**API STANDARDS**

**AND**

**BEST PRACTICES**

**v3.0**

****

**Acknowledgements**

This document is the result of collaborative efforts of several individuals from RBI, ReBIT and the industry.

We are grateful to the external committee members who are subject matter experts and eminent leaders in the industry. Prof. H. Krishnamurthy (Chief Research Scientist (Retired), IISc) for leading the team as a Chairman of API Standards Advisory Committee, Other committee members – Mr. Sachin Shende (COO, IFTAS), Ms. Darshana Kulkarni (GM, DIT, RBI), Mr. Amol Pai (Ex-CTO, SBI), Mr. Ramesh Lakshmi Narayanan (CIO, HDFC Bank) and Mr. Arul Kumar (Architect, NPCI). This endeavor would not have taken a final shape without their valuable feedback.

I would also like to acknowledge the contributions made by the members of the ReBIT Team. Special mention is warranted for Ms. Sayali Kulkarni (Software Architect) supported by Mr. Shukti Sarwade (Sr. Technical Architect) without whom this document would not have been possible. I would like to acknowledge the other core contributors from ReBIT team, Mr. Kamlesh Ahire (Sr. Architect, Cyber Security), Mr. Rohan Sahu (Sr. Software Architect), Mr. Shashank Sadavarte (Software Architect), Mr. Kanchan Durgapal (Software Architect, Ex. ReBIT) and Ms. Balaka Dash (Manager-Cyber Security). Other contributors to this document include Mr. Jay Karia, Ms. Ashwini Shetty and Ms. Amruta More.

Lastly, I would like to thank Mr. Santhosh George (Sr. Vice President, ReBIT) for carrying the entire project through till its logical end and all the members of the ReBIT leadership team for their support.

**A O Basheer**

*CEO, ReBIT*

Table of Contents

[Executive Summary 1](#_Toc183079986)

[1. Types of APIs 7](#_Toc183079987)

[1.1 What is an API? 9](#_Toc183079988)

[1.2 Types of APIs 9](#_Toc183079989)

[1.3 Recommendation 16](#_Toc183079990)

[2. Need for APIs and API Identification Strategy 18](#_Toc183079991)

[2.1 When to use the APIs? 20](#_Toc183079992)

[2.2 API Identification Strategy 21](#_Toc183079993)

[2.3 API Identification Strategy – Example Use case 21](#_Toc183079994)

[2.4 Layered Architecture 28](#_Toc183079995)

[2.4.1 Layered Architecture Principles 28](#_Toc183079996)

[2.4.2 Layers in Layered Architecture 29](#_Toc183079997)

[3. API Design Patterns 32](#_Toc183079998)

[3.1 API Design Patterns Summary 34](#_Toc183079999)

[3.2 Synchronous APIs 35](#_Toc183080000)

[3.3 Asynchronous APIs (Reactive – Push response) 35](#_Toc183080001)

[3.4 Asynchronous APIs (Notification – Pull response) 36](#_Toc183080002)

[3.5 Atomic APIs and Composite APIs 37](#_Toc183080003)

[3.6 HATEOAS APIs 39](#_Toc183080004)

[3.7 Common Error Handling Strategy 40](#_Toc183080005)

[3.8 API Design Pattern – Example Use case 40](#_Toc183080006)

[4. API Development Life Cycle 48](#_Toc183080007)

[4.1 API Development Life cycle 50](#_Toc183080008)

[4.2 Roles and responsibilities of the project team 58](#_Toc183080009)

[5. API Specification Standards Checklist 60](#_Toc183080010)

[5.1 REST API Design Features 62](#_Toc183080011)

[5.2 API Specification Standards Checklist 63](#_Toc183080012)

[6. API Logging and Error Handling Standards 68](#_Toc183080013)

[6.1 Overview of API Logging 70](#_Toc183080014)

[6.2 API Logging Standards Checklist 71](#_Toc183080015)

[6.3 Overview of API Error Handling Strategy 73](#_Toc183080016)

[6.4 API Error Handling Standards 74](#_Toc183080017)

[6.5 API Error Handling Standards Checklist 77](#_Toc183080018)

[Annexure – Log message format 77](#_Toc183080019)

[Annexure – HTTP Status Codes 78](#_Toc183080020)

[7. API Security Standards and Best Practices 80](#_Toc183080021)

[7.1 What is API Security? 82](#_Toc183080022)

[7.2 API Security Framework 82](#_Toc183080023)

[8. API Governance Process and Versioning Strategy 88](#_Toc183080024)

[8.1 Background and Purpose 90](#_Toc183080025)

[8.2 API Governance Process 91](#_Toc183080026)

[8.3 API Versioning Strategy 92](#_Toc183080027)

[8.4 API Deprecation and Decommission Strategy 93](#_Toc183080028)

[8.6 Amendment of existing version 95](#_Toc183080029)

[8.7 Adoption of new API version – scenarios 97](#_Toc183080030)

[9. API Gateway Overview and API Policy Guidelines 98](#_Toc183080031)

[9.1 What is API Gateway? 100](#_Toc183080032)

[9.3 Service Mesh 101](#_Toc183080033)

[9.4 API Architecture 103](#_Toc183080034)

[9.5 API Policy 104](#_Toc183080035)

[10. API Developer Portal –Overview and Implementation Guidelines 106](#_Toc183080036)

[10.1 Background and Purpose 108](#_Toc183080037)

[10.2 API Developer Portal Overview 108](#_Toc183080038)

[10.3 API Developer Portal Details 111](#_Toc183080039)

[10.4 API Developer Portal Example 116](#_Toc183080040)

[11. API Performance & API Monitoring 118](#_Toc183080041)

[11.1 What is API Performance? 120](#_Toc183080042)

[11.2 Why API Performance is Important? 121](#_Toc183080043)

[11.3 Key Parameters for API Performance 121](#_Toc183080044)

[11.4 API Monitoring 123](#_Toc183080045)

[12. APIs and Artificial Intelligence 129](#_Toc183080046)

[12.1 Introduction 131](#_Toc183080047)

[12.2 Artificial Intelligence Enabling APIs 131](#_Toc183080048)

[12.2 APIs Enabling Artificial Intelligence 133](#_Toc183080049)

[List of Abbreviations 135](#_Toc183080050)

[References 136](#_Toc183080051)

# Executive Summary

**ReBIT (Reserve Bank Information Technology Pvt. Ltd.)** serves the IT and cyber security needs of the Reserve Bank of India. ReBIT executes a blend of maker and checker projects of RBI as well as public initiatives handled by RBI. Nature of projects handled by ReBIT is currently focused on Non-payment applications catering to various departments of RBI.

During ReBIT’s collaboration with RBI for various projects, it is noted that, as the IT landscape is changing rapidly with modern technologies taking over the legacy systems, RBI is also adopting the latest technologies in their infrastructure and systems. There is increase in the use of Application Programming Interfaces (APIs), REST protocol and microservices in RBI projects instead of monolith applications and traditional integration practices. This leads to various opportunities as described below.

**Opportunity Statement**

* Modularizing standalone monolith web applications by keeping front-end and back-end separate and integrating and securing the modules through APIs. Also, migrating monolith applications to microservices where inter-service communications will happen through APIs.
* Integrations between various applications of RBI and exchange of information.
* Integrations between RBI applications and outside applications from regulated entities for exchange of information.
* Creating API wrappers to encapsulate legacy applications.

The notable RBI applications connecting to regulated entities/external organizations using APIs include:

1. E-Kuber which is Core Banking Solution of the RBI. E-Kuber enables RBI to perform key fiduciary functions such as:
2. Banker to the Banks
3. Banker to the Government
4. Managing Public Debt etc.

Legacy E-Kuber system is being revamped with modern technologies like microservices and by using APIs.

1. Public Credit Registry (PCR) which will serve as a registry of all credit information for all lending in India. The PCR system is an API driven platform having integrations with the credit institutions, ancillary data providers, borrowers, and other stakeholders.

PCR system is using APIs

1. within the PCR web application,
2. to connect with other RBI applications and
3. to connect with regulated entities.
4. Central Information Management System (CIMS) which will replace the existing data warehouse of RBI. It collects information of 200+ returns from all financial services companies operating in India on periodic basis.

As such large-scale projects mentioned above are relying on the APIs for integration and interoperability between different systems, it is crucial that the APIs follow a set of standards which will ensure the interoperability, design scalability, and security of the APIs. More than performance scaling, focus is on the accuracy and secure API transactions with audit capabilities.

Hence, ReBIT has come up with API standards and best practices which can be adopted by all the RBI projects involving APIs. Initial focus is on setting generic API standards and in the subsequent stages more industry/domain focused standards will be created.

This executive summary gives an overview of the API standards and best practices defined by ReBIT.

**Overview of API Standards and Best Practices**

Below is the summary of topics covered:

|  |  |  |
| --- | --- | --- |
| **Topic** | **Category** | **Description** |
| Types of APIs | API Basics | This chapter contains API definition and Types of APIs. |
| Need for API and API Identification Strategy | API Design | This chapter explains the strategy to identify the APIs required for a particular requirement/process flow with an example use case. |
| API Design Patterns | API Design | This chapter explains the various design patterns of APIs particularly REST APIs such as Sync APIs and Async APIs. |
| API Development Life Cycle | API Development | This chapter explains the entire lifecycle of an API starting from design, implementation, testing, deploy and manage phase. |
| API Specification Standards Checklist | API Development | This chapter contains standards for defining the API specification. The document covers the resource naming conventions, use of proper HTTP methods, conventions for defining request/response body, error codes etc. |
| API Logging and Error Handling Standards | API Development | This chapter contains the logging standards and the error handling strategy of APIs. Proper logging and error handing is critical to debug the issues and to provide efficient support for resolving the errors. |
| API Security Standards and Best Practices | API Development | This chapter contains the API security standards and best practices. A Layered security approach is defined which covers the API security at different levels. |
| API Governance Process and Versioning Strategy | API Management | This chapter contains the API governance process and API versioning strategy which is critical for the efficient management of APIs. |
| API Gateway Overview and API Policy Guidelines | API Management | This chapter contains the overview of API gateway, its usage, different types of API policies and best practices to use the API policies. |
| API Developer Portal – Overview and Implementation Guidelines | API Management | This chapter contains the overview of API developer portal, its usage, and best practices for implementation of API developer portal. The portal is important aspect of API management when external entities are consuming the APIs developed by an organization. |
| API Performance & API Monitoring | API Monitoring | This chapter contains the API performance requirements and the API metrics which can be defined to enable API monitoring. |
| APIs and Artificial Intelligence | NA | This chapter contains the overview of how AI can be leveraged in API life cycle stages and how APIs can help to integrate AI capabilities in traditional applications. |

**Goals to be achieved**

By defining the API standards and best practices, we aim to achieve below goals:

1. Adoption of API standards for the API driven applications of RBI
2. Adoption of API standards by regulated entities of RBI in non-payment application development space in first phase. Further with increased maturity level to be adopted in other projects in subsequent phases.
3. Streamline the development, review, testing, and deployment process of APIs
4. Adoption of API standards by the vendors of RBI

**Expected Outputs of API Standardization**

By defining the API standards and best practices, following outputs are expected:

1. Set of guidelines and best practices for API developers.
2. Consistent interpretation of the API specifications across applications and integrations.
3. Specifications for API design, development, testing, security, and management.
4. Governance framework for API review and audit practices with traceability.

# 1. Types of APIs

## 1.1 What is an API?

An Application Programming Interface (API) is a set of clearly defined methods of communication between software components without any user intervention. API consists of a set of instructions and standards to be followed by the participating applications. APIs operate on an agreement of inputs and outputs and are independent of any specific programming language.

## 1.2 Types of APIs

To understand types of APIs, a familiar example of Digital Banking application provided by bank to the customer is depicted below.

The components of the example include customer, the web portal or mobile app provided by the bank and underlying Core Banking System (CBS) of the bank where the data is stored.

The types of APIs can be as below:

1. Purpose
2. Intended use
3. Architectural Style

**1.2.1 Purpose of API**

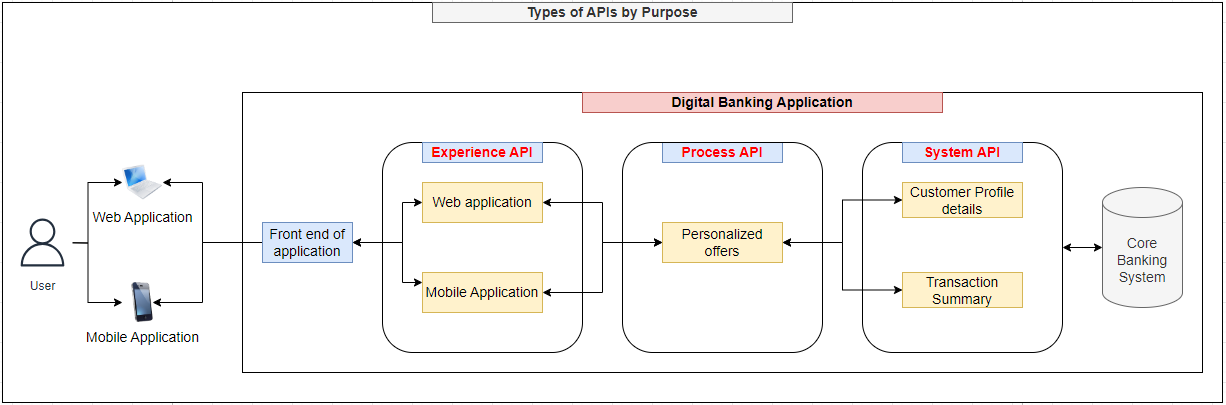


Figure 1.1: Types of APIs based on Purpose

The organizations choose to build the APIs due to a requirement to connect with heterogenic systems. In above diagram, the bank has a purpose to build the API – extract or insert data in the Core Banking System, perform orchestration on data such as providing personalized offers, and enable customers to use services through different channels such as a website or mobile app.

Based on the purpose, the APIs can be categorized as below:

1. **System API:**

System APIs are used to unlock the data from an application.

In above example, the System APIs interacts with the core banking system to retrieve or update data, providing a foundational layer for the banking application and serving as a building block for higher-level process and experience APIs.

1. **Process API:**

Process APIs are used for combining multiple system APIs and facilitate orchestration.

In above example, process API takes input from two system APIs – Customer profile details and transaction summary to provide personalized offers to customer.

1. **Experience API:**

Experience APIs provide a business context for the data and processes that were unlocked with System and Process APIs. Experience APIs expose the data tailored to its intended audience — such as customers accessing bank services through mobile applications, websites etc.

In above example, the experience API is taking input from personalized offers process API and showing offers to customer based on the channel i.e. web portal or mobile app.

**1.2.2 Intended Use of API**

The APIs can be categorized as External/partner APIs and Internal APIs based on who will be accessing the APIs or the intended use of the API.

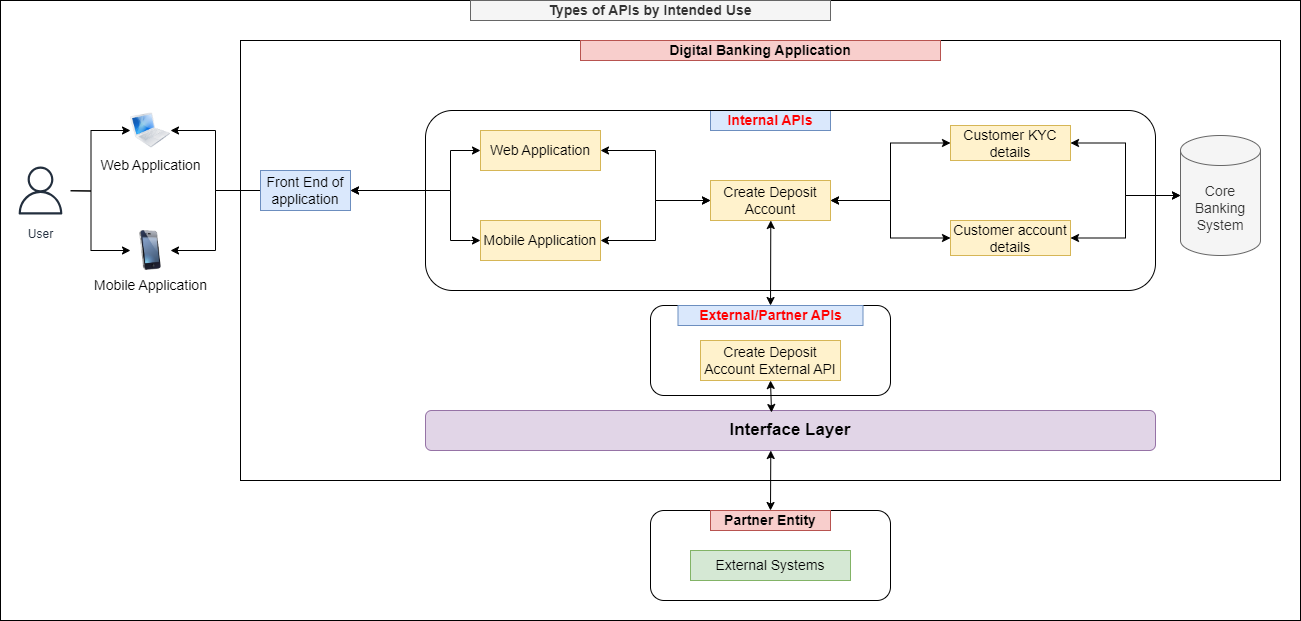


Figure 1.2: Types of APIs based on Intended use

1. **External/Partner API**

An external or partner API can be accessed by third parties that are external to the organization. An agreement can be setup between two organizations to use the API for exchange of data with right set of permissions. In some cases, the external APIs can also be available publicly and can be accessed by anyone who is interested in the API.

In above example, partner entity of the bank can get access to Create Deposit account API after completing the necessary due diligence process setup by the bank. Customer may use the partner entity web portal or mobile app to open deposit account in the bank.

1. **Internal API**

An internal (or private) API is accessible only to the members within the organization.

In above example, the internal APIs are developed for the Digital banking application and are used within organization only.

**1.2.3 Architectural Style of API**

The architectural style of an API is a design decision which needs to be taken by considering factors such as infrastructure – on premise or cloud setup, protocols used, existing technology stack, security controls required etc.

Below are some of the common architectural styles used for the API:

1. **REST (Representational State transfer)**

REST is an architectural style which separate the duties of the API consumer and provider by relying on the underlying protocol. It leverages HTTP methods (GET, POST, PUT, DELETE, etc.) and HTTP status codes to represent actions and responses. HTTP is the most used protocol for building a RESTful API.

Below are the key features of a RESTful API:

* **Client-Server**: Client and server are considered separate and allowed to evolve individually.
* **Stateless**: REST API calls can be made independently, and each call contains all the data necessary for that task. Hence, REST APIs are stateless.
* **Uniform** **Interface**: RESTful APIs offer a uniform interface using which client and server interact with each other independent of the underlying technology.

For Example:

If a customer wants to check account balance on the digital banking Application, the REST API for the same may be designed as below:

Syntax:

|  |
| --- |
| POST /accounts/balance |

Request:

|  |
| --- |
| POST /accounts/balance HTTP/1.1  Host: api.example.com  Authorization: Bearer your\_access\_token  {  "correlationid": "dfhj-afhj-fr54-ht3c",  "timestamp": "2024-10-15T04:35:018Z",  "accountId": "enchrtsjktyaf=",  } |

Response:

|  |
| --- |
| {  "correlationid": "dfhj-afhj-fr54-ht3c",  "timestamp": "2024-10-15T04:35:030Z",  "accountId": "enchrtsjktyaf=",  "accountBalance": 50000.00,  "currency": "INR"  } |

1. **SOAP (Simple Object Access Protocol)**

SOAP is a protocol for exchanging data in a standard format across network. SOAP uses XML data format for data exchange and has rigid guidelines in terms of the data format and documentation.

