# DevOps
# AIDE501 – M2 – 3 Credits
# 2026/2027

# Course Contents

- **Chapter 1: Cloud Computing**
- **Chapter 2: Containerization**
- **Chapter 3: Shell Scripting**
- **Chapter 4: DevOps**
- **Chapter 5: Orchestration**
- **Chapter 6: GitLab CI/CD**

# Chapter 2:

# Containerization

# Chap.2: Containerization

- Containerization basics (Docker, Podman)
- Registry, Docker Compose
- Managing and Securing Registries
- Identity and Access Management (IAM)
- Authentication, Authorization, API

# Managing and Securing Registries

# Container Registry

- A Container Registry is a service used to store, organize, manage, and distribute container images.
- When working with Docker, registries are important because they allow developers to share images and deploy applications on different machines.
- However, registries must be properly managed and secured to prevent unauthorized access, image modification, and the distribution of vulnerable images.

# Registry management

- Registry management involves organizing, controlling, maintaining, and monitoring container images.

**Creating repositories**

A repository is a collection of related container images, usually identified by a name and different tags.

For example, in Docker Hub:

```
myusername/
    webapp/
        v1
        v2
        latest
    database/
        v1
        v2
```

Each repository can contain multiple versions of an image.

# Registry management

**Uploading images to a registry**

To upload an image to Docker Hub:

Step 1: Login

```
docker login
```

Step 2: Build an image

```
docker build -t myapp .
```

Step 3: Tag the image

```
docker tag myapp myusername/myapp:v1
```

Step 4: Push the image

```
docker push myusername/myapp:v1
```

The image is now stored in Docker Hub and can be downloaded by other users who have access to it.

# Registry management

## Downloading images

To download an image:

```
docker pull myusername/myapp:v1
```

To run it:

```
docker run myusername/myapp:v1
```

## Managing image versions

Image tags help identify different versions of an application.

For example:

`myapp:v1.0`

`myapp:v1.1`

`myapp:v2.0`

It is good practice to use meaningful version tags rather than relying only on `latest`.

