OpenAPI 3.1 Cheat Sheet

bump.sh

Document Structure

An OpenAPI document is a JSON or YAML file containing the following root elements:

```yaml
openapi: 3.1    # The spec version
info: {}        # API and document info
servers: {}     # List of available servers
paths: {}       # List of endpoints
webhooks: {}    # List of webhooks
security: {}    # Authentication description
components: {} # Reusable components ($ref)
tags: {}        # Define the grouping tags
```

General Information

```yaml
info:
  title: Your Awesome API
  version: 1.2.14
  description: What our API does is...

servers:
  - url: https://example.com/api
    description: Production
  - url: https://staging.example.com/api
```

API Structure

Describe the different operations that your API exposes - such as POST /things - with the paths statement, and the events emitted by your API with the webhooks statement.

```yaml
paths:
 /things:
   post: # Operation object (HTTP Verb)
     operationId: url-friendly-identifier
     summary: Name of Operation
     description: Longer **with CommonMark!**
     requestBody:
       description: Create a Thing
       content:
         application/json: # Content type
           schema: {} # Schema Object
     responses:
       '201':
         description: "Created"
         content:
           application/json:
             schema: {} # Schema Object

webhooks:
  newThing: # Nickname for webhook
    post: # Operation object (HTTP Verb)
      ...
```

Data Types and Schemas

The most important keyword is type, which should be one of:

`null`	JSON "null" value
`boolean`	JSON true or false value
`object`	JSON object
`array`	Ordered list of instances
`integer`	Integer
`number`	Base-10 decimal number
`string`	String of Unicode code points

```yaml
type: object
properties:
  id:
    type: string
    format: uuid
    readOnly: true
  name:
    type: string
    examples:
      - Bert
```

```yaml
type: array
items:
  type: object
  required:
    - password
  properties:
    password:
      type: string
      writeOnly: true
```

Most tools will filter readOnly properties out of a response body, and writeOnly out of a request body.

Security

Define the APIs Security Schemes, then apply them globally or per operation using the security keyword.

Define security schemes

```yaml
components:
  securitySchemes: # Define for use later
    ApiKey: # Arbitrary name
      type: http
      scheme: bearer
```

Apply security schemes

```yaml
security: # Document's root: apply globally
  - ApiKey: []

paths: # Apply on this operation only
  /widgets:
    post:
      security:
        - ApiKey: []
```

Allowed types

`apiKey`, `http` (basic or bearer), `oauth2`, `mutualTLS`, `openIdConnect`

Reuse Elements

Avoid duplicating elements by defining reusable components:

```yaml
components:
  securitySchemes: {}
  requestBodies: {}
  responses: {}
  schemas: {}
  ...
```

Use your components with the $ref keyword:

```yaml
paths:
  /widgets:
    responses:
      '404':
        $ref: #/components/responses/404
```

Components can be reached:

internally: `#/components/schemas/User`
through a remote URL: `https://example.com/user.yml`
on file system: `./user.yml#/components/schemas/User`

Polymorphism

Combine several schemas using polymorphism statements:

`oneOf:`	Exactly one of the schemas (XOR)
`anyOf:`	One or more of the schemas (OR)
`allOf:`	All the schemas (AND)

```yaml
schema:
  allOf: # An admin user
    - $ref: '#/components/schemas/User'
    - $ref: '#/components/schemas/Admin'
```

Grouping and sorting

Group operations with metadata using tags.
Define them globally then apply them per operation.

```yaml
tags:
  - name: Things
    description: >
      This is my thing **description**.

paths:
   /things:
      tags: [Things]
```

Most tools will sort your documentation endpoints according to your global tags written order.