eazy bytes

Zero To Master

OpenAPI Specification & Swagger Tools

COURSE AGENDA

Welcome to the world of OpenAPI

History of OpenAPI & Swagger. Relation between them

Details about Swagger tools

How to get started with OpenAPI in code first & Design first scenarios

COURSE AGENDA

How to write a valid OpenAPI yaml document

Writing re-usable content inside OpenAPI with components

Data types supported by OpenAPI & their details

Inheritance & polymorphism inside OpenAPI

COURSE AGENDA

How to describe APIs security inside OpenAPI

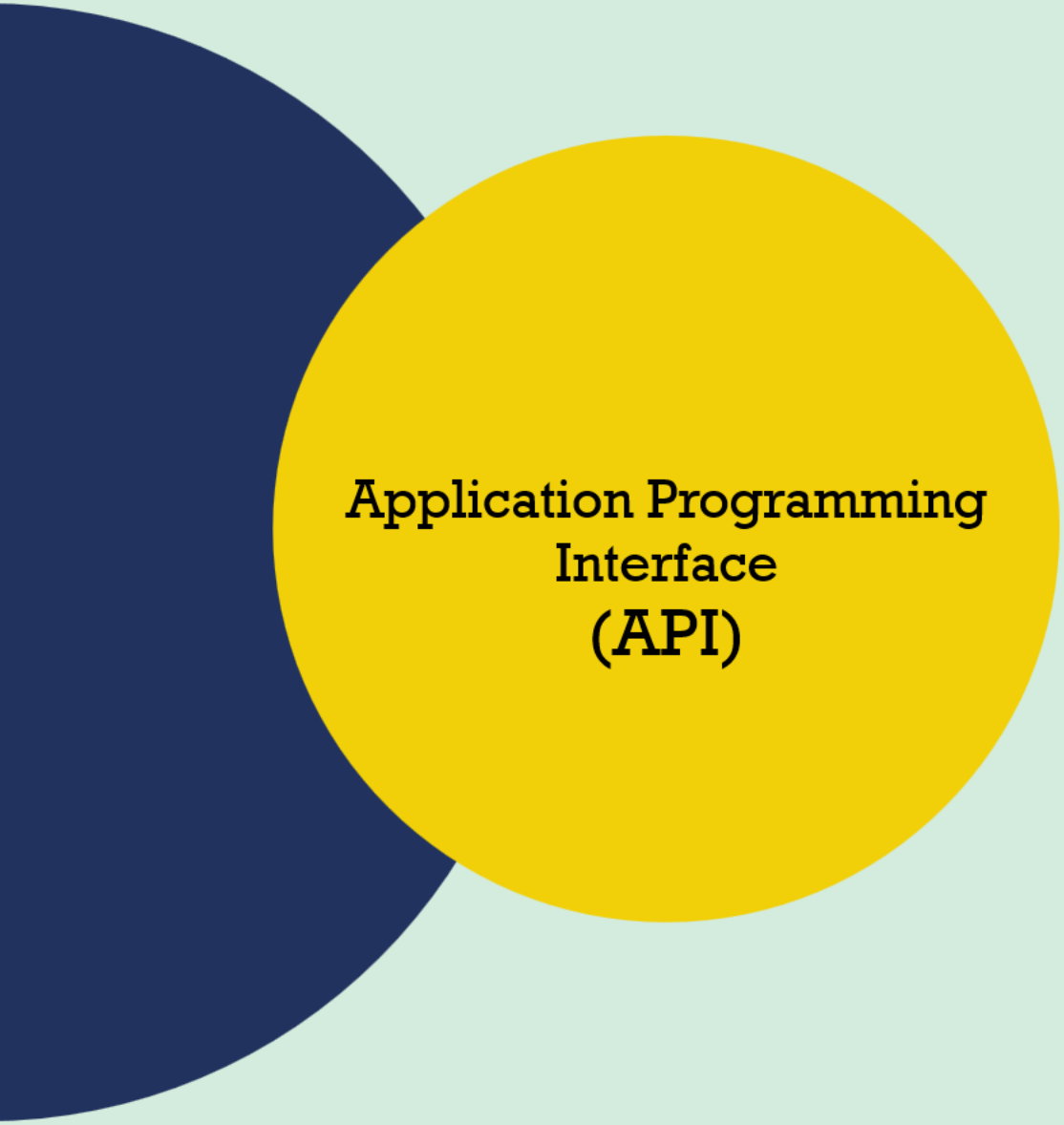
How to mock APIs with OpenAPI specification

How to generate client code & server stubs using OpenAPI

Deploying & Hosting OpenAPI along with Swagger UI

eazy bytes

WHAT IS AN API



An API defines the allowed interactions between two pieces of software. It is composed of the list of possible methods to call (requests to make), their parameters, return values and any data format they require.

APIs can be local, where both interacting parties run on the same machine or remote where the interacting parties run on separate machines and communicate over a network.

The party offering up its services through an API is called the provider and the one requesting these services is the consumer. APIs are also called Contracts, because they are assumed to be unbreakable: The provider promises not to change its API w/o communication and to keep honoring it.

APIs adhere to standards (typically HTTP and REST), that are developer-friendly, easily accessible and understood broadly. They are designed for consumption for specific audiences (e.g., mobile developers), they are documented, and versioned.

eazy bytes

APIS EVERYWHERE

Ecommerce Payments

Consumer

API Communication

PayPal

Producer

Flight Ticket Bookings

Consumer

API Communication

Producer

APIs are every where in the world & they are unsung heroes

eazy bytes

WHY DO WE NEED OPENAPI SPECIFICATION ?

IMPROPER API DOCUMENTATION

Unclear API documentation leads to mistakes due to interpretation differences. To find the required information developers might have to read source code, debug programs or analyze network traffic, which are gigantic time sinks

LOT OF COLLABORATION

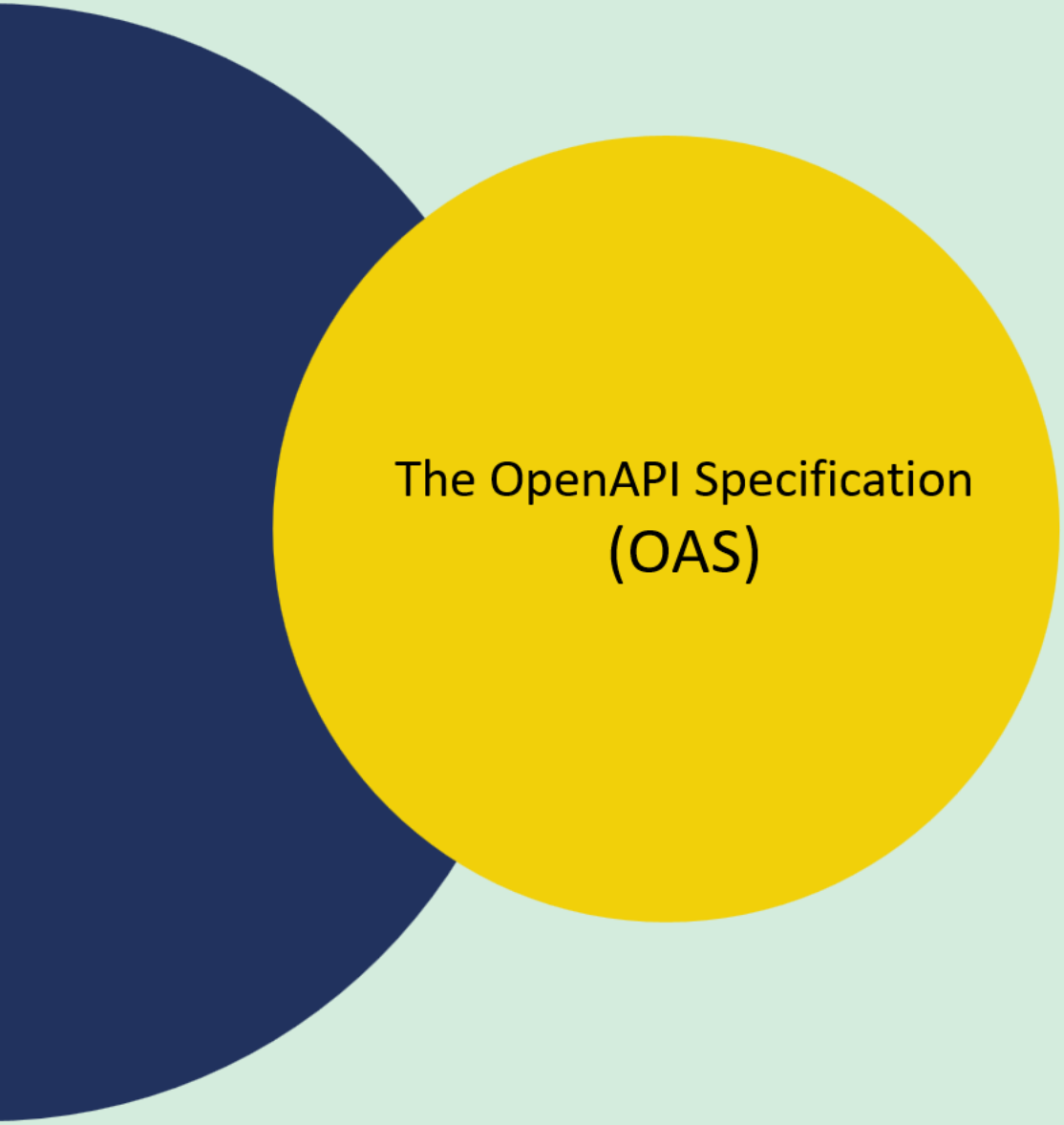
Both consumers and producers of the API need to do lot of collaboration discussing the request, response formats. For public APIs or product APIs, this will be a nightmare task for producers when ever a new consumer want to use the exposed API

THERE IS NO STANDARD

With out OpenAPI specification, there will be no formal standards and every organization will follow their own format. Due to this there will be no open-source community building tools, fixing bugs

eazy bytes

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.

