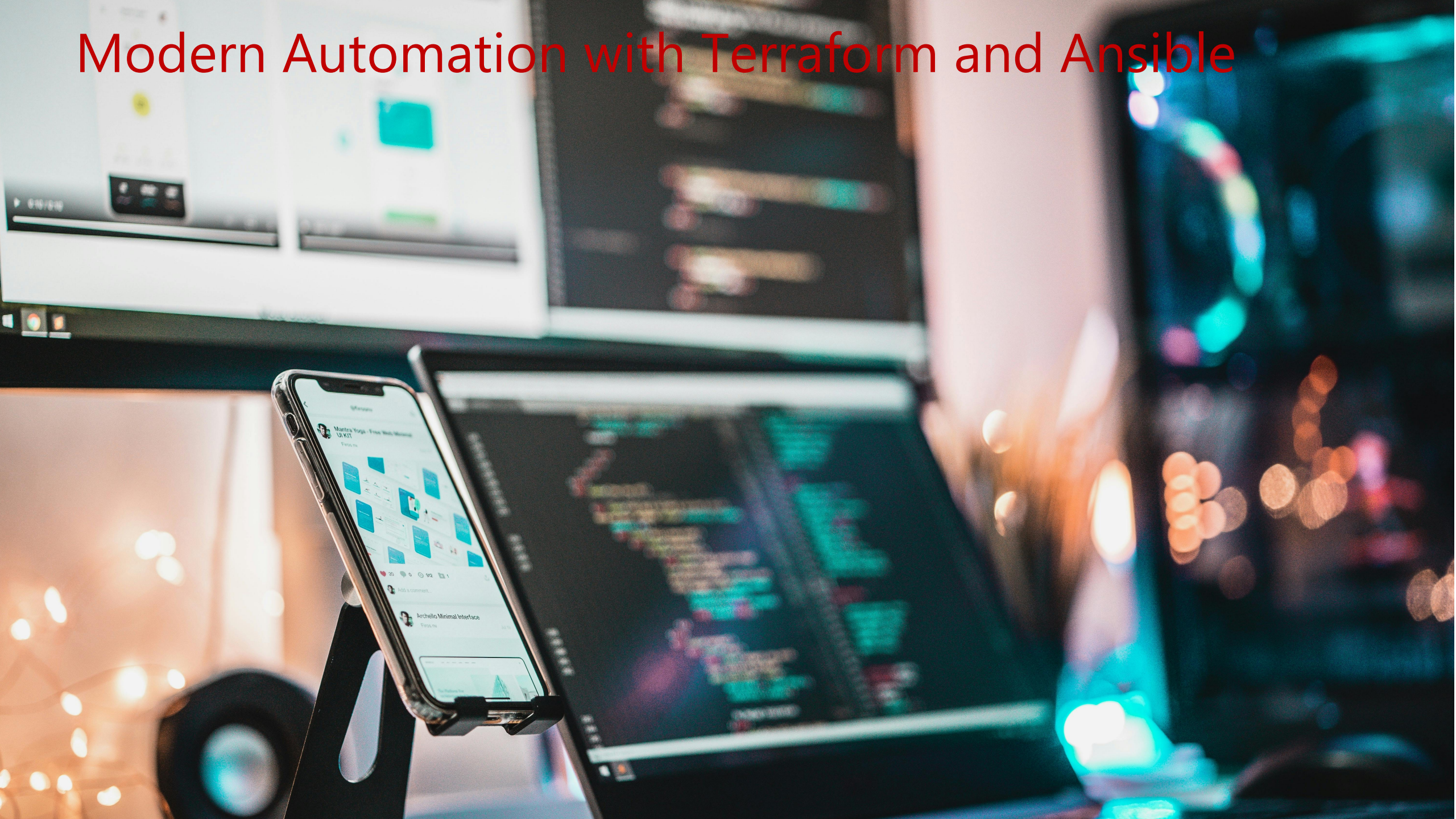
Modern Automation with Terraform and Ansible



TEKsystems Global Services

WORKFORCE DEVELOPMENT

Logistics

- Class Hours:
- Instructor will provide class start and end times.
- Breaks throughout class

- Telecommunication:
- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students

- Lunch:
- 1 hour 15 minutes

- Miscellaneous
- Courseware
- Bathroom

Course objectives

- Python Fundamentals
- Interact with APIs
- Understand the benefits of configuration management
- Deploy infrastructure with Terraform
- Configure infrastructure with Ansible
- Write advanced Ansible playbooks and roles
- Work with Ansible Automation Platform
- More!

Hi!

Jason Smith

Cloud Consultant with a Linux sysadmin background.
Focused on cloud-native technologies: automation, containers & orchestration

<u>Expertise</u>

- Cloud
- Automation
- CICD
- Docker
- Kubernetes

Introductions

- Name
- Job Role
- Your experience with Python (scale 1 - 5)
- Your experience with Ansible (scale 1 - 5)
- Your experience with Terraform (scale 1 - 5)
- Expectations for the course (please be specific)

Lab page

https://jruels.github.io/modern-automation/

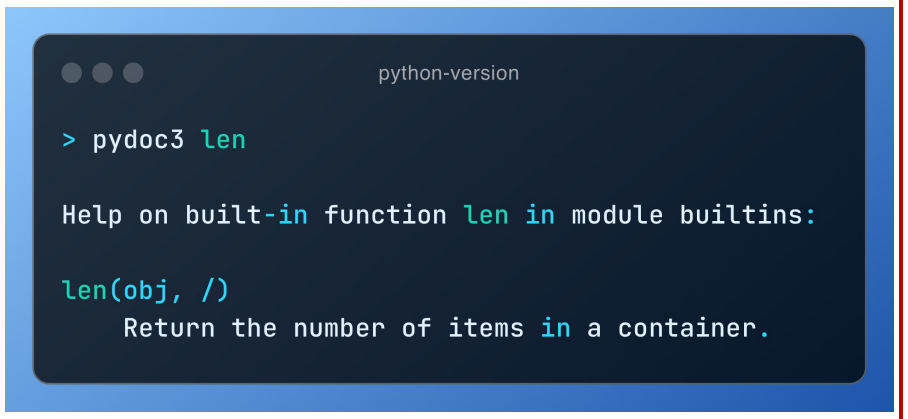
Python

Develop a passion for learning.

Properly configured

After the installation is complete, hopefully, the Python environment will be successfully installed. Test the Python install from a command window. Enter the "**python –V**" command. The command is case sensitive (uppercase V). If the version is presented, congratulations! This will confirm the correctness of at least a basic setup and configuration.

pydoc



- Python documentation is available using the pydoc command tool, which is a Python module.
- For convenience, add the sub-directory where pydoc.py is installed to the path variable. Use the same steps as provided before.
- For help on a particular command, type pydoc.py command

Python interpreter

A command line interpreter is provided with Python. The command line interpreter is a sandbox for testing Python commands. The interpreter provides immediate gratification.
Start the Python interpreter:
• Type py or `python` from the command prompt.
• Alternatively, start the interpreter from the Python installation directory.

python-interpreter.py

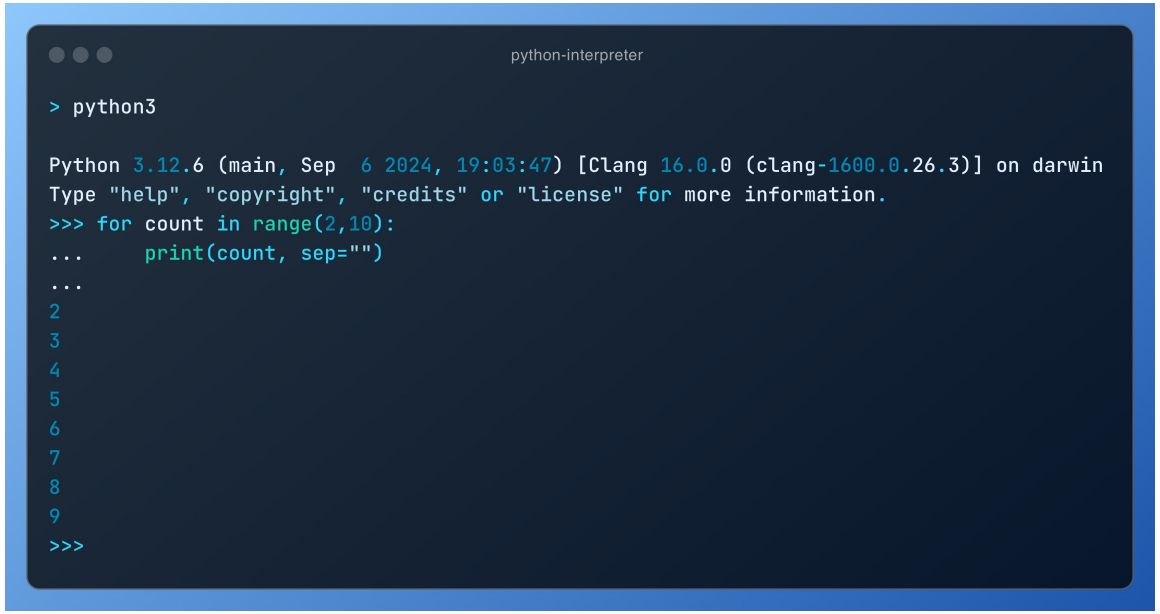
```
>python3
Python 3.12.6 (main, Sep  6 2024, 19:03:47) [Clang 16.0.0 (clang-1600.0.26.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import datetime
>>> first="Jason"
>>> last="Smith"
>>> currentdate=datetime.date.today()
>>> print(first, last, currentdate)
Jason Smith 2024-11-11
>>> exit()
```

Notes on Python interpreter

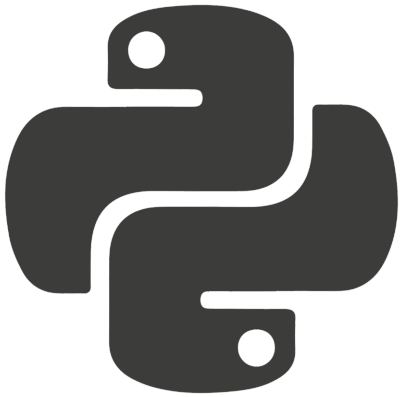
- An instance of the Python interpreter represents a single coding session.
- Help is available with the help command: help(command)
- Ellipses (…) indicates intermediate results.
- The interpreter will display the results of any expression – even without a print command.

Try it! Type 7*6 <enter>.

- There are a couple of ways to exit the Python interpreter. The exit function and ctrl-Z.



