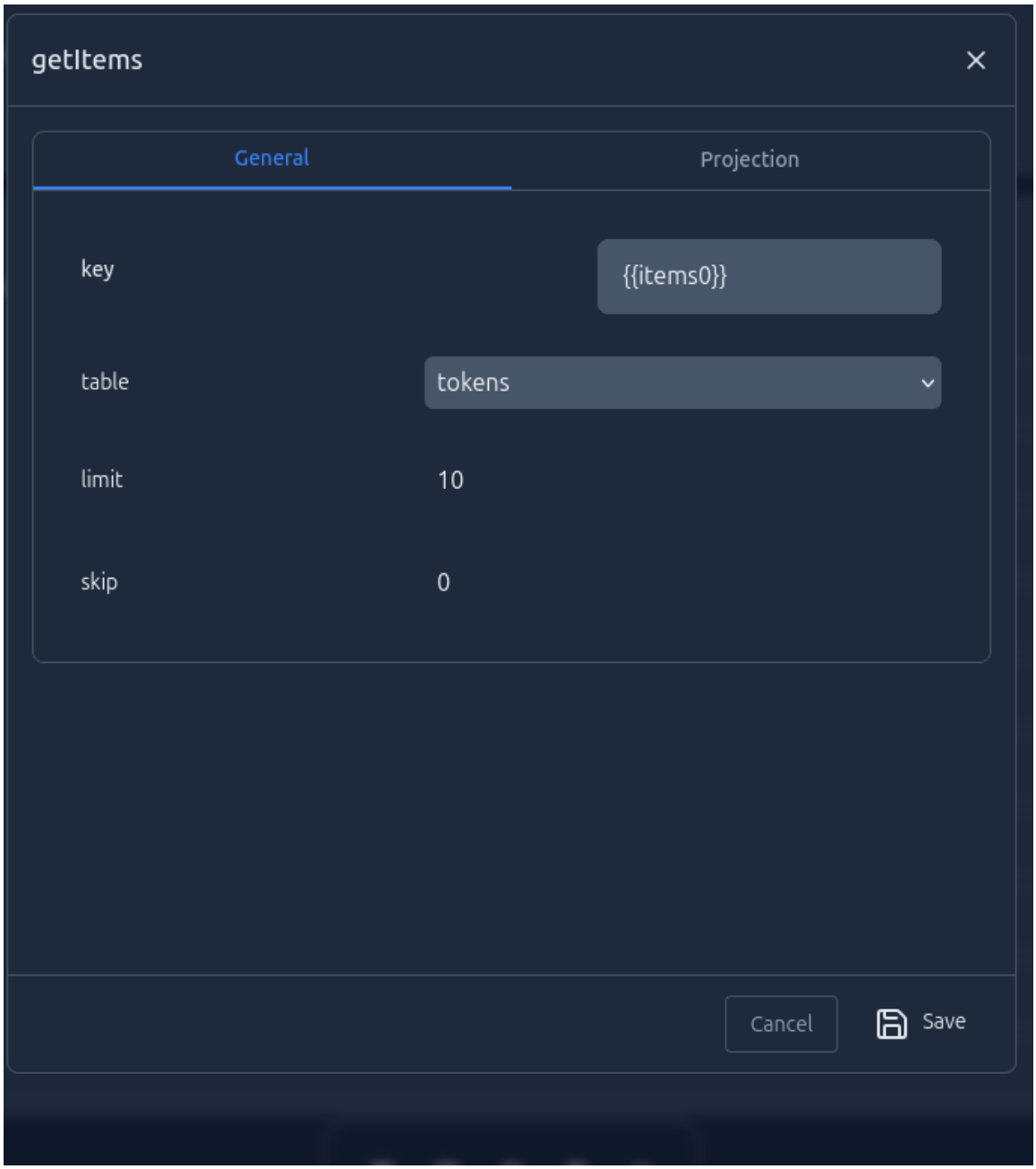


Figure 4.9: GetITem Configuration Modal

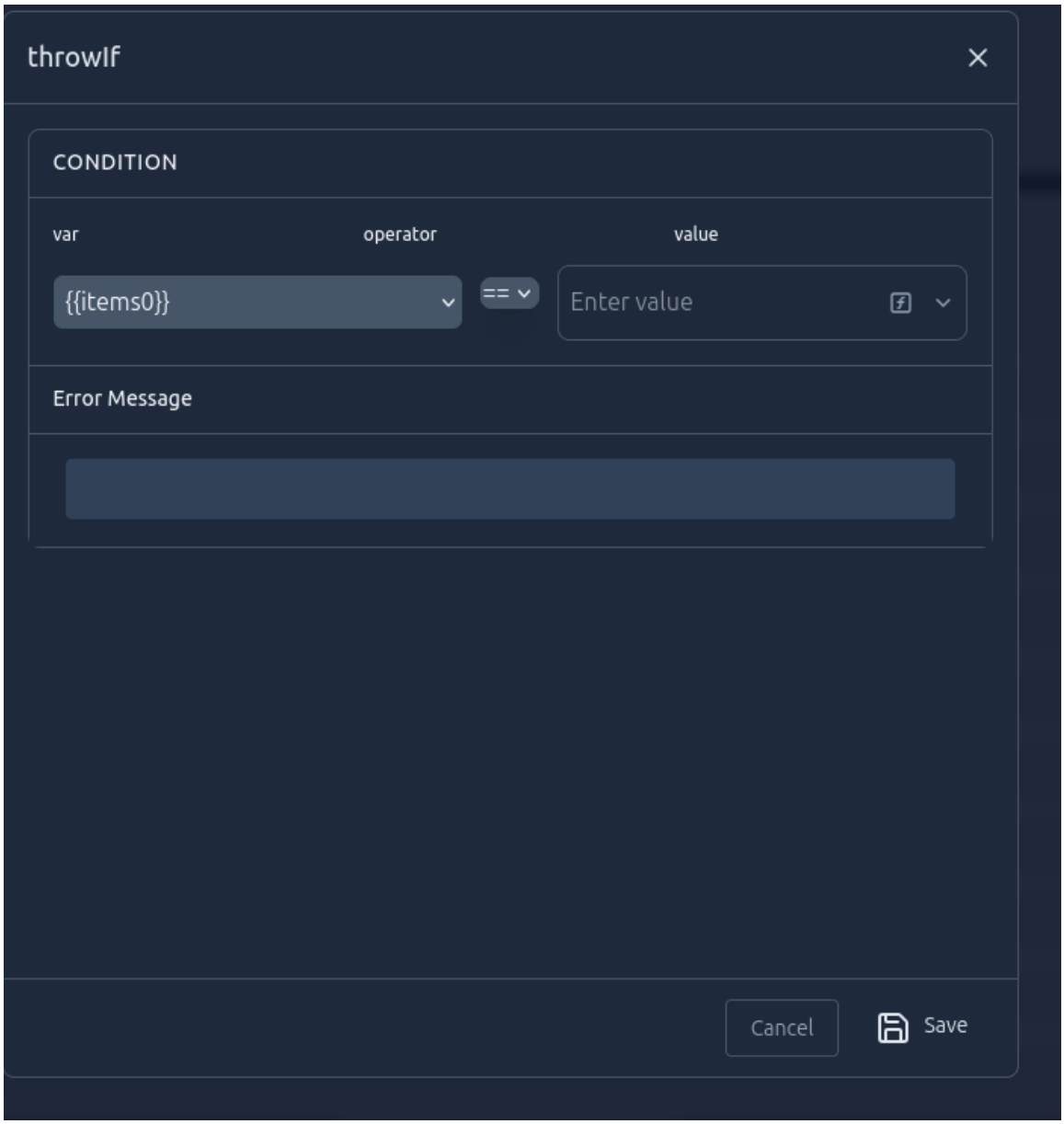


Figure 4.10: ThrowIf Configuration Modal

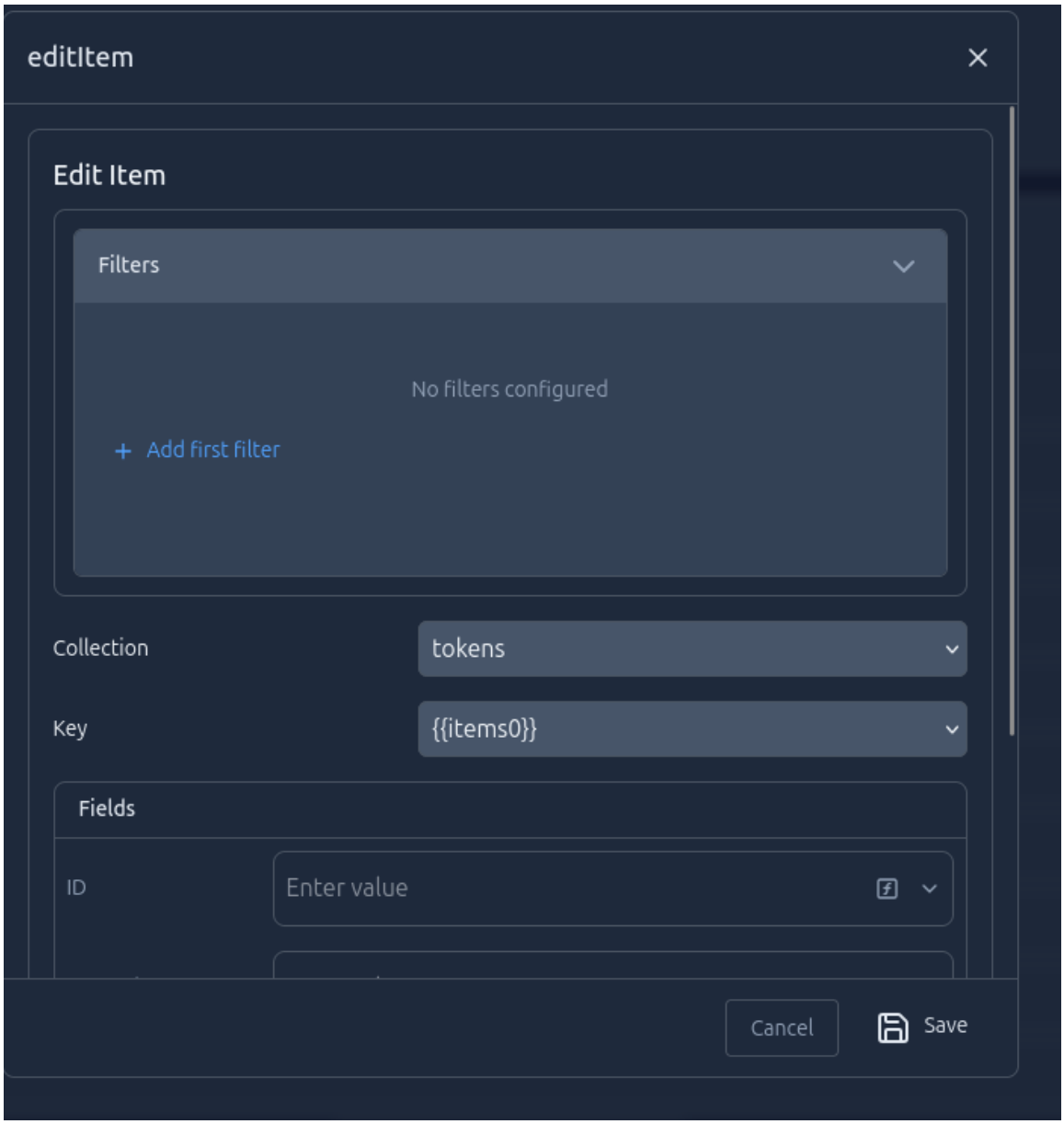


Figure 4.11: EditITem Configuration Modal

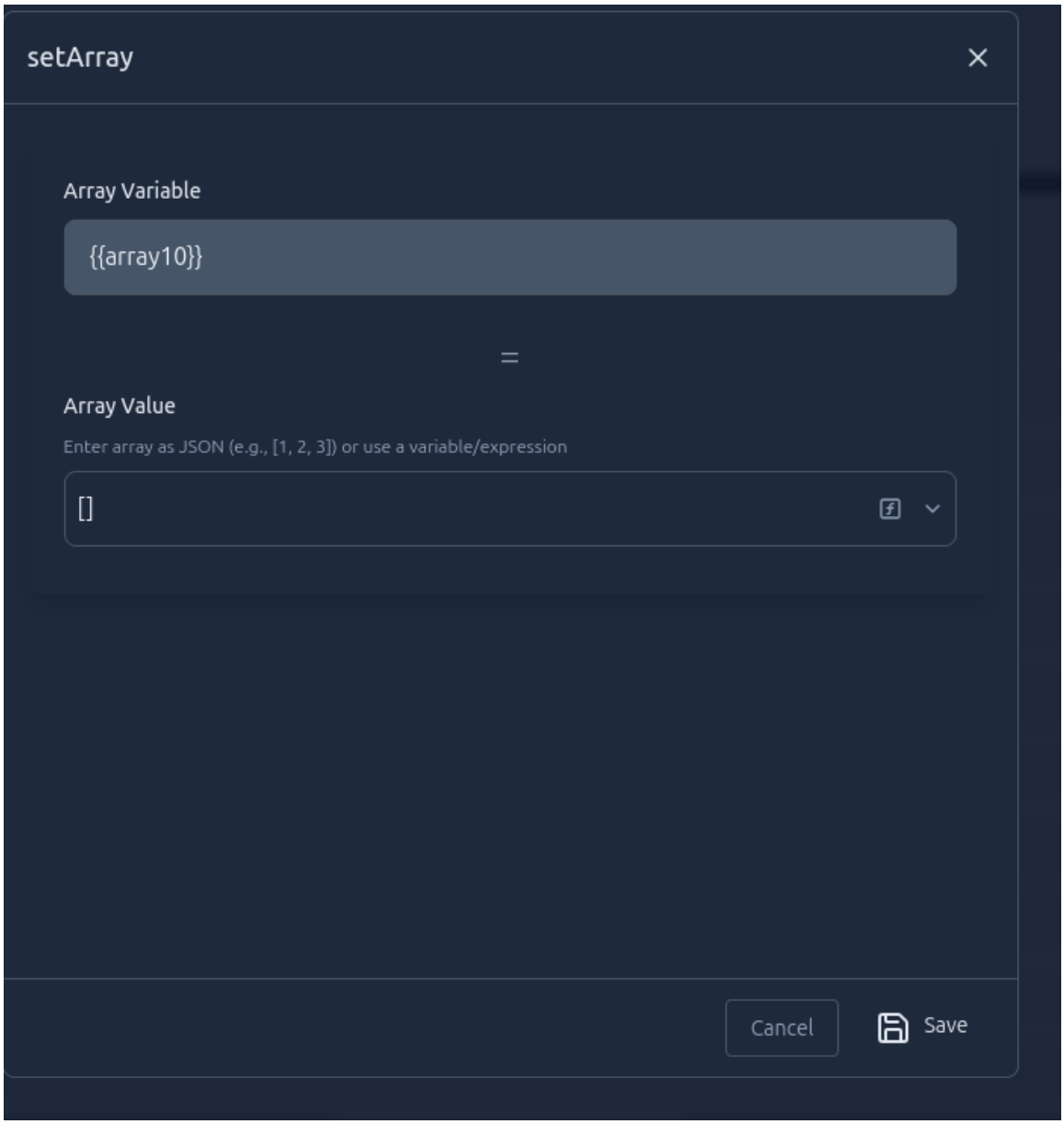


Figure 4.12: SetArray Configuration Modal

