Utilizing Mocking During the Development Phase

Bachelor's thesis

Information and Communication Technology

Autumn, Riihimäki,

2023

Chiya Hekmatyar

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES
Information and Communication Technology
Author Chiya Hekmatyar
Subject Utilising Mocking During the Development Phase
Supervisor Teemu Järvenpää

Abstract
Year 2023

The thesis introduces the utilization of a mock server and mocking during development phase. The research commenced with an exploration of company adopting mocking in development phase. The objective of the thesis is to outline the general advantages and disadvantages of employing mocks, both for companies and in the context of this project. The thesis delves into the Mock API and its corresponding server, which replicates the behavior of an actual API server by furnishing realistic API responses to incoming requests. It addresses the broader utilization of APIs and acquaints the reader with the diversity of APIs and their various types.

The physical interconnection between distinct API components as well as the interface between machines. The benefits of REST and its functionalities in terms of API utilization are scrutinized. Additionally, six requirements for utilizing mocks are explored, establishing the foundation for a rapid and adaptable API development process.

This thesis imparts information about mock tools and their applications—elaborating on their nature and rationale. In addition, the implementation of mock tools is described to highlight the easiness of integrating them into projects. Through demonstration, the aim is to illustrate the straightforwardness of initiating a project with the mock tool. The installation process of the mock tool and the subsequent integration of a personal project are detailed, with the intention of simplifying the process. The demonstration serves to underline the utility of mock tools and is supported by comprehensive documentation.

Keywords Mock, application interface, backend, API, REST

Pages 26 pages

Contents

1 Introduction 1
2 Understanding Mock servers in Software Development 2
2.1 Definition and Core Functionality 2
2.2 Creating Controlled Testing Environments 3
2.3 Accelerating Development Cycles 3
2.4 Supporting Parallel Development 3
2.5 Facilitating Continuous Integration and Testing 3
2.6 Addressing External Service Dependencies 4
3 What is API Mocking 4
3.1 Mocking During Development 4
3.2 Mocking For Functional Tests 5
3.3 Mocking for External Components 6
4 Using a mock in comparison to using a backend 6
4.1 Testing Environments and Scenarios 7
4.2 Development Speed and Iteration 7
4.3 Resource Utilization and Cost Efficiency 7
4.4 Realism and Simulation Accuracy 7
4.5 Adaptability to Changing Requirements 8
4.6 Summary 8
5 Application Programming Interface(API) 8
5.1 Open API 9
5.2 Partner API 9
5.3 Internal API 9
5.4 Composite API 9
6 REST application interface 10
7 REST API 10
7.1 How REST APIs work 10
7.2 Chararestics of REST APIs 11
7.3 REST API USAGE 12
7.4 Simple Object Access Protocol- ja Remote Procedure Call API 13

8 Mock tools 14
8.1 Existing tools for Mocking 15
8.1.1 Introducing Mock Tools 15
8.1.2 Postman 16
8.1.3 Spotlight 16
8.1.4 Mocky.io 17
8.1.5 MockServer 17
9 Mock demo 18
9.1 Mock creation 19
9.2 Editing mock and sending data 21
9.3 Demo summary 24
10 Conclusion 25
References 26

List of figures

Figure 1. Mock servers interaction (SoniAnshu, n.d.) 2

Figure 2. Mocking application during development instead real application (SoapUI, n.d.) 5

Figure 3. Mock testing with mock instead of real API (SoapUI, n.d.) 5

Figure 4. Mocking with external dependencies (SoapUI, n.d.) 6

Figure 5. Starting the mock server 19

Figure 6. Creating URL and response body 20

Figure 7. Mock name, environment, delay and privacy selection 21

Figure 8. URL created by Postman 21

Figure 9. Response body location..22

Figure 10. Response body filling..22

Figure 11. Choosing code language..23

Figure 12. Mock server body 23

List of Abbreviations

Mock	Simulated object that mimics the behavior of real objects in controlled ways, often as part of a software testing intiative.
API	Application Programming Interface. A programming interface through which different applications make requests and exchange information with each other.
HTTP	Hypertext Transfer Protocol. Protocol for the transfer of data between WWW servers and Internet browsers.
HTTPS	Hypertext Transfer Protocol Secure. Combination of HTTP traffic and TLS protocols for encrypted data transfer.
REST	Representational State Transfer describes the physical connection of the application interface between separate components.
RPC	Remote Procedure Call is a high-level communication protocol for operating system functions.
SOAP	Simple Object Access Protocol is the specification of a communication protocol for exchanging structured data in the implementation of web services over computer networks.

1 Introduction

In the dynamic landscape of contemporary software development, the need for efficient and reliable testing mechanisms has become paramount. As applications grow in complexity and interconnectivity, developers face the challenge of ensuring integration and functionality across various components. The advent of Mock servers has emerged as a critical solution to address these challenges by providing a simulated environment for testing and development.

A mock server, in the context of software development, serves as a virtual counterpart to real servers, enabling developers to mimic server responses and interactions. This simulation not only facilitates early-stage testing but also empowers developers to assess the behavior of their applications in diverse scenarios without relying on external dependencies. This thesis aims to delve into the multifaceted role of Mock servers in the software development life cycle, exploring their impact on testing methodologies, collaboration among development teams, and overall software quality.

Embarking on this exploration involves delving into the fundamental concepts that underpin Mock servers, examining their evolution and the rationale behind their adoption. The thesis will also investigate the use cases and scenarios where mock servers prove invaluable, shedding light on their efficacy in enhancing the efficiency of development workflows.

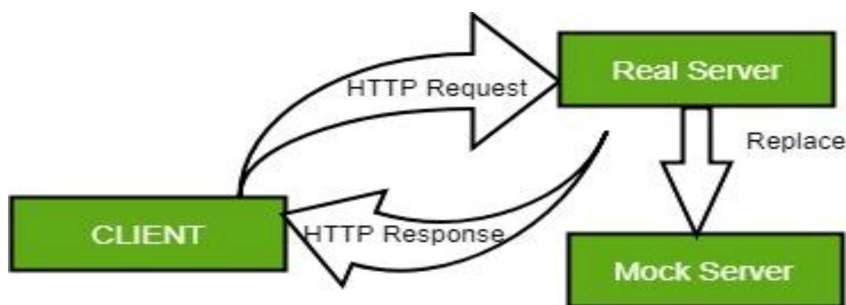
Through a comprehensive examination of the literature, case studies, and practical implementations, this thesis seeks to provide a nuanced understanding of the significance of mock servers in the modern software development landscape. By doing so, it aims to contribute valuable insights to developers, project managers, and stakeholders who are navigating the complexities of ensuring software quality and reliability in an ever-evolving technological ecosystem.