Python types



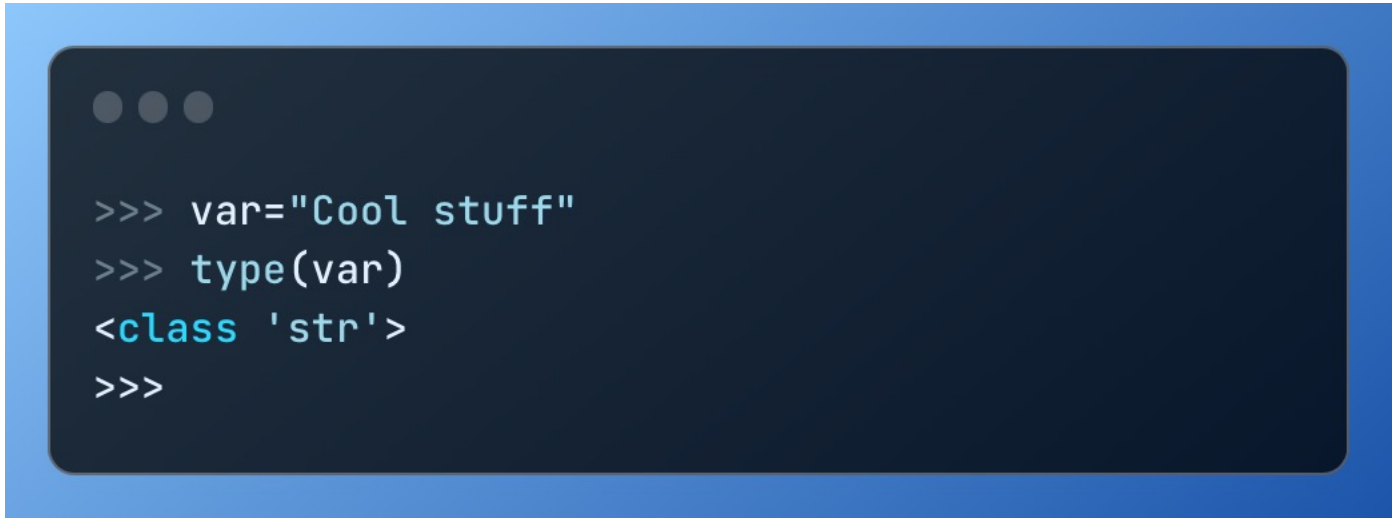
Variables are labels that are assigned to memory locations. Each memory location is a storage bin, which can contain data of a specific type.

- There are built-in and custom types
  - Integers, floats, strings, etc.
  - User-defined types (classes)
- Python is a dynamic language where variables are not assigned types declaratively.
- Nonetheless, Python is strongly typed.
- Objects can be either mutable or immutable. For example, lists are mutable. Sets and strings are immutable.

Python – declare a variable

It is simple to declare a variable in Python; assign a value!

- You do not specify a type, such as integer or float.
- The type is set based on the `rvalue` type `varname=rvalue`
- After the assignment, the variable is strongly typed



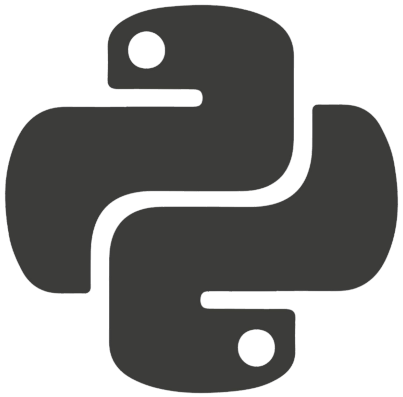
Python – declare a variable

5 MINUTES

Open the Python Interpreter

1. Declare three variables as listed
    1. var1: 42 – Integer
    2. var2: 10.0 - Float
    3. var3: "Hello" – String
2. Use the type function to display the type of each variable.

Python - Print



You will often use the `print` command to validate your growing knowledge of Python.

- Variable length argument list
- Must be bounded with parentheses

python_print.py

```
print(1,2,3, sep=",", end="")
print("", 4,5,6, sep=",")

# output:  1,2,3,4,5,6
```

Python - Print

You can also do basic formatting with the method: string.format . Using parameter placeholders { n }, where n indicates parameter nth in the parameter list. The function prototype for the string.format method is:

string.format(“output string with placeholders”, param1, param2, param3…)

Here is sample code:

python_print_formatting.py

```
var1="ABC"
var2="DEF"
print("Forward {0}{1}".format(var1, var2))
print("Reverse {1}{0}".format(var1, var2))

#output:
#Forward ABCDEF
#Reverse DEFABC
```

Python - Input

The input function reads data from the console. The result is stored in a variable. The only parameter is a prompt. Use the prompt to provide instruction to the user.

python_input.py

```
first=input("please provide first name ")
last=input("please provide last name ")
print(first, last)

# output: Jason Smith
```

Python - Input

The input function can be useful for suspending execution. The user then determines when to continue. Simply insert this statement or something similar strategically located in an application.

```
input("Press any <key> to end")
```

For example:

```
first=input("please provide first name ")
last=input("please provide last name ")
print=(first , last)
Input("Press any <key> to end")
```

Python – If

An if statement is the fundamental transfer of control statement. If a Boolean expression is True , the if block is executed. If False , the block is skipped. Execution continues immediately after the block

python-if_statement.py

```
#!/usr/bin/env python

theanswer=42
prompt="You are correct!"
_youranswer=input("what is the answer to life? ")
youranswer=int(_youranswer)
if(youranswer!=theanswer):
        print("Really!?")
        prompt="""Please read:
                the Hitchhicker's Guide to the
Galaxy"""
        print(prompt)
```

Python – If Else

The else statement adds a False branch to the if statement. That branch (block) executes if the condition is False .

Here is the syntax:

```
if(Boolean):
     True block
else:
     False block
```

python-if__else_statement.py

```
#!/usr/bin/env python
theanswer=42
_youranswer=input("what is the answer to life? ")
youranswer=int(_youranswer)
if(youranswer!=theanswer):
        print("Really!?")
        prompt="""Please read:
                the Hitchhicker's Guide to the Galaxy"""
else:
        prompt="You are correct!"
print(prompt)
```

Python – Elif

Deep nesting of if statements can lower the readability of code. The elif statement is an alternative syntax.

python-elif_statement.py

```
#!/usr/bin/env python
employee=“E"
if employee=="E":
        print("calculate exempt pay")
elif employee=="H":
        print("calculate hourly pay")
elif employee=="M":
        print("calculate management pay")
else:
        pass  # do nothin
```

Python – Break and continue

• *break* statement: interrupts an iteration. Immediately ends the current loop and proceeds to the next statement after the iteration block.

•*continue* statement: skips the remainder of the current iteration. Resumes at the top of the next iteration.

python-break-continue.py

```
while(True):
    var=input("Enter  value:  ")
    if  var:
        continue
    else:
        break
```

Python – while else

The break statement explicitly ends a loop. Alternatively, the loop may end naturally, such as when the while condition is False . The else block is executed when a loop finishes naturally (no break).

Here is the syntax:

```
while:
        block
else:
        block
```

python-while-else.py

```
var= 4
while(var!= 0 ):
    print(var)
    var-= 1
else:
    print("Loop  ended  naturally")    print("Loop
exited"
```

Python – switch statement

In Python 3.10 a switch/case statement was added!

The break keyword's functionality is done for you behind the scenes.

python-case_statement.py

```
#!/usr/bin/env python
lang = input("What's the programming language you want to learn? ")

match lang:
    case "JavaScript":
        print("You can become a web developer.")

    case "Python":
        print("You can become a Data Scientist")

    case "PHP":
        print("You can become a backend developer")

    case "Solidity":
        print("You can become a Blockchain developer")

    case "Java":
        print("You can become a mobile app developer")
    case _:
        print("The language doesn't matter, what matters is solving problems.")
```

Python – for loops

The *for* loop iterates the elements of an iterative collection in sequence.

- The *range* statement is a common way to create a sequence.
- The *break* , *continue* , and *else* statement work the same as in a while loop.

python-for-loop.py

```
for var in range(1, 5):
    print(var)
else:
    print("Loop ended naturally")
print("Loop exited")
```

Python – functions

Python functions are reusable blocks of code designed to perform a specific task, helping to keep code organized and maintainable. They allow developers to encapsulate functionality, making code easier to debug and understand.

python-functions.py

```python
def greet_user():
    name = input("Please enter your name: ")
    print(f"Hello, {name}! Nice to meet you!")

# Call the function
greet_user()
```

Python – functions

Python functions are reusable blocks of code designed to perform a specific task, helping to keep code organized and maintainable. They allow developers to encapsulate functionality, making code easier to debug and understand.

- Defined using the def keyword, followed by the function name and parentheses.

python-functions.py

```
def greet_user():
    name = input("Please enter your name: ")
    print(f"Hello, {name}! Nice to meet you!")

# Call the function
greet_user()
```

Python – functions

Python functions are reusable blocks of code designed to perform a specific task, helping to keep code organized and maintainable. They allow developers to encapsulate functionality, making code easier to debug and understand.

- Defined using the def keyword, followed by the function name and parentheses.
- Parameters can be included to allow functions to accept input values, enhancing flexibility.

Python – functions

Well-documented functions improve readability, and calling functions is simple, enabling you to execute the code within the function from anywhere in your program.

Add docstrings to functions for clear documentation, which explains what the function does and how to use it.

- Docstring: The docstring """Prompts the user to enter their name..."""" describes the function's behavior, noting that no parameters are required.

python-functions.py

```python
# Define a function to greet a user with an input prompt
def greet_user():
    """
    Prompts the user to enter their name and prints a personalized greeting.

    No parameters are needed for this function.
    """
    name = input("Please enter your name: ")
    print(f"Hello, {name}! Nice to meet you!")

# Call the function
greet_user()
```

Python – functions

Well-documented functions improve readability, and calling functions is simple, enabling you to execute the code within the function from anywhere in your program.

Add docstrings to functions for clear documentation, which explains what the function does and how to use it.

- Function Call: greet_user() calls the function, which then prompts the user to enter their name and outputs a greeting based on the input.

python-functions.py

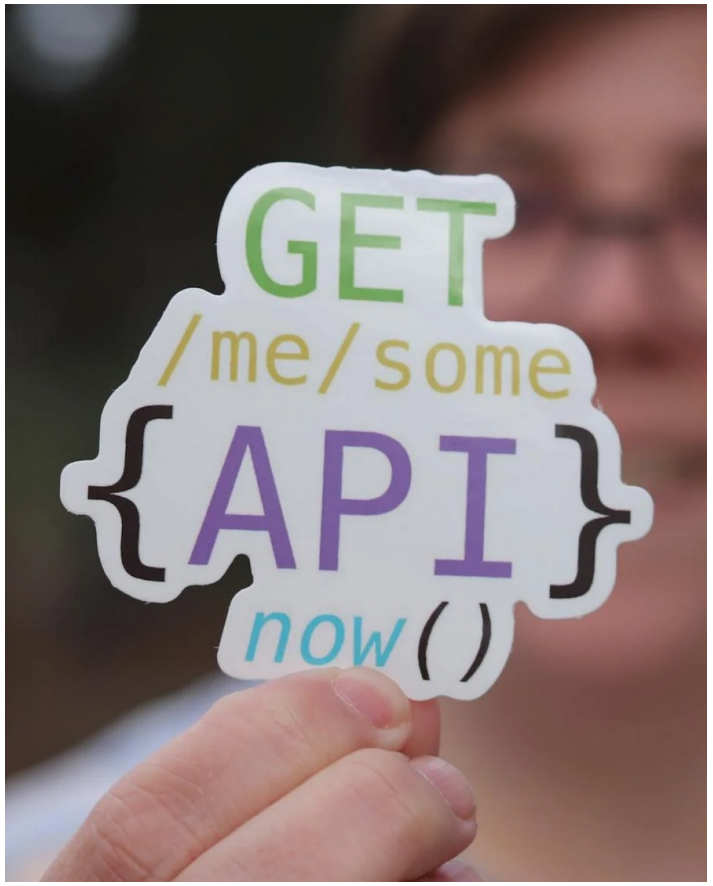
```python
# Define a function to greet a user with an input prompt
def greet_user():
    """
    Prompts the user to enter their name and prints a personalized greeting.

    No parameters are needed for this function.
    """
    name = input("Please enter your name: ")
    print(f"Hello, {name}! Nice to meet you!")

# Call the function
greet_user()
```

Lab: Python guessing game

REST APIs

Develop a passion for learning.

