

Data Serialization

Data Serialization

We saw in the previous section that the main difference between a Web API and non-web API is the data transfer part, which in Web API happens over a network

Web API involves receiving a request, doing the processing and sending the response back

Both request and response contains information! And it is obviously natural that the information should be in a language that both systems understand

In Web API and services, data serialization plays a fundamental role in the interaction between the client and the server

Data Serialization

When a Web API exchanges data, both the client and the server need a **common format** to interpret the transmitted data accurately

It might seem strange, but computers are not smart enough to communicate in free-flowing languages like we humans do

And therefore a simpler, more focussed and rule-based languages are created in which computers can talk to each other and pass information

There are numerous languages that exist today that we can choose our systems to talk in. Like XML, JSON, YAML, CSV, etc

Data Serialization

With time the languages have also evolved with XML being the first, then JSON and lately YAML

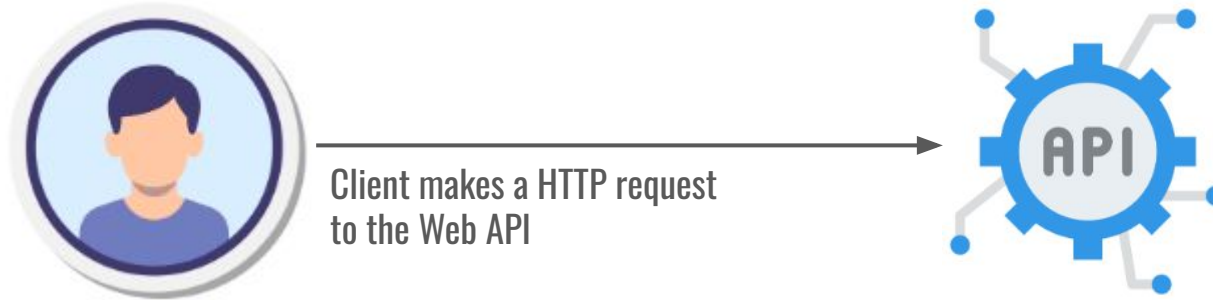
But what is data serialisation?

It's a process of converting language specific data structure to **language agnostic data formats** so that data can be exchanged between any two language (python, java, go, javascript, etc) easily

Remember our REST API, just before sending our response to client we did - jsonify (data)
To convert Python **dictionary** (language specific data format) to **JSON** (language agnostic data format)

