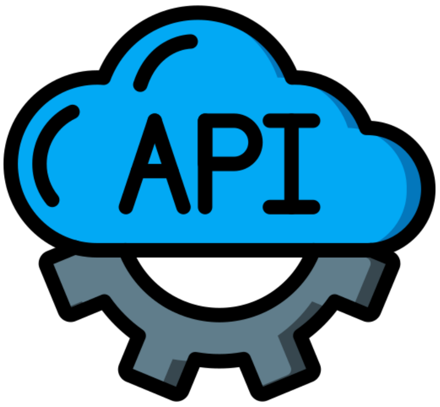


APPLICATION PROGRAMMING INTERFACE

Vishal Kurane

vishalarunkurane@gmail.com | LinkedIn Profile

Contents

Application Programing Interface

1. What is an API?

APIs (Application Programming Interfaces) are essential tools in modern software development, allowing different systems and software applications to communicate with each other. An API is a set of rules and protocols that allow one software application to interact with another. It defines the methods and data formats that applications can use to communicate with each other.

Example:

- **Web API**: Allows communication between a web server and a client (e.g., a web browser or mobile app).
- **Library/API**: Provides functions or classes to be used within a software application (e.g., the Python standard library).

2. API Architecture Styles

API architecture styles define how APIs are structured, designed, and how they communicate with other systems. Understanding these styles is crucial for choosing the right approach for your API development needs. Here's a detailed explanation of the most common API architecture styles:

1. REST (Representational State Transfer)

REST is an architectural style that uses HTTP protocols and URLs to access and manipulate resources. RESTful APIs are stateless and communicate through standard HTTP methods.

Key Characteristics:

- **Statelessness:** Each request from the client to the server must contain all the information needed to understand and process the request. The server does not store any client context between requests.
- **Resource-Based:** Everything is a resource, identified by a URI (Uniform Resource Identifier). For example, /users/123 might represent a user with the ID 123.
- **HTTP Methods:** Commonly uses GET (retrieve), POST (create), PUT (update), DELETE (remove) methods to perform CRUD operations.
- **Representation:** Resources can be represented in multiple formats, such as JSON, XML, or HTML. JSON is the most common format used today.
- **HATEOAS (Hypermedia as the Engine of Application State):** Clients interact with a RESTful API entirely through hypermedia provided dynamically by application servers.

When to Use:

- For web services that need to be consumed by a wide variety of clients (browsers, mobile apps, etc.).
- When you need to leverage HTTP caching and other mechanisms provided by the HTTP protocol.

2. SOAP (Simple Object Access Protocol)

SOAP is a protocol for exchanging structured information in web services using XML. It is more rigid and standardized than REST.

Key Characteristics:

- **Protocol-Based:** SOAP defines its own security and messaging framework, making it more complex.
- **XML-Based:** SOAP messages are XML-based, making them more verbose but also more extensible.
- **WS-Security:** SOAP has built-in security extensions like WS-Security for message integrity and confidentiality.
- **Transport-Independent:** While commonly used over HTTP/HTTPS, SOAP can also be transported over other protocols like SMTP.
- **Strict Contracts:** SOAP uses WSDL (Web Services Description Language) for describing the services offered by the web service.

When to Use:

- In enterprise environments where you need formal contracts and standardized protocols.
- For scenarios requiring complex transactions, security, and reliability.
- When interacting with legacy systems that already use SOAP.

3. GraphQL

GraphQL is a query language for APIs, allowing clients to request exactly the data they need. Developed by Facebook, it provides more flexibility compared to REST.

Key Characteristics:

- **Single Endpoint:** Unlike REST, which often has multiple endpoints, GraphQL typically has a single endpoint through which all queries are sent.
- **Client-Specified Queries:** Clients specify exactly what data they need, avoiding over-fetching or under-fetching of data.
- **Strongly Typed Schema:** The API defines a schema using GraphQL's type system. The schema is a contract between the client and server.
- **Real-Time Data:** Supports subscriptions to push real-time updates to clients.
- **Hierarchical:** Queries mimic the shape of the JSON response structure, making it intuitive to request related data.

When to Use:

- When clients need precise control over the data they request.
- For applications with complex data relationships where minimizing the number of API calls is crucial.

- When dealing with multiple data sources or microservices.

4. gRPC (Google Remote Procedure Call)

gRPC (Google Remote Procedure Call) is a high-performance RPC framework that uses Protocol Buffers (Protobuf) for serialization. It's developed by Google and is widely used for communication between microservices.

Key Characteristics:

- **Protocol Buffers:** Uses Protobuf, a binary format for serializing structured data, which is more efficient than JSON or XML.
- **HTTP/2:** gRPC uses HTTP/2 for transport, enabling features like multiplexing, streaming, and flow control.
- **Bidirectional Streaming:** Supports bi-directional streaming, allowing for real-time data exchange between clients and servers.
- **Code Generation:** Automatically generates client and server code in multiple languages from a .proto file.
- **Strongly Typed:** The service contracts are strictly defined in the .proto files, ensuring consistency.

When to Use:

- For microservices architecture where you need low latency and high throughput.
- In polyglot environments where different microservices are written in different programming languages.
- When streaming data or handling real-time communication is a priority.

5. RPC (Remote Procedure Call)

RPC is an older architecture style where a client executes a procedure (function) on a remote server. This is a more tightly coupled architecture compared to REST or GraphQL.

Key Characteristics:

- **Simplicity:** The client calls a function with parameters and waits for the result, similar to a local function call.
- **Tight Coupling:** Clients and servers are tightly coupled, meaning changes on the server can impact the client.
- **Synchronous:** Traditional RPCs are synchronous, though asynchronous versions exist.
- **Languages:** Typically implemented using XML-RPC, JSON-RPC, or other formats.

When to Use:

- In legacy systems or environments where RPC is already established.
- For simple, straightforward communication where the client and server are tightly controlled.
- When the overhead of REST or GraphQL is unnecessary.

3. Types of APIs

APIs (Application Programming Interfaces) come in various types based on their use cases, communication styles, and protocols. Here's a detailed overview of the different types of APIs:

1. Web APIs

Web APIs are the most common type of API, allowing applications to communicate over the web using HTTP/HTTPS protocols.

- **REST APIs:**
    - **Definition:** Representational State Transfer (REST) APIs follow a stateless architecture and use standard HTTP methods (GET, POST, PUT, DELETE) to interact with resources.
    - **Usage:** Commonly used in web and mobile applications for CRUD operations.
    - **Example:** A REST API for a blog might have endpoints like /posts, /posts/{id}, allowing clients to create, retrieve, update, and delete blog posts.
- **SOAP APIs:**
    - **Definition:** Simple Object Access Protocol (SOAP) is a protocol that uses XML for messaging and follows strict standards for communication.
    - **Usage:** Often used in enterprise environments for secure and reliable web services.
    - **Example:** A SOAP API might be used by a financial institution for processing secure transactions.
- **GraphQL APIs:**
    - **Definition:** GraphQL is a query language for APIs that allows clients to request specific data, reducing over-fetching and under-fetching of data.
    - **Usage:** Used when clients need precise control over the data they request, especially in applications with complex data relationships.
    - **Example:** A GraphQL API for a social media app might allow clients to request a user's profile data and their recent posts in a single query.
- **gRPC APIs:**
    - **Definition:** gRPC (Google Remote Procedure Call) is a high-performance, language-agnostic framework that uses Protocol Buffers (Protobuf) for serialization and HTTP/2 for transport.
    - **Usage:** Commonly used in microservices architecture for low-latency communication.
    - **Example:** A gRPC API might be used in a distributed system where services need to communicate efficiently across different programming languages.