Below are the key components of SOAP:

* **Web Services Description Language (WSDL)**: Describes the service interface, including operations, parameters, and data types.
* **Envelope**: Contains the SOAP message, header, and body.
* **Header**: Optional and can contain metadata or security information.
* **Body**: Contains the actual data to be exchanged.

For example:

If a customer wants to check account balance on the digital banking Application, the SOAP service for the same may be designed as below:

Request:

|  |
| --- |
| <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">  <SOAP-ENV:Body>  <ns1:checkAccountBalanceRequest xmlns:ns1="http://example.com/banking">  <ns1:correlationid>dfhj-afhj-fr54-ht3c</ns1:correlationid>  <ns1:timestamp>2024-10-15T04:35:018Z</ns1:timestamp>  <ns1:accountId>enchrtsjktyaf=</ns1:accountId>  </ns1:checkAccountBalanceRequest>  </SOAP-ENV:Body>  </SOAP-ENV:Envelope> |

Response:

|  |
| --- |
| <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">  <SOAP-ENV:Body>  <ns1:checkAccountBalanceResponse xmlns:ns1="http://example.com/banking">  <ns1:correlationid>dfhj-afhj-fr54-ht3c</ns1:correlationid>  <ns1:timestamp>2024-10-15T04:35:030Z</ns1:timestamp>  <ns1:accountId>enchrtsjktyaf=</ns1:accountId>  <ns1:accountBalance>50000.00</ns1:accountBalance>  <ns1:currency>INR</ns1:currency>  </ns1:checkAccountBalanceResponse>  </SOAP-ENV:Body>  </SOAP-ENV:Envelope> |

1. **RPC (Remote Procedure Call)**

RPC is the oldest architecture style of API. RPC allows client to execute a block of code on server by calling a procedure and passing arguments. However, in this case client needs to exactly know which method to call and which parameters are to be passed. The gRPC[[1]](#footnote-1) and GraphQL[[2]](#footnote-2) protocol are modern forms of RPC architecture style.

1. **gRPC(Google Remote Procedure Call)**

gRPC(Google Remote Procedure Call) was introduced by Google in 2015 as an open-source RPC framework. It was initially developed for internal use within Google to address the need for a high-performance, efficient, and language-neutral communication mechanism for its distributed systems. It uses HTTP/2, an advanced version of HTTP for transport, and Protocol Buffers for message serialization. It provides a simple and efficient way for services to communicate with each other, regardless of the programming language or platform they are written in. In simpler terms, gRPC allows different applications to talk to each other like they are calling functions or methods directly, even if they are running on different machines or using different programming languages.

Below are the key features of gRPC API:

* **Protocol Buffers:** Protocol Buffersare useful in developing programs that communicate with each other over a network or for storing data**.** It offers a language-neutral, efficient, and strong typing data format that is highly versatile, compact, and efficient, and ensures type safety, reducing the risk of errors and making code more reliable.
* **HTTP/2:** HTTP/2 is a major update to the HTTP protocol that improves web performance by allowing multiple requests to be sent over a single connection, compressing headers, and enabling server push. These features reduce latency and enhance the efficiency of data transfer. HTTP/2 in gRPC incorporates multiple multiplexing to allow multiple requests and responses over a single TCP connection, which enhances performance and reduces latency. Additionally, it utilizes header compression to reduce network overhead and improve transmission speed. Furthermore, HTTP/2 supports server push, enabling the server to proactively send content to the client without waiting for a request, thereby improving application responsiveness.
* **Streaming:** Streaming allows bidirectional streaming where both the client and server can send and receive data continuously, enabling real-time communication and data transfer. Additionally, it enables the Server-side streaming which helps the server to send a stream of data to the client in response to a single request, making it ideal for streaming media or large datasets. Additionally, it also allows client-side streaming where the client can send a stream of data to the server, allowing for efficient batch processing or uploading large files.
* **Strong Typing:** Strong typing has stricter typing rules at compile time, which implies that errors and exceptions are mostly caught during compilation rather than runtime.

For Example:

If a customer wants to check account balance on the digital banking Application, the gRPC service for the same may be designed as below:

Syntax:

|  |
| --- |
| service BankingService {  rpc GetAccountBalance(AccountRequest) returns (AccountResponse);  }  message AccountRequest {  string correlationId = 1;  string timestamp = 2;  string accountID = 3;  }  message AccountResponse {  string correlationId = 1;  string timestamp = 2;  string accountID = 3;  double balance = 4;  string currency = 5;  } |

Request:

|  |
| --- |
| <correlationid: “dfhj-afhj-fr54-ht3c”>  <timestamp: “2024-10-15T04:35:018Z”>  <accountID: "enchrtsjktyaf=”> |

Response:

|  |
| --- |
| <correlationid: “dfhj-afhj-fr54-ht3c”>  <timestamp: “2024-10-15T04:35:030Z”>  <accountID: "enchrtsjktyaf=">  <balance: 50000.00>  <currency: “INR”> |

1. **GraphQL**

GraphQL is a query language for APIs that provides a flexible and efficient way for clients to request data. Unlike traditional REST APIs, GraphQL allows clients to specify exactly what data they need, reducing over-fetching and under-fetching of data. Developed by Facebook in 2012, it was initially used internally to power Facebook mobile applications. It was open sourced in 2015 and has since gained significant popularity and adoption in the developer community. GraphQL's flexibility, efficiency, and developer-friendliness have made it a compelling alternative to traditional REST APIs.

Below are the key features of graphQL API:

* **Declarative:** Clients specify the data they need, and the server returns exactly that.
* **Strongly Typed:** Ensures type safety and reduces errors.
* **Introspection:** Allows clients to discover the API's capabilities.
* **Batching:** Multiple queries can be combined into a single request.

For Example:

if a customer wants to check account balance on the digital banking Application, the GraphQL API for the same may be designed as below:

Syntax:

|  |
| --- |
| type Account {  correlationId: String!  timestamp: String!  accountID: ID!  balance: Float!  currency: String!  }  type Query {  getAccountBalance(id: ID!): Account  }  query GetAccountBalance($accountID: ID!) {  accountBalance(id: $accountID) {  correlationId  timestamp  accountID  balance  currency  }  } |

Request:

|  |
| --- |
| {  “correlationid”: “dfhj-afhj-fr54-ht3c”,  “timestamp”: “2024-10-15T04:35:018Z”,  "accountID": "enchrtsjktyaf="  } |

Response:

|  |
| --- |
| {  "data": {  "accountBalance": {  "correlationId": "dfhj-afhj-fr54-ht3c",  "timestamp": "2024-10-15T04:35:030Z"  "accountID": "enchrtsjktyaf=",  "balance": 50000.00,  "currency": “INR”  }  }  } |

## 1.3 Recommendation

Based on the project requirement, the APIs needs to be classified in each category mentioned above. Identifying the API type at the start helps to decide the design strategy and implementation standards with respect to authentication, security etc. for the API.

# 2. Need for APIs and API Identification Strategy

## 2.1 When to use the APIs?

Before going into the API design approaches and identification strategy, need of using the APIs for a project should be identified.

**Below are the scenarios in which using APIs is recommended:**

1. **Multiple systems are Involved**

In case multiple heterogeneous systems are interacting, and the data needs to be shared amongst them, it is recommended to take the API based approach as the database of the systems cannot be directly exposed to each other.

**Example:** It may be used to transfer information from Application of one Department to the application of another Department.

1. **Microservice based architecture being implemented**

In case the applications are modular in nature where front-end and back-end are operating as separate services, APIs can be used for inter-service communications.

**Example:** Highly Complex applications (Like Core Banking Solution) broken down into different microservices can use API for internal communication.

1. **Multiple front-end applications with single back end**

In case the applications have multiple front-end systems on different platforms (Like Website, Android Application, IOS Application etc.), APIs may be used to connect the front-end applications to back-end.

**Example:** For Core Banking, omni-channel presence is provided to customers via Website and Mobile applications. These frond-end applications can refer to the common core backend via APIs.

**Below are the scenarios in which using APIs is not recommended:**

1. **Monolithic Applications/systems are Involved**

In case of applications which are monolithic in nature, the Front end can connect with the back end to access the data without need of an API.

**Example:** In case the applications of Departments, where front end and back end are deployed as part of the same application.

1. **High Volume Data is Involved**

In case high volume data needs to be referred, alternative options like Database Replication or file transfers through SFTP may be considered.

1. **Rare Business/Exception Business Scenarios**

In certain scenarios where the data from the system may be needed in rare scenarios (once in year) or only in certain scenarios, where the cost of developing and maintaining the API shall be more than the benefits, the approach of API may be avoided.

**Example:** Ad-hoc requests to obtain certain information from the Database.

## 2.2 API Identification Strategy

API identification strategy defines a process to identify the APIs from the requirements and map them to the design modules appropriately.

API identification strategy contains below steps:

1. Creation of User Stories
2. Creation of High-level design and Low-level design documents
3. Identification of APIs
4. Creation of API Specification

**Assumption**:

System requirements specification (SRS) is present. Agile method is followed for the project.

## 2.3 API Identification Strategy – Example Use case

The below use case depicts the execution of the process flow to identify the APIs and create an API specification for the project.

**2.3.1 Example Use Case details**

**Use Case:** Avail a personal loan

**Scope:**

To design a portal where users can register and apply for a loan, check status of the loan and raise disbursement request.

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Requirement** | **Description** |
| 1 | Registration | First time User should be able to register on the portal. |
| 2 | Login | User should be able to login into the portal. |
| 3 | Forgot Password | User should be able to regenerate a new password in case he has forgotten it. |
| 4 | Change Password | User should be able to change the password after logging in into the portal. |
| 5 | New Loan application | User should be able to raise a new loan application on the portal. |
| 6 | Check KYC | Portal should support integration with the external KYC verification services to verify customer details**.** |
| 7 | Check customer risk assessment | Portal should support integration with external customer risk assessment services. |
| 8 | Perform customer credit score evaluation | Portal should support integration with external credit score report services. |
| 9 | Loan approval/rejection | Portal should have process defined for loan approval/rejection after evaluation of loan documents. |
| 10 | Loan disbursement | Customer should be able to raise loan disbursement request through portal after loan approval. |
| 11 | Notifications | Customer should get notifications on email/SMS about loan status, disbursement status, EMIs etc. |
| 12 | Manage OTP | Portal should support OTP generation and validation for customer authentication. |

**Non-Functional requirements:**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Requirement** | **Description** |
| 1 | Confidentiality | System should ensure that all the sensitive/PII data is protected properly. |
| 2 | Integrity | System should ensure that the integrity of data-in-motion and data-at-rest is preserved. |
| 3 | Availability | System shall achieve 99.9% up time. |

**2.3.2 Use case development**

**Step 1: Creation of user stories**

User stories are created from the SRS. Acceptance criteria of user stories should also cover non-functional requirements.

***Note:*** *It is assumed that this step is executed.*

**Step 2: Creation of High-Level design and Low-Level design documents**

1. High Level Design and Low-Level design documents are created for each Epic and user stories under that Epic.
2. HLD and LLD should define overall strategy for non-functional requirements of API (Confidentiality, Integrity, and Availability).

The High-level design and Low-level design documents will contain following modules:

|  |  |
| --- | --- |
| **Document** | **Modules** |
| High Level Design (HLD) | 1. User Management 2. Loan Management 3. Notifications 4. Manage OTP |
| Low Level Design (LLD) | 1. User Management    1. Registration    2. Login    3. User profile    4. Forgot Password    5. Change Password 2. Loan Management    1. Get Loan applicant details    2. Get Loan details    3. Upload Documents 3. Loan Processing    1. Know Your Customer (KYC) Verification    2. Customer risk assessment    3. Customer Credit Score evaluation    4. Loan approval/rejection 4. Loan disbursement 5. Notification    1. Email notification    2. SMS notification 6. Manage OTP    1. Generate OTP    2. Validate OTP |

**Loan application portal Architecture:**

**Diagram

Description automatically generated**

Figure 2.1: Loan Application Portal Architecture

**Loan application process flow:**

**Diagram

Description automatically generated**

Figure 2.2: Loan Application Request Process Flow

**Diagram

Description automatically generated**

Figure 2.3: Loan Application Processing Flow

**Step 3: Identification of the APIs**

Illustrative APIs for some modules in HLD and LLD are identified as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **HLD Module** | **LLD Module** | **Resource** | **APIs** | **Description** |
| User Management | Registration | Users | POST /users/register | Used to register a user. |
| Login | Users | POST /users/login | Used for user login. |
| User Profile | Users | GET /users/profile/{id} | Used to get user profile details. |
| User Profile | Users | POST /users/profile | Used to update user profile details such as contact, email etc. |
| Registration/Forgot Password | Users | POST /users/check | Used to check if user already exists. |
| Loan Management | Loan Application | Loan | POST /loan/applicants | Used to submit loan applicant details. |
| Loan Application | Loan | POST /loan/details | Used to submit loan details such as type of loan, required amount, loan tenure etc. |
| Loan Application | Loan | POST /loan/documents | Used to upload required loan documents. |
| Loan Processing | KYC Verification | KYC | POST /kyc/verify/PAN | Used to integrate with third party KYC service which verifies PAN  /Driving license  /Aadhar details. |
| KYC Verification | KYC | POST /kyc/verify/drivingLicense |
| KYC Verification | KYC | POST /kyc/verify/aadhar |
| Customer risk assessment | Customers | POST /customers/riskAssessment | Used to integrate with third party service which performs risk assessment of the customer. |
| Customer Credit Score Evaluation | Customers | POST /customers/creditScore | Used to integrate with third party service which gives customer credit score report. |
| Manage OTP | Generate OTP | OTP | POST /otp/generate | Used to generate OTP for customer authentication. |
| Validate OTP | OTP | POST /otp/validate | Used to validate OTP for customer authentication. |
| Notifications | Email notification | Notifications | POST /notifications/email | Used to send notifications to customer over email. |
| SMS notification | Notifications | POST notifications/sms | Used to send notifications to customer over SMS. |

**Third party APIs:**

|  |  |  |  |
| --- | --- | --- | --- |
| **HLD Module** | **LLD Module** | **API** | **Description** |
| Loan Processing | KYC Verification | Third party KYC verification services | External system/vendor providing KYC related services which are consumed by loan application portal. |
| Loan Processing | Customer Risk Assessment | Third party API Customer risk assessment services | External system/vendor providing customer risk assessment services which are consumed by loan application portal. |
| Loan Processing | Customer Credit Score Evaluation | Third party API for getting customer credit score report | External system/vendor providing customer credit score report which is used by loan application portal to evaluate customer. |

**Recommended Step: Mapping APIs with LLD modules and sequence diagrams**

It is recommended to add sequence diagrams for each module in the LLD. It is easier to map the APIs to each step in the sequence diagram and it also helps to identify the process flow gaps.

**Sample process flow for forgot password functionality:**

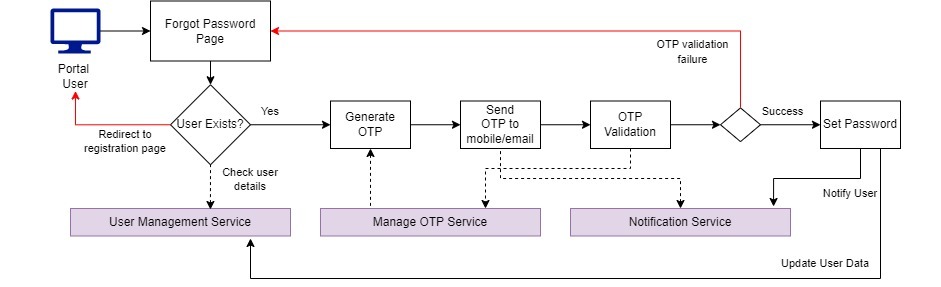
****

Figure 2.4: Forgot Password Functionality Process Flow

**Mapping of APIs with forgot password sequence diagram:**

**Timeline

Description automatically generated with medium confidence**

Figure 2.5: Forgot Password Sequence Diagram

**Step 4: Creation of API Specification**

Once the APIs are identified, they should be defined in detail with request, response parameters, field descriptions and validations in an API Specification file.

It is recommended to use Swagger[[3]](#footnote-3) and RAML[[4]](#footnote-4) for defining API specifications. Kindly refer the API specification standards checklist section which covers guidelines and best practices for:

1. Request and response
2. Field descriptions
3. Validations
4. Enumeration values
5. Security parameters
6. Error Codes
7. Response structure
8. Version

## 2.4 Layered Architecture

After APIs are identified, the development of APIs should follow the modular or layered approach. Related code should be placed together (cohesion) and the dependencies between code should be purposeful and understandable (coupling). One way to achieve this is to place individual code modules into larger units called layers. A layer consists of a set of related modules which are at the same level of abstraction. Layers form a hierarchy so that a layer is only allowed to depend on lower layers.

## 2.4.1 Layered Architecture Principles[[5]](#footnote-5)

* **Separation of Concerns**: Each layer should focus on a specific aspect of the application, preventing tight coupling between components. This principle promotes modularity and maintainability by dividing the application into distinct layers with well-defined responsibilities. By separating concerns, it becomes easier to understand, develop, and test each layer independently.
* **Modularity**: Layers should be designed as independent modules that can be developed, tested, and deployed separately. Modularity enhances flexibility and reusability. Changes to one module are less likely to affect others, reducing the risk of unintended side effects. It also facilitates parallel development and testing.
* **Abstraction**: Layers should interact through well-defined interfaces, hiding implementation details from other layers. Abstraction promotes loose coupling between layers, making the system more flexible and easier to maintain. It allows for changes to be made within a layer without affecting other layers, as long as the interface remains consistent.
* **Scalability**: Layers should be designed to be scalable, allowing for efficient resource allocation and performance optimization. Scalability is essential for handling increasing workloads and ensuring the application's performance. By designing layers to be scalable, it becomes easier to add resources (e.g., servers, hardware) as needed to meet demand.
* **Reusability**: Components within a layer should be designed to be reusable in other applications or contexts. Reusability can reduce development time and effort by leveraging existing code. However, it's important to balance reusability with maintainability and security considerations.

## 2.4.2 Layers in Layered Architecture

The layered architecture can contain the layers as below:

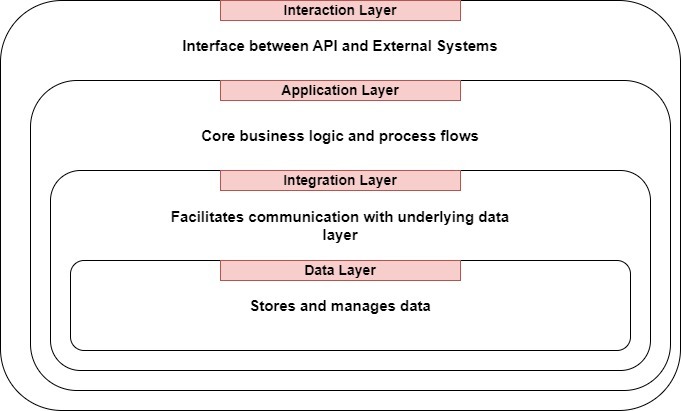


Figure 2.6: Layers in Layered Architecture

1. **Interaction Layer**: The interaction layer serves as the interface between the API and external systems, applications, or users. It manages incoming requests, handles authentication, and communicates responses. As the gateway for API interactions, it provides communication with the outside world.
2. **Application Layer**: The heart of the API architecture resides in this layer. The application layer encapsulates the core business logic and workflows. It defines how the API functions and what operations it can perform based on the received requests. It hosts the business logic and functionality, interpreting and processing incoming requests. It also executes the necessary operations and orchestrates the overall behavior of the API.
3. **Integration Layer**: The integration layer enables interoperability by managing the integration of various systems and services. It sits between the data layer and the application layer, facilitating communication and coordination of data. It manages the interaction with the underlying database in the application. It also plays a vital role in tasks such as data transformation, validation, and ensuring continuous information flow between different components.
4. **Data Layer**: The data layer is responsible for storing, retrieving, and transforming data. It typically includes databases, data storage systems, and persistent data components. The primary role of the data layer is to ensure that information is stored and retrieved efficiently.