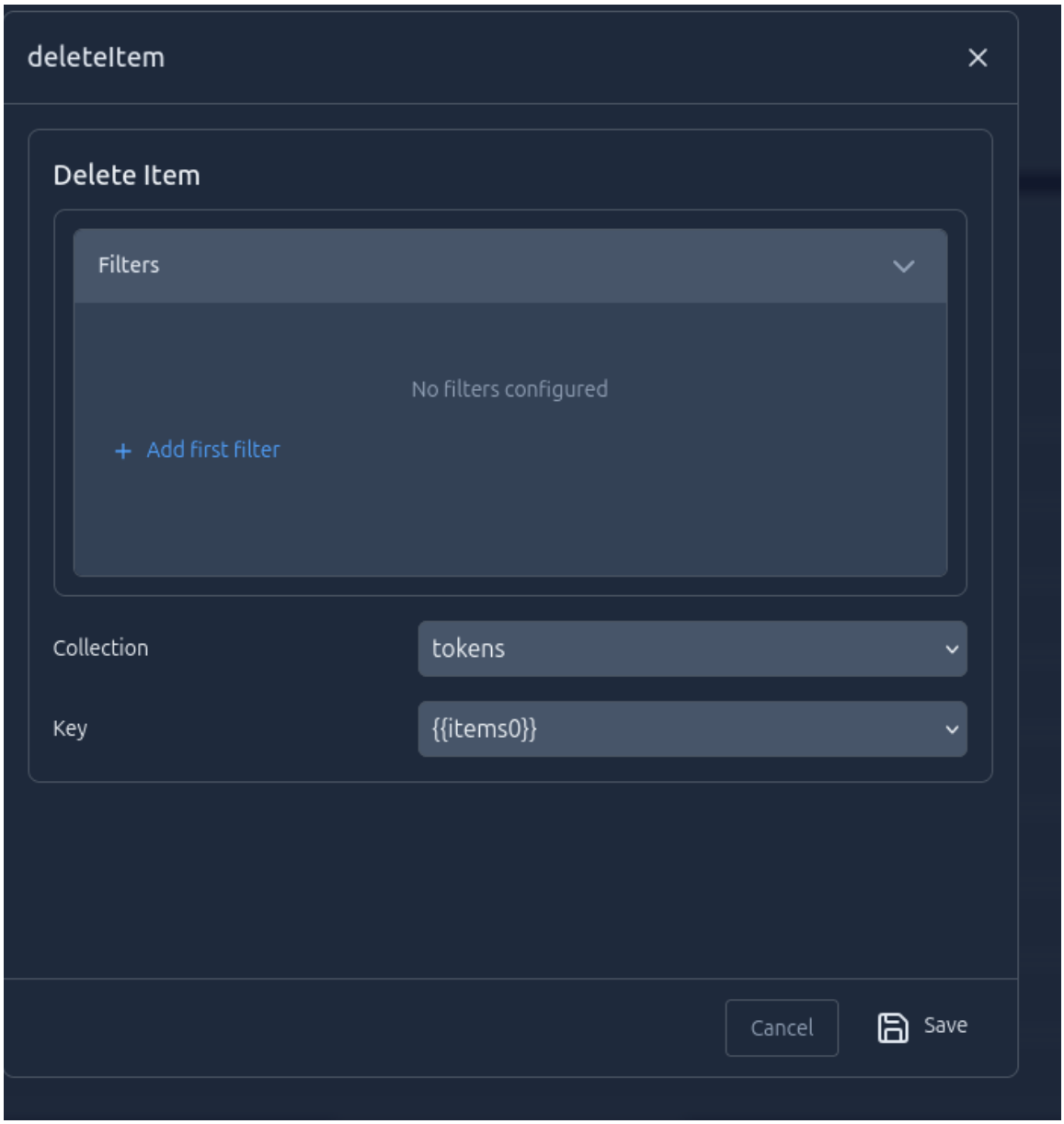


Figure 4.13: DeleteItem Configuration Modal

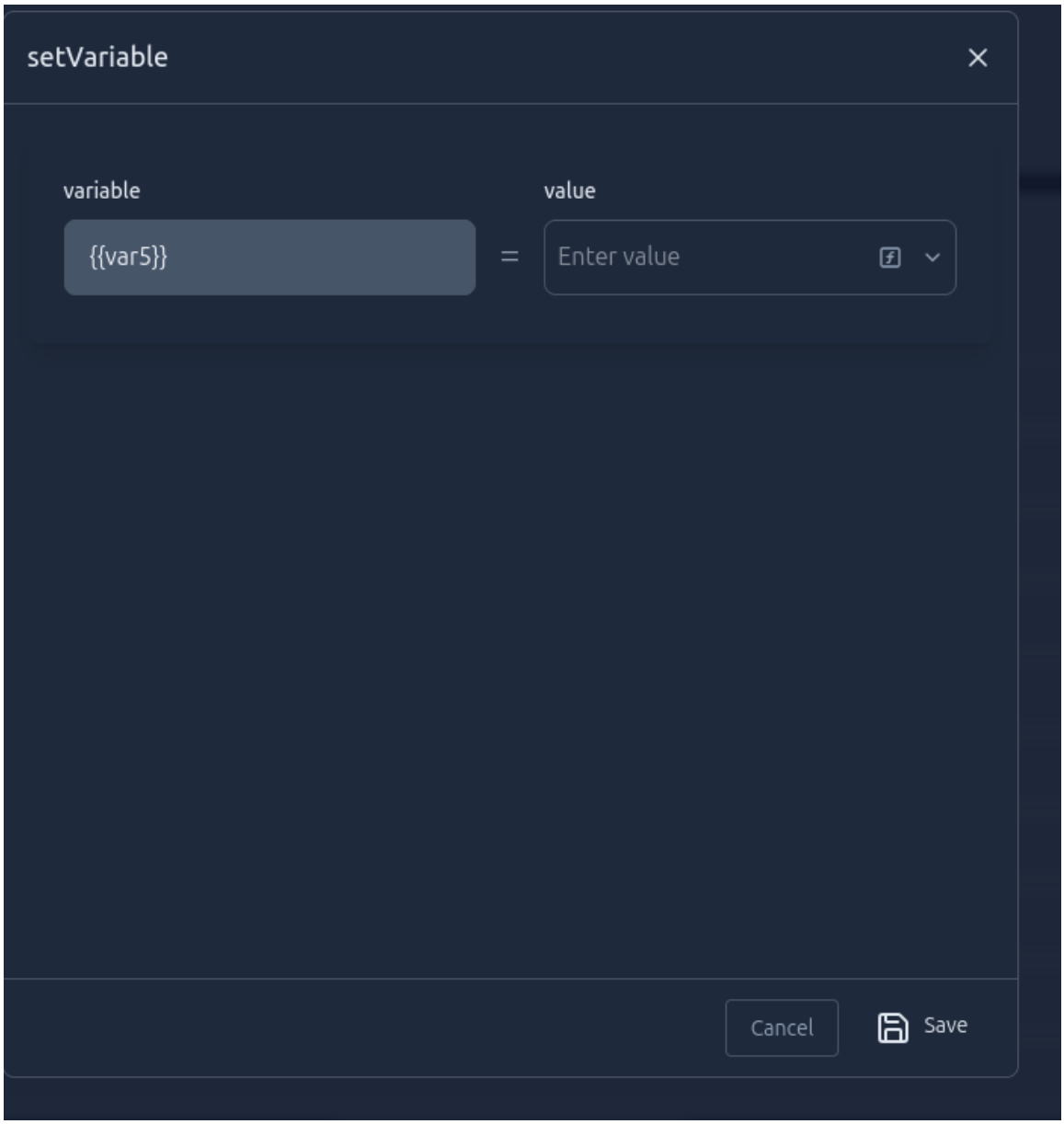


Figure 4.14: SetVarible Configuration Modal

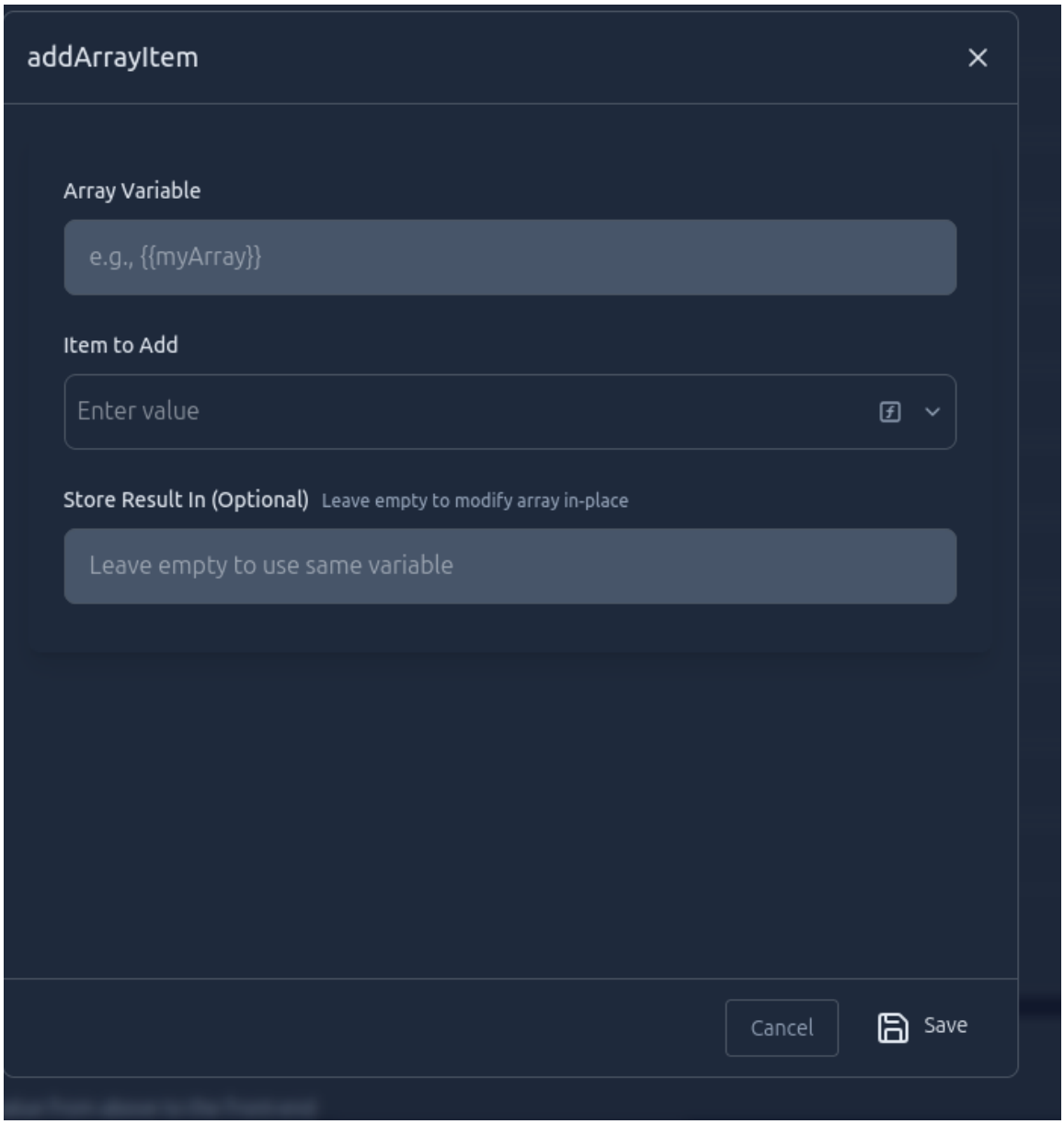


Figure 4.15: AddArrayItem Configuration Modal

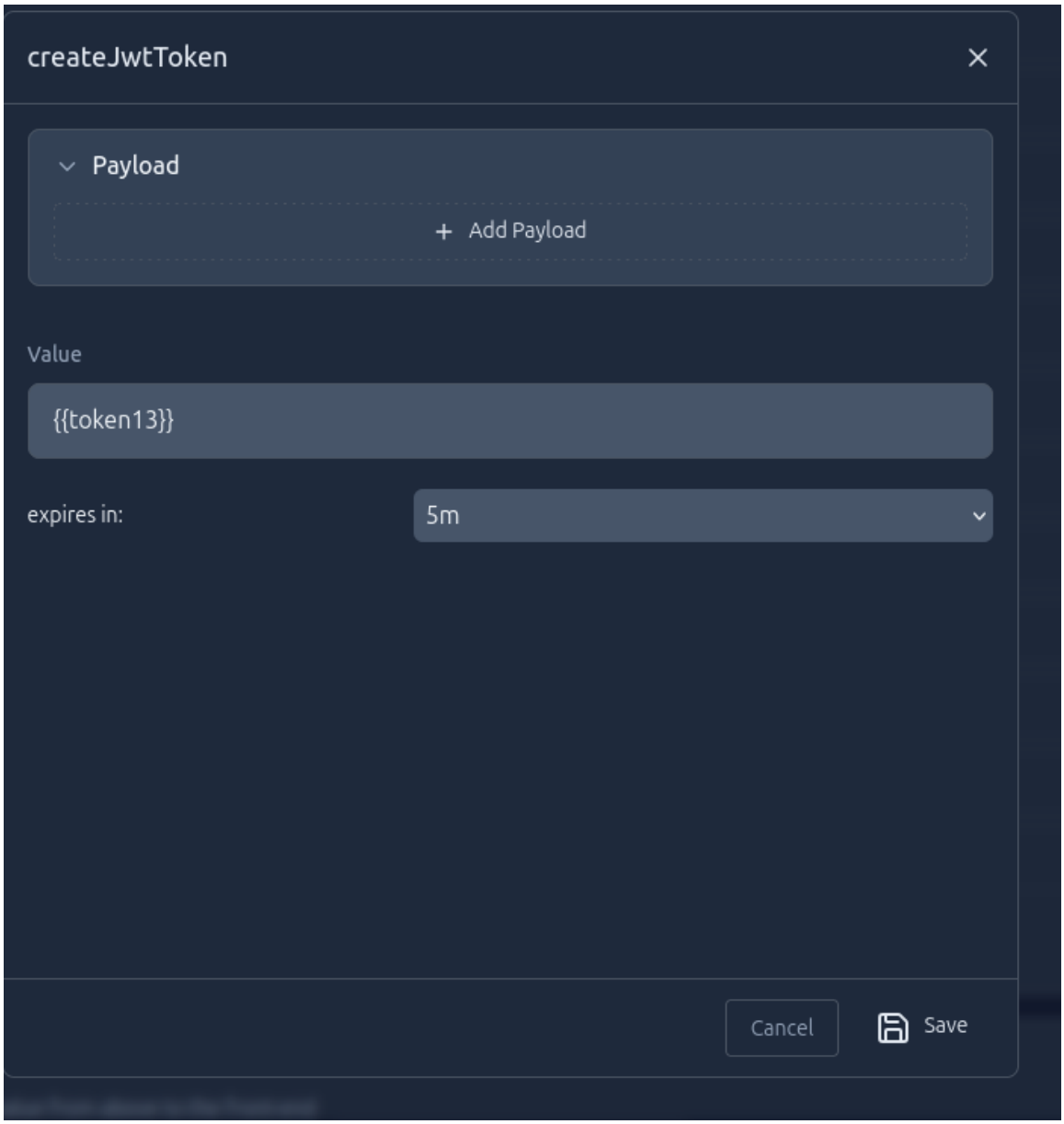