2. Operating System APIs

Operating system APIs allow applications to interact with the underlying operating system's functions and resources.

- **Windows API (WinAPI):**
    - **Definition:** A set of APIs provided by Microsoft for interacting with Windows OS, including system services, file management, and user interface components.
    - **Usage:** Used by desktop applications to access OS-level features.
    - **Example:** An application might use the WinAPI to open files, read system information, or create windows and dialogs.
- **POSIX API:**
    - **Definition:** A standardized API for Unix-like operating systems, providing functions for file operations, process control, and threading.
    - **Usage:** Used by software that needs to run on multiple Unix-based systems, such as Linux and macOS.
    - **Example:** A POSIX API might be used by a server application to manage processes and handle file I/O operations.

3. Library APIs

Library APIs are sets of functions and procedures provided by software libraries that developers can use to perform specific tasks.

- **Standard Library APIs:**
    - **Definition:** APIs provided by programming language standard libraries, offering a range of functions for tasks like string manipulation, data handling, and networking.
    - **Usage:** Used in almost every software project to perform common operations.
    - **Example:** The Python Standard Library provides APIs for handling JSON data (json module), managing files (os module), and networking (socket module).
- **Third-Party Library APIs:**
    - **Definition:** APIs provided by third-party libraries that extend the functionality of a programming language or framework.
    - **Usage:** Used to add specific features to an application without reinventing the wheel.
    - **Example:** The requests library in Python provides an easy-to-use API for making HTTP requests.

4. Database APIs

Database APIs allow applications to interact with databases, enabling CRUD operations and complex queries.

- **SQL-Based APIs:**
  - **Definition:** APIs that use SQL (Structured Query Language) to interact with relational databases like MySQL, PostgreSQL, and SQL Server.
  - **Usage:** Used by applications to perform database operations such as querying, inserting, updating, and deleting data.
  - **Example:** An application might use a database API to execute a SQL query that retrieves user data from a users table.
- **NoSQL-Based APIs:**
  - **Definition:** APIs designed for interacting with NoSQL databases like MongoDB, Cassandra, and DynamoDB, which store data in formats like JSON, documents, or key-value pairs.
  - **Usage:** Used in applications that require scalable and flexible data storage.
  - **Example:** A NoSQL API might be used by an e-commerce application to store and retrieve product information in a document database like MongoDB.

5. Remote APIs

Remote APIs allow applications to interact with remote systems or services over a network.

- **Remote Procedure Call (RPC) APIs:**
  - **Definition:** RPC APIs allow a program to execute a procedure (function) on a remote server as if it were a local function.
  - **Usage:** Used in distributed systems where different components or services need to communicate over a network.
  - **Example:** A microservice might use an RPC API to call a function on another microservice to retrieve user data.
- **Cloud Service APIs:**
  - **Definition:** APIs provided by cloud service providers like AWS, Azure, and Google Cloud, enabling interaction with cloud resources.
  - **Usage:** Used by applications to manage cloud infrastructure, such as creating virtual machines, storing data, or deploying applications.
  - **Example:** An application might use the AWS S3 API to upload files to an S3 bucket.

6. Hardware APIs

Hardware APIs allow software to interact with hardware devices like sensors, cameras, and printers.

- **Device Driver APIs:**
    - **Definition:** APIs provided by device drivers that allow software to communicate with hardware components.
    - **Usage:** Used in embedded systems, operating systems, and desktop applications to control hardware devices.
    - **Example:** A printer driver API might be used by an application to send print jobs to a printer.
- **IoT APIs:**
    - **Definition:** APIs for interacting with Internet of Things (IoT) devices, allowing communication and data exchange between devices and applications.
    - **Usage:** Used in smart home systems, industrial automation, and wearable devices.
    - **Example:** An IoT API might be used by a smart thermostat to retrieve temperature data from a sensor and adjust the heating system.

7. Partner APIs

Partner APIs are designed for external use, allowing third-party developers to access certain parts of a system or service.

- **Public APIs:**
    - **Definition:** APIs that are openly available to external developers, allowing them to integrate with a company's services or data.
    - **Usage:** Used by companies to extend their services, create new applications, or foster innovation through third-party developers.
    - **Example:** The Twitter API allows developers to access and interact with Twitter's data, enabling them to create applications that tweet, search for tweets, and follow users.
- **Private APIs:**
    - **Definition:** APIs that are restricted to internal use within a company or organization.
    - **Usage:** Used to integrate internal systems, automate processes, or expose functionality to specific partners.
    - **Example:** A company might use a private API to allow its internal applications to communicate with its billing system.

4. API Components

When designing or interacting with APIs, several key components play crucial roles. Understanding these components in detail will help you create and manage your own APIs effectively.

1. Endpoints

- **Definition**: Endpoints are the URLs where APIs receive requests. Each endpoint corresponds to a specific resource or service.
- **Structure**: Typically, endpoints are structured to reflect the hierarchy of resources. For example:
  - /users: might retrieve a list of users.
  - /users/123: retrieves the user with ID 123.
- **Best Practices**: Use clear, resource-based paths, and follow conventions like using plurals for resource collections (e.g., /orders for a list of orders).

2. HTTP Methods

- **Definition**: HTTP methods define the type of action the API will perform on the resource.
- **Common Methods**:
  - **GET**: Retrieves data from the server. (e.g., GET /users)
  - **POST**: Creates a new resource on the server. (e.g., POST /users)
  - **PUT**: Updates an existing resource on the server. (e.g., PUT /users/123)
  - **DELETE**: Removes a resource from the server. (e.g., DELETE /users/123)
- **Best Practices**: Follow RESTful principles by using the appropriate HTTP method for each operation.

3. Headers

- **Definition**: Headers are key-value pairs sent with an API request or response that provide additional context or metadata.
- **Common Headers**:
  - **Authorization**: Used to pass authentication tokens (e.g., Bearer <token>).
  - **Content-Type**: Specifies the format of the data being sent (e.g., application/json).
  - **Accept**: Indicates the format in which the client expects the response (e.g., Accept: application/json).
- **Best Practices**: Ensure that headers are used consistently and securely, especially for sensitive information like authentication tokens.

4. Request Body

- **Definition**: The request body contains the data sent to the server when creating or updating a resource, typically used with POST, PUT, and PATCH methods.
- **Format**: The body is often formatted as JSON or XML. Example of a JSON request body:
- **Best Practices**: Validate and sanitize input data to prevent security vulnerabilities such as SQL injection or cross-site scripting (XSS).

5. Response

- **Definition**: The response is the data sent back from the server to the client after processing the request.
- **Components**:
  - **Status Code**: Indicates the result of the request (e.g., 200 OK, 404 Not Found, 500 Internal Server Error).
  - **Headers**: Provide metadata about the response.
  - **Body**: Contains the data requested or a message about the outcome.
- **Best Practices**: Ensure that responses are standardized, use appropriate status codes, and provide useful error messages.