Apart from request & response format, OpenAPI 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 (such as web developers).

Currently, the OAS is maintained, evolved and promoted by the OpenAPI Initiative (OAI), a consortium of industry experts with an open governance structure under the Linux Foundation umbrella. Since it is open source, you have a say in the future direction of the Specification!

The openness has encouraged the creation of a vast amount of tools. It's probably because of the amount of tools available when working with OpenAPI that it has become the most broadly adopted industry standard for describing modern APIs.

eazy bytes

ADVANTAGES OF USING OPENAPI

OpenAPI specification can be understand by both machines & humans

Using OpenAPI specification we can generate both server and client code in any programming language, freeing developers from having to perform data validation or write SDK glue code

We can use mock servers to provide example responses which enable both consumers & producers to start development at a same time

We can perform the validations on the data flowing through your API (in both directions) is correct or not, during development & once deployed

Since OpenAPI is a widely accepted standard, a large developer community & vast number of tools are available for everyone

eazy bytes

HISTORY OF SWAGGER & OPENAPI

2010 – SWAGGER BORN

Swagger started out as a simple, opensource specification for designing RESTful APIs in 2010. Opensource tooling like the Swagger UI, Swagger Editor and the Swagger Codegen were also developed to better implement and visualize APIs defined in the specification.

2015 – OPENAPI BORN

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

Today, as a result of this historical quirk, you'll find the terms used interchangeably. But we should use the term "OpenAPI" to refer the the specification and to use "Swagger" to refer to the specific set of tools managed by SmartBear (which includes Swagger UI, Swagger Editor, Swagger Hub etc.)

eazy bytes

TOOLS AVAILABLE IN SWAGGER

Swagger has a set of open-source & commercial tools built around the OpenAPI Specification that can help you design, build, document and consume REST APIs.

Swagger Open Source Tools

Swagger Editor – browser-based editor where you can write OpenAPI specs.

Swagger UI – renders OpenAPI specs as interactive API documentation.

Swagger Codegen – generates server stubs and client libraries from an OpenAPI spec.

Swagger Commercial Tools

SwaggerHub - Design & document all your REST APIs in one collaborative platform.

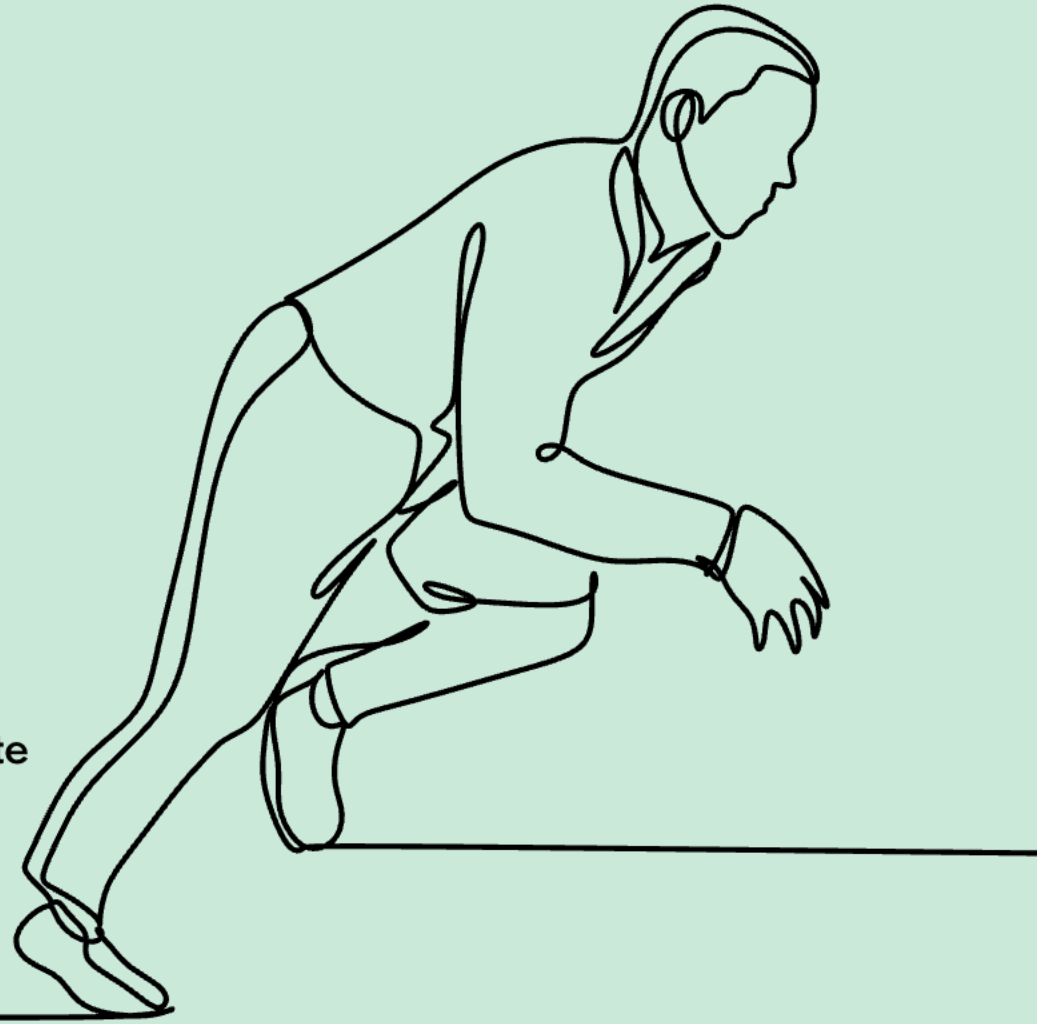
SwaggerHub Enterprise - Store your organization's common styles and models in Domains, that can then be referenced across hundreds of API projects, or check individual APIs for style consistency with the Style Validator.

SwaggerHub Explore - Test and generate API definitions from your browser in seconds.

eazy bytes

HOW TO START

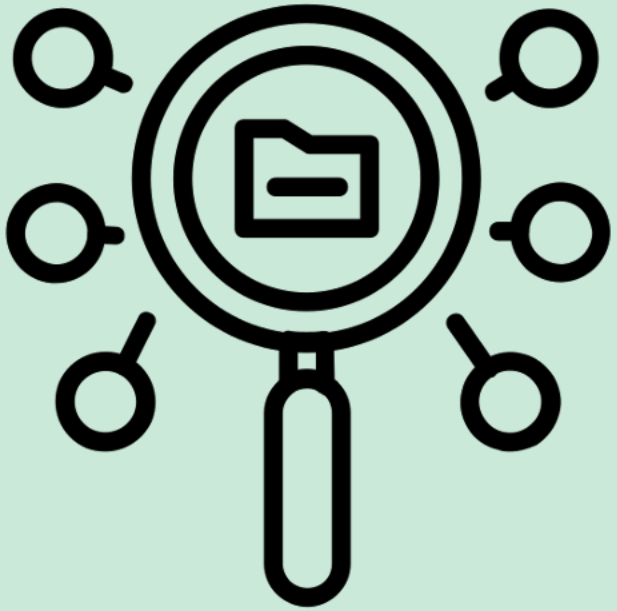
WITH OPENAPI IN CODE-FIRST APPROACH



If you have APIs already developed and available, you can easily generate the OpenAPI specification using SwaggerHub & SwaggerHub Explore

eazy bytes

SwaggerHub & SwaggerHub Explore



SwaggerHub is a collaborative platform where we can define our APIs using the OpenAPI Specification and manage our APIs throughout their lifecycle. SwaggerHub is brought to you by the same people behind the open-source Swagger tools.

SwaggerHub can act as a central repository for hosting and accessing API definitions. It also integrates with the core Swagger tools (UI, Editor, Codegen, Validator) into a single platform to help you coordinate the entire workflow of an API's lifecycle.