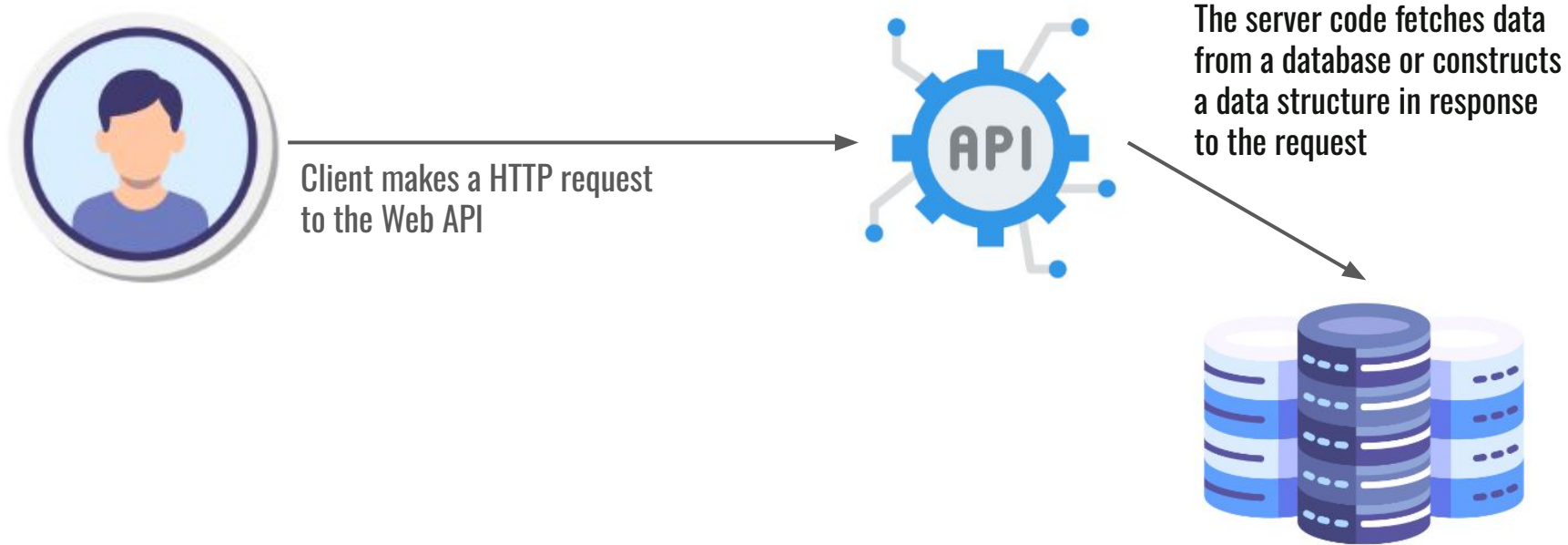
Steps Involved in Data Serialization

Server Side



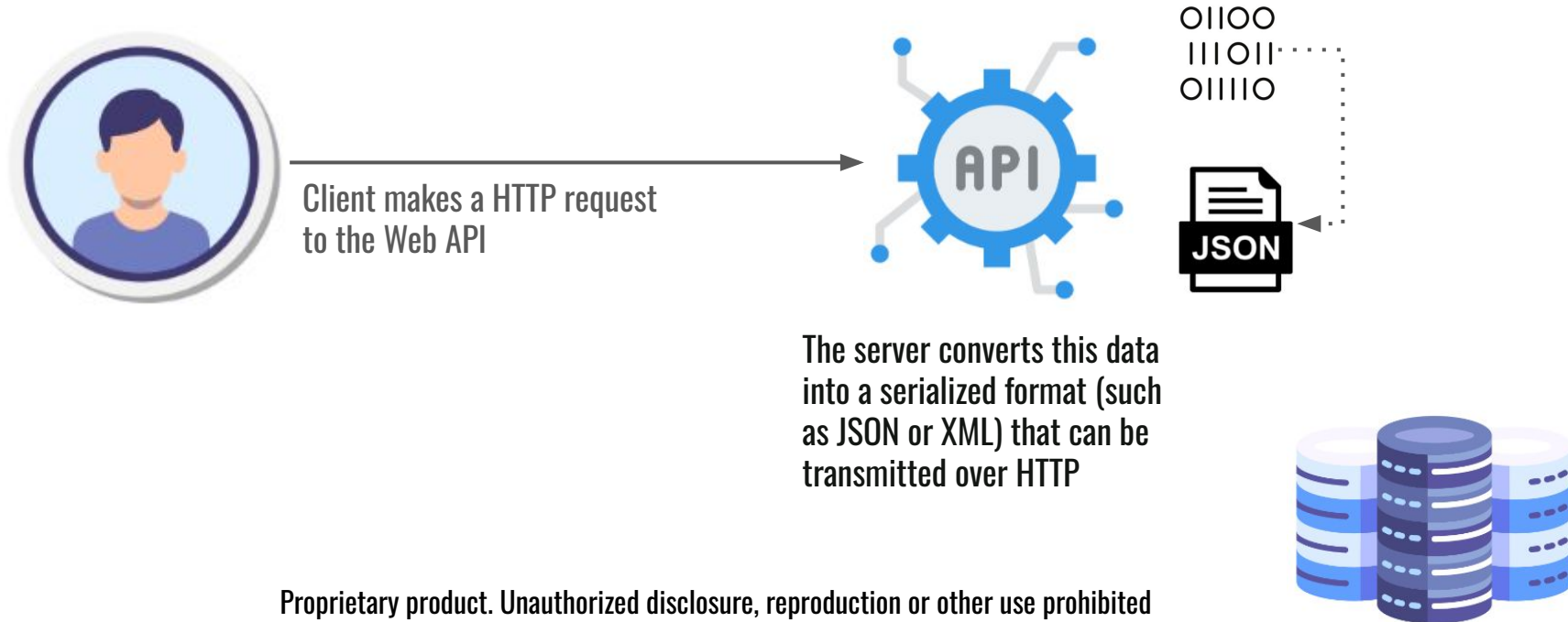
Steps Involved in Data Serialization

Server Side: Creating/Fetching the Data



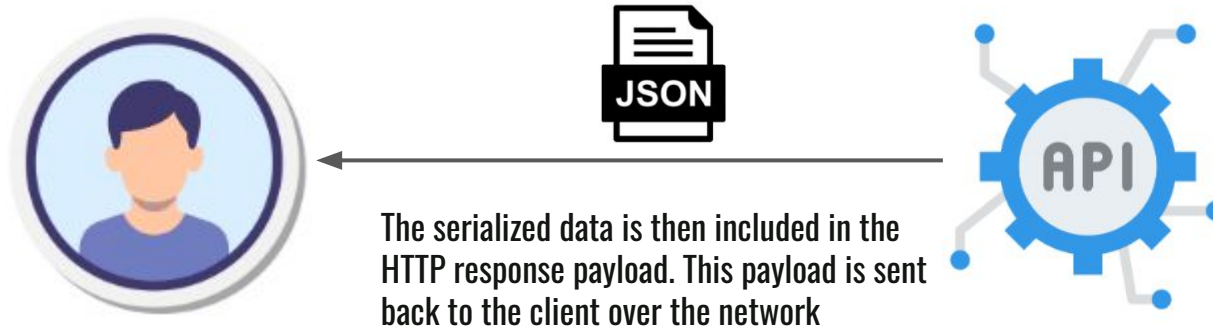
Steps Involved in Data Serialization

Server Side: Serialization Process



Steps Involved in Data Serialization

Server Side: Sending Serialized Data



Steps Involved in Data Serialization

Client Side: Data Deserialization



When the client receives the HTTP response from the server, it must convert the serialized data back into a usable form



JSON: JavaScript Object Notation

Early Beginnings

The origins of JSON can be traced back to the early days of the web, specifically around the late 1990s and early 2000s

During this era, web applications were becoming increasingly complex, needing more sophisticated ways to exchange data between the client (browser) and the server

The primary method for data exchange at the time was the XML (eXtensible Markup Language) which had some challenges in use

Emergence of JSON

JSON was first described by Douglas Crockford in the early 2000s

The motivation behind JSON was to provide a more straightforward, more human-readable form for data interchange, specifically to work well with web technologies

Since the JSON derives a lot of features from JavaScript, it gels particularly well with web applications which are mostly written in JavaScript

JSON's lightweight nature made it perfect for mobile apps and single-page applications (SPAs), which require efficient data transfer between the client and server

It became a de facto standard for APIs, especially with the rise of RESTful (Representational State Transfer) architecture

Introduction

The JavaScript Object Notation (JSON) data format enables applications to communicate over a network, typically through RESTful APIs

JSON is technology-agnostic, nonproprietary, and portable

All modern languages (e.g., Java, JavaScript, Ruby, C#, PHP, Python, and Groovy) and platforms provide excellent support for producing (serializing) and consuming (deserializing) JSON data

JSON is not limited to Representational State Transfer (REST); it also works with nodeJS, JSON databases like MongoDB and Redis, Messaging platforms like Kafka

A Brief Sample

Before we go further, let's look at a small JSON sample

```
{ "thisIs": "My first JSON document" }
```

A valid JSON document can be either of the following:

An Object surrounded by curly braces, { and }

An Array enclosed by brackets, [and]

A Brief Sample

```
{ "thisIs": "My first JSON document" }
```

Overall its called an object that contains a single key-value pair where the key is “thisIs” and the value is “My First JSON document”

Just to keep us honest, let’s validate this document by using [JSONLint](#).
Just paste the text into the text area, click the Validate button, and you should see the “JSON is valid” message

Second Sample

```
[  
  "also",  
  "a",  
  "valid",  
  "JSON",  
  "doc"  
]
```

You can validate this in JSONLint as well

But we're getting ahead of ourselves. We'll cover JSON syntax more thoroughly after few slides. First let's understand "Why JSON"?

Why JSON?

Reason 1	The explosive growth of RESTful APIs based on JSON
Reason 2	The simplicity of JSON's basic data structures
Reason 3	The increasing popularity of JavaScript
Reason 4	Small size compared to XML
Reason 5	Better readability compared to XML

Core JSON

The Core JSON data format includes JSON Data and Value Types. We'll also cover versions, comments, and File/MIME Types

JSON Data Types

Simple key value pair

Values as object

Values are array or list

Core JSON: Simple Name-Value Pairs

```
{  
  "conference": "OSCON",  
  "speechTitle": "JSON at Work",  
  "track": "Web APIs"  
}
```

Name/value pairs have the following characteristics:

Each name (e.g., "conference")

Is on the left side of the colon (:)

Is a String, and must be surrounded by double quotes

The value (e.g., "OSCON") is to the right of the colon. In the preceding example, the value type is a String, but there are several other Value Types

Core JSON: **Objects as Values**

```
{  
  "address" : {  
    "line1" : "555 Any Street",  
    "city" : "Denver",  
    "stateOrProvince" : "CO",  
    "zipOrPostalCode" : "80202",  
    "country" : "USA"  
  }  
}
```

```
{  
  "speaker" : {  
    "firstName": "Larson",  
    "lastName": "Richard",  
    "topics": [ "JSON", "REST", "SOA" ]  
  }  
}
```

Core JSON: **Objects as Values**

```
{
  "speaker" : {
    "firstName": "Larson",
    "lastName": "Richard",
    "topics": [ "JSON", "REST", "SOA" ],
    "address" : {
      "line1" : "555 Any Street",
      "city" : "Denver",
      "stateOrProvince" : "CO",
      "zipOrPostalCode" : "80202",
      "country" : "USA"
    }
  }
}
```

Objects are enclosed within a beginning left curly brace { and an ending right curly brace }

Consist of comma-separated, unordered, name/value pairs

Can be empty, { }

Can be nested within other Objects or Arrays

Core JSON: **Arrays as Values**

```
{
  "presentations": [
    {
      "title": "JSON at Work: Overview and Ecosystem",
      "length": "90 minutes",
      "abstract": [ "JSON is more than just a simple replacement for XML when",
                    "you make an AJAX call." ],
      "track": "Web APIs"
    },
    {
      "title": "RESTful Security at Work",
      "length": "90 minutes",
      "abstract": [ "You've been working with RESTful Web Services for a few years",
                    "now, and you'd like to know if your services are secure." ],
      "track": "Web APIs"
    }
  ]
}
```

Are enclosed within a beginning left brace [and an ending right brace]

Consist of comma-separated, **ordered** values

Can be empty, []

Can be nested within other Arrays or Objects

Have indexing that begins at 0

JSON Simple Value Types

JSON Value Types represent the Data Types that occur on the right hand side of the colon (:) of a Name/Value Pair. JSON Value Types include the following:

object

boolean

array

null

string

string

JSON Value Types: **Strings**

Strings consist of zero or more characters enclosed in quotation marks " "

Strings wrapped in single quotes (') are not valid

```
[  
  "fred",  
  "fred\t",  
  "\b",  
  "",  
  "\t",  
  "\u004A"  
]
```


JSON Value Types: **Numbers**

Numbers are always in base 10 (only digits 0–9 are allowed) with no leading zeros

Numbers can have a fractional part that starts with a decimal point (.)

Numbers can have an exponent of 10, which is represented with the e or E notation

Octal and hexadecimal formats are not supported

Unlike JavaScript, numbers can't have a value of NaN or infinity

```
{  
  "age": 29,  
  "cost": 299.99,  
  "temperature": -10.5,  
  "unitCost": 0.2,  
  "speedOfLight": 1.23e11,  
  "speedOfLight2": 1.23e+11,  
  "avogadro": 6.023E23,  
  "avogadro2": 6.023E+23,  
  "oneHundredth": 10e-3,  
  "oneTenth": 10E-2  
}
```

JSON Value Types: **Boolean**

Booleans can have a value of only true or false

The true or false value on the right hand side of the colon(:) is not surrounded by quotes

```
{  
  "isRegistered": true,  
  "emailValidated": false  
}
```

JSON Value Types: **null**

Null values are not not surrounded by quotes

Indicate that a key/property has no value (Act as a placeholder)

```
{
  "address": {
    "line1": "555 Any Street",
    "line2": null,
    "city": "Denver",
    "stateOrProvince": "CO",
    "zipOrPostalCode": "80202",
    "country": "USA"
  }
}
```

JSON Style Guidelines

According to the core JSON specification, .json is the standard JSON file type when storing JSON data on filesystems

Property Names (key name) are on the left side of the colon in a name/value pair (and Property Values are on the right hand side of the hyphen). Two main styles can be used to format a JSON Property Name

lowerCamelCase

snake_case

Indentation is just for human users like you and me, for inter-system communication like APIs - indentation doesn't matter and they are not applied

Assignment

JSON Data Modeling

[Link](#)

Stub/Mock Servers

Stub/Mock Servers

In any API design - there are two parties, Clients and Server

There are two main coding part on the client side.