Figure 4.16: CreteJWTToken Configuration Modal

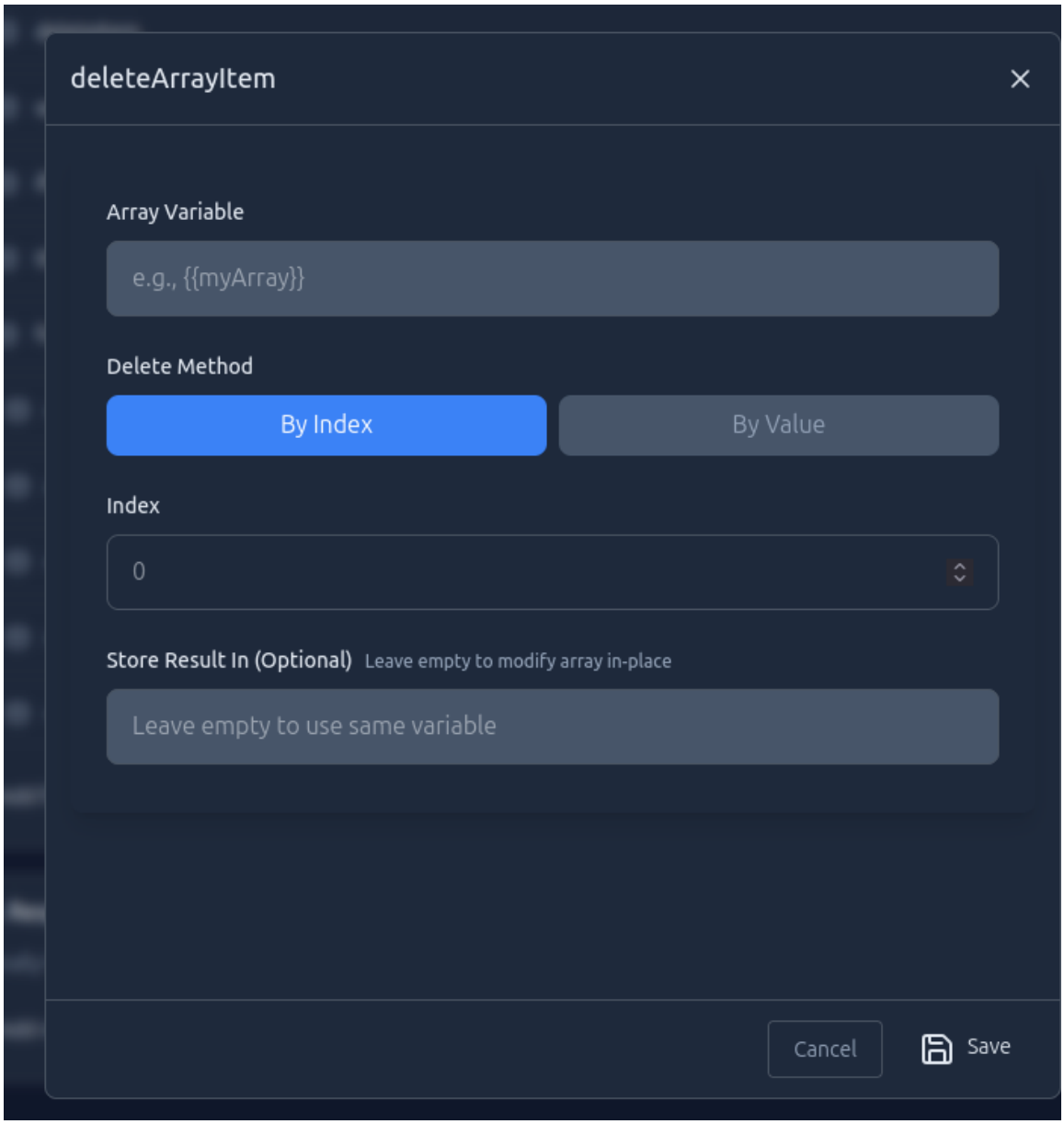


Figure 4.17: DeleteArrayItem Configuration Modal

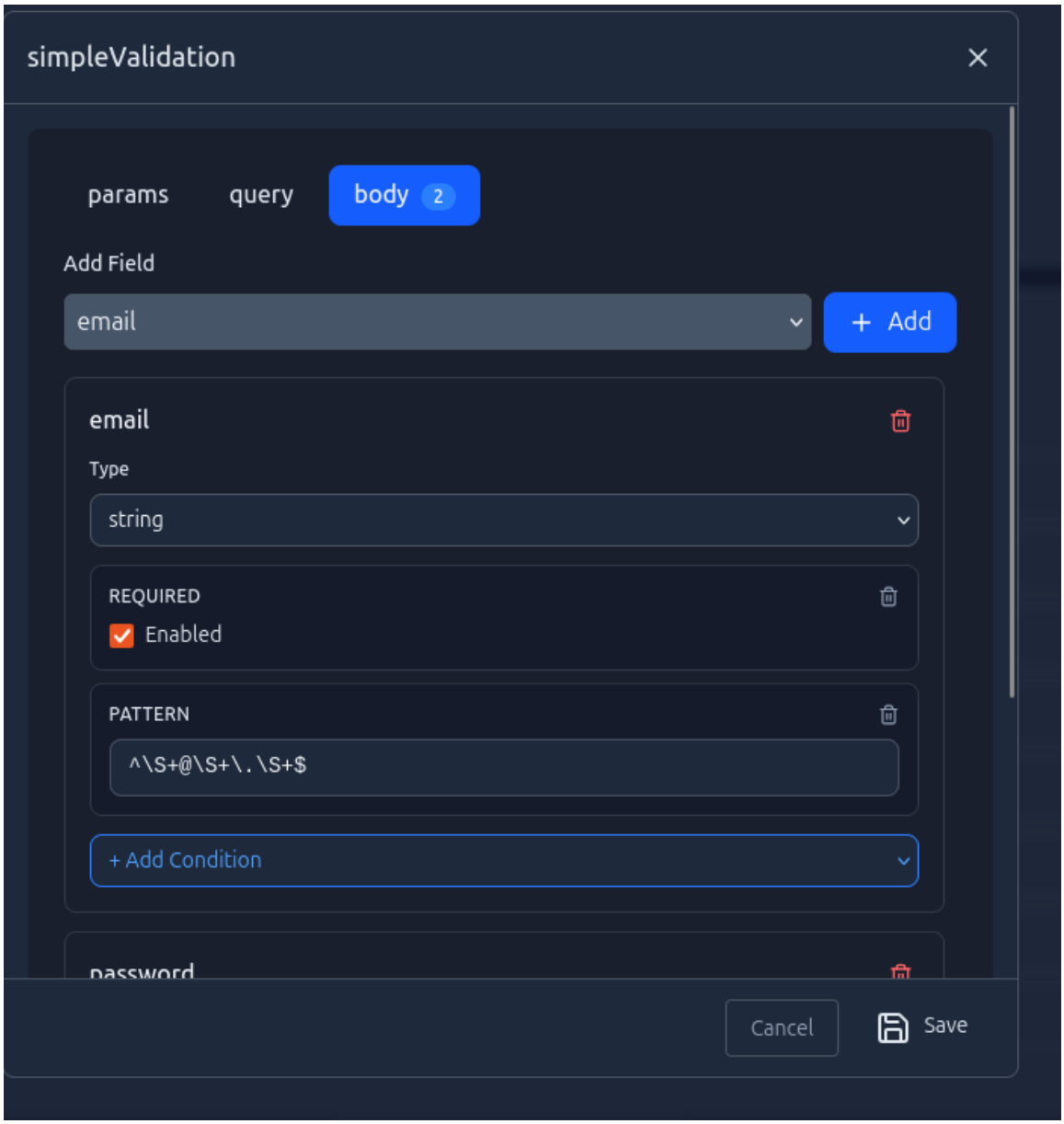


Figure 4.18: SimpleValidation Configuration Modal

The configuration interface includes variable autocomplete that suggests available variables based on the current position in the workflow. This helps users understand what data is available and ensures correct variable references. The interface also provides type hints and validation feedback, making it clear when configurations are invalid and how to fix them.

4.2.5 AI-Powered Workflow Generation

The platform includes an AI chat assistant that can generate workflows from natural language descriptions. Users describe what they want their API to do in plain English, and the AI generates an appropriate function stack. The chat interface provides a conversation-style interaction that feels natural and approachable.