Understanding APIs



API Basics

- An API (Application Programming Interface) allows different software applications to communicate and share functionality, enabling seamless integration across systems.

Topics Covered

Overview of APIs and REST:

- Learn the basics of APIs, with a focus on REST architecture and how it supports structured, scalable communication.

API Endpoints and Methods:

- Understand how endpoints and HTTP methods (GET, POST, PUT, DELETE) define actions on resources within an API.

Authenticating with APIs:

- Explore methods for secure API access, including API keys, OAuth, and JWT (JSON Web Tokens).

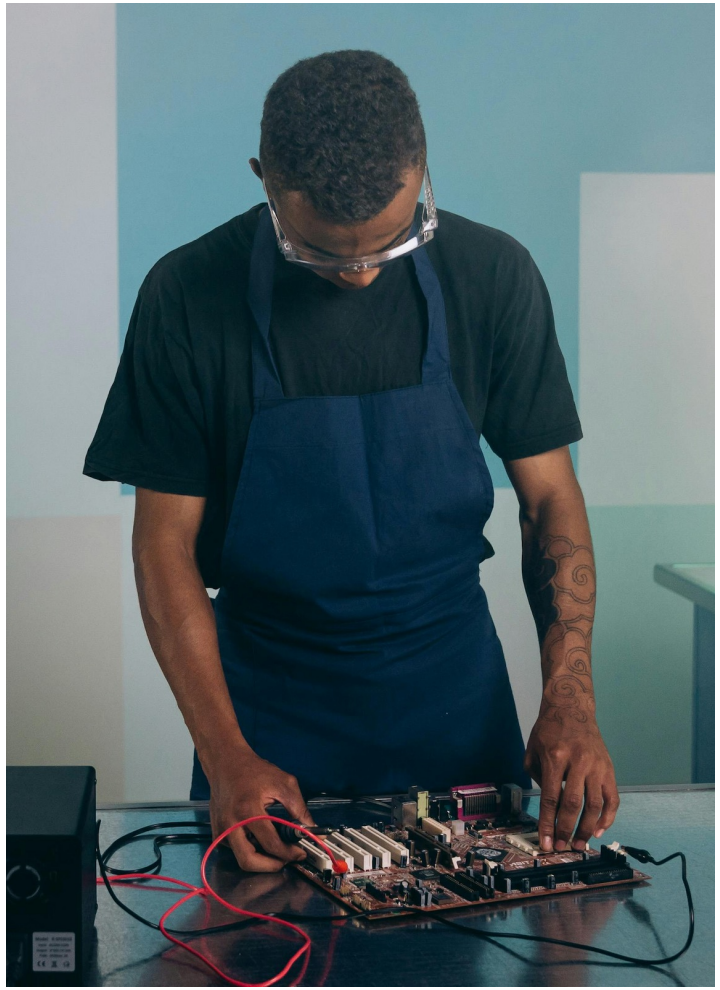
Introduction to APIs

An **API**, or **Application Programming Interface**, allows different software applications to communicate with each other. APIs are like contracts: they define how systems interact.

APIs enable applications to access data and functions from other services, creating integrations without needing to share full source code.

- **Real-world Examples:** APIs allow weather data on websites, payment processing via PayPal, and authentication through social media logins.
- **Significance in Modern Applications:** APIs are crucial for web, mobile, and cloud-based applications, supporting diverse ecosystems and creating seamless user experiences.

KEY CONCEPTS AND COMPONENTS OF APIs



Endpoints: Specific URLs that denote where API resources or services are located. Each endpoint represents an action or data point.

Requests and Responses: API communication is a two-way exchange. The client sends a request, and the server responds, often with data or an action status.

Data Formats: JSON (JavaScript Object Notation) and XML (eXtensible Markup Language) are commonly used. JSON is preferred for its readability and ease of use with JavaScript-based web applications.

API Keys and Authentication: Many APIs require authentication via keys, tokens, or OAuth to secure and control access, ensuring only authorized users can interact with the API.

Rest Overview - The API Architecture

REST (Representational State Transfer) is an architectural style for designing networked applications, relying on a stateless communication model.

Core REST Philosophy: RESTful APIs are designed around resources, where each resource is represented by a unique URL, and operations are conducted using standard HTTP methods.

Benefits: REST APIs are scalable, adaptable, and straightforward to use, making them ideal for web-based interactions. RESTful design emphasizes simplicity, making it developer-friendly and well-suited for large, distributed systems.

Fundamental Principles of Rest Architecture

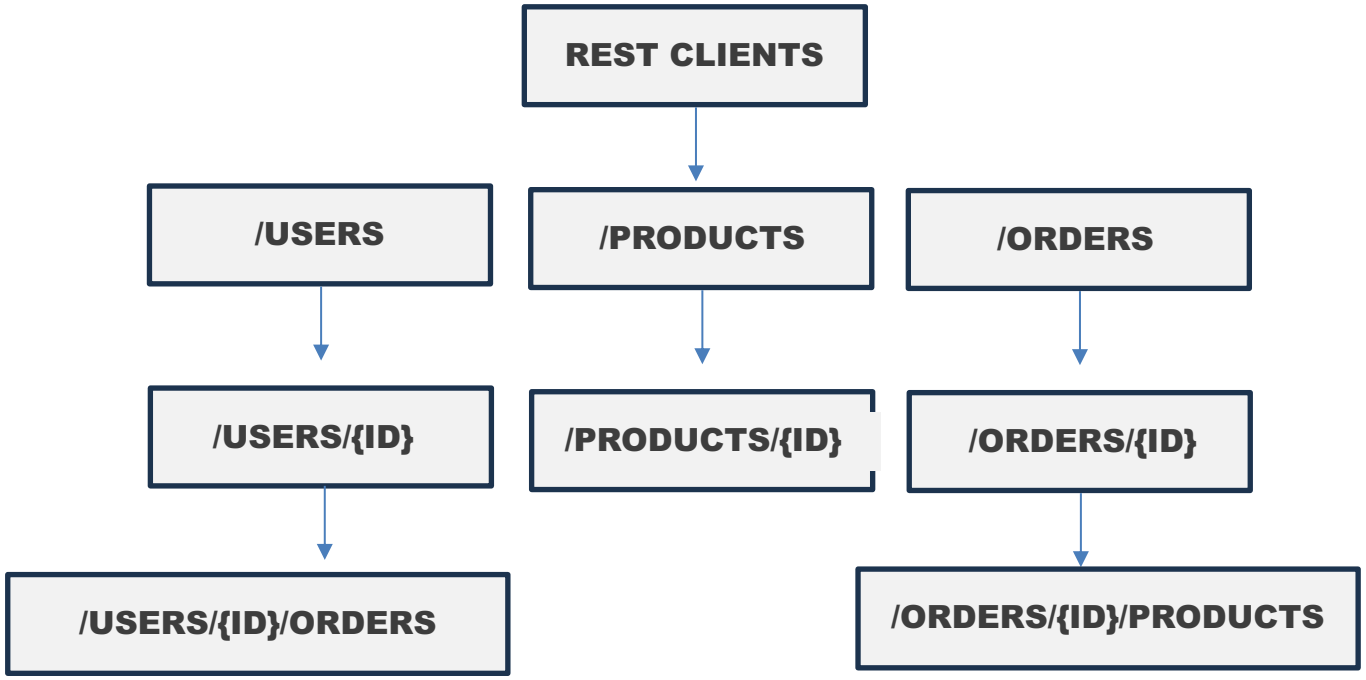
Stateless Communication: Each client request must contain all necessary information, allowing the server to process it independently of previous interactions. This approach simplifies scaling and troubleshooting.

Uniform Interface: REST APIs adhere to a uniform interface that simplifies and standardizes API usage. Clients interact with the server via defined methods (GET, POST, PUT, DELETE).

Layered System: A REST API can be layered to increase scalability and redundancy. Clients interact with the endpoint directly without knowing the exact back-end structure or intermediary layers.

Cacheable Responses: REST supports caching, which reduces server load and accelerates response times for repeated requests.

Structuring Restful APIs

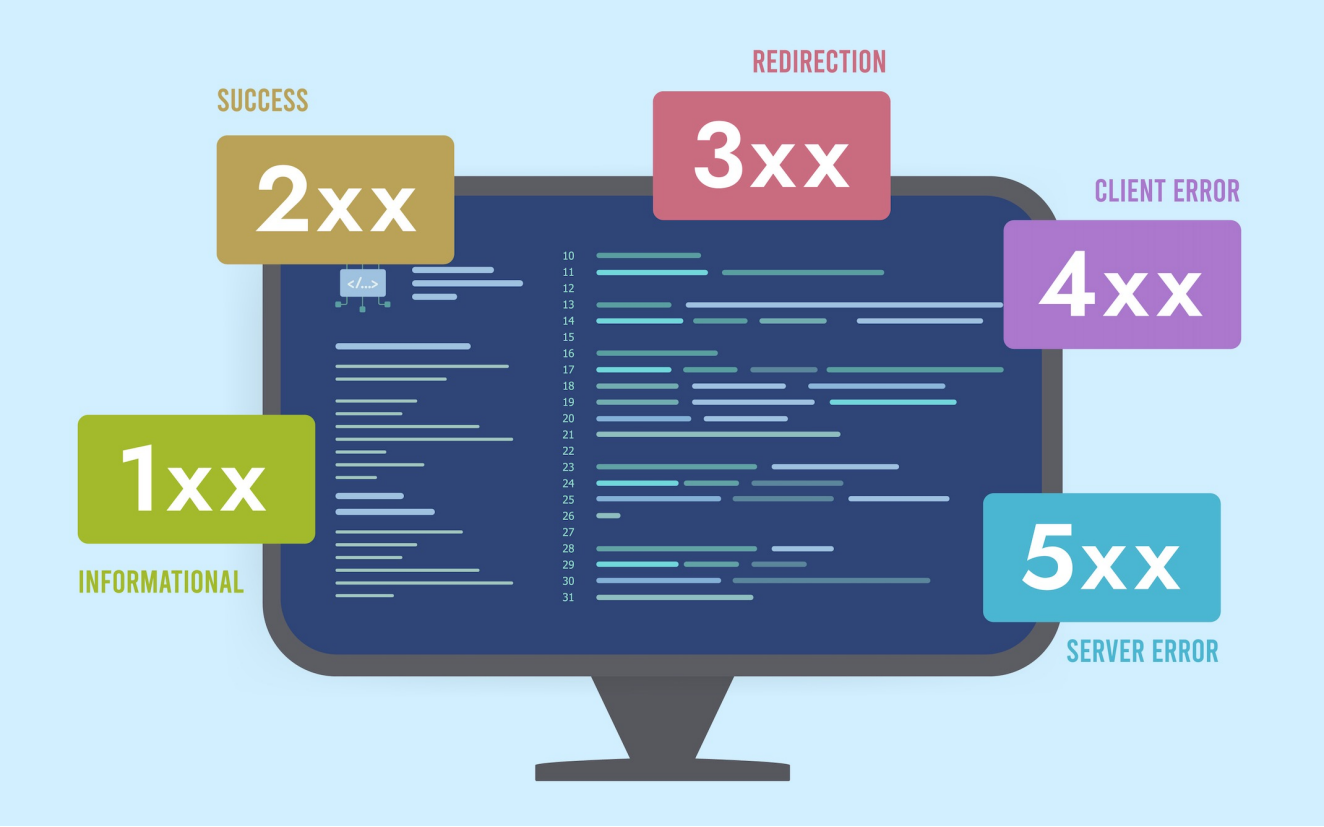


Resources as Central Elements: Each piece of data or functionality in a REST API is a "resource" and is represented by a unique URL. Examples of resources include users, products, orders, and accounts.

