NGN Technologies Private Limited

API/DB Usage Guideline

v 1.0

Date: 19/03/2024

DOCUMENT REVISION HISTORY					
Revision No.	**Issue Date**	**Change Details**	**Authored by**	**Reviewed by**	**Approved by**
1.0	20th Feb, 2024	First Release	Khemlal Chhetri		
2.0	27th Mar, 2024	Second Release	Khemlal Chhetri		

Contents

Introduction

These guidelines are designed to help developers integrate and interact with our API effectively. Please read through the following sections to understand how to use the API securely and efficiently.

1. Purpose

This API guideline document serves multiple essential purposes in the development and integration of an API. Primarily, it aims to provide clarity and understanding by outlining the API's functionalities, endpoints, parameters, and expected responses. By establishing standards and best practices, these documents ensure consistency across applications and developers, facilitating interoperability and ease of maintenance. Through tutorials, code examples, and practical demonstrations, API guideline documents streamline the integration process for app vendors, reducing development time and errors.

Additionally, they serve as a reference point for ongoing support, fostering collaboration between providers and consumers and enabling effective adaptation to changes and updates. Moreover, these documents often include information about security measures and compliance requirements, ensuring the secure and regulatory-compliant use of the API. In essence, the creation of an API guideline document is intended to empower app vendors with the necessary knowledge and resources for successful API implementation, thereby contributing to the overall efficiency and effectiveness of the API ecosystem.

2. Overview of the Platform

The Digital Health Platform is a comprehensive system designed to revolutionize healthcare delivery by leveraging advanced technology and AWS services. AWS provides a scalable, reliable, and secure infrastructure for running various workloads, from simple web applications to complex enterprise solutions. Here's an overview of the platform's key components and how it utilizes AWS services for authentication, ID management, consent management, API management, and data access control:

2.1. Authentication and ID Management:

- The DHP utilizes AWS Identity and Access Management (IAM) to manage user identities, access permissions, and authentication mechanisms.
- Enables secure authentication of healthcare providers, patients, and other stakeholders accessing the platform's resources and services.
- Implements role-based access control (RBAC) to ensure proper user authentication and authorization.
- Authentication will be used by two types of users, citizens and health professionals. For citizens, first layer of authentication will be taken care by NDI.

2.2. Consent Management:

- Leverages AWS services to implement robust consent management mechanisms, allowing patients to control access to their health data and make informed decisions about data sharing.
- Integrates with AWS Key Management Service (KMS) for encrypting sensitive patient data and managing encryption keys securely.

2.3. API Management:

- Utilizes AWS API Gateway to manage APIs securely, control access, and enforce usage policies.
- Enables developers to create, publish, monitor, and secure APIs, facilitating seamless integration with third-party applications and services.

The main system flow happens through API and DHP will host several API’s for other systems to function. For the development of API, we are using AWS Lambda, RDS, S3 and CloudWatch services. By leveraging AWS Lambda, RDS, S3, and CloudWatch, the Digital Health Platform offers a scalable, reliable, and secure solution for transforming healthcare delivery. These AWS services enable the platform to handle diverse healthcare workloads, manage patient data effectively, and ensure operational excellence with proactive monitoring and management capabilities.

2.4. Benefits:

- **Enhanced Security**: Leveraging AWS security features ensures the confidentiality, integrity, and availability of patient health data, mitigating security risks and compliance challenges.
- **Scalability and Reliability**: AWS infrastructure provides scalable and reliable services, enabling the platform to handle varying workloads, accommodate growth, and maintain high availability.
- **Interoperability**: Facilitates seamless integration with existing healthcare systems and interoperability standards, enabling data exchange and collaboration across different healthcare organizations and systems.
- **Cost Optimization**: AWS's pay-as-you-go pricing model allows the platform to optimize costs by only paying for the resources and services consumed, minimizing upfront investments and infrastructure overhead.

