Washington
Systems
Center

API Development Overview (REST APIs for z/OS Connect)

John J. Brefach

Washington System Center

John.J.Brefach@ibm.com

Agenda

- API Overview (REST APIs)
- API Maturity Model
- z/OS Connect
- API Development Lifecycle
- Design First vs Code First
- OpenAPI Specification
- Swagger Tooling
- API Status Codes

Notes and Disclaimers

- The information in this presentation was derived from various IBM & product documentation websites, individual research, as well as personal experience.
- Some slides that you will see are from the z/OS Connect Introduction Wildfire Workshop, you can find more information about this workshops and other workshops on this GitHub Site: https://ibm.biz/zCEEWorkshopMaterial. These slides will be marked with a Wildfire Workshop Logo as you see in the top right-hand corner.
- This educational package will be an introduction to API Development, YAML, OpenAPI specification, Swagger tooling, and z/OS Connect. It is recommended for users with little to moderate level of experience in these topics.
- You may have used or heard of the term Swagger with the use of APIs. As the use of APIs has grown this term has become in some respects misleading. To be more precise, OpenAPI refers to the API specifications (OpenAPI 2 and OpenAPI 3) where Swagger refers to the tooling used to implement the specifications.

API Overview

Overview of APIs, specifically REST APIs, and what makes a REST API “RESTful”

What is an API?

-An API is a set of rules that define how applications or devices can connect to and communicate with each other

-API stands for Application Programming Interface

-APIs act as an intermediary layer that processes data transfers between systems, letting companies open their application data and functionality to external third-party developers, business partners, and internal departments within their companies.

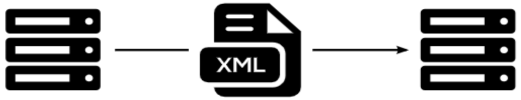
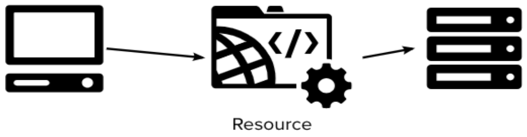
https://www.ibm.com/topics/api

Types of APIs

Open APIs	Open-source application programming interfaces you can access with the HTTP protocol (Known as Public APIs w/ defined API endpoints and request & response formats)
Partner APIs	Connect strategic business partners (Typically, developers access these APIs in self-service mode through a public API developer portal. Still need onboarding and login credentials to access)
Internal APIs	Hidden from external users (aren't available for users outside of the company & are intended to improve productivity and communication across different internal development teams. Known as Private APIs)
Composite APIs	Combine multiple data or service APIs (Allow programmers to access several endpoints in a single call. Useful in microservices architecture where performing a single task may require information from several sources)

https://www.ibm.com/topics/api

API Architecture Styles

Style	Visual Representation	Use Cases
SOAP (Simple Object Access Protocol)		XML-based for enterprise applications
REST (Representational State Transfer)		Resource-based for web servers

Overview of REST APIs

- A REST API is an API that conforms to the design principles of the REST architectural style
- REST is used for accessing and updating resources over the internet using Hyper Text Transfer Protocol.
  - Allows you to access resources without necessarily knowing where exactly they reside.
- RESTful APIs are mostly comprised of HTTP methods that have well-defined and unique actions for any resources

https://www.ibm.com/topics/rest-apis

Key Principles of the REST API

Use Path and Query parameters to refine the request

Use HTTP verbs for Create, Read, Update, Delete (CRUD) operations

```
POST
GET        http://<host>:<port>/path/parameter?name=value&name=value
PUT
DELETE
```

URI path identifies a resource (or lists of resources)

URL identifies the protocol, host and port and includes the URI Path

Request/Response Body is used to represent the data object

```
GET http://www.acme.com/customers/12345?personalDetails=true
RESPONSE: HTTP 200 OK
BODY { “id” : 12345
       “name” : “Joe Bloggs”,
       “address” : “10 Old Street”,
       “tel” : “01234 123456”,
       “dateOfBirth” : “01/01/1980”,
       “maritalStatus” : “married”,
       “partner” : “http://www.acme.com/customers/12346” }
```

REST vs RESTful

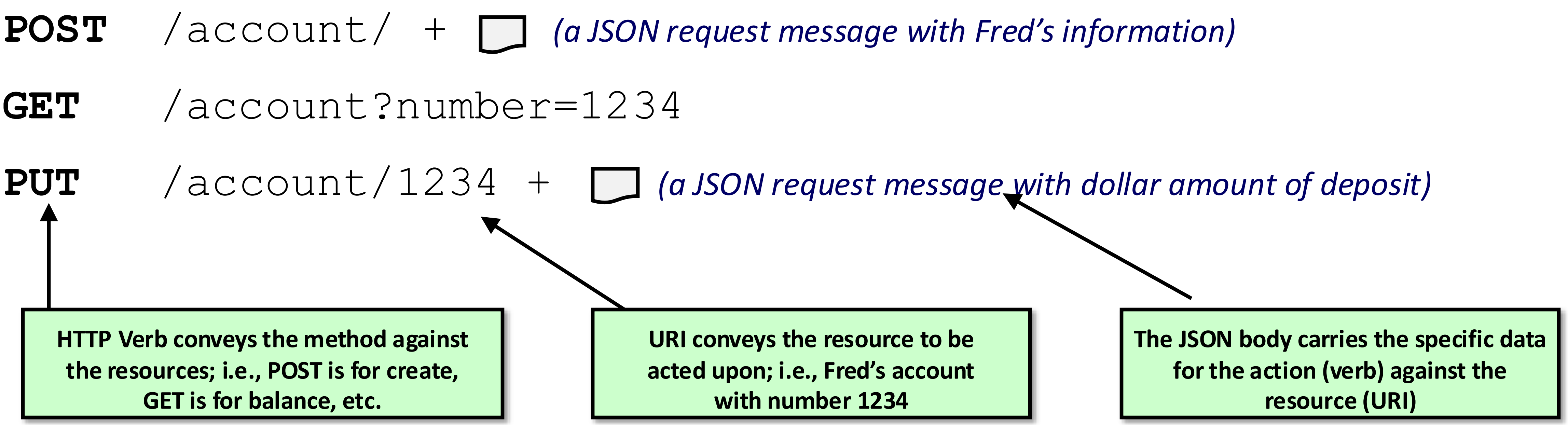
REST is an architectural style of development having these principles plus..

- It should be stateless (transaction management should be managed by the client)
- It should access and/or identify all server resources using only a single URI
- For performing CRUD operations, it should use HTTP verbs such as get, post, put and delete
- It should return the result only in the form of consistent and simple JSON

When an API follows these basic principles, it is considered a RESTful API, whereas a REST API only follows some but not all the above principles

- Remember - Not all REST APIs are RESTful APIs
- The key is consistency, RESTful APIs are consistent with these basic principles, REST APIs are not

RESTful Examples



```
POST   /account/ + □ (a JSON request message with Fred's information)
GET    /account?number=1234
PUT    /account/1234 + □ (a JSON request message with dollar amount of deposit)
```

HTTP Verb conveys the method against the resources; i.e., POST is for create, GET is for balance, etc.

URI conveys the resource to be acted upon; i.e., Fred's account with number 1234

The JSON body carries the specific data for the action (verb) against the resource (URI)

REST APIs are increasingly popular as an integration pattern because it is stateless, relatively lightweight, is relatively easy to program

https://martinfowler.com/articles/richardsonMaturityModel.html

Not every REST API is a RESTful API

(How to know if an API is not RESTful)

1. Different URIs with the same method for operations on the same object

```
POST http://www.acme.com/customers/GetCustomerDetails/12345
POST http://www.acme.com/customers/UpdateCustomerAddress/12345?address=
```

2. Different representations of the same objects between request and response messages

```
POST http://www.acme.com/customers
BODY { “firstName”:  “Joe”,
       “lastName”  :  “Bloggs”,
       “addr”       :  “10 Old Street”,
       “phoneNo”    :  “01234 0123456” }
```

```
RESPONSE HTTP 201 CREATED
BODY { “id”       : “12345”,
       “name”     : “Joe Bloggs”,
       “address”:  “10 New Street”
       “tel”      :  “01234 0123456”}
```

3. Operational data (update, etc.) embedded in the request body

```
POST http://www.acme.com/customers/12345
BODY { “updateField”:  “address”,
       “newValue”     :  “10 New Street”}
```

```
RESPONSE HTTP 200 OK
BODY { “id”       : “12345”,
       “name”     : “Joe Bloggs”,
       “address”: “10 New Street”
       “tel”      : “01234 123456” }
```

The goal is to strive to design API to use RESTful properties

1. Use the same URIs for the same resource with the appropriate method for operations

```
GET   http://www.acme.com/customers/12345
PUT   http://www.acme.com/customers/12345?address=10%20New%20Street
```

2. Use the same JSON property names between request and response messages

```
POST http://www.acme.com/customers/12345
BODY { “name”:  “Joe Bloggs”,
       “address”:  “10 Old Street”,
       “phoneNo”:  “01234 0123456”  }
```

→

```
RESPONSE HTTP 201
BODY { “id”        :  “12345”,
       “name”      :  “Joe Bloggs”,
       “address”:  “10 New Street”
       “phoneNo”:  “01234 0123456”}
```

3. Use JSON name/value pairs

```
PUT http://www.acme.com/customers/12345
BODY { “address”     :  “10 New Street”}
```

→

```
RESPONSE HTTP 200 OK
```

API Maturity Levels

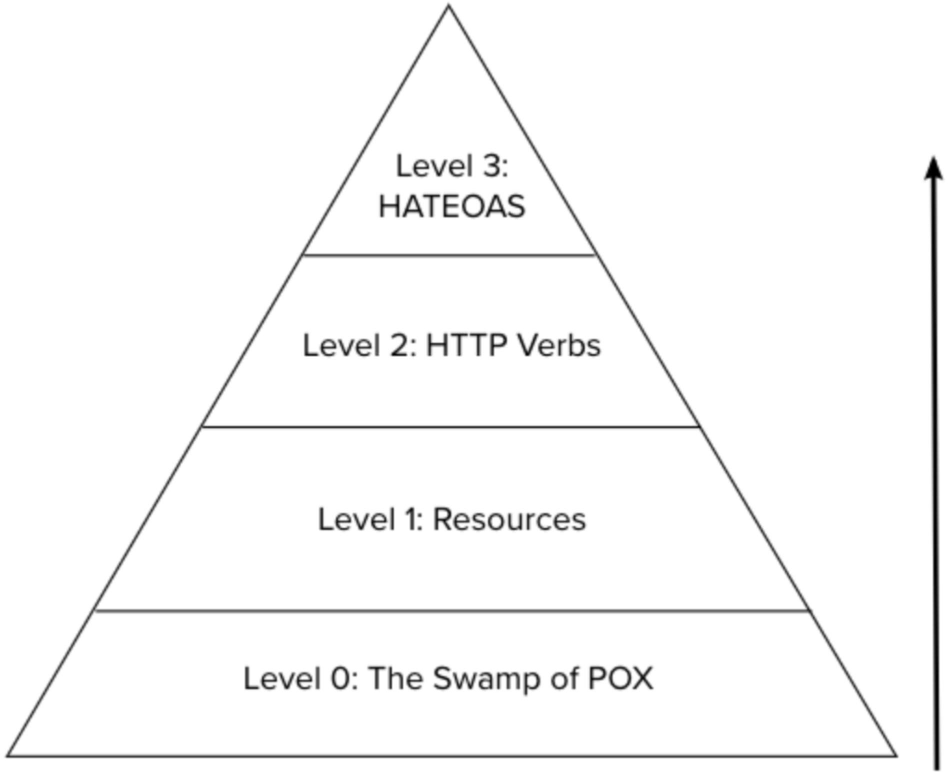
The Richardson Maturity Model (RMM)

The Richardson Maturity Model (RMM)

The **Richardson Maturity Model (RMM)**, introduced by Leonard Richardson in 2008, is a valuable tool to help understand the concept of API maturity and how well an API conforms to the REST concepts.