Based on the Loan Application portal use case mentioned in above sections, the layered Architecture can be explained as below:

A computer screen shot of a application

Description automatically generated

Figure 2.7: Layered Architecture Example

1. **Interaction Layer**: Interface between API and external systems or users.
   * **Login/Registration form**: Collects username and password, validates credentials, and redirects to the appropriate dashboard.

* **Loan application form**: Collects loan amount, tenure, and other required information from the user.
* **Status check page**: Displays the current status of the loan application.
* **Notification center**: Displays email and SMS notifications related to the loan.

1. **Application Layer**: Core business logic and process flows.

* **LoanApplicationProcessor**: Validates loan applications, checks KYC and credit score, and forwards applications for approval.
* **KYCVerificationService**: Integrates with external KYC verification services to verify customer details.
* **CreditScoreEvaluator**: Integrates with external credit score report services to evaluate the customer's creditworthiness.
* **NotificationManager**: Sends email and SMS notifications to the customer based on loan status changes.

1. **Integration Layer**: Facilitates communication with underlying data layer.

* **Customerinformation**: Handles operations related to customer registration, login, and profile updates.
* **Loandetails**: Stores and retrieves loan application data.
* **Creditscore**: Stores, calculates, and updates the customer's credit score.

1. **Data Layer**: It will be storing customer information, loan applications, loan statuses, notifications, and other relevant data.

# 3. API Design Patterns

## 3.1 API Design Patterns Summary

**An Application Programming Interface (API)** is a set of clearly defined methods of communication between software components without any user intervention. API consists of a set of instructions and standards to be followed by the participating applications.

API design patterns provide a description or templates to solve specific, recurring API design problems that any software architects and API designers would like to adopt in their API designs.

This document covers below design patterns for APIs:

**Based on the type of response**:

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Design Pattern** | **Description** |
| 1 | Synchronous | * Provides the response in real time. * Client waits for the server to provide the response. |
| 2 | Asynchronous (Reactive – Push response) | * Provides the response in near real time. * Client does not wait for the server to respond. * The server acknowledges the request and pushes the data over another endpoint to the client as and when it gets ready. |
| 3 | Asynchronous (Notification – Pull response) | * Provides the response in near real time. * Client does not wait for the server to respond. * The server acknowledges the request and notifies client when data is ready. Client pulls the data using separate endpoint. |

**Based on the functionality:**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Design Pattern** | **Description** |
| 1 | Atomic API | * The API supports a single functionality. For example: User registration. |
| 2 | Composite API | * The API supports multiple functionalities. * A composite API can accept a single request for multiple functionalities which are interdependent and provide a single response after executing all the requests. |

**Based on the REST architecture:**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Design Pattern** | **Description** |
| 1 | HATEOAS | * Hypermedia as the Engine of Application State (HATEOAS) is feature of the REST architecture where server provide information to client about next step to be executed through hypermedia. |

## 3.2 Synchronous APIs

In case of synchronous APIs, the data is made available by the API in real time. The application requests for data and waits until the API returns the data.

Table

Description automatically generated

Figure 3.1: Synchronous API Design Pattern

**Error handling of Synchronous APIs:**

As synchronous APIs provide response on the same thread on which client has invoked the request, errors can also be captured and handled on the same thread.

The Synchronous API error handling strategy should follow the common error handling strategy.

## 3.3 Asynchronous APIs (Reactive – Push response)

In the case of asynchronous APIs, the data is made available at a later point in time.

This design is based on a “Push” mechanism. When the requested data is ready, the application calls an API endpoint provided by the client and sends the data.

Diagram

Description automatically generated

Figure 3.2: Asynchronous API - Response Based Design Pattern

**Reactive APIs**

Reactive APIs are based on the asynchronous API – response-based design pattern.

When a client calls a reactive API, server sends the data using multiple events. Client receives events one by one and collate the data from all events to build a complete response.

Below three basic events are given by the server:

1. Data event
2. Success event indicating the response events are ended
3. Error event indicating error in providing the response

**Error handling in reactive APIs:**

The reactive APIs should follow the common error handling strategy.

In addition, client should be able to handle the errors in server sent events and can choose to display partial data or complete failure of the response accordingly.

## 3.4 Asynchronous APIs (Notification – Pull response)

In the case of asynchronous APIs, the data is made available at a later point in time or near real time.

This design is based on a “Pull” mechanism. When the requested data is ready, the application notifies the client that data is ready. Client then pulls the data using another API exposed by the application.

Diagram

Description automatically generated

Figure 3.3: Asynchronous API - Notification Based Design Pattern

**Error handling of Asynchronous APIs (Notification – Pull response):**

In this design pattern, error handling needs to be present at two places:

1. Errors returned when the API request is made
2. Errors returned when the client tries to pull the data after receiving data ready notification from server.

For both the above scenarios, common error handling strategy should be followed.

## 3.5 Atomic APIs and Composite APIs

**Atomic API** supports a single functionality i.e., for each step in the process flow, a separate API will be defined.

For Example: in case of user registration, there can be two atomic APIs defined; first, to check if the user exists and second to create the new user.

**Composite API** support multiple functionalities i.e., for a process flow, a single API can be defined which takes the input requests for all the steps and a single response is returned.

**API granularity** is a critical aspect to be considered while designing atomic or composite APIs. API granularity decides the level at which the functionalities in a project are segregated. If API granularity is more in the project, there will be many atomic APIs otherwise there will be fewer composite APIs.

**Comparison of Atomic and Composite APIs:**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Atomic API** | **Composite API** |
| **API Granularity Type** | Fine grained | Coarse grained |
| **Total No. of APIs in the Project** | More | Fewer |
| **API Performance** | Many calls need to be made over network which can affect performance | Fewer calls to be made over the network which might be useful in some cases |
| **Error Handling** | Errors will be handled at each API separately. | As number of requests are bunched under a single API, two error handling strategies can be used: return partial success or return complete failure in case one of the request fails. |

**Considerations for designing Composite APIs:**

1. **Request representation:** how many requests will be bundled under a composite API, their respective method, and URLs.
2. **Authentication:** Whether the authentication will be done at composite API level or at individual sub request level as well.
3. **Response rendering:** The final response structure including the response data for all the sub requests.
4. **Error handling:** Whether to return a partial success or fail the complete request in case one of the sub requests returns errors.

**Error handling of Atomic APIs:**

Error handling strategy for atomic APIs should follow the common error handling strategy.

**Error handling of Composite APIs:**

Composite API errors can be handled in two ways:

1. Process all the requests even if a particular request gives error and return partial success response.
2. Stop the processing in case a particular request gives error and return a complete failure response. Also, in this case, roll back of previous successful requests should also be carried out.

Also, the common error handling strategy will be applicable to each sub request of composite API.

## 3.6 HATEOAS APIs

HATEOAS stands for **“Hypermedia as the Engine of Application State”** and it is a component of REST architecture.

With the use of HATEOAS, the client needs minimal knowledge about how to interact with a server. This is made possible by the application responding to the client’s requests with dynamically generated information using hypermedia.

When accessing a webpage through a browser, users can interact with the webpage by using buttons, inputs, clicking on links, etc. However, traditional API responses have no such functionality present to allow an application to interact with the server through the response. A HATEOAS API sends the response data as well as specify the next related action to be performed by the client. This enables the client to move from one application state to the next just by interacting with the details contained in the responses by the server.

**For example:**

In a loan application form, once the client submits basic details, the response of API will have link to the next API call in which required loan details can be filled.

API Request URL: /v1/userDetails

API Response:

|  |
| --- |
| HTTP/1.1 200 OK  Content-Type: application/+json  Content-Length: ...  {  "UserDetailsResponse": {  "userid": "uid123",  "message": "user details added successfully"  "links": {  "href": "/v1/uid123/loandetails"  }  }  } |

**Error handling for HATEOAS APIs:**

HATEOAS APIs error handling strategy should follow common error handling strategy. Also, for each http error code, a link should be sent to the client to take the necessary action.

For example, in case of 401 authentication failure, link to the forgot password page/login page should be provided.

## 3.7 Common Error Handling Strategy

The error handling strategy for APIs should handle the common http status error codes as below:

* **4XX codes related to client-side error**
  + 4XX error codes happen due to incorrect data passed from client side such as missing authentication data or malformed request.
  + Requests with 4XX error response codes cannot be retried as it will return the same error every time.
  + However, server should provide proper error code along with error details in the response so that client can easily figure out and fix the issue.
* **5XX codes related to server-side error**
  + 5XX error codes happen due to error at server side such as memory issues, application unavailable, server outages etc.
  + Requests with 5XX error response codes can be retried as the same request can get processed once the server is up and running.
  + Hence, client should setup a retry mechanism to handle server-side error.
  + Also, at server side a circuit breaker[[6]](#footnote-6) functionality should be in place to avoid unnecessary hits to the server when server is down.

## 3.8 API Design Pattern – Example Use case

**Use Case:** Avail a personal loan

**Scope:**

To design a portal where users can register and apply for a loan, check status of the loan and raise disbursement request.

**Functional requirements:**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Requirement** | **Description** |
| 1 | Registration | First time User should be able to register on the portal. |
| 2 | Login | User should be able to login into the portal. |
| 3 | Forgot Password | User should be able to regenerate a new password in case he has forgotten it. |
| 4 | Change Password | User should be able to change the password after logging in into the portal. |
| 5 | New Loan application | User should be able to raise a new loan application on the portal. |
| 6 | Check KYC | Portal should support integration with the external KYC verification services to verify customer details**.** |
| 7 | Check customer risk assessment | Portal should support integration with external customer risk assessment services. |
| 8 | Perform customer credit score evaluation | Portal should support integration with external credit score report services. |
| 9 | Loan approval/rejection | Portal should have process defined for loan approval/rejection after evaluation of loan documents. |
| 10 | Loan disbursement | Customer should be able to raise loan disbursement request through portal after loan approval. |
| 11 | Notifications | Customer should get notifications on email/SMS about loan status, disbursement status, EMIs etc. |
| 12 | Manage OTP | Portal should support OTP generation and validation for customer authentication. |

**Non-Functional requirements:**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Requirement** | **Description** |
| 1 | Confidentiality | System should ensure that all the sensitive/PII data is protected properly. |
| 2 | Integrity | System should ensure that the integrity of data-in-motion and data-at-rest is preserved. |
| 3 | Availability | System shall achieve 99.9% up time. |

**Loan application portal Architecture:**

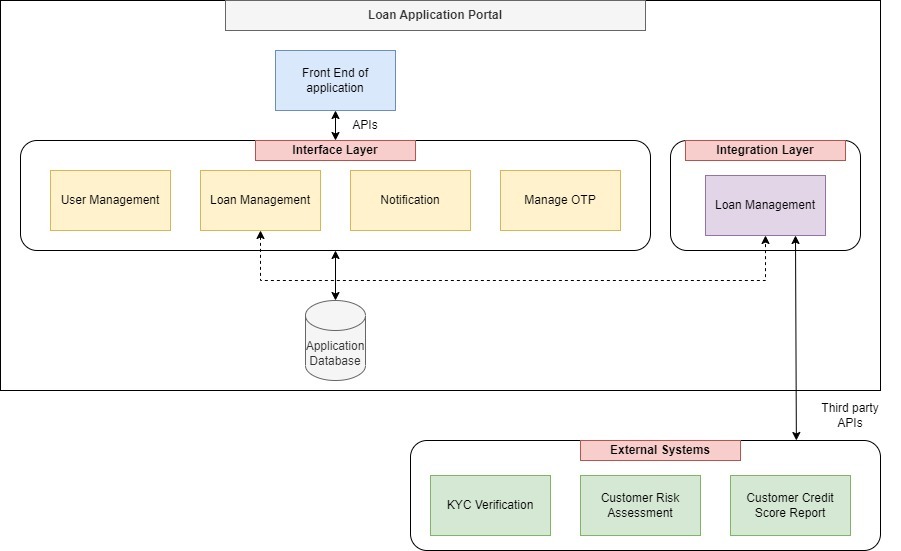
****

Figure 3.4: Loan Application Portal Architecture

**Loan application process flow:**

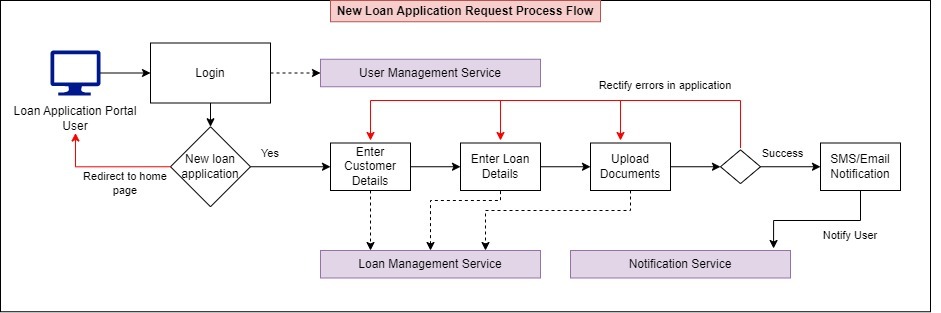
****

Figure 3.5: Loan Application Request Process Flow

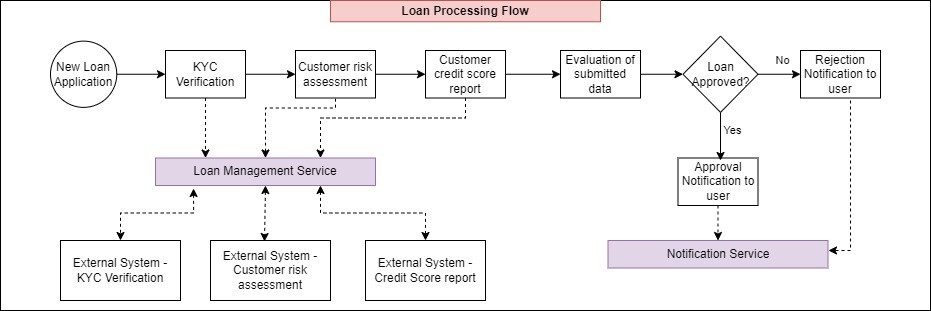
****

Figure 3.6: Loan Application Processing Flow

**3.8.1 Synchronous API example**

The API used to log in into the loan application portal is a synchronous API.

URL: /users/login

Request:

|  |
| --- |
| {  "ver": "1.0.0",  "correlationId": "kjhd-6487-adkh-8753",  "timestamp": "2022-12-15T04:35:018Z",  "username": "abc",  "password": "\*\*\*\*"  } |

Response:

|  |
| --- |
| {  "ver": "1.0.0",  "correlationId": "kjhd-6487-adkh-8753",  "timestamp": "2022-12-15T04:35:049Z",  "message": "User authenticated successfully"  } |

**3.8.2 Asynchronous API – push response example**

The API used for fetching the customer credit score from third party can be implemented using asynchronous API – push response pattern.

There will be two APIs implemented for fetching the customer credit score.

**API at Credit score external system** which will accept the request from Loan application portal.

URL: POST /creditScore/request **(External API)**

Once the Credit score data is ready, Credit score external system will call the API exposed by loan application portal to send the data.

URL: POST /customer/creditScore/fetch **(API hosted by loan application portal)**

**3.8.3 Asynchronous API – pull response example**

The API used for fetching the customer credit score from third party can be implemented using asynchronous API – pull response pattern.

There will be two APIs implemented by external credit score system to request and retrieve credit score report.

**External APIs:**

1. POST /creditScore/request
2. POST /creditScore/fetch

One API will be implemented by Loan application portal to receive notification once the data is ready.

API:

1. /customer/creditScore/notification

Steps:

1. Loan application portal will call the POST /creditScore/request API to request credit score data from external system. External system will provide an acknowledgement and a unique id to retrieve the data.
2. External system will get the data ready and will provide a notification on /customer/creditScore/notification API to loan application portal.
3. Loan application portal will retrieve the credit report data from external system using POST /creditScore/fetch API.

**3.8.4 Atomic and Composite API example**

The APIs used to submit loan application details can be designed using atomic or composite API design pattern.

**Atomic APIs:**

1. POST /loan/applicant
2. POST /loan/details
3. POST /loan/documents

**Composite API:**

1. POST /loan/application

This single API will take inputs for the three atomic APIs mentioned above in the request.

For example:

Request:

|  |
| --- |
| {  "ver": "1.0.0",  "correlationId": "kjhd-6487-adkh-8753",  "timestamp": "2022-12-15T04:35:018Z",  "applicantDetails": {  "method": "POST"  "url": "/loan/applicants"  "requestbody": {  ....  }  },  "loanDetails": {  "method": "POST"  "url": "/loan/details"  "requestbody": {  ....  }  },  "uploadDocuments": {  "method": "POST"  "url": "/loan/documents"  "requestbody": {  ....  }  }  } |

Response:

|  |
| --- |
| {  "ver": "1.0.0",  "correlationId": "kjhd-6487-adkh-8753",  "timestamp": "2022-12-15T04:37:018Z",  "applicantDetails": {  "statusCode": "200",  "message": "applicant details added successfully"  },  "loanDetails": {  "statusCode": "200",  "message": "loan details added successfully"  },  "uploadDocuments": {  "statusCode": "200",  "message": "documents uploaded successfully"  }  } |

**3.8.5 HATEOAS API example**

The APIs used to submit loan application details can be designed using HATEOAS API design pattern.

Below APIs used to submit loan application details will be executed sequentially. Hence, the link to call the next API can be provided in the response of previous API.