With SwaggerHub & SwaggerHub Explore, we can

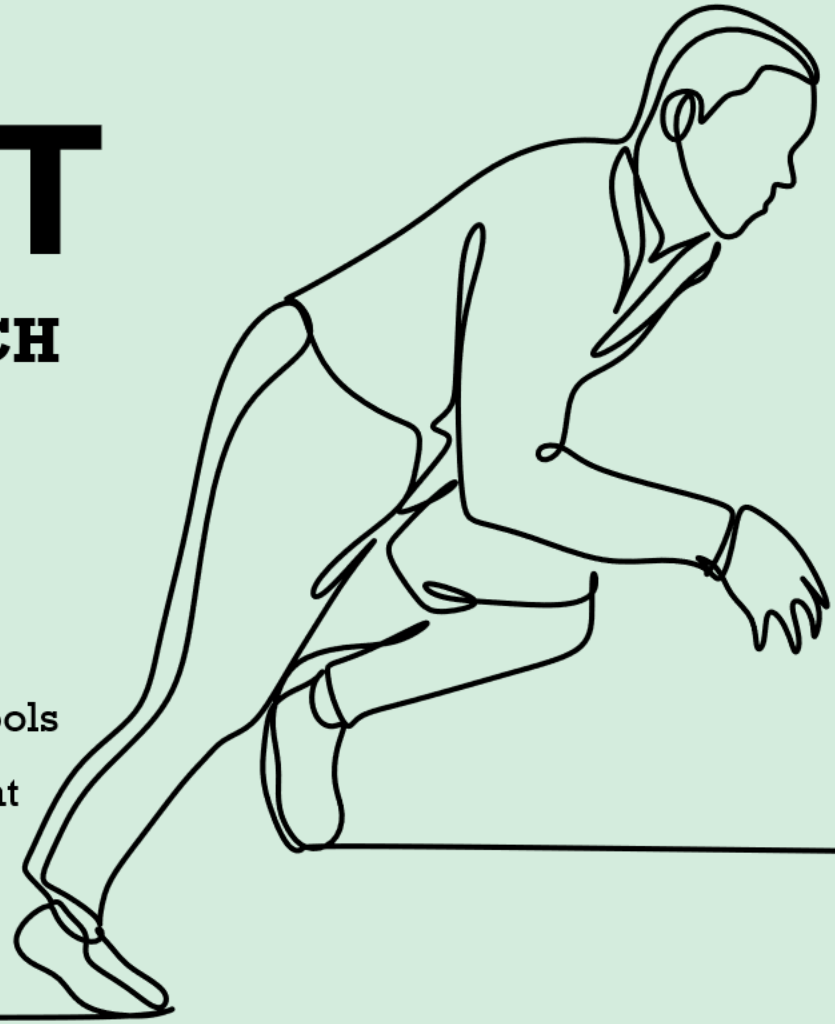
- ✓ Test our API calls
- ✓ Mock API calls
- ✓ Document our APIs

Creating documentation for your APIs is easy with SwaggerHub Explore and Swagger Hub. You can autogenerate documentation with ease by selecting your previously tested endpoints from your history and clicking "create definition".

eazy bytes

HOW TO START

WITH OPENAPI IN DESIGN-FIRST APPROACH



In Design-first approach, the API description can be written first using tools like Swagger Editor and then the code follows. The advantage here is that the code already has a skeleton upon which to build, and that some tools can provide boilerplate code automatically.

XML vs JSON vs YAML

eazy bytes

```xml
<applications>
  <application>
      <name>Accounts</name>
      <technology>Python</technology>
   </application>
   <application>
      <name>Cards</name>
      <technology>Java</technology>
    </application>
    <application>
      <name>Loans</name>
      <technology>Ruby</technology>
    </application>
</applications>
```

XML

eXtensible Markup Language

```json
{
    "applications": [
      {
        "name": "Accounts",
        "technology": "Python"
      },
      {
        "name": "Cards",
        "technology": "Java"
      },
      {
        "name": "Loans",
        "technology": "Ruby"
      }
    ]
}
```

JSON

JavaScript Object Notation

```yaml
applications:
- name: Accounts
  technology: Python
- name: Cards
  technology: Java
- name: Loans
  technology: Ruby
```

YAML

YAML Aint Markup Language

eazy bytes

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.

```yaml
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

eazy bytes

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.

```yaml
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)

eazy bytes

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.

```yaml
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.

eazy bytes

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.

```yaml
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}.

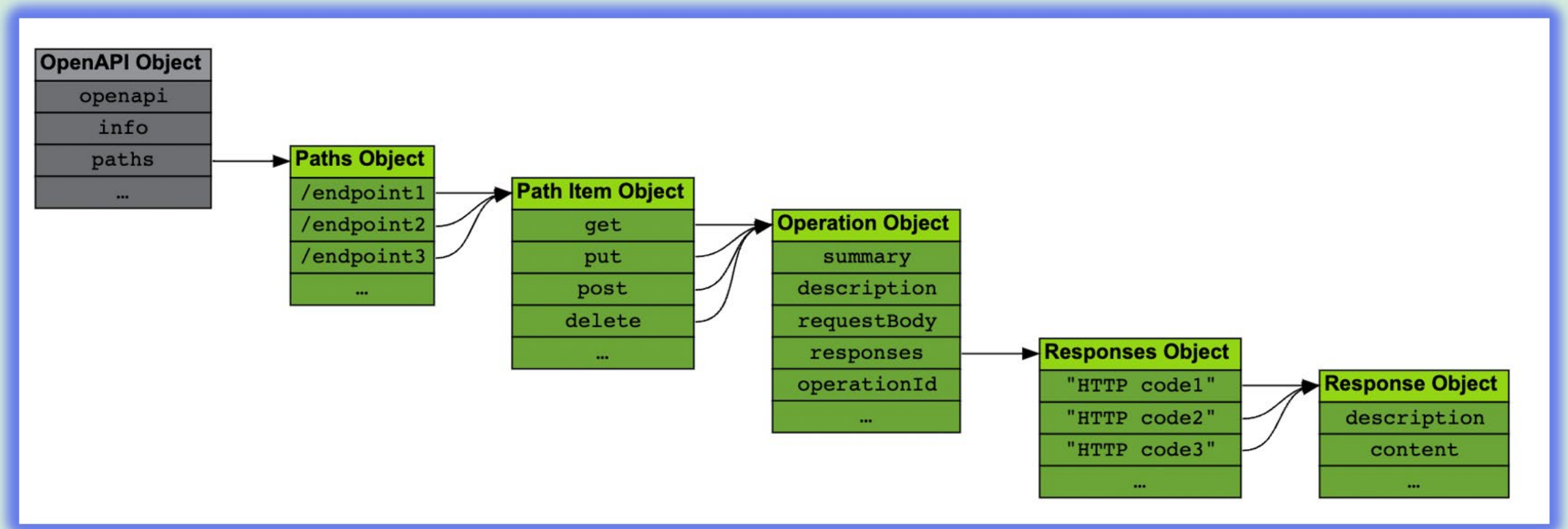
description(string): An optional string describing the host designated by the URL.

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.

eazy bytes

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



eazy bytes

The Operation Object

Besides giving the operation a summary and a description, the Operation Object basically describes the operation's parameters, payload and possible server responses.

```
paths:
  /board:
    get:
      summary: Get the whole board
      description: Retrieves the current state of the board and the winner.
      parameters:
        ...
      responses:
        ...
```

The Responses Object

The Responses Object is a container for the expected answers the server can give to this request. Each field name is an HTTP response code enclosed in quotation marks and its value is a Response Object

```
paths:
  /board:
    get:
      responses:
        "200":
          ...
        "404":
          ...
```

eazy bytes

The Response Object

The Response Object contains a mandatory description of the meaning of the response in the context of this operation, complementing the sense of the HTTP response codes (which are generic in nature). This helps developers understand better how to react to this particular code.

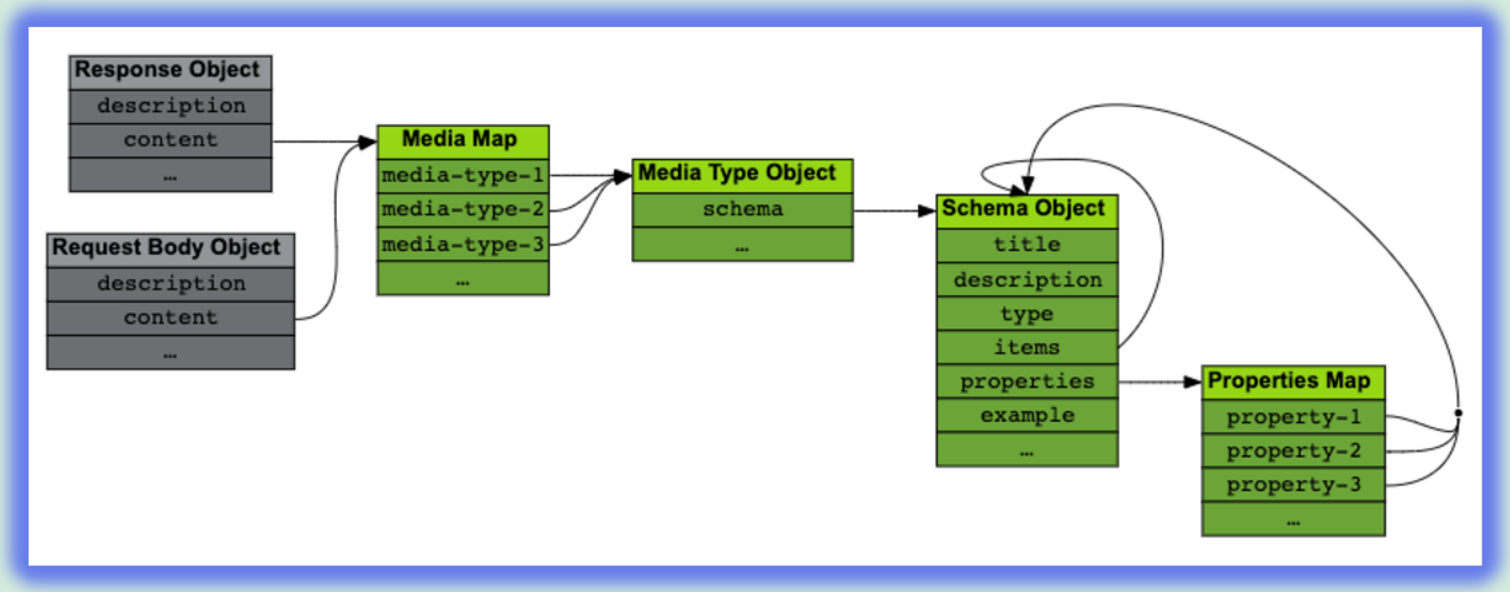
The most important field, though, is content because it describes the possible payloads of the response.

```
paths:
  /board:
    get:
      responses:
        "200":
          description: Everything went fine.
          content:
            ...
```

eazy bytes

Content of Message Bodies

The content field can be found both in Response Objects and Request Body Objects.