Resource Naming Conventions: Naming conventions are essential for consistency. Resource names are typically nouns and often in plural form (e.g., /users, /products).

Hierarchical Organization: Resources can be organized in a hierarchical way, using endpoints like /users/123/orders to show relationships between entities, enhancing the logical structure and usability.

HTTP status codes in REST APIs



Status codes provide feedback to the client about the outcome of a request.

Key Codes:

- **200 OK:** Request succeeded.
- **201 Created:** A new resource was successfully created.
- **400 Bad Request:** The request was invalid or improperly formatted.
- **401 Unauthorized:** Authentication is required but was missing or failed.
- **404 Not Found:** The requested resource was not found.
- **500 Internal Server Error:** Server encountered an unexpected issue.

Value in Debugging: Status codes help developers diagnose and handle errors more effectively, improving application reliability and user experience.

Benefits And Use Cases Of Restful APIs

Benefits

- **Scalability:** REST APIs can handle large amounts of requests due to their stateless design.
- **Simplicity:** RESTful APIs are easy to learn and implement, with widely adopted conventions.
- **Flexibility:** REST APIs can be used with various programming languages and platforms.

Use Cases

- **Web and Mobile Apps:** REST APIs are foundational in client-server architecture for apps needing remote data.
- **Third-party Integrations:** RESTful design facilitates integration with other services and APIs (e.g., payment gateways, cloud services).
- **IoT (Internet of Things):** REST APIs enable communication between connected devices and applications, supporting smart ecosystems.

Challenges And Best Practices For Restful API Design

Challenges

- **Handling Complexity:** As systems grow, REST APIs can become challenging to manage and version.
- **Error Handling:** Designing robust error responses and status codes can be difficult but essential for usability.
- **Rate Limiting:** Controlling usage rates is vital for preventing abuse and ensuring service availability.

Best Practices

- **Use Consistent Naming Conventions:** Clear and predictable endpoint names improve API usability.
- **Document API Endpoints and Responses:** Comprehensive documentation (e.g., OpenAPI) helps developers use the API effectively.
- **Versioning:** Implement version control for API updates to ensure backward compatibility.

Conclusion: A well-designed RESTful API is scalable, secure, and easy for developers to understand and use, making it a powerful tool for modern applications.

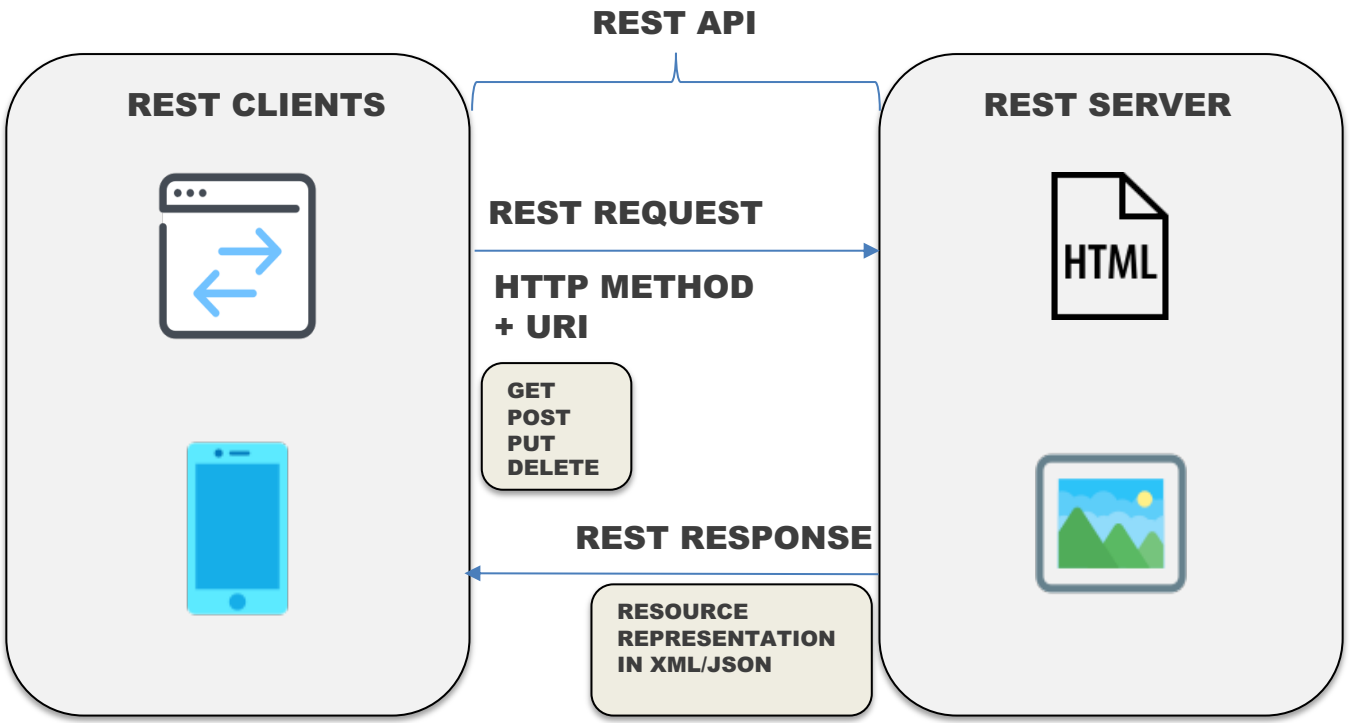
Rest API Architecture In Action

Client-Server Communication: REST APIs rely on a client-server model, where the client (e.g., a web or mobile application) makes requests, and the server processes them and returns responses. This separation allows for flexibility in development, as the client and server can evolve independently.

Stateless Interactions: Each request from the client is self-contained, meaning the server doesn't retain any session information. This approach enhances scalability, as each request can be handled independently.

Resource-Centric Design: REST APIs focus on resources, each with a unique URL. Clients interact with resources by sending requests to these URLs, allowing a clear and structured way to access and manipulate data.

Rest API Architecture in Action



1. **Client Initiates Request:** A REST client (web or mobile app) starts by sending a request to access or modify a resource on the server.

2. **HTTP Method and URI:** The request includes an HTTP method (GET, POST, etc.) and a URI to specify the desired resource.

3. **Server Processes Request:** The REST server receives the request and accesses the specified resource.

4. **Server Sends Response:** The server retrieves or updates the resource and sends back a response.

5. **Data in JSON/XML Format:** The response contains data in JSON or XML format for the client to use.

Rest API Use Cases Across Industries

E-commerce Platforms: REST APIs support product catalogs, orders, and customer accounts, enabling seamless online shopping experiences.

Social Media: APIs provide access to user data, posts, and media, allowing for integration with third-party applications and services.

Banking and Finance: REST APIs facilitate secure and efficient transactions, account management, and customer interactions through apps.

Healthcare: APIs enable the sharing of patient data across systems, supporting interoperability and improving patient care.

Smart Devices and IoT: RESTful communication between devices and servers supports smart home ecosystems and connected appliances.

API Endpoints and Methods

Develop a passion for learning.

Understanding API Endpoints

API Endpoints act as the gateways for applications to access specific functionalities or resources on a server. They define the location where a request is made, allowing clients to interact with server-based services seamlessly. Each endpoint is a unique URL tied to a resource, such as a database of users, a list of products, or a set of customer orders.

Endpoints are critical because they determine the “entry points” for client applications, making it possible for users to access services without knowing internal server structures. For example, an endpoint like /products/123 provides access to a product’s details using a simple URL structure, masking the complexity of the underlying system.

Components Of An API Endpoint

Base URL: The main URL for all API requests (e.g., https://api.example.com). This acts as the foundation for accessing different resources.

Path: The path specifies the exact resource and its hierarchical structure, such as /users/123/orders, where users is the main resource, 123 is a specific user ID, and orders indicates the user's order history.

Query Parameters: These optional components allow for additional customization, such as filtering or sorting (e.g., ?sort=price). This flexibility allows the client to refine the data returned.

Headers: Headers provide metadata about the request, including authorization tokens (Authorization: Bearer <token>) and content types (Content-Type: application/json). These ensure secure, efficient, and well-defined communication.

Common HTTP Methods and Their Roles

GET: Used to retrieve or "read" data from the server. For example, GET /products retrieves the entire product catalog, while GET /products/123 provides details about a specific product.

POST: Used to create new resources. Sending a POST request to /orders with order details adds a new order to the system. This method requires a request body with necessary information (e.g., product IDs, quantities, payment details).

PUT: Employed for updating existing resources. For instance, PUT /users/123 updates the details of user 123. Unlike POST, which adds data, PUT modifies an existing entry.

DELETE: Removes a resource from the server. DELETE /users/123 permanently deletes user 123's data, often requiring authentication to ensure secure access.

Practical Examples Of API Endpoints and Methods

To see these methods in practice, consider a User Management API for handling user accounts:

- **GET /users:** Returns a list of all users, which could be used to populate a user directory.
- **GET /users/{id}:** Retrieves details for a specific user, allowing clients to pull up profile information for that user.
- **POST /users:** Creates a new user. The client sends the user data (such as name, email, and password) in the request body, and the server responds with a new user ID upon success.
- **PUT /users/{id}:** Updates an existing user's information by replacing or modifying data. This is useful for updating profile fields like address or phone number.
- **DELETE /users/{id}:** Deletes a user account. Since deletion is a critical action, this endpoint may require higher permissions, like admin access.