# Registry management

**Removing images**

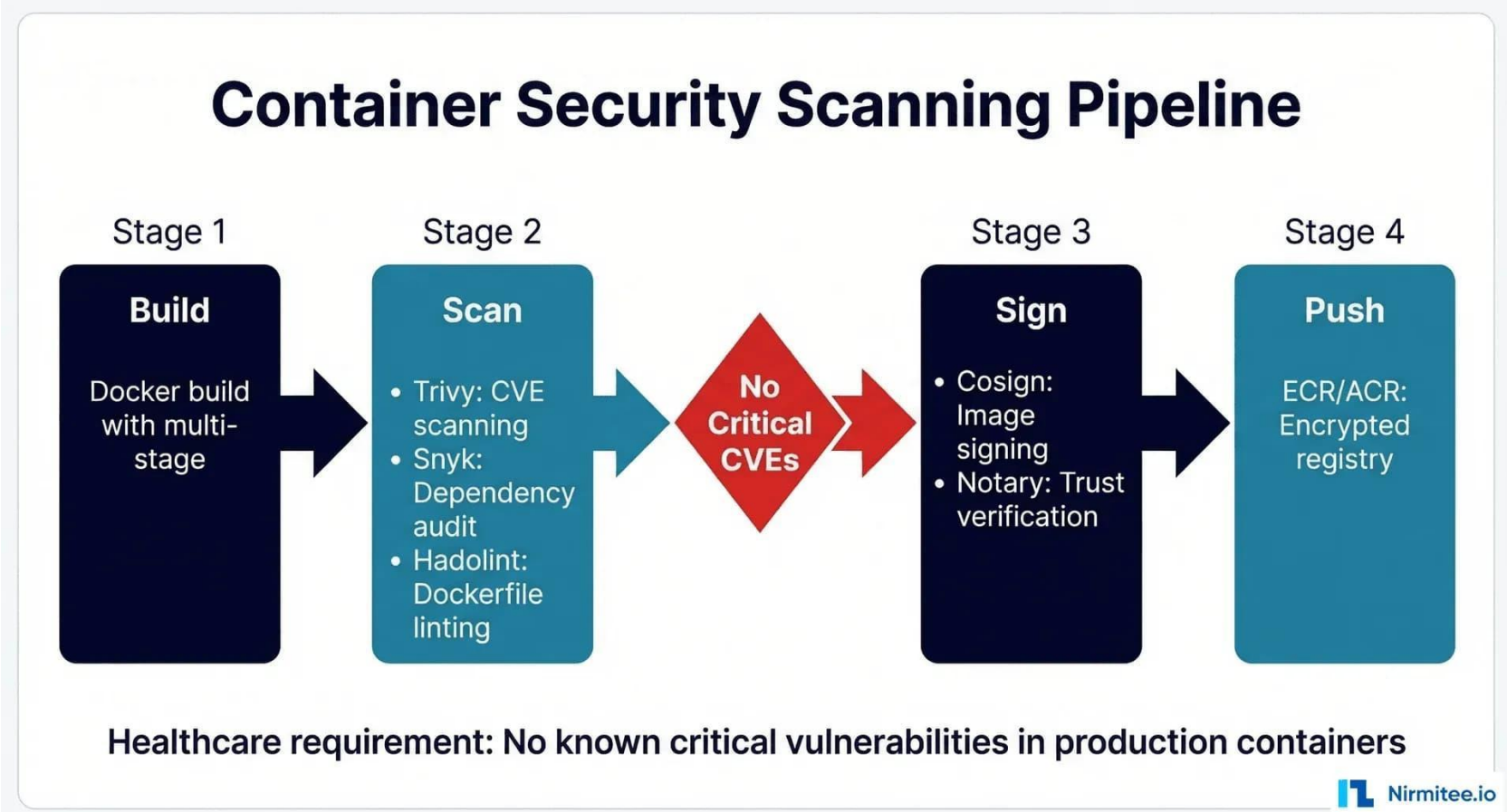
To remove an image from the local Docker environment:

```
docker rmi myapp:v1
```

To remove images from a remote registry, use the registry's management interface or API, according to its capabilities and permissions.

# Securing Container Registries

- Registry security protects container images against unauthorized access, modification, theft, and malicious software.



# Securing Container Registries

There are several important security mechanisms.

**A. Authentication**

Authentication verifies the identity of a user before allowing access to a registry.

For example:

```
docker login
```

Docker asks for credentials or uses an appropriate authentication mechanism.

Other authentication methods include:

- Username and password.
- Personal Access Tokens (PAT).
- OAuth authentication.
- Cloud provider identity mechanisms.

# Securing Container Registries

## B. Authorization and Access Control

Authentication answers:

`Who are you?`

Authorization answers:

`What are you allowed to do?`

For example, a registry can define different permissions:

| User | Permissions |
|---|---|
| Administrator | Full control |
| Developer | Push and pull images |
| Student | Pull images only |
| Guest | No access to private repositories |

This is often implemented using Role-Based Access Control (RBAC).

# Securing Container Registries

**C. Secure communication using HTTPS/TLS**

Communication between Docker clients and registries should be encrypted using HTTPS and TLS.

For example:

```
docker login https://registry.example.com
```

TLS helps protect:

- Login credentials.
- Authentication tokens.
- Container image transfers.
- Communication between clients and registries.

Without encryption, sensitive information could potentially be intercepted.

# Securing Container Registries

**D. Image vulnerability scanning**

Container images may contain vulnerable operating system packages, outdated libraries, or insecure dependencies.

Image scanning identifies known security vulnerabilities.

Popular tools include:

- Trivy.
- Clair.
- Docker Scout.

Example using Trivy:

```
trivy image nginx:latest
```

Trivy analyzes the image and reports known vulnerabilities.

# Securing Container Registries

**E. Image signing and verification**

Image signing helps verify the authenticity and integrity of container images.

The objective is to ensure that an image comes from a trusted source and has not been modified unexpectedly.

One tool used for this purpose is Cosign.