3. Overview of the API Ecosystem

The DHP API ecosystem serves as the backbone of digital connectivity, facilitating seamless communication and integration between various systems, applications, and stakeholders. APIs provide standardized interfaces for accessing data, services, and functionalities, enabling agility, scalability, and innovation in today's interconnected world. Some of the key components of the ecosystem are:

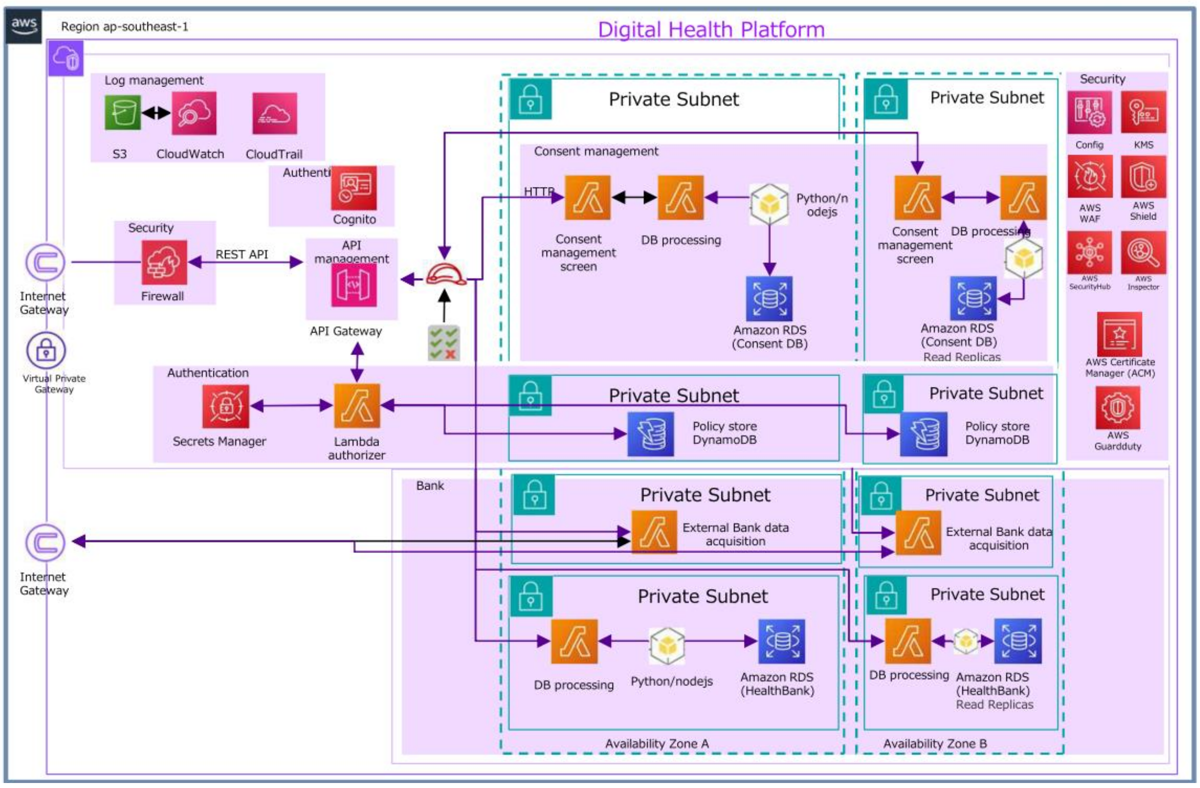
- A curated collection of APIs offering a range of functionalities and services, accessible to developers and consumers.
- The API Gateway acts as the central entry point for external requests, providing security, routing, and monitoring capabilities.
- The backend services power the APIs by executing business logic, processing data, and interacting with underlying systems or databases.
- APIs are consumed by developers, applications, and systems to access data, services, and functionalities.
- APIs facilitate the exchange of data between different systems, ensuring seamless interoperability and information flow.

- APIs enable rapid development and deployment of new features, promoting agility and innovation.
- APIs support scalable architectures, accommodating growth and adapting to changing demands or requirements.

4. API Architecture

The API architecture provides a comprehensive view of the system's API layer, showcasing the various components and their interactions. Key elements of the architecture include:

- API gateway which acts as the entry point for all external requests. It also enforces security policies, throttling, and request validation to ensure secure and reliable API access.
- Lambda function that powers the backend logic and processing of API requests, enabling serverless execution of code in response to events. It also handle specific API endpoints and operations, performing tasks such as data validation, authentication, and business logic execution.
- RDS for storing and managing the application's data, including user information, preferences, and transactional data. It will be used by Lambda functions to perform CRUD (Create, Read, Update, Delete) operations and retrieve data for API responses.
- Integrates with IAM (Identity and Access Management) or other authentication services to validate user credentials and authorize access to protected resources.
- Utilizes CloudWatch or other monitoring tools to track API performance, error rates, and usage metrics.