2 Understanding Mock servers in Software Development

Mock servers have emerged as instrumental tools in modern software development, providing developers with a versatile mechanism to simulate the behavior of real server environments. This section delves into the fundamental concepts, functionalities, and implications of mock servers, aiming to establish a comprehensive understanding of their role in the software development lifecycle.

Illustrated in Figure 2, depicts a interaction between client to real server and how real server is replaced with mock server to replicate real server but real server is not used but mock server is.

Figure 1. Mock servers interaction (SoniAnshu, n.d.)



2.1 Definition and Core Functionality

At its core, a mock server is a simulated server environment designed to emulate the responses and behaviors of a live server. Unlike traditional Backend Systems, which may introduce complexities and dependencies during development and testing, mock servers offer a lightweight and controlled alternative. Developers employ mock servers to create an environment where various scenarios and responses can be simulated, facilitating efficient and comprehensive testing without reliance on external services.

2.2 Creating Controlled Testing Environments

One of the primary advantages of mock servers lies in their ability to create controlled testing environments. Developers can design and manipulate responses to mimic a wide range of scenarios, including edge cases, error conditions, and varying network conditions. This controlled testing environment enables thorough validation of code behavior under diverse circumstances, contributing to the overall robustness and reliability of the software.

2.3 Accelerating Development Cycles

Mock servers play a pivotal role in expediting development cycles. By decoupling development from live backend systems, developers can work independently and iterate rapidly without waiting for stable infrastructure or external services. This agility in development is particularly advantageous in the context of agile methodologies, allowing teams to respond quickly to changing requirements and deliver features with increased speed.

2.4 Supporting Parallel Development

In collaborative development environments, where multiple teams or developers work on different components concurrently, mock servers prove invaluable. They enable parallel development by providing each team with the autonomy to create and test against simulated server responses. This parallelization minimizes bottlenecks associated with shared resources and dependencies, fostering a more streamlined and collaborative development process.

2.5 Facilitating Continuous Integration and Testing

The integration of mock servers aligns with continuous integration and testing practices. Automated testing pipelines can leverage Mock Servers to simulate various scenarios, ensuring that code changes are thoroughly validated before integration into the main

codebase. This integration not only enhances the reliability of software but also contributes to the early detection and resolution of potential issues.

2.6 Addressing External Service Dependencies

In scenarios where software components interact with external services, mock servers provide a solution to mitigate dependencies on live services during development and testing. By simulating the responses of these external services, developers can isolate their code and ensure that it functions as intended, even when external services are unavailable or exhibit unpredictable behavior.

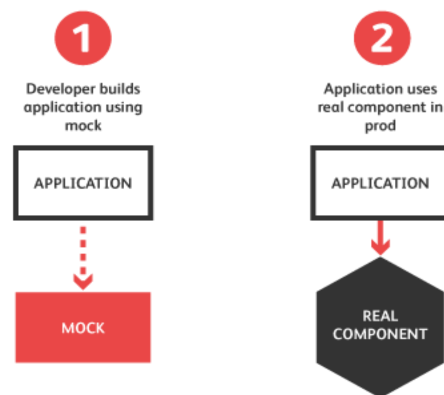
3 What is API Mocking

The most common term for creating simulated components is mocking, but others have also been used, and partly applied to different things like stubbing, simulation and virtualization. The basic concept is always same instead of using an actual software component, a replacement version of that API is created and used instead. It behaves as the original API but lacks many of the functions and non-functional characteristics of the original component.

3.1 Mocking During Development

Developers are frequently tasked with writing code that integrates with other system components via APIs. Unfortunately, it might not always be desirable or even possible to actually access those systems during development. There could be security, performance or maintenance issues that make them unavailable or they might simply not have been developed yet. This is where mocking comes in instead of developing code with actual external dependencies in place, a mock of those dependencies is created and used instead. Figure 3 shows interaction when mock is used and is built around using mock versus using real component directly in production to simulate testing.

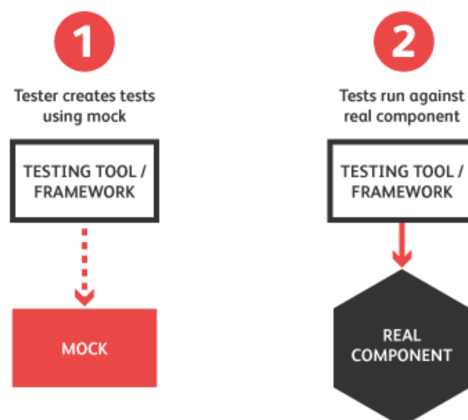
Figure 2. Mocking application during development instead real application (SoapUI, n.d.)



3.2 Mocking For Functional Tests

The testing usage of mocking is similar to development. Components may be unavailable for security, performance, maintenance or non-existence reasons. A basic mock of the component to be tested can be more than enough to get testing efforts started. Figure 4 shows things like discovering operations that need to be tested, creating initial test scripts, scheduling test execution and so on, can be done even with a fairly basic mock. There is a limit here a mock will not be a full simulation of the corresponding component which will put a limit to what kind of functional tests that can be used with it, but it can definitely be enough to get the testing effort started.

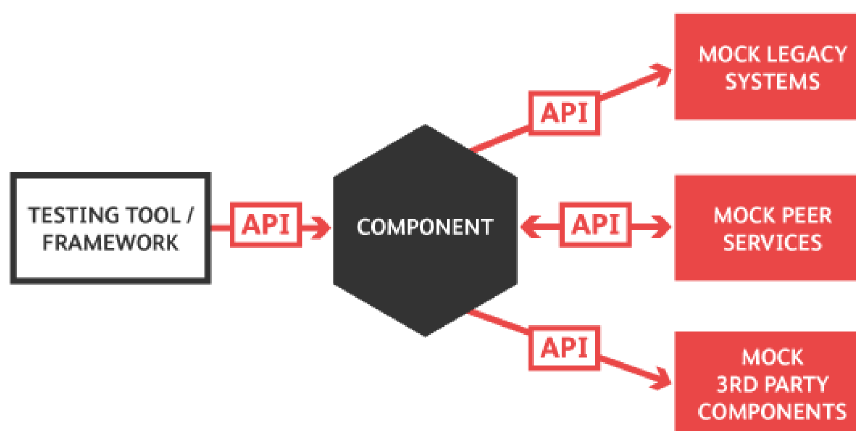
Figure 3. Mock testing with mock instead of real API (SoapUI, n.d.)