For example:

APIs:

1. POST /loan/applicant
2. POST /loan/details
3. POST /loan/documents
4. POST /loan/applicant

Request:

|  |
| --- |
| {  "ver": "1.0.0",  "correlationId": "kjhd-6487-adkh-8753",  "timestamp": "2022-12-15T04:35:018Z",  "name": "abc",  "address": "",  …..  } |

Response:

|  |
| --- |
| {  "ver": "1.0.0",  "correlationId": "kjhd-6487-adkh-8753",  "timestamp": "2022-12-15T04:37:018Z",  "statusCode": "200",  "message": "applicant details added successfully",  "links": {  "href": "/loan/details",  }  } |

1. POST /loan/details

Request:

|  |
| --- |
| {  "ver": "1.0.0",  "correlationId": "kjhd-6487-adkh-8753",  "timestamp": "2022-12-15T04:35:018Z",  "loanAmount": "",  "tenure": "",  …..  } |

Response:

|  |
| --- |
| { "ver": "1.0.0",  "correlationId": "kjhd-6487-adkh-8753",  "timestamp": "2022-12-15T04:37:018Z",  "statusCode": "200",  "message": "loan details added successfully",  "links": {  "href": "/loan/documents",  } |

# 4. API Development Life Cycle

## 4.1 API Development Life cycle

**What is API Development Life Cycle?**

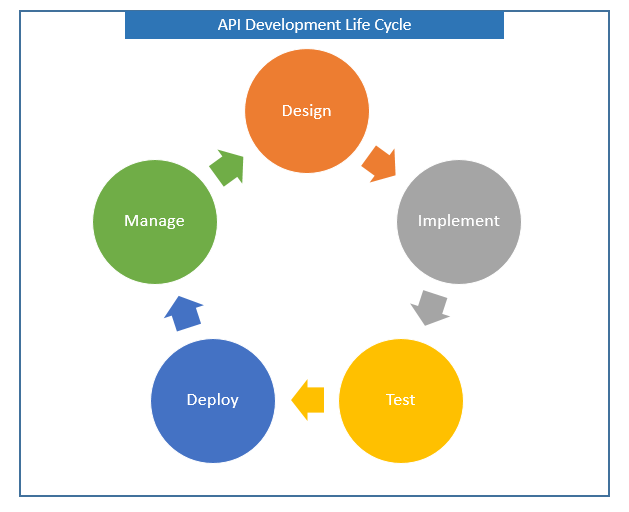
API development life cycle is the entire lifespan of an API that begins at a planning stage and ends when an API is retired. When APIs for a project are identified, each API should go through the API development life cycle.

The stages of API development life cycle are:

1. Design
2. Implement
3. Test
4. Deploy
5. Manage

***Note:*** *The API development life cycle depicts only the lifecycle of APIs identified for the project and does not include development of API developer portal, API Gateway etc. If such components are to be built from scratch for the project, separate planning and effort estimation is required.*

This section illustrates the development life cycle of an API.

*Figure 4.1: API Development Life Cycle*

**4.1.1 Design Phase**

There are commonly used approaches to design API –

1. Code First
2. API-Design First
3. **Code First Approach**

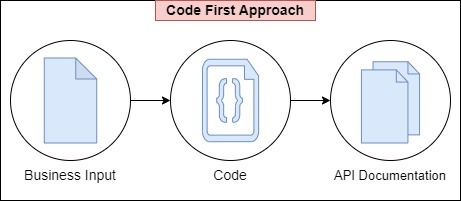


Figure 4.2: API Design - Code First Approach

A code-first API process focuses on implementing the API first, and then creating the API description document after the implementation. In this model, the API implementation is the source of truth for the API and drives the rest of the development process. This approach is suitable while creating APIs from existing code base or legacy application.

1. **API-Design First Approach**

A diagram of a test

Description automatically generated

Figure 4.3: API Design - Design First Approach

It is recommended to follow the **API-Design First**[[7]](#footnote-7) approach while designing the APIs. **API-Design First** approach means the APIs are designed in an iterative way by taking the feedback from the API consumers and improving the APIs till all the parties are satisfied.

The API specification once finalized in design phase should be concrete and immutable. This means that the API specification should be the single source of truth and code should not deviate from or override the spec. If there are any gaps/issues identified, instead of changing the implementation directly, the API specification should be revisited, the necessary changes performed in the API spec and accordingly the code should be modified.

**Benefits of API-Design First Approach**

* **Ability to work in parallel – The API specification can be used to create mock server and mock the APIs which allows API consumers to test the API before it is implemented.**
* **Ensuring a positive developer experience – The API specification can be used to create documentation, client libraries and SDKs using various third-party tools**[[8]](#footnote-8)[[9]](#footnote-9) **to improve the developer experience.**
* **Reduce development costs - Fixing issues once the API is coded costs far more than fixing them during the design phase. The development costs can be reduced if the API is designed properly in the design phase itself. Also, the development costs can be reduced by designing reusable APIs across projects.**

The design phase can be executed as below:

|  |  |
| --- | --- |
| **Input** | SRS, HLD, LLD, List of APIs identified |
| **Design phase execution** | 1. **Design:** The first draft of API specification for an API is created by having discussions with internal team, business owner and API consumers with below parameters:    1. Request and response    2. Field descriptions    3. Validations    4. Enumeration values    5. Security parameters    6. Error Codes    7. Response structure    8. Version   The section API specification standards checklist should be referred here.  **Publish:** The first draft of API specification is published to the API consumers. The API specification can be shared over the developer portal or via any other channels such as email.   1. **Simulate:** The API specification will be mocked and tested by API consumers and feedback will be provided. The feedback can be accepted over various channels such as meetings, email, community forums etc. 2. **Validate:** The feedback will be incorporated in the API specification and updated draft will be shared with API consumers. 3. The steps 1 to 4 will be repeated till all the parties are satisfied. |
| **Security Controls** | 1. Threat Modeling    1. Application Architecture    2. Infrastructure Architecture 2. Architecture security review |
| **Output** | API Specification which covers both the functional and non-functional requirements of the project.  The API specification created should be concrete and immutable which should not change drastically in the later phases. |
| **Recommended tools** | Swagger 2.0, Swagger editor, RAML |

***Note: The API specification created in the design phase should be the single source of truth****. The code should not deviate or override the specification at any point in time. If there are implementation challenges and changes need to be done, API specification should be revisited first and steps 1 to 4 in above table should be re-executed. After the spec change, code should be modified to match the specification.*

**4.1.2 Implementation Phase**

Once the API specification is created in the design phase, actual code can be implemented by the developers for that API.

Implementation phase can be executed as below:

|  |  |
| --- | --- |
| **Input** | SRS, HLD, LLD, API Specification |
| **Implementation phase execution** | 1. The API is implemented by the developer as per the API specification. 2. Developer performs the unit testing of the API. 3. Developer writes automated unit test scripts for the API covering the happy path as well as failure scenarios. The automated unit tests are expected to be integrated in the CI/CD pipeline and should be executed as part of each build. 4. The API is implemented to the lower environment such as dev. 5. The API is tested by the developer in dev environment. 6. A postman collection is created by the developer for white box[[10]](#footnote-10) testing of the API. The postman collection is attached in the LLD for reference. White-box testing enables to test the specific API for its functionalities. 7. Once the testing is complete, the API is released for the API consumers to integrate with their respective lower environment. 8. If there are any issues reported by API consumers in the implementation phase, the API specification should be revisited and modified first before making the code changes. |
| **Security Controls** | 1. SAST 2. DAST 3. OWASP Top 10 web application checklist 4. OWASP Top 10 API checklist |
| **Output** | API implementation completed by developers and tested by API consumers and signed off for the QA release. |
| **Recommended tools** | Swagger 2.0, Swagger editor, RAML, Developer portal developed in respective technology stack of the project(optional), project specific technology stack, CI/CD tools  White-box testing: Postman collection[[11]](#footnote-11), postman scripts[[12]](#footnote-12), SoapUI |

***Note:*** *Static application security testing (SAST) should be performed at least once in the implementation phase itself to identify critical issues early in the development and the SAST findings to be closed before the QA sign off.*

**4.1.3 Testing Phase**

Once the development is complete and white-box testing is performed as part of the implementation phase, the APIs are ready to be released for QA. Testers should perform the black box testing of the APIs with test cases covering the end-to-end process flow. QA testers will be performing end to end testing of multiple APIs as part of the System Integration Testing.

Also, Vulnerability assessment and penetration testing (VAPT) should be performed in parallel or after the QA sign off.

The QA testing phase will be executed as below:

|  |  |
| --- | --- |
| **Input** | SRS, HLD, LLD, API Specification, Source code of API implementation, API implementation deployed on QA (SIT) environment |
| **Testing phase execution** | 1. QA team will perform the black box testing of API implementation with test cases covering the end-to-end process flow. 2. Dev team will fix the defects. For fixing the defects, API specification should be modified first and then the corresponding code changes should be performed. 3. QA team will provide sign off after the release is stable. |
| **Output** | API implementation signed off from QA team |
| **Recommended tools** | Regular tools used by testing team |

The security testing phase will be executed as below:

|  |  |
| --- | --- |
| **Input** | API implementation signed off by QA team |
| **Testing phase execution** | 1. Security team will perform the VAPT. 2. Dev team will fix the VAPT findings. 3. Security team will perform the SAST and provide report. 4. Dev team will fix the SAST findings. |
| **Security Controls** | 1. Vulnerability Assessment 2. Penetration Testing |
| **Output** | API implementation signed off from Security team |
| **Recommended tools** | Regular tools used by testing team |

The UAT testing phase will be executed as below:

|  |  |
| --- | --- |
| **Input** | API implementation signed off by QA and Security team |
| **Testing phase execution** | 1. Business team will perform the UAT of API implementation with test cases covering the end-to-end process flow. 2. Dev team will fix the defects. For fixing the defects, API specification should be modified first and then the corresponding code changes should be performed. 3. Business team will provide sign off after the release is stable. |
| **Security Controls** | 1. Vulnerability Assessment 2. Penetration Testing |
| **Output** | API implementation signed off from UAT team |
| **Recommended tools** | Regular tools used by testing team |

**4.1.4 Deploy Phase**

Once the UAT testing is complete, the API implementation will be deployed to production.

The Deploy phase will be executed as follows:

|  |  |
| --- | --- |
| **Input** | API implementation signed off by UAT team, production environment infrastructure, API developer portal (optional), API gateway |
| **Deploy phase execution** | 1. Production readiness steps performed by production team including environment setup, onboarding of API consumers, API developer portal launch, distribution of API keys, API gateway deployment etc. 2. Production team will deploy the API implementation into the production environment. 3. Production support team will provide support and perform defect fixes. 4. The bug fixes can be managed as per the branching/release strategy[[13]](#footnote-13) adopted by the project. |
| **Security Controls** | 1. Vulnerability Assessment 2. Penetration Testing 3. DAM and SIEM Integration |
| **Output** | API implementation live on production |
| **Recommended tools** | NA |

**4.1.5 Manage Phase**

Once the API is live on the production, it enters in the manage phase. The API versioning strategy is applied to the APIs in manage phase. After the API is live, any change required in the API should be carefully articulated and analyzed. A robust versioning strategy should be in place to handle all the possible scenarios in which the API can change and release new version of the API smoothly.

**Versioning Strategy:**

It is recommended to use semantic versioning[[14]](#footnote-14) in the header of an API and also use the URL versioning.

For example:

Version in header -

Version: 2.0.0

Version in URL –

/v2/users

The detailed versioning strategy is described in a separate section API Governance Process and Versioning Strategy.

**API Monitoring:**

API Monitoring refers to the practice of monitoring APIs, most commonly in production, to gain visibility into performance, availability, and functional correctness.

As the application is built on the foundation of APIs, API monitoring is extremely important to ensure that consumers of the API always have access to it. If the application is consuming some third-party APIs, then it is important to monitor those APIs as well.

Many API management tools[[15]](#footnote-15)[[16]](#footnote-16) offer API monitoring dashboards out of the box. API monitoring portal can also be developed specific to each project.

A complete API monitoring and analytics solution should contain below functionalities:

1. **Operational metrics:** real-time, end-to-end visibility of API usage.
2. **Developer metrics:** API consumer adoption metrics with visibility to top consumers, churn risk and quota status. They are important to understanding API program business success.
3. **API usage metrics:** measure most and least popular, volume, traffic, and Transactions per second.
4. **API performance metrics:** API quality metrics with latency and error rates.
5. **API infrastructure metrics:** analysis of resource bottlenecks, error rates and latency.
6. **Traffic monitoring:** monitor and troubleshoot individual transactions, including drilling into processing steps and message content to perform root-cause analysis.
7. **Security Incident and Event Monitoring:** Monitor API transaction security incident and using SIEM and DAM tools. Perform the root cause analysis.

## 4.2 Roles and responsibilities of the project team

The project team generally consists of below members:

1. Project Manager
2. Architect
3. Developer
4. Sr. Developer / Team leads
5. Business analysts
6. DevOps consultant
7. API Administrator
8. Testers
9. Production support team

The responsibilities of the project team members with respect to API Development life cycle are as below:

|  |  |  |
| --- | --- | --- |
| **Phase** | **Task** | **Team members** |
| **All** | API release management | Project Manager, Architect |
| **Design** | API Identification Strategy | Business Analysts, Architects, Team Leads |
| Allocate and manage infrastructure | API admin, Project manager, Architect |
| API Versioning Strategy | Architect, DevOps Consultant |
| **Implement** | API development | Developer |
| API development review | Team Lead, Architect |
| Publish API to developer portal | Team Lead |
| API white-box testing | Developer |
| **Test** | API Black-box testing | Testers |
| Bug fixes and testing support | Developer, Team Lead |
| **Deploy** | API deployment | Architect, DevOps Consultant, Project Manager |
| **Manage** | API production support | Production support team |

# 5. API Specification Standards Checklist

## 5.1 REST API Design Features

Diagram

Description automatically generatedBelow diagram depicts the REST API Design features. It covers many aspects which are mentioned in this book.

Figure 5.1: REST API Design Features

As API Design is the initial phase of writing the APIs for a project, it is crucial to write the correct specifications around this. A checklist is defined in this chapter which can be used to ensure that the design features are incorporated while writing the API Specification.

## 5.2 API Specification Standards Checklist

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr. No.** | **Category** | **Item** | **Incorporated?**  **(Y/N)** | **Comments** | **Examples** |
| 1 | REST API Design best practices | Use nouns instead of verbs in endpoint paths. |  |  | Use /accounts instead of /getAccount |
| 2 | REST API Design best practices | Name collections with plural nouns. |  |  | Use /accounts instead of /account |
| 3 | REST API Design best practices | Use appropriate REST method as per operation on resource. |  |  | POST for create |
| 4 | REST API Design best practices | Avoid using PUT and DELETE methods for APIs. |  |  |  |
| 5 | REST API Design best practices | Use lowercase or camelCase or - in the endpoint paths. |  |  | /accounts or /savingAccounts or /saving-accounts |
| 6 | REST API Design best practices | Use input, output type as application/json |  |  |  |
| 7 | REST API Design best practices | Use camelCase for the payload fields and PascalCase for elements. |  |  | "Details": {  "emailId": "abc@mail.com",  "name": "abc"  } |
| 8 | REST API Design best practices | Add Enum values, JSON schema level validations wherever possible. |  |  | "AccountDetails": {  "type": "SAVINGS" //Enum values can be ["SAVINGS", "CURRENT"] } |
| 9 | REST API Design best practices | Define required and optional fields in payload. |  |  |  |
| 10 | REST API Design best practices | Use nesting resources for hierarchical objects. |  |  | /accounts:  /savings:  /current: |
| 11 | REST API Design best practices | Allow filtering, sorting, and pagination as per requirement. |  |  | Filtering on accid: /account/{accid}  Sorting: Provide a field or header with values in request such as order: Ascending, descending and the attribute on which sorting to be performed.  Pagination: Provide attributes such as PageNumber, Start, End and page size in request body or headers. |
| 12 | REST API Design best practices | Handle errors gracefully and return standard error codes. |  |  | 400 - bad request 401 - unauthorized 404 - not found 500 - internal server error and so on. |
| 13 | REST API Design best practices | For passing sensitive information such as PII data, session tokens always use POST method. |  |  |  |
| 14 | REST API Design best practices | Preferably use URI parameters instead of query parameters. |  |  | Use /account/{accid} instead of /account?accid="123" |
| 15 | REST API Design best practices | Add audit logging fields in request and response of API. |  |  | {  correlationId:"",  timestamp: ""  } |
| 16 | REST API Design best practices | Define common fields/object once and reuse them in API specification. |  |  |  |
| 17 | API Documentation | Add appropriate description and example for each field in the request and response of an API. |  |  |  |
| 18 | API Documentation | Add appropriate description to each API. |  |  |  |
| 19 | API Documentation | Group the APIs logically and tag them with common group name. |  |  | POST /accounts and PUT /accounts will be grouped under Account Management. |
| 20 | API Versioning | Add major version in the URI Path. |  |  | POST /v1/accounts |
| 21 | API Versioning | Use semantic version for API Versioning. |  |  | V 1.1.3 - Major.minor.patch |
| 22 | API Versioning | Add semantic version in the header of the API. |  |  | version: 1.1.3 |
| 23 | API Versioning | Add optional deprecation and decommission headers in the API which needs to be populated once the API is deprecated or decommissioned. |  |  | x-api-deprecation-date: Sun, 4 Sep 2022 23:59:59 GMT+5:30  x-api-deprecation-info: << Link of latest API version >> x-api-decommission-date: Sun, 5 Mar 2023 23:59:59 GMT+5:30 x-api-decommission-info: << Link of latest API version >> |
| 24 | API Security | Always Use TLS or mutual TLS for API endpoint as per the requirement. |  |  |  |
| 25 | API Security | Use security measures such as client id/secret, Oauth 2.0, JWT for authentication and authorization. |  |  |  |
| 26 | API Security | Use signatures such as Detached JWS to ensure integrity of data passed using API. |  |  |  |
| 27 | API Security | Use security measures such as client id/secret, IP whitelisting, session token, rate limiting policies to access the APIs. |  |  |  |
| 28 | API Security | Encrypt the specific fields in payload containing sensitive data such as PII, session token whenever possible. The entire payload can also be encrypted for higher level of security. |  |  | Encrypt a specific field: {  correlationid: "dajhs-fjh3k5-afkj2-afj2", timestamp: "",  "Customer": {  "username": "abc",  "password ": "dfjehrtjhsaxwr="  } }  Encrypt the complete payload: {  correlationid: "dajhs-fjh3k5-afkj2-afj2",  timestamp: "",  encryptedReq: "dfhkjhghahfjhfafdf=" } |
| 29 | Error Handling | The error response should be defined in such a way that it should not give too much technical information. For example, stack trace or name of a class. |  |  |  |
| 30 | Error Handling | The errors can be categorized as BUSINESS or TECHNICAL and accordingly a strategy should be in place to handle different error types. |  |  |  |
| 31 | Error Handling | The Error response should contain these minimum fields in addition to audit logging fields- responseStatusCode, errorType, errorCode, errorDescription. |  |  |  |

# 6. API Logging and Error Handling Standards

## 6.1 Overview of API Logging

**An Application Programming Interface (API)** is a set of clearly defined methods of communication between software components without any user intervention. API consists of a set of instructions and standards to be followed by the participating applications. APIs operate on an agreement of inputs and outputs and are independent of any specific programming language.

**An API log** is the automatically produced and time-stamped documentation of events relevant to a particular system.

Logging plays a crucial role to support the application once it is live in production. Without proper logging, application support team can face difficulties to find the root cause of the issue highlighted in the application. As the modern applications rely heavily on the APIs for integration, it is extremely important to implement proper logging at API level as well.

**Goals for API Logging:**

1. **End to end correlation:**

As a single API caters to a specific operation, generally multiple API calls are performed for an end-to-end process flow. Hence, it is important to keep track of all the API calls happening in between so that a single transaction can be tracked end-to-end. Each API request should have a unique identifier that is tied up with all log statements related to the request.

1. **Contextual Log statements:**

The Logs should be easily readable and should define the context of the operation.

For example: The log for an API used to fetch user details can be -

server01.dev.01 | 172.25.187.64 | ENV\_ID =DEV | EVENT\_TIME= 2022-12-16 16:40:34.118 | LOG\_LEVEL= INFO | CORRELATION\_ID= 4352-h4gf-ssf4-hkjs | USE\_CASE = CreateOrder | SERVICE\_NAME= GetUserDetails | EVENT\_STATUS\_CODE=200 | EVENT\_STATUS= SUCCESS | EVENT\_MESSAGE=UserDetails retrieved successfully

1. **Logging of the response status code:**

The response status code (e.g., HTTP status code such as 200) should be always logged.