Best Practices For Designing API Endpoints and Methods

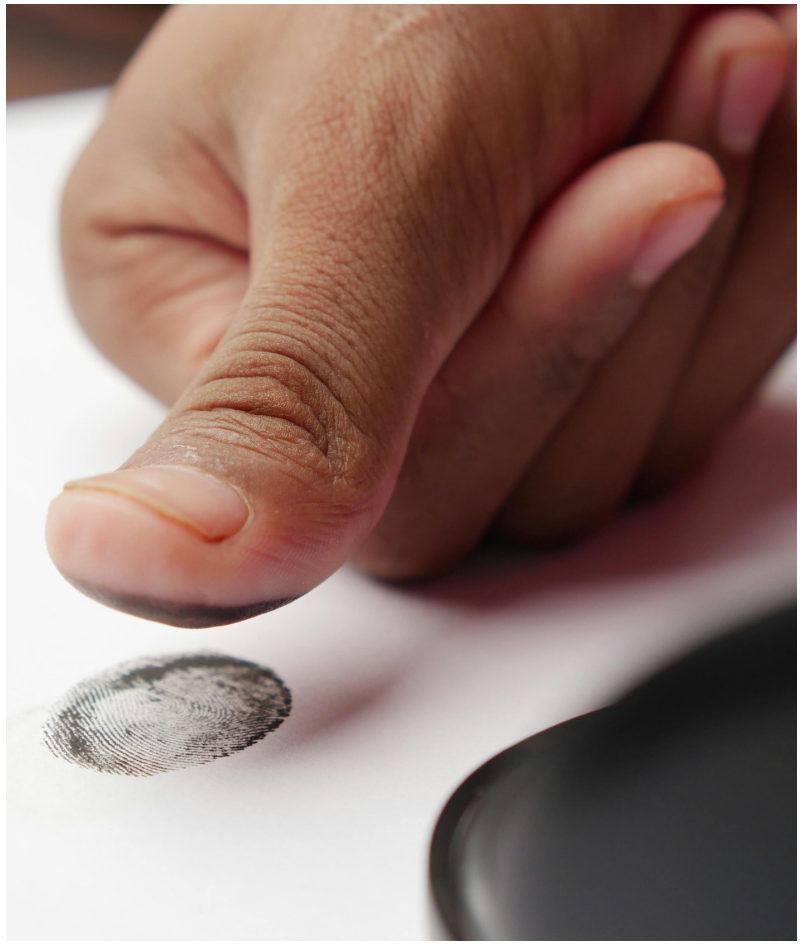
Consistent Naming Conventions: Use clear, descriptive names for endpoints, like /products, /orders, and /users. Consistency in naming makes APIs predictable and easy to understand.

Logical, Hierarchical Organization: Structure endpoints to reflect logical relationships, such as /users/123/orders for a user's orders. This makes it clear which data belongs to which resource and supports easy navigation.

Use Appropriate HTTP Status Codes: Provide clear responses for each request, like 200 OK for successful operations, 404 Not Found for missing resources, and 500 Internal Server Error for issues. Status codes help clients understand the outcome of each request, making error handling simpler.

Stateless Design: Ensure each request is self-contained, including all necessary information. This allows the API to handle requests independently and improves scalability by not relying on stored session data.

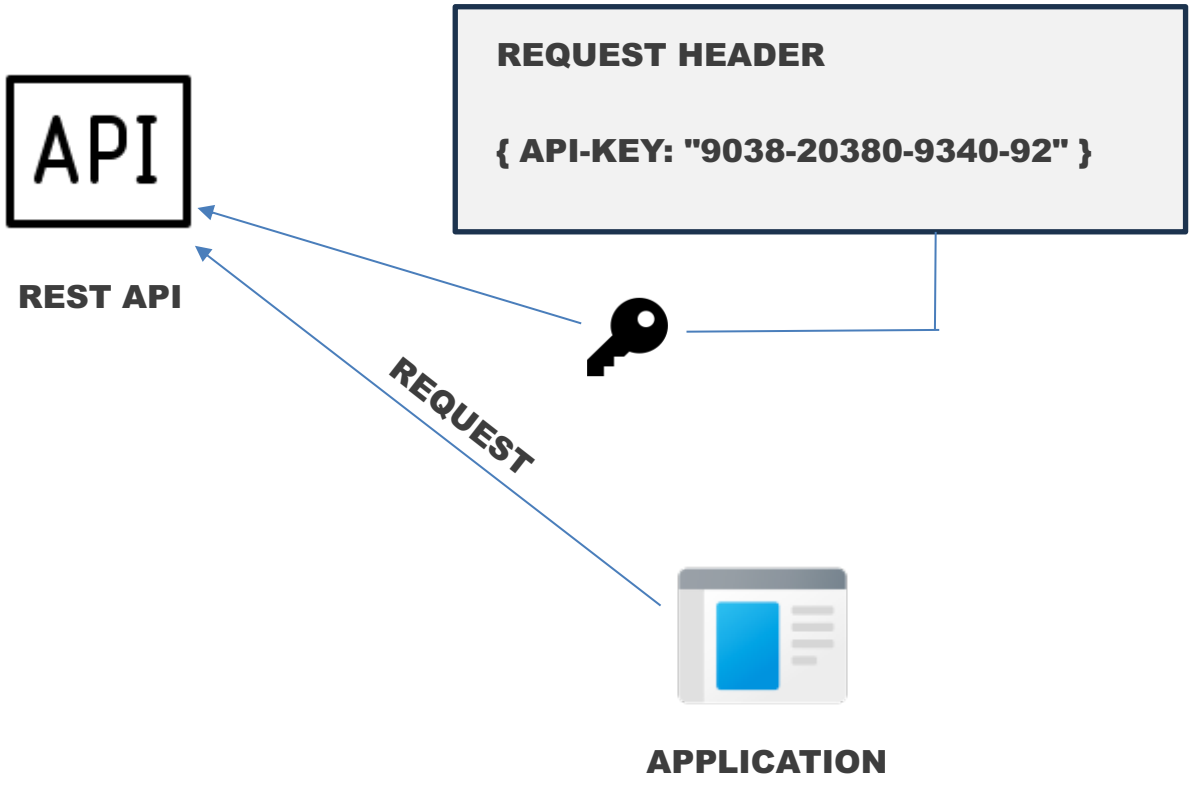
API Authentication



API authentication is the process of verifying the identity of a user or application trying to access an API. By authenticating requests, APIs can secure data and ensure that only authorized users and apps access sensitive information.

Different methods are used to authenticate, each suited to specific needs, including **API Keys** for **simple access**, **OAuth** for delegated permissions, and **JWT (JSON Web Tokens)** for secure token-based sessions. API authentication is essential for safeguarding resources and managing access control.

API Authentication – API Keys



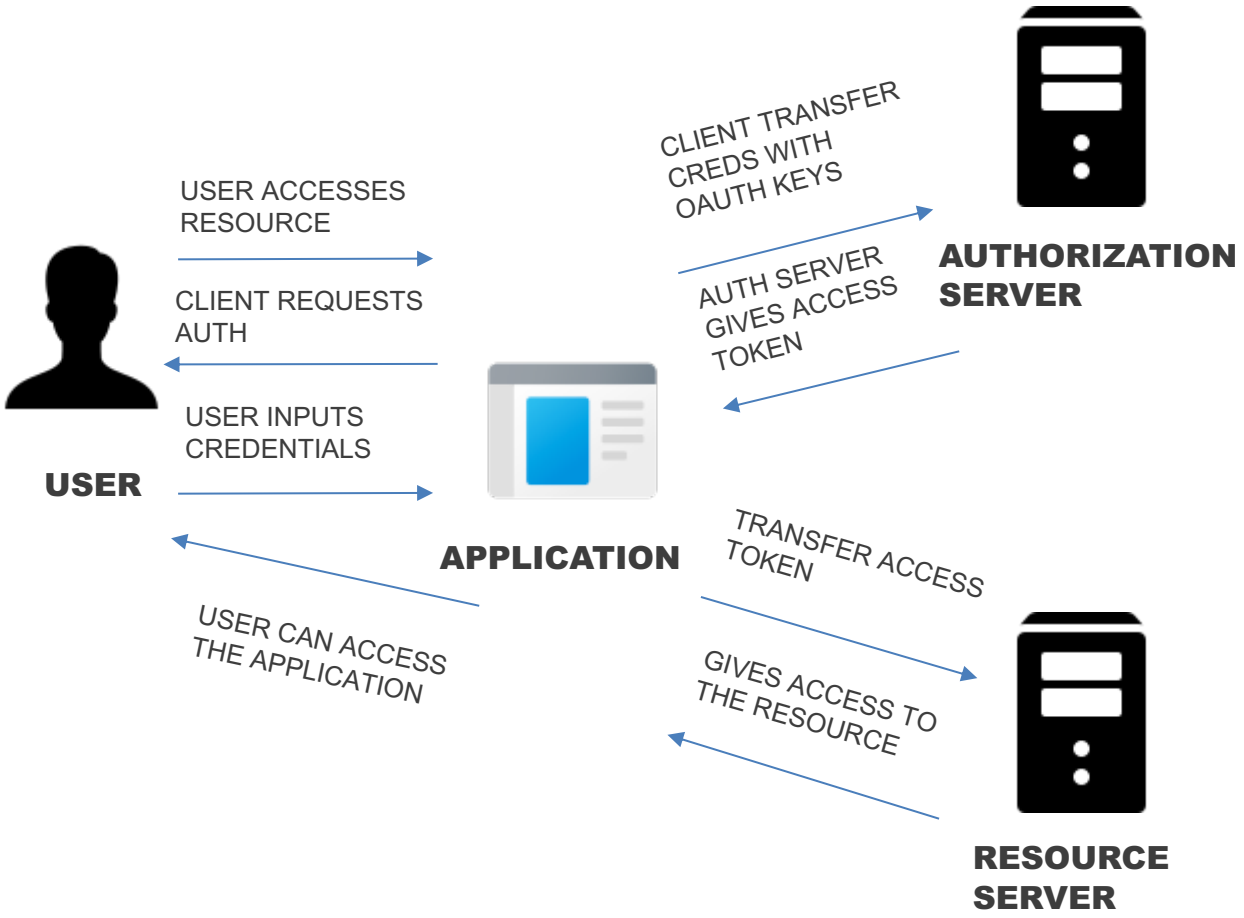
API Keys are unique strings generated by the server and assigned to each user or application to identify them.

- **How They Work:** An API key is sent with each request, typically in the request header, URL, or body, to verify the client's identity. For example, GET /data?api_key=YOUR_API_KEY

- **Pros:** API keys are easy to implement and suitable for public or low-security APIs where users only need basic access control.

- **Cons:** They provide limited security, as they can be easily exposed if not handled properly, lacking encryption and token expiration.

API AUTHENTICATION – OAUTH



OAuth is a secure method for allowing users to grant limited access to their data on a third-party app without sharing their passwords.