Example:

```
cosign sign --key cosign.key \
  myusername/myapp:v1
```

To verify the image:

```
cosign verify --key cosign.pub \
  myusername/myapp:v1
```

The verification checks the signature against the trusted public key.

# Securing Container Registries

**F. Managing access to private registries**

Not all container images should be publicly accessible.

Private registries restrict access to authorized users and applications.

For example, to log in to a private registry:

```
docker login registry.example.com
```

Then upload an image:

```
docker tag myapp registry.example.com/myapp:v1
```

```
docker push registry.example.com/myapp:v1
```

Only users with the appropriate permissions can access the private repository.

# Securing Container Registries

**G. Registry maintenance and cleanup**

Registry administrators should regularly maintain repositories.

Important tasks include:

- Removing unused or obsolete images.
- Deleting vulnerable image versions when appropriate.
- Keeping only necessary image versions.
- Monitoring registry storage usage.
- Reviewing user access permissions.
- Backing up important images and registry data.
- Rotating credentials and access tokens.

# Identity and Access Management (IAM)

# Identity and Access Management (IAM)

- IAM is the cornerstone of cloud security and effectively replaces the traditional network perimeter.
- In cloud environments, **identity becomes the new security boundary**, governing access to all resources through centrally managed policies.

# Fundamental Concepts

- **Identity**: A uniquely identifiable entity that can authenticate and be authorized (human users, applications, virtual machines, containers, serverless functions).
- **Authentication (AuthN)**: The process of verifying an identity (passwords, certificates, tokens, MFA).
- **Authorization (AuthZ)**: The process of determining what an authenticated identity is allowed to do.
- **Accounting/Auditing**: Recording actions for traceability and compliance.

# Types of Identities in the Cloud

- **Human Identities**:
- Administrators
- Developers
- End users

- **Non-Human (Machine) Identities**:
- Virtual machines and instances
- Containers and Kubernetes service accounts
- Serverless functions
- CI/CD pipelines and automation tools

- Modern cloud breaches often involve **compromised machine identities**, not human users.

# Access Control Models

- **Role-Based Access Control (RBAC)**:
- Permissions grouped into roles
- Roles assigned to identities
- Widely used in Kubernetes and cloud IAM

- **Attribute-Based Access Control (ABAC)**:
- Decisions based on attributes (user, resource, environment)
- More flexible but complex

- **Policy-Based Access Control**:
- JSON/YAML policies evaluated by an authorization engine
- Explicit allow and implicit deny logic

# IAM Threats and Attack Vectors

- Credential leakage
- Privilege escalation
- Lateral movement
- Token theft

# IAM Best Practices

- Enforce MFA everywhere
- Use roles instead of static credentials
- Apply least privilege by default
- Monitor and audit IAM activity
- Automate identity lifecycle management

# Authentication

# Key Authentication Methods

- **1. Username & Password**:

The most basic method, where users log in with credentials.

- **2. Multi-Factor Authentication (MFA)**:

Adds additional verification such as: SMS code, Email OTP (One-Time Password), Authenticator app (Google Authenticator), Biometrics. It improves security, especially in multi-tenant SaaS.

- **3. Single Sign-On (SSO)**:

Users log in once and access multiple systems. It uses common SSO protocols: SAML 2.0, OAuth 2.0, OpenID Connect. It is used in enterprise SaaS (e.g., Microsoft 365, Salesforce).

# Key Authentication Methods in SaaS

- **4. Token-Based Authentication**:

Instead of sending username/password each time, the system issues a token. Types of tokens: JWT (JSON Web Token), Access tokens, Refresh tokens. It is secure, stateless, and scalable for multi-tenant systems.

- **5. OAuth 2.0 (**Open Authorization**) Authentication**:
- Used for granting limited access to user accounts: "Sign in with Google", "Sign in with Facebook". Common in public SaaS applications.

- **6. Tenant-Aware Authentication**:

Subdomains: tenant1.app.com, Tenant IDs: Separate login pages, Tenant-specific identity providers (IdP)

# Authentication Challenges

- **Scalability**

The authentication system must handle thousands or millions of logins.