- The RMM offers an effective framework to help us better understand and implement RESTful principles in our API design

- The main factors that decide the maturity of a service are its URI (Uniform Resource Indicator), HTTP methods, and HATEOAS (Hypermedia as the Engine of Application State).

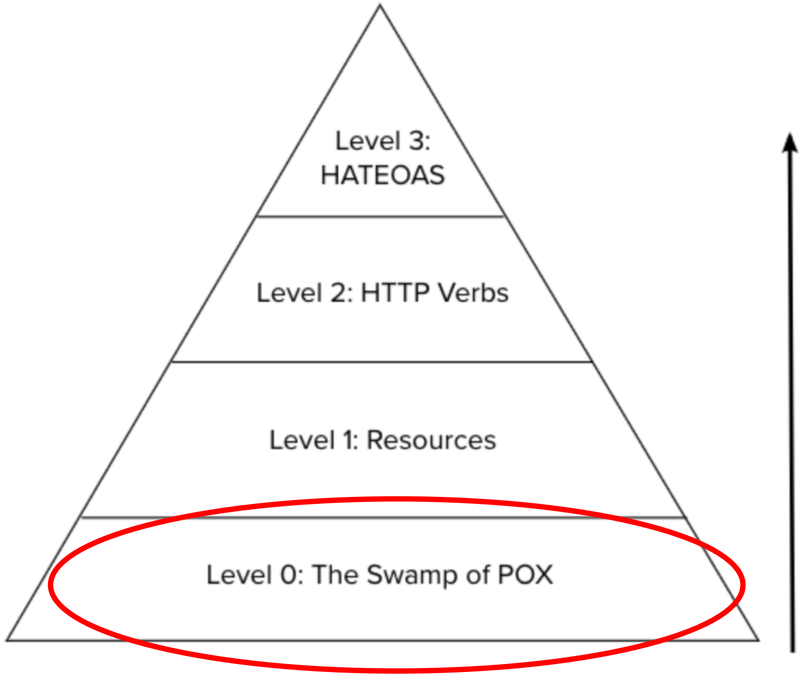


Richardson Maturity Model

https://restfulapi.net/richardson-maturity-model/

Level 0: The Swamp of POX

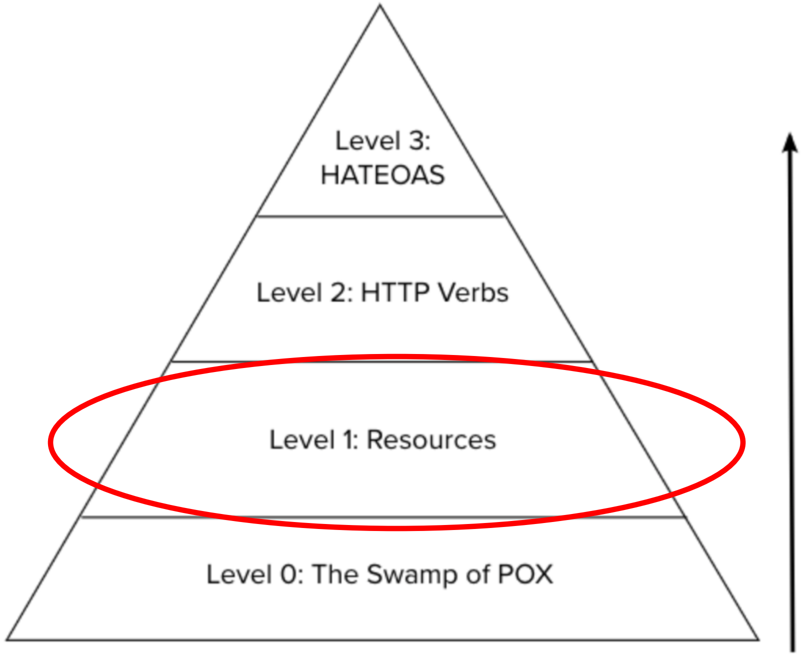
- At Level 0, APIs use a single URI and a single HTTP Method (usually a POST Method)
  - This approach does not leverage the true capabilities of the HTTP protocol and lacks a uniform way to interact with system resources
  - “The Swamp of POX (Plain Old XML)” due to its simplistic, RPC-style system.



Richardson Maturity Model

Level 1: Resources

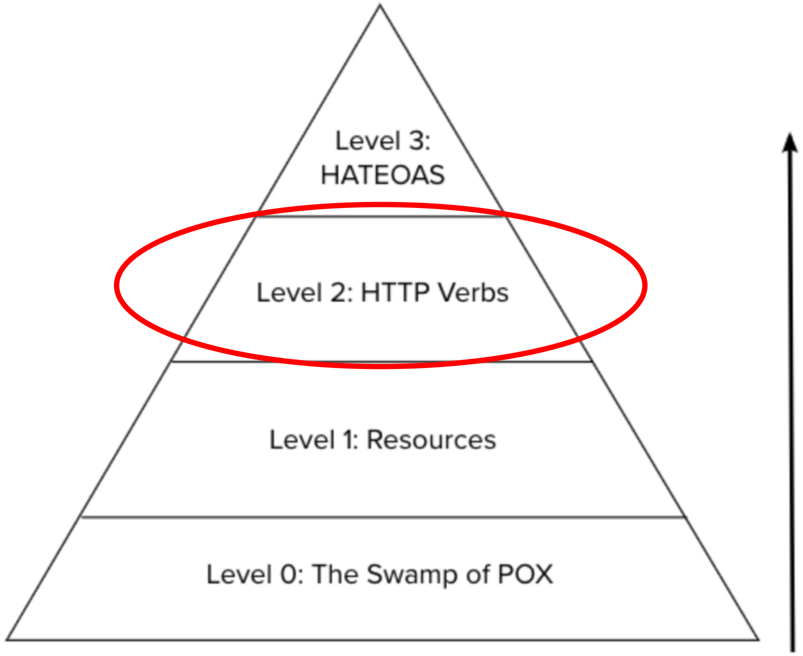
- Level 1 introduces the concept of resources
  - Each resource is uniquely identified by a URI
    - Creating an easier way to manage and interact with different elements of a system
  - It still uses only one HTTP method, POST, limiting the full potential of REST



Richardson Maturity Model

Level 2: HTTP Verbs

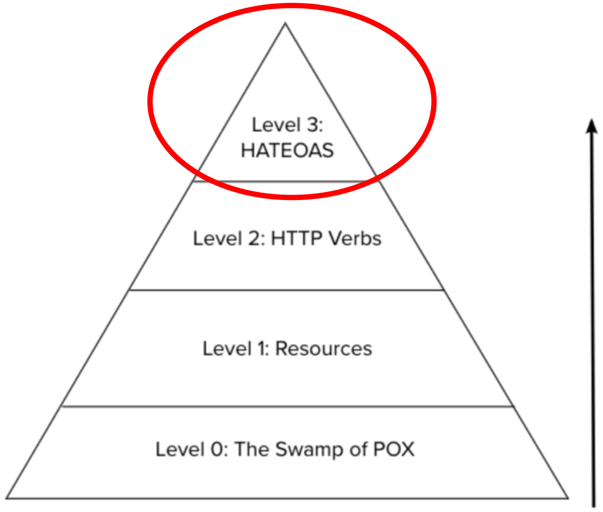
- Level 2 represents an advancement in RESTful design.
  - The services at this level use unique URIs for resources and take advantage of different HTTP methods that correspond to operations on these resources
  - This approach makes our APIs more intuitive and aligns them more closely with the principles of the web.



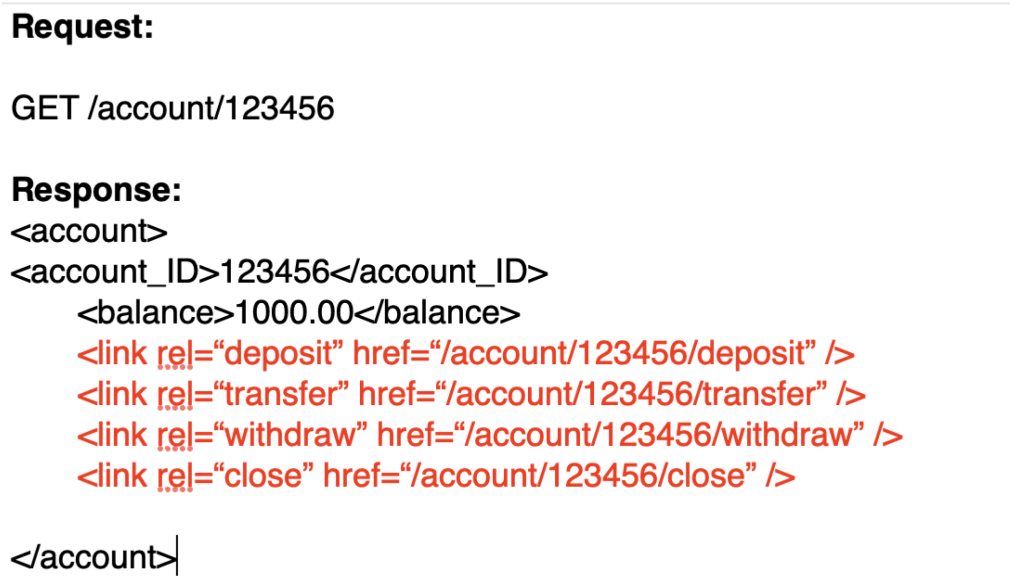
Richardson Maturity Model

Level 3: HATEOAS

- Level 3 brings in the concept of HATEOAS
  - When a client interacts with a resource, the API provides information about the resource itself as well as related resources and possible actions, all represented through hypermedia links



Richardson Maturity Model



Request:

```
GET /account/123456
```

Response:

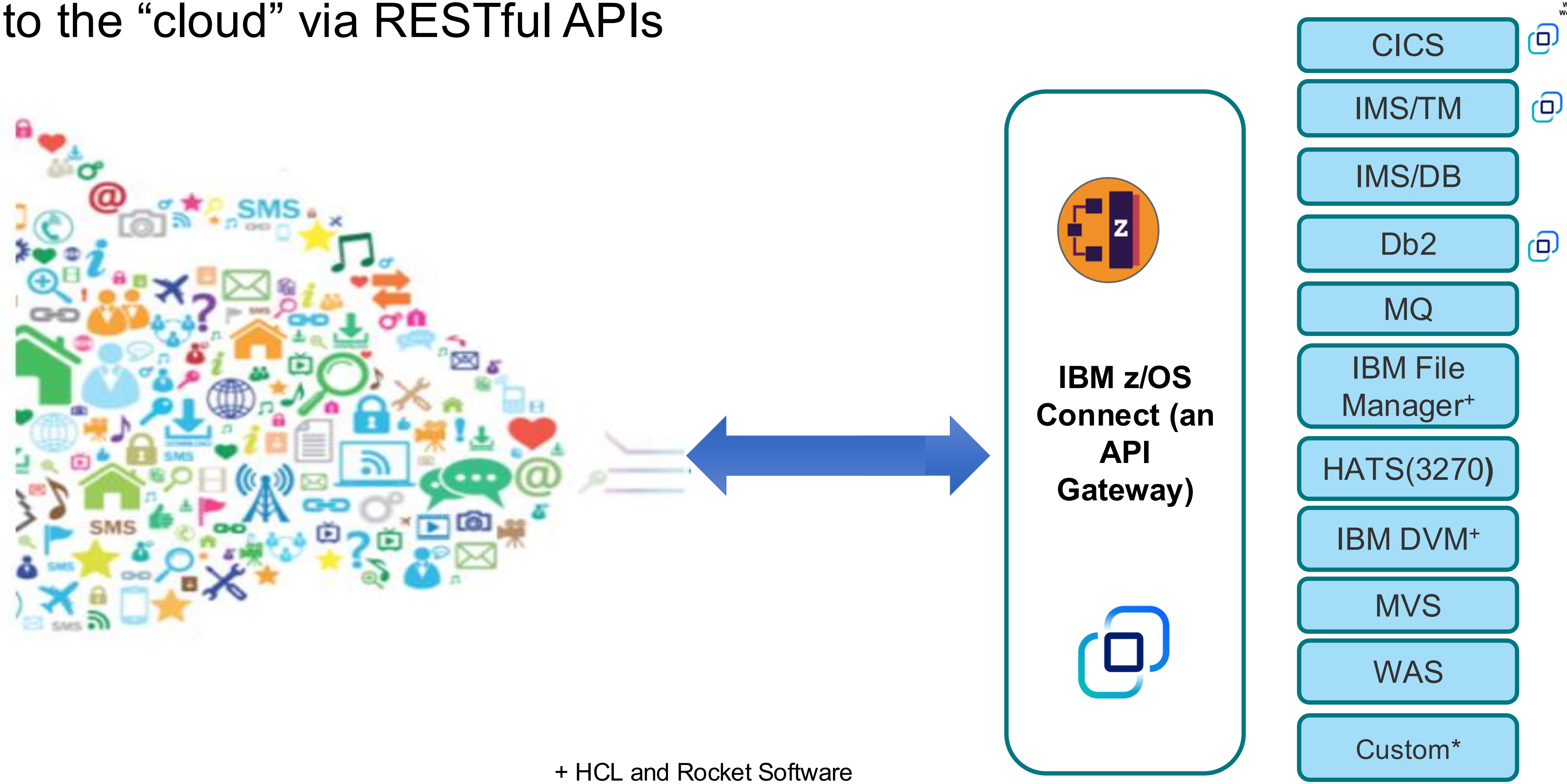
```
<account>
<account_ID>123456</account_ID>
      <balance>1000.00</balance>
      <link rel="deposit" href="/account/123456/deposit" />
      <link rel="transfer" href="/account/123456/transfer" />
      <link rel="withdraw" href="/account/123456/withdraw" />
      <link rel="close" href="/account/123456/close" />

</account>
```

Intro to z/OS Connect

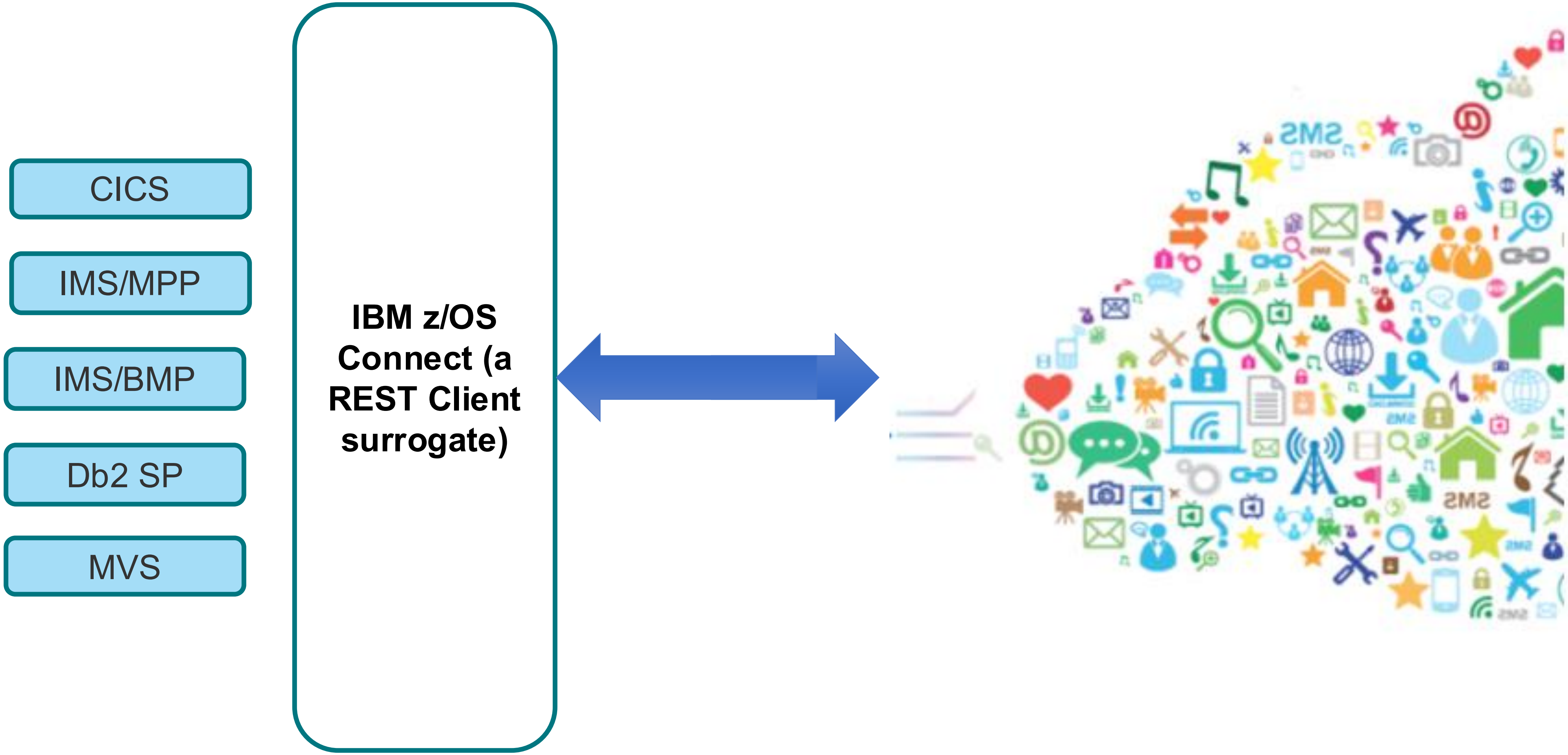
How does z/OS Connect utilize REST APIs

z/OS Connect EE exposes z/OS resources to the “cloud” via RESTful APIs



+ HCL and Rocket Software
*Other Vendors or your own implementation

z/OS Connect EE exposes external REST APIs in the “cloud” to z/OS applications


CICS

IMS/MPP

IMS/BMP

Db2 SP

MVS

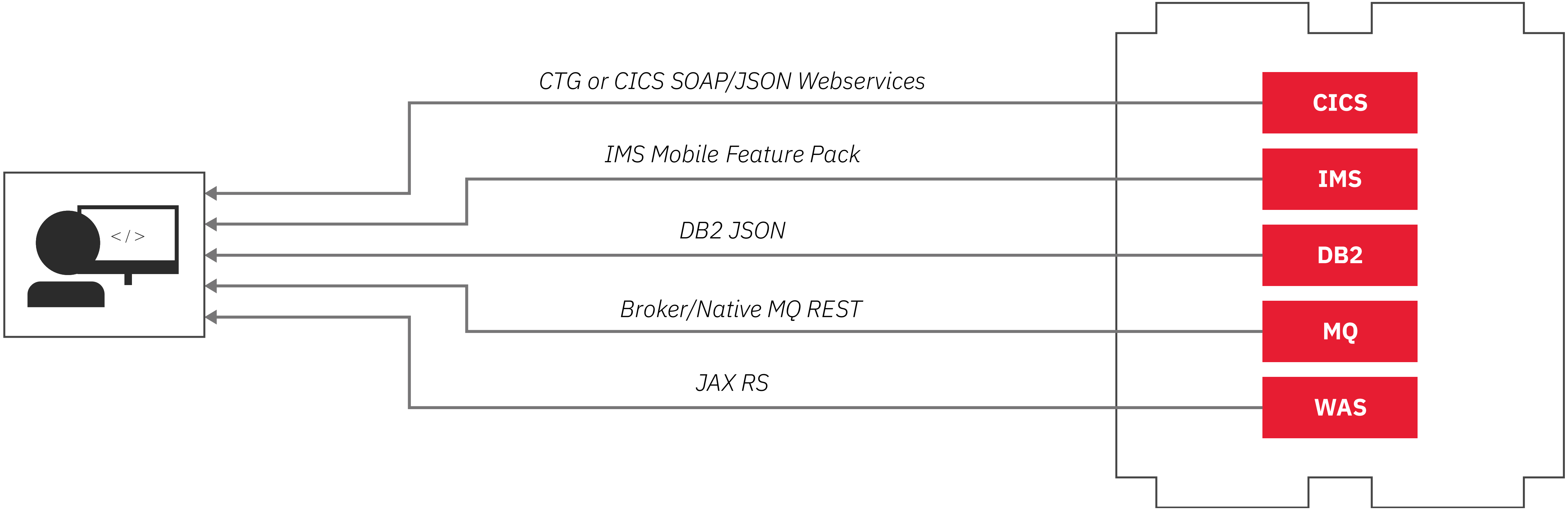
IBM z/OS Connect (a REST Client surrogate)

There was support for REST before z/OS Connect but..

Completely different configuration and management.

Multiple endpoints for developers to call/maintain access to.