5. Design Principles

Security and authentication are paramount in the design of any digital platform, especially in the healthcare sector where sensitive patient data is involved. Here are some design principles to ensure robust security and authentication mechanisms in the Digital Health Platform (DHP):

- **Least Privilege:** Follow the principle of least privilege by granting users only the minimum level of access rights necessary to perform their roles or tasks within the DHP. Limit access to sensitive data and functionalities to authorized personnel only.
- **Encryption Everywhere**: Implement encryption mechanisms to protect data both in transit and at rest within the DHP. Use strong encryption algorithms and key management practices to safeguard sensitive information, such as patient health records and authentication credentials.
- **Authentication and Authorization**: Users must authenticate using valid credentials (e.g., API keys, OAuth tokens) before accessing protected API endpoints. Access to sensitive resources and operations is restricted based on user roles and permissions defined in the authorization layer.
- Each API endpoint will be thoroughly documented, including its purpose, input parameters, expected response format, and error handling guidelines.
- Error Handling: APIs should adhere to standard HTTP status codes for indicating the success or failure of requests (e.g., 200 OK, 404 Not Found, 500 Internal Server Error).
- Implement rate limiting and throttling mechanisms to prevent abuse and ensure fair usage of API resources.
- Adhere to secure coding practices and conduct regular security assessments, code reviews, and penetration testing to identify and remediate security vulnerabilities in DHP components and applications.

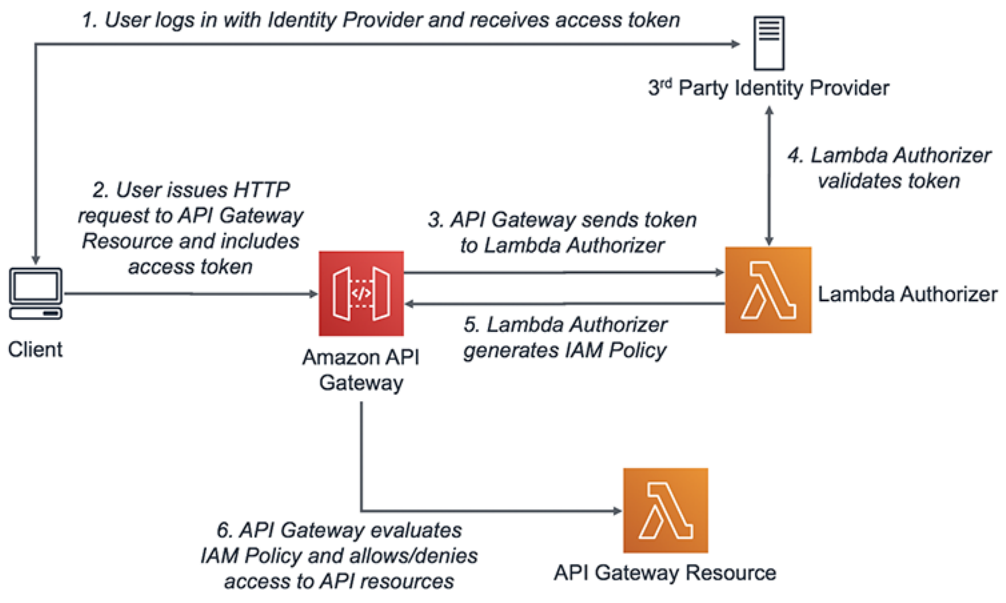
5.1. Authentication patterns

DHP uses following authentication patterns:

Bearer Authentication (OAuth 2.0):

- **Description:** Bearer authentication involves issuing access tokens to clients, which they include in the Authorization header of API requests. These tokens are typically obtained through an OAuth 2.0 authorization flow.

- **Usage:** Bearer authentication is widely used for securing APIs due to its flexibility, scalability, and support for various grant types (e.g., authorization code, implicit, client credentials). It provides fine-grained access control and supports token expiration and refresh mechanisms for improved security.


1. User logs in with Identity Provider and receives access token
3rd Party Identity Provider
4. Lambda Authorizer validates token
2. User issues HTTP request to API Gateway Resource and includes access token
3. API Gateway sends token to Lambda Authorizer
Client
Amazon API Gateway
Lambda Authorizer
5. Lambda Authorizer generates IAM Policy
6. API Gateway evaluates IAM Policy and allows/denies access to API resources
API Gateway Resource

5.2. API keys for authorization

API keys are typically associated with the Amazon API Gateway service.