eazy bytes

The Content Object

Allows returning content or accepting content in different formats, each one with a different structure described by the Media Type Object. Wildcards are accepted for the media types, with the more specific ones taking precedence over the generic ones.

```
content:
  application/json:
    ...
  text/html:
    ...
  text/*:
    ...
```

The Media Type Object

The Media Type Object describes the structure of the content and provides examples for documentation and mocking purposes. The structure is described in the schema field like shown below.

```
content:
  application/json:
    schema:
      ...
```

The Schema Object

The Schema Object defines a data type which can be a primitive (integer, string, …), an array or an object depending on its type field. type is a string and its possible values are: number, string, boolean, array and object. Depending on the selected type a number of other fields are available to further specify the data format.

Example integer with limited range

```yaml
content:
  application/json:
    schema:
      type: integer
      minimum: 1
      maximum: 100
```

Example string with only three valid options

```yaml
content:
  application/json:
    schema:
      type: string
      enum:
      - Alice
      - Bob
      - Carl
```

Array types must have an items field, which is a Schema Object itself, and defines the type for each element of the array. Additionally, the size of the array can be limited with minItems and maxItems.

```yaml
content:
  application/json:
    schema:
      type: array
      minItems: 1
      maxItems: 10
      items:
        type: integer
```

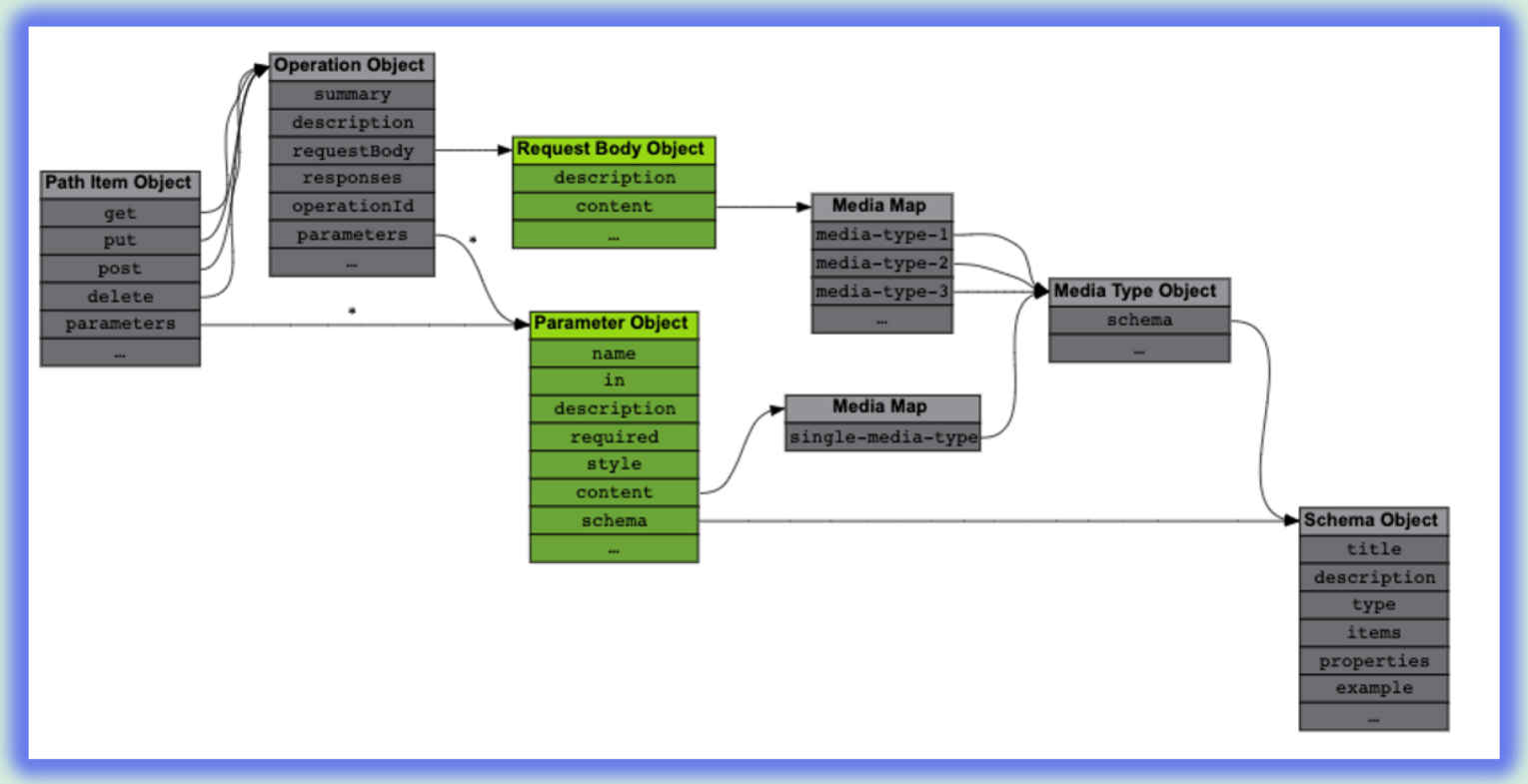
Finally, object types must have a properties field listing the properties of the object. This field is a map pairing property names with a Schema Object defining their type. This allows building data types as complex as required.

```yaml
content:
  application/json:
    schema:
      type: object
      properties:
        productName:
          type: string
        productPrice:
          type: number
```

eazy bytes

Parameters and Payload of an Operation

OpenAPI provides two mechanisms to specify input data, parameters and request body (message payload). Parameters are typically used to identify a resource, whereas the message payload provides content for that resource.



eazy bytes

The Parameter Object

The parameters field in the Path Item and Operation Objects is an array containing Parameter Objects. When provided in the Path Item Object, the parameters are shared by all operations on that path.

```yaml
parameters:
- name: id
  in: query
  schema:
    type: integer
    minimum: 1
    maximum: 100
```

parameters (Parameter object)

in (string)*: Location of the parameter. Valid values are like *path, query, header, cookie*

name (string)*: Case-sensitive. Must be unique in each location.

description (string): Useful for documentation. Might contain usage examples, for instance.

required (boolean): Whether this parameter must be present or not. The default value is *false*.

Parameter Type (Schema) : Parameter's type can be specified by using a Schema Object in the schema field. Schema objects allow defining primitive or complex types (like arrays or objects) and impose additional restrictions on them.

eazy bytes

The Request Body Object

The message body of a request is specified through the requestBody field in the Operation Object, which is a Request Body Object. The only mandatory field in the Request Body Object is content which we already discussed for response.

```
paths:
  /board:
    put:
      requestBody:
        ...
```

```
requestBody:
  content:
    application/json:
      schema:
        type: integer
        minimum: 1
        maximum: 100
```

eazy bytes

Providing Better Documentation

Besides machine-readable descriptions, an OpenAPI document can also include traditional documentation meant to be read by developers. Automatic documentation generators can then merge both and produce comprehensive, nicely-structured reference guides.

Almost every object in the OpenAPI Specification accepts a description field which can provide additional information for developers, beyond what can be automatically generated from the API descriptions. description fields allow rich text formatting by using CommonMark Syntax

The CommonMark Syntax

Headings:

```
# Level 1
## Level 2
### Level 3
```

Emphasis:

```
*Emphasis*
**Strong Emphasis**
***Both***
```

Lists:

```
- Item 1
- Item 2
  - Item 2.1
```

Links:

```
[Link text](Link URL)
![Alt text](Image URL)
```

Code:

```
An inline `code span`.
```
```
A fenced code block
```
```
```

eazy bytes

The Examples Object

We can list examples explicitly instead of having them embedded in the description field, enabling automated processing by tools. Two different fields provide this functionality: example allows one sample whereas examples allows multiple.