Making a request -> Wait for response -> **Process the response**

Even though we have tried to keep our Payments API relatively straightforward, how many days did we take to implement it?

Generally speaking, the development of Server side is more complex as we have to take care of core business logic, security, scaling under load, traffic distribution, monitoring, logging, alerting, architecting the solution, etc

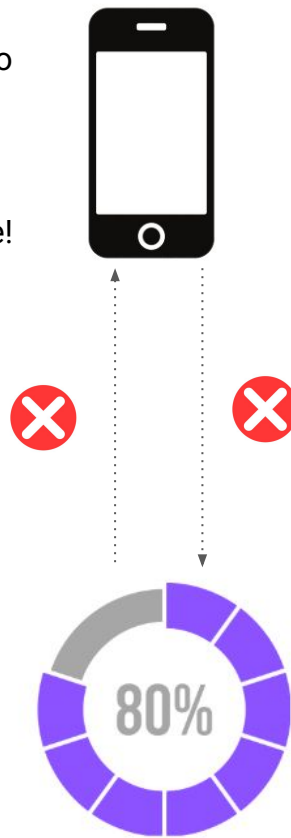
Our Journey So Far

Day 1	Jobs to be done (JTBD) - to document high level feature requirements
Day 2	Basic API setup, routes and development
Day 3	Database choice, design, caching, filtration, pagination
Day 4	Microservice creation, Routing, Load Balancing

What will the client do for so many days?

They can't make a request!

They can't get any response!



Architectural Style—noBackend

With noBackend, the developer doesn't have to worry about the nuts and bolts of application servers or databases at the early stages of application development

We maintain focus on our application from a business perspective (services and data first) so that we can support APIs

We'll deploy JSON data with simple tools such as json-server to emulate a RESTful API

Why This Approach?

By using this approach, we take an interface-first approach to designing and building an API, which provides the following

1

More Agile, rapid, iterative client development due to the decoupling from the backend

2

Faster feedback on the API itself. Get the data and URI out there quickly for rapid review

3

A separation of concerns between the Resource b/w client and API and its (eventual) internal implementation (e.g., application server, business logic, and data store). This makes it easier to change implementation in the future. If you create and deploy a real API with [Node.js/Rails/Java](#) (or other framework) too early, you've already made design decisions at a very early stage that will make it difficult to change after you start working with API Consumers

What does Stub API do?

1

Eliminates the initial need to work with servers and databases

2

Allows API Producers (those developers who write the API) to focus on API Design, how best to present the data to the Consumers, and initial testing

3

Enables API Consumers to work with the API at an early stage and provide feedback to the API development team

By using the lightweight tools in this book, you'll see that you can go a long way before writing code and deploying it on a server. Of course, you'll eventually need to implement an API

JSON Generator

We want to generate lots of test data quickly

Test data can be tricky because of the data volume needed to do any meaningful testing

By hand it will take a great deal of effort to create the volume of test data we're looking for

We need another tool to help create the data we need to create our first version of the API, and that's where JSON Generator comes in

Hands On

Let's create a transaction JSON data for UPI

GET /payments

POST /payments

GET /payments?status=success

GET /payments/<txn_id>

PATCH /payments/<txn_id>

DELETE /payments/<txn_id>

[Steps](#)

YAML: Yaml Ain't Markup Language

Proprietary product. Unauthorized disclosure, reproduction or other use prohibited

YAML

~~Yet Another Markup Language~~

YAML Ain't Markup Language

What is YAML?

YAML is a light weight, human readable data serialization language. It is primarily designed to make the format easy to read while including advanced features.

YAML stands for “YAML Ain't Markup Language”

It is similar to XML and JSON files but uses a more minimalist syntax even while maintaining similar capabilities

YAML files are created with extensions “.yaml ” or yml ” . You can use any IDE or text editor to open/create YAML files

It is very easy and simple for represent complex mapping. Due to which it is heavily used in configuration settings

Serialisation Evolution



Highly Verbose
Large Size
Poor Readability



Fairly Verbose
Moderate Size
High Readability



Least Verbose
Lowest Size
High Readability

Let's Compare

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <chapter>data-serialisation</chapter>
  <data>payment-history</data>
  <transactions>
    <mode>online</mode>
    <name>upi</name>
    <amount>100</amount>
    <status>success</status>
  </transactions>
  <transactions>
    <mode>online</mode>
    <name>imps</name>
    <amount>200</amount>
    <status>success</status>
  </transactions>
</root>
```

388 Bytes

```
{
  "chapter": "data-serialisation",
  "data": "payment-history",
  "transactions": [
    {
      "mode": "online",
      "name": "upi",
      "amount": 100,
      "status": "success"
    },
    {
      "mode": "online",
      "name": "imps",
      "amount": 200,
      "status": "success"
    }
  ]
}
```

313 Bytes (20% Less)

```
chapter: data-serialisation
data: payment-history
transactions:
- mode: online
  name: upi
  amount: 100
  status: success
- mode: online
  name: imps
  amount: 200
  status: success
```

194 Bytes (50% Less)

How is the readability compare? Verbosity?

The Numbers

When you see these numbers in Bytes - it might seem inconsequential. But if you put them in context of your organization, it can result in huge improvement

On an average if UPI has 40 crore transactions a day with each transaction size as show in previous slide

XML: 155 GB

JSON: 125 GB

YAML: 75GB

Our goal should always be to make payload as smaller as possible (without compromising on other aspects like legibility, readability, etc)

History of YAML

Clark Evans, Ingy döt Net, and Oren Ben-Kiki created the YAML specification in 2001

In early 2004, the first official version of the YAML 1.0 specification was released

In 2005, the YAML 1.1 standard was released with an enhancement to support boolean values as 'yes'-'no', 'y'-'n', 'on'-'off'

YAML 1.2 was released in 2009 which allowed json to be part of YAML

YAML 1.2.2 is the latest revision of the YAML specification. It was released on 1st October 2021

YAML Syntax: Key-Value Pairs

A key-value pair is a simple data structure that consists of a unique identifier (the key) and the corresponding value of that identifier

The key can be any data type, such as a text string or an integer

The value can also be of any type of data, including string, integer, float, boolean, list, or key-value pairs



YAML Syntax: Key-Value Pairs

Key-value pairs are represented using the following syntax

<key>: **<value>**


where <key> represents key name and <value> represents data separated by colon (:). The whitespace after the colon is mandatory

transaction_id	amount	status
1efaa	510	success

transaction_id: 1efaa
amount: 510
status: success

YAML Syntax: Key-Value Pairs

transaction_id	amount	status
1efaa	510	success



```
transaction_id: 1efaa
amount: 510
status: success
```

The whitespace is very important; without this, the YAML will be invalid

For a text or a string, you may use quotation marks, a single quote (') or a double quote ("), or no quotes at all

In YAML, we use white spaces for defining the structure. White Spaces provide structure to the document (we will see this later)

You should always be careful that you use only white spaces, and in case you accidentally add tabs, the YAML will become invalid sometimes

Comparison

transaction_id	amount	status
1efaa	510	success

```
<? xml version="1.0?">
  <transaction>
    <id>1efaa</id>
    <amount>510</amount>
    <status>success</status>
  </transaction>
```