These are typically not RESTful!



z/OS Connect provides a single-entry point

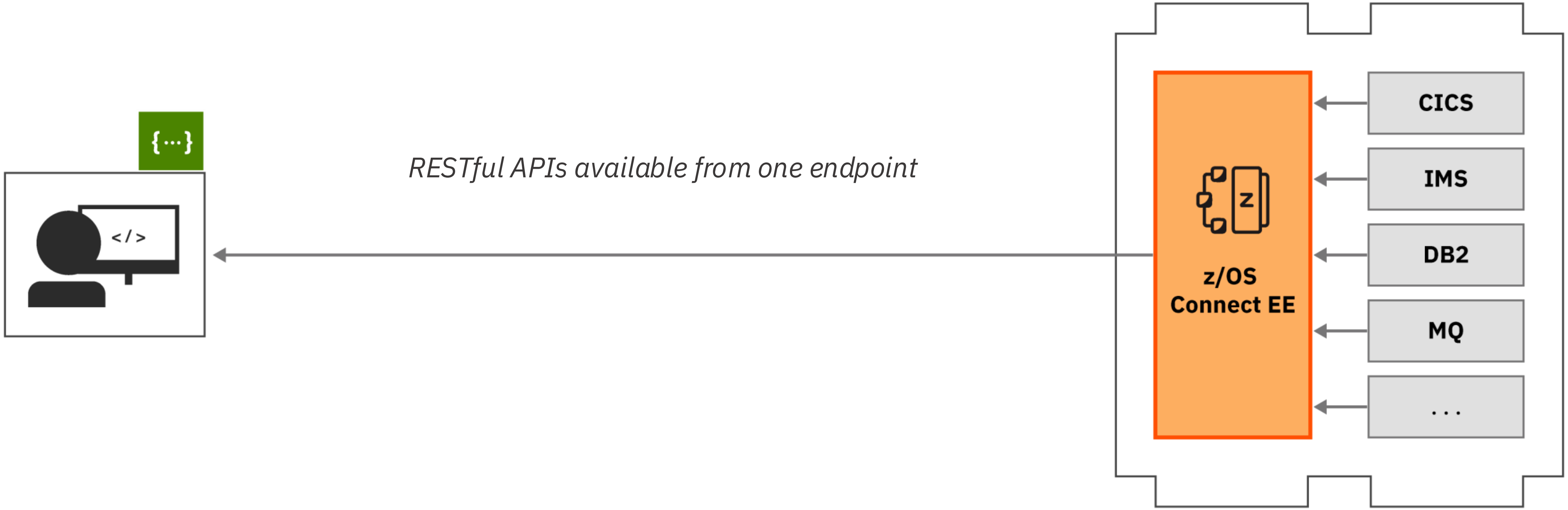
- And exposes z/OS resources without writing any code.

z/OS Connect EE provides

❑ Single Configuration Administration

❑ Single Security Administration

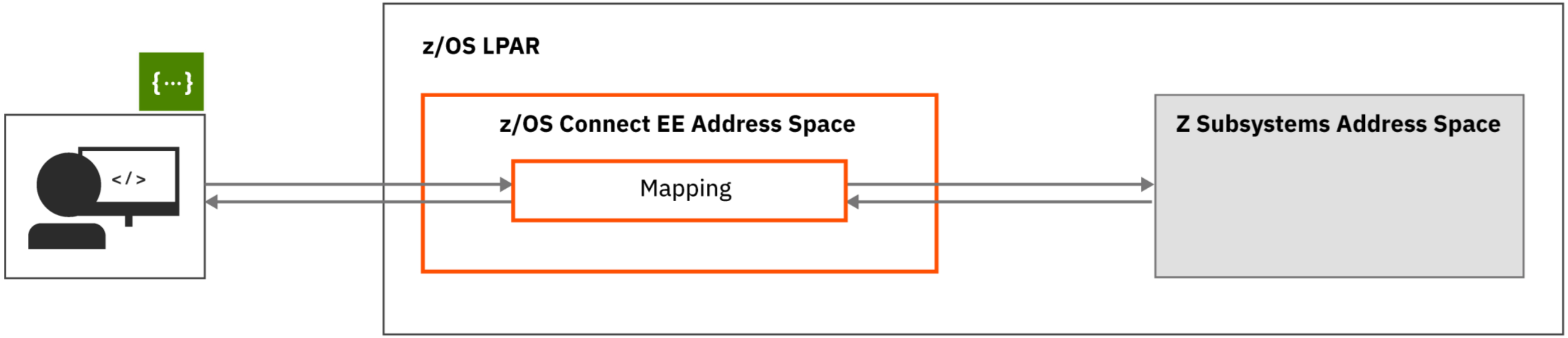
❑ With sophisticated mapping of truly RESTful APIs to existing mainframe and services data without writing any code.



John.J.Brefach@ibm.com

Data mapping/transformation

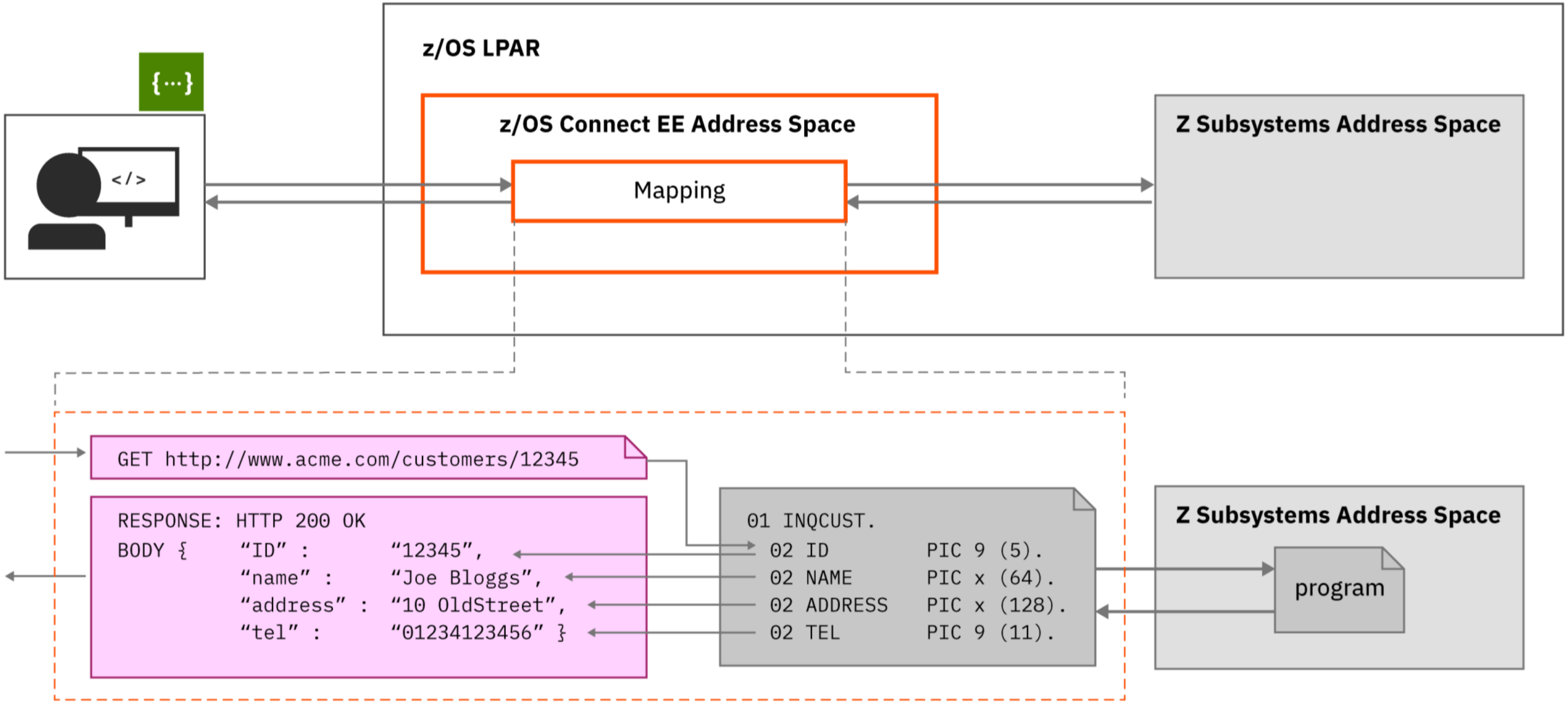
- Converting the JSON message to the format the target's subsystem expects* .



* Most z/OS subsystems depend information in a serial data format and do not normally work with JSON request/response messages. Examples of serialized messages are CICS COMMAREAs and CONTAINERS, IMS or MQ messages, or records stored in sequential or VSAM data sets. Data mapping and transformation refers to the process of converting JSON messages to a serialized layout (e.g., sequentially arranged in storage).

Data mapping and transformation example

- A closer look

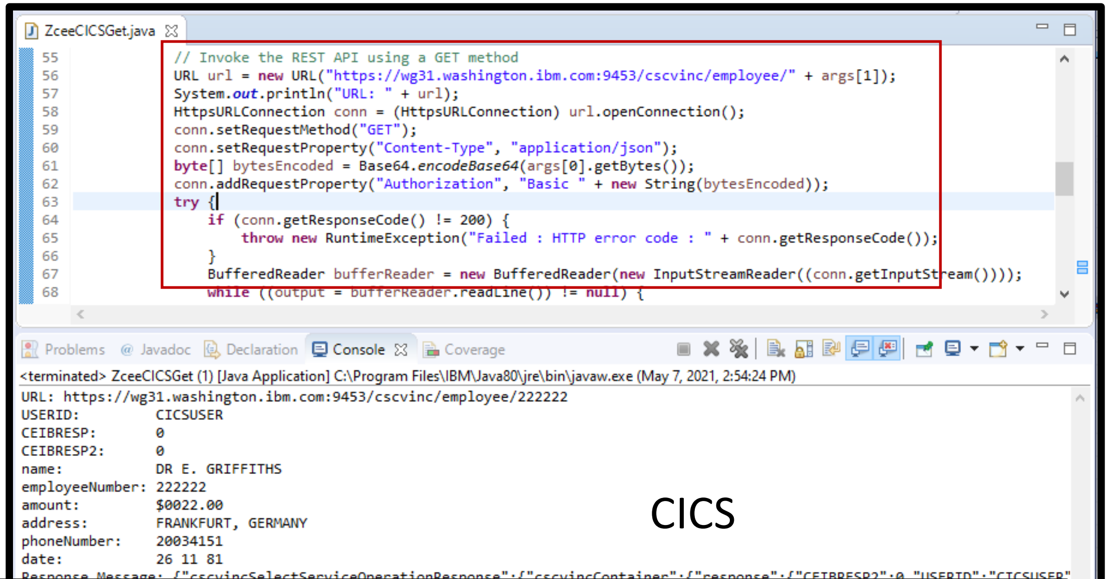

{…}
</>
z/OS LPAR
z/OS Connect EE Address Space
Mapping
Z Subsystems Address Space
Z Subsystems Address Space
program

```
GET http://www.acme.com/customers/12345
```

```
RESPONSE: HTTP 200 OK
BODY {     “ID” :       “12345”,
           “name” :     “Joe Bloggs”,
           “address” :  “10 OldStreet”,
           “tel” :      “01234123456” }
```

```
01 INQCUST.
   02 ID          PIC 9 (5).
   02 NAME        PIC x (64).
   02 ADDRESS     PIC x (128).
   02 TEL         PIC 9 (11).
```