6. Authentication and Authorization

- **Authentication**: Confirms the identity of the user or system making the request.
  - **Techniques**:
    - **API Keys**: Simple tokens provided to authorized users.
    - **OAuth2**: A more secure and standardized approach, often using JWTs (JSON Web Tokens).
  - **Best Practices**: Always use secure, encrypted connections (HTTPS) for transmitting tokens.
- **Authorization**: Determines what the authenticated user or system is allowed to do.
  - **Role-Based Access Control (RBAC)**: Assigns permissions based on user roles.
  - **Best Practices**: Implement fine-grained permissions and least privilege principles to limit access to sensitive resources.

7. Rate Limiting

- **Definition**: Controls the number of requests a client can make in a specific period.
- **Purpose**: Protects against abuse, ensures fair use, and prevents server overload.
- **Implementation**: Often enforced by API gateways, with responses that include headers like X-Rate-Limit-Limit, X-Rate-Limit-Remaining, and Retry-After.
- **Best Practices**: Clearly communicate rate limits to users and provide graceful error handling when limits are exceeded.

8. Caching

- **Definition**: Storing frequently requested data to reduce load on backend services and speed up response times.
- **Techniques**:
  - **HTTP Caching**: Using headers like Cache-Control, ETag, and Expires.
  - **Server-Side Caching**: Storing data in-memory (e.g., Redis) for quick retrieval.
- **Best Practices**: Use caching where appropriate to balance freshness and performance, and invalidate caches correctly when data changes.

9. Tools and Documentation

- **Purpose**: Provides developers with the necessary information to use the API effectively.
- **Tools**:
  - **Swagger/OpenAPI**: Automatically generate documentation from API definitions.
  - **Postman Collections**: Share example requests and responses.
- **Best Practices**: Ensure documentation is up-to-date, comprehensive, and includes code examples, usage scenarios, and error handling.

5. API Authentication and Authorization

Security is a critical aspect of APIs, ensuring that only authorized users can access certain data or perform actions.

a. API Keys

- **Definition**: A unique key provided to users for authenticating requests.
- **Use Case**: Simple authentication for public APIs.

b. OAuth

- **Definition**: An open standard for access delegation, commonly used for token-based authentication.
- **Flow**: Involves obtaining an access token through authorization, which is then used to authenticate API requests.
- **Use Case**: Used in more complex scenarios like third-party app integrations.

c. JWT (JSON Web Tokens)

- **Definition**: A compact, URL-safe means of representing claims to be transferred between two parties.
- **Structure**: Consists of a header, payload, and signature.
- **Use Case**: Often used in stateless authentication scenarios.

6. Rate Limiting and Throttling

To prevent abuse and ensure fair usage, APIs often implement rate limiting and throttling mechanisms.

a. Rate Limiting

- **Definition**: Restricts the number of API requests a user can make within a certain time frame.
- **Example**: A limit of 1000 requests per hour.

b. Throttling

- **Definition**: Regulates the speed at which requests are processed.
- **Use Case**: Ensures API stability and availability by slowing down excessive request rates.

7. API Versioning

APIs evolve over time, and versioning allows for changes without breaking existing client applications.

a. URL Versioning

- **Example**: https://api.example.com/v1/
- **Advantage**: Clear and explicit versioning method.

b. Header Versioning

- **Example**: Using a custom header like API-Version: v2.
- **Advantage**: Cleaner URLs, but requires clients to handle headers correctly.

c. Query Parameter Versioning

- **Example**: https://api.example.com/resource?version=2
- **Advantage**: Allows versioning within the same endpoint structure.

8. Error Handling in APIs

Proper error handling is crucial for API usability and debugging.

a. HTTP Status Codes

- **4xx Client Errors**: Errors due to invalid requests (e.g., 400 Bad Request, 401 Unauthorized).
- **5xx Server Errors**: Errors due to server issues (e.g., 500 Internal Server Error).

b. Error Messages

- **Definition**: Detailed messages explaining what went wrong.
- **Best Practice**: Should be descriptive and guide the developer on how to fix the issue.

c. Error Codes

- **Custom Error Codes**: Specific codes that represent particular errors within the API (e.g., ERR_1001: Invalid API Key).

9. API Documentation

Good documentation is key to API adoption and usability.

a. Swagger/OpenAPI

- **Definition**: A framework for designing and documenting REST APIs.
- **Features**: Allows for interactive documentation and testing of API endpoints.

b. API Reference Documentation

- **Content**: Includes endpoint descriptions, parameters, request/response examples, and error codes.
- **Best Practice**: Should be up-to-date, clear, and comprehensive.

10. API Testing

Testing ensures that APIs function correctly and securely.

a. Unit Testing

- **Definition**: Testing individual components of the API.
- **Example**: Testing a single endpoint to ensure it returns the expected response.

b. Integration Testing

- **Definition**: Testing how different parts of the API work together.
- **Example**: Ensuring that a sequence of API calls functions correctly.

c. Load Testing

- **Definition**: Testing how the API performs under high load or stress.
- **Tools**: Tools like Apache JMeter can be used for this purpose.

d. Security Testing

- **Definition**: Ensuring that the API is secure against common vulnerabilities (e.g., SQL injection, XSS, etc.).
- **Tools**: Tools like OWASP ZAP can be used for security testing.

11. API Lifecycle

APIs go through several stages during their lifecycle:

a. Design

- **Definition**: Planning and designing the API structure, endpoints, data formats, etc.
- **Tools**: Tools like Swagger and Postman can assist in the design phase.

b. Development

- **Definition**: Coding the API based on the design specifications.
- **Languages**: APIs can be developed in various programming languages (e.g., Python, Java, Node.js).

c. Testing

- **Definition**: Validating the API's functionality, performance, and security.
- **Automation**: Tools like Postman and Newman can be used to automate API testing.

d. Deployment

- **Definition**: Making the API available for use.
- **Environment**: APIs can be deployed on cloud platforms (e.g., AWS, Azure) or on-premises servers.

e. Monitoring

- **Definition**: Tracking API usage, performance, and uptime.
- **Tools**: Monitoring tools like Prometheus, Grafana, and API Gateway monitoring.

f. Versioning

- **Definition**: Releasing new versions of the API while maintaining backward compatibility.

g. Deprecation

- **Definition**: Phasing out older versions of the API.
- **Process**: Usually involves notifying users, providing migration paths, and eventually retiring the old version.

12. API Gateways

An API Gateway serves as an intermediary between clients and backend services, providing a centralized entry point for API requests. It offers several critical functions:

a. Routing

- **Definition**: Directs incoming API requests to the appropriate backend services.
- **Example**: A gateway can route a /user request to the user service and a /order request to the order service.

b. Authentication and Authorization

- **Definition**: Manages user authentication and access control for API requests.
- **Implementation**: The gateway can validate tokens (like JWTs) and enforce role-based access control (RBAC).

c. Rate Limiting

- **Definition**: Controls the number of requests a client can make in a specific time frame.
- **Purpose**: Helps prevent abuse and ensure fair usage of resources.

d. Load Balancing

- **Definition**: Distributes incoming requests across multiple servers or instances to ensure optimal performance.
- **Purpose**: Enhances scalability and reliability by preventing any single server from becoming overwhelmed.

e. Caching

- **Definition**: Stores frequently requested data to reduce the load on backend services and improve response times.
- **Example**: Cached responses for static content like product lists or user profiles.

f. Logging and Monitoring

- **Definition**: Collects and analyzes logs and metrics for API usage, errors, and performance.
- **Tools**: Integrated with monitoring tools like Grafana, Prometheus, or ELK stack (Elasticsearch, Logstash, Kibana).

g. Security

- **Definition**: Implements security measures like SSL/TLS termination, DDoS protection, and IP whitelisting.
- **Purpose**: Enhances the security of API communications by ensuring that all data is transmitted securely.

