What Is Swagger?

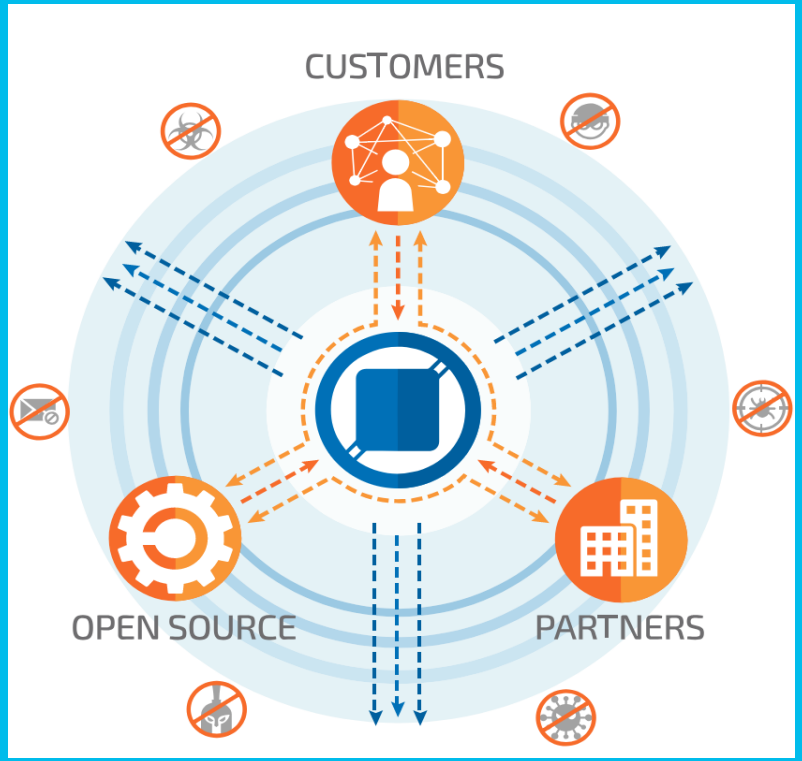
Jay Chow

Software Engineer, Cisco Operations, IT, eCommerce Engineering

Cisco, San Jose

1st July 2020

Agenda



1. What Is Swagger?
2. Why Swagger?
3. After getting Swagger spec
4. Kubernetes
5. Microservices
6. Kubernetes with microservices architecture
7. What's next for us?
8. Q & A
9. References

What Is Swagger?

- Swagger allows you to describe the structure of your APIs so that machines can read them.

- The ability of APIs to describe their own structure is the root of all awesomeness in Swagger. Why is it so great? Well, by reading your API's structure, we can automatically build beautiful and interactive API documentation. We can also automatically generate client libraries for your API in many languages and explore other possibilities like automated testing.

- Swagger does this by asking your API to return a YAML or JSON that contains a detailed description of your entire API. This file is essentially a resource listing of your API which adheres to OpenAPI Specification.

- The specification asks you to include information like:
  1) What are all the operations that your API supports?
  2) What are your API's parameters and what does it return?
  3) Does your API need some authorization?
  4) And even fun things like terms, contact information and license to use the API.

What Is Swagger? (cont'd)

- You can write a Swagger spec for your API manually, or have it generated automatically from annotations in your source code
- Check swagger.io/open-source-integrations for a list of tools that let you generate Swagger from code.
- There are a few ways in which Swagger can help drive your API development further: Design-first users: use Swagger Codegen to **generate a server stub** for your API. The only thing left is to implement the server logic – and your API is ready to go live!
- Use Swagger Codegen to **generate client libraries** for your API in over 40 languages.
- Use Swagger UI to generate **interactive API documentation** that lets your users try out the API calls directly in the browser.
- Use Swagger Editor, a browser-based editor where you can write OpenAPI specs
- Use the spec to connect API-related tools to your API. For example, import the spec to SoapUI to create automated tests for your API.

What Is OpenAPI and why use OpenAPI?

- **OpenAPI Specification** (formerly Swagger Specification) is an API description format for REST APIs.