1. **Define and use proper type of Log:**

There are different types of logs available such as INFO, DEBUG and ERROR.

The categories of log and when to use which type of log should be clearly defined and followed throughout the application.

1. **Avoid logging sensitive or PII information:**

When logging payload of the API such as request or response, ensure that sensitive or PII information is not logged in the plain text.

## 6.2 API Logging Standards Checklist

**API Log Structure:**

Below checklist can be used to implement the API logging structure:

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Item** | **Implemented: Yes or No?** |
| 1 | Each log should contain a unique id related to the end-to-end transaction such as correlation-id. |  |
| 2 | Each log should contain a timestamp. |  |
| 3 | Each log should contain environment name, server IP and server name. |  |
| 4 | Each log should contain an API name for which the log is written. |  |
| 5 | Each log should contain a logging message which provides contextual text in below format and is compatible with SIEM solution:  <<SERVER\_NAME>> | <<SERVER\_IP>> | ENV\_ID =<<>> | EVENT\_TIME= <<Date & Time>>| LOG\_LEVEL= <<INFO/DEBUG>>|CORRELATION\_ID=<<>>| USE\_CASE = <<>> | SERVICE\_NAME= <<>>| EVENT\_STATUS\_CODE=<<>>| EVENT\_STATUS= <<INITIATED/IN\_PROGRESS/SUCCESS/FAILURE>>| EVENT\_MESSAGE=<<>> |  |
| 7 | All the sensitive and PII information should be masked or encrypted and should not be disclosed in any way. |  |
| 8 | In case of exception, each log should provide the error details in below format:  <<SERVER\_NAME>> | <<SERVER\_IP>> | ENV\_ID =<<>> | EVENT\_TIME=<<Date & Time>>| LOG\_LEVEL= <<ERROR>>|CORRELATION\_ID=<<>>| USE\_CASE = <<>> | SERVICE\_NAME= <<>> | EVENT\_STATUS\_CODE=<<>>|EVENT\_ERROR\_CODE= <<>> | EVENT\_STATUS= << FAILURE>>| EVENT\_MESSAGE=<<>> |  |

**Example Log Structure:**

**Success Log:**

server01.dev.01 | 172.25.187.64 | ENV\_ID =DEV | EVENT\_TIME= 2022-12-16 16:40:34.118 | LOG\_LEVEL= INFO | CORRELATION\_ID= 4352-h4gf-ssf4-hkjs | USE\_CASE = CreateOrder | SERVICE\_NAME= GetUserDetails | EVENT\_STATUS\_CODE=200 | EVENT\_STATUS= SUCCESS | EVENT\_MESSAGE=UserDetails retrieved successfully

**Error Log:**

server01.dev.01 | 172.25.187.64 | ENV\_ID =DEV | EVENT\_TIME= 2022-12-16 16:40:34.118 | LOG\_LEVEL= INFO | CORRELATION\_ID= 4352-h4gf-ssf4-hkjs | USE\_CASE = SignUp | SERVICE\_NAME= CreateUser | EVENT\_STATUS\_CODE=400 | EVENT\_ERROR\_CODE = 4001 | EVENT\_STATUS= FAILURE | EVENT\_MESSAGE= Name cannot be blank

***Note:*** *Kindly refer Annexure – Log message format in this chapter for more details about the fields given in above example.*

**API Logging Implementation:**

Below checklist can be used to implement the API logging in application:

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Item** | **Implemented: Yes or No?** |
| 1 | Each log should be from one of the below types:   * **INFO:** This log is generally used at the start and end of processing within API. * **DEBUG:** This log is used in between the processing and should be logged before and after critical business logic implementation. This helps to debug the issues in live application quickly. DEBUG logs are generally disabled in the production systems unless a specific need arises. * **ERROR:** This log type should be used to log all the errors within API. |  |
| 2 | INFO logs should be present at start and end of API processing. |  |
| 3 | DEBUG logs should be present during the processing and specially before and after implementing critical business logic. |  |
| 4 | For all errors, ERROR logs should be logged. |  |

## 

## 6.3 Overview of API Error Handling Strategy

**Error handling strategy** helps in handling software error gracefully and helps execution to resume when interrupted. When the application is using APIs, an overall error handling strategy specific to APIs should be in place.

Diagram

Description automatically generated

Figure 6.1: API Error Handling Strategy

API errors can be categorized into below two categories:

1. **System errors:** The errors at infrastructure level before reaching the application such as server is not reachable, network issues etc.

***Note:*** *System errors are not in scope of this section.*

1. **Application errors:** The errors occurred after the request has reached the application.

The application errors can be further categorized as below:

* **Business errors:** Errors due to issue at the client side such as incorrect data passed in the request, incorrect authentication etc. 4XX http status error codes are used for business errors.
* **Technical errors:** Errors due to issue at the server side. Technical errors can occur in the application if any call to external system returns a server related error. 5XX http status error codes are used for technical errors.

## 6.4 API Error Handling Standards

This section defines the API error handling standards for application errors.

API Error handling standards are defined using below components:

1. API error response structure
2. Internal error codes
3. Error handling strategy based on different categories of errors

**API Error response structure:**

API error response must pass below **three basic criteria** to be helpful.

* **An HTTP Status Code,** so that the source and scope of the problem can be recognized.
* **An Internal error code** which provides more information about the error.
* **Human readable messages** that summarize the context, cause, and general solution for the error.

The API error response need to follow the below format:

|  |  |
| --- | --- |
| **Field** | **Description** |
| correlationId | Unique id for end-to-end tracing of the request. |
| timestamp | Date and time of the request/response. |
| errorType | errorType should be BUSINESS or TECHNICAL. |
| responseStatusCode | The HTTP status code of response. |
| errorCode | Internal error code derived from the HTTP status code which provides additional information about error. |
| errorDescription | Brief description of the error. |

**Example:**

|  |
| --- |
| {  "correlationId": “0b811819-9044-4856-b0ee-8c88035f8858”,  "timestamp": "2022-12-01T11:33:34.509Z",  "errorDetails": {  "errorType": "BUSINESS",  "responseStatusCode": "400"  "errorCode": "1011",  "errorDescription": "Field should not contain special characters."  }  } |

**Internal error codes:**

Internal error codes are useful as they give more information to the client instead of only HTTP status code such as 400 – bad request. They can be defined based on the functionality.

Internal error codes which should be used for user management module are defined below.

The internal error code range should be defined for each module as follows:

|  |  |  |
| --- | --- | --- |
| **Functionality** | **HTTP Status code** | **Internal Error Code Range** |
| User Management | 4XX,5XX | 1000-1999 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Functionality** | **HTTP Status Code** | **Internal Error Code** | **Internal Error Code Description** |
| User Management | 401 (Unauthorized) | 1000 | You have entered wrong credentials. Please enter correct credentials. |
| 401 (Unauthorized) | 1001 | Entered Captcha is incorrect. |
| 401 (Unauthorized) | 1002 | Entered OTP is incorrect. |
| 400 (Bad request) | 1011 | Field should not contain special characters. |
| 400 (Bad request) | 1012 | Field length is exceeded. |
| 400 (Bad request) | 1013 | date format is not correct. Enter the date in below format: <<>> |
| 404 (not found) | 1021 | User not found. |
| 403 (Forbidden) | 1031 | User does not have appropriate privileges. |
| 403 (Forbidden) | 1032 | Sign-in attempt has reached. Kindly try after some time. |
| 500 (Internal Server error) | 1041 | Internal server error has occurred. Kindly try after some time. |
| 502 (Bad gateway) | 1042 | Bad gateway. Kindly try after some time. |
| 503 (Service unavailable) | 1043 | Service is unavailable. Kindly try after some time. |
| 504 (Gateway timeout) | 1044 | Gateway timeout has occurred. Kindly try after some time. |

Based on the above, internal error codes for other modules of the project should be defined.

**API Error Handling strategy:**

Once the API error response structure and internal error codes are defined, API error handling strategy helps to define how the API error response structure should be implemented within API and how the system should respond to the different API errors.

The API error handling strategy should handle the common http status error codes as below:

* **4XX codes related to client-side error**
  + 4XX error codes happen due to incorrect data passed from client side such as missing authentication data or malformed request.
  + Requests with 4XX error response codes cannot be retried as it will return the same error every time.
  + However, it is important to provide proper internal error code along with error details in the response so that client can easily figure out and fix the issue.
* **5XX codes related to server-side error**
  + 5XX error codes happen due to error at server side such as memory issues, application unavailable, server outages etc.
  + Requests with 5XX error response codes can be retried as the same request can get processed once the server is up and running.
  + Hence, it is important to provide client with a retry mechanism details so that client can setup it up in case of server errors.
  + Also, at server side a circuit breaker[[17]](#footnote-17) functionality should be in place to avoid unnecessary hits to the server when server is down.

Diagram, schematic

Description automatically generated

*Figure 6.2: Retry Mechanism for error handling*

## 6.5 API Error Handling Standards Checklist

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Item** | **Implemented: Yes or No?** |
| 1 | The API error response should contain below fields:   1. correlationId 2. timestamp 3. errorType (BUSINESS or TECHNICAL) 4. responseStatusCode 5. errorCode 6. errorDescription |  |
| 2 | Internal error codes should be defined for each project. |  |
| 3 | Internal error codes defined for user management should be used in each project. |  |
| 4 | 4XX http status codes should be used for client-side errors. |  |
| 5 | 5xx http status codes should be used for server-side errors. |  |
| 6 | Retry mechanism to handle 5XX errors should be present. |  |
| 7 | Circuit breaker functionality should be present to handle 5XX errors. |  |

## Annexure – Log message format

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Field Name** | **Description** |
| 1 | SERVER\_NAME | Name of the server. |
| 2 | SERVER\_IP | IP address of the server. |
| 3 | ENV\_ID | Environment value such as DEV, SIT, UAT etc. |
| 4 | EVENT\_TIME | Date and Time at which the log is written. |
| 5 | LOG\_LEVEL | Log level can be INFO, DEBUG or ERROR. |
| 6 | CORRELATION\_ID | Unique Id for each request which can track the request end to end. |
| 8 | USE\_CASE | Can have the functionality name such as LoginUser. |
| 10 | SERVICE\_NAME | The specific API/service name such as GetUserData. |
| 11 | EVENT\_STATUS\_CODE | The HTTP status code in case of logging after an API call. |
| 12 | EVENT\_ERROR\_CODE | The HTTP status error code in case of logging after an API call. |
| 13 | EVENT\_STATUS | Event status can be INITIATED/IN\_PROGRESS/SUCCESS/FAILURE. The Event status will be INITIATED at the start of the process and can be SUCCESS/FAILURE at the end of the process. |
| 14 | EVENT\_MESSAGE | Human readable contextual message. |

## Annexure – HTTP Status Codes

Commonly used HTTP Status codes are given below:

|  |  |  |
| --- | --- | --- |
| **Status Code Type** | **Status Code** | **Description** |
| 2XX – Success | 200 | OK |
| 201 | Created |
| 202 | Accepted |
| 204 | No Content |
| 4XX- Client Error | 400 | Bad Request |
| 401 | Unauthorized |
| 403 | Forbidden |
| 404 | Not Found |
| 405 | Method not allowed |
| 415 | Unsupported media type |
| 429 | Too many requests |
| 5XX – Server Error | 500 | Internal Server Error |
| 502 | Bad Gateway |
| 503 | Service Unavailable |
| 504 | Gateway Timeout |

# 7. API Security Standards and Best Practices

## 7.1 What is API Security?

API security is the process of protecting APIs from attacks. Because APIs are very commonly used, and because they enable access to sensitive software functions and data, they are becoming a primary target for attackers. API Security focuses on strategies and solutions to understand and mitigate the unique vulnerabilities and security risks of API.

By implementing API security measures, organizations can:

* **Prevent unauthorized access**: Restrict API access to only authorized users and prevent unauthorized access to sensitive data and functionality.
* **Protect data confidentiality and integrity**: Ensure that data transmitted by APIs remains confidential and is not tampered with.
* **Detect and respond to security threats**: Monitor API usage for suspicious activity and take appropriate measures to mitigate security risks.
* **Adopt industry best practices**: Adopt industry best practices related to security, such as OWASP API Security Top 10

## 7.2 API Security Framework

This section describes an illustrative API security framework that can be tailored as per the organization and specific application’s needs.

As per the framework, API security can be applied at three levels as below:

A diagram of a security system

Description automatically generated

Figure 7.1: API Security Overview

1. **Network Layer**: The Network Layer is responsible for managing data transmission across networks, ensuring secure and reliable communication between API producers and consumers.
2. **Gateway Layer**: The Gateway Layer acts as a centralized security and management point for APIs, providing essential features like authentication, authorization, rate limiting, and throttling to protect APIs.
3. **Application Layer**: The Application Layer is responsible for implementing the business logic of APIs, handling data validation, and protecting sensitive information.

Following diagram depicts the components at each layer of API security framework:

A diagram of a software development process

Description automatically generated

Figure 7.2: API Security Framework

The APIs at a broader level, are consumed by 3 different types of applications, which are:

* **External Systems**: APIs are consumed by external systems, such as other applications, services, or platforms, to integrate with the API provider's system and exchange data. For example, a partner entity of bank may consume the APIs exposed by bank to build innovative use cases.
* **Web Portal**: APIs can be used for web portals, providing dynamic content, user interactions, and access to data from backend systems.
* **Mobile App**: APIs can be leveraged by mobile apps to access data and perform actions enabling seamless integration with the backend services.

**7.2.1 Network Layer**

The Network Layer is responsible for managing data transmission across networks. It ensures that data packets are routed correctly from the source to the destination and handles error detection and correction.

The Network Layer plays a crucial role in API security by:

* **Routing and Addressing**: It ensures that API requests and responses are routed correctly to the intended destination using IP addresses and routing protocols.
* **Packet Switching**: It breaks down data into packets and reassembles them at the destination, enabling efficient transmission over networks.
* **Error Detection and Correction**: It detects and corrects errors that may occur during transmission, ensuring data integrity and reliability.
* **Traffic Management**: It helps manage network traffic to prevent congestion and ensure efficient delivery of API requests and responses.

The network layer can be secured by implementing some approaches from the following list based on application’s needs:

**7.2.1.1 Secure Communication:**

* **SSL/TLS (One-Way):**

SSL/TLS is a security protocol used to encrypt data transmitted between a client and a server. In this configuration, the server's identity is verified by the client, but the client's identity is not verified by the server.

* **mTLS (Two-Way):**

Mutual TLS (MTLS) is a security protocol that establishes mutual authentication between a client and a server. This means that both parties verify the identity of the other before communication takes place. This helps in establishing mutual authentication between the client and server which verifies the identity of both parties.

* **IPSec:**

IPsec is a group of protocols that are used together to set up encrypted connections between systems. It helps keep data sent over public networks secure. IPsec is often used to set up VPNs, and it works by encrypting IP packets, along with authenticating the source where the packets come from.

**7.2.1.2 Restrict Access:**

* **IP Whitelisting:**

IP whitelisting is a security measure that restricts access to an API or system to a specific list of approved IP addresses. This helps to prevent unauthorized access from unknown or malicious sources.

* **Geo Fencing:**

Geo-fencing is a methodology that restricts access to an API or system based on the user's geographic location. This method is useful for preventing access from certain regions.

**7.2.2 Gateway Layer**

Gateway Layer provides essential security and management functions for APIs.

The Gateway Layer plays a crucial role in API security by:

* **Centralized Control**: The Gateway Layer acts as a centralized point of control for enforcing policies and managing API access. This simplifies security management and reduces the risk of vulnerabilities at the individual API level.
* **API Traffic Management**: Gateway Layer handles API traffic, managing requests, responses, and load balancing to ensure optimal performance and availability. This helps protect APIs from overload and ensures that they can handle high traffic volumes.
* **API Protection**: The Gateway Layer provides various security features to protect APIs from common threats.

The gateway layer can be secured by implementing some approaches from the following list based on application’s need:

**7.2.2.1 Authentication and Role based access:**

* **Authentication:** Verifying the identity of users or clients accessing the API ensures that only authorized users can access the API.
* **Authorization**: Granting specific permissions to users or clients based on their roles or privileges controls which actions users can perform within the API, preventing unauthorized access to sensitive resources.
* **API Key Management**: Managing API keys to control access and prevent unauthorized usage.

**7.2.2.2 Policy Enforcement:**

Provides a centralized mechanism for managing API access and enforcing security policies. Some of the policies which can be applied are:

* **Rate Limiting:** Limiting the rate at which API requests can be processed prevents abuse and ensures fair resource allocation, protecting APIs from Distributed denial of service (DDoS) attacks and preventing overloading of backend systems.
* **Throttling**: Limiting the overall rate of API usage by individual users or applications prevents abuse and ensures fair resource allocation among different users and applications.
* **IP Whitelisting/IP Blacklisting**: Allow access of API only to specific IP range or block traffic from specific IPs.

**7.2.2.3 Circuit Breaker:**

Circuit breakers are a mechanism used to protect APIs and services from cascading failures caused by backend system failures or overload. They work by automatically stopping connections to a backend service if it fails to respond within a specified time or if the number of consecutive failures exceeds a threshold.

**7.2.3 Application Layer**

The Application Layer handles the core business logic of APIs, processing requests, performing calculations, and returning responses.

The application layer can be secured by implementing some approaches from the following list based on application’s need:

**7.2.3.1 Authentication and Role Based Access:**

Authentication helps in verifying the user identity. Role Based Access helps in granting appropriate permissions based on roles to prevent unauthorized access and data breaches.

The Authentication and Role Based Access can be achieved by:

* **Authentication and Authorization:** Ensuring that only authorized users can access specific API endpoints and perform designated actions.
* **API Access Mapping:** Controlling which users or groups can access specific APIs and perform different actions.

**7.2.3.2 Data Protection:**

Data Protection can be achieved by the following measures:

* **Payload Encryption & Decryption:** Encrypting and decrypting the payload itself adds an additional layer of security, especially for highly sensitive data.

The Payload Encryption & Decryption can be achieved by:

* **Generate an AES Session Key:** Use the Advanced Encryption Standard (AES) to create a session key for encrypting the payload.
* **Encrypt Sensitive Data:** Use the AES key to encrypt the data.
* **Encrypt the AES Key:** Encrypt the AES session key with the recipient’s RSA public key. This ensures that only the intended recipient, who possesses the corresponding private key, can decrypt the AES key and subsequently the payload.
* Input Data Validation: Validating input data to prevent injection attacks and to ensure data integrity.
* Geo Fencing: Geo-fencing is a methodology that restricts access to an API or system based on the user's geographic location. This method is useful for preventing access from certain regions.

**7.2.3.3 Application Protection:**

Application Protection can be achieved by the following measures:

* **Runtime Application Self-Protection (RASP):** Detecting and preventing attacks at runtime using Runtime Application Self-Protection, providing a proactive security layer for API applications.
* **Device Binding:** Restricting API access to specific devices or platforms ensures that only authorized devices can interact with the API.

Throughout the API security framework, there is a need for API audit logging which acts as an essential part of API management. It helps to monitor and record all the activities which ensures non-repudiation.

* **Audit Logging**

Following details should be captured in audit logging.

* Who performed the action and what client did they use?
* When was the request received?
* What kind of request was it, such as a read or modify operation?
* What resource was being accessed?
* Was the request successful? If not, why?
* What other requests did they make around the same time?