3.3 Mocking for External Components

When conducting functional testing of a component that depends on external components accessed via APIs, a situation where mocking can be very useful arises. For example, testing functionality in the component might involve the use of APIs for coordinate lookups. Instead of using the actual API, a corresponding mock could be employed to return known results to the component. This approach helps validate that the results from the component are returned correctly, unaffected by eventual inconsistencies in the external components. Figure 5 shows how mocking is done with external dependencies and how interaction with these dependencies look when interacting with mock.

Figure 4. Mocking with external dependencies (SoapUI, n.d.)



4 Using a mock in comparison to using a backend

In contemporary software development, the choice between utilizing a mock server and a traditional backend system represents a decision that profoundly influences the efficiency and effectiveness of the development process. Here is conducted a thorough comparative analysis, dissecting key dimensions to provide insights into the advantages, limitations, and contextual considerations associated with the use of mock servers in contrast to traditional backend systems.

4.1 Testing Environments and Scenarios

Testing forms a cornerstone of software development, and the choice between a mock server and a live backend system significantly impacts the testing environment. Mock servers provide a controlled setting for testing, enabling developers to simulate various scenarios and responses. This contrasts with the challenges posed by live backend systems, such as dependency on external services, data availability, and the potential for unintended side effects during testing.

4.2 Development Speed and Iteration

Speed and iteration are paramount in agile development methodologies. Mock servers excel in accelerating development cycles by allowing developers to create and modify simulated responses swiftly. This stands in stark contrast to traditional backend systems, where the pace of development may be impeded by dependencies on external services, database connections, and the need for a stable and complete infrastructure.

4.3 Resource Utilization and Cost Efficiency

The allocation of resources, both in terms of infrastructure and associated costs, is a crucial consideration. Mock servers, being lightweight and self-contained, often present a cost-efficient alternative compared to maintaining and scaling backend systems. This section explores the implications for resource utilization and cost efficiency, considering factors such as server maintenance, scalability, and infrastructure costs.

4.4 Realism and Simulation Accuracy

While mock servers provide controlled environments for testing, questions arise regarding their ability to replicate the realism of live backend systems. This part of the analysis examines the trade-off between simulation accuracy and realism in mock servers, contrasting it with the authenticity inherent in testing against actual backend systems, which operate in real-world conditions with live data and dynamic interactions.

4.5 Adaptability to Changing Requirements

The adaptability of development processes to changing requirements is a hallmark of successful software projects. Mock servers offer flexibility in adapting to evolving project needs, allowing developers to modify responses and simulate different scenarios easily. This section contrasts the adaptability of mock servers with the potential challenges associated with modifying or updating backend systems, particularly in cases where dependencies are tightly coupled.

4.6 Summary

This comparative analysis aims to explain the multifaceted considerations surrounding the choice between mock servers and backend systems. The insights derived from this exploration will serve as a foundation for subsequent chapters, guiding a comprehensive understanding of the strategic implementation of mock servers in the software development lifecycle.

5 Application Programming Interface(API)

In the realm of software development, the Application Programming Interface(API) has played a pivotal role in facilitating communication and interoperability between diverse systems. Historically, APIs served as the linchpin for connecting disparate software applications, enabling developers to harness the functionalities of existing systems and build upon them. In the past, APIs were predominantly utilized for data exchange, allowing applications to retrieve and share information efficiently. These interfaces became integral components of software architecture, fostering modularity and abstraction. As technology advanced, the role of APIs evolved beyond mere data exchange to encompass a wide spectrum of functionalities, including authentication, security, and the facilitation of complex business logic. This retrospective analysis acknowledges the foundational impact of APIs in shaping the interconnected digital landscape, highlighting their historical significance as catalysts for innovation and collaboration in software development.

5.1 Open API

Open APIs are accessible for utilization by all developers. Consequently, open APIs have minimal authentication and authorization measures, which often leads to resource restrictions. The primary advantage of exposing an API is the transparent sharing of information, encouraging external entities or developers to incorporate the API into their applications, thus promoting development and harnessing the provided API data. (Juviler, 2023)

5.2 Partner API

Partner APIs are shared among entities that have a business relationship with the entity providing the API. Access is restricted, but only authorized customers with licenses are granted entry. Security measures are more stringent for partner APIs compared to public APIs. Businesses opt for partner APIs to exert control over resource access and usage limits. (Juviler, 2023)

5.3 Internal API

Internal APIs are designed for in-house use and are not accessible to external parties. These APIs are employed within a company to facilitate data exchange between teams and systems. Company developers utilize these APIs to enhance application interfaces and streamline data movement. The use of APIs for internal data transfer enhances efficiency, security, and traceability within the organization, expediting data retrieval and sharing. This approach is particularly beneficial when a company introduces a new internal system that interacts with existing systems through their application interfaces. (Juviler, 2023)

5.4 Composite API

Composite APIs amalgamate multiple application interfaces, with developers bundling calls or requests to obtain a unified response from various servers. Composite APIs are employed when data is needed from diverse applications or sources. They enable initiating or

automating actions without direct intervention. This reduces the overall number of API calls, contributing to server load optimization and system acceleration. The reduction in system complexity is advantageous for businesses. (Juviler, 2023)

6 REST application interface

In the historical narrative of software architecture, Representational State Transfer(REST) emerged as a foundational paradigm, fundamentally altering the way systems communicated and interacted over the internet. In the past, REST, conceived by Roy Fielding in his doctoral dissertation in the year 2000, (Fielding, 2000) provided a simple yet robust set of principles for designing networked applications. RESTful architectures, characterized by stateless communication and resource-based interactions, became the backbone of countless web services. RESTs emphasis on scalability, simplicity, and the uniformity of interfaces facilitated widespread adoption and laid the groundwork for interoperability between diverse systems. Throughout its historical journey, REST transcended its initial application in the World Wide Web to permeate various domains, influencing the design of APIs, web services, and mobile applications. This retrospective view recognizes REST as a pivotal chapter in the evolution of software design, underscoring its enduring impact on the architecture of distributed systems.

7 REST API

To harness the capabilities provided by REST, the utilization of application interfaces necessitates adherence to specific prerequisites. These prerequisites lay the groundwork for an agile and adaptable API. In this chapter, distinctive attributes that differentiate REST APIs from various other types of application interfaces are examined further.