```
{
  "transaction_id": "1efaa",
  "amount": 510,
  "status": "success"
}
```

```
transaction_id: 1efaa
amount: 510
status: success
```


YAML Syntax: Lists

A list is represented by preceding its items with - (hyphen). It's an ordered collection of data

Let's say you want to have attributes of a transaction in list: [1effab, 200,failed,2024-07-24]

In YAML, you would write it like:

- 1effab
- 200
- failed
- 2024-07-24

Also note, YAML is case sensitive, meaning 1effab and 1EFFAB are not the same thing

YAML Multi Document Support

To define a YAML file, we use .yaml or .yml extensions, e.g. config.yaml

You can add multiple documents to a single YAML file

We begin a YAML document with three hyphens (---). This is optional

To mark end of one piece of document and start another piece we add three hyphens (---) between them

Once you have added all your data, in the end you have to add 3 dots (...)
Three dots tells the parser that the data has ended

YAML Multi Document Support

Let's say I want to represent two lists. First lists tell about available data formats and 2nd list tells us different kinds of APIs

```
---  
- xml  
- json  
- yaml  
---  
- rest  
- rpc  
- graphql  
...
```

Sometimes, these YAML files containing one or more documents are also called YAML streams

If there is single document then we don't have to add (...) in the end

YAML Simple Key Value Pair: Practice

```
{  
  "amount": 500,  
  "payer_id": "abc@hdfc",  
  "payee_id": "xyz@idfc",  
  "status": "success",  
  "date": "2024-07-11",  
  "time": 45909,  
  "app": "gpay",  
  "platform": "android"  
}
```

JSON



??

YAML

YAML Simple Key Value Pair: Practice

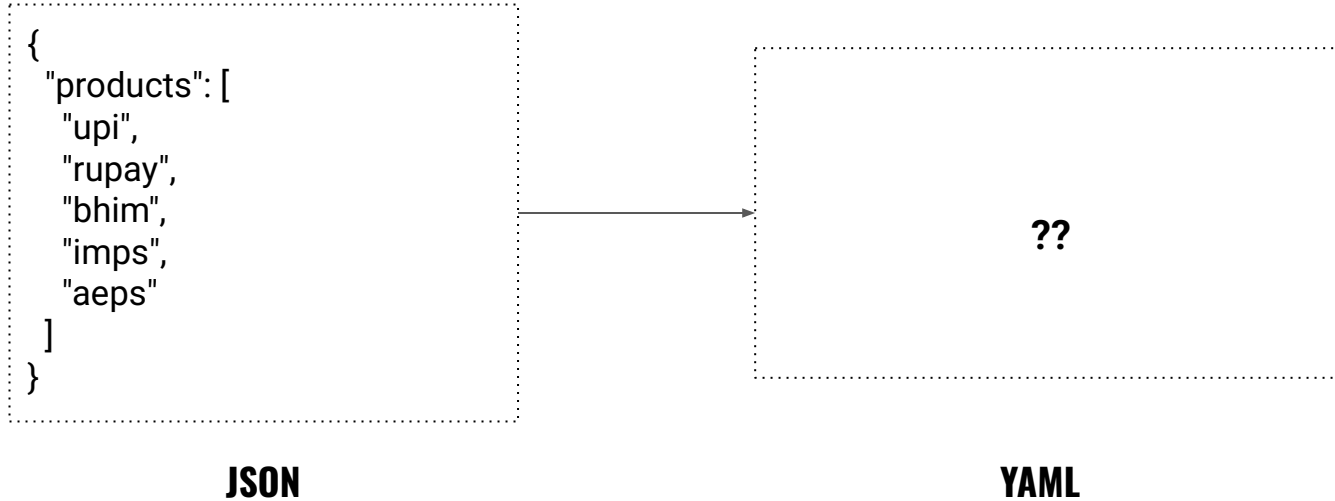
```
{  
  "amount": 500,  
  "payer_id": "abc@hdfc",  
  "payee_id": "xyz@idfc",  
  "status": "success",  
  "date": "2024-07-11",  
  "time": 12:45:09,  
  "app": "gpay",  
  "platform": "android"  
}
```

JSON

```
amount: 500  
payer_id: abc@hdfc  
payee_id: xyz@idfc  
status: success  
date: 2024-07-11  
time: 12:45:09  
app: gpay  
platform: android
```

YAML

YAML List: Practice



YAML List: Practice

```
{  
  "products": [  
    "upi",  
    "rupay",  
    "bhim",  
    "imps",  
    "aeps"  
  ]  
}
```

JSON

```
products:  
- upi  
- rupay  
- bhim  
- imps  
- aeps
```

YAML

Complex YAML: Practice

```
{  
  "users": [  
    {  
      "name": "John Smith",  
      "age": 25,  
      "job": "programmer"  
    },  
    {  
      "name": "Jane Doe",  
      "age": 30,  
      "job": "manager"  
    }  
  ]  
}
```

JSON

?

YAML

Complex YAML: Practice

```
{  
  "users": [  
    {  
      "name": "John Smith",  
      "age": 25,  
      "job": "programmer"  
    },  
    {  
      "name": "Jane Doe",  
      "age": 30,  
      "job": "manager"  
    }  
  ]  
}
```

JSON

```
users:  
- name: John Smith  
  age: 25  
  job: programmer  
- name: Jane Doe  
  age: 30  
  job: manager
```

YAML

Try It Out

Google YAML2JSON

Create some JSON and see their equivalent YAML

YAML: Comments

Any text after # not enclosed in "(quotes) or "" (double quotes) is considered as comments

It marks the beginning of a comment, and any text until the end of the line is completely ignored

You use this to write notes on the file or temporarily disable some sections of this file

Generally, it's a good idea to use comments sparingly and only when necessary. Adding too many comments can make the code more difficult to read and understand

```
# A single-line comments
```

```
# Document starts below
```

```
---
```

```
key: "value" # mapping
```

```
  # A list of two items
```

```
list:
```

```
  - "item 1" # first value
```

```
  - "item 2" # second value
```

```
---
```

```
# end of the document
```

YAML Data Types

Boolean

Numbers

Strings

Arrays

Maps

null

YAML: Scalars

A name followed by a colon (:) and a single space () defines a variable (also known as scalars)

Scalars are the simplest data type in YAML and can represent basic types, including boolean, integers, and floating-point numbers

See the example below about how to represent a string, an integer, a floating-point number, and a boolean value in YAML

```
string: "Hello"  
integer: 123  
float: 12.345  
boolean: No
```

Specifying Data Types Explicitly Using Tags

YAML can autodetect types. However, it is often necessary to explicitly specify the type using a tag

To force a type, we can prefix the type with a !! symbol. Here's an example:

```
company: !!str npci  
pincode: !!int 50001  
operational: !!bool true  
valuation: !!float 1048.0
```

!!bool	Denotes a boolean value
!!int	Denotes a integer value
!!float	Denotes a floating point number
!!str	Denotes a string

YAML: Dictionary

Let's represent countries with their capital in YAML format

```
{  
  "countries": {  
    "Afghanistan": "Kabul",  
    "Albania": "Tirana",  
    "Algeria": "Algiers",  
    "India": "Delhi",  
    "Angola": "Luanda"  
  }  
}
```

JSON

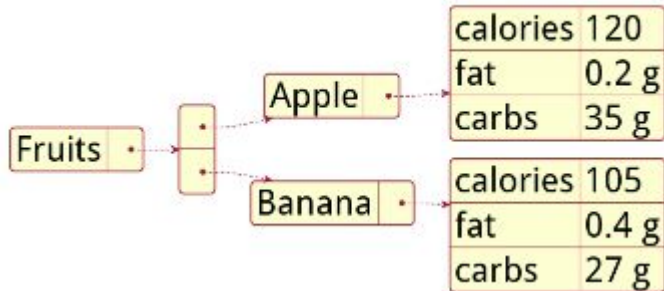
```
countries:  
Afghanistan: Kabul  
Albania: Tirana  
Algeria: Algiers  
India: Delhi  
Angola: Luanda
```

YAML

YAML: Complex Dictionary

Dictionaries are a standard data structure in many programming languages. A dictionary is a data structure that allows us to store data in key-value pairs

Let's say we have to create a list of food items and add nutrient information to all the items