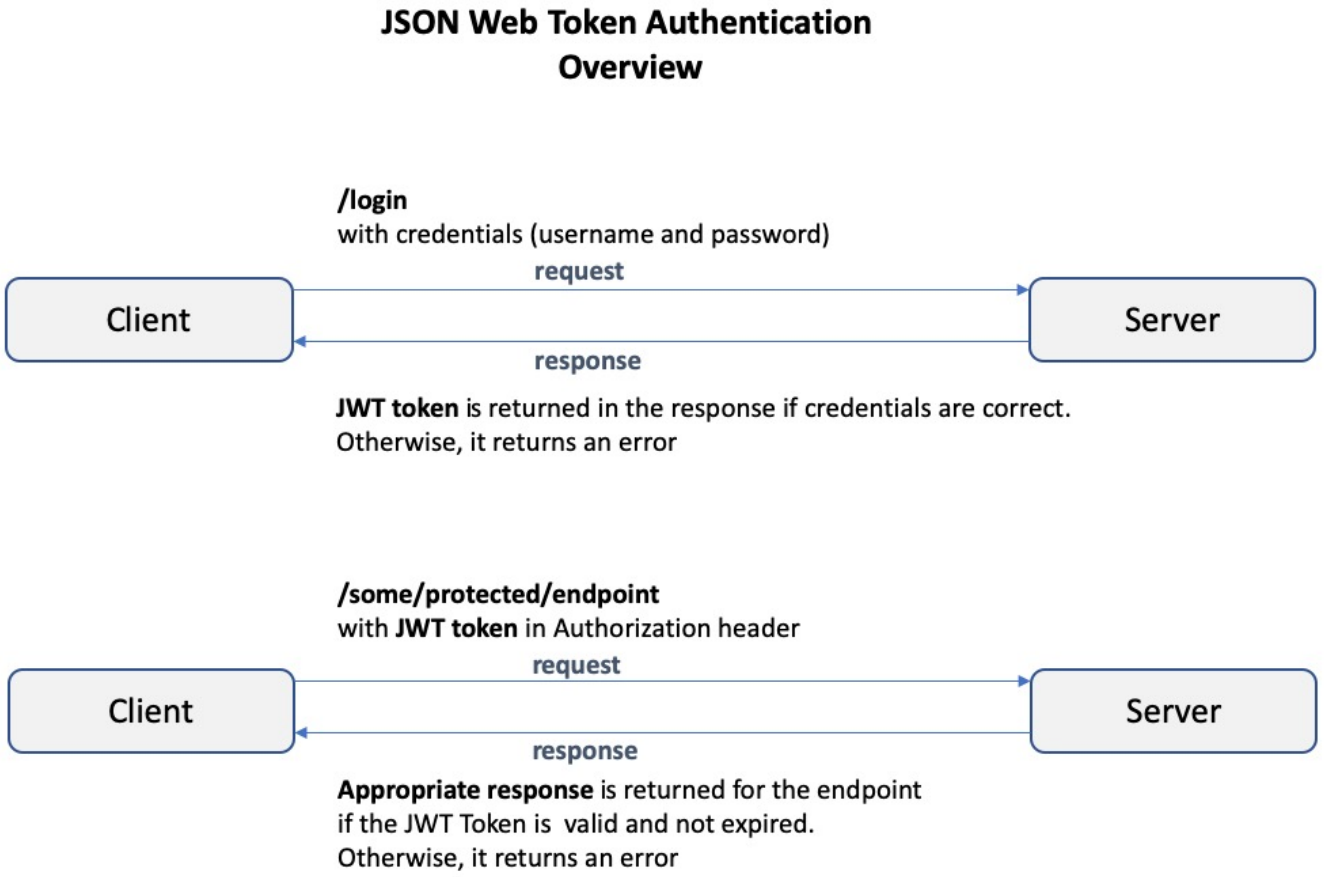
How It Works: OAuth uses an authorization server to authenticate users. The server issues an access token to the app, which the app includes in API requests to access resources on the user's behalf.

OAuth Steps:

1. The user is directed to the OAuth provider for login.
2. Once logged in, the provider issues a token.
3. The app uses this token to make requests with the user's permissions.

Pros and Cons: OAuth is secure and allows fine-grained permissions, making it ideal for applications that need access to specific user data, though it is more complex to set up.

API Authentication – JWT



JWT is a popular token-based method for secure, stateless sessions.

How JWT Works: Upon user login, the server creates a JWT containing user info, which is returned to the client. The client includes the JWT in the authorization header (e.g., Authorization: Bearer <JWT>) for future requests.

JWT Structure:

- Header: Specifies token type and algorithm.
- Payload: Contains user data and claims, like roles.
- Signature: Verifies token integrity and authenticity.

Benefits: JWTs support stateless sessions, making them scalable and ideal for applications needing secure, repeated requests without server-stored sessions.

Best Practices For API Authentication

To improve **API security**, follow these best practices for authentication:

- **Use HTTPS:** Always encrypt data in transit to protect credentials.
- **Rotate API Keys and Limit Access:** Regularly change keys and restrict access to specific IPs or actions to reduce the risk of exposure.
- **Token Expiration and Refresh:** Set expiration times for OAuth and JWT tokens and enable refresh tokens to mitigate risks from stolen tokens.
- **Verify JWTs:** Ensure tokens are verified on the server to confirm their authenticity and integrity.
- **Scope Permissions:** In OAuth, use scopes to limit data access, allowing users to grant only necessary permissions.

Making API requests – python 'requests' library

APIs (Application Programming Interfaces) allow applications to interact with external services by sending requests and receiving responses. Through APIs, Python applications can access remote servers, fetch data, and perform actions that add dynamic capabilities to an application.

- Python's requests Library: The requests library simplifies interacting with APIs by allowing Python to send HTTP requests with minimal code. With this library, Python can make GET, POST, PUT, and DELETE requests, which are essential for data retrieval and management.

- Core Use Cases: Using APIs, Python can connect with real-world data sources and external services, creating applications that are dynamic, responsive, and capable of real-time interactions.

python

```
# Install requests library
# (if not already installed)
!pip install requests
```

Python - Making GET requests

The **GET** method is used to retrieve data from a specified API endpoint. GET is non-destructive, meaning it only accesses data without modifying it, making it safe for repeated use and ideal for simply reading information.

How GET Works: Using *requests.get(),* we provide the API's URL, and Python sends a GET request to that URL. The server processes the request and responds with the requested data, typically in JSON format.

Common Use Case: GET requests are widely used to fetch lists or specific records. Examples include retrieving a list of products, fetching user details, or accessing posts on a website.

python

```python
import requests

url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.get(url)

print(response.status_code)
print(response.json())
```

Python - Making POST requests

The **POST** method allows us to send data to the API to create a new resource on the server. POST is typically used to add a new item, like creating a new record in a database or posting a comment.

How POST Works: With *requests.post()*, we specify the API's URL and send data in the request body, usually formatted as JSON. The server processes this data, creates a new entry, and returns a response containing the details of the created resource.

Common Use Case: POST requests are essential for creating new entries in a system, such as registering a new user, adding a product to a catalog, or posting a new blog entry.

```python
url = "https://jsonplaceholder.typicode.com/posts"
data = {
    "title": "New Post",
    "body": "This is a new post.",
    "userId": 1
}

response = requests.post(url, json=data)

print(response.status_code)
print(response.json())
```

Python - Making PUT requests

The **PUT** method is used to update an existing resource by replacing it with new data. PUT is typically used for modifying records, such as updating a user's details or editing content in an existing post.

How PUT Works: Using requests.put(), we specify the resource URL and send updated data in JSON format within the request body. The server processes this request, replaces the old data with the new data, and responds with the updated resource details.

Common Use Case: PUT is ideal for updates, such as editing user information (name, email) or changing details in a specific item.

python

```python
url = "https://jsonplaceholder.typicode.com/posts/1"
data = {
    "title": "Updated Post",
    "body": "This is an updated post."
}

response = requests.put(url, json=data)

print(response.status_code)
print(response.json())
```

Python - Making DELETE requests

The **DELETE** method removes a specified resource from the server, permanently deleting it. DELETE requests are typically restricted to users with higher permissions, as deletion is irreversible.

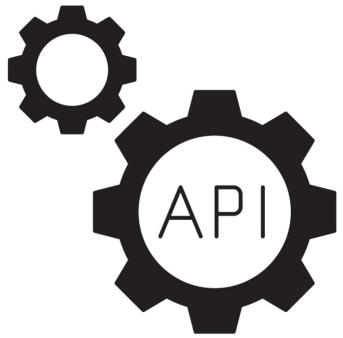
How DELETE Works: With requests.delete(), we provide the URL of the resource to delete, and the server removes it if found. DELETE is helpful for managing data when we need to remove outdated or incorrect entries.

Common Use Case: Deleting user accounts, removing posts, or clearing specific entries in a database.

python

```
url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.delete(url)

print(response.status_code)
```

Python HTTP requests



Let's discuss making a simple `GET` request using 5 of Python's most popular requests-related packages.

- requests - Easily the most popular package for making requests using Python
- urllib3 - Not to be confused with urllib, which is part of the Python standard library
- httplib2 - Fills some of the gaps left by other libraries
- httpx - A newer package that offers HTTP/2 and asynchronous requests

Python Requests module

The requests package for Python is so popular that it's currently a requirement in more than 1 million GitHub repositories and has had nearly 600 contributors to its code base over the years!

The package's clear and concise documentation is almost certainly responsible for its widespread use.

http_requests.py

```
import json
import os
import webbrowser

import requests


nasa_api_key = os.environ.get('NASA_API_KEY')
api_url = 'https://api.nasa.gov/planetary/apod?api_key=
{}'.format(nasa_api_key)

def use_requests(api_url):

    response = requests.get(api_url)
    json_response = json.loads(response.text)
    photo_url = json_response['url']
    webbrowser.open_new_tab(photo_url)

    return

use_requests(api_url)
```

Python urllib3 module

Despite the similarity in names, `urllib3` is a 3rd-party package and is completely different from `urllib`, which is part of the Python standard library.

There are more than 650,000 GitHub repositories that list `urllib3` as a requirement which make it a massively popular alternative to the Requests library.

urllib3_requests.py

```
import json
import os
import webbrowser

import urllib3


nasa_api_key = os.environ.get('NASA_API_KEY')
api_url = 'https://api.nasa.gov/planetary/apod?api_key={}'.format(nasa_api_key)

def use_urllib3(api_url):

    http = urllib3.PoolManager()
    response = http.request('GET', api_url)
    json_response = json.loads(response.data)
    photo_url = json_response['url']
    webbrowser.open_new_tab(photo_url)

    return

use_urllib3(api_url)
```

Python httplib2 module

While not as popular as the first two modules, httplib2 provides additional features like:

- Persistent connections
- Caching

httplib2_requests.py

```
import json
import os
import webbrowser

import httplib2


nasa_api_key = os.environ.get('NASA_API_KEY')
api_url = 'https://api.nasa.gov/planetary/apod?api_key={}'.format(nasa_api_key)

def use_httplib2(api_url):

    http = httplib2.Http()

    # The response is sent as a 2-item tuple, with the content at index 1
    response = http.request(api_url)
    json_response = json.loads(response[1])
    photo_url = json_response['url']
    webbrowser.open_new_tab(photo_url)

    return

use_httplib2(api_url)
```

Python httpx module

The httpx Python package is the newest one on our list. It supports the HTTP/2 protocol and asynchronous requests in addition to the standard synchronous HTTP/1 protocol. It's built to be very "Requests-like" and mirrors a lot of the code patterns present in the Requests package.

httpx_requests.py

```
import json
import os
import webbrowser

import httpx


nasa_api_key = os.environ.get('NASA_API_KEY')
api_url = 'https://api.nasa.gov/planetary/apod?api_key={}'.format(nasa_api_key)

def use_httpx(api_url):

    request = httpx.get(api_url)
    json_request = request.json()
    photo_url = json_request['url']
    webbrowser.open_new_tab(photo_url)

    return

use_httpx(api_url)
```

Python – handling request headers

Headers allow us to add metadata to requests, which can include authentication tokens or specify data formats. Headers help the server interpret the request correctly and manage secure access.