- **Security**

The authentication system must Prevent:

- Brute force
- Session hijacking
- Weak passwords
- Token theft

# Authentication and Authorization

- **Authentication** is the process of verifying who a user/application is, while **Authorization** determines what that authenticated user/application is allowed to do (Which data a user can access, Which features they can use, What actions they can perform).

- The most common method for modern SaaS integration is using **Bearer Tokens** (usually derived from OAuth 2.0 or simple API keys).

# Authentication flow with Bearer tokens

| Component | Description | Role in Security |
| --- | --- | --- |
| **Client Sends Request** | The application initiates communication. | The request must include proof of identity and permission. |
| **Authorization Header** | A standard HTTP header used to carry the security token. | Format: `Authorization: Bearer <TOKEN_STRING>`. This token is the digital key. |
| **Bearer Token** (e.g., JWT, API Key) | A secret string issued by the SaaS provider, granting access. | It proves the client is authenticated and is authorized to access the requested resource. The token is checked on **every request**. |
| **SaaS Authentication Service** | The SaaS server component that verifies the token's validity and permissions. | If the token is invalid or expired, the server returns an **HTTP 401 Unauthorized** error. |
| **Access Granted/Denied** | The decision to fulfill the request. | If the token is valid, the request proceeds to the API logic. If the token is valid but lacks permissions (e.g., trying to delete a user without admin rights), the server returns **HTTP 403 Forbidden**. |