13. Microservices and APIs

APIs play a crucial role in microservices architecture, where applications are composed of small, loosely coupled services.

a. Service Communication

- **Definition**: Microservices communicate with each other using APIs.
- **Protocols**: Common protocols include REST, gRPC, and AMQP (Advanced Message Queuing Protocol).

b. Service Discovery

- **Definition**: Dynamically locating microservices in a distributed system.
- **Implementation**: Using tools like Consul or Eureka for service discovery, which integrates with API gateways.

c. Inter-Service Authentication

- **Definition**: Ensuring that microservices can authenticate and authorize requests from other services.
- **Example**: Using OAuth tokens or mutual TLS (mTLS) for secure communication.

14. REST vs. RPC vs. GraphQL

Different API design paradigms suit different use cases, with REST, RPC (Remote Procedure Call), and GraphQL being the most common.

a. REST

- **Focus**: Resources, with each resource identified by a unique URI.
- **Use Case**: Ideal for CRUD operations and web services where the standard HTTP methods align with the application's needs.

b. RPC (Remote Procedure Call)

- **Focus**: Function calls that execute on a remote server.
- **Example**: gRPC, a high-performance RPC framework developed by Google.
- **Use Case**: Useful in scenarios requiring low latency and high throughput, such as inter-microservice communication.

c. GraphQL

- **Focus**: Flexible queries that allow clients to request exactly what they need.
- **Example**: A single GraphQL endpoint can handle multiple queries and mutations, reducing the need for multiple API calls.
- **Use Case**: Ideal for applications requiring efficient data fetching, like mobile apps where network bandwidth is a concern.

15. API Security Best Practices

Ensuring the security of APIs is paramount to protecting sensitive data and maintaining trust.

a. Authentication and Authorization

- **OAuth2**: A widely adopted framework for access delegation.
- **JWT**: JSON Web Tokens are often used for stateless authentication.
- **Best Practice**: Use strong encryption for token generation and storage.

b. Input Validation

- **Purpose**: Prevents injection attacks by validating and sanitizing all user inputs.
- **Techniques**: Use libraries and frameworks that provide built-in input validation mechanisms.

c. Rate Limiting and Throttling

- **Purpose**: Protects against DDoS attacks and abuse.
- **Best Practice**: Implement rate limiting at the API gateway level.

d. Encryption

- **Transport Layer Security (TLS)**: Encrypts data in transit between clients and servers.
- **Best Practice**: Always use HTTPS to ensure data is encrypted during transmission.

e. Logging and Monitoring

- **Purpose**: Detects and responds to security incidents in real-time.
- **Best Practice**: Implement centralized logging and monitor for unusual patterns or anomalies.

f. API Security Testing

- **Tools**: Use tools like OWASP ZAP or Burp Suite to test APIs for common vulnerabilities.
- **Best Practice**: Regularly perform penetration testing and security audits.

16. API Management

API management involves overseeing the entire lifecycle of an API, from creation to retirement.

a. API Gateway

- **Role**: Acts as a centralized entry point for managing API traffic, enforcing policies, and monitoring usage.
- **Features**: Rate limiting, analytics, security, and protocol translation.

b. Developer Portal

- **Purpose**: Provides a platform for developers to explore, test, and consume APIs.
- **Features**: Documentation, sandbox environments, and API keys management.

c. Analytics

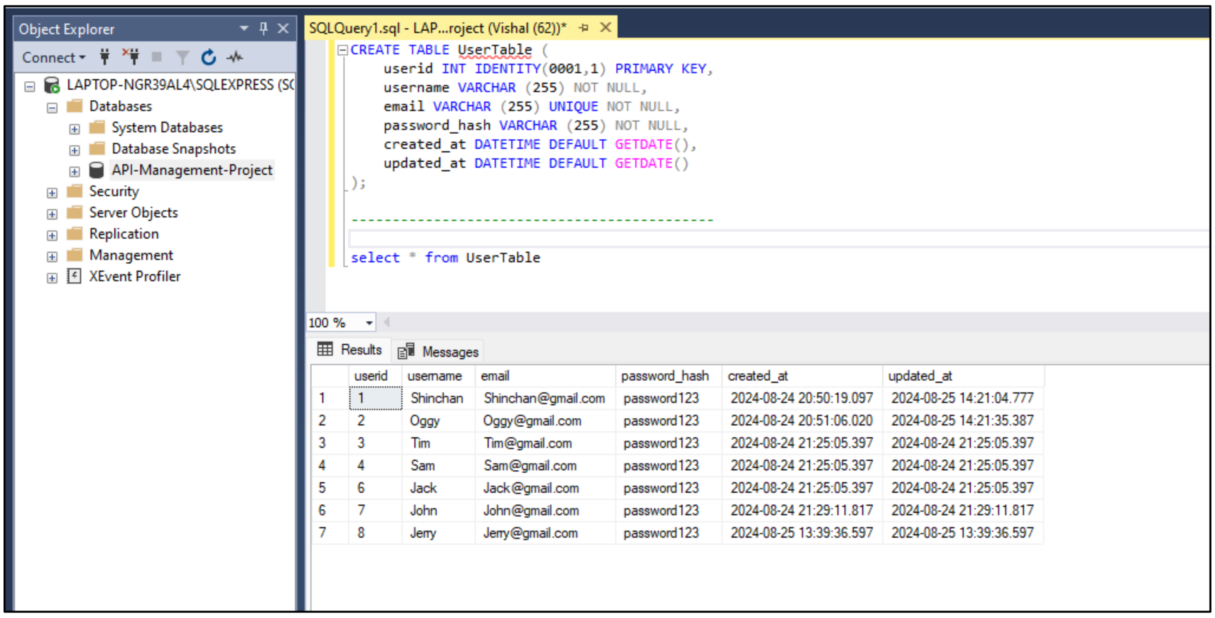
- **Purpose**: Tracks API usage, performance, and errors.
- **Best Practice**: Use analytics to optimize API performance and improve user experience.

d. Policy Enforcement

- **Purpose**: Ensures compliance with organizational policies and standards.
- **Examples**: Enforcing authentication, IP whitelisting, and rate limits.