How They Work: Headers are added to requests to specify details, such as authorization credentials or the expected data format. Without headers, secure endpoints may reject requests due to missing information.

Types of Headers:

- **Authorization:** Provides access control (e.g., Bearer <token>).
- **Content-Type:** Tells the server about the format of the data being sent, often set to application/json.

python

```python
url = "https://jsonplaceholder.typicode.com/posts"
headers = {
    "Authorization": "Bearer your_token_here",
    "Content-Type": "application/json"
}

response = requests.get(url, headers=headers)

print(response.status_code)
```

Python - handling response headers

Response headers offer information about the server's response, like the content type and caching details. These headers help interpret the response and understand how the data should be handled or stored.

How They Work: Accessing response.headers provides a dictionary of response headers, enabling us to inspect specific information such as the data format and response time.

Common Headers:

- **Content-Type:** Indicates the format of the response data, like application/json.
- **Date:** Shows the exact date and time when the server generated the response.

```python
url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.get(url)

print(response.headers)
print(response.headers['Content-Type'])
```

Python - parsing json responses

JSON is the most common format for API responses, which can be easily converted into Python dictionaries or lists. This parsing process enables structured data access and manipulation within Python.

How JSON Parsing Works: Using *response.json()*, JSON data is parsed into Python objects, making it easy to access specific elements like user profiles or item lists. This process enables applications to utilize the returned data in a meaningful way.

Common Use Case: Accessing nested data fields, such as product names, user details, or blog post contents.

python

```python
url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.get(url)

data = response.json()
print(data["title"])
```

Python - example workflow

Using Multiple HTTP Methods: In a real-world application, we often use multiple HTTP methods to perform different actions on resources. For instance, a full workflow might include retrieving a resource, updating it, and then deleting it when it's no longer needed.

This combination of **GET**, **PUT**, and **DELETE** actions allows for full **CRUD** (Create, Read, Update, Delete) functionality, essential for data management applications.

Common Use Case: Complete data operations, such as viewing, modifying, and deleting a user's data in a user management system.

python

```python
# Step 1: GET
url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.get(url)
print("GET:", response.json())

# Step 2: PUT (Update)
data = {"title": "Updated Title"}
response = requests.put(url, json=data)
print("PUT:", response.json())

# Step 3: DELETE
response = requests.delete(url)
print("DELETE Status:", response.status
```

API Retry Logic and Error Handling

Develop a passion for learning.

Introduction

Retry Logic: In API calls, network issues or temporary server errors can cause requests to fail. Retry logic helps automatically resend requests a few times, improving resilience without requiring manual intervention.

Error Handling: Error handling ensures that your application responds appropriately to different kinds of issues, such as connectivity problems, server errors, or invalid responses. Proper error handling prevents the application from crashing and provides feedback.

Common Errors:

- **Timeout:** The server takes too long to respond.
- **Connection Error:** Network issues prevent reaching the API.
- **5xx Errors:** Server-side issues, indicating that the problem is temporary.

Implementing Basic Retry Logic

Using a Loop for Retries: A common way to implement retry logic is by using a loop that will retry the request a set number of times if an error occurs. You can add a delay between retries to give the server time to recover.

Setting Retry Limits: Define a maximum number of retries to avoid endless loops and potential overload. Also, use *time.sleep()* to introduce a short pause between retries.

python

```python
import requests
import time

url = "https://jsonplaceholder.typicode.com/posts/1"
max_retries = 3
retry_count = 0

while retry_count < max_retries:
    try:
        response = requests.get(url)
        if response.status_code == 200:
            print(response.json())
            break
    except requests.exceptions.RequestException:
        retry_count += 1
        print(f"Retrying {retry_count}/{max_retries}")
        time.sleep(2)
```

Retry Decorators

Using a Decorator for Retry Logic: A decorator can be used to wrap API calls and automatically retry them in case of failure. This approach keeps retry logic organized and reusable across multiple API calls.

Handling Specific Exceptions: Use try/except blocks within the decorator to catch specific exceptions, such as *requests.exceptions.Timeout* or *requests.exceptions.ConnectionError*.

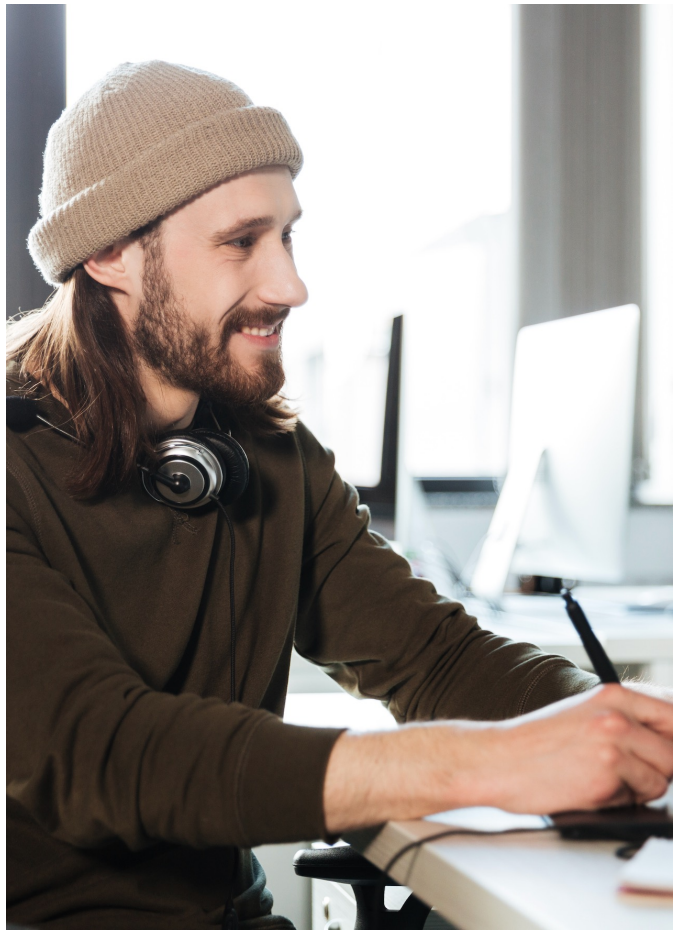
python

```python
from functools import wraps
import requests
import time

def retry(max_retries=3, delay=2):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_retries):
                try:
                    return func(*args, **kwargs)
                except requests.exceptions.RequestException:
                    print(f"Retry {attempt + 1}/{max_retries}")
                    time.sleep(delay)
            return None
        return wrapper
    return decorator

@retry(max_retries=3, delay=2)
def get_data(url):
    response = requests.get(url)
    response.raise_for_status()
    return response.json()

url = "https://jsonplaceholder.typicode.com/posts/1"
print(get_data(url))
```

Asynchronous Programming

Develop a passion for learning.

Asynchronous Programming And Asyncio



Purpose of Asynchronous Programming: Asynchronous programming allows tasks to run concurrently, making applications more efficient by not waiting for each task to finish before moving to the next. This is especially useful for tasks that involve waiting, such as API calls or file operations.

What is asyncio: asyncio is a Python library for writing concurrent code using the async and await keywords. It allows functions to run asynchronously, making it easier to handle multiple tasks at once without blocking other operations.

Core Concepts:

- **Coroutines:** Functions defined with async def that can be paused and resumed.
- **Event Loop:** Manages and schedules coroutines, running them concurrently.

Creating And Running Async Functions

Using async and await for Asynchronous Code: To define an asynchronous function, use async def instead of def. This designates the function as a coroutine, allowing it to pause execution to wait for other tasks.

Using await in Async Functions: The await keyword pauses the coroutine to wait for the result of another coroutine. While waiting, other tasks can execute, making it ideal for I/O-bound tasks like network calls or file operations.

Running an Async Function with asyncio.run(): To start an async function, use *asyncio.run()*, which runs the event loop and executes the coroutine. This is the recommended way to start an async function from a synchronous context, ensuring the function and its awaited tasks are handled by the event loop.

python

```python
import asyncio

async def greet():
    print("Hello...")
    await asyncio.sleep(1)
    print("...World!")

asyncio.run(greet())
```

Running Multiple Async Tasks Concurrently

Concurrent Execution with *asyncio.gather():* *asyncio.gather()* is used to run multiple coroutines concurrently. When passed multiple coroutines, it schedules them to run at the same time, allowing Python to handle multiple tasks efficiently. This can lead to significant time savings, especially with I/O-heavy tasks, as they can wait simultaneously instead of sequentially.