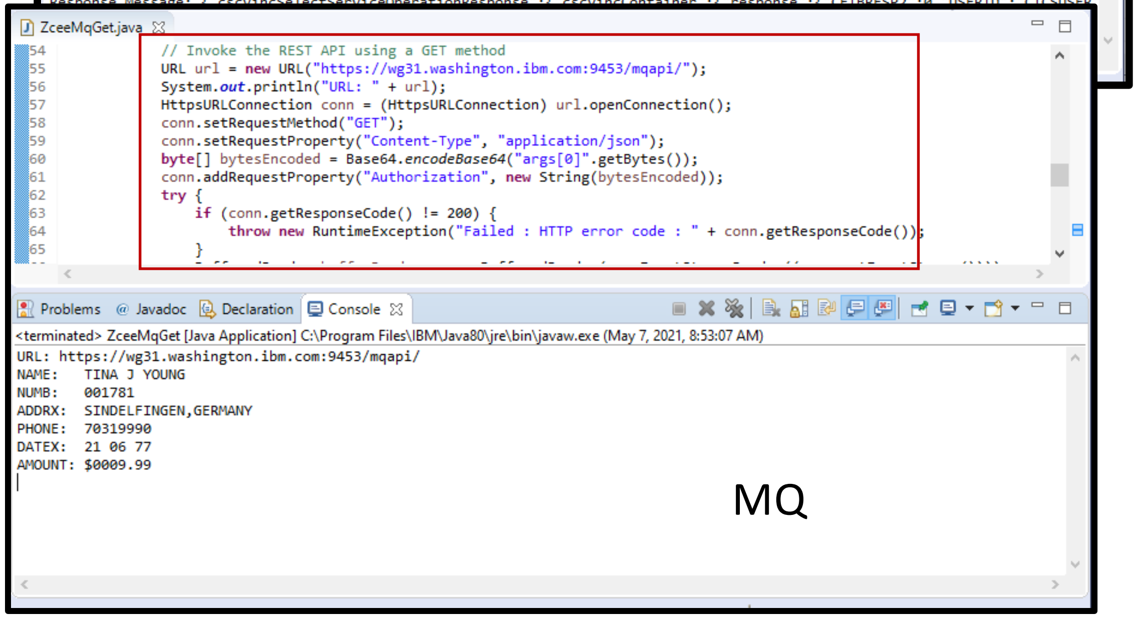
Results or goal: Client code is unaware of the z/OS infrastructure



```
        // Invoke the REST API using a GET method
        URL url = new URL("https://wg31.washington.ibm.com:9453/cscvinc/employee/" + args[1]);
        System.out.println("URL: " + url);
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Content-Type", "application/json");
        byte[] bytesEncoded = Base64.encodeBase64(args[0].getBytes());
        conn.addRequestProperty("Authorization", "Basic " + new String(bytesEncoded));
        try {
            if (conn.getResponseCode() != 200) {
                throw new RuntimeException("Failed : HTTP error code : " + conn.getResponseCode());
            }
            BufferedReader bufferReader = new BufferedReader(new InputStreamReader((conn.getInputStream())));
            while ((output = bufferReader.readLine()) != null) {
```

```
<terminated> ZceeCICSGet (1) [Java Application] C:\Program Files\IBM\Java80\jre\bin\javaw.exe (May 7, 2021, 2:54:24 PM)
URL: https://wg31.washington.ibm.com:9453/cscvinc/employee/222222
USERID:         CICSUSER
CEIBRESP:       0
CEIBRESP2:      0
name:           DR E. GRIFFITHS
employeeNumber: 222222
amount:         $0022.00
address:        FRANKFURT, GERMANY
phoneNumber:    20034151
date:           26 11 81
Response Message: {"cscvincSelectServiceOperationResponse":{"cscvincContainer":{"response":{"CEIBRESP2":0,"USERID":"CICSUSER"
```

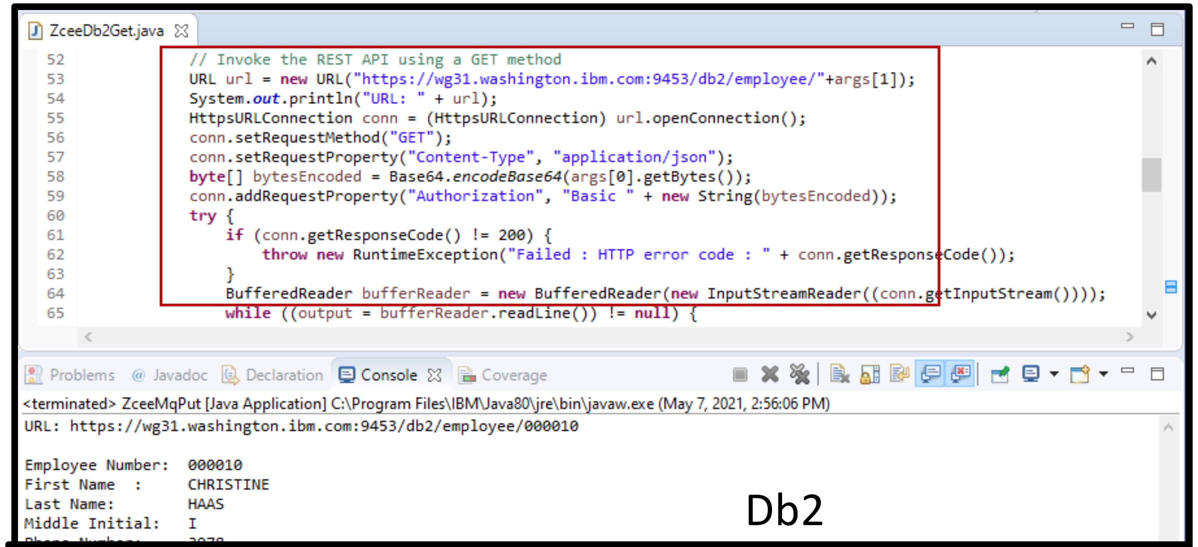
CICS



```
        // Invoke the REST API using a GET method
        URL url = new URL("https://wg31.washington.ibm.com:9453/mqapi/");
        System.out.println("URL: " + url);
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Content-Type", "application/json");
        byte[] bytesEncoded = Base64.encodeBase64("args[0]".getBytes());
        conn.addRequestProperty("Authorization", new String(bytesEncoded));
        try {
            if (conn.getResponseCode() != 200) {
                throw new RuntimeException("Failed : HTTP error code : " + conn.getResponseCode());
            }
```

```
<terminated> ZceeMqGet [Java Application] C:\Program Files\IBM\Java80\jre\bin\javaw.exe (May 7, 2021, 8:53:07 AM)
URL: https://wg31.washington.ibm.com:9453/mqapi/
NAME:   TINA J YOUNG
NUMB:   001781
ADDRX:  SINDELFINGEN,GERMANY
PHONE:  70319990
DATEX:  21 06 77
AMOUNT: $0009.99
```

MQ

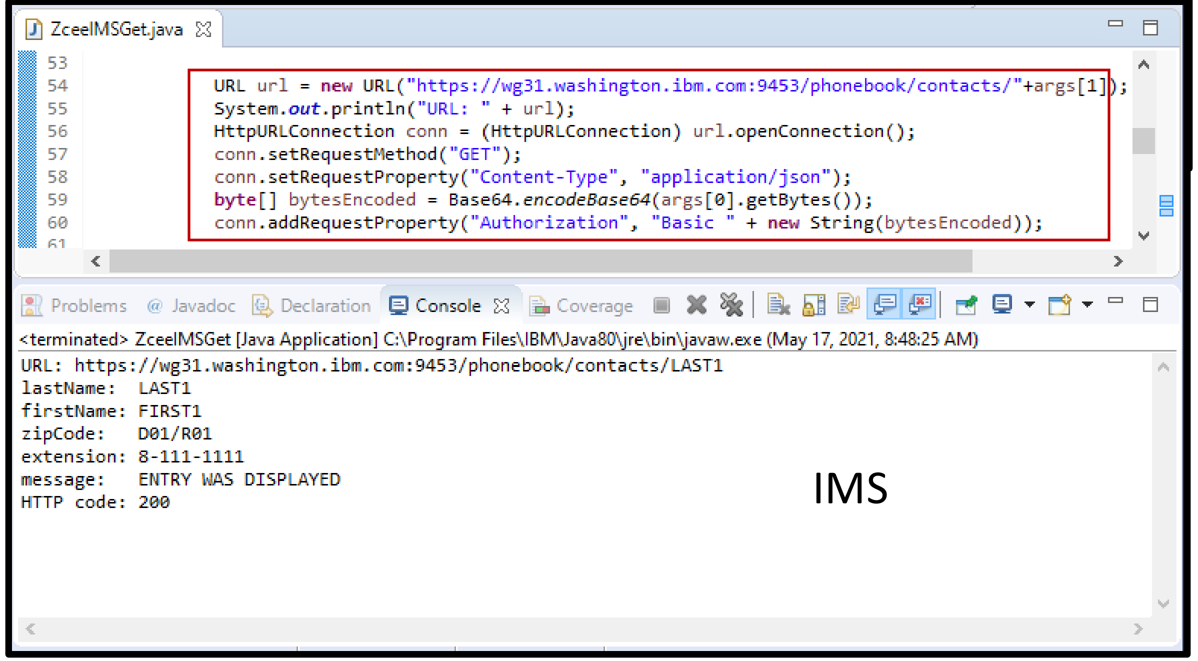


```
        // Invoke the REST API using a GET method
        URL url = new URL("https://wg31.washington.ibm.com:9453/db2/employee/"+args[1]);
        System.out.println("URL: " + url);
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Content-Type", "application/json");
        byte[] bytesEncoded = Base64.encodeBase64(args[0].getBytes());
        conn.addRequestProperty("Authorization", "Basic " + new String(bytesEncoded));
        try {
            if (conn.getResponseCode() != 200) {
                throw new RuntimeException("Failed : HTTP error code : " + conn.getResponseCode());
            }
            BufferedReader bufferReader = new BufferedReader(new InputStreamReader((conn.getInputStream())));
            while ((output = bufferReader.readLine()) != null) {
```

```
<terminated> ZceeMqPut [Java Application] C:\Program Files\IBM\Java80\jre\bin\javaw.exe (May 7, 2021, 2:56:06 PM)
URL: https://wg31.washington.ibm.com:9453/db2/employee/000010

Employee Number:   000010
First Name   :     CHRISTINE
Last Name:         HAAS
Middle Initial:    I
Phone Number:      3978
```

Db2



```
        URL url = new URL("https://wg31.washington.ibm.com:9453/phonebook/contacts/"+args[1]);
        System.out.println("URL: " + url);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Content-Type", "application/json");
        byte[] bytesEncoded = Base64.encodeBase64(args[0].getBytes());
        conn.addRequestProperty("Authorization", "Basic " + new String(bytesEncoded));
```

```
<terminated> ZceeIMSGet [Java Application] C:\Program Files\IBM\Java80\jre\bin\javaw.exe (May 17, 2021, 8:48:25 AM)
URL: https://wg31.washington.ibm.com:9453/phonebook/contacts/LAST1
lastName:  LAST1
firstName: FIRST1
zipCode:   D01/R01
extension: 8-111-1111
message:   ENTRY WAS DISPLAYED
HTTP code: 200
```

IMS

Wildfire Workshop

John.J.Brefach@ibm.com

Let’s stop and ask what is the significance of the OpenAPI Specification to z/OS Connect?

The industry standard framework for describing REST APIs

- **z/OS Connect and Swagger 2.0 (Open API Specification 2), supported initially by z/OS Connect**

  Initially, accessing z/OS resources was the only desire for developing APIs .The interactions with the z/OS resources was driven by the layout of the CICS COMMAREA or CONTAINER, the IMS or MQ messages or the Db2 REST service.