- API keys are used to control access to APIs deployed on Amazon API Gateway.
- They provide a simple way to authenticate clients and enforce usage limits on API endpoints.
- API keys can be generated and managed directly within the Amazon API Gateway console or through the AWS CLI/API.
- Users can create multiple API keys and associate them with different stages (e.g., development, production) of their APIs.
- API keys are typically included in the headers of HTTP requests made to API Gateway endpoints.
- They are often used in combination with other authentication mechanisms, such as IAM roles and policies, for added security.
- API keys can be associated with IAM policies to control access to specific API resources and methods.
- Users can revoke or regenerate API keys as needed to mitigate security risks, such as key compromise or unauthorized usage.
- IAM policies can grant or restrict permissions based on API key usage, allowing fine-grained access control.

6. API Technical Specification

6.1. API Architecture and Style:

- The APIs will follow a Representational State Transfer (REST) architecture style, which emphasizes a stateless client-server communication model and uniform resource identifiers (URIs) for resource identification.
- RESTful principles will guide the design of API endpoints, HTTP methods, and resource representations for accessing and manipulating healthcare data.

6.2. Data Formats:

- Data exchanged between clients and the API will primarily use JavaScript Object Notation (JSON) format due to its lightweight and human-readable nature.
- JSON will be used for both request payloads and response bodies, enabling efficient data transmission and consumption across different platforms and devices.

6.3. HTTP Methods and Status Codes:

- The APIs will support standard HTTP methods, including GET, POST, PUT, PATCH, and DELETE, for performing CRUD (Create, Read, Update, Delete) operations on healthcare data.
- HTTP status codes will be used to indicate the outcome of API requests, such as success (2xx), client errors (4xx), and server errors (5xx), ensuring clear communication between clients and servers.

6.4. Authentication and Authorization Mechanisms:

- API authentication will be implemented using JSON Web Tokens (JWT) with OAuth 2.0 authorization framework.
- Clients will obtain JWT tokens by authenticating with the Authentication Service and include them in the Authorization header of API requests.
- Authorization will be enforced based on the roles and permissions associated with the authenticated user, allowing granular access control to API resources.

6.5. Error Handling and Response Codes:

- The APIs will adhere to consistent error handling practices, returning appropriate HTTP status codes and error messages in response to invalid requests or server-side errors.
- Common error scenarios, such as validation failures, unauthorized access, or resource not found, will be properly documented and communicated to clients for troubleshooting purposes.

6.6. Data Validation and Sanitization:

- Input data received from clients will undergo validation and sanitization to prevent injection attacks, data corruption, or security vulnerabilities.
- Validation rules will be applied to request payloads to ensure data integrity, format compliance, and adherence to business rules before processing the requests.

6.7. Obtaining API Credentials

To use our API, you'll need to obtain API credentials (API key, tokens, etc.). First you need to obtain the token generation url to access other API's. The token generation URL is listed below:

URL = http://localhost:8080/getToken

6.8. Making Your First Request

Make a simple API request to ensure your setup is correct. Here's an example using [curl](http://localhost:8080/getToken/):

Bash example:

curl -X GET "http://localhost:8080/validateUser" -H "Authorization: Bearer YOUR_ACCESS_TOKEN"

6.9. Endpoints, Request Format, Response Format and Methods

- All the clients need to send requests in valid JSON format.
- Each API endpoint will have specific HTTP method. This will be found on the swagger what kind of methos is used. We normally use (GET, POST, PUT, DELETE).
- The endpoint URL defined includes path parameters, query parameters, or request body, as applicable. This will also be made available in swagger.

Sl No	API Name	Endpoint	Method	Request Format	Response Format	Authentication
1	Token generation	/generate-token	GET	None	JSON { “token”:”string” }	JWT
2	Authentication Function (Common to citizen and professionals)	/auth/validate	GET	JSON *{ “userid”:“string” }*	JSON *{ “usertype”:”string” }*	API Gateway
3	Authentication for professionals	/auth/login	GET	*{ “userid”:“string”, “password”:“string” }*	*{ “status”:“string” }*	
4	Authentication for citizens using NDI	/auth/citizenlogin	GET	*{ “secret”:“string” }*	*{ “value”:“string” } Value received is the CID*	
5	ID Management Save	/idm/saveid	POST	*{ “dhpId”:“string”, “hbId”:“ string” }*	*{ “saveStatus”:“string” } Either success or failure*	
6	Edit and update information for ID Management	/idm/update	POST	*{ “Id”:“string” }*	*{ “saveStatus”:“string” } Either success or failure*	
7	Display ID management information	/idm/getAllById	GET	*{ “Id”:“string” }*	*{ “dhpId”:“string”, “nid”:“string”,*	