Mentioning example(s) data inside the OpenAPI specification will help in Special rendering of the examples inside the documentation & Example objects can be used by mock servers as return values.

```yaml
responses:
        '200':
          description: Return category details
          content:
            application/json:
              schema:
                type: object
                properties:
                  categoryId:
                    type: integer
                    example: 101
                  name:
                    type: string
                    example: Mobiles
```

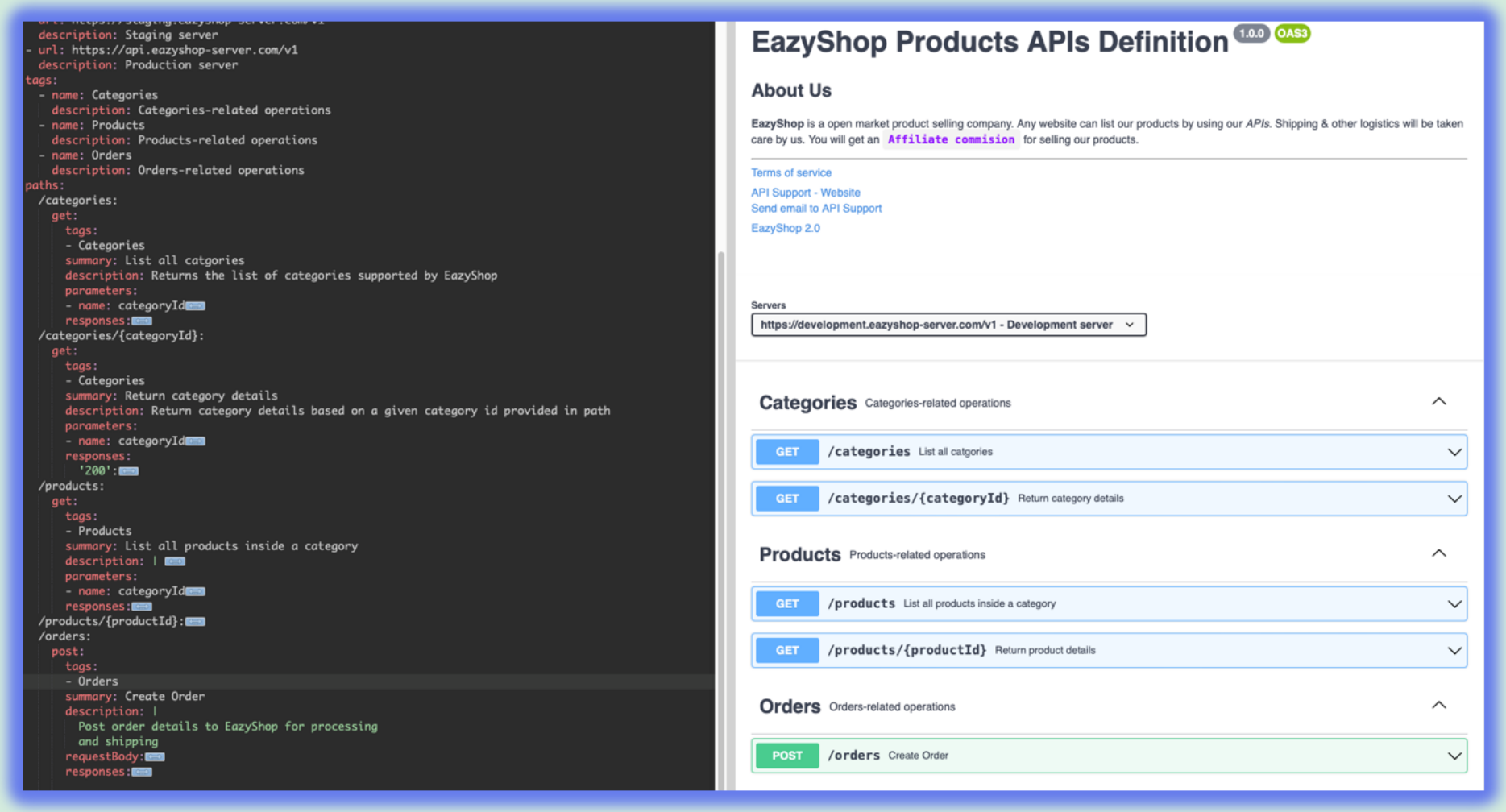
```yaml
paths:
  /categories:
    get:
      summary: List all catgories
      description: Returns the list of categories
      parameters:
      - name: categoryId
        in: query
        schema:
          type: integer
        examples:
          mobiles:
            value: 101
          laptops:
            value: 102
          cameras:
            value: 103
          headphones:
            value: 104
          televisions:
            value: 105
```

Organizing operations with tags

eazy bytes

To help organize operations within an API definition, OpenAPI supports a feature called tags. One or more tags can be added to operations to better categorize and group different operations together.

In Swagger UI these tags will show up as sections, with the relevant operations grouped underneath each section. Using tags is a great way to organize related operations.

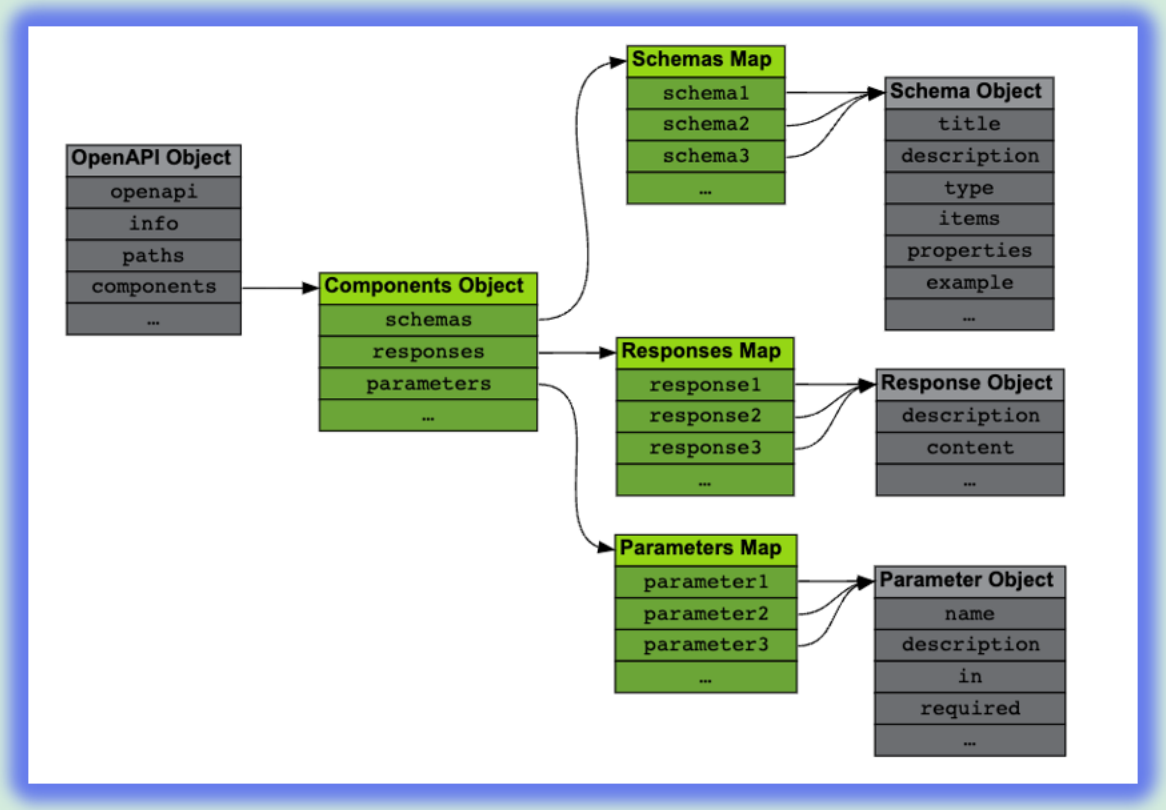


The Components Object

The Components Object, accessible through the components field in the root OpenAPI Object, contains definitions for objects to be reused in other parts of the document.

Most objects in an OpenAPI document can be replaced by a reference to a component, drastically reducing the document's size and maintenance cost (just like methods do in programming languages).

Not all objects can be referenced, though, only those listed as fields of the Components Object like schemas, responses and parameters to name a few.



For example, below we declare a components with schema. We have declared a Address schema object and the same can be referenced using below statement inside requestBody and responses

$ref: '#/components/schemas/Address'

```yaml
components:
  schemas:
    Address:
      type: object
      properties:
        addressLine:
          type: string
          example: 173 Bellevue St.
        city:
          type: string
          example: Sacramento
        state:
          type: string
          example: California
        zipcode:
          type: string
          example: 94536
```

eazy bytes

The Components Object

The following example lists the available subsections inside components object. All subsections are optional.

```
components:
  schemas:
    ...
  parameters:
    ...
  securitySchemes:
    ...
  requestBodies:
    ...
  responses:
    ...
  headers:
    ...
  examples:
    ...
  links:
    ...
  callbacks:
    ...
```

eazy bytes

The Components Object

Individual definitions in components can be specified either inline (as in the previous slides) or using a $ref reference to an external definition like shown below.

```yaml
components:
  schemas:
    Product:
      $ref: '../models/product.yaml'
    User:
      $ref: 'https://example.com/Product.yaml'
  responses:
    GenericError:
      $ref: '../responses-api.yaml#/components/responses/GenericError'
```

eazy bytes

Data Types

The data type of a schema is defined by the type keyword, for example, type: string. OpenAPI defines the following basic types:

- string (this includes dates and files)
- number
- integer
- boolean
- array
- Object

Using these types, you can describe any data structures. Note that there is no null type; instead, the nullable attribute is used as a modifier of the base type.

eazy bytes

Data Type: Numbers

Numbers

OpenAPI has two numeric types, number and integer, where number includes both integer and floating-point numbers.