17. Project: User Management API

1. Create Table in Database

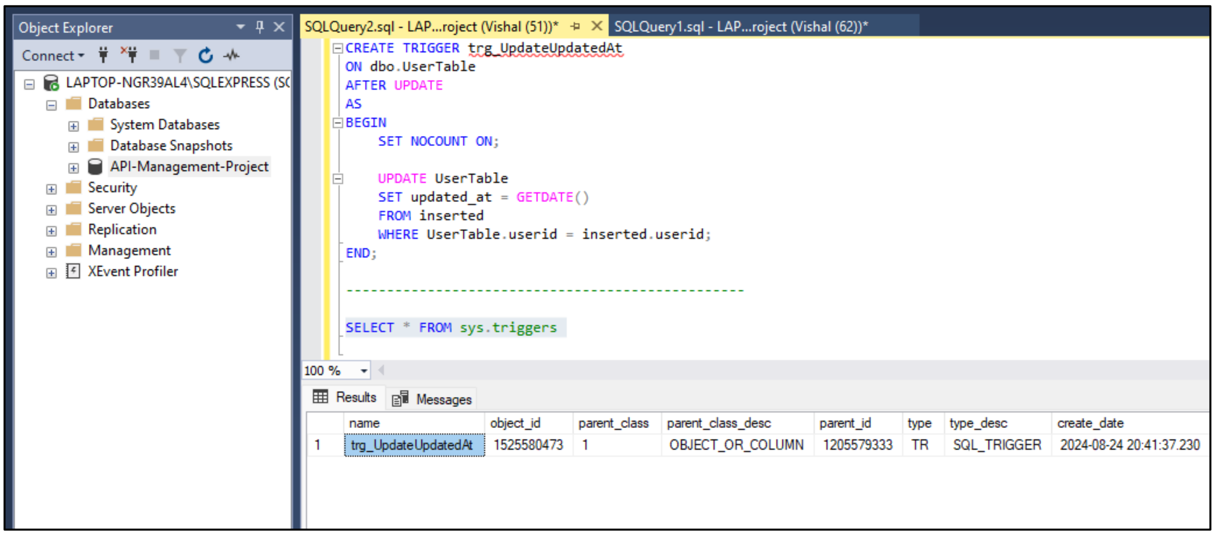


```
CREATE TABLE UserTable (
    userid INT IDENTITY(0001,1) PRIMARY KEY,
    username VARCHAR (255) NOT NULL,
    email VARCHAR (255) UNIQUE NOT NULL,
    password_hash VARCHAR (255) NOT NULL,
    created_at DATETIME DEFAULT GETDATE(),
    updated_at DATETIME DEFAULT GETDATE()
);

--------------------------------------------

select * from UserTable
```

	userid	username	email	password_hash	created_at	updated_at
1	1	Shinchan	Shinchan@gmail.com	password123	2024-08-24 20:50:19.097	2024-08-25 14:21:04.777
2	2	Oggy	Oggy@gmail.com	password123	2024-08-24 20:51:06.020	2024-08-25 14:21:35.387
3	3	Tim	Tim@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
4	4	Sam	Sam@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
5	6	Jack	Jack@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
6	7	John	John@gmail.com	password123	2024-08-24 21:29:11.817	2024-08-24 21:29:11.817
7	8	Jerry	Jerry@gmail.com	password123	2024-08-25 13:39:36.597	2024-08-25 13:39:36.597

2. Create Trigger

Whenever an entity is updated for table “UserTable” the column[updated_at] will be update accordingly.

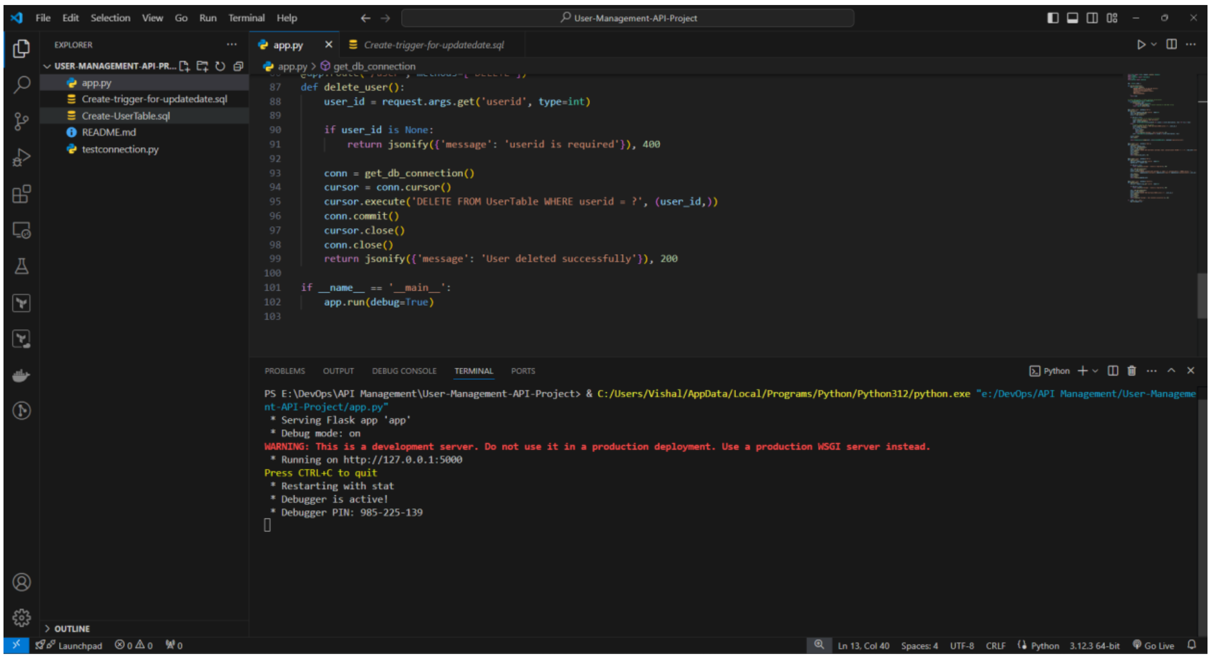


```
CREATE TRIGGER trg_UpdateUpdatedAt
ON dbo.UserTable
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE UserTable
    SET updated_at = GETDATE()
    FROM inserted
    WHERE UserTable.userid = inserted.userid;
END;

-------------------------------------------------

SELECT * FROM sys.triggers
```

	name	object_id	parent_class	parent_class_desc	parent_id	type	type_desc	create_date
1	trg_UpdateUpdatedAt	1525580473	1	OBJECT_OR_COLUMN	1205579333	TR	SQL_TRIGGER	2024-08-24 20:41:37.230

3. Setup the Development Environment

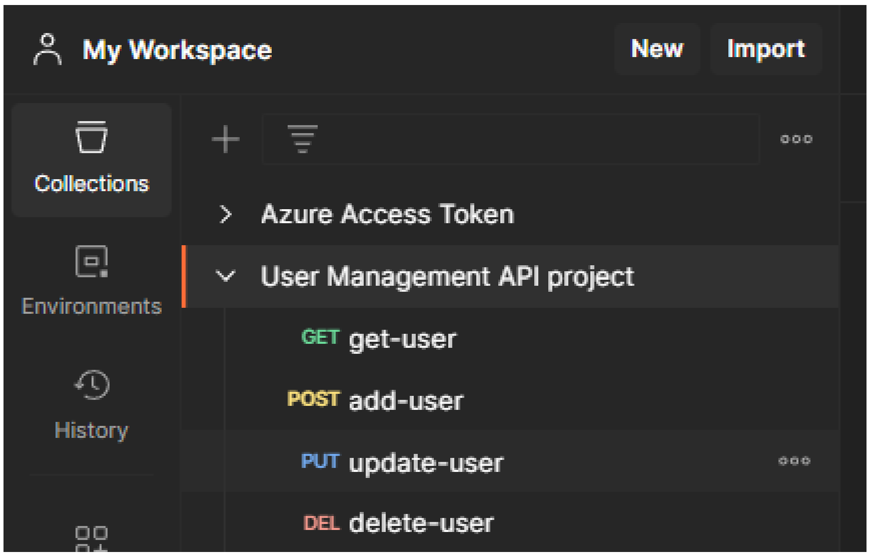
Refer the GitHub Repo: https://github.com/VishalKurane/User-Management-API-Project



```
def delete_user():
    user_id = request.args.get('userid', type=int)

    if user_id is None:
        return jsonify({'message': 'userid is required'}), 400

    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute('DELETE FROM UserTable WHERE userid = ?', (user_id,))
    conn.commit()
    cursor.close()
    conn.close()
    return jsonify({'message': 'User deleted successfully'}), 200

if __name__ == '__main__':
    app.run(debug=True)
```

```
PS E:\DevOps\API Management\User-Management-API-Project> & C:/Users/Vishal/AppData/Local/Programs/Python/Python312/python.exe "e:/DevOps/API Management/User-Manageme
nt-API-Project/app.py"
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 985-225-139
```