					"hbId":"string", *"bbId":"string",* *"uhId":"string",* *"regDate":"date",* *"updDate":"date",* *"regApp":"string"* *}*	

8	Error handling	/error	GET	*{*
				"errorcode":"string"
				}
					{
					"response":"string"
					}	

9	ID & Login Validation	/validateID	GET	*{*
				"dhpID":"string"
				}
					{
					"status":"string"
					}	

10	Register Consent	/consent/register (First time register)	POST	*{*
				"dhpID":"string",
				"category":"string",
				"consent":"boolean",
				"date":"date"
				}
					{
					"status":"string",
					}	

11	Edit & Update Consent	/consent/update	POST	*{*
				"dhpID":"string",
				"category":"string",
				"consent":"boolean",
				"date":"date"
				}
					{
					"status":"string",
					}	

12	Read Consent	/consent/getAll	GET	*{*
				"dhpID":"string"
				}
					{
					"dhpId":"string",
					"category":"string",
					"healthBank":"string",
					"bioBank":"string",
					"hhBank":"string",
					"medBank":"string",
					"regDate":"date",
					"updDate":"date",
					"regApp":"string"
					}	

13	Register Assessment Result	/assessment/save	POST	*{*
				"hbID":"string",
				"question":"int",
				"answer":"string",
				"assessmentDate":
				"date",
				"regDate":"date",
				"updatedDate":
				"date",
				"regApp":"string",
				"sequence":"int",
				}
					{
					"status":"string"
					}	

14	Update Assessment Result	/assessment/ update	POST	*{ “hbID”:“string”, “question”:“int”, “answer”:“string”, “assessmentDate”: “date”, “regDate”:“date”, “updatedDate”: “date”, “regApp”:“string”, “sequence”:“int”, }*	*{ “status”:“string” }*	
15	Delete Assessment Result	/assessment/del	POST	*{ “hbID”:“string” }*	*{ “status”:“string” }*	
16	Read Assessment Result	/assessment/getAll	GET	*{ “hbID”:“string” }*	*{ “hbID”:“string”, “question”:“int”, “answer”:“string”, “assessmentDate”: “date”, “regDate”:“date”, “updatedDate”: “date”, “regApp”:“string”, “sequence”:“int”, }*	
17	Register Examination Result	/exam/save	POST	*{ “hbID”:“string”, “data”:“int”, “measureDate”: “date”, “regDate”:“date”, “updatedDate”: “date”, “regApp”:“string”, “sequence”:“int”, }*	*{ “status”:“string” }*	
18	Update Examination Result	/exam/update	POST	*{ “hbID”:“string”, “data”:“int”, “measureDate”: “date”, “regDate”:“date”, “updatedDate”: “date”, “regApp”:“string”, “sequence”:“int”, }*	*{ “status”:“string” }*	

19	Delete Examination Result	/exam/del	POST	*{ "hbID":"string" }*	*{ "status":"string" }*	
20	Read Examination Result	/exam/getAllByID	GET	*{ "hbID":"string" }*	*{ "hbID":"string", "data":"int", "measureDate": "date", "regDate":"date", "updatedDate": "date", "regApp":"string", "sequence":"int", }*	
21	File Format Conversion	/file/convert	GET	*{ "data":"string" }*	*{ "response":"string" }*	

6.10. Data Validation and Sanitization

Below are the data validation and sanitization guidelines:

6.10.1. Data Validation:

- dhpID: Ensure that the `dhpID` parameter is a non-empty string and follows any specific format requirements (e.g., alphanumeric characters).
- category: Validate that the `category` parameter is a non-empty string and matches predefined categories (if applicable).
- healthBank, bioBank, hhBank, medBank: Validate each of these fields to ensure they are valid strings and meet any length or format requirements.
- regDate, updDate: Validate that the `regDate` and `updDate` parameters are in the correct date format (e.g., YYYY-MM-DD) and represent valid dates.
- regApp: Ensure that the `regApp` parameter is a non-empty string and meets any specific requirements (e.g., maximum length).