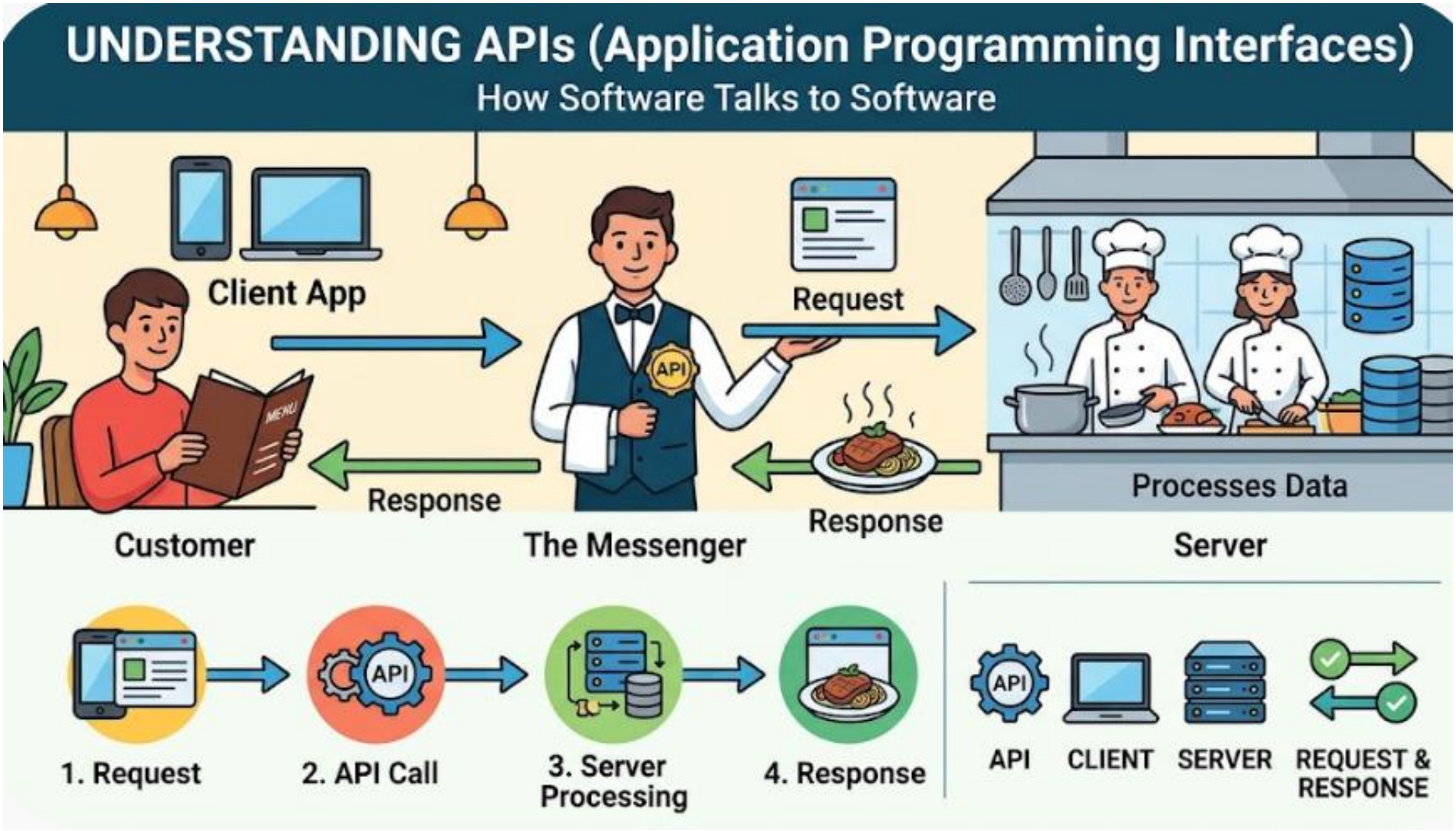
# Example of OAuth 2.0 Flow

1. The user logs in with Google (OAuth provider).
2. Google asks the user to grant permission to your app.
3. If approved, Google sends back an **access token**.
4. Your app uses that token to access Google APIs securely.

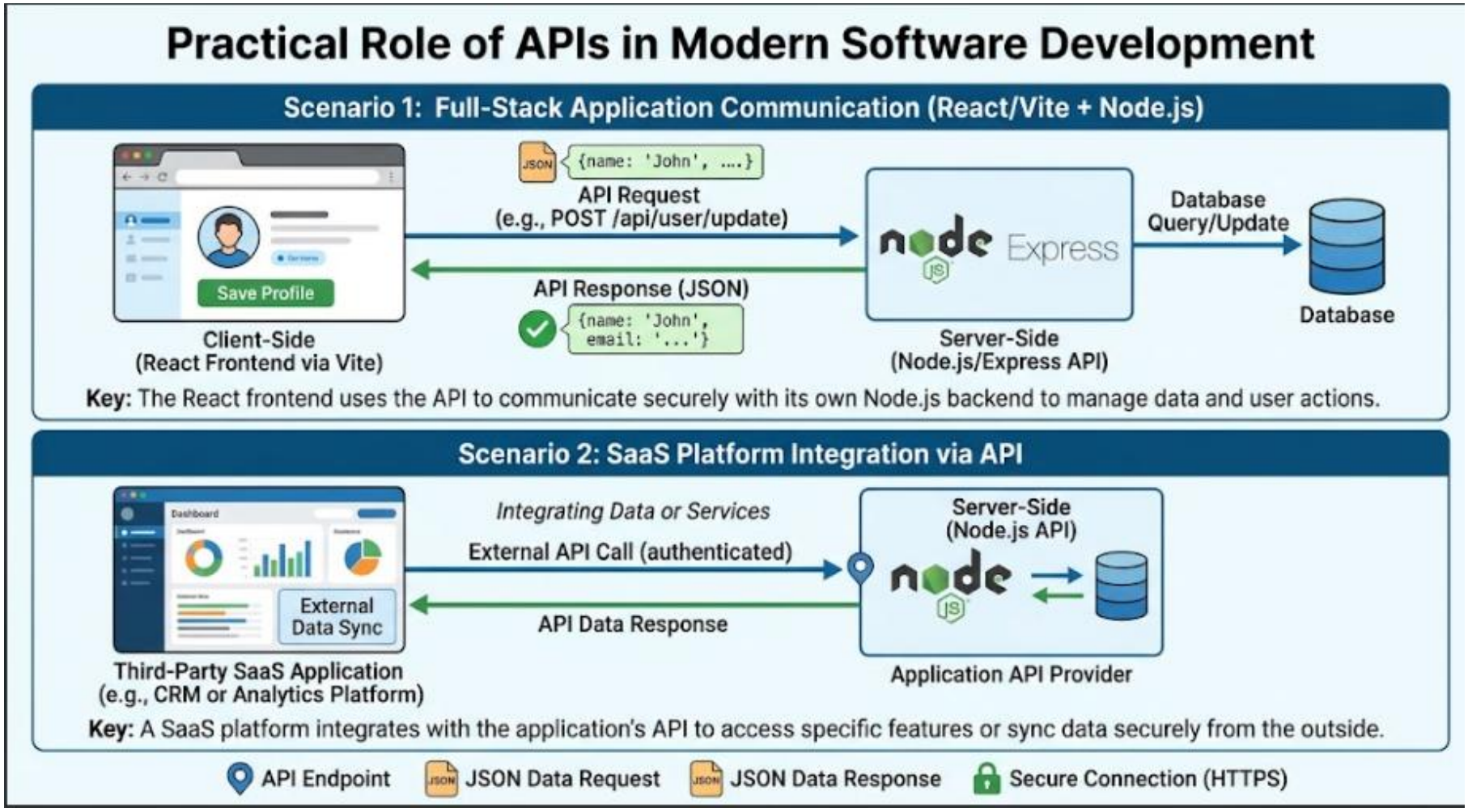
- **No passwords are shared** — only temporary, scoped tokens.