7.1 How REST APIs work

In the historical evolution of software development, REST APIs functioned as a cornerstone, fundamentally altering the dynamics of communication between clients and servers. Shaped

by the principles delineated by Roy Fielding in his influential dissertation in 2000, REST APIs operated within a unidirectional interaction pattern. Clients initiated exchanges by forwarding requests to servers, which, in turn, responded with corresponding replies. This unidirectional communication framework emphasized client-driven interactions and defined the initiation of the entire engagement with the server. Within RESTful APIs, this communication model preserved the autonomy of both clients and servers, allowing for the enhancement of client software without exerting influence on other servers. (Fielding, 2000)

A critical facet of the REST architecture was the insistence that all requests and responses adhere to standardized protocols or message formats. This uniform interface served as a linchpin, providing a shared language for communication across diverse REST APIs. The absence of standardized communication could lead to potential issues such as confusion and data loss when translating requests and responses between different software systems. Consequently, each API update necessitated corresponding adjustments to the request processes of applications.

Given the diversity in programming languages used for application and server development, the establishment of a uniform interface became imperative to ensure communication without intermediaries. HTTP, although not originally designed exclusively for REST, emerged as the widely adopted language in REST application interfaces. (Fielding, 2000) This historical perspective illuminates the operational mechanics of REST APIs, emphasizing their role in fostering standardized, scalable, and interoperable communication across the landscape of software development.

7.2 Chararestics of REST APIs

REST APIs are characterized by their statelessness, meaning that every interaction stands alone, and each request and response carries all the requisite data to complete the interaction. The server interprets each client request as a fresh instance, devoid of any recollection of previous interactions. This stateless nature minimizes memory consumption on the server, leading to improved response efficiency, as the server does not need to retrieve past data. This ensures that interactions remain effective as software expands,

handling more requests without concerns about memory utilization or overload. (Juviler, 2022)

In a layered system, API requests generally occur between the client and server, even though there can be multiple intermediate layers in real-world scenarios. These intervening layers serve purposes such as security enhancement, traffic distribution, and other essential functions. The consistent formatting and processing of messages between clients and target servers, regardless of these intermediate layers, are paramount. These intermediate layers augment functionality without altering the core client-server interaction. This design principle enables server systems to undergo reorganization, updates, or modifications without impinging on the fundamental interaction. (Juviler, 2022)

Caching is employed to store media on the clients device while accessing a website. When the client revisits the same site, cached data is swiftly retrieved from local storage, obviating the need to fetch it anew from the server. Caching conserves server resources and bandwidth, leading to reduced page load times, a common practice on large websites. An optional aspect of REST is the ability for an API to transmit computer code as a response to the client. These guiding principles, however, grant developers the flexibility to customize their API functionalities, setting REST APIs apart from other conventional web application interfaces. (Juviler, 2022)

7.3 REST API USAGE

The utilization of REST APIs in software development marked a transformative era, characterized by streamlined communication and interoperability. In the past, REST APIs served as a linchpin, facilitating interactions between clients and servers across diverse applications. Developers embraced REST principles as they sought to enhance the efficiency and flexibility of their systems. This approach, encapsulated in Roy Fieldings dissertation in 2000, (Fielding, 2000) emphasized a uniform and stateless interface, allowing clients to initiate requests and servers to respond accordingly. The autonomy bestowed upon clients and servers in RESTful interactions empowered developers to enhance and evolve client software independently, devoid of influence on other servers. The uniform interface,

requiring adherence to standardized protocols, became a cornerstone, providing a shared language for communication between applications developed in different programming languages. The adoption of REST APIs mitigated potential challenges arising from diverse technological ecosystems, ensuring communication without the need for intermediaries. As software systems evolved, the uniformity and simplicity of REST APIs endured, making them a ubiquitous choice for data transfer between interacting applications, with HTTP emerging as the de facto standard protocol. This historical perspective acknowledges the instrumental role of REST API usage in fostering a more interconnected and adaptable landscape for software development.

7.4 Simple Object Access Protocol- ja Remote Procedure Call API

In the historical context of software development, the Simple Object Access Protocol (SOAP) API played a significant role as a communication protocol for exchanging structured information in a decentralized and distributed environment. SOAP, which emerged in the late 1990s, was characterized by a set of messaging patterns and an Extensible Markup Language(XML)-based message format. Historically, SOAP APIs were widely adopted for their versatility in enabling communication between applications, especially in enterprise-level systems. Developers leveraged SOAP APIs to facilitate the exchange of structured data over a variety of protocols, including HTTP and Simple Mail Transfer Protocol (SMTP). Despite its historical prominence, SOAP has witnessed a shift in adoption patterns with the emergence of more lightweight and flexible alternatives, such as RESTful APIs. The historical analysis of SOAP APIs provides insights into the evolution of communication protocols and the dynamic landscape of software development. (W3C, 2007)

In the evolution of software development, the Remote Procedure Call (RPC) API served as a fundamental mechanism for enabling communication between distributed systems. Historically, RPC provided a standardized protocol for invoking procedures or functions on a remote server, allowing interaction between applications. However, with RPC facing challenges related to platform dependence and interoperability issues. The advent of more contemporary communication protocols, such as RESTful APIs, contributed to a reevaluation of RPC methodologies in favor of lightweight and versatile alternatives. This historical

analysis of RPC APIs sheds light on the dynamic nature of communication protocols in the ever-evolving field of software development. (Birrell, 1984, p. 39)

8 Mock tools

The installation of a mock tool within a testing environment constitutes a crucial phase for many teams. This process involves navigating aspects such as usability, version compatibility, data management, service group requisites, and alignment with all customer specifications. However, ensuring the stability of these testing environments can pose a formidable challenge, as any alterations introduced may have a ripple effect on their overall reliability. A shared testing environment compounds this complexity, impacting both testing and development teams. A prevalent practice involves conducting acceptance tests prior to merging branches, but when the testing environment is shared, adhering to this practice becomes notably intricate. Consequently, these factors can significantly impede the productivity of the development team. Instances of misdirected efforts can proliferate, resulting in misaligned development and fostering discord among teams. (Ramdeo, 2016)

According to Ramdeo (2016), several implications arise from operating within a shared testing environment:

- Extended test execution time: The resource-sharing nature of a shared testing environment elongates the duration required to run tests due to resource contention.

- Impact of environment changes: Modifications introduced to the testing environment may yield false test errors, thereby skewing the accuracy of test outcomes.

- Unintended side effects: Tests executed by one customer can inadvertently impact the outcomes of tests conducted by other customers, leading to unforeseen consequences.

- Feedback delay: Even if a systematic test queue is upheld, the duration for receiving feedback from test execution escalates in tandem with the test suites size.