6.10.2. Data Sanitization:

- Input Sanitization: Apply input sanitization techniques to mitigate the risk of injection attacks (e.g., SQL injection, XSS). This may involve removing or encoding special characters from user-supplied data before processing.
- Date Sanitization: For date parameters (`regDate`, `updDate`), ensure that any input outside the expected date range or with invalid characters is sanitized or rejected.
- String Sanitization: Apply string sanitization methods to remove or escape potentially dangerous characters from string inputs to prevent unintended behavior.

6.11. Error codes

Error Code	Description
400	Bad Request - Malformed syntax or invalid request message.
401	Unauthorized - Request requires user authentication.
403	Forbidden - Server refuses to authorize the request.
404	Not Found - Requested resource is not available on the server.
405	Method Not Allowed - The specified method is not allowed.
408	Request Timeout - Server timed out waiting for the request.
429	Too Many Requests - User has sent too many requests in a given time.
500	Internal Server Error - Something has gone wrong on the server.
502	Bad Gateway - Invalid response received from the upstream server.
503	Service Unavailable - Server is currently unable to handle requests.

7. API Development Guidelines

7.1. Naming Conventions:

- Resources: Use plural nouns to represent resources. Use camelCase for multi-word resource names.
  Example: `/consents`

- Endpoints: Use descriptive verbs to represent actions. Use lowercase for endpoint paths.
  Example: `/consents/getAll`

- Parameters: Use camelCase for query parameters and path parameters.
  Example: `dhpId`

- Response Attributes: Use camelCase for attribute names in response bodies.
  Example: `regDate`

7.2. Documentation Standards:

- API Description: Provide a brief overview of the API's purpose and functionality, including any authentication requirements or usage limitations.
- Endpoint Documentation: Document each endpoint with its URL path, supported HTTP methods, request/response formats, and example usage scenarios.
- Parameter Documentation: Document each parameter with its name, data type, description, and whether it's required or optional.
- Authentication and Authorization: Document the authentication mechanism (e.g., JWT tokens) and any authorization rules (e.g., role-based access control) enforced by the API.

7.3. Testing and Quality Assurance Procedures:

- Unit Testing: Write unit tests using testing frameworks like JUnit or pytest to validate the functionality of each API endpoint.

- Integration Testing: Conduct integration tests to verify the interaction between API endpoints and external dependencies (e.g., databases, third-party services).
- Quality Assurance: Implement code review processes to ensure adherence to coding standards, security best practices, and performance optimization techniques.

8. API Governance

8.1. API Usage and Policies Guidelines:

- Define clear policies and guidelines for API usage, including rate limits, authentication mechanisms, data usage policies, and compliance requirements.
- Enforce standards for API naming conventions, request/response formats, error handling, and versioning to ensure consistency and interoperability across APIs.
- Provide documentation and examples to illustrate best practices and usage guidelines for API consumers.
- Regularly review and update API policies to accommodate changes in business requirements, security standards, and regulatory compliance.

8.2. API Support Channels:

- Establish dedicated support channels for API consumers to seek assistance, report issues, and provide feedback.
- Offer multiple channels for support, such as email support, community forums, knowledge bases, and developer portals.
- Assign qualified support staff to respond promptly to inquiries, troubleshoot technical issues, and escalate critical issues as needed.
- Provide self-service resources, including FAQs, troubleshooting guides, and API documentation, to empower users to resolve common issues independently.

8.3. Monitoring and Logging:

- Implement robust monitoring and logging mechanisms to track API usage, performance metrics, and error rates in real-time.
- Utilize monitoring tools like AWS CloudWatch, Amazon CloudWatch Logs, and AWS X-Ray to collect and analyze API metrics, logs, and traces.
- Set up alerts and notifications to proactively detect and respond to anomalies, service disruptions, and performance degradation.
- Use log aggregation and analysis tools to identify trends, troubleshoot issues, and optimize API performance and reliability.

8.4. Incident Response Plan:

- Develop a comprehensive incident response plan to address potential API disruptions, security breaches, and service outages.
- Define clear roles and responsibilities for incident response team members, including communication protocols, escalation procedures, and incident severity levels.
- Conduct regular tabletop exercises and simulations to test the effectiveness of the incident response plan and ensure readiness to handle emergencies.