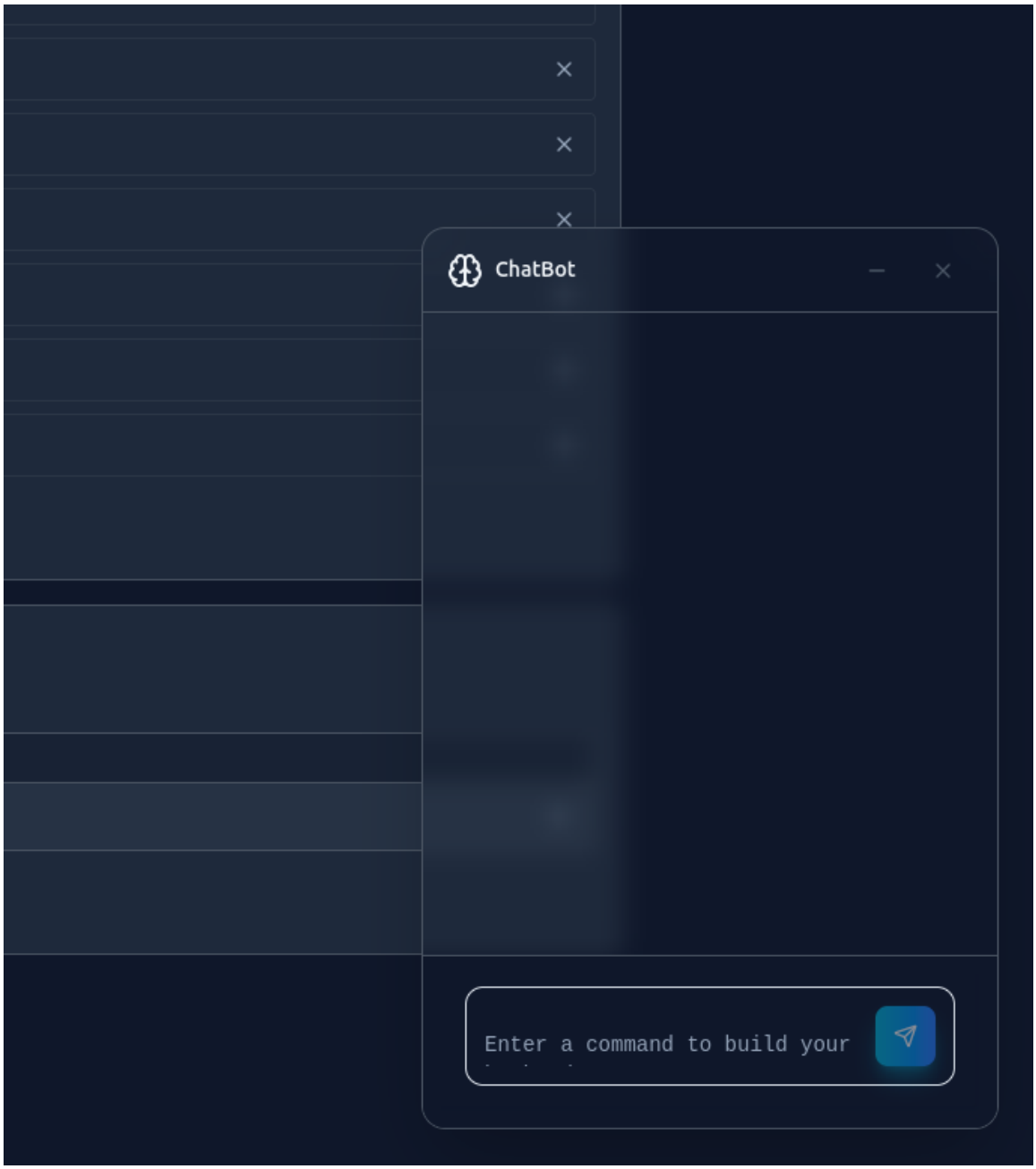


Figure 4.19: AI-Powered Workflow Generation
[Figure placeholder: AI chat interface showing natural language input, AI responses, and generated workflow application]

The AI assistant understands user requirements, queries available functions and database schemas, then constructs appropriate workflows. Generated workflows are automatically applied to endpoints, and users can see results in the workflow builder. Users can modify generated workflows as needed, combining AI assistance with manual control.

4.2.6 Marketplace Interface

Users can browse, search, and purchase APIs from the marketplace. The marketplace interface displays available APIs with descriptions, pricing, tags, and features. Users can search by name or description, filter by tags or price, and sort by popularity or date. The

interface provides detailed views of each listing, showing features, pricing, and purchase options.

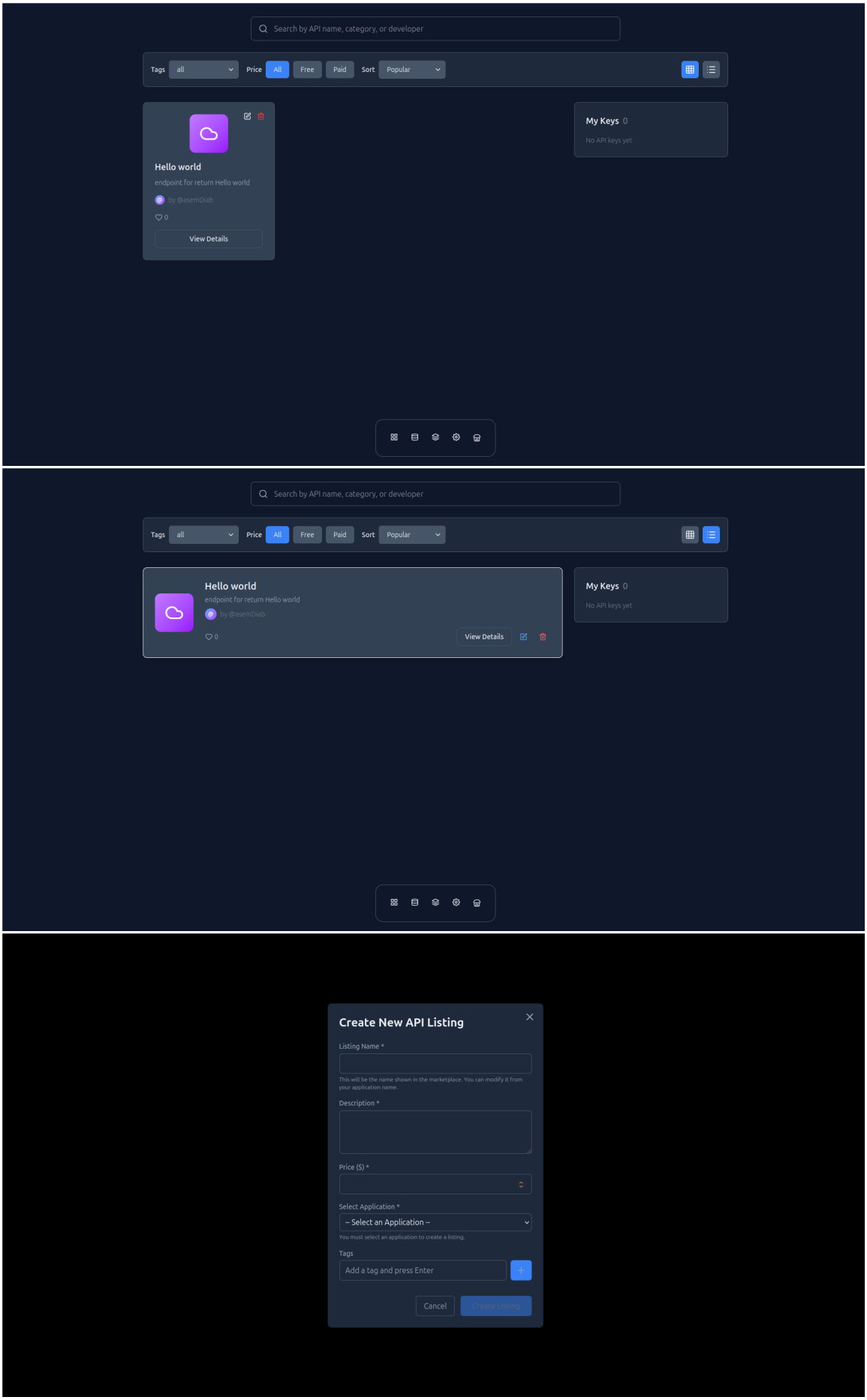


Figure 4.20: Marketplace Interface

The marketplace includes social features such as likes and favorites, enabling users to discover popular APIs. Users can view their purchased APIs and access API keys for purchased applications. The purchase flow integrates with Stripe for secure payment processing, and purchasers receive API keys automatically after purchase.

4.3 System Capabilities and Features

4.3.1 Multi-Tenant Architecture

FlowAPI successfully implements a multi-tenant architecture with database-level isolation. Each application has its own MongoDB database, identified by the application ID. This ensures complete data isolation between applications, meaning that even if there is a security breach or bug in one application, it cannot affect data belonging to other applications.

The system handles connection management efficiently through connection pooling and caching. When a request arrives for a specific application, the DB-Backend service checks if a connection exists for that application's database. If a connection exists, it is reused, optimizing performance. If not, a new connection is created and cached for future use. This approach balances performance with resource efficiency.

4.3.2 Function Stack System

The function stack system provides comprehensive functionality for API development. Database operations include full CRUD operations with filtering, sorting, pagination, and reference expansion. The getItems function supports complex MongoDB queries with aggregation pipelines, enabling sophisticated data retrieval. Reference expansion automatically joins related data, making it easy to work with relationships.