Navigating these challenges that Ramdeo (2016) proposed requires careful orchestration and thoughtful management of the testing environment to uphold the integrity and effectiveness of the testing process. (Ramdeo, 2016)

8.1 Existing tools for Mocking

Collaboration of tools played a pivotal role for API development, facilitating the planning, design, and development. Many of these platforms were equipped with readily available mock servers tailored to integrate API collections. Overcoming the hurdle of developing and testing code amidst the ongoing development of external components emerged as a prominent challenge during API application integration. In this landscape, the creation of mock servers emerged as a highly effective strategy for mitigating these interdependencies, culminating in delivery of a thoroughly tested product. (Hossein, 2022)

8.1.1 Introducing Mock Tools

The introduction of mock tools in the past revolutionized the software development landscape, offering essential support for development teams. Tools such as Postman, Spotlight, Mocky.io, and MockServer played a crucial role in this transformation. These tools were designed to simulate real system behaviors, providing developers with the capability to generate illustrative responses for a variety of scenarios. With the integration of Postman, Spotlight, Mocky.io, and MockServer into the development workflow, teams could emulate API behaviors, allowing for thorough testing without the dependence on actual external services. This approach significantly streamlined the testing phase, enabling developers to identify and address potential issues early in the development cycle.

8.1.2 Postman

Utilizing Postman, the setup process for configuring a mock server was integrated with its desktop application. The mock server, functioning as a simulation of an authentic API server, was engineered to accept incoming requests and furnish corresponding responses. Through the integration of the mock server into a collection, coupled with the inclusion of examples within requests, one could effectively replicate the operational dynamics of a genuine API. Upon dispatching a request to the mock server, Postman undertook a matching process with the example stored in the collection, subsequently orchestrating a response enriched with data. Perusing existing mock servers housed within the workspace could be effortlessly executed through the desktop application. (Hossein, 2022)

Postman stands out as an effective tool for mock operations due to its widespread popularity and its comprehensive documentation encompassing APIs nuances, utilization, and even the creation of personalized mock servers. The platform grants access to a free-of-charge mock server for your utilization, while also offering a premium service for more advanced and sophisticated mocking capabilities.

8.1.3 Spotlight

Spotlight, stemming from the proprietary Prism platform, stood as a robust mock tool renowned for its prowess in the realm of designing and documenting REST APIs. This tool served as a dynamic conduit through which collection files could swiftly turn into functional API servers, replete with meticulously crafted mock variations and confirmations. Spotlight, by its very essence, empowered users to fashion synthetic REST API endpoints, with diverse HTTP methods at their disposal. Within the realm of Spotlights offerings, the free iteration furnished users with the ability to enable APIs and generate comprehensive documentation for them. Meanwhile, the enterprise version of Spotlight ushered in a host of additional capabilities, including a versatile version control system, collaborative API design functionalities, and the liberty to engineer mock APIs for multiple projects sans constraints. This tool integrated with an array of version control systems, further enhancing its versatility and convenience. (Hossein, 2022)

Elevating its status as a proficient mock tool, Spotlight boasts both a complimentary and premium iteration. While the free variant encompasses a gamut of Spotlights features, the paid edition emerges as an advantageous proposition for businesses, ushering in an array of advanced attributes and unbounded API utilization capabilities.

8.1.4 Mocky.io

Mocky.io presents a streamlined approach to mock server creation, requiring merely a single click to set up a simulated server. Dispensing with the need for user authentication, Mocky.io grants users access to its pre-fashioned templates and intuitive designer interface. While it might not encompass the full spectrum of advanced functionalities, it does wield an array of capabilities that can stand out from the offerings of other tools. Notably, Mocky.io introduces unique attributes like response delays and expiration times, enriching the spectrum of options available for mocking. (Hossein, 2022)

Mocky.io comes completely cost-free, making its services accessible to all without financial constraints. This propensity to abstain can be attributed to Mocky.ios open-source nature, which, although fostering collaboration, might also raise concerns pertaining to security vulnerabilities.

8.1.5 MockServer

The versatility of the MockServer tool extends beyond mere mock server functionality, encompassing proxy server capabilities that facilitate the modification and emulation of requests and responses. With its dynamic adaptability, MockServer serves as a standalone server application, capable of programmatic execution. This tool is used in both simulation and analysis by capturing requests, thereby furnishing insights for testing or in-depth scrutiny. By crafting verification tests by hand ascertains the congruence of requests from the system under scrutiny, obviating the necessity for request mocking. The recording of requests provides a comprehensive dataset for analysis, with the tool meticulously logging all outgoing requests. This is particularly significant due to the limitations of network analysis tools in browsers, which might not encompass the entirety of network interactions.

Additionally, MockServer overcomes proxy server limitations by autonomously generating certificates using its root certificate, even for encrypted HTTPS traffic. The recorded requests can be transformed into multifaceted expectations, contributing to the simplification of testing scenarios. (Hossein, 2022)

MockServer stands out as tool for mocking, by its popularity and extensive API documentation. Its open-source nature reflects a commitment to ongoing development and improvement. Nevertheless, this openness can engender security concerns for enterprises, potentially dissuading their adoption. The tool comes complete with a wealth of documentation and comprehensive guides, readily accessible on its official website, ensuring its integration into diverse testing environments.