- Establish incident response playbooks with predefined steps and procedures for mitigating common API-related incidents, such as DDoS attacks, data breaches, and system failures.

9. Examples

9.1. Step-by-Step Guides for Common Use Cases:

Creating a New User Account:

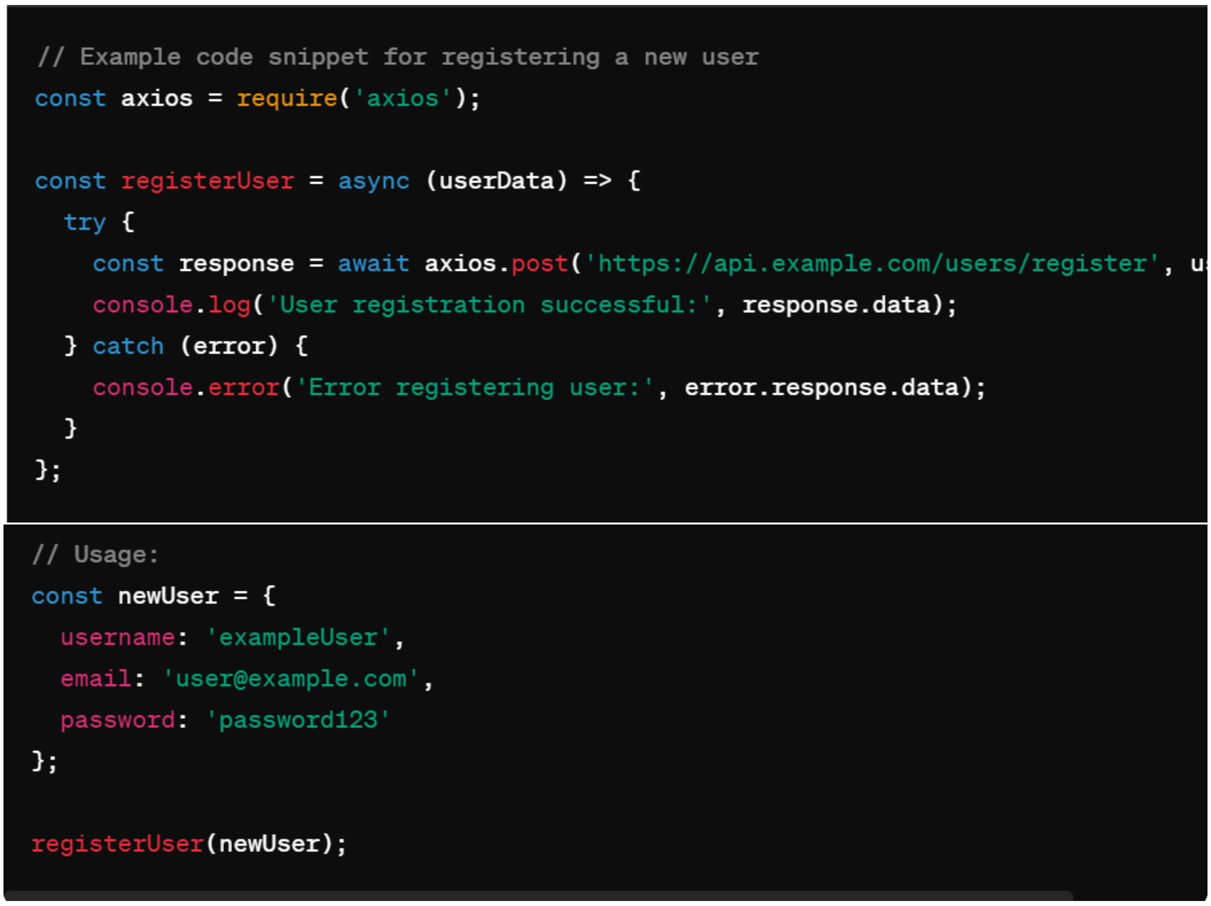
- Step 1: Navigate to the registration endpoint `/users/register`.
- Step 2: Provide required user information such as username, email, and password in the request body.
- Step 3: Send a POST request to the registration endpoint to create a new user account.
- Step 4: Receive a success response with the newly created user's details.

Fetching User Profile Information:

- Step 1: Authenticate the user by obtaining an access token using the `/auth/login` endpoint.
- Step 2: Use the access token to authorize requests to the `/users/profile` endpoint.
- Step 3: Send a GET request to the profile endpoint to retrieve the user's profile information.
- Step 4: Receive a success response with the user's profile data.