An optional format keyword serves as a hint for the tools to use a specific numeric type.

Use the minimum and maximum keywords to specify the range of possible values

type	format	Description
number	-	*Any numbers*
number	float	*Floating-point numbers*
number	double	*Floating-point numbers with double precision*
integer	-	*Integer numbers*
integer	int32	*Signed 32-bit integers*
integer	int64	*Signed 64-bit integers (long type)*

eazy bytes

Data Type: Strings

Strings

A string of text is defined as:
type: string

An optional format modifier serves as a hint at the contents and format of the string.

String length can be restricted using minLength and maxLength.

The pattern keyword lets you define a regular expression template for the string value.

```
ssn:
    type: string
    pattern: '^\d{3}-\d{2}-\d{4}$'
```

format	Description
date	*full-date notation, for example, 2017-07-21*
date-time	*the date-time notation, for example, 2017-07-21T17:32:28Z*
password	*a hint to UIs to mask the input*
byte	*base64-encoded characters, for example, U3dhZ2dlciByb2Nrcw==*
binary	*binary data, used to describe files*

However, format is an open value, so you can use any formats like email,uuid,uri,hostname,ipv4,ipv6 etc. even not those defined by the OpenAPI Specification. Tools can use the format to validate the input or to map the value to a specific type in the chosen programming language. Tools that do not support a specific format may default back to the type alone, as if the format is not specified

eazy bytes

Data Type: Boolean, Null & Files

Boolean

type: boolean represents two values: true and false.

Note that truthy and falsy values such as "true", "", 0 or null are not considered boolean values.

Null

OpenAPI 3.0 does not have an explicit null type as in JSON Schema, but you can use nullable: true to specify that the value may be null.

Note that null is different from an empty string "".

Files

Unlike OpenAPI 2.0, Open API 3.0 does not have the file type. Files are defined as strings.

```
type: string
format: binary  # binary file contents

type: string
format: byte    # base64-encoded file contents
```

eazy bytes

Data Type: Objects

Objects

An object is a collection of property/value pairs. The properties keyword is used to define the object properties – you need to list the property names and specify a schema for each property.

```
type: object
properties:
  id:
    type: integer
  name:
    type: string
```

By default, all object properties are optional. You can specify the required properties in the required list:

```
type: object
properties:
  id:
    type: integer
  username:
    type: string
  name:
    type: string
required:
  - id
  - username
```

Note that required is an object-level attribute, not a property attribute. An empty list required: [] is not valid. If all properties are optional, do not specify the required keyword.

eazy bytes

Data Type: Objects

Objects

You can use the readOnly and writeOnly keywords to mark specific properties as read-only or write-only. readOnly properties are included in responses but not in requests, and writeOnly properties may be sent in requests but not in responses.

```
type: object
properties:
  id:
    type: integer
    readOnly: true
  username:
    type: string
  password:
    type: string
    writeOnly: true
```

Free-Form Object allows arbitrary property/value pairs inside an object.

```
type: object
    (or)
type: object
additionalProperties: true
```

The minProperties and maxProperties keywords let you restrict the number of properties allowed in an object. This can be useful when using additionalProperties or free-form objects.

```
type: object
minProperties: 2
maxProperties: 10
```

In OpenAPI, objects are usually defined in the global components/schemas section rather than inline in the request and response definitions.

eazy bytes

Data Type: Arrays

Arrays

Arrays are defined as:

```
type: array
items:
    type: string
```

The items keyword is required in arrays. The value of items is a schema that describes the type and format of array items.

Item schema can be specified inline or referenced via $ref

```
type: array
items:
  $ref: '#/components/schemas/Order'
```

You can define the minimum and maximum length of an array using minItems & maxItems. You can use uniqueItems: true to specify that all items in the array must be unique

```
type: array
items:
  type: integer
minItems: 1
maxItems: 10
uniqueItems: true
```

Arrays can be nested as well like below,

```
# [ [1, 2], [3, 4] ]
type: array
items:
  type: array
  items:
    type: integer
```

Enums

Enums

You can use the enum keyword to specify possible values of a request parameter or a model property.

All values in an enum must adhere to the specified type.

Note that null must be explicitly included in the list of enum values. Using nullable: true alone is not enough here.

For example, the sort parameter in GET /items?sort=[asc|desc] can be described like shown below,

```
paths:
  /items:
    get:
      parameters:
        - in: query
          name: sort
          description: Sort order
          schema:
            type: string
            enum:
              - asc
              - desc
```

eazy bytes

oneOf, anyOf, allOf, not

OpenAPI 3.0 provides several keywords which you can use to combine schemas. You can use these keywords to create a complex schema, or validate a value against multiple criteria.

- ✓ oneOf – validates the value against exactly one of the subschemas
- ✓ allOf – validates the value against all the subschemas
- ✓ anyOf – validates the value against any (one or more) of the subschemas
- ✓ not - use to make sure the value is not valid against the specified schema

Inheritance using allOf

OpenAPI lets you combine and extend model definitions using the allOf keyword. allOf takes an array of object definitions that are used to together compose a single object.

```yaml
Product:
    type: object
    properties:
      productId:
        type: integer
        example: 199
      name:
        type: string
        example: Apple IPhone 13 Pro
      price:
        type: number
        example: 999.00
      quantity:
        type: integer
        example: 1
      brand:
        type: string
        example: Apple
    required: [productId, quantity, networkType]
  Mobile:
    allOf:
      - $ref: '#/components/schemas/Product'
      - type: object
        properties:
          networkType:
            type: string
            enum:
              - 4G
              - 5G
            example: 5G
        required: [networkType]
```

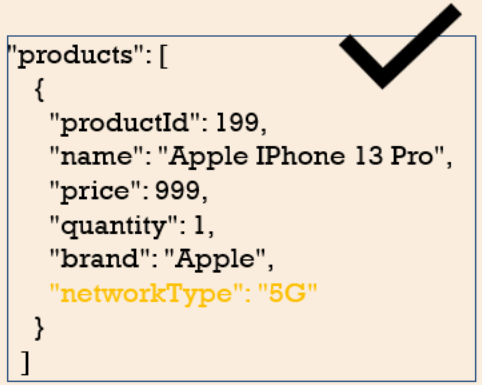
Below is the sample representation of data for Mobile schema. As you can see, all of the properties from Product are inherited into Mobile schema due to allOf

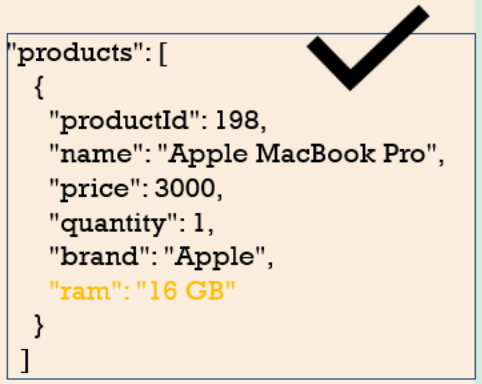
```json
{
  "productId": 199,
  "name": "Apple IPhone 13 Pro",
  "price": 999,
  "quantity": 1,
  "brand": "Apple",
  "networkType": "5G"
}
```

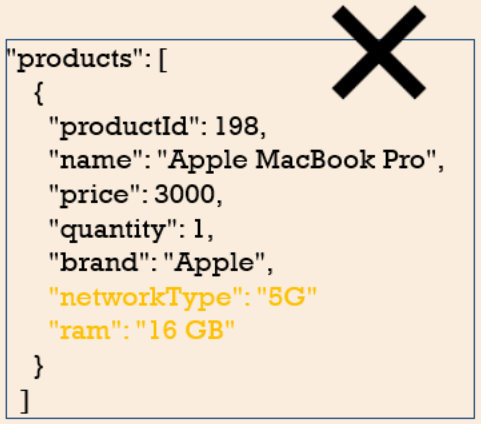
oneOf

Use the oneOf keyword to ensure the given data is valid against one of the specified schemas. In the below scenario, the API will accept either Mobile or Laptop schema as input. Mobile schema has networkType and Laptop Schema has ram as differentiator.

```
/oneOfOrder:
  post:
    tags:
    - Orders
    summary: Insert OneOf the Order Details
    description: Insert OneOf the Order Details into Eazyshop
    requestBody:
      content:
        application/json:
          schema:
            type: object
            properties:
              products:
                type: array
                items:
                  oneOf:
                  - $ref: '#/components/schemas/Mobile'
                  - $ref: '#/components/schemas/Laptop'
              address:
                $ref: '#/components/schemas/Address'
            required:
              - products
              - address
    responses:
      '200':
        description: Order placed successfully
```







anyOf

eazy bytes

Use the anyOf keyword to validate the data against any amount of the given subschemas. That is, the data may be valid against one or more subschemas at the same time. oneOf matches exactly one subschema, and anyOf can match one or more subschemas.

```
/anyOfOrder:
  get:
    tags:
    - Orders
    summary: Get Order Details
    description: Get Order Details based on Order ID
    parameters:
      - $ref: "#/components/parameters/orderIdQueryParam"
      - name: fetchType
        in: query
        required: true
        schema:
          type: string
          enum:
            - summary
            - details
    responses:
      '200':
        description: Return product details
        content:
          application/json:
            schema:
              anyOf:
                - $ref: '#/components/schemas/OrderSummary'
                - $ref: '#/components/schemas/OrderAddress'
```

Based on the Query param fetchType, the response will be only OrderSummary or both OrderSummary + OrderAddress.

```
{
 "products": [
  {
   "productId": 199,
   "name": "Apple IPhone 13 Pro",
   "price": 999,
   "categoryName": "Mobiles",
   "quantity": 1
  }
 ]
}
```

For fetchType: summary

For fetchType: details

```
{
 "products": [
  {
   "productId": 199,
   "name": "Apple IPhone 13 Pro",
   "price": 999,
   "categoryName": "Mobiles",
   "quantity": 1
  }
 ],
 "address": {
  "addressLine": "173 Bellevue St.",
  "city": "Sacramento",
  "state": "California",
  "zipcode": "94536"
 }
}
```

eazy bytes

oneOf, anyOf, allOf, not

Input	Keyword	Valid Ouput
	oneOf - cat - red	
	anyOf - cat - red	
	allOf - cat - red	
	not: cat	

eazy bytes

operationId

operationId is an optional unique string used to identify an operation. If provided, these IDs must be unique among all operations described in your API.