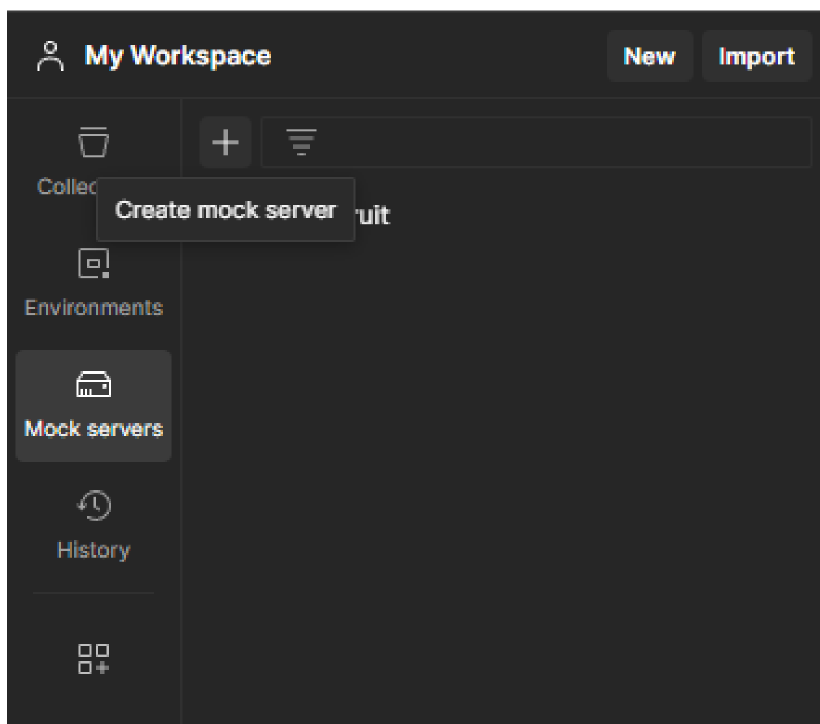
9 Mock demo

Postman emerged as tool of choise for demo. Postman is catered to a multitude of purposes and integration into diverse projects. Abundant documentation guided users on implementing Postman into their projects and initiating the mocking process. While certain other tools might have offered simpler integration. Postman stood out as an excellent choice. Notably, the MockServer tool also presented itself as a good option for mocking. Countless sources held both alternatives in high regard, alongside various other contenders. Ultimately, the crux lay in aligning the chosen tool with the projects objectives and demands, ensuring that mocking efficiently propelled the projects trajectory and advancement. Introducing such tools into a project imbued them with a sense of purpose and empowered users to effectively showcase their utilization. Within Postman, the creation of mock servers stood as a valuable asset in API development and testing endeavors. Mock servers accurately replicated genuine API behavior, accepting requests and promptly delivering corresponding responses. By integrating the mock server into a collection and embedding illustrative examples within the requests, one effectively duplicated authentic API behavior. Consequently, as requests were sent to the mock server, Postman adeptly correlated them with the stored examples in the collection, orchestrating responses that harmonized with the furnished data.

9.1 Mock creation

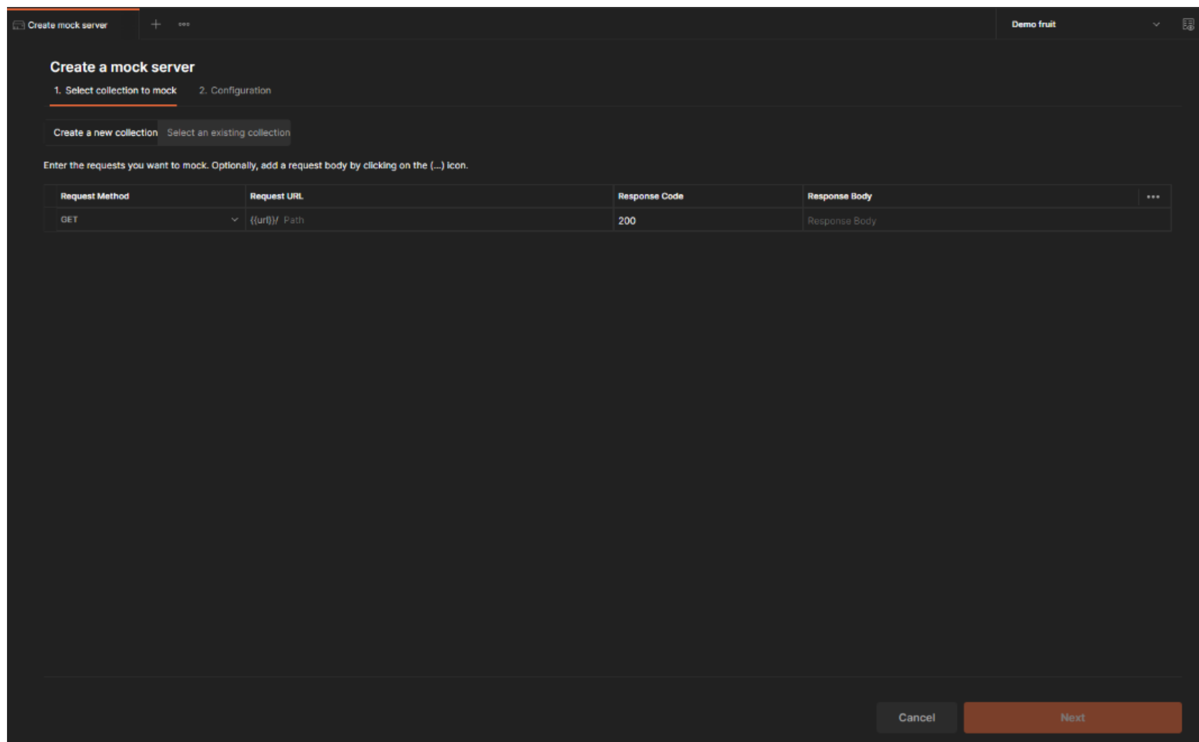
The initiation of the mocking process typically relies on the existence of an established project. Nevertheless, Postman introduces straightforward approach which allows users to create versatile mock servers and generate a range of responses for these servers. The process of commencing mocking through the Postman tool is notably straightforward. To initiate this process, user could simply click on the designated element, as illustrated in figure 5, which allowed them to begin creating a mock server and input its details. Subsequently, the mock server panel became accessible after creating mock server.

Figure 5. Starting the mock server



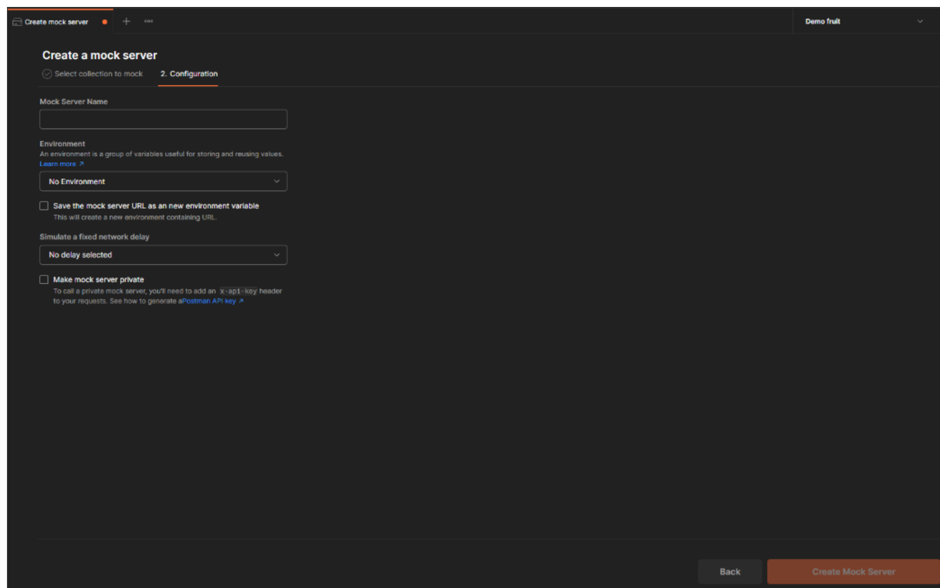
On figure 6, mock details and vital information that needs to be provided can be observed. These details encompassed the address and the preferred request method, playing a crucial role in defining the desired type of response. The user-friendly interface of this feature illustrated how simple the process had been, empowering users to adeptly customize their mock server experience to align with their specific requirements.