  - The details of the interactions with the z/OS resource determined the contents of the API request and response messages and the subsequent specification document.
  - **z/OS Connect produces the specification document that describes the methods and request and response messages.**

- **z/OS Connect and Open API Specification 3, supported by z/OS Connect starting in March 2022 service, V3.0.55**

  As companies mature their API strategy, they begin to introduce API governance boards to drive consistency in their API design. As more public APIs are created, government and industry standards bodies begin to regulate and drive for standardization. This drives the need for “API first” functional mapping capabilities within the integration platform. The external API design determined the layouts of the API request and response messages provided by the specification documents which was consumes by z/OS Connect to describe the z/OS resource interactions.

  - The API details of the methods and layouts of request and response messages are provided in advance and access to the z/OS resource is driven by the API design
  - **z/OS Connect consumes the specification document that describes the methods and request and response messages**

API Development Lifecycle Overview

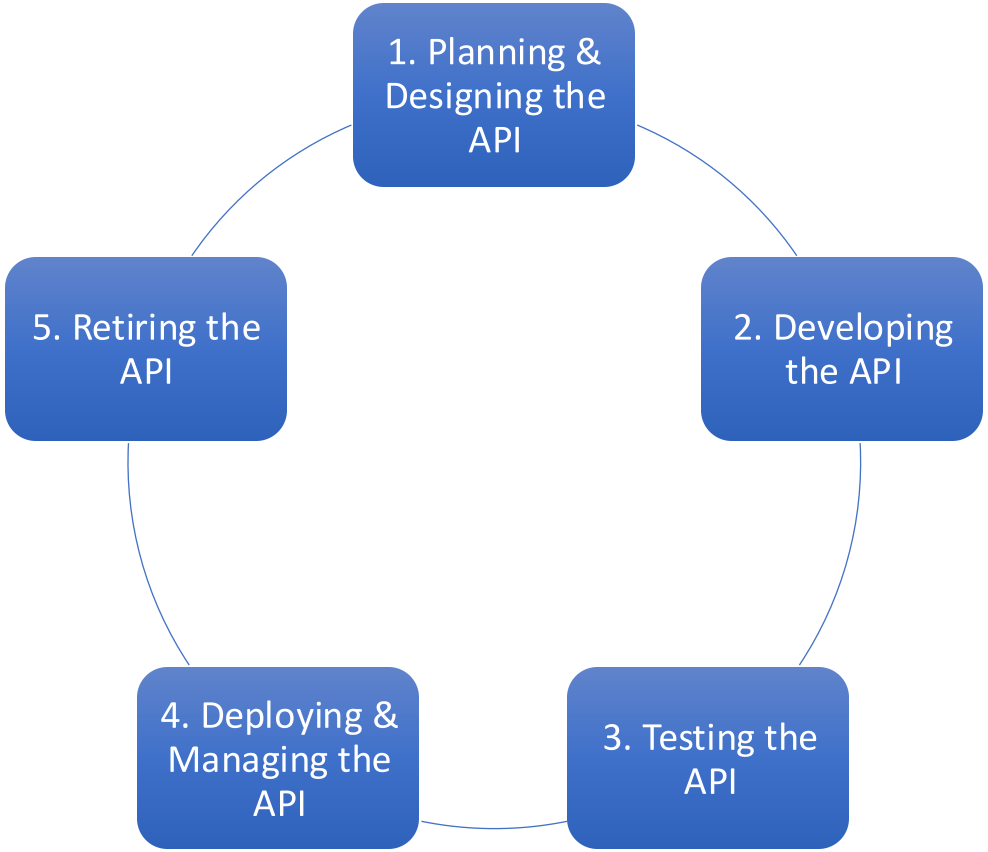
Plus, an In-depth analysis of each chronological stage

API Development Lifecycle

The API lifecycle is a series of steps that teams take in order to successfully design, develop, deploy, and consume APIs

-Each of these stages is a vital piece to making sure an API is successful.

While there are many frameworks/methodologies for the API lifecycle, this presentation will discuss five chronological stages



1. Planning & Designing the API

2. Developing the API

3. Testing the API

4. Deploying & Managing the API

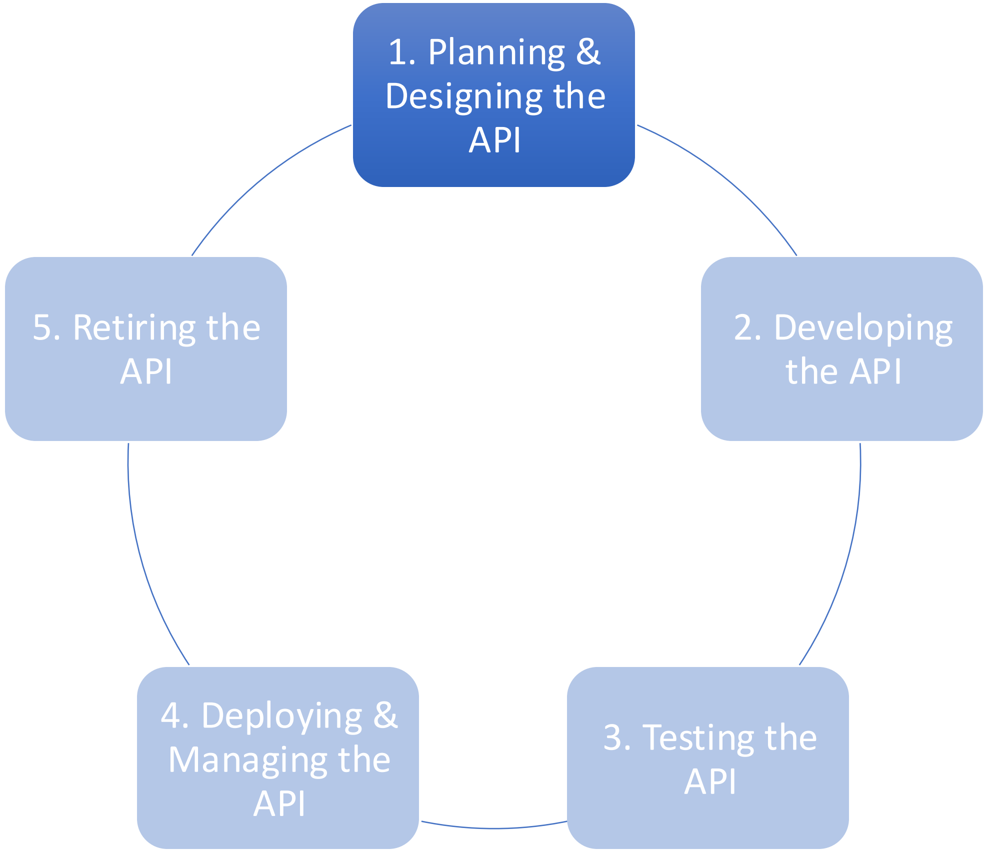
5. Retiring the API

Stage 1: Planning & Designing the API

This stage involves formulating and mapping out the various resources and operations with their associated use cases before the API is fully implemented in code

Team members will begin by mapping out the capabilities of the API and the data it should expose

- This process captures the API user's needs and the requirements from the stakeholders


1. Planning & Designing the API
2. Developing the API
3. Testing the API
4. Deploying & Managing the API
5. Retiring the API

Benefits of Investing Time and Resources into the Design Phase

ORGANIZATIONAL ALIGNMENT

INCREMENTAL DEVELOPMENT

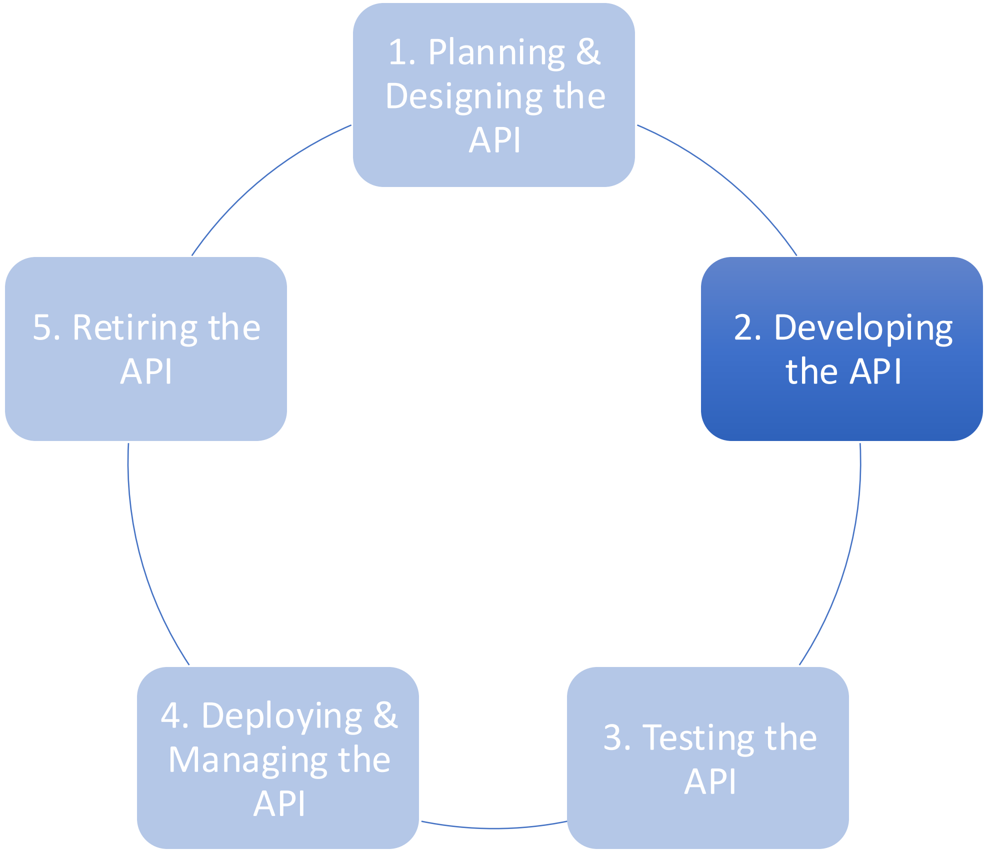
BETTER DOCUMENTATION

Stage 2: Developing the API

This stage focuses on implementing the API based on the plan and design

-This is when engineering teams will do the necessary work to bring the API to life.

Typically, this is also where the API is documented. Often when APIs are treated as a product, teams will spend just as much time documenting their API as they will on any of the other phases.