```
{
  "Fruits": [
    {
      "Apple": {
        "calories": 120,
        "fat": "0.2 g",
        "carbs": "35 g"
      }
    },
    {
      "Banana": {
        "calories": 105,
        "fat": "0.4 g",
        "carbs": "27 g"
      }
    }
  ]
}
```

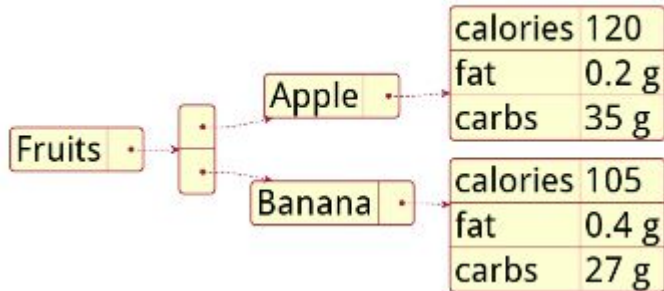
JSON

YAML

YAML: Complex Dictionary

Dictionaries are a standard data structure in many programming languages. A dictionary is a data structure that allows us to store data in key-value pairs

Let's say we have to create a list of food items and add nutrient information to all the items



```
{
  "Fruits": [
    {
      "Apple": {
        "calories": 120,
        "fat": "0.2 g",
        "carbs": "35 g"
      }
    },
    {
      "Banana": {
        "calories": 105,
        "fat": "0.4 g",
        "carbs": "27 g"
      }
    }
  ]
}
```

JSON

```
Fruits:
- Apple:
  calories: 120
  fat: 0.2 g
  carbs: 35 g
- Banana:
  calories: 105
  fat: 0.4 g
  carbs: 27 g
```

YAML

Assignment: 10 mins

Convert this Transaction JSON in YAML

Solution

```
{
  "transactionId": "9e8d1223-c25a-4845-a37d-1a24f1e96d86",
  "timestamp": "2024-07-07T16:00:55.393Z",
  "amount": 67218,
  "currency": "EUR",
  "payer": {
    "bank": "idfc",
    "name": "Esther",
    "mobile": "(883) 652-4890 x791"
  },
  "payerUpiId": "Esther-idfc",
  "payee": {
    "bank": "hdfc",
    "name": "Clotilde",
    "mobile": "431.577.5699 x51412"
  },
  "payeeUpiId": "Clotilde-hdfc",
  "status": "processing",
  "note": "Shirt",
  "metadata": {
    "latitude": 41.9584,
    "longitude": -168.9186,
    "ip": "94.177.133.63"
  }
}
```

Assignment: 10 mins

Convert the following JSON to YAML

[Questions](#)

YAML Tools

YAML Validation > YAML Lint www.yamllint.com

JSON To YAML: <https://jsonformatter.org/json-to-yaml>
<https://www.bairesdev.com/tools/json2yaml/>

YAML To JSON: <https://jsonformatter.org/yaml-to-json>

Documenting APIs with OAS: OpenAPI Specifications and Swagger

How To Document an API?

Once an API has been designed, the rules/steps has to be sent to all the clients whoever want to use it

What are different paths available? What are the methods that those paths support? What will be the request body for making an API request to the API

How do we send these information to the clients in a standard way?

One way is to send it as document in a word or text file. But that will not be standard. As a developer we might miss to add some details, or we might explain things poorly

Also, we if send a free-form types documentation - tools cannot be built around it

Proprietary product. Unauthorized disclosure, reproduction or other use prohibited

Enter Open API Specification (OAS)

OAS provides us with a set of rules to document our APIs in a standard way that everyone in the world can easily understand and use

Once we write the documentation of API in OAS format - the consumers of the API can read it to understand everything about our API

The available routes, methods, request body, sample request body, possible response code supported by each route, the response body schema, sample values

Advantage of using a specific format is that using file diff tools - we can easily find out what changed in an API - we will see this later

On top of if once a API is documented in OAS format - a Swagger UI gets automatically generated which is a great tool to interact with API with a GUI instead of reading the documentation line by line

What is OpenAPI?

Proprietary product. Unauthorized disclosure, reproduction or other use prohibited

What is OpenAPI?

OpenAPI specifies a way of describing HTTP-based APIs, which are typically RESTful APIs

An OpenAPI definition comes in the form of a
YAML or JSON file that describes the inputs and outputs of an API

It can also include information such as where the API is hosted, what authorization is required to access it,
and other details needed by consumers and producers (developers)

Once an API has been written down, we say it has been described, and it then becomes
a platform that tools and humans can make use of

Proprietary product. Unauthorized disclosure, reproduction or other use prohibited

Uses of OpenAPI

Testing

Getting early feedback on design

Comparing API changes across versions

Generate Server and Client side Code

What is Swagger?

What is Swagger?

In the beginning there was Swagger UI which create a rough guide for writing YAML files that described HTTP APIs

Later, more tools were built that relied on this guide, which soon became a specification and a standard

The tools and this specification were collectively known as “Swagger”

The specification grew more mature and was released as open source, which encouraged the community to create even more tools

They soon began to contribute features to the specification, which finally began to be adopted by large companies

What is Swagger?

In 2015 Swagger was adopted by SmartBear, which then donated the specification part to the Linux Foundation

During that transfer, the specification was renamed as the “OpenAPI specification,” and SmartBear retained the copyright for the term “Swagger”

Today, as a result of this historical quirk, you’ll find the terms used interchangeably

We use the term “OpenAPI” to refer the heart of this ecosystem—the specification

We use “Swagger” to refer to the specific set of tools managed by SmartBear (Swagger UI, Swagger Editor, Swagger Parser, etc)

Using Swagger Editor To Write Open API Specifications

Official Documentation

<https://swagger.io/specification/>

Swagger Editor

OpenAPI definitions have a lot of nuances that most of us can't be bothered to learn right away

Swagger Editor is a tool that helps us write OpenAPI specifications

It is a web application hosted at <https://editor.swagger.io>

We can either use it directly from the URL or download and host it on our infra itself.
Like a lot of Swagger tools, it is open source

The web application contains both a text editor and a panel showing the generated documentation.
The documentation pane shows the results of what we type, giving us immediate feedback and a great affirmation that we typed the right things

Let's Start Swagger Editor

The image shows the Swagger Editor web application. It has a dark header bar with the 'SwaggerEditor' logo and menu items: File, Edit, Generate Server, and Generate Client. The main area is split into two panels. The left panel, labeled 'Editor panel (OpenAPI YAML editor)', contains a text editor with a YAML definition for a 'Swagger Petstore' API. The right panel, labeled 'UI Docs panel (Swagger UI)', displays a rendered version of the API documentation. Labels with arrows point to the 'File' menu (Toolbar), the text editor (Editor panel), and the rendered documentation (UI Docs panel).

Toolbar

Editor panel (OpenAPI YAML editor)