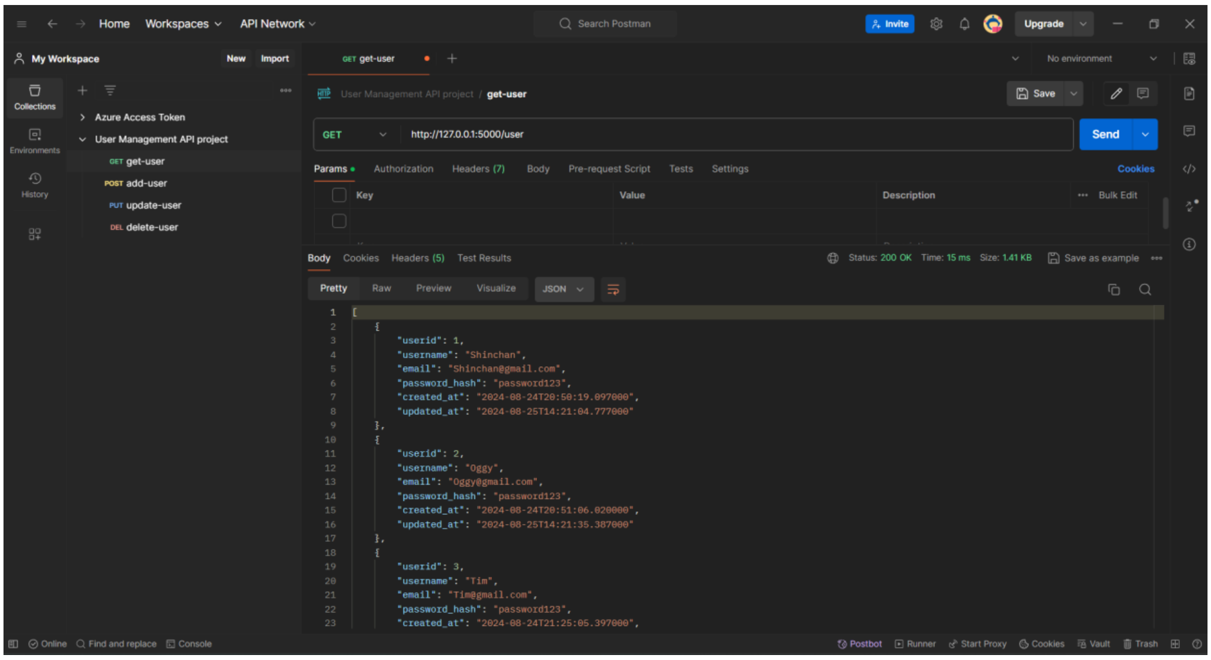
4. Test all the APIs



1. GET: /user

Query Parameters: ‘userid’ (optional)

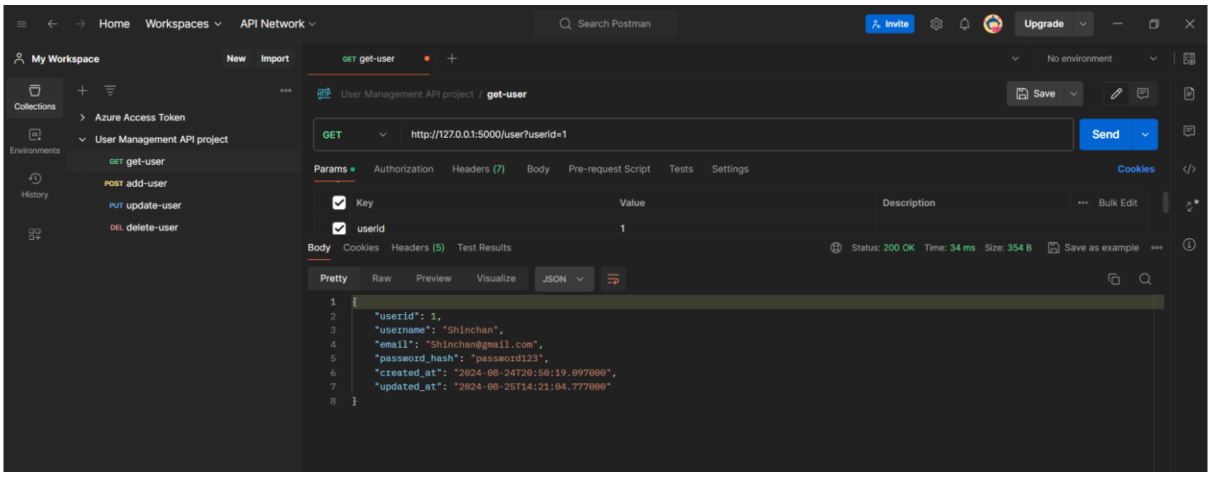
Response: Returns a list of all users or a single user if `userid` is provided.



GET http://127.0.0.1:5000/user

Status: 200 OK Time: 15 ms Size: 1.41 KB

```
[
    {
        "userid": 1,
        "username": "Shinchan",
        "email": "Shinchan@gmail.com",
        "password_hash": "password123",
        "created_at": "2024-08-24T20:50:19.097000",
        "updated_at": "2024-08-25T14:21:04.777000"
    },
    {
        "userid": 2,
        "username": "Oggy",
        "email": "Oggy@gmail.com",
        "password_hash": "password123",
        "created_at": "2024-08-24T20:51:06.020000",
        "updated_at": "2024-08-25T14:21:35.387000"
    },
    {
        "userid": 3,
        "username": "Tim",
        "email": "Tim@gmail.com",
        "password_hash": "password123",
        "created_at": "2024-08-24T21:25:05.397000",
```