1. Planning & Designing the API
2. Developing the API
3. Testing the API
4. Deploying & Managing the API
5. Retiring the API

API Documentation is for:

Decision-makers

Evaluator

Problem Solver

Users

Debugger

Newcomer

Include Resources in your Documentation

- Getting Started Guide
- SDKs and Libraries
- Interactive Console

Best Practices in API Documentation

- Detailed Error Messages
- A list of all Exposed Resources
- A Terms of Use Agreement
- A Changelog
- Limited Technical Vocab
- Examples of all Requests & Responses
- An Authentication Guide

Benefits of a Well-Crafted API documentation

- Improved Developer Experience (DX)
- Increased Awareness
- Saves Support Time and Costs
- Easier Maintenance

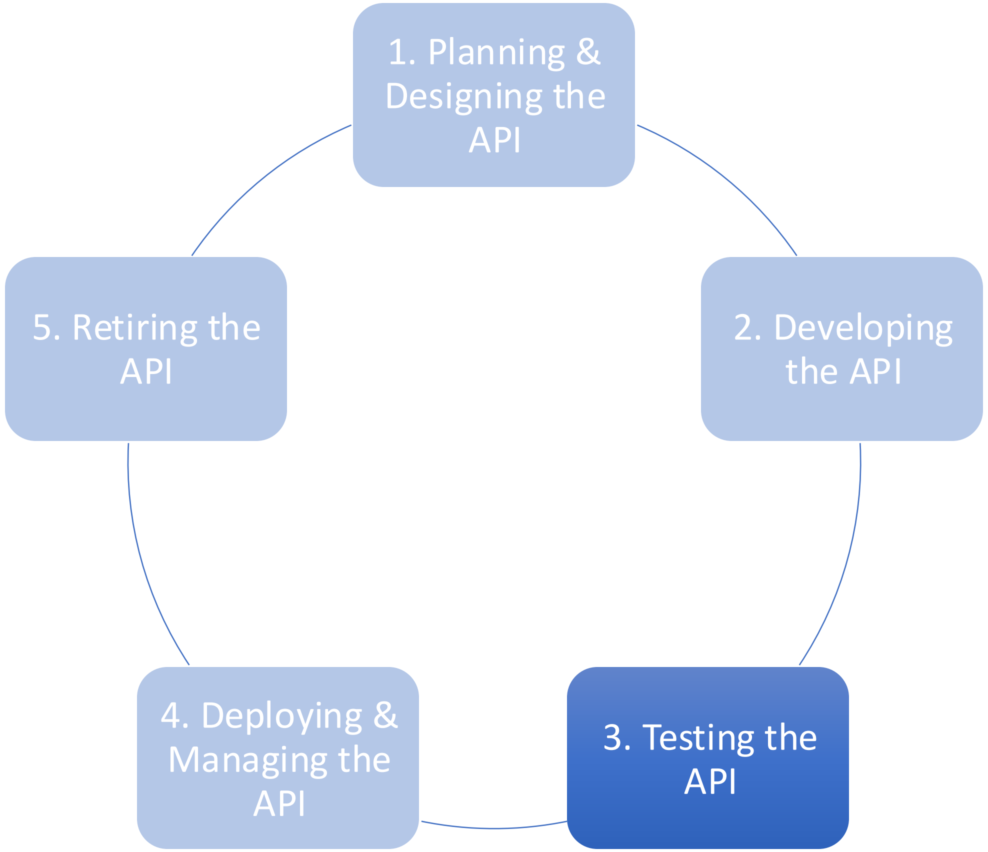
Stage 3: Testing the API

In the testing phase, the API would be thoroughly tested and monitored for performance issues

This phase typically catches any issues so that engineering teams can refine the API before it is released to the end-users

On some occasions, mainly when APIs are public-facing, APIs are released into beta where end-users can test and experiment with the API and give feedback and report bugs

-The beta helps finalize any outstanding issues as well as gain valuable feedback before a full API release.


1. Planning & Designing the API
2. Developing the API
3. Testing the API
4. Deploying & Managing the API
5. Retiring the API

John.J.Brefach@ibm.com

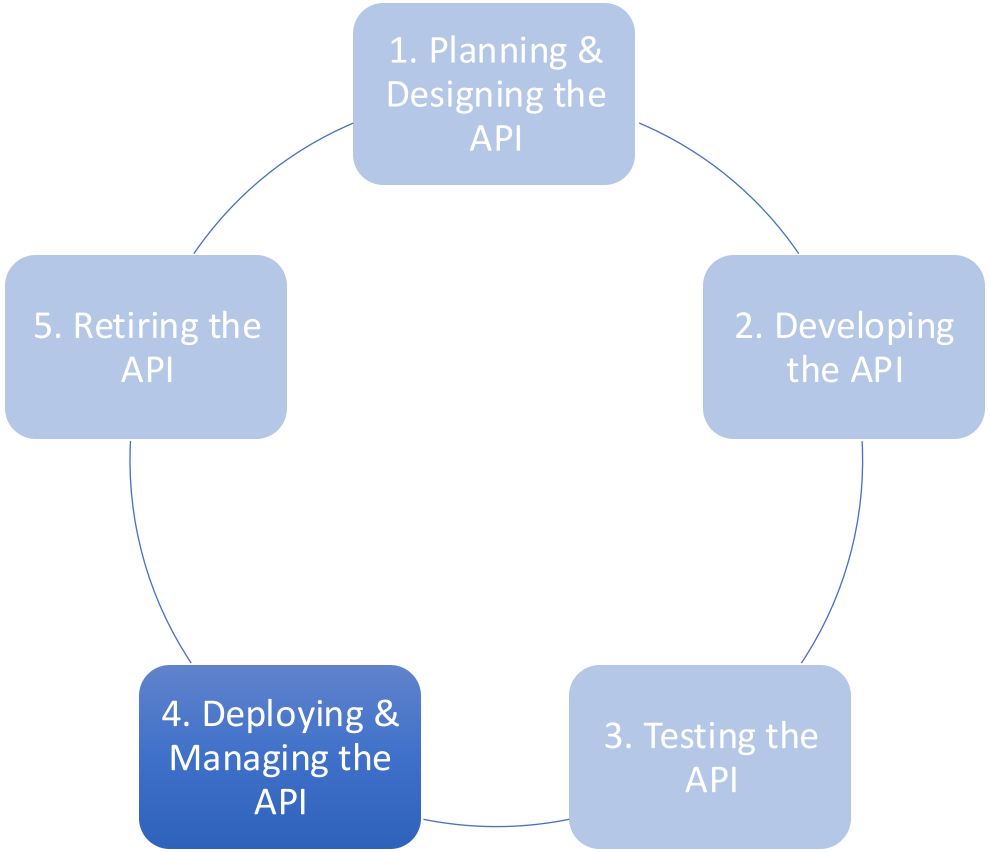
Characteristics of a Well-Designed API

- Easy to read & work with
- Hard to misuse
- Well documented
- Reliable

Stage 4: Deploying & Managing the API

After the testing phase, APIs are ready for release and are deployed to a secure environment to facilitate easy discovery and consumption.

This phase is also where APIs are managed for the rest of their lifetime to deliver guaranteed and high-quality API performance.

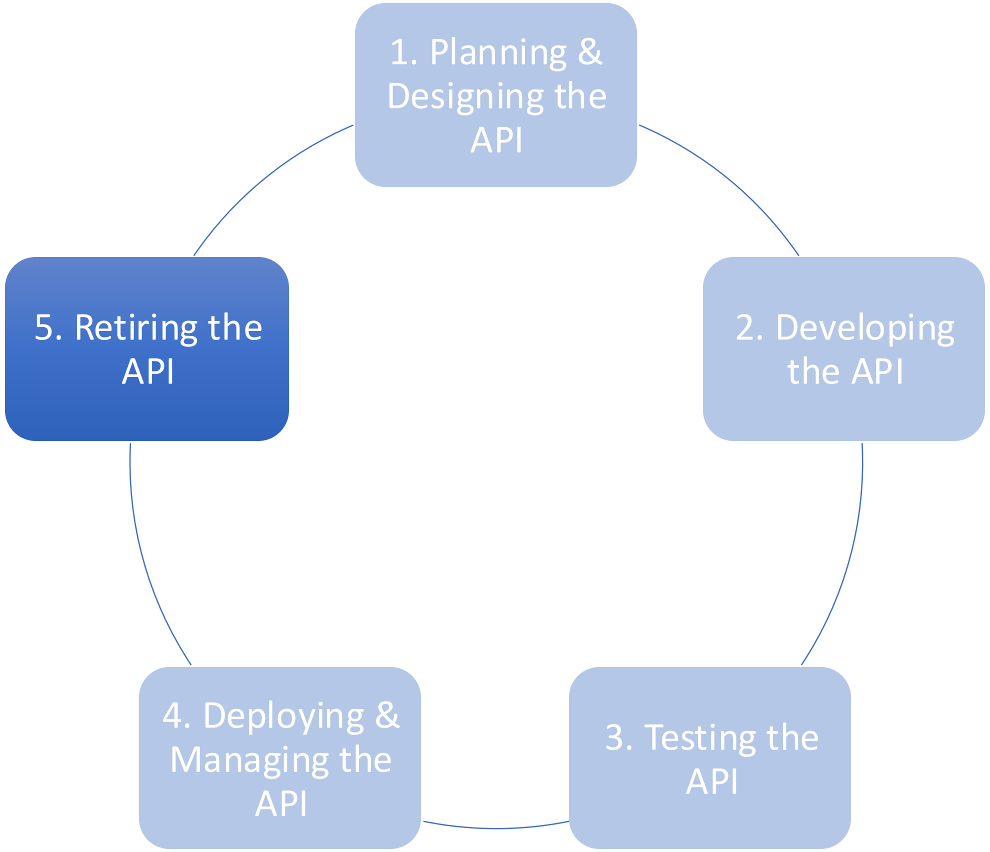

1. Planning & Designing the API
2. Developing the API
3. Testing the API
4. Deploying & Managing the API
5. Retiring the API

Stage 5: Retiring the API

This phase is where support for an API's version, or an entire API itself, is discontinued

Deprecation involves creating a detailed plan on migrating users away from the API

- Deprecation needs to be handled with extreme care as it may impact many end-users and associated products that utilize the API

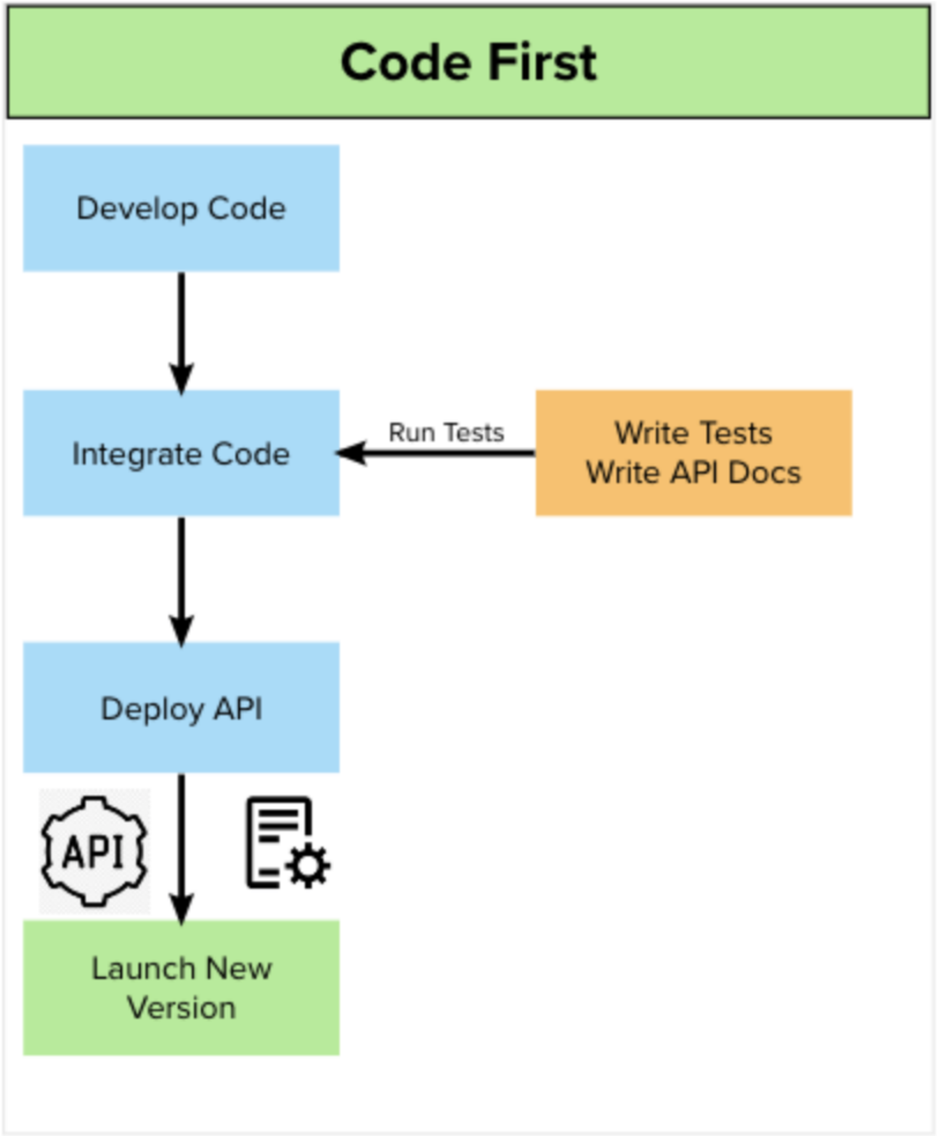

1. Planning & Designing the API
2. Developing the API
3. Testing the API
4. Deploying & Managing the API
5. Retiring the API