```
1 swagger: "2.0"
2 info:
3   description: "This is a sample server
4     Petstore server. You can find out more
5     about Swagger at [http://swagger.io]
6     (http://swagger.io) or on [irc.freenode.net
7     , #swagger](http://swagger.io/irc/).
8     For this sample, you can use the api key
9     `special-key` to test the authorization
10    filters."
11  version: "1.0.0"
12  title: "Swagger Petstore"
13  termsOfService: "http://swagger.io/terms/"
14  contact:
15    email: "apiteam@swagger.io"
16  license:
17    name: "Apache 2.0"
18    url: "http://www.apache.org/licenses
19    /LICENSE-2.0.html"
20  host: "petstore.swagger.io"
21  basePath: "/v2"
22  tags:
23    - name: "pet"
24      description: "Everything about your Pets"
25      externalDocs:
26        description: "Find out more"
```

UI Docs panel (Swagger UI)

Swagger Petstore 1.0.0

[Base URL: petstore.swagger.io/v2]

This is a sample server Petstore server. You can find out more about Swagger at <http://swagger.io> or on [#swagger](http://irc.freenode.net). For this sample, you can use the api key **special-key** to test the authorization filters.

[Terms of service](#)
[Contact the developer](#)
Apache 2.0
[Find out more about Swagger](#)

Schemes
HTTPS [Authorize](#)

pet Everything about your Pets Find out more: <http://swagger.io>

POST /pet Add a new pet to the store

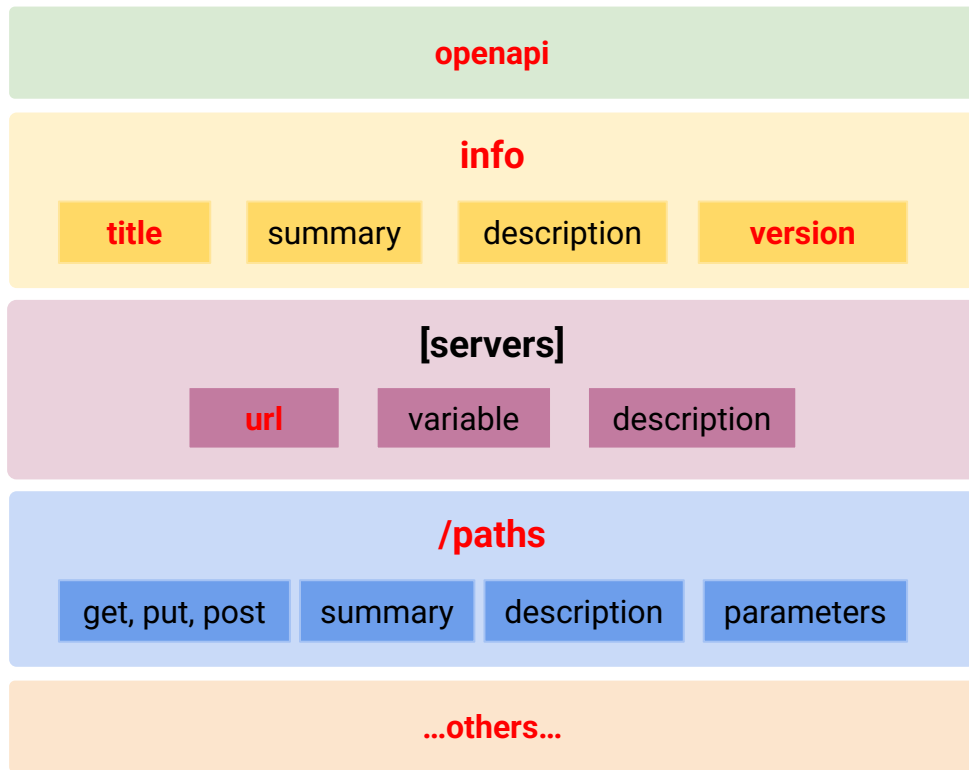
The **Editor panel** is the text editor where we will write the YAML for our OpenAPI definition. The content of this panel will be our OpenAPI definition

The **UI Docs panel** reflects what is in the Editor panel. As you type or make changes in the Editor panel, you will see immediate feedback in the UI Docs panel

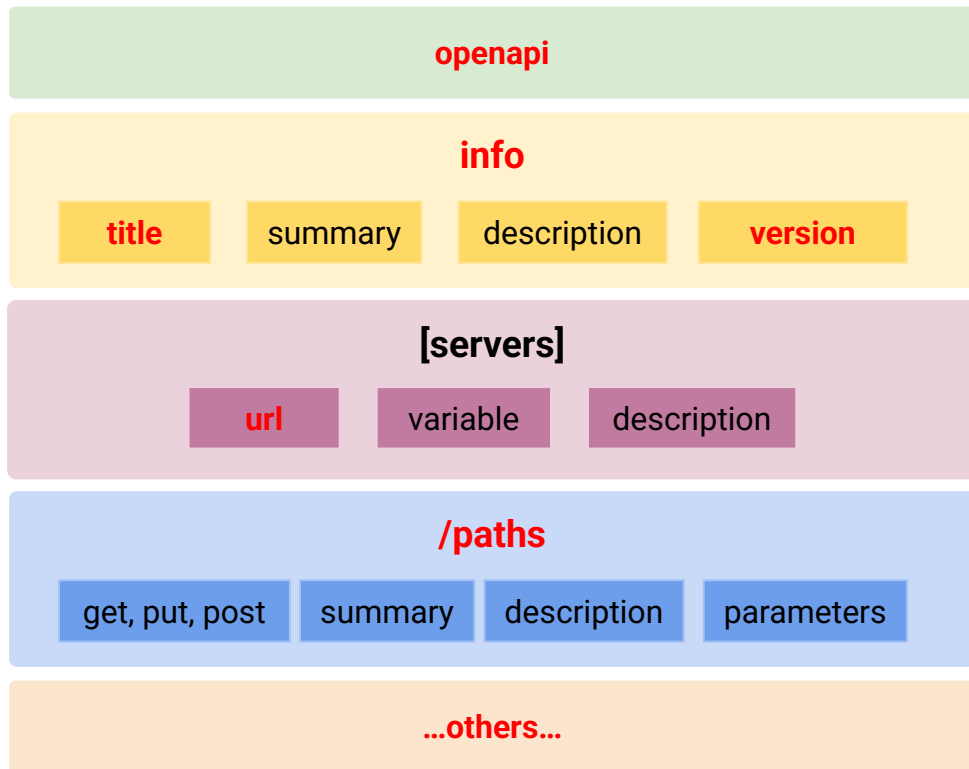
OAS: OpenAPI Specifications

Proprietary product. Unauthorized disclosure, reproduction or other use prohibited

Structure of OAS File

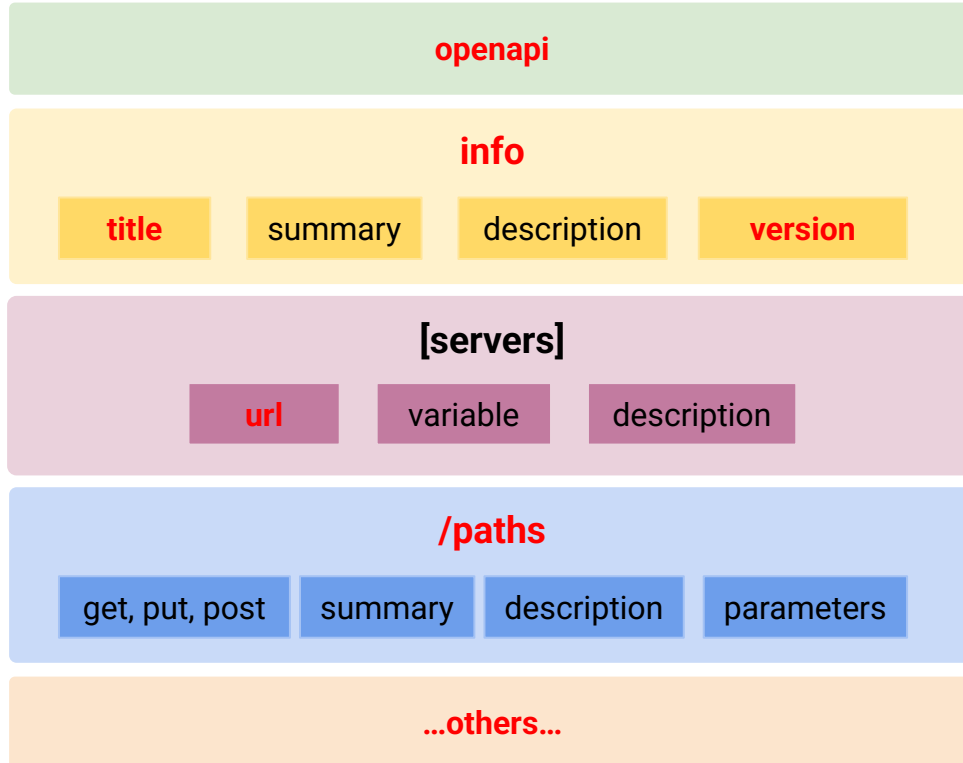


Structure of OAS File



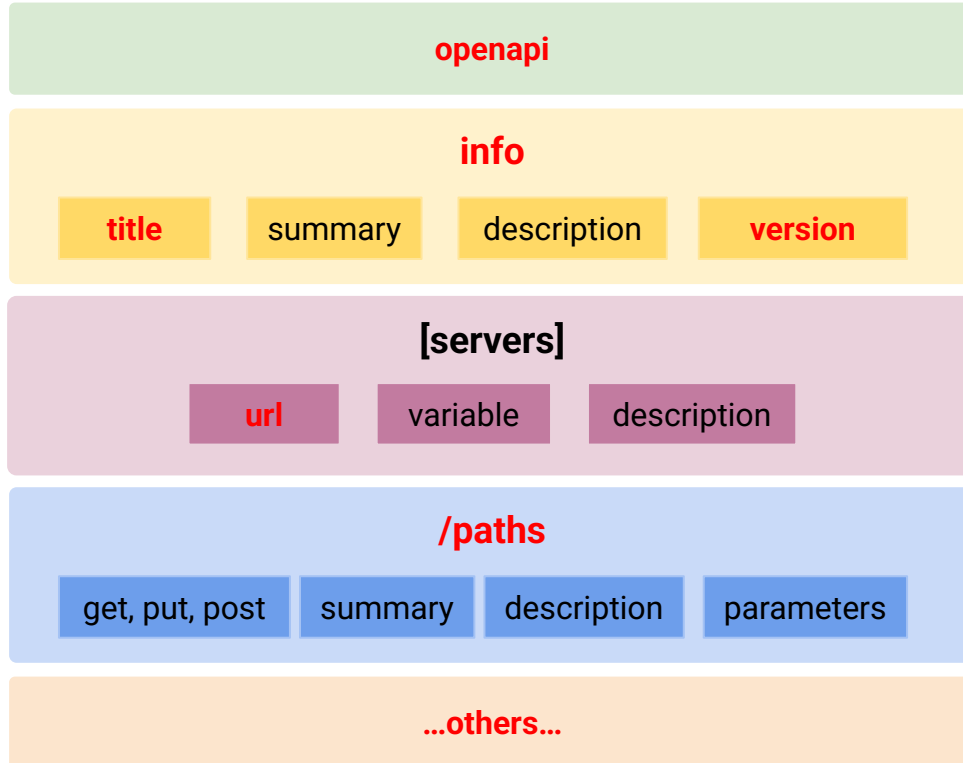
REQUIRED. Provides metadata about the API informing the clients about the high level summary of what the API does

Structure of OAS File



Array of servers providing the connectivity options by providing hostnames, port etc

Structure of OAS File



Provides information about available paths like /payments, /payments<txn_id>, /apiStatus along with usage details like methods, payload schema etc

Minimal OpenAPI Document Structure

The OAS structure is long and complex. So, this slide just describes the minimal set of fields it must contain, while upcoming slides give more details about all the objects one can write inside OAS.