Figure 6. Creating URL and response body



In Figure 7, a dedicated space was illustrated for entering crucial information. This included assigning a distinctive name, specifying the corresponding project environment, and configuring the desired delay. Additionally, users had the option to designate the mock as private, which hid the project. Making the project private generated a unique Postman API key, with these keys being unique from each other, and each project generated its own Postman API key. The Postman key could be used for accessing a specific project or providing a key for accessing the current project. This diverse range of customization options enabled users to tailor the mock server experience for themselves, aligning it with the unique dynamics and objectives of their project.

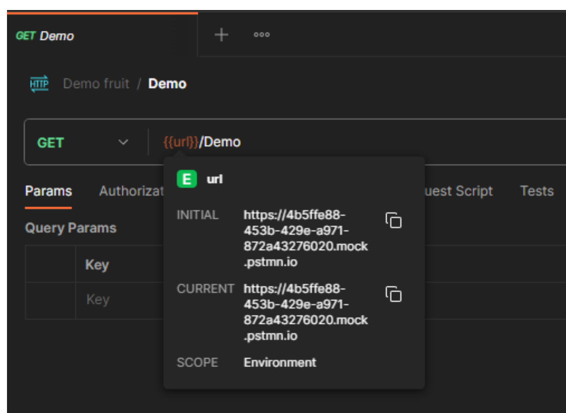
Figure 7. Mock name, environment, delay and privacy selection



9.2 Editing mock and sending data

To provide a more detailed insight into the process, initiating a new tab allowed for a focused exploration of the nuances associated with the connection type and the mock URL. The construction of the mock URL held pivotal importance, encompassing both the unique identifier of the mock server and the pathway of the request for emulation. In the demonstration, HTTP URL was generated by Postman, which was automatically populated in the corresponding field when the mock server was created. Figure 8 showcased an example URL created by Postman.

Figure 8. URL created by Postman



Prior to transmitting data to the mock server, configuring a response body becomes a prerequisite, outlining the anticipated content in the response. Figure 9 illustrated the location where access to the response body is available within collections. Users could locate the file, either automatically generated upon mock server creation or manually created beforehand, by navigating to the 'demo fruit' file. Within this file, all 'ping bodies' created were listed, and under each 'ping body,' the respective response bodies were available. In this specific example, the response body is denoted as 'Default'.

Figure 9. Response body location

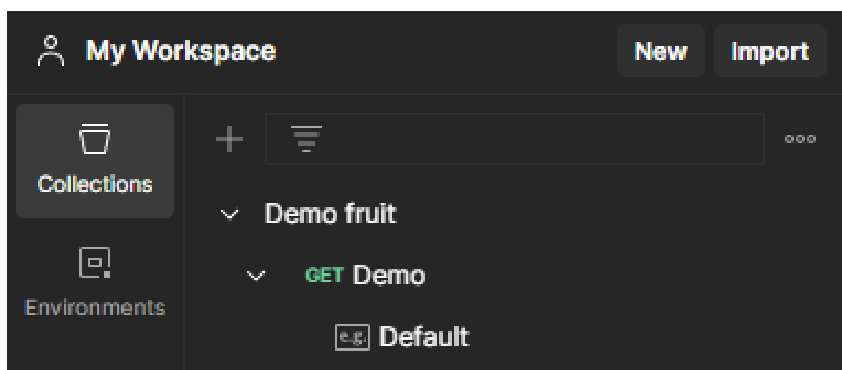


Figure 10. Response body filling

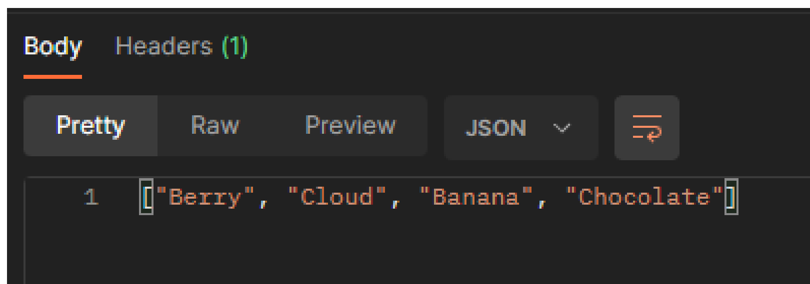


Figure 10 provided the interface for configuring the data to be transmitted to the mock server. In this instance, a JSON code was crafted as a simple response body. Users have the flexibility to customize the text body according to their requirements, and they can also switch between various code languages or opt for automatic language detection. Figure 11 showcased a comprehensive list of code languages available in Postman for setting up response bodies.