Audit logs should be pushed to the Security information and event management (SIEM) tool. SIEM will help in security incident analysis. Also, sensitive data like password, credit card number and personally identifiable information (PII) like account no, email, Aadhar card number, PAN etc. should be masked while logging.

# 8. API Governance Process and Versioning Strategy

## 8.1 Background and Purpose

**8.1.1 What is API Governance?**

API Governance ensures that a standard process is applied to the onboarding of API consumers and release of new changes in the APIs so that the APIs are managed effectively.

API Governance process can be divided into two steps:

1. **API consumer onboarding process:**

Process to onboard API consumers which will be using the APIs.

The process can include

* defining eligibility criteria,
* performing evaluations of API consumer and
* periodic reviews to ensure eligibility of API consumer.

1. **API versioning and release strategy:**

Process to

* receive requests for new changes in the APIs,
* perform impact analysis of the proposed change,
* decide on version of the API and release new version of the API.

**8.1.2 Purpose of API Governance**

* Uniform onboarding process for API consumers.
* Ensure alignment across stakeholders with respect to API specification.
* Improve the quality and consistency of the APIs.
* Provide API standards and guidelines to stakeholders.
* Reduce errors by assuring that the common guidelines are understood by and accessible to stakeholders.
* Standardize data dictionaries and versioning so that the stakeholders aren’t reinventing the wheel with each new or updated API.
* Standardize endpoint and parameter naming which helps create an ecosystem that makes it easier to integrate new APIs into larger, more complex flows.

## 8.2 API Governance Process

The API Governance process can be setup as per the project requirement. There can be two types of APIs developed within project:

1. **Development of private APIs which are consumed by a single front end application**

In this case, the API governance and versioning process is not of much relevance as there is a single source which is consuming the API and that is also within the same project.

1. **Development of public or partner APIs which are exposed to outside world and consumed by various entities**

The API governance process is crucial in such projects as change in the APIs will impact the stakeholder business.

**Depending on the type of APIs, a governance process can be established which covers below points:**

1. **API Consumer onboarding process**

* Define eligibility criteria for the API consumer
* Receive application from the proposed API consumer
* Perform technical evaluation of the API consumer to ensure that
  + API consumer has the required infrastructure to process and store the data received from APIs
  + API consumer has the information security controls in place
  + API consumer is handling the data as agreed with the organization who has developed the APIs
* Grant access to API consumer if eligibility criteria is met and technical evaluation is passed.
* Periodic review of API consumer to ensure its eligibility to access the APIs

1. **API versioning and release strategy**

* A new change is proposed in the API
* Impact analysis of the proposed change
* Decide version of API for the proposed change as per versioning strategy
* Decide deprecation and decommission timelines for previous versions of the API
* Approval on the new version (from stakeholder/project owners)
* Publish the new version of API and publish deprecation and decommission dates for previous versions
* Communication to stakeholders for API release

## 8.3 API Versioning Strategy

API versioning strategy is crucial part of the API governance.

It is recommended to:

* Use semantic[[18]](#footnote-18) format for API versioning
* Keep the major version in the URL

**For Example:**

URL: POST /v1/employee

Request Body:

|  |
| --- |
| {  "ver": "1.1.2",  "txnid": "0b811819-9044-4856-b0ee-8c88035f8858'",  "timestamp": "2017-07-13T11:33:34.509Z",  } |

API version is to be decided on basis of below change categories:

1. **Breaking change**: API change which will break the existing functionality and new implementation must be required to make it compatible with the latest API version. This will also include change to existing legacy system, change to DB schema due to API Spec change.
2. **Backward compatible change:** API change will not break the existing functionality. Users who wish to avail new features can do so however the existing implementation will work with latest API version.
3. **No Impact:** API change which is an enhancement in API specification/documentation such as updating the description of fields, restructuring within API spec without any impact on implementation.

The possible changes to the APIs are listed in below table along with the version to be considered:

|  |  |  |
| --- | --- | --- |
| **Change in API** | **Category** | **API Version** |
| HTTP Method change | Breaking change | Major version |
| URI Change | Breaking change | Major version |
| Addition of new mandatory field | Breaking change | Major version |
| Changing existing field name | Breaking change | Major version |
| Deletion of enumeration values | Breaking change | Major version |
| Change in data type of field or request/response body | Breaking change | Major version |
| Addition of new required headers | Breaking change | Major version |
| Addition of new error code | Breaking change | Major version |
| Addition of enumeration values | Breaking change | Major version |
| Addition of new optional field | Backward compatible | Minor version |
| Updating documentation/description of fields | No Impact | Patch |
| Addition of new optional headers | Backward compatible | Minor version |

## 8.4 API Deprecation and Decommission Strategy

When the new version of an API is released, it is necessary to stop the support for earlier API versions and eventually retire the earlier APIs. Below are the statuses which the API goes through during its life cycle:

|  |  |
| --- | --- |
| **Status** | **Description** |
| Active | API is currently active. |
| Deprecated | API is deprecated and next version is available. However, the API will still be available till decommissioned. |
| Decommissioned | API version is officially ended, and it should not be used. |
| Cancelled[[19]](#footnote-19) | In case the current version is amended, and the changes are no longer applicable. |

The deprecation and decommission of an API can be done at the individual API level or at API catalog[[20]](#footnote-20) level.

**The deprecation and decommission strategy should be applied only to the major versions of the API.** Minor and patch versions should not be deprecated individually. Hence minor and patch version can be called as optional releases and major version is a mandatory release.

**For example:**

Publish version timeline:

Mar 2022 – v1.1.3

June 2022 – v 1.2.0

Sept 2022 – v 1.3.0

Dec 2022 - v 2.0.0

After publishing v 2.0.0, all the previous 1.x.x versions will be deprecated and decommissioned as per the timeline agreed. However, there will not be any separate timeline for deprecation and decommission of v 1.2.0 when v 1.3.0 will be published.

There are two approaches to communicate the deprecation of an API:

**Non-technical approach:**

Before deprecating the APIs, all the stakeholders to be informed well in advance through various communication channels such as mail or a meeting.

**Technical approach:**

Add deprecation[[21]](#footnote-21) and decommission[[22]](#footnote-22) headers to the API response. These headers can return the respective deprecation and decommission date of that API. Also, another header can provide deprecation strategy or alternate resource for the deprecated API.

Below optional headers are to be added to the changed APIs after deprecation of the API.

|  |  |  |
| --- | --- | --- |
| **Header** | **Description** | **Example** |
| x-api-deprecation-date | Specifies the deprecation date of an API. Deprecated API will be fully functional till the decommission date. This header indicates that consumers should start planning to move to next version before the decommission date. | x-api-deprecation-date: Sun, 6 Jun 2021 23:59:59 GMT |
| x-api-deprecation-info | Provide link to latest version of an API. | x-api-deprecation-info: https://api.example.com/v2  /accounts; rel="latest-version"; |
| x-api-decommission-date | Specifies the decommission date of an API. The API will become unresponsive at this date. All the consumers of API will have to switch to new version before decommission date. | x-api-decommission-date: Sun, 12 Dec 2022 23:59:59 GMT |
| x-api-decommission-info | Provide link to latest version of an API. | x-api-decommission-info: https://api.example.com/v2  /accounts; rel="latest-version"; |

Both the above approaches should be used effectively with each other to have clear communication with stakeholders and to ensure smooth transition while upgrading the API version.

8.5 Version an API - Timeline

A standard timeline can be set to deprecate and decommission an API as per the project requirement.

The deprecation and decommission will be applicable only when a major version is released.

The below parameters should be considered for the API release timeline:

1. Communication to all stakeholders about new release version and

deprecation of current one

1. New version release
2. Deprecation of previous version
3. Decommission of previous version

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Communication to stakeholders prior to release (T0)** | **Version release (T1)** | **Deprecation of previous version (T2)** | **Decommission of previous version (T3)** |
| Major | T0 | T0 + 1 Month | T1 + 3 Months | T2 + 6 Months |

***Note:*** *The timeline given in above table is indicative and subject to change as per specific project requirements.*

## 8.6 Amendment of existing version

Once the API version is published, there may be a need to rollback/modify certain changes published in latest release due to unforeseen challenges.

In such a case, instead of publishing a new version, amendment to the existing version is recommended.

**Minor/Patch version amendment:**

The changes in the minor/patch versions are backward compatible. Hence there will be no implementation impact due to minor/patch version release. However, if due to some issue, the version needs to be amended then following process can be followed:

1. The API will be modified to roll back the changes.
2. The change log of the version will be updated to document the amendment.

**For example:**

Version 1.1.0 is released for an API with addition of optional field and some field description updates.

The version needs to be amended to update incorrect description of the fields.

In this case, the API with version 1.1.0 will be modified to update description of the fields.

The change log of API with version 1.1.0 will be updated to document the amendment.

**Major version amendment:**

***Note:*** *Major version amendment will be a rare scenario as the required impact analysis will be done by the organization before publishing the change. However, in case the major version amendment is proposed, there needs to be a process established to take necessary approvals from the stakeholders of the project.*

There can be three possible scenarios for amendment of major version.

1. **Multiple breaking changes are published**

The major version release contains more than one breaking change such as addition of two or more mandatory fields with version 2.0.0.

The amendment process will be as follows:

* 1. The API will be modified to remove one mandatory field.
  2. Other published changes will remain as is in the published version.
  3. The change log of the version 2.0.0 will be updated to document the amendment.

1. **Single breaking change is published**

The major version release contains a single breaking change such as addition of one mandatory field with version 2.0.0.

The amendment process will be as follows:

1. The API will be modified to remove the mandatory field.
2. The status of API version 2.0.0 will be marked as Cancelled.
3. The status of all previous 1.x.x versions will be marked as Active again.
4. The deprecation and decommission date of all previous 1.x.x versions will be removed.
5. The change log of the API version 2.0.0 as well as all the versions of 1.x.x will be updated.
6. **Both breaking changes and backward compatible changes are published**

The major version release contains multiple changes such as addition of one mandatory field and one optional field with version 2.0.0.

The amendment process will be as follows:

1. The API will be modified to remove the mandatory field.
2. The status of API version 2.0.0 will be marked as Cancelled.
3. A new minor version of 1.x.x may be published for addition of optional field.
4. The status of all previous 1.x.x versions will be marked as Active again.

## 8.7 Adoption of new API version – scenarios

This section covers the various scenarios on how a major/minor/patch version can be adopted by the stakeholders. There are two types of entities involved – API provider and API consumer.

**API Producer:** Entity which implements the API and exposes it for others to use.

**API Consumer:** Entity which consumes the API exposed by API Provider.

**When a project is developing the APIs, it will be considered as API Producer.**

**Scenario 1: API provider has published latest minor/patch version and consumer is on old minor/patch version**

|  |  |
| --- | --- |
| **API Consumer** | 1.1.2 |
| **API Provider** | 1.1.3 |

In this case, as the API Provider’s version is backward compatible, the existing consumer request with v 1.1.2 should work seamlessly.

**Scenario 2: API provider has published latest major version and API consumer is on old minor/patch version**

|  |  |
| --- | --- |
| **API Consumer** | 1.2.0 |
| **API Provider** | 2.0.0 |

The API provider will support the earlier version till decommission date. However, the API consumer should implement the major version as soon as it is published so that it is using the major version before the earlier v1 version is decommissioned.

# 9. API Gateway Overview and API Policy Guidelines

## 9.1 What is API Gateway?

An API gateway is an API management tool that acts as a single point of entry for API calls, routing requests from the client to backend services, gathering the requested data, and delivering it to the client in a single, combined package. It also provides analytics, threat protection, and other security features for the application.

A diagram of application process

Description automatically generated

Figure 9.1: Overview of API Gateway

**Features of API gateway:**

* **API Security**: An API gateway acts as a single-entry point for the APIs and provides a centralized way to apply API security configurations. It employs authentication and authorization mechanisms to verify the identity of clients and grant them appropriate permissions. Rate limiting prevents abuse by setting limits on the number of requests a client can make within a given time. IP whitelisting/blacklisting can filter requests from IP addresses, while data encryption protects sensitive information. Additionally, security headers provide protection against common web vulnerabilities.
* **Performance Optimization:** API gateways can be used to achieve performance optimization using below features:
  + **Caching:** Reduces latency and improve response times.
  + **Load balancing:** Distribute traffic across multiple backend servers, preventing bottlenecks and ensuring high availability.
  + **Compression:** Reduces the size of responses, optimizing network bandwidth.
  + **Throttling:** Limits the number of concurrent requests, preventing system overload and maintaining consistent performance.
* **Management and Monitoring**: API gateway provides a single point for overseeing APIs, policies and configuration. API gateway can provide analytics and monitoring capabilities which can be used to track API usage and performance metrics. API gateway may also provide version control which facilitates management of different API versions, ensuring smooth transition and backward compatibility.
* **API Orchestration**: API orchestration is a powerful capability that extends beyond the core functions of an API gateway. It involves intelligently combining and coordinating multiple APIs to create new, composite services. API gateway acts as an intermediary invoking multiple APIs in the backend and providing a single, consolidated response back to the client.
* **Policy Enforcement**: An API gateway enforces policies that govern API usage, ensuring compliance with security, governance, and business requirements. These policies can include:
  + **Access Control:** Defining who can access specific APIs and what actions they are permitted to perform.
  + **Usage Quotas:** Setting limits on API usage to prevent abuse and manage costs.
  + **Data Validation:** Validating input data to ensure it meets predefined criteria and prevent errors.
  + **Error Handling:** Defining how errors should be handled and communicated to clients.
  + **Security Best Practices:** Enforcing security best practices like HTTPS, input validation, and encryption.

**Examples of API Gateway:**

1. **https://aws.amazon.com/api-gateway/**
2. **https://konghq.com/kong**

## 9.3 Service Mesh

Microservice architectures are becoming more and more popular in enterprise organizations as they allow for greater agility with smaller, more targeted services — compared to a monolithic architecture that’s difficult to develop and maintain.

As organizations build more microservices, complexity grows. The governance and security considerations behind microservice interactions are often custom coded into the service logic. Teams build in different languages and deploy to multiple environments, and an organization’s services are typically siloed with decentralized management.

The concept of a **service mesh** has been introduced to address the challenges that come with microservice implementations. A service mesh can abstract the governance considerations behind microservices that primarily interact with one another.

A service mesh is a software architectural pattern used for microservices deployments that uses a sidecar proxy to enable secure, fast, and reliable service-to-service communications.

**Key Components of Service Mesh:**

A diagram of a service

Description automatically generated

Figure 9.2: Key Components of Service Mesh

* **Control plane:** This centralized component manages the configuration and policies of the service mesh. It handles tasks like service discovery, traffic management, and security.
* **Data plane:** This distributed component is deployed as a sidecar proxy alongside each service instance. It handles the actual communication between services, including load balancing, fault tolerance, and security.

**Benefits of a Service Mesh:**

* **Improved reliability:** Service meshes improves the reliability of microservices applications by providing features like fault tolerance, retries, and circuit breaking.
* **Enhanced security:** Service meshes protects microservices applications from security threats by providing features like encryption, authentication, and authorization.
* **Simplified development:** Service meshes can simplify the development and management of microservices applications by abstracting away the complexities of service-to-service communication.
* **Better observability:** Service meshes can provide valuable insights into the behavior of microservices applications, making it easier to identify and troubleshoot problems.

**API Gateway vs Service Mesh:**

Service mesh and API gateways are both essential components of modern microservices architectures, but they serve distinct purposes. Service mesh focus on managing internal communication between services, providing features like service discovery, load balancing, and fault tolerance. API gateways, on the other hand, act as a single-entry point for external clients, offering functionalities such as authentication, authorization, and rate limiting. While they can sometimes be used together to create a more comprehensive solution, understanding their differences is crucial for making informed architectural decisions.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Service Mesh** | **API Gateway** |
| Primary function | Manages internal communication between services | Acts as a single-entry point for APIs |
| Key features | Service discovery, load balancing, fault tolerance, security, observability | API Security, performance optimization, API management and monitoring, Policy enforcement |
| Focus | Internal communication between services | External communication between clients and microservices |
| Deployment | Sidecar proxies alongside each service instance | Centralized component |
| Use cases | Complex microservices architectures with many interconnected services | Controlling access to microservices from external clients |

## 9.4 API Architecture

Organizations may have different types of application such as:

* legacy applications (monoliths) which may integrate using Service Oriented Architecture (SOA),
* modern API based (REST/JSON) applications, and
* newer microservices applications.

To support all these various scenarios and creating and publishing APIs for consumers below architecture can be considered as a reference model.

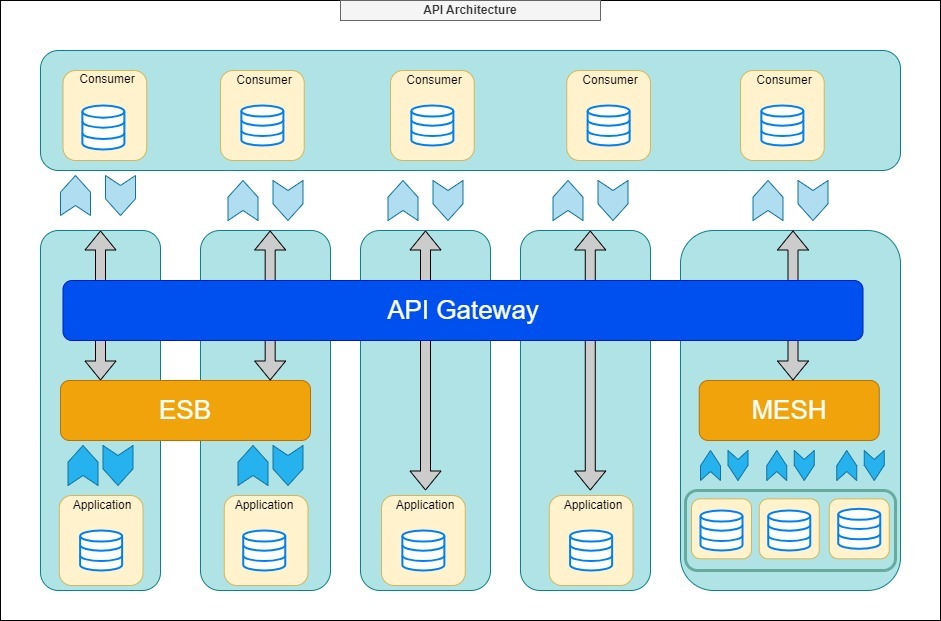


Figure 9.3: API Architecture

1. At the left are the legacy applications, which require additional decoupling via the ESB. The service of the ESB can be wrapped into the API gateway.
2. The modern applications are in the middle, which can be exposed directly via the API gateway. In this pattern the application uses modern communication, such as REST/JSON.
3. At the right are the microservices which communicate internally via the service mesh. APIs, which need to be exposed externally, are decoupled via the API gateway.

By placing the ESB, API gateway, and service mesh under the API Architecture umbrella, a single interface is created, from which all service communication can be controlled. Each domain is ring-fenced, speaks its own domain language, and uses the first-hand language for external communication.

## 9.5 API Policy

API Policies enable organizations to enforce regulations to help manage security, control traffic, and improve adaptability of the APIs. An API policy is like a module that implements a specific, limited management function.

For example, a policy can control authentication, access, allotted consumption, and service level access (SLA).

Policies provide features like security, rate-limiting, transformation etc.