Benefits of Concurrent Execution: By running tasks concurrently, applications can complete multiple operations in the same amount of time it would take to complete just one if each ran separately. This is especially useful for handling tasks that involve waiting, like downloading data from several APIs, reading multiple files, or handling other network tasks.

python

```python
import asyncio

async def task(name, delay):
    print(f"Starting {name}")
    await asyncio.sleep(delay)
    print(f"Completed {name}")

async def main():
    await asyncio.gather(
        task("Task 1", 1),
        task("Task 2", 2),
        task("Task 3", 1)
    )

asyncio.run(main())
```

Lab: Interacting with APIs

Configuration Management

Configuration Drift

POTHOLE

- Your infrastructure requirements change
- Configuration of a server falls out of policy

- Manage with Infrastructure as Code (IAC)
  - Terraform
  - Ansible
  - Chef
  - Puppet

Manual

Common manual tasks

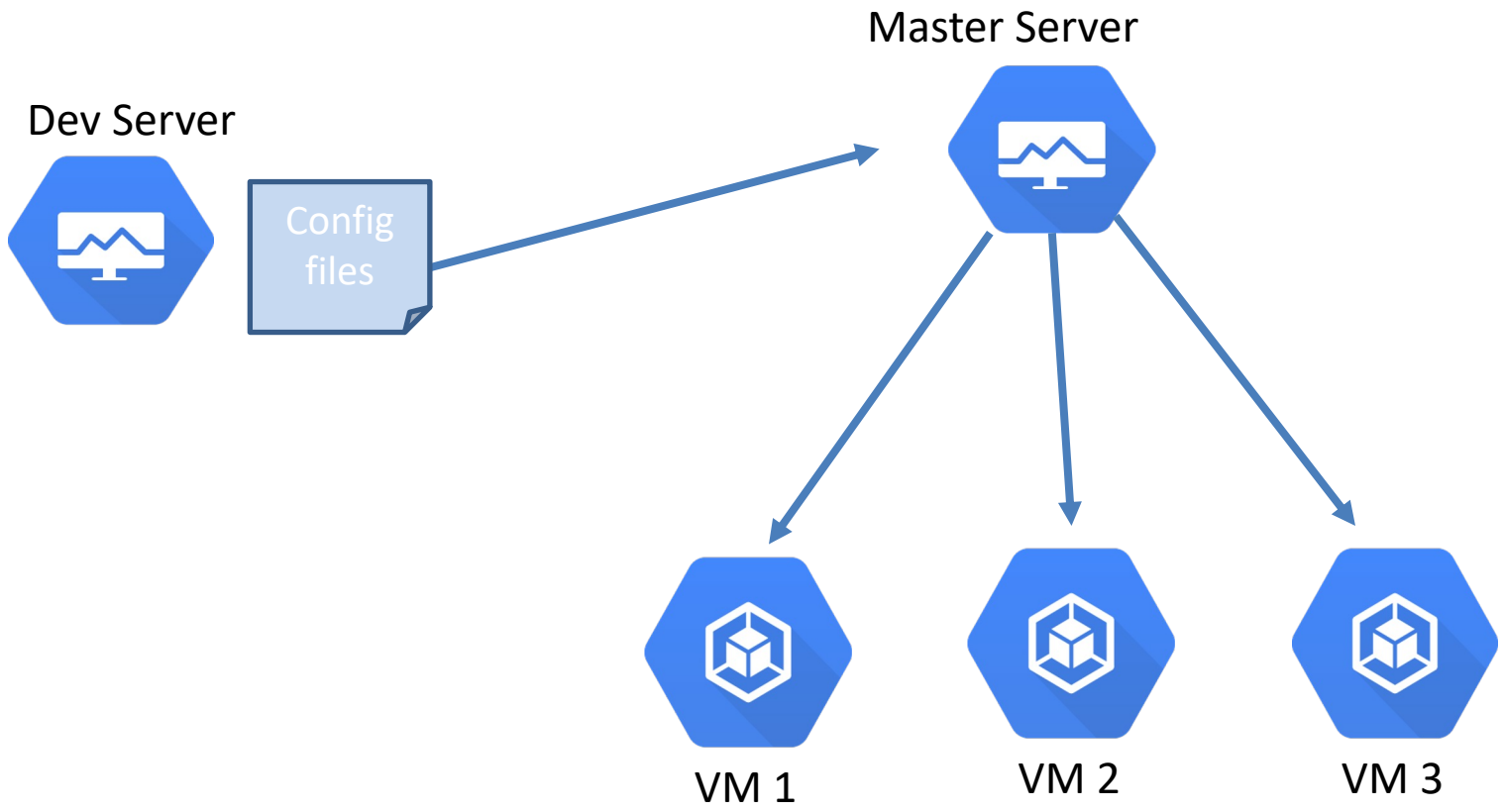
Installation and configuration:

- Operating System
- Runtime
- Libraries
- Utilities
- Configuration files

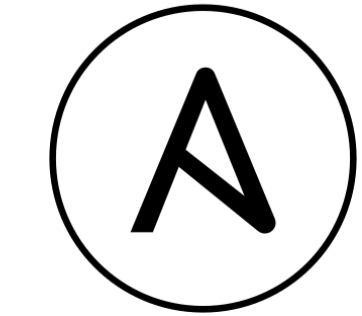
- Build application
- Test application
- Deploy application

Automated

Automated



Automation tools



ANSIBLE

CHEF

puppet

HashiCorp Terraform

Infrastructure as Code

What is IaC?

Infrastructure as code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files
Used with bare-metal as well as virtual machines and many other resources.
Normally, a declarative approach

Infrastructure as Code

- Programmatically provision and configure components
- Treat like any other code base
  - Version control
  - Automated testing
  - data backup

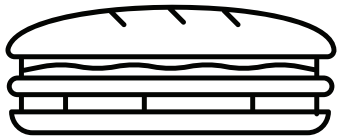
Infrastructure as Code

Provisioning infrastructure through software to achieve consistent and predictable environments.

Core Concepts

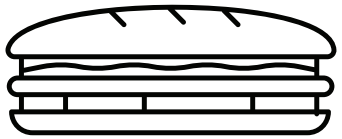
- Defined in code
- Stored in source control
- Declarative or imperative

Imperative



```
# Make a sandwich
get bread
get mayo
get turkey
get lettuce

spread mayo on bread
put lettuce in between bread
put turkey in between bread
on top of turkey
```

Declarative



```
# Make a sandwich
food sandwich "turkey" {
  ingredients = [
    "bread", "turkey",
"mayo", "lettuce"
  ]
}
```

POP QUIZ:

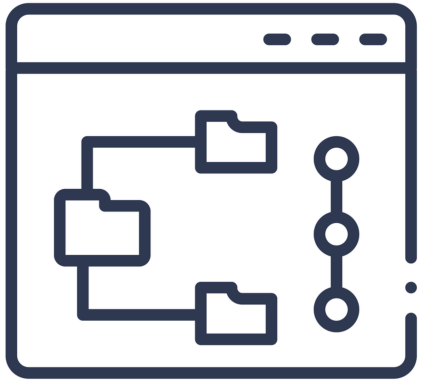
What challenges does Infrastructure as Code solve?

Demo: Automation Tools

Terraform Introduction

Automating Infrastructure

Provisioning resources

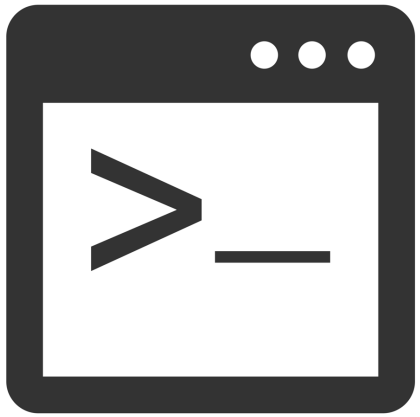


Version Control

Plan Updates

Reusable Templates

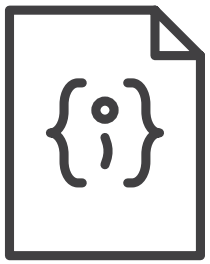
Terraform components



Terraform executable

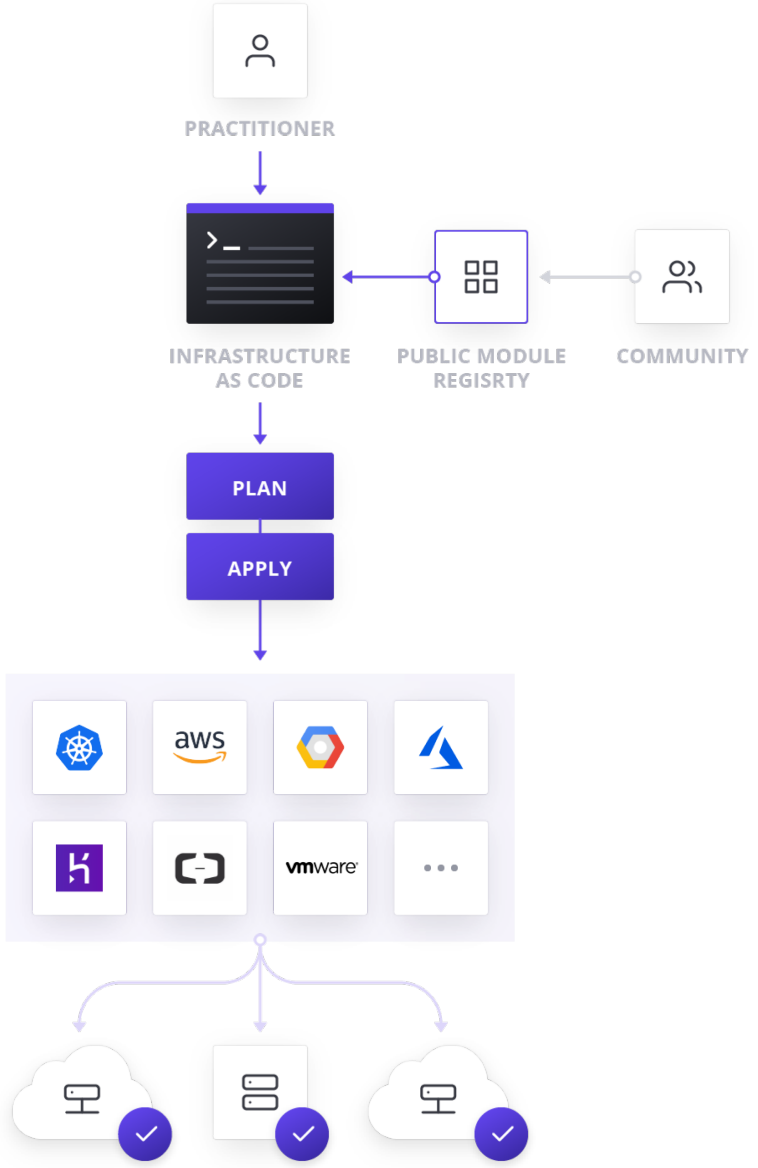
Terraform files

Terraform plugins



Terraform state

Terraform architecture



POP QUIZ:

What is Terraform used for?

Terraform CLI

Run `terraform` command

Output:

```
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform
interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    get                Download and install modules for the
configuration
    graph              Create a visual graph of Terraform
resources
```

Terraform CLI

Command:

```
terraform init
```

Output:

```
Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "docker"...
```

Terraform fetches any required providers and modules and stores them in the .terraform directory. Check it out and you'll see a plugins folder.

Terraform CLI

Command:

```
terraform validate
```

Validate all of the Terraform files in the current directory. Validation includes basic syntax check as well as all variables declared in the configuration are specified.

Terraform CLI

Command:

```
terraform plan
```

Output:

```
Terraform will perform the following actions:

  + aws_instance.aws-k8s-master
      id:                                   <computed>
      ami:                                  "ami-01b45..."
      instance_type:                        "t3.small"
```

Plan is used to show what Terraform will do if applied. It is a dry run and does not make any changes.

Terraform CLI

Command:

```
terraform apply
```

Output:

```
❯ terraform apply "rapid-app.out"
aws_security_group.k8s_sg: Creating...
  arn:                                      "" => "<computed>"
  description:                              "" => "Allow all
inbound traffic necessary for k8s"
  egress.#:                                 "" => "1"
  egress.482069346.cidr_blocks.#:           "" => "1"
  egress.482069346.cidr_blocks.0:           "" => "0.0.0.0/0"
```

Performs the actions defined in the plan

Terraform CLI

Command:

```
terraform destroy [-auto-approve]
```

Destroys all the resources in the state file.
-auto-approve (don't prompt for confirmation)

Lab: Setup AWS environment