The value mentioned against operationId will be considered for method/function names by Code Generator tools and while defining links between your APIs.

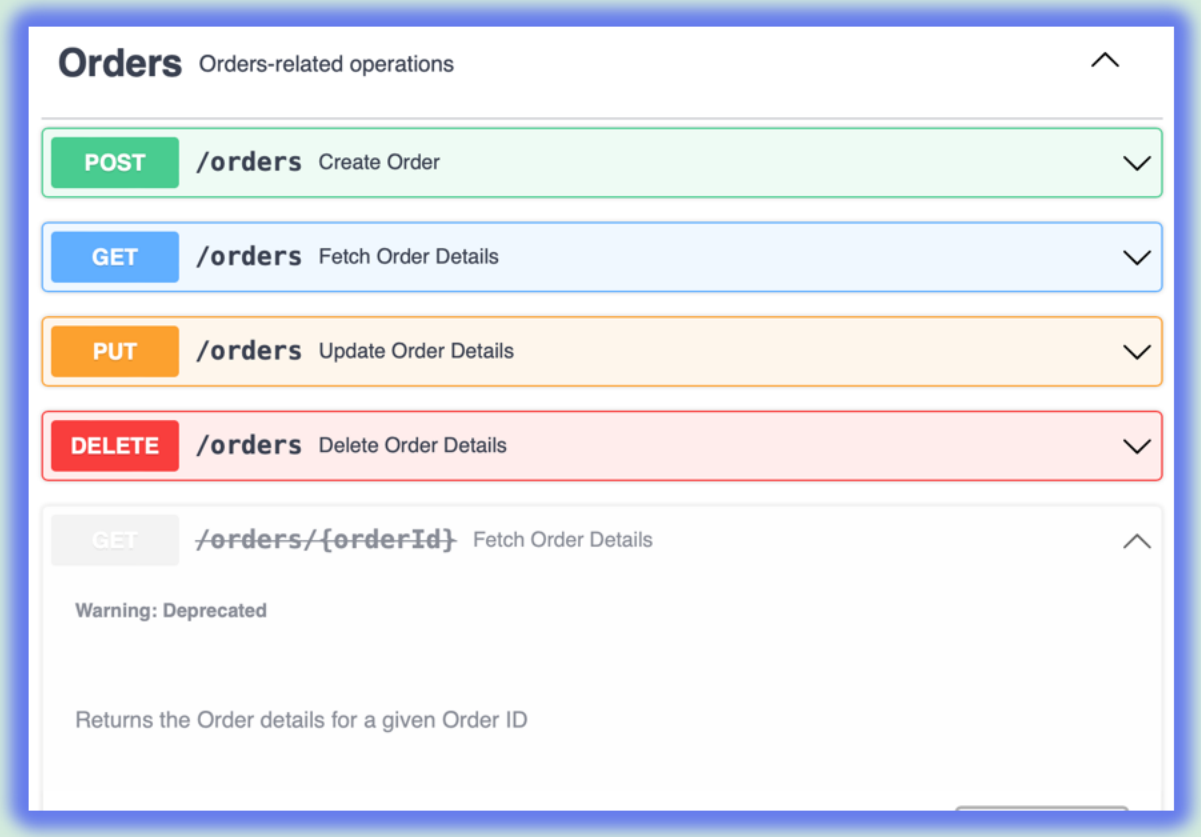
```
/categories:
  get:
    operationId: getCategories
    summary: List all catgories
    ...
/orders:
  post:
    operationId: saveOrder
    summary: Save Order Details
    ...
  put:
    operationId: updateOrder
    summary: Update Order Details
    ...
```

eazy bytes

Deprecated Operations

You can mark specific operations as deprecated to indicate that they will be eventually transitioned out of usage. Tools may handle deprecated operations in a specific way. For example, Swagger UI displays them with a different style like shown below.

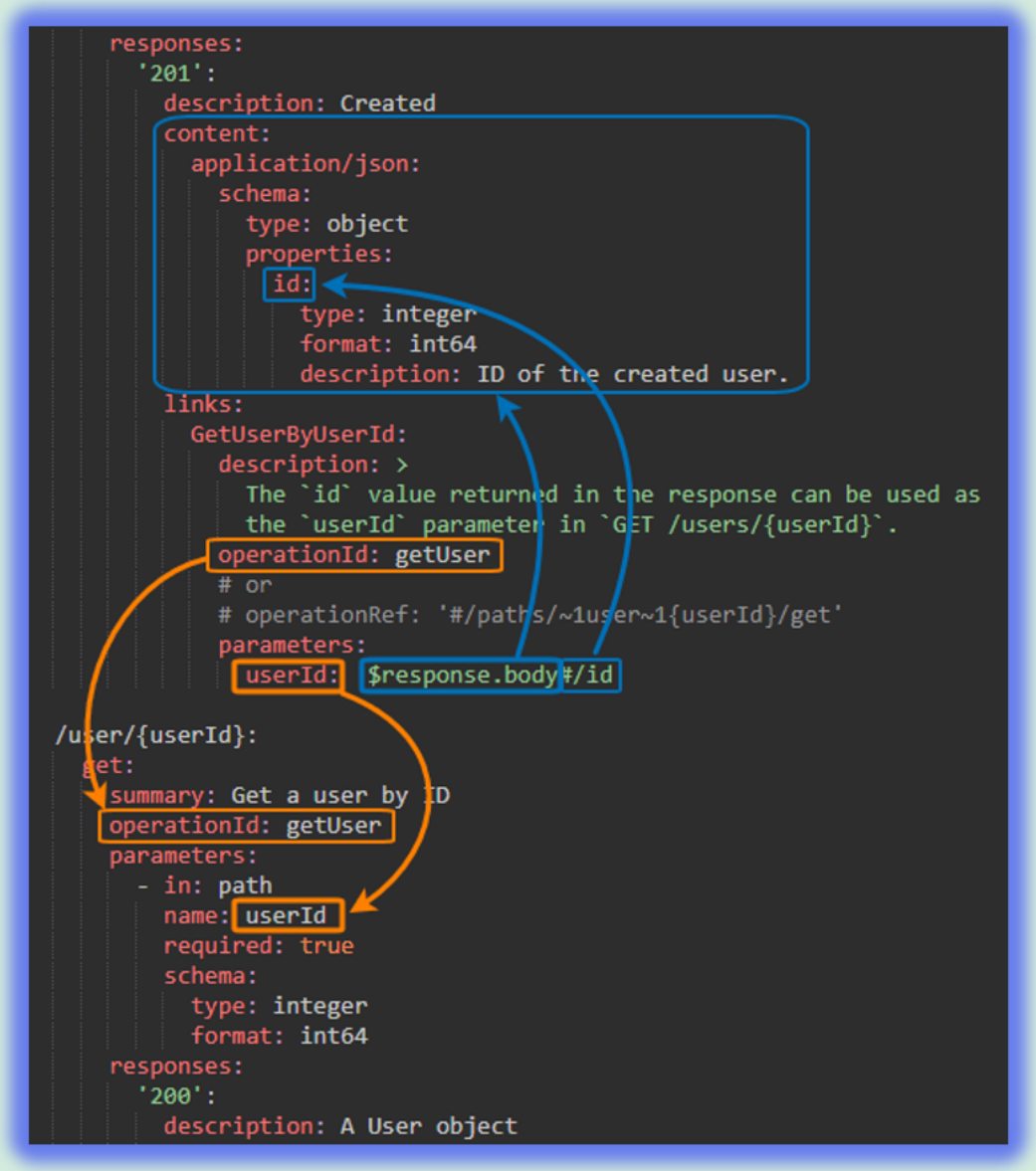
```
/orders/{orderId}:
  get:
    tags:
    - Orders
    summary: Fetch Order Details
    description: Returns the Order details for a given Order ID
    deprecated: true
    parameters:
      - $ref: "#/components/parameters/orderIdPathParam"
    responses:
      '200':
        description: Return product details
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Product'
```



eazy bytes

Links

Links are one of the new features of OpenAPI 3.0. Using links, you can describe how various values returned by one operation can be used as input for other operations. This way, links provide a known relationship and traversal mechanism between the operations.



eazy bytes

OpenAPI Extensions

Extensions

Extensions are custom properties that start with x-. They can be used to describe extra functionality that is not covered by the OAS.