GET http://127.0.0.1:5000/user?userid=1

Key	Value
userid	1

Status: 200 OK Time: 34 ms Size: 354 B

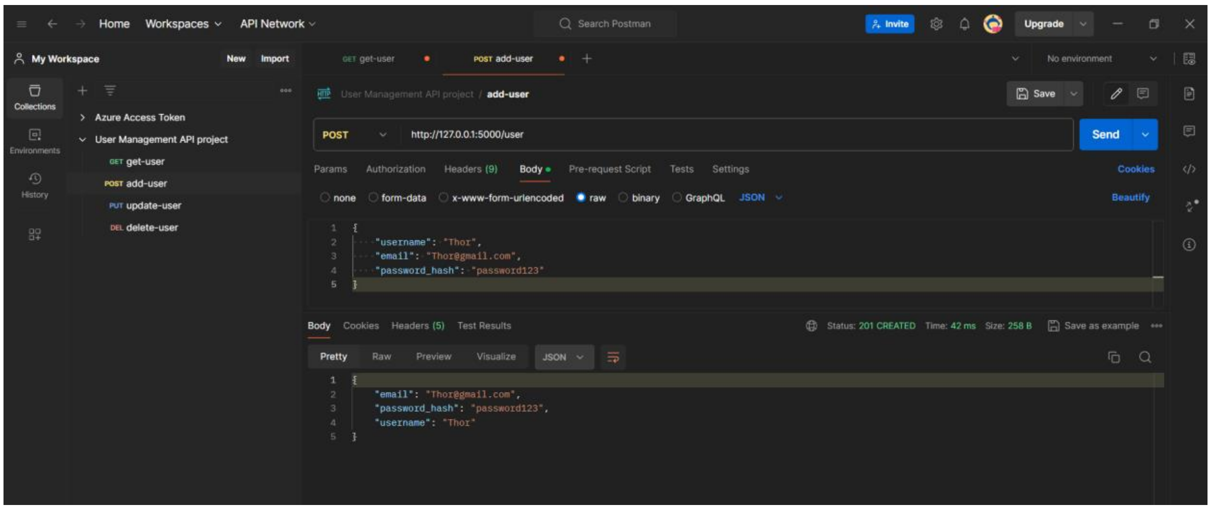
```
{
    "userid": 1,
    "username": "Shinchan",
    "email": "Shinchan@gmail.com",
    "password_hash": "password123",
    "created_at": "2024-08-24T20:50:19.097000",
    "updated_at": "2024-08-25T14:21:04.777000"
}
```

2. POST: /user

Body: JSON object with the user details:

```
{
   "username": "Thor",
   "email": "Thor@gmail.com",
   "password_hash": "password123"
}
```

Response: Returns the created user object

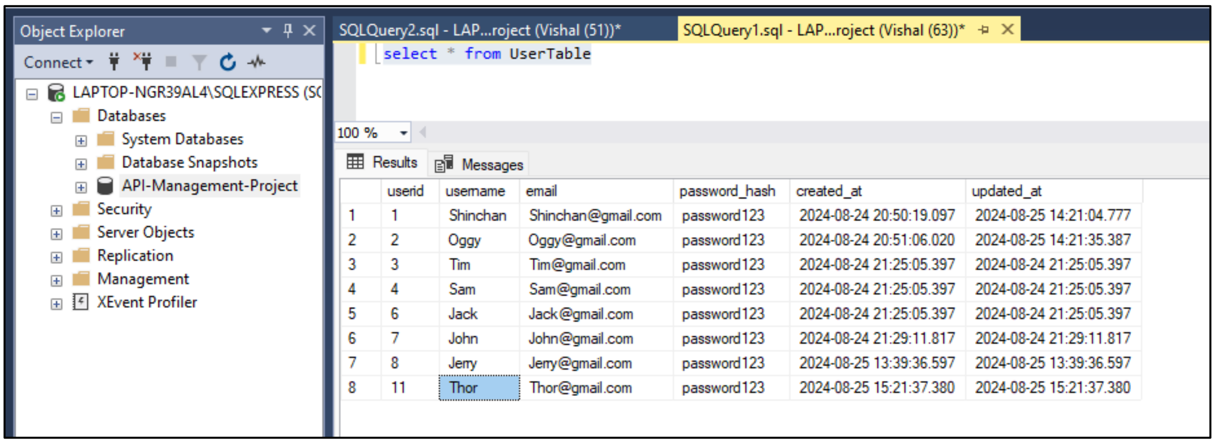


POST http://127.0.0.1:5000/user

```
{
····"username": "Thor",
····"email": "Thor@gmail.com",
····"password_hash": "password123"
}
```

Status: 201 CREATED Time: 42 ms Size: 258 B

```
{
    "email": "Thor@gmail.com",
    "password_hash": "password123",
    "username": "Thor"
}
```



```
select * from UserTable
```

	userid	usemame	email	password_hash	created_at	updated_at
1	1	Shinchan	Shinchan@gmail.com	password123	2024-08-24 20:50:19.097	2024-08-25 14:21:04.777
2	2	Oggy	Oggy@gmail.com	password123	2024-08-24 20:51:06.020	2024-08-25 14:21:35.387
3	3	Tim	Tim@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
4	4	Sam	Sam@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
5	6	Jack	Jack@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
6	7	John	John@gmail.com	password123	2024-08-24 21:29:11.817	2024-08-24 21:29:11.817
7	8	Jerry	Jerry@gmail.com	password123	2024-08-25 13:39:36.597	2024-08-25 13:39:36.597
8	11	Thor	Thor@gmail.com	password123	2024-08-25 15:21:37.380	2024-08-25 15:21:37.380

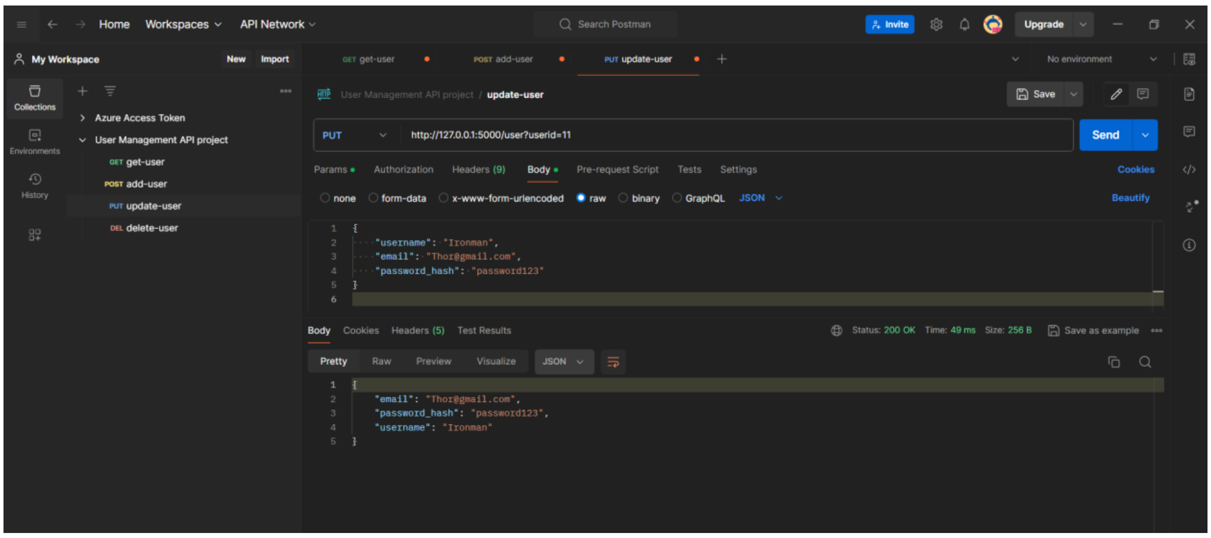
3. PUT: /user

Query Parameters: ‘userid’ (required)

Body: JSON object with the user details:

{
"username": "Ironman",
"email": "Thor@gmail.com",
"password_hash": "password123"
}