- An OpenAPI file allows you to describe your entire API, including:
  - Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
  - Operation parameters Input and output for each operation
  - Authentication methods
  - Contact information, license, terms of use and other information

- Use OpenAPI as the ability of APIs to describe their own structure is powerful

Basic Structure

- Write OpenAPI definitions in either YAML(below) or JSON, keywords are sensitive:

```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9

servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing

paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Metadata

- Every API definition must include the version of the OpenAPI Specification that this definition is based on:

```
openapi: 3.0.0
```

- The OpenAPI version defines the overall structure of an API definition – what you can document and how you document it. OpenAPI 3.0 uses semantic versioning with a three-part version number. The available versions are 3.0.0, 3.0.1, 3.0.2, and 3.0.3; they are functionally the same. The info section contains API information: title, description (optional), version:

```
info:
  title: Sample API
  description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9
```

Metadata (cont'd)

- title is your API name.
- description is extended information about your API.
- It can be multiline and supports the CommonMark dialect of Markdown for rich text representation. HTML is supported to the extent provided by CommonMark. version is an arbitrary string that specifies the version of your API (do not confuse it with file revision or the openapi version).
- You can use semantic versioning like *major.minor.patch*, or an arbitrary string like *1.0-beta* or *2017-07-25*. info also supports other keywords for contact information, license, terms of service, and other details.

Servers and Base URL

- The servers section specifies the API server and base URL. You can define one or several servers, such as production and sandbox

```
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
```

- All API paths are relative to the server URL. In the example above, /users means http://api.example.com/v1/users or http://staging-api.example.com/users, depending on the server used.

- All API endpoints are relative to the base URL. For example, assuming the base URL of https://api.example.com/v1, the /users endpoint refers to https://api.example.com/v1/users.

```
https://api.example.com/v1/users?role=admin&status=active
\_____________________/\____/ \_____________________/
        server URL       endpoint    query parameters
                           path
```

Paths

- The servers section specifies the API server and base URL. You can define one or several servers, such as production and sandbox

```yaml
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
```

- All API paths are relative to the server URL. In the example above, /users means http://api.example.com/v1/users or http://staging-api.example.com/users, depending on the server used.

```yaml
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML
      responses:
        '200':
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Parameters

- An operation definition includes parameters, request body (if any), possible response status codes (such as 200 OK or 404 Not Found) and response contents.

- Operations can have parameters passed via URL path (/users/{userId}), query string (/users?role=admin), headers (X-CustomHeader: Value) or cookies (Cookie: debug=0). You can define the parameter data types, format, whether they are required or optional, and other details:

```
paths:
  /user/{userId}:
    get:
      summary: Returns a user by ID.
      parameters:
        - name: userId
          in: path
          required: true
          description: Parameter description in CommonMark or HTML.
          schema:
            type : integer
            format: int64
            minimum: 1
      responses:
        '200':
          description: OK
```

Request Body

- If an operation sends a request body, use the requestBody keyword to describe the body content and media type.

```yaml
paths:
  /users:
    post:
      summary: Creates a user.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                username:
                  type: string
      responses:
        '201':
          description: Created
```

Responses

- For each operation, you can define possible status codes, such as 200 OK or 404 Not Found, and the response body schema. Schemas can be defined inline or referenced via $ref. You can also provide example responses for different content types:

```yaml
paths:
  /user/{userId}:
    get:
      summary: Returns a user by ID.
      parameters:
        - name: userId
          in: path
          required: true
          description: The ID of the user to return.
          schema:
            type: integer
            format: int64
            minimum: 1
      responses:
        '200':
          description: A user object.
          content:
            application/json:
              schema:
                type: object
                properties:
                  id:
                    type: integer
                    format: int64
                    example: 4
                  name:
                    type: string
                    example: Jessica Smith
        '400':
          description: The specified user ID is invalid (not a number).
        '404':
          description: A user with the specified ID was not found.
        default:
          description: Unexpected error
```

Input and Output Models

- The global components/schemas section lets you define common data structures used in your API. They can be referenced via $ref whenever a schema is required – in parameters, request bodies, and response bodies. For example, this JSON object:

```json
{
  "id": 4,
  "name": "Arthur Dent"
}
```

can be represented as:

```yaml
components:
  schemas:
    User:
      properties:
        id:
          type: integer
        name:
          type: string
      # Both properties are required
      required:
        - id
        - name
```

Input and Output Models (cont'd)

... and referenced in the request body schema and response body schema as:

```yaml
paths:
  /users/{userId}:
    get:
      summary: Returns a user by ID.
      parameters:
        - in: path
          name: userId
          required: true
          type: integer
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
  /users:
    post:
      summary: Creates a new user.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '201':
          description: Created
```

Authentication

The securitySchemes and security keywords are used to describe the authentication methods used in your API:

```
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic

security:
  - BasicAuth: []
```

Supported authentication methods are:

HTTP authentication: Basic, Bearer, and so on.
API key as a header or query parameter or in cookies
OAuth 2
OpenID Connect Discovery

References

https://swagger.io/docs/specification/about/
https://swagger.io/docs/specification/api-host-and-base-path/