```
openapi: 3.1.0
info:
  title: A minimal OpenAPI document
  version: 0.0.1
paths: {} # No endpoints defined
```

openapi (string)*

Indicates the version of the OAS that this document is using, e.g., "3.1.0".

info (Info object)*

Provides general information about the API (like its description, author and contact information) but the only mandatory fields are title and version.

title (string)*: A human-readable name for the API

version (string)*: Indicates the version of the API document (not to be confused with the OAS version above)

paths (Paths Object)*

This describes all the endpoints of the API, including their parameters and all possible server responses. It can be empty array for minimal OAS document

Info Object

The info object provides metadata about the API. The metadata MAY be used by the clients if needed, and MAY be presented in editing or documentation generation tools for convenience.

```
title: Sample Pet Store App
summary: A pet store manager.
description: This is a sample server for a pet store.
termsOfService: https://example.com/terms/
contact:
  name: API Support
  url: https://www.example.com/support
  email: support@example.com
license:
  name: Apache 2.0
  url: https://www.apache.org/licenses/LICENSE-2.0.html
version: 1.0.1
```

info (Info object)*

title (string)*: A human-readable name for the API

summary(string): A short summary of the API

description(string): A detail description of the API

termsOfService(string): A URL to the Terms of Service for the API. This MUST be in the form of a URL.

contact(Contact Object): The contact information for the exposed API.

license(License Object): The license information for the exposed API.

version (string)*: Indicates the version of the API document (not to be confused with the OAS version above)

Contact & License Objects

Both Contact & License objects are optional inside the root Info object. These objects can be used to mention the contact information & license information respectively for the exposed APIs.

