API-FIRST ARCHITECTURE: A MODERN APPROACH AND PRACTICAL IMPLEMENTATION IN SOFTWARE ENGINEERING*

TANER A. AHMEDOV

ABSTRACT: *This paper examines the transition from code-first software development to API-first architecture. It discusses bottlenecks in sequential workflows, in which frontend teams depend on completed backend components, and outlines a structured workflow based on the OpenAPI 3.0 specification, upfront contract definition, and automated API mocking. A qualitative comparison explores how these practices can support parallel development, reduce integration risks, and facilitate support for multiple client platforms. A service management example illustrates contract definition and mock server use. The discussion highlights the potential benefits of upfront contract design; it does not provide empirical measurements of development speed or integration defect rates.*

KEYWORDS: *API-first architecture, OpenAPI Specification, Swagger, software engineering, microservices, RESTful APIs, mock servers.*

DOI: https://doi.org/10.46687/ZQQU2261

1 Introduction

Software engineering continues to evolve in response to the need to develop scalable, cross-platform applications rapidly. The growth of cloud computing and microservices architectures, together with the proliferation of client applications across the web, iOS, Android, and the Internet of Things (IoT), has prompted a reassessment of traditional software delivery methods [1].

Traditional development approaches, often referred to as *code-first* or *implementation-first*, prioritize backend infrastructure and database logic. In this model, application programming interfaces (APIs) are often treated as implementation byproducts, developed alongside or after the application logic [2].

Although the *code-first* approach can suit small, monolithic projects with a single client interface, it can create coordination difficulties in larger or distributed engineering teams. Potential challenges include:

- Development delays caused by frontend teams depending on completed backend endpoints.
- Outdated or inconsistent API documentation.
- Data structure mismatches discovered late in system integration.

This paper outlines an **API-first development methodology** based on the **OpenAPI 3.0** specification and discusses its implications for the software development life cycle (SDLC).

*Partially supported by Scientific Research Grant DDVU 02/91 of MON and Scientific Research Grant RD-08-75/2016 of Shumen University.

2 Problem Analysis and Methodological Comparison

To assess the potential benefits of API-first development, it is useful to examine the limitations of sequential code-first workflows.

2.1 Traditional Workflow (Code-First)

In a sequential *code-first* workflow, backend engineers design the database schema and implement API endpoints before communicating interface definitions to frontend developers, sometimes through informal documentation or verbal agreements. This sequence can create integration bottlenecks when teams encounter mismatched data types or inconsistent assumptions about business logic.

2.2 API-First Approach

In contrast, the *API-first* approach treats the API as a **primary design artifact**. Before implementing the frontend or backend, cross-functional teams collaboratively define an **API contract** using the OpenAPI Specification [3]. Table 1 summarizes the main differences between the two approaches.

Table 1: Comparison of Code-First and API-First Development

Criterion	**Code-First**	**API-First**
Starting point	Implementation and data model	Interface contract
Development flow	Often sequential	Supports parallel work
Documentation	Derived from implementation	Derived from the contract
Team coordination	Client work often awaits endpoints	Client work can begin with mocks
Integration risks	Interface mismatches may emerge late	Contract checks can reveal mismatches earlier

3 Architectural Model and Workflow Integration

The proposed API-first workflow consists of four main phases:

1. **Contract design:** Domain models, resource paths, HTTP methods, and error schemas are defined in an `openapi.yaml` file.
2. **Mock server provisioning:** Automated mocking tools, such as Prism, use the specification to create local HTTP mock servers.
3. **Parallel development:** Frontend engineers build user interface (UI) components using mock endpoints, while backend engineers implement controller and persistence layers.
4. **Automated contract validation:** Continuous integration (CI) pipelines run contract tests to check backend responses against the API contract [4].

4 Practical Implementation and API Mocking

The following example illustrates how a service management API can be described using the OpenAPI 3.0 specification.

4.1 Specification Definition (OpenAPI 3.0)

Listing 1 presents an `openapi.yaml` snippet that defines a resource path and a reusable schema.

Listing 1: **Example OpenAPI 3.0 Specification Snippet**

```
openapi: 3.0.3
info:
  title: Service Management API
  version: 1.0.0
paths:
  /services:
    get:
      summary: Retrieve a list of available services
      responses:
        '200':
          description: A list of available services
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/ServiceItem'
components:
  schemas:
    ServiceItem:
      type: object
      required: [id, name, price]
      properties:
        id:
          type: integer
          example: 101
        name:
          type: string
          example: "Hardware␣Diagnostics"
        price:
          type: number
          format: float
          example: 45.00
```

4.2 Synthetic Endpoint Generation

The specification is passed to the Prism mocking engine through its command-line interface (CLI):

```
prism mock openapi.yaml
```

The engine reads the specification and starts a local mock server. For a valid GET request to the services endpoint in this example, it can return an HTTP 200 response with a payload that conforms to the specified schema.

5 Application in SaaS and Multi-Platform Ecosystems

Software-as-a-service (SaaS) systems often use a shared backend to serve multiple clients, including single-page applications, server-rendered web applications, native mobile applications, and business-to-business integrations. Using the OpenAPI specification as a shared interface definition supports the automated generation of TypeScript interfaces and data transfer objects (DTOs), reducing the need to synchronize these definitions manually.

6 Conclusion

This paper has outlined an **API-first development workflow** based on OpenAPI 3.0 and illustrated its use through a service management example. Upfront contract design and early mock server provisioning can reduce frontend dependencies on backend implementation and enable parallel development. Contract validation can help identify interface mismatches before integration. These benefits depend on maintaining alignment between the contract and the implementation; empirical evaluation is needed to quantify their effects on development speed and defect rates.

REFERENCES:

[1] M. Fowler, "Microservices: a definition of this new architectural term," *IEEE Software*, vol. 31, no. 3, pp. 87–92, 2014.

[2] A. Lauret, *The Design of Everyday APIs*. Manning Publications, 2019.

[3] SmartBear Software, "OpenAPI Specification v3.0.3," 2020. [Online]. Available: `https://spec.openapis.org/oai/v3.0.3`

[4] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly Media, 2021.

Taner Alimov Ahmedov:
Affiliation: Student / Sofware Engineering / University of Shumen
E-mail: taner@entaro.bg