Figure 11. Choosing code language

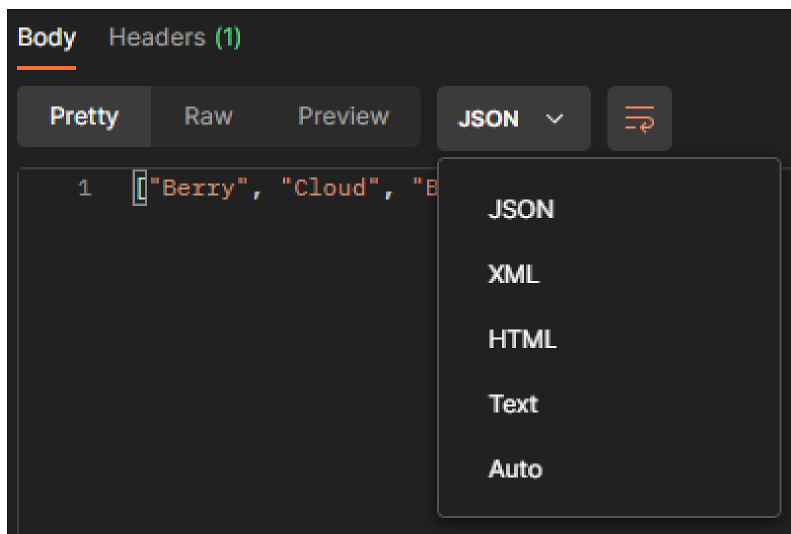
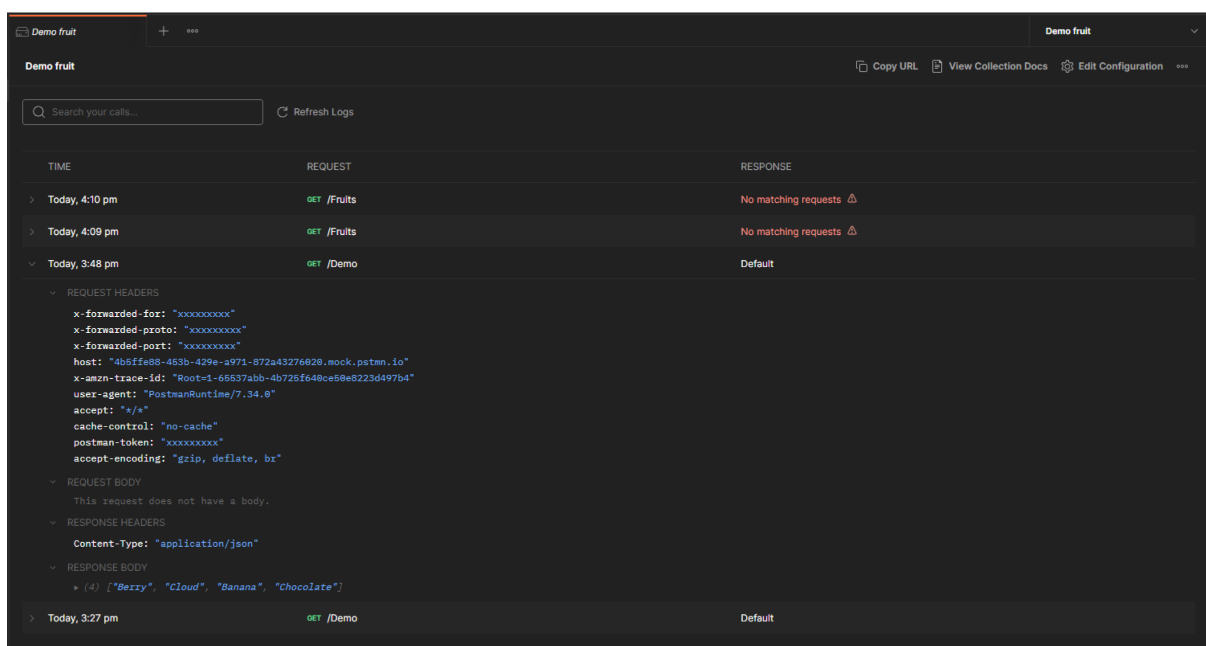


Figure 12 presented a comprehensive display of all data generated by the mock server. Within this figure, details such as request headers, request body, response headers, and response body were visible. Each of these components can be expanded to reveal more detailed information. This feature allowed users to access and review data from every instance the mock server received information, creating a cumulative list of interactions over time.

Figure 12. Mock server body



Displayed in the figure 12, the type of response body generated by pinging the mock server was evident, specifically identified as JSON, as illustrated in figure 10. Additionally, the figure displayed the response body that was received by the mock server, a configuration previously set to ['Berry', 'Cloud', 'Banana', 'Chocolate'], as outlined in figure 10.

9.3 Demo summary

In configuring all settings and executing an HTTP request through Postman, remarkable user-friendliness is exhibited by the tool. The setup process for the mock server proves to be expeditious, and no complications are encountered during its establishment. Upon acquiring the mock URL and assigning it as the designated mock server URL, the requisite response body for the mock is furnished. This configured response body is subsequently utilized to transmit data to the mock server.

Within the mock server, the received data can be comprehended and correlated with the anticipated data, thereby confirming the accuracy of the transmitted information. In conclusion, Postman can be safely recommended as a tool for mocking, catering to both businesses and individuals. The concise demo presented via Postman showcases the user-friendly nature of starting the mocking process and why it should be an integral part of API development.

Additionally, considering the challenges associated with integrating a mock server after a project's completion, the significance of planning for mocking from the project's inception cannot be overstated. This approach significantly simplifies the integration and maintenance of mock servers in the long run.

10 Conclusion

In conclusion, the adoption of mock servers and the strategic use of mocks in software development offer significant advantages, enhancing the efficiency and reliability of the testing and development processes. Mock servers provide a valuable solution when actual access to external components, especially via APIs, is impractical or poses challenges. By creating intelligent mocks that simulate the behavior of real components, developers can ensure the robustness of their code and validate its functionality without being hindered by the potential unavailability or inconsistencies in external systems. Moreover, the controlled environment afforded by mock usage allows for thorough testing and validation of diverse scenarios, contributing to the overall quality of the software. As a result, the integration of mock servers into development workflows stands as a valuable practice, enabling developers to build resilient and effective software solutions.

The utilization of mock tools in software development represents a pivotal strategy for enhancing efficiency and reliability throughout the development lifecycle. These tools play a crucial role in creating simulated environments, allowing developers to test and validate their code seamlessly, especially when actual access to external components or services is impractical or poses challenges. By generating intelligent mocks, developers can simulate diverse scenarios and ensure that their applications respond effectively. The versatility of mock tools extends to a variety of use cases, from functional testing to API development, contributing to more robust, resilient, and well-tested software solutions. As an integral part of modern development practices, mock tools empower developers to streamline workflows, mitigate dependencies, and deliver higher-quality software with confidence.

References

Javier, B. (2022). *Why a mock server.* https://www.mocks-server.org/docs/overview/

Fielding, R. T (2000). *Architectural Styles and the Design of Network-based Software Architectures. PhD dissertation, University of California*

Hossein. (2022). *Top 7 Free & Paid mock API tools (2022 Review)*. https://testfully.io/blog/mock-api/

Juviler, J. (2023). *4 Types of APIs All Marketers Should Know*. https://blog.hubspot.com/website/types-of-apis

Juviler, J. (2022). *REST APIs: How They Work and What You Need to Know*. https://blog.hubspot.com/website/what-is-rest-api

Liu, A. (2020). *Mock APIs vs. Real Backends – Getting the Best of Both Worlds*. https://www.confluent.io/blog/choosing-between-mock-api-and-real-backend/

Postman. (2022). *Understanding example matching*. https://learning.postman.com/docs/designing-and-developing-your-api/mocking-data/matching-algorithm/

Ramdeo, A. (2016). *MockServer - Three reasons why you should use them for your test automation*. https://www.linkedin.com/pulse/mockserver-three-reasons-why-you-should-use-them-your-anand-ramdeo/

Stoplight. (2022). *Mock api guide*. https://stoplight.io/mock-api-guide/basics

W3C. (2007). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). https://www.w3.org/TR/soap12-part1/

Birrell, A., & Nelson, B. J. (1984). Implementing Remote Procedure Calls. p. 39 doi:10.1145/2080.357392

SoniAnshu, (n.d.) *Soctware Engineering MOCK introduction* *https://www.geeksforgeeks.org/software-engineering-mock-introduction/*

SoapUI, (n.d.) *What is API mocking* *https://www.soapui.org/learn/mocking/what-is-api-mocking/*