```
title: Sample Pet Store App
summary: A pet store manager.
description: This is a sample server for a pet store.
termsOfService: https://example.com/terms/
contact:
  name: API Support
  url: https://www.example.com/support
  email: support@example.com
license:
  name: Apache 2.0
  url: https://www.apache.org/licenses/LICENSE-2.0.html
version: 1.0.1
```

contact (Contact object)

name (string): The identifying name of the contact person/organization

url(string): The URL pointing to the contact information

email(string): The email address of the contact person/organization.

license (License object)

name (string)*: The license name used for the API

identifier(string): The license identifier for the API

url(string): A URL to the license used for the API.

Servers Object

An object representing a list of servers where the APIs are hosted. We can mention multiple server details like Dev, Staging, Prod etc.

servers:

- url: `https://development.gigantic-server.com/v1`
description: Development server
- url: `https://staging.gigantic-server.com/v1`
description: Staging server
- url: `https://api.gigantic-server.com/v1`
description: Production server

servers (server object)

url (string)*: A URL to the target host. This URL supports Server Variables and MAY be relative, to indicate that the host location is relative to the location where the OpenAPI document is being served. Variable substitutions will be made when a variable is named in {brackets}.

description(string): An optional string describing the host designated by the URL.

variables(Map[string, Server Variable Object]): A map between a variable name and its value. The value is used for substitution in the server's URL template.

Document the Payments API with OAS

Let's recall what different paths are available in our Payments microservice

Recap Payments API

Check if API is running

GET

/apiStatus

Get all transactions

GET

/payments

Make a payment

POST

Filter transactions

GET

/payments?status=failed

Get one transactions

GET

/payments/<txn_id>

Update one payment

PATCH

Delete one payment

DELETE

POST

Create one user

/signUp

Let's Try It Out

Open <https://editor.swagger.io/>
For hints refer [this](#)

Self Managed Swagger Editor

**We cannot write our API implementation details on a cloud platform for security reasons
In some organisation, these sites might also be blocked
Lets see how we can install Swagger tool on our server**

[Running Swagger Editor On Our Own Servers](#)

Installing Dependencies & Launching Swagger Editor

Install NodeJS. Follow the steps [here](#)

Check node version and npm version

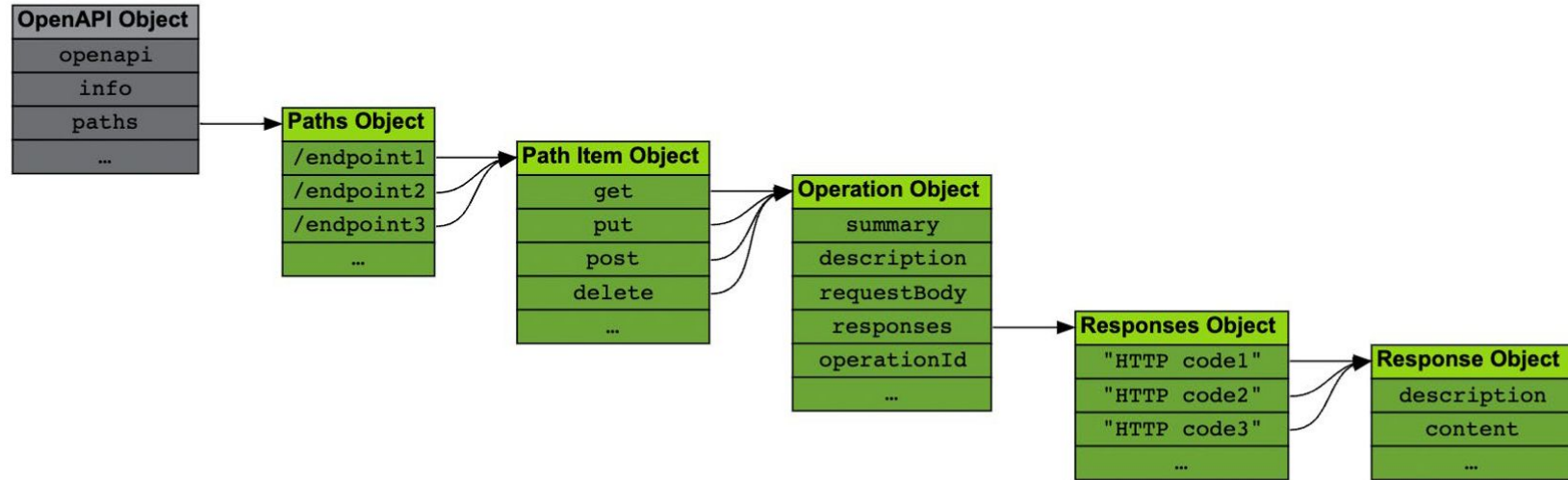
Install HTTP server: *npm install -g http-server*

Download Swagger Editor from [here](#)

Launch Swagger Editor: *http-server swagger-editor-master --cors*

Paths Object

Every field in the Paths Object is a Path Item Object describing one API endpoint. The Path Item Object describes the HTTP operations that can be performed on a path with a separate Operation Object for each one. Allowed operations match HTTP methods names like get, put or delete, to list the most common



To get a map view of OAS document, you can visit [here](#)

API Testing

Software defect removal is the most expensive and time-consuming form of work for software

Introduction To the Section

When building an API product or platform, it is important to have an API testing strategy established

Selecting the right approach for API testing contributes to the success of an API program's supportability

It also contributes to faster delivery while avoiding one of the costliest aspects of software development: defect removal

Types of API Testing

**Acceptance
Testing**

**Security
Testing**

**Functional
Testing**

**Load
Testing**

**Integration
Testing**

Acceptance Testing (Overall Product Working)

Acceptance testing, also called solution-oriented testing, ensures that the API supports the captured business requirements

Acceptance testing seeks to answer the following questions:

Does the API solve real problems that our customers have?

Does it produce the desired outcomes for the jobs to be done?

The internals of the API can and likely will change over the course of development, but this should not affect the results of the acceptance tests

Acceptance testing is the most valuable style of testing for an API. It is where the most testing effort should be spent, after code testing, when limited time is available

Acceptance Testing (Overall Product Working)

Instead of focussing on low level details of implementation, acceptance testing is more towards business capability

For example: For an e-commerce company, placing an order will be a business requirement. In the acceptance testing the focus is on checking if on an overall level customers are able to place an order or not, and not worrying too much about the backend implementation, request, response, etc

This testing provides business stakeholders confidence of certifying the API is ready to go in Production

Security Testing

Each week, a new headline appears that indicates a company has been hacked and private information exposed

Security is a process, not a product, and a continual one at that.

Security testing aims to answer the following questions:

Is the API protected against attacks?

Does the API offer opportunities for sensitive data to be leaked?

Is someone scraping my API and compromising business intelligence through data?

We will see the steps that needs to be taken to secure your API in the “Securing API” section

Load Testing

Once the functionalities are in place - the last thing to check is the scalability

It's important to arrive at a reasonable throughput that is expected by an API and always test the API with 2X, 5X and 10X of that load depending of the seasonality variances

Additionally it's also of interest to us to know what happens to the SLA when the load increases and are the systems built to accommodate the change in SLA for transient time.

When the load increases, it doesn't mean that API will crash all the time. Sometimes it's response time will increase, sometimes timeouts might increase and the systems should be prepared to prevent cascading failures

Hands On

Let's load test our Payments API - [Link](#)

- Find the total number of users till which your API doesn't fail (10 -> 100 -> 1000)
- Find the average response time your API can handle (See how response time increases when load increases)

Contract (Functional) Testing

API contract testing, sometimes referred to as functional testing, is used to verify that each API operation meets the expected behavior and honors the API's defined contract for the consumer

Contract testing answers the following questions:

Is each operation working to the specification for all success cases?

Are input parameters being followed? How are bad inputs handled?

Are the expected outputs received?

Is response formatting correct? Are the proper data types used?

Are errors being handled correctly? Are they reported back to the consumer?

Contract (Functional) Testing

API contract testing must first ensure the correctness of each API operation in terms of response body, code, etc

Handling thousands of clients per minute does no good if the information that the API is providing or acting on does not meet the API's specification

Identifying and eliminating bugs, hunting out inconsistencies, and verifying that an API meets the specification against which it has been designed all fall under the umbrella of testing for correctness

Next, API contract testing must focus on reliability. The API should provide the correct information every time an operation is called. Executing the same action repeatedly for an API operation designed to be idempotent should produce the same results.

API Monitoring

Using Prometheus and Grafana

Proprietary product. Unauthorized disclosure, reproduction or other use prohibited

What can go wrong?

You have done everything we discussed so far:

- *Started with designing the API*
- *Then you documented the API using OAS*
- *Deployed stub servers for client side to start the development*
- *Both client and server side testing is done*
- *You have deployed the code in Production*
- *You have setup the monitoring systems and everything is set*

In any software systems, things can always go wrong. In case of APIs, what are the different things that can go wrong? (Apart from request/response errors)

List them out

What can go wrong?

Your API can go down

Network data input and output - to tell if any fishy traffic is coming in

API might not be able to cope up with the traffic

Database on which API reads and writes to might not be able to cope up with the traffic

API's response time can go very high

CPU utilization to check if server is under high load

Your API can be attacked by a hacker sending too many requests

Memory utilization to see if any process is stuck or large requests are coming in

Clients are making request to your API with wrong request body without getting proper error message

You might have missed to cover some edge cases and now clients are using your API in undesired ways

Observability



Things To Monitor in APIs

Write Down 5-7 possible things you would want to monitor in your APIs

Things To Monitor

Traffic: Also known as **requests per second** or **throughput**

Response Code Distribution (200, 4XX, 5XX)

Response time: Avg response time, also known as **latency**

Machine Resource: CPU, RAM, Network Input/Output

Requests under SLA (say 250ms)

Requests under SLA (say 250ms)

p90 Request Statistics: Mean, Media, Avg, Min, Max

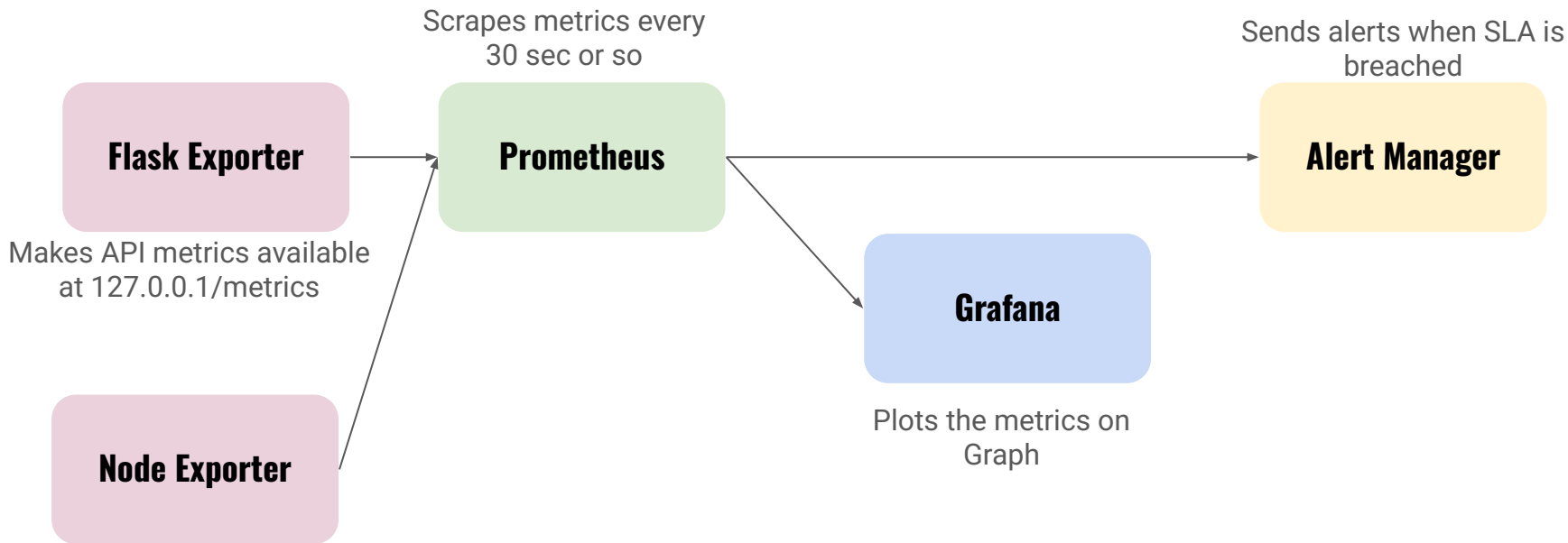
p50, p90, p99 Request Statistics: Mean, Media, Avg, Min, Max

Error Rate: Error per second, minute, hours

End Goal



Observability Tech Stack



Steps

Make required change to Payments API script (imports, Prometheus app)

Launch Payment API

Download node exporter

Launch Node exporter

Download prometheus, add exporter details in prometheus.yaml and launch prometheus

Download grafana, add prometheus and data source, create node dashboard, create API dashboard

Hands On

[Link](#)

Proprietary product. Unauthorized disclosure, reproduction or other use prohibited