9.2. Code Snippets for Various Programming Languages:

9.2.1. Node.js

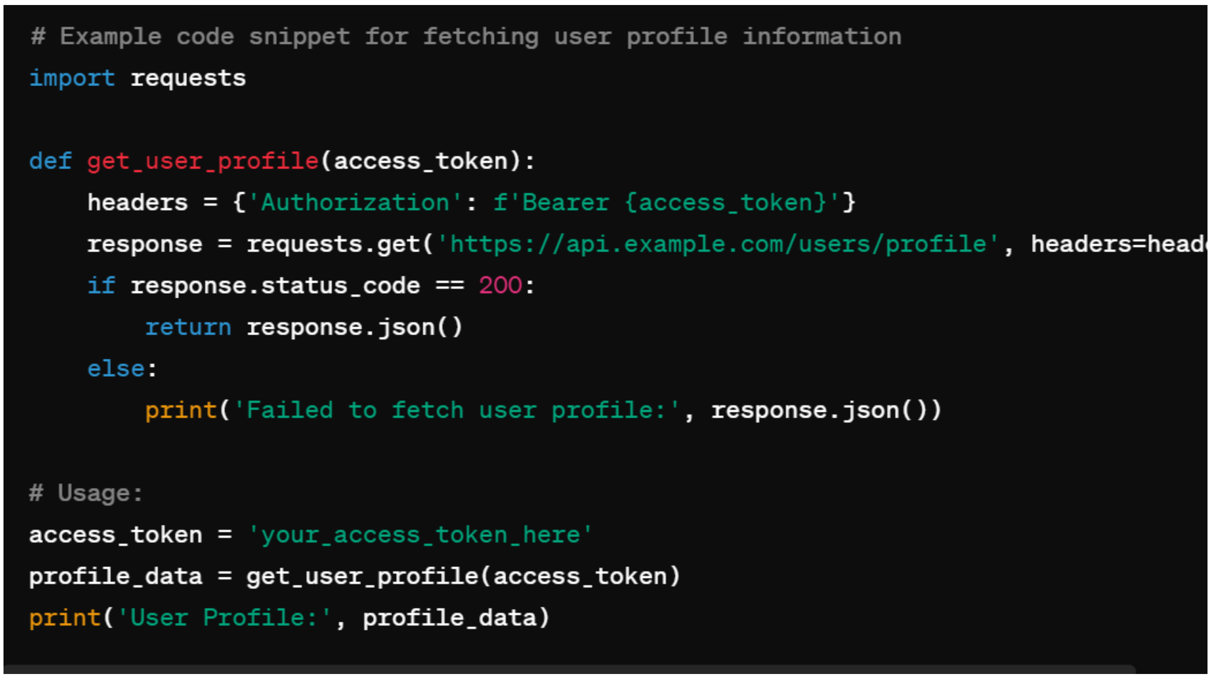


```
// Example code snippet for registering a new user
const axios = require('axios');

const registerUser = async (userData) => {
  try {
    const response = await axios.post('https://api.example.com/users/register', u
    console.log('User registration successful:', response.data);
  } catch (error) {
    console.error('Error registering user:', error.response.data);
  }
};
// Usage:
const newUser = {
  username: 'exampleUser',
  email: 'user@example.com',
  password: 'password123'
};

registerUser(newUser);
```

9.2.2. Python



```
# Example code snippet for fetching user profile information
import requests

def get_user_profile(access_token):
    headers = {'Authorization': f'Bearer {access_token}'}
    response = requests.get('https://api.example.com/users/profile', headers=head
    if response.status_code == 200:
        return response.json()
    else:
        print('Failed to fetch user profile:', response.json())

# Usage:
access_token = 'your_access_token_here'
profile_data = get_user_profile(access_token)
print('User Profile:', profile_data)
```

10. Glossary of Terms

DHP	Digital Health Platform
NDI	National Digital Identity
API	Application Programming Interface
AWS	Amazon Web Services
IAM	Identity and Access Management
RBAC	Role Based Access Control
KMS	Key Management Service
RDS	Relational Database Service
S3	Storage Bucket Service
CRUD	Create, Read, Update, Delete
CLI	Command Line Interface
REST	Representational State Transfer
JSON	JavaScript Object Notation
HTTP	Hyper Text Transfer Protocol
JWT	JSON Web Tokens