Data manipulation functions enable variable management, array operations, and data transformation. Users can set variables, initialize arrays, add and remove array items, and perform various data transformations. These functions enable complex data processing without requiring code.

Control flow functions provide conditional logic and loops. The ifElse function enables branching based on conditions, supporting various comparison operators. The forEach function enables iteration over arrays, executing nested functions for each item. These functions enable users to implement complex business logic visually.

Validation functions ensure data integrity and handle errors. The simpleValidation function validates and normalizes request inputs with comprehensive validation rules. The throwIf function enables conditional error throwing, allowing users to implement business rule validation.

Security functions enable authentication. The createJwtToken function generates JWT tokens with configurable payload and expiration, enabling users to implement authentication in their APIs. This function uses the application's secret key to sign tokens, ensuring that tokens are valid only for that application.

4.3.3 Placeholder Resolution

The placeholder resolution system enables dynamic data flow between functions, supporting variable references from previous functions, request data access (params, query,

body), nested property access, and expression evaluation for calculations. This system provides the flexibility needed for complex business logic while maintaining simplicity in the visual interface.

The resolution process handles various placeholder types, including simple variable references, nested property access, and expressions. The system resolves placeholders recursively, handling nested objects and arrays. This enables functions to work with complex data structures without requiring additional data manipulation.

4.3.4 AI Workflow Generation

The AI server successfully generates workflows from natural language descriptions. The system understands user requirements through LLM processing, queries available functions and database schemas, constructs appropriate function stacks, validates generated blocks before adding them, and handles errors gracefully with safety measures.

The AI assistant makes API development accessible to non-technical users by allowing them to describe their requirements in plain English. The generated workflows are automatically applied to endpoints, and users can see the result immediately. Users can then modify the generated workflow if needed, combining AI assistance with manual control.

4.4 Performance Analysis

4.4.1 Response Times

The system demonstrates good performance characteristics across different operations. API endpoint execution typically responds in under 500ms for simple endpoints, with more complex endpoints taking longer depending on the number of functions and database operations. The sequential execution of functions maintains predictable performance, as execution time scales linearly with the number of functions.

Database operations are efficient through MongoDB aggregation pipelines, which are highly optimized by MongoDB. Queries with filtering, sorting, and pagination perform well even with large datasets. The connection pooling and caching mechanisms ensure that database operations don't suffer from connection overhead.

Function stack execution maintains predictable performance due to sequential execution. While this prevents parallelism, it ensures execution time is predictable and functions can depend on previous outputs. The placeholder resolution system is optimized through caching during execution, avoiding repeated resolution.

AI workflow generation times vary based on complexity, typically taking 10-30 seconds. Simple workflows with few functions generate quickly, while complex workflows with many functions and nested logic take longer. The system includes timeout protection to ensure that generation doesn't take too long, and recursion limits prevent infinite loops.

4.4.2 Scalability

The microservices architecture enables horizontal scaling, where additional instances of services can be added to handle increased load. All services are designed to be stateless, meaning they do not maintain session state or in-memory data that would prevent multiple instances from handling requests. This stateless design allows load balancers to distribute requests across multiple service instances.

MongoDB supports horizontal scaling through sharding, where data is distributed across multiple servers. Each application's database can be scaled independently, allowing for fine-grained resource allocation. Applications with high data volumes can be scaled separately from those with lower volumes, optimizing resource usage.

Connection pooling and caching enable efficient resource utilization. Database connections are pooled and cached, reducing overhead and enabling efficient resource utilization. The architecture supports load balancing at the API gateway level, distributing requests across multiple service instances. The stateless design of services ensures that any instance can handle any request, making load balancing straightforward.

4.5 Limitations and Future Work

4.5.1 Current Limitations

The system has some limitations that could be addressed in future work. The array-based record storage structure is limited by MongoDB's 16MB document size limit, which constrains the number of records per table. For very large tables, the system would need to restructure to a document-per-record format, where each record is stored as a separate document.

Functions execute sequentially, limiting parallelism. While this ensures predictable execution and enables data flow between functions, it means that independent functions cannot execute in parallel. This could be addressed by analyzing function dependencies and executing independent functions in parallel.

AI generation time can be significant for complex workflows, typically taking 10-30 seconds. While this is acceptable for workflow generation, it could be improved through better prompt engineering, model optimization, or caching of common patterns. The system includes timeout protection to ensure that generation doesn't take too long.

Error recovery mechanisms are limited in function execution. If a function fails, execution stops and an error is returned. While this prevents incorrect results, it doesn't provide mechanisms for retrying failed operations or recovering from transient errors. This could be addressed through retry logic and error recovery strategy.

4.5.2 Future Enhancements

Potential improvements include restructuring large tables to document-per-record format for better scalability. This would enable efficient pagination and better performance for large datasets. The system could automatically migrate tables when they exceed a certain size, or provide a hybrid approach that uses arrays for small tables and documents for large tables.

Parallel function execution could be implemented by analyzing function dependencies and executing independent functions in parallel. This would improve performance for workflows with independent functions, while maintaining sequential execution for dependent functions. The system would need to analyze the function stack to identify dependency and determine which functions can execute in parallel.

Advanced AI features could include workflow optimization, where the AI suggests improvements for existing workflows. The AI could analyze workflows and suggest combining functions, optimizing execution order, or simplifying logic. Error explanation could help users understand errors in workflows and suggest fixes, making the system more helpful.

Real-time collaboration could enable multiple users to edit workflows simultaneously, with changes synchronized in real-time. This would enable effective team collaboration on API development. Version control could provide workflow versioning and rollback capabilities, enabling users to track changes and revert to previous versions if needed.

Advanced analytics could provide enhanced usage analytics and performance monitoring. The system could track detailed metrics about API usage, function execution times, and error rates. This would enable users to understand how their APIs are being used and identify optimization opportunities.

Chapter 5

Conclusion

5.1 Summary of Contributions

FlowAPI makes an important contribution to the field of no-code API development. It show that visual tools could be both easy to use and powerful. The platform makes users build complex APIs without writing code by combining an interface with strong functionality. Its multi-tenant architecture, function stack system, and AI-powered workflow generation solve key problems in existing API development tools.

The microservices design ensures that the platform is scalable and maintainable. The visual workflow builder allow non-technical users to create APIs easily, while AI assistance make users describe their needs in plain language and get working API implementations. With features like database management, workflow execution, AI support, marketplace, and team collaboration, FlowAPI supports a complete solution for building APIs.

5.2 Achievements

FlowAPI meet its main goals of offering a full-featured visual API development platform. Users can create, configure, and deploy RESTful APIs using a drag-and-drop ui, without needing to writing code. Integrated database tools let users manage tables, define schemas, handle records, and set up relationships without directly accessing the database.

The workflow engine make sequential execution of functions, conditional logic, loops, and dynamic data handling, enabling complex business logic without needing to write code. The AI assistant help generate workflows from natural language descriptions, making the platform accessible to users without technical backgrounds. Multi-tenant database isolation make sure security.

The marketplace feature makes users to share, find, and buy APIs, encouraging collaboration and innovation. Team features with role-based access let multiple developers work on projects together. Overall, FlowAPI combines these capabilities to support both individual developers and teams.

5.3 Impact and Significance

FlowAPI has the potential to change how APIs are developed and deployed. By making API creation accessible to non-technical users, organizations can respond faster to business needs. The visual interface reduces the time.

The platform can be applied across many fields like e-commerce, healthcare, education, and enterprise software, where fast API development and integration are essential. The marketplace encourages innovation by allowing users to share and buy APIs.

Multi-tenant design makes the platform to scale to many users while maintaining security and isolation. Microservices architecture adds flexibility, making it easier for the platform to evolve with future requirements.

5.4 Future Work

While FlowAPI shows that visual API development is useful, there are ways to improve it. Performance can be enhanced by running independent functions in parallel, using document per record storage for large tables, or implementing advanced caching strategies.

AI features can be expanded to suggest workflow optimizations, explain errors, or generate documentation automatically. This would make the AI assistant even more helpful.

Collaboration could be improved with real-time editing, workflow version control, and rollback capabilities. Analytics could provide detailed insights into API usage and performance. These enhancements would make the platform more collaborative and informative.