# API

# What is API ?


UNDERSTANDING APIs (Application Programming Interfaces)
How Software Talks to Software
Client App
Request
API
MENU
Response
Response
Processes Data
Customer
The Messenger
Server
1. Request
2. API Call
API
3. Server Processing
4. Response
API
CLIENT
SERVER
REQUEST & RESPONSE

# Examples


Practical Role of APIs in Modern Software Development
Scenario 1: Full-Stack Application Communication (React/Vite + Node.js)
Save Profile
Client-Side
(React Frontend via Vite)
JSON
{name: 'John', ....}
API Request
(e.g., POST /api/user/update)
API Response (JSON)
{name: 'John',
email: '...'}
node
Express
Server-Side
(Node.js/Express API)
Database Query/Update
Database
Key: The React frontend uses the API to communicate securely with its own Node.js backend to manage data and user actions.
Scenario 2: SaaS Platform Integration via API
Dashboard
External Data Sync
Third-Party SaaS Application
(e.g., CRM or Analytics Platform)
Integrating Data or Services
External API Call (authenticated)
API Data Response
Server-Side
(Node.js API)
node
Application API Provider
Key: A SaaS platform integrates with the application's API to access specific features or sync data securely from the outside.
API Endpoint
JSON Data Request
JSON Data Response
Secure Connection (HTTPS)

# 1-In code (internal API)

If you are writing software locally, an API can literally be a class interface or function signature.

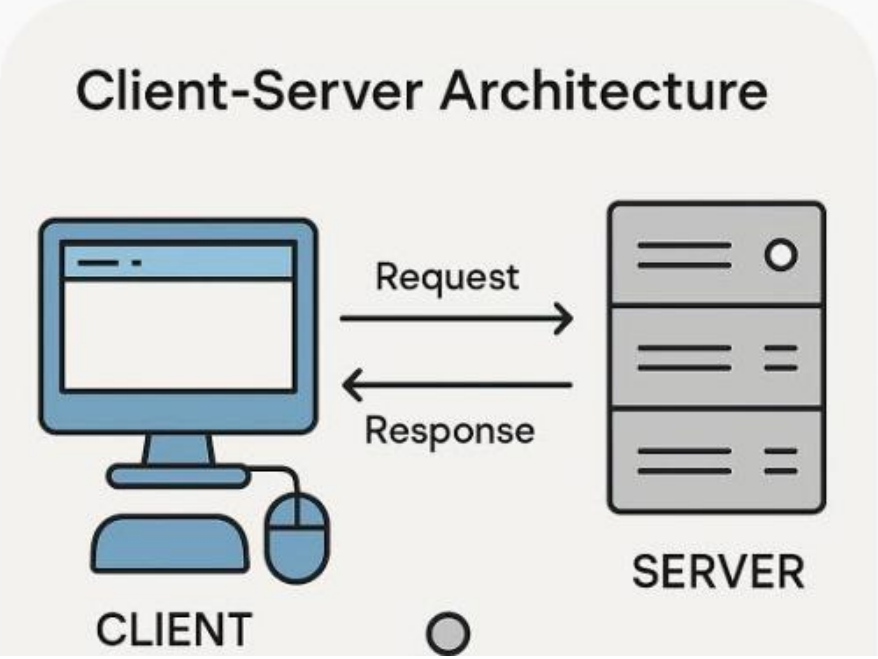
Python

```python
# The API specification (the agreement of what you can ask for):
class PaymentAPI:
    def process_payment(self, amount: float, card_number: str) -> bool:
        """Takes an amount and card number, returns True if successful."""
        pass
```