Response: Returns the updated user object



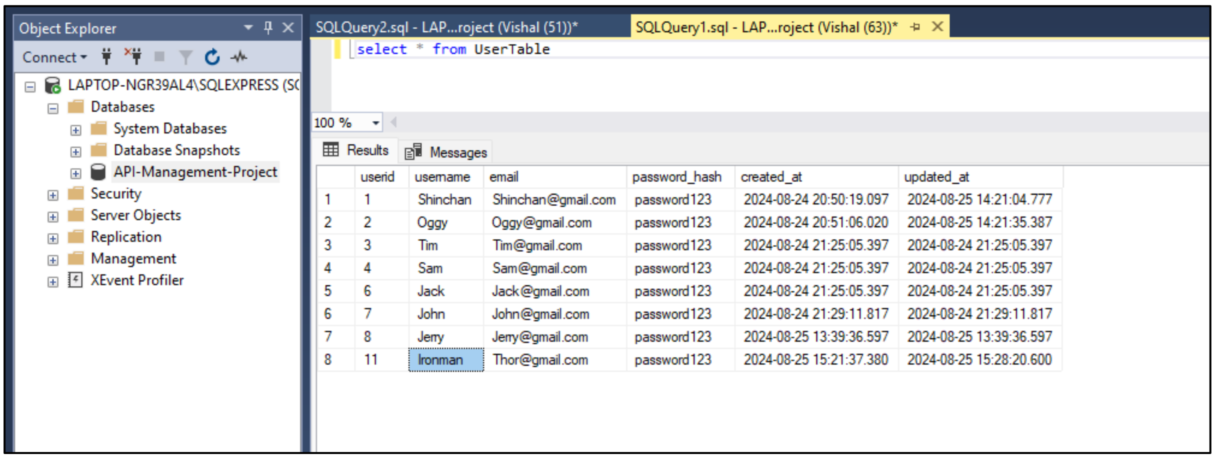
PUT http://127.0.0.1:5000/user?userid=11

Body: none / form-data / x-www-form-urlencoded / [✓ raw] / binary / GraphQL — JSON

```
{
    "username": "Ironman",
    "email": "Thor@gmail.com",
    "password_hash": "password123"
}
```

Status: 200 OK Time: 49 ms Size: 256 B

```
{
    "email": "Thor@gmail.com",
    "password_hash": "password123",
    "username": "Ironman"
}
```



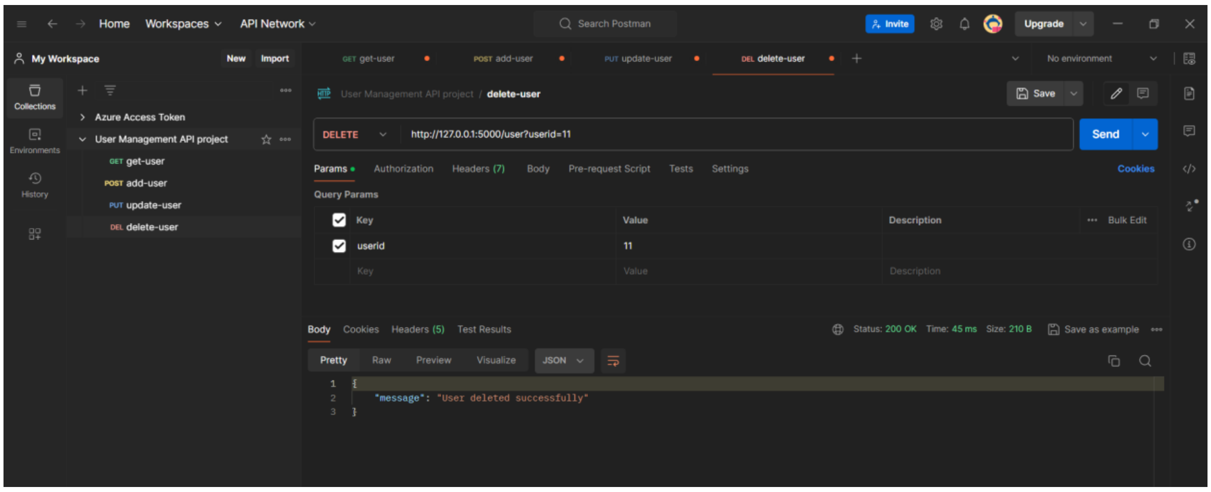
```
select * from UserTable
```

	userid	username	email	password_hash	created_at	updated_at
1	1	Shinchan	Shinchan@gmail.com	password123	2024-08-24 20:50:19.097	2024-08-25 14:21:04.777
2	2	Oggy	Oggy@gmail.com	password123	2024-08-24 20:51:06.020	2024-08-25 14:21:35.387
3	3	Tim	Tim@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
4	4	Sam	Sam@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
5	6	Jack	Jack@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
6	7	John	John@gmail.com	password123	2024-08-24 21:29:11.817	2024-08-24 21:29:11.817
7	8	Jerry	Jerry@gmail.com	password123	2024-08-25 13:39:36.597	2024-08-25 13:39:36.597
8	11	Ironman	Thor@gmail.com	password123	2024-08-25 15:21:37.380	2024-08-25 15:28:20.600

4. DELETE: /user

Query Parameters: ‘userid’ (required)

Response: Returns a success message if the user was deleted.



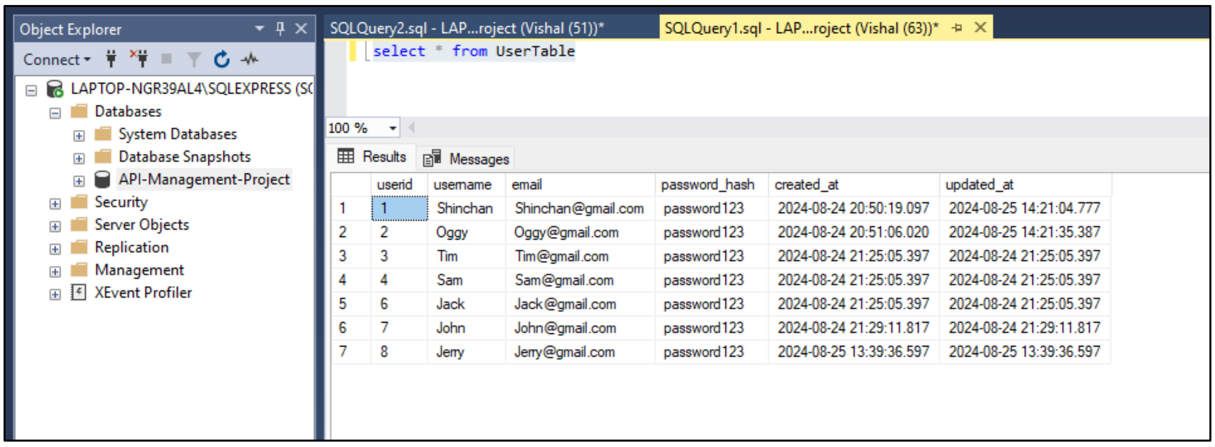
DELETE http://127.0.0.1:5000/user?userid=11

Query Params

Key	Value	Description
userid	11	

Status: 200 OK Time: 45 ms Size: 210 B

```
{
    "message": "User deleted successfully"
}
```



```
select * from UserTable
```

	userid	username	email	password_hash	created_at	updated_at
1	1	Shinchan	Shinchan@gmail.com	password123	2024-08-24 20:50:19.097	2024-08-25 14:21:04.777
2	2	Oggy	Oggy@gmail.com	password123	2024-08-24 20:51:06.020	2024-08-25 14:21:35.387
3	3	Tim	Tim@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
4	4	Sam	Sam@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
5	6	Jack	Jack@gmail.com	password123	2024-08-24 21:25:05.397	2024-08-24 21:25:05.397
6	7	John	John@gmail.com	password123	2024-08-24 21:29:11.817	2024-08-24 21:29:11.817
7	8	Jerry	Jerry@gmail.com	password123	2024-08-25 13:39:36.597	2024-08-25 13:39:36.597