5.5 Final Remarks

FlowAPI successfully bridge the gap between complex development frameworks and limited low-code tools, offering a platform that is both accessible and powerful. With visual development, database management, AI support, and collaborative features, it provides a complete solution for API creation.

The platform prove that visual tools can handle complex applications while remaining easy for non-programmers. By making API development more accessible, FlowAPI can help organizations innovate faster and respond more effectively to changing requirements.

Overall, this work show that no-code platforms can be practical for serious software development. FlowAPI lays the groundwork for future research in visual programming and no-code tools, highlighting their potential to simplify and accelerate complex software development tasks.

Appendix A

Project Management

This appendix describes the project management approach, development process, and supporting practices used in the development of the FlowAPI platform.

A.1 Development Methodology

The project followed an Agile-inspired development methodology based on iterative implementation cycles. Development activities were organized into short phases with regular reviews to evaluate progress and adjust priorities when necessary.

Features were implemented incrementally, allowing core components to be developed and validated before additional functionality was added. Code reviews and structured architectural patterns were used to maintain consistency and readability across the code base.

A.2 Project Timeline

The FlowAPI platform was developed within an accelerated execution period of approximately four months. Development activities were carried out in parallel where possible to meet time constraints:

- **Phase 1 – Frontend and Database Management:** Development of the web frontend, including the database management interface, table creation pages, and basic CRUD operations. This phase focused on building the user interface and connecting it to the database services.
- **Phase 2 – Workflow System Development:** Implementation of the workflow engine and workflow execution logic. This phase included defining workflow steps, function stacks, and connecting workflows to API endpoints.
- **Phase 3 – Mobile Application and Backend Services:** Development of the mobile application and its backend using C#. This phase focused on exposing APIs, handling user interactions from the mobile app, and ensuring proper communication with existing services.
- **Phase 4 – AI Integration:** Integration of AI services to support workflow generation and automation. This phase added natural language input processing and AI-assisted workflow creation.

A.3 Technology Stack

The platform was implemented using a multi-service architecture with the following technologies:

- **Backend:** C# .NET 9.0 following Clean Architecture principles with CQRS and MediatR
- **Workflow Engine:** Node.js, Express, MongoDB
- **AI Services:** Python FastAPI, LangChain, OpenAI,
- **Frontend:** React Native with Expo (mobile application) and React js for web page.
- **Databases:** MongoDB for workflow execution and state management, SQL Server as the primary relational database

A.4 Testing Approach

Testing activities were limited in scope due to the accelerated development timeline and focused primarily on validating core functionality:

- Manual testing of essential user workflows
- Basic verification of API endpoint behavior
- Manual validation of service interactions across backend components
- Automated testing was minimal and not comprehensively implemented

A.5 Tools and Practices

The following tools and practices supported development and collaboration:

- Git and GitHub for version control and source code management
- Issue tracking tools for organizing development tasks
- Project-level documentation through README.md files
- Team communication via Discord
- Docker and Docker-Compose for containerization and environment consistency

A.6 Code Quality and Standards

Code quality was maintained through structured design and consistent development practices:

- Clear separation of concerns using Clean Architecture layers
- Use of CQRS and MediatR to organize application logic
- Centralized error handling and structured logging
- Inline code comments and documentation to support maintainability

A.7 Deployment and DevOps

Deployment relied on container-based practices to ensure consistent execution across environments:

- Docker used to containerize all services
- Docker Compose used for local development and service orchestration
- Environment variables used for configuration management
- Database migrations applied automatically during service startup

A.8 Challenges and Risk Mitigation

The primary challenges during development included integration complexity between multiple services, limited time for comprehensive testing, and coordination across different technology stacks.

These challenges were mitigated by prioritizing core platform functionality, maintaining clear API boundaries between services, and focusing testing efforts on critical execution paths. Containerization also helped reduce deployment-related risks by ensuring consistent runtime environments.

Appendix B

Source Code Repositories

The FlowAPI platform is divided into several repositories . both repository has a clear role and focuses on a specific part of the system . This part explains the purpose of each repository and the technologies used .

B.1 Repository Structure

The FlowAPI-Backend repository contains the basic backend service built with ASP.NET Core . This service is responsible for authentication and authorization, managing applications , API endpoints , API groups , and marketplace features . It follows a Clean Architecture approach with the CQRS pattern . SQL Server is used for relational data through Entity Framework Core , while MongoDB is used for endpoint-related data . This service acts as the central backend service of the platform .

The FlowAPI-DB-Backend repository contains a Node.js and Express microservice that handles database operations. It manages multi-tenant MongoDB databases and supports table creation , schema updates , and basic CRUD operations . Its main role is to provide a dedicated API for database-related actions .

The FlowAPI-Workflow-Engine repository contains a Node.js and Express microservice that executes API workflows . It runs endpoints based on predefined function stacks and handles variable resolution , authentication checks , and endpoint matching . The service supports different function types such as database operations , data processing , control logic , validation , and security rules .

The FlowAPI-Frontend repository contains a React-based web application built using Vite. The project represents a frontend setup for the FlowAPI platform. It uses Redux Toolkit for state management and React Router for navigation. Multiple UI libraries are used for styling, including Material-UI, Ant Design, PrimeReact, and Tailwind CSS.

The ai-server repository contains a Python service built with FastAPI. This service generates workflows based on natural language input. It uses LangChain to manage interactions with large language models. The service supports tasks such as selecting workflow blocks, validating workflows, and generating execution steps. Safety mechanisms are included to limit execution and prevent invalid workflows.

B.2 System Architecture

The platform follows a microservices-based design. Each repository represents an independent service with a specific responsibility. Services communicate through well-defined APIs, allowing each component to evolve independently.

Docker is used to containerize services, which helps ensure consistent behavior across different environments. Environment configuration files are used to control service behavior depended on deployment requirements.

B.3 Key Technologies

The FlowAPI-Backend uses ASP.NET Core as the web framework. Entity Framework Core is used for database access, MediatR handles request processing, FluentValidation is used for input validation, AutoMapper manages object mapping, and Serilog is used for logging.

The FlowAPI-DB-Backend uses Node.js and Express for the server, Mongoose for MongoDB access, express-validator for request validation, and Helmet.js for basic security.

The FlowAPI-Workflow-Engine uses Node.js and Express, Mongoose for MongoDB access, and jsonwebtoken for handling authentication tokens.

The Software-GP-frontend uses React 19 with Vite for development and building. Redux Toolkit manages application state, React Router handles navigation, and multiple UI libraries are used to build the interface.

The ai-server uses Python and FastAPI, with LangChain for language model coordination. It integrates with external language models such as OpenAI and Google Gemini.

B.4 Documentation

Each repository contains basic documentation to support development. README.md files basically focus on setup instructions and how to run each service.

Backend services provide API documentation through Swagger, which allows developers to explore and test available endpoints. Inline comments are used in code to explain complex logic when demand.

The documentation focuses on practical usage and service setup rather than detailed development processes.

References

[1] B. A. Myers, "Taxonomies of visual programming and program visualization," *Journal of Visual Languages and Computing*, vol. 1, no. 1, pp. 97–123, 1990.

[2] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 171–178.

[3] Microsoft, "Microsoft power automate," 2024. [Online]. Available: https://powerautomate.microsoft.com/

[4] Zapier, "Zapier - automate your workflows," 2024. [Online]. Available: https://zapier.com/

[5] F. Chong, G. Carraro, and R. Wolter, "Multi-tenant data architecture," *Microsoft Corporation*, 2006. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/dn45199(v=vs.108)

[6] K. Inc., "Kong - api gateway and service mesh," 2024. [Online]. Available: https://konghq.com/

[7] G. Cloud, "Apigee - api management platform," 2024. [Online]. Available: https://cloud.google.com/apigee

[8] A. W. Services, "Amazon api gateway," 2024. [Online]. Available: https://aws.amazon.com/api-gateway/

[9] Backendless, "Backendless - backend as a service," 2024. [Online]. Available: https://backendless.com/

[10] Google, "Firebase - backend as a service," 2024. [Online]. Available: https://firebase.google.com/

[11] n8n, "n8n - open source workflow automation," 2024. [Online]. Available: https://n8n.io/