Here, `process_payment` is the **API**. You don't need to see the hundreds of lines of code handling bank servers, encryption, or database updates behind it—you just invoke the function according to its specified signature.

# 2-Over the web: WEB/REST API

When software communicates across the internet, the API takes the form of **URL Endpoints** that accept data and return structured responses (usually JSON).



# 2-Over the web: WEB/REST API

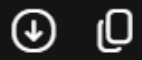
Example: Weather API

- Request (What your app sends):

```
GET [https://api.weather.com/v1/current?city=Beirut]
(https://api.weather.com/v1/current?city=Beirut)
```

- Response (What the API returns back):

JSON



```
{
  "city": "Beirut",
  "temperature": 22,
  "unit": "Celsius",
  "condition": "Sunny"
}
```

# Whats REST stands for?

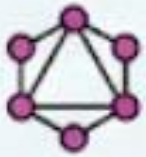
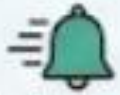
**REST** stands for **Representational State Transfer**.

It is an architectural style (a set of guidelines) for designing web APIs. Rather than calling complex remote functions, a REST API treats data as **Resources** identified by standard URLs.

- **Representational:** Data is delivered in a specific format (representation), usually **JSON** or **XML**.
- **State:** The data represents the current condition (state) of an object at that exact moment.
- **Transfer:** The server transfers that representation to the client over standard HTTP methods:
  - `GET` (Fetch data)
  - `POST` (Create new data)
  - `PUT` (Update existing data)
  - `DELETE` (Remove data)

# API types and examples

## Figure 1: Common API Types & Examples

| | REST | SOAP | GraphQL | Webhooks |
|---|---|---|---|---|
| |   |   |  |  |
| **Methods** | HTTP Verbs GET/POST | XML Messaging | Query Language | HTTP POST |
| **Characteristics** | Stateless | Structured/ WSDL | Flexible/ Client-defined | Event-driven |
| **Typical Use Cases** |  Web/Mobile Apps |  Enterprise Systems |  Complex Data Fetching |  Real-time Notifications |

# API Communication Flow

| Component | Description | Relevance to SaaS |
| --- | --- | --- |
| **Client Application** (Your App) | The external software or system that needs data or service from the SaaS. | This is your React/Vite application (the **frontend**) or your Node.js/Express app (the **backend**). |
| **Request** ( `GET /users` ) | A standardized HTTP message sent from the client to the server. | **CRUD** (Create, Read, Update, Delete) operations are usually mapped to HTTP methods: `GET` (Read), `POST` (Create), `PUT` / `PATCH` (Update), `DELETE` (Delete). |
| **SaaS API Endpoint** | The specific URL on the SaaS server that accepts the request. | This is the public entry point (e.g., `https://api.saasapp.com/v1/users` ). |
| **API Server/Logic** | The SaaS server-side code that executes the requested action. | Processes the request, interacts with the SaaS database, and generates the response. |
| **Response (Data)** | The data returned by the SaaS server, usually in **JSON** (JavaScript Object Notation) or XML format. | This data is then consumed and displayed by the Client Application. |

# API Communication Flow