**API Policy categories:**

* **Security:** Enforces Authentication (Choose between methods such as Basic, OAuth2.0, JWT etc), allow / deny access based on IP Range (IP Black / White- listing).
* **Compliance:** Enforce requirements such as Client ID and Cross Origin Resource Sharing (CORS).
* **Transformation:** Allows to transform / enrich headers for request or response. This could be used to add or remove any default or sensitive information from within the headers.
* **Quality of Service:** QoS based policies mainly provide SLA support, Spike control to queue traffic and Caching.
* **Troubleshooting:** For troubleshooting incoming requests by logging messages.

**API Policy usage guidelines:**

* **Number of Policies:** Efficiently apply policies. Increased number of policies could affect APIs performance as they are executed in the runtime.
* **Order of Policies:** Policies are executed in the same order as they are listed when sending request to API as well as executed in the reverse order when sending response. Applying policies like threat protection, caching earlier could help with performance.
* **Custom Policies:** Use custom policies to apply business needs that are not met using default policies.

# 10. API Developer Portal –Overview and Implementation Guidelines

## 10.1 Background and Purpose

**What is an API Developer portal?**

An API developer portal is an application that is used to publish APIs which can be made available to the stakeholders/developers for consumption. Depending on the project technology stack, an out of the box developer portal[[23]](#footnote-23)[[24]](#footnote-24) provided by API management solutions can be used or it can also be built from scratch as per the project requirement.

Below are the minimum components which an API developer portal should contain:

1. **API Documentation** – details about usage of each API, request/response structure, field descriptions, related documents etc.
2. **User on-boarding** – user management module where users can sign up, login, manage preferences and check the published APIs.
3. **Key management** – Manage API keys which are used to call the APIs: assign keys to users, revoke keys etc.
4. **Sandbox** - an isolated testing environment that enable developers to call APIs without affecting the real application, system, or platform on which they run.

**Benefits of API Developer Portal:**

When the APIs are published on the API developer portal, stakeholders can easily:

* Discover which APIs are available.
* Browse the API documentation.
* Register for—and immediately receive—their own API key that can be used to build applications.
* Try out the APIs in the developer portal UI.
* Monitor their own API usage.

## 10.2 API Developer Portal Overview

Though an API developer portal can include many additional features, a minimum viable developer portal (MVP) should contain below components:

1. API Documentation
2. User on-boarding
3. Key Management
4. Sandbox

**10.2.1 API Documentation**

API documentation and console allow developers to visualise and interact with the API's resources without having any of the implementation logic in place. The specification format is easy to learn and readable to both humans and machines.

**10.2.2 User On-boarding**

User registration and subsequent manual or automatic approval of new user accounts should be part of the API developer portal.

**10.2.3 Key Management**

An API key is a unique identifier used to connect to, or perform, an API call. To enable the consumers to call the APIs, generating and managing the API keys is a required component of the API Developer portal.

Also, API Developer portal should support for automatic or manual API key approval for users.

API key management covers multiple actions and features, including:

* Create new API keys
* Define the restrictions attached to a particular API key
* Assign the relevant credentials to the API key so the appropriate users and applications can access the correct data
* Rotate API key strings as needed
* Verify and maintain API key security
* Regenerate API keys to support increased security
* Add API key authentication to an existing API without requiring new code
* Update the applications that use a particular API key with a newly generated key

Administrators should be able to revoke keys.

**10.2.4 Sandbox**

A sandbox is an isolated testing environment that enable developers to call APIs without affecting the real application, system, or platform on which they run. The API sandbox makes it possible to: reduce the cost and risks associated with calling real APIs during testing.

**Benefits of API sandbox:**

1. **Perform continuous simulation**

Once the users are registered with the API sandbox, API simulation can be performed continuously, depending on the integration need of users.

1. **Run testing sessions**

The API sandbox also allows developers to perform System Integration Testing (SIT) and Unit Acceptance Testing (UAT).

1. **Reduce the risk of integration failure**

Simulation is crucial to find API incompatibility with the product development process. Therefore, the API sandbox can be used to discover loopholes in the integration process.

**10.2.5 Additional Features**

Once a minimum viable portal has been delivered, the project can also include below additional features:

**10.2.5.1 Landing Page**

A fully customised landing page that matches with corporate theme and makes a right impression to the developer community is key to get the right attention and start. Landing page can be customised by HTML 5, CSS, images, and Drupal modules.  Landing page optimisation also includes adding control menu appearance and content.

**10.2.5.2 SDKs**

SDK (Software Development Kit) is usually a client-side library in different languages that facilitates usage of an API.  This helps developer understanding and facilitate the usage of the APIs.

**10.2.5.3 FAQs**

A list of Frequently Asked Questions (FAQs) and other supporting documentation like Examples, Tutorials can be added to accelerate developer journey.

**10.2.5.4 Community (Threads, Blogs, Knowledge base, Feature request)**

Creating blog posts, role-based access of blogs, creating threaded developer forums, role-based moderation can be included in API developer portal as necessary.

**10.2.5.5 Web-hook Notification**

The Web-hooks API allow developers to subscribe to events happening at the provider of API developer portal. Developers should be able to subscribe to different types of web-hooks based on their interest and need, from the list of web-hooks offered by the provider.

**10.2.5.6 Pricing/Monetization**

API Monetization is a powerful solution that helps enterprises leverage their digital assets into new revenue streams and business models. API Monetization can also help partners, companies and developers manage their quota, limits, and subscription types.

**10.2.5.7 API Monitoring**

Monitoring and business reports help developers to control their usage of APIs in real-time and through historical analytics. It is important part of dashboard in developer portal.

**Example Developer Tools:**

These are some of the Developer Tools:

Visa Developer Portal

Spotify for Developers Portal

Plaid Docs

Wells Fargo Developer Gateway

Xero Developer

Twilio API Reference

## 10.3 API Developer Portal Details

Some of the important API developer portal components are explained below in detail:

**10.3.1 API Documentation**

The below sections are fundamental for a good API documentation.

* 1. **Authentication**

This is the information about using authentication schemes to start consuming the API. Most APIs have authentication schemes, and consumers must authenticate before gaining access to the API. This section should be properly documented to ensure that users are able to successfully authenticate against the API.

* 1. **Error messages**

Error messages are important because they tell users when they're integrating with the API services in an incorrect way. The error standards should be explained, and details should be provided on how to overcome them when a user gets an error.

* 1. **Resources**

Resources are the core components of the API which users interact with constantly. All the resources of the API should be documented correctly with request and responses.

* 1. **Terms of use**

This is the legal agreement between the consumer and the organization, defining how the consumer should ideally use the services. API limits, constraints related to API usage and practices should be clearly stated in the terms and conditions.

* 1. **Change log**

Detail updates and versions of the APIs and how that might affect API consumer should be clearly stated. This will help consumers know the stability of the API and see if any changes need to be made for an effective API call.

* 1. **Avoid Jargon**

Many people working with the API may not have intimate knowledge of the domain or jargon used while describing the APIs. Documentation should cater to the “very technical” developer audience, and the less technical decision makers (like Product Managers).

* 1. **Requests and Responses Description**

The request and responses should be provided in detail with context around each parameter.

* 1. **Add API Documentation Resources**

Documentation can be supplemented with additional resources like:

* **Getting Started Guide**

The Getting Started guide provides a detailed account of how to quickly start working with the published APIs. The emphasis in the guide should be on ensuring consumers reach success with APIs as quickly as possible, hand holding them throughout this journey.

* **SDKs and Libraries**

Code libraries help developers quickly call different resources. Having quick and easy methods in different languages to work with API helps developers feel more comfortable working with the API. SDKs can help in API adoption.

* **Interactive Console**

Interactive console helps developers to experiment with the API and know the value of the API visually.

**10.3.2 User On-boarding**

Users using API Developer Portal can be:

A picture containing diagram

Description automatically generated

Figure 10.1: User Types in API Developer Portal

**Admin:**

Users with this role will have full access to the API developer portal including create users, roles, granting permissions, assigning API keys, revoking API keys, manage API lifecycle, API monitoring dashboards etc.

**Publisher:**

Users with this role can create and deploy APIs within the developer portal. Users with this role can view API monitoring dashboards to understand how the APIs that their organization owns is consumed by other organizations.

**Consumer:**

Users with this role develop applications that consume APIs. Depending on the user's permission settings, users with this role can select APIs to be consumed by their application and can choose an API plan for that API's consumption.

**Authentication Methods:**

To enable users to login into the API developer portal, below authentication methods are recommended:

* 1. **Password-based authentication**

Password-based authentication relies on a username and password or PIN. This is the most common authentication method. However, there should be strong password policy set with adequate complexity and which require users to change the password regularly.

* 1. **Two-factor/multifactor authentication**

Two-factor authentication (2FA) requires users provide at least one additional authentication factor beyond a password. MFA requires two or more factors. Additional factors can be a one-time password sent to the user via text or email. Factors can include out-of-band authentication, which involves the second factor being on a different channel from the original device to mitigate man-in-the-middle attacks. This authentication type strengthens the security of accounts because attackers need more than just credentials for access.

**10.3.3 Key Management**

An API key is a unique identifier used to connect to, or perform, an API call. To enable the consumers to call the APIs, generating and managing the API keys is a required component of the API Developer portal.

Also, API Developer portal should support for automatic or manual API key approval for users.

API key management covers multiple actions and features, including:

* Create new API keys
* Define the restrictions attached to a particular API key
* Assign the relevant credentials to the API key so the appropriate users and applications can access the correct data
* Rotate API key strings as needed
* Verify and maintain API key security
* Regenerate API keys to support increased security
* Add API key authentication to an existing API without requiring new code
* Update the applications that use a particular API key with a newly generated key

Administrators should be able to revoke keys.

API Key can be implemented by below methods:

1. **Bearer Authentication**

Bearer authentication (also called token authentication) is an HTTP authentication scheme that involves security tokens called bearer tokens.

The name “Bearer authentication” can be understood as “give access to the bearer of this token.” The bearer token allowing access to a certain resource or URL and most likely is a cryptic string, usually generated by the server in response to a login request.

The client must send this token in the Authorization header when making requests to protected resources:  
Authorization: Bearer <token>

Bearer authentication should only be used over HTTPS (SSL).

1. **OAuth (2.0)**

The most common implementations of OAuth use one or both tokens instead:

* **access token** sent like an API key, it allows the application to access a user’s data; optionally, access tokens can expire.
* **refresh token**: optionally part of an OAuth flow, refresh tokens retrieve a new access token if they have expired. OAuth2 combines Authentication and Authorization to allow more sophisticated scope and validity control.

OAuth 2.0 is the best choice for identifying personal user accounts and granting proper permissions. In this method, the user logs into a system. That system will then request authentication, usually in the form of a token. The user will then forward this request to an authentication server, which will either reject or allow this authentication. From here, the token is provided to the user, and then to the requester. Such a token can then be checked at any time independently of the user by the requester for validation and can be used over time with strictly limited scope and age of validity.

1. **JSON Web Tokens (JWT)**

JSON Web Tokens[[25]](#footnote-25) are an open, industry-standard RFC 7519 method  
for representing claims securely between two parties. To authenticate a user, a client application must send a JSON Web Token (JWT) in the authorization header of the HTTP request to the API. API Gateway validates the JWT token as per the predefined configuration.

**10.3.4 API Monitoring**

Monitoring and business reports help developers to control their usage of APIs in real-time and through historical analytics. It is important part of dashboard in developer portal.

API monitoring & analytics provide the ability to measure and monitor success with visibility across the digital business value chain.

A complete API monitoring and analytics solution should provide real-time analytics with below capabilities:

**Operational metrics**: real-time, end-to-end visibility of API usage.

**Developer metrics**: API consumer adoption metrics with visibility to top consumers, churn risk and quota status. They are important to understanding API program business success.

**API usage metrics**: measure most and least popular, volume, traffic and TPS.

**API performance metrics**: API quality metrics with latency and error rates.

**API infrastructure metrics**: analysis of resource bottlenecks, error rates and latency.

**Traffic monitoring**: monitor and troubleshoot individual transactions, including drilling into processing steps and message content to perform root-cause analysis.

**Predictive analytics**: they are important for the proactive identification of abnormal situations and key operational insight for decision-making.

Developer portal security is a critical component of Hosting Public API and should account for authorization, privacy, non-repudiation, and attack prevention. An overall focus on avoiding misuse or malicious attacks should be emphasized.

## 10.4 API Developer Portal Example

**Application and API Relation**

The developer can access the API based on the number of plans available against that API and their consumption fitment. In case developer wish to consume another API, the developer must create another application and subscribe to the plan of the other API. The inference that can be called out is that each application is bound with an API.

Diagram

Description automatically generated

Figure 10.2: Application and API Sandbox

In API Sandbox, there will be free plan tagged to the default application while in production API Portal the developer can create multiple applications to subscribe to different custom API plans. By creating multiple applications and subscribing to different plans will help get better visibility and analytics of the API consumption.

# 11. API Performance & API Monitoring

## 11.1 What is API Performance?

API performance refers to how well an API meets the needs of its users and how it operates in terms of speed, reliability, security and efficiency.

The API performance can be measured at Business and Technical Level.

**A diagram of a building

Description automatically generated**

Figure 11.1: API Performance

**11.1.1 Business Level:**

API performance at the business level refers to how well an API meets the needs of its users and contributes to overall business goals.

Key aspects include:

* **User experience**: How easy it is for developers to use the API, how quickly the API can get up and running, and how intuitive the documentation is.
* **Business value**: How the API contributes to the business's core objectives, such as increasing revenue, reducing costs, or improving customer satisfaction.
* **Time to market**: How quickly new APIs or API features can be released.
* **Developer adoption**: The number of developers using the API and how actively they engage with it.

**11.1.2 Technical Level:**

Technically, API performance involves various metrics that measure how efficiently an API processes requests and delivers responses.

Key aspects include:

* **Response time**: The time taken by the API to process a request and return a response.
* **Throughput**: The number of requests an API can handle per unit of time.
* **Latency**: The delay between the sending of a request and the receipt of a response.
* **Error rate**: The frequency of errors or exceptions occurring during API operations.
* **Scalability**: The ability of the API to handle increasing load without compromising performance.
* **Reliability**: The consistency and dependability of the API.
* **Availability**: The percentage of time the API is accessible and functional.
* **Security**: The measures taken to protect the API from unauthorized access or attacks.

## 11.2 Why API Performance is Important?

API performance is crucial for several reasons. It directly impacts user experience, business outcomes, and competitive advantage. A well-performing API can enhance customer satisfaction, drive revenue, and differentiate a business from its competitors.

Key aspects include:

* **Enhanced user experience**: Fast and reliable APIs lead to satisfied users, which can drive customer loyalty and repeat business.
* **Improved business outcomes**: Good API performance can drive revenue, increase customer satisfaction, and reduce operational costs.
* **Competitive advantage**: Well-performing APIs can differentiate a business from its competitors and attract more developers to use its platform.

## 11.3 Key Parameters for API Performance

To ensure optimal API performance, organizations must consider various parameters at both the business and technical levels. These parameters provide a framework for measuring and improving API performance.

**11.3.1 Business Parameters:**

Business parameters focus on how the API contributes to the overall business strategy and objectives. These parameters measure the API's impact on revenue, customer satisfaction, and market competitiveness.

**11.3.1.1 Time to market**:

The speed at which new APIs or API features can be released. A faster time to market can help organizations to respond quickly to changing market conditions and stay ahead of competitors.

**11.3.1.2 Time to onboard:**

The ease with which developers can integrate and use the API. A shorter time to onboard can attract more developers and increase API adoption.

**11.3.1.3 Revenue generation:**

The contribution of the API to overall business revenue. A well-performing API can drive revenue by enabling new business models, increasing customer acquisition, and improving customer retention.

**11.3.1.4 Number of consumers:**

The number of developers or applications using the API. A larger number of consumers indicates a successful and widely adopted API.

**11.3.1.5 Developer satisfaction:**

How satisfied developers are with the API's usability, documentation, and support. A high level of developer satisfaction can lead to increased API adoption and usage.

**11.3.1.6 API monetization:**

If the API is monetized, how effective are the pricing and monetization strategies. A well-designed monetization strategy can generate revenue from the API.

**11.3.2 Technical Parameters:**

Technical parameters measure the API's performance from a technical standpoint. These parameters assess factors such as response time, throughput, error rates, scalability, etc.

**11.3.2.1 Scalability:**

The ability of the API to handle increasing load without compromising performance. A scalable API can accommodate growth and avoid performance bottlenecks.

**11.3.2.2 Elasticity:**

The ability of the API to automatically adjust its resources based on demand. An elastic API can optimize resource utilization and reduce costs.

**11.3.2.3 Availability:**

The percentage of time the API is accessible and functional. High availability is essential for ensuring a positive user experience and minimizing downtime.

**11.3.2.4 Reliability:**

The consistency and dependability of the API. A reliable API is less likely to experience errors or unexpected behavior.

**11.3.2.5 Security:**

The measures taken to protect the API from unauthorized access or attacks. A secure API is essential for protecting sensitive data and preventing security breaches.

**11.3.2.6 Latency:**

The delay between sending of a request and the receipt of a response. Low latency is important for providing a fast and responsive user experience.

**11.3.2.7 Throughput:**

The number of requests an API can handle per unit of time. High throughput is essential for handling large volumes of traffic and ensuring that the API can meet demand.

**11.3.2.8 Error rate:**

The frequency of errors or exceptions occurring during API operations. A low error rate indicates a well-designed and robust API.

## 11.4 API Monitoring

API monitoring is the process of collecting, analysing, and interpreting data related to API performance, usage, and health. By monitoring APIs, organizations can gain valuable insights into their API's performance, identify areas for improvement, and ensure that it is meeting the needs of its users and contributing to overall business success.

The API monitoring can be performed at business, application and infrastructure level

A diagram of a building

Description automatically generated

Figure 11.2: API Monitoring

**11.4.1 Business Level:**

Business-level API monitoring focuses on understanding how the API is being used and its impact on the business. This involves tracking metrics related to API usage, revenue, customer engagement, and adoption. By monitoring these metrics, businesses can gain insights into the API's success, identify areas for improvement, and make data-driven decisions to optimize its performance.

A diagram of a business level

Description automatically generated

Figure 11.3: API Monitoring - Business Level

**11.4.1.1 API usage growth:**

This metric indicates the API's popularity and potential for future growth.

* + **Daily active users**: The number of unique users interacting with the API daily.
  + **Monthly active users**: The number of unique users interacting with the API on a monthly basis.
  + **API call volume**: The total number of API calls made per day, week, or month.

**11.4.1.2 Revenue:**

The revenue generated by the API, whether through direct pricing or indirectly through increased sales of other products or services.

* + **API subscription revenue**: The revenue generated from API subscriptions or usage-based pricing.
  + **Indirect revenue**: The revenue generated through increased sales of other products or services that rely on the API.

**11.4.1.3 Unique API Consumers:**

The number of distinct entities (individuals, organizations) using the API. This metric provides insights into the API's reach and customer base.

* + **Customer acquisition rate**: The rate at which new customers are acquiring and using the API.
  + **Customer churn rate**: The rate at which existing customers stop using the API.

**11.4.1.4 Top customers by API usage:**

The customers that are using the API most frequently or consuming the most resources. Identifying top customers can help businesses tailor support and offerings to their needs.

* + **Customer segmentation**: Grouping customers based on their usage patterns and characteristics.
  + **Customer lifetime value**: The total revenue a customer generates over their lifetime.

**11.4.1.5 API Retention:**

The percentage of existing API consumers who continue to use the API over time. This metric indicates the API's ability to retain customers and prevent churn.