Extensions are supported on the root level of the API spec and in the following places:

- ✓ *info section*
- ✓ *paths section, individual paths and operations*
- ✓ *operation parameters*
- ✓ *responses*
- ✓ *tags*
- ✓ *security schemes*

The extension value can be a primitive, an array, an object or null. If the value is an object or array of objects, the object's property names do not need to start with x-.

```
info:
  version: 1.0.0
  title: Sample title
  description: Sample decription
  x-custom-info:
    comment: Some comments
    authors:
      - name: John Doe
        email: john@doe.com
      - name: Jane Doe
        email: jane@doe.com
```

eazy bytes

External Documentation Object

externalDocs object allows referencing an external resource for extended documentation.

```
openapi: 3.0.3
info:
  ....
servers:
  ....
externalDocs:
  description: Find more info here
  url: https://example.com
paths:
  ....
```

eazy bytes

Security scheme

OpenAPI uses the term security scheme for authentication and authorization schemes. OpenAPI 3.0 lets you describe APIs protected using the following security schemes:

- HTTP authentication schemes (they use the Authorization header):
  - ✓ Basic
  - ✓ Bearer
- API keys in headers, query string or cookies
- OAuth 2
- OpenID Connect Discovery

Security is described using the securitySchemes and security keywords. You use securitySchemes to define all security schemes your API supports, then use security to apply specific schemes to the whole API or individual operations.

Basic Authentication

eazy bytes

Basic authentication is a simple authentication scheme built into the HTTP protocol. The client sends HTTP requests with the Authorization header that contains the word Basic word followed by a space and a base64-encoded string username:password. For example, to authorize as eazybytes/Admin123 the client would send Authorization: Basic ZWF6eWJ5dGVzOkFkbWluMTIz

```
openapi: 3.0.3
...

components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
...
security:
  - BasicAuth: []
```

The first section, securitySchemes, defines a security scheme named BasicAuth (an arbitrary name). This scheme must have type: http and scheme: basic. The security section then applies Basic authentication to the all the APIs globally. The square brackets [] denote the security scopes used; the list is empty because Basic authentication does not use scopes.

```
openapi: 3.0.3
...

components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
...
paths:
  /example:
    get:
      security:
        - BasicAuth: []
```

The security can be set on the operation level as well like shown above

Bearer Authentication

eazy bytes

Bearer authentication (also called token authentication) is an HTTP authentication scheme that involves security tokens called bearer tokens. The name "Bearer authentication" can be understood as "give access to the bearer of this token." The bearer token is a cryptic string, usually generated by the server in response to a login request. The client must send this token in the Authorization header when making requests to protected resources: Authorization: Bearer <token>

```
openapi: 3.0.3
...

components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
...
security:
  - BearerAuth: []
```

Optional bearerFormat is an arbitrary string that specifies how the bearer token is formatted. Since bearer tokens are usually generated by the server, bearerFormat is used mainly for documentation purposes, as a hint to the clients. In the example above, it is "JWT", meaning JSON Web Token. The square brackets [] in bearerAuth: [] contain a list of security scopes required for API calls. The list is empty because scopes are only used with OAuth2 and OpenID Connect.

```
openapi: 3.0.3
...

components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
...
paths:
  /example:
    get:
      security:
        - BearerAuth : []
```

The security can be set on the operation level as well like shown above

API Keys

Some APIs use API keys for authorization. An API key is a token that a client provides when making API calls. The key can be sent in the query string or as a request header or as a cookie.

```
openapi: 3.0.3
...

components:
  securitySchemes:
    ApiKeyAuth:
      type: apiKey
      in: header
      name: X-API-KEY
...
security:
  - ApiKeyAuth: []
```

This example defines an API key named X-API-Key sent as a request header X-API-Key: <key>. The key name ApiKeyAuth is an arbitrary name for the security scheme. The name ApiKeyAuth is used again in the security section to apply this security scheme to the API.

```
openapi: 3.0.3
...

components:
  securitySchemes:
    ApiKeyAuth:
      type: apiKey
      in: header
      name: X-API-KEY
...
paths:
  /example:
    get:
      security:
        - ApiKeyAuth : []
```

The security can be set on the operation level as well like shown above.

OAuth 2.0

eazy bytes

OAuth 2.0 is an authorization protocol that gives an API client limited access to user data on a web server. GitHub, Google, and Facebook APIs notably use it. OAuth relies on authentication scenarios called flows, which allow the resource owner (user) to share the protected content from the resource server without sharing their credentials. For that purpose, an OAuth 2.0 server issues access tokens that the client applications can use to access protected resources on behalf of the resource owner.

```
components:
  securitySchemes:
    oAuth2AuthCode:
      type: oauth2
      flows:
        authorizationCode:
          authorizationUrl: https://example.com/oauth/authorize
          tokenUrl: https://example.com/api/oauth.access
          scopes:
            read: Read only access
            write: Read Write access
            admin: Admin access
security:
  - oAuth2AuthCode :
          - read
          - write
          - admin
```

The flows keyword specifies one or more named flows supported by this OAuth 2.0 scheme. The flow names are:

- ✓ authorizationCode – Authorization Code flow
- ✓ implicit – Implicit flow
- ✓ password – Resource Owner Password flow
- ✓ clientCredentials – Client Credentials flow

The flows object can specify multiple flows, but only one of each type.

OpenID Connect Discovery

eazy bytes

OpenID Connect (OIDC) is an identity layer built on top of the OAuth 2.0 protocol and supported by some OAuth 2.0 providers, such as Google and Azure Active Directory. It defines a sign-in flow that enables a client application to authenticate a user, and to obtain information (or "claims") about that user, such as the user name, email, and so on. User identity information is encoded in a secure JSON Web Token (JWT), called ID token.

```
openapi: 3.0.0
...
components:
  securitySchemes:
    OpenId:
      type: openIdConnect
      openIdConnectUrl: https://example.com/.well-known/openid-configuration
security:
  - OpenId:
      - read
      - write
      - admin
```

- ✓ The first section, components/securitySchemes, defines the security scheme type (openIdConnect) and the URL of the discovery endpoint (openIdConnectUrl). Unlike OAuth 2.0, you do not need to list the available scopes in securitySchemes – the clients are supposed to read them from the discovery endpoint instead.
- ✓ The security section then applies the chosen security scheme to your API. The actual scopes required for API calls need to be listed here. These may be a subset of the scopes returned by the discovery endpoint.
- ✓ If different API operations require different scopes, you can apply security on the operation level instead of globally.

eazy bytes

Hosting

Hosting our API documentation using GitHub pages. Other options including AWS S3 Bucket, GitLab pages, Dropbox, Google Drive etc.

Mock APIs using OpenAPI specification

eazy bytes

Using Mock HTTP Servers, we can build mock APIs based on the OpenAPI specification document. For the same, we can use Prism. Prism is an open-source HTTP mock server that can mimic your API's behavior as if you already built it based on OpenAPI v3/v2 (formerly known as Swagger) documents.

Below are the steps to be followed to mock APIs using Prism,

Step 01

Since Prism is a Node.js CLI tool, download & install Node.js v17+ (https://nodejs.org/en/download/)

Step 02

Install Prism using the below command,

npm install –global @stoplight/prism-cli

Step 03

Start the Prism server using the command,

prism mock ./openapi.yml –p 8080

CodeGeneration

eazy bytes

Moving from design to development has never been easier with Swagger Codegen. API Definition files can be used to create stubs in popular languages, like Java, Python, Scala, and Ruby, with just a few clicks.

Client SDKs

Client code generated by Codegen is a type of software development kit (SDK). In the context of APIs, an SDK is a library that wraps an API so that developers creating an application that integrates the API don't have to build their API calls as HTTP requests. Instead, developers can call an SDK method that almost looks like a method in the standard library of their programming language, and the SDK converts that into an HTTP request.

Server Stubs

Server code generation with Codegen doesn't generate a library but rather a draft version of the structure of a server implementation, with lots of blanks to fill. It's a starting point for developers, or a template, if you will.

Swagger Codegen is always updated with the latest and greatest changes in the programming world

OpenAPI support in Spring (Java framework)

springdoc-openapi java library helps to automate the generation of API documentation using spring boot projects. springdoc-openapi works by examining an application at runtime to infer API semantics based on spring configurations, class structure and various annotations

For the integration between spring-boot and swagger-ui, add the library to the list of your project maven dependencies,

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.6.9</version>
</dependency>
```

This will automatically generates documentation in JSON/YAML and HTML format APIs. This documentation can be completed by comments using swagger-api annotations.

The Swagger UI page will then be available at http://server:port/context-path/swagger-ui.html and the OpenAPI description will be available at the following url for json format:
http://server:port/context-path/v3/api-docs

List of important Annotations that can be used to improve the OpenAPI specifications inside a SpringBoot project,

- ✓ @Tag
- ✓ @Schema
- ✓ @Operation(summary = "foo", description = "bar")
- ✓ @Parameter
- ✓ @ApiResponse(responseCode = "404", description = "foo")

All these annotations can be found inside io.swagger.v3.oas.annotations package

For more details please refer to the below official website,

https://springdoc.org/

eazy
bytes

THANK YOU