* + **Customer retention rate**: The percentage of customers who remain active and continue to use the API.
  + **Customer lifetime value analysis**: Analysing the factors that contribute to customer retention and identifying opportunities for improvement.

**11.4.1.6 Time to first hello world:**

The time it takes for a new developer to successfully make their first API call. A shorter time to first hello world indicates a well-documented and easy-to-use API.

* + **Developer onboarding experience:** Assessing the ease with which developers can get started using the API.
  + **Documentation quality:** Evaluating the quality and comprehensiveness of the API documentation.

**11.4.1.7 SDK and Version Adoption:**

The popularity of different SDKs and API versions among developers. This information can help organizations prioritize support and development efforts.

* + **SDK usage statistics**: Tracking the adoption of different SDKs and their usage patterns.
  + **API version usage**: Monitoring the usage of different API versions and identifying trends.

**11.4.2 Application Level:**

Application-level API monitoring focuses on the performance and behavior of the API from the perspective of individual applications. This involves tracking metrics related to response times, request rates, errors, and latency.

A diagram of application level

Description automatically generated

Figure 11.4: API Monitoring - Application Level

**11.4.2.1 Average API response time:**

The average time it takes for the API to process a request and return a response. A low average response time indicates a fast and efficient API.

* + **Response time distribution**: Analysing the distribution of response times to identify outliers and potential performance issues.
  + **Response time by API endpoint**: Tracking response times for different API endpoints to identify performance bottlenecks.

**11.4.2.2 Requests per minute & bytes per request:**

The number of requests the API receives per minute and the average size of those requests. These metrics provide insights into the API's load and resource usage.

* + **Request rate trends**: Analysing trends in request rates over time to identify peak usage periods and potential bottlenecks.
  + **Request size distribution**: Examining the distribution of request sizes to identify patterns and optimize API design.

**11.4.2.3 Average and max latency:**

The average and maximum delay between the sending of a request and the receipt of a response. Low latency is essential for a positive user experience.

* + **Latency distribution**: Analysing the distribution of latency to identify outliers and potential performance issues.
  + **Latency by API endpoint**: Tracking latency for different API endpoints to identify performance bottlenecks.

**11.4.2.4 Errors per minute:**

The number of errors or exceptions occurring during API operations. A high error rate can indicate issues with the API's functionality or performance.

* + **Error types**: Identifying the types of errors occurring and their frequency.
  + **Error correlation analysis**: Correlating errors with other metrics to identify root causes.

**11.4.3 Infrastructure Level:**

Infrastructure-level API monitoring focuses on the health and performance of the systems and infrastructure supporting the API. This involves tracking metrics related to uptime, resource utilization, and system health.

A diagram of a computer process

Description automatically generated

Figure 11.5: API Monitoring - Infrastructure Level

**11.4.3.1 API uptime:**

The percentage of time the API is available and functional. High uptime is essential for ensuring a reliable and consistent API experience.

* + **Uptime SLA**: Defining and tracking service level agreements (SLAs) for API uptime.
  + **Outage analysis**: Investigating the causes of API outages and implementing measures to prevent them.

**11.4.3.2 CPU usage:**

The percentage of CPU resources being utilized by the API. High CPU usage can indicate performance bottlenecks or resource constraints.

* + **CPU utilization trends**: Analysing trends in CPU usage to identify peak usage periods and potential bottlenecks.
  + **CPU utilization by process**: Tracking CPU usage for different processes to identify resource-intensive components.

**11.4.3.3 Memory Usage:**

The amount of network bandwidth being consumed by the API. High network usage can indicate performance bottlenecks or network congestion.

* + **Network traffic analysis**: Analyzing network traffic to identify patterns, anomalies, and potential security threats.
  + **Network latency**: Measuring the latency of network connections to identify potential performance issues.

**Examples of API Management & API Monitoring tools:**

* + 1. Red Hat 3 scale[[26]](#footnote-26)
    2. Amazon API gateway[[27]](#footnote-27)
    3. Microsoft Azure API management[[28]](#footnote-28)
    4. Apigee API management[[29]](#footnote-29)
    5. Mulesoft Anypoint Platform[[30]](#footnote-30)

# 12. APIs and Artificial Intelligence

## 12.1 Introduction

In today's digital age, APIs (Application Programming Interfaces) have become the backbone of modern software development, facilitating seamless communication and data exchange between different systems. Artificial Intelligence (AI), on the other hand, has revolutionized various industries with its ability to analyze data, learn from patterns, and make intelligent decisions. When combined, AI and APIs can work together to unlock new possibilities and drive innovation across a wide range of applications.

This chapter will explore the interconnected relationship between AI and APIs, focusing on:

* **AI-Enabling APIs**: How AI empower developers to design, implement and manage APIs effectively.
* **APIs Enabling AI**: How APIs provide the necessary infrastructure and resources to support the development and deployment of AI applications.

## 12.2 Artificial Intelligence Enabling APIs

Artificial Intelligence (AI) can be helpful in all stages of API Life cycle such as API Design, Development, Testing and Management.

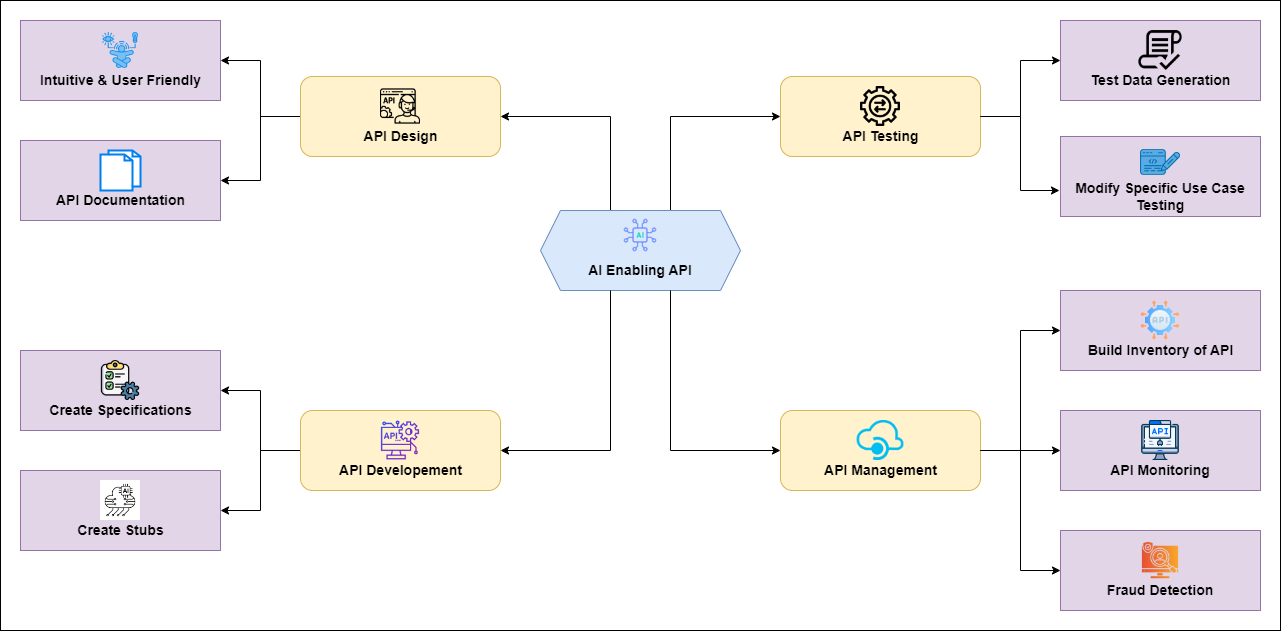


Figure 12.1: AI Enabling APIs

* + 1. **API Design:**

**12.2.1.1 Intuitive & User friendly**

AI can assist in API design by using large language models (LLMs). It can help the developers in improving the layout of API resources, making them more intuitive and user-friendly. Additionally, AI can assist in mapping use cases into the correct API protocol and style semantics, ensuring that the APIs align with industry best practices.

**12.2.1.2 API documentation**

AI with the help of Natural Language Processing (NLP), can generate detailed documentation, including examples, usage guidelines, and error-handling information, directly from the API code. This makes it easier for developers to find the information they need.

* + 1. **API Development:**

**12.2.2.1 Create Specifications**

AI can assist in creating API specifications by analyzing existing code, data structures, and business requirements.

**12.2.2.2 Create Stubs to Integrate or Expose APIs**

AI can generate stubs or mock APIs that simulate the behavior of real APIs, allowing developers to test and integrate their applications without relying on the actual APIs. This can accelerate the API development process.

* + 1. **API Testing:**

**12.2.3.1 Test Data Generation**

AI can generate realistic and diverse test data to thoroughly test APIs under various scenarios.

**12.2.3.2 More Specific Use Case Testing**

AI can analyze API usage patterns and identify specific use cases to focus on testing. This can ensure that the API is optimized for the most common scenarios and reduce the risk of unexpected failures.

* + 1. **API management:**

**12.2.4.1 Build Inventory of APIs**

AI can automatically catalog and classify APIs, making it easier to manage and discover them. This can help organizations get a better understanding of their API landscape and identify opportunities for reuse and optimization.

**12.2.4.2 API Monitoring**

AI can monitor API usage in real time and detect anomalies or performance issues. This can help prevent downtime and ensure that APIs are meeting service level agreements (SLAs).

**12.2.4.3 Fraud Detection**

AI can analyze API traffic and identify patterns that may indicate fraudulent activity. This can help protect APIs and prevent unauthorized access.

## 12.2 APIs Enabling Artificial Intelligence

APIs also play a crucial role in enabling AI applications by providing access to data, algorithms, and computational resources. Here are some keyways in which APIs are used for AI:

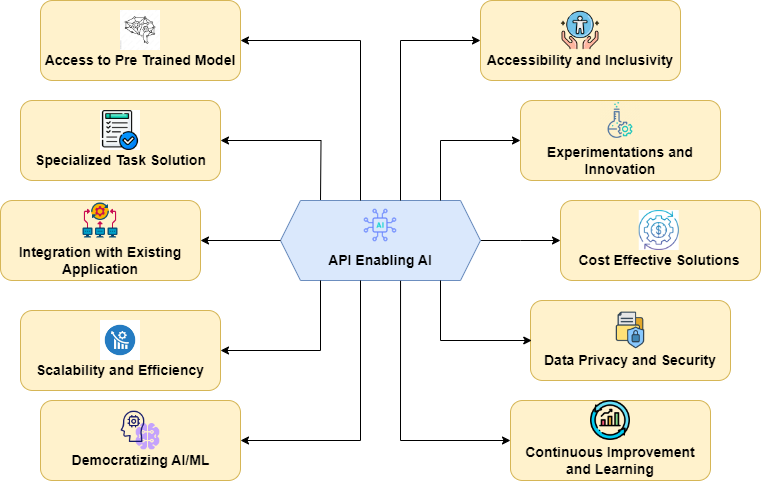


Figure 12.2: APIs enabling AI

**12.2.1 Access to Pre-trained Models:**

APIs offer access to pre-trained models that have been developed and trained on massive datasets by experts. This eliminates the need for organizations to invest significant time and resources in training their own models.

**12.2.2 Specialized Task Solutions:**

APIs can be tailored to address specific tasks within the AI/ML domain, such as sentiment analysis, fraud detection, or recommendation systems. These specialized APIs offer ready-to-use solutions that are optimized for that particular task.

**12.2.3 Integration with Existing Applications:**

APIs provide a standardized way for AI/ML models to interact with other software applications, enabling seamless integration of AI/ML capabilities into existing systems.

**12.2.4 Democratizing AI/ML:**

By lowering the technical barriers to entry, APIs make AI/ML technology accessible to a wider range of developers and businesses, fostering innovation and creativity.

**12.2.5 Cost-Effective Solutions:**

Using AI/ML APIs can be a cost-effective way to incorporate AI/ML capabilities into applications, compared to building and training models from scratch.

**12.2.6 Experimentation and Innovation:**

APIs provide a flexible platform for experimentation and innovation, allowing developers to try out different AI/ML models and algorithms to find the best solutions for their specific needs.

**12.2.7 Accessibility and Inclusivity:**

AI/ML APIs can be used to create more accessible and inclusive applications.

# List of Abbreviations

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Abbreviation** | **Full Form** |
| 1 | ReBIT | Reserve Bank Information Technology Private Limited |
| 2 | SSDLC | Secure Software Development Life Cycle |
| 3 | API | Application Programming Interface |
| 4 | SRS | Software Requirement Specification |
| 5 | HLD | High Level Design |
| 6 | LLD | Low Level Design |
| 7 | CS | Cyber Security |
| 8 | QA | Quality Assurance |
| 9 | CI/CD | Continuous Integration/Continuous deployment |
| 10 | SAST | Static Application Security Testing |
| 11 | VAPT | Vulnerability Assessment and Penetration Testing |
| 12 | UAT | User Acceptance Testing |
| 13 | OTP | One Time Password |
| 14 | SLA | Service level access |
| 15 | CORS | Cross-origin resource sharing |
| 16 | SDKs | Software development kit |
| 17 | B2B | Business-to-business |
| 18 | JWT | JSON Web Token |
| 19 | HTTP | Hypertext Transfer Protocol |
| 20 | URL | Uniform Resource Locator |
| 21 | JSON | JavaScript Object Notation |
| 22 | XML | Extensible Markup Language |
| 23 | XSLT | Extensible Stylesheet Language Transformations |
| 24 | IT | Information technology |
| 25 | SIT | System Integration Testing |
| 26 | MVP | Minimum viable developer portal |
| 27 | TPS | Transactions Per Second |
| 28 | FAQs | Frequently Asked Questions |
| 29 | REST | Representational state transfer |
| 30 | OAuth | Open Authorization |
| 31 | EULAs | End User License Agreement |
| 32 | Org | Organization |
| 33 | SMTP | Simple Mail Transfer Protocol |
| 34 | APM | Application Performance Monitoring |
| 35 | CPU | Central Processing Unit |
| 36 | DX | Developer Experience |
| 37 | KPI | Key Performance Indicator |
| 38 | MAU | Monthly Active Users |
| 39 | RBI | Reserve Bank of India |
| 40 | RPM | Request Per Minute |
| 41 | SaaS | Software As A Service |
| 42 | SDK | Software Development Kit |
| 43 | SMS | Short Message Service |
| 44 | SQL | Structured Query Language |
| 45 | SSH | Secure Socket Shell |
| 46 | TTFHW | Time To First Hello World |

References**:**

This book has taken references from various sources like websites and blogs from various people and organizations. Their ideas and contributions to the specific topics can be read at below links:

API Security:

https://www.geeksforgeeks.org/network-layer-services-packetizing-routing-and-forwarding/

https://www.oreilly.com/library/view/api-traffic-management/9781492056393/ch01.html

[https://www.akamai.com/glossary/what-is-api-gateway-security#:~:text=API%20gateways%20provide%20a%20centralized,favorite%20target%20of%20malicious%20actors](#:~:text=API%20gateways%20provide%20a%20centralized,favorite%20target%20of%20malicious%20actors)

https://learn.microsoft.com/en-us/azure/api-management/api-management-gateways-overview

API Gateway and API Policy Guidelines:

https://www.tigera.io/learn/guides/service-mesh/service-mesh-architecture/

https://blog.dreamfactory.com/service-mesh-vs-api-gateway

https://konghq.com/blog/learning-center/what-is-a-service-mesh

MuleSoft Policies: Govern and Manage APIs efficiently | by Prashanth Kurimella | Medium

https://www.oreilly.com/library/view/data-management-at/9781492054771/ch04.html

API Developer Portal:

https://developers.bri.co.id/en/news/api-sandbox-definition-benefits-and-best-practices

https://www.neosalpha.com/api-developer-portal-features/

https://www.pingidentity.com/en/resources/identity-fundamentals/authentication/single-factor-two-factor-multi-factor-authentication.html

https://techdocs.broadcom.com/us/en/ca-enterprise-software/layer7-api-management/api-developer-portal/4-5/user-types-roles-and-permissions.html

API Performance and API Monitoring:

[https://www.ioriver.io/terms/api-performance#api-performance-testing-metrics](#api-performance-testing-metrics)

https://www.moesif.com/blog/technical/api-metrics/API-Metrics-That-Every-Platform-Team-Should-be-Tracking/

API and Artificial Intelligence:

[https://nordicapis.com/how-will-ai-enable-apis/#:~:text=AI%20can%20be,industry%20best%20practices](#:~:text=AI%20can%20be,industry%20best%20practices)

[https://www.ibm.com/topics/natural-language-processing#:~:text=In%20document%20processing,context%20and%20nuances](#:~:text=In%20document%20processing,context%20and%20nuances)

1. https://grpc.io/ [↑](#footnote-ref-1)
2. https://graphql.org/ [↑](#footnote-ref-2)
3. https://swagger.io/ [↑](#footnote-ref-3)
4. https://raml.org/ [↑](#footnote-ref-4)
5. https://medium.com/@juannegrin/fundamental-software-architecture-principles-separation-of-concerns-modularity-and-abstraction- [↑](#footnote-ref-5)
6. https://microservices.io/patterns/reliability/circuit-breaker.html [↑](#footnote-ref-6)
7. https://blog.stoplight.io/api-first-api-design-first-or-code-first-which-should-you-choose [↑](#footnote-ref-7)
8. https://github.com/Redocly/redoc [↑](#footnote-ref-8)
9. https://editor.swagger.io/ [↑](#footnote-ref-9)
10. https://www.softwaretestinghelp.com/white-box-testing-techniques-with-example/ [↑](#footnote-ref-10)
11. https://learning.postman.com/docs/getting-started/creating-the-first-collection/ [↑](#footnote-ref-11)
12. https://learning.postman.com/docs/writing-scripts/intro-to-scripts/ [↑](#footnote-ref-12)
13. A “branching strategy” refers to the strategy a software development team employs when writing, merging, and shipping code in the context of a version control system like Git. Branching Strategy can differ from project to project. [↑](#footnote-ref-13)
14. https://semver.org/ [↑](#footnote-ref-14)
15. https://docs.apigee.com/api-monitoring [↑](#footnote-ref-15)
16. https://docs.mulesoft.com/monitoring/ [↑](#footnote-ref-16)
17. https://microservices.io/patterns/reliability/circuit-breaker.html [↑](#footnote-ref-17)
18. https://semver.org/ [↑](#footnote-ref-18)
19. The details about **cancelled** status are provided in the “Amendment of API version” section. [↑](#footnote-ref-19)
20. API Catalog - APIs grouped together as per functionality or system/entity. [↑](#footnote-ref-20)
21. https://tools.ietf.org/id/draft-dalal-deprecation-header-01.html [↑](#footnote-ref-21)
22. https://tools.ietf.org/html/rfc8594 [↑](#footnote-ref-22)
23. https://access.redhat.com/documentation/en-us/red\_hat\_3scale\_api\_management/2.6/html/creating\_the\_developer\_portal/index [↑](#footnote-ref-23)
24. https://www.mulesoft.com/platform/api/developer-portal [↑](#footnote-ref-24)
25. https://jwt.io/introduction/ [↑](#footnote-ref-25)
26. https://www.redhat.com/en/technologies/jboss-middleware/3scale [↑](#footnote-ref-26)
27. https://aws.amazon.com/api-gateway/ [↑](#footnote-ref-27)
28. https://azure.microsoft.com/en-us/products/api-management [↑](#footnote-ref-28)
29. https://cloud.google.com/apigee [↑](#footnote-ref-29)
30. https://anypoint.mulesoft.com/home/ [↑](#footnote-ref-30)