Design First vs Code First

What method works best for you

Code First Approach

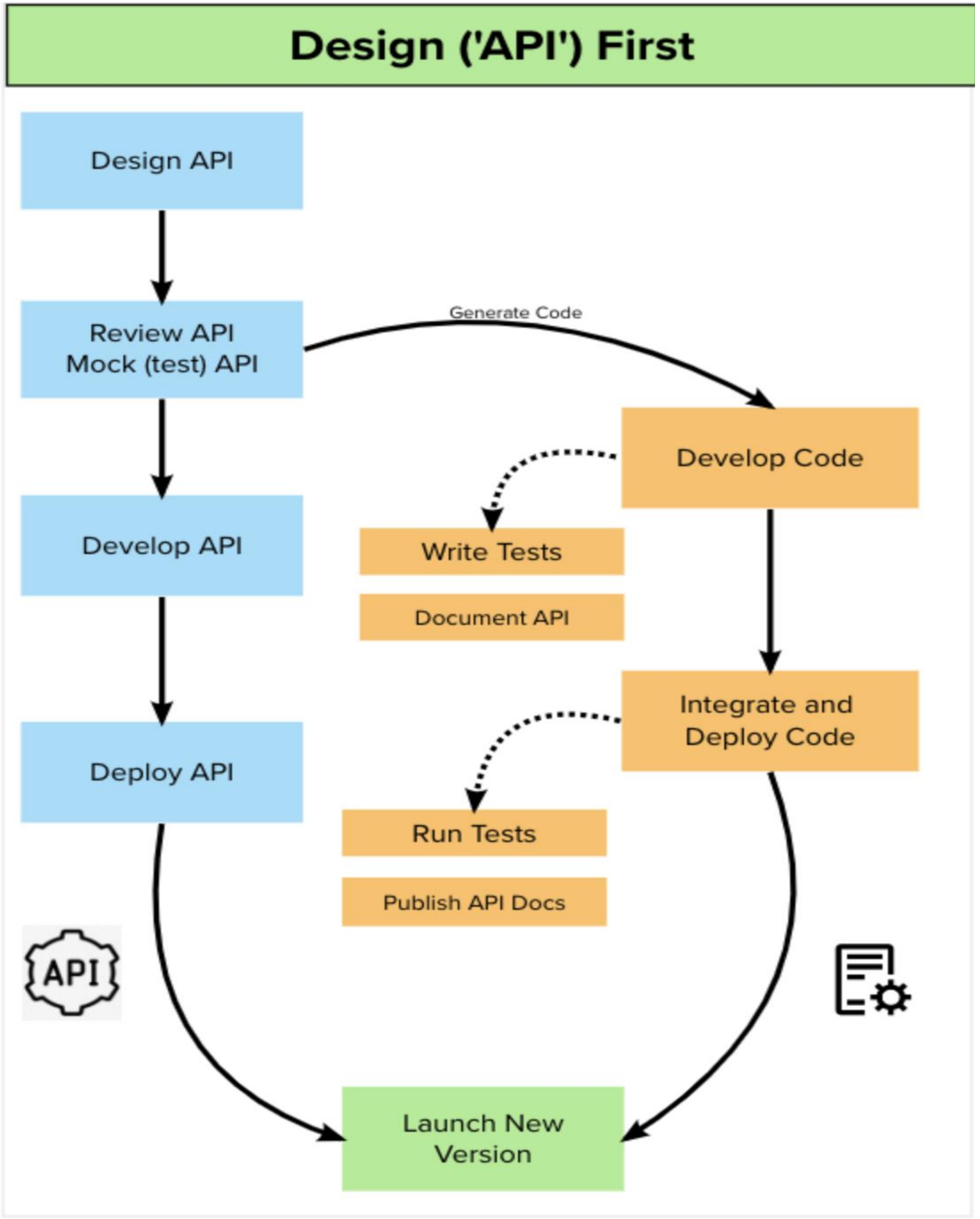
- The API is directly coded from a business plan
  - Since the API is already implemented, it typically serves the role of documentation
  - From the API, a human & machine-readable document can be generated



https://swagger.io/blog/code-first-vs-design-first-api/

Design ('API') First Approach

- The API design plan is converted to a machine & human readable contract from which the code will be built
- The API Contract serves as a blueprint for the APIs structure and features to guide the development process



https://swagger.io/blog/code-first-vs-design-first-api/

Advantages of Design First

Improved system integration

Enhanced collaboration & quality

Increased Scalability

Decide what approach works best for YOU

Design First when:

Developer Experience Matters

Delivering Mission Critical APIs

Ensuring Good Communication

Code First when:

Delivery Speed matters

Developing internal APIs

OpenAPI Specification

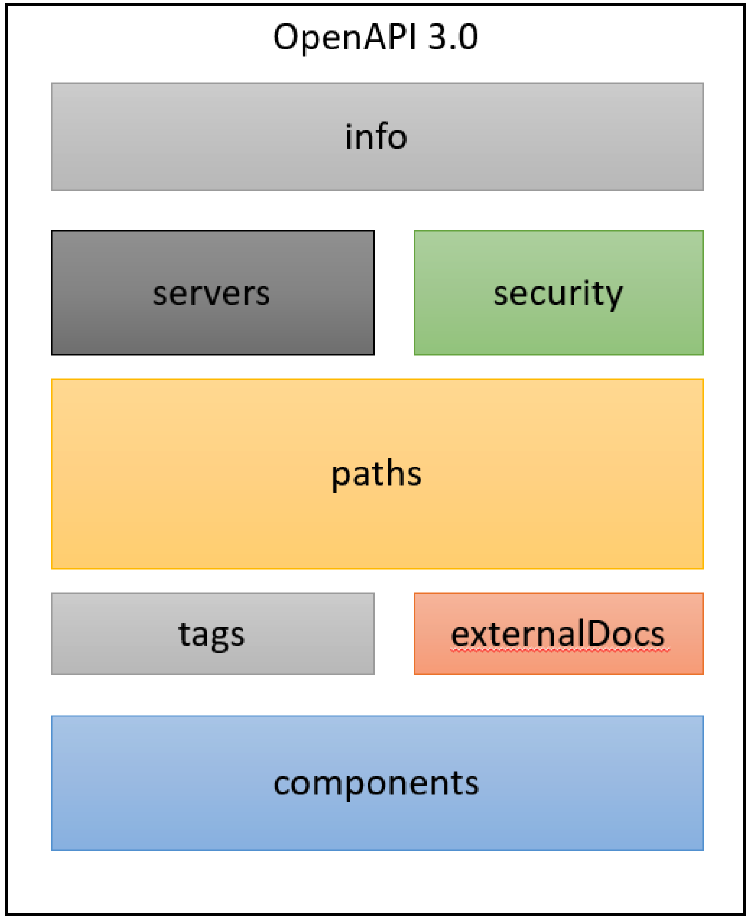
Overview of the Structure and Syntax of OpenAPI Specification

John.J.Brefach@ibm.com

OpenAPI Specification Overview

- OpenAPI Specification(OAS) is one of the most popular standards for designing human-readable API contracts
- The OAS specifies the rules and syntax required to describe an API's interface

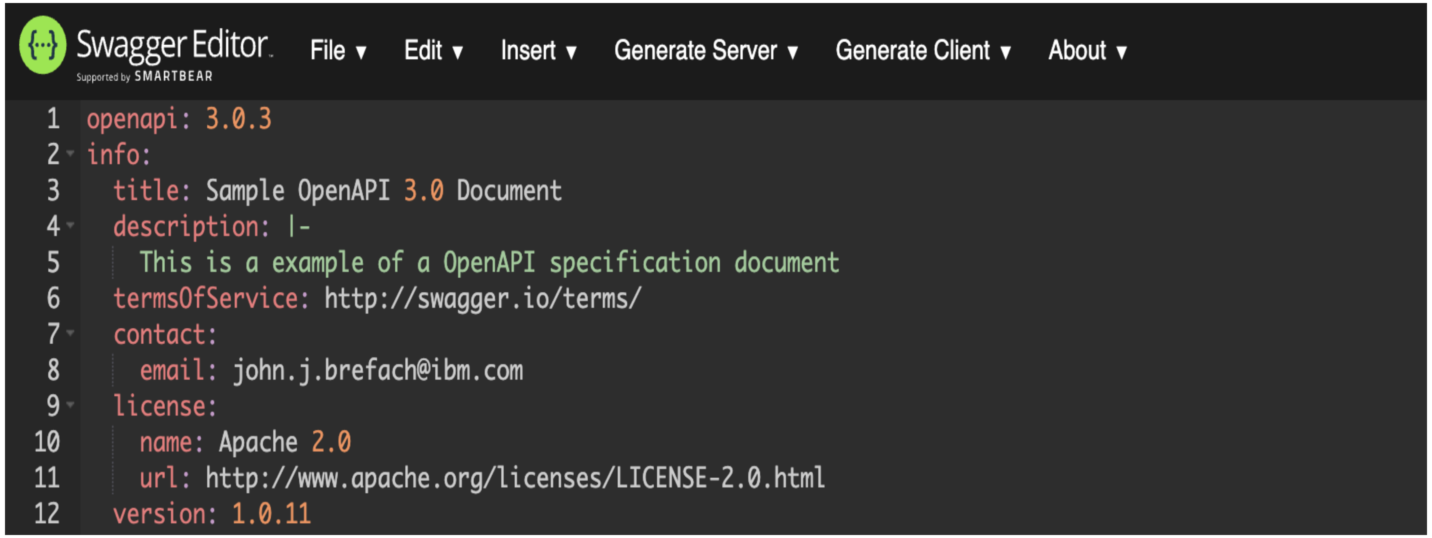
General Structure



https://swagger.io/specification/v2/
https://swagger.io/specification/v3/

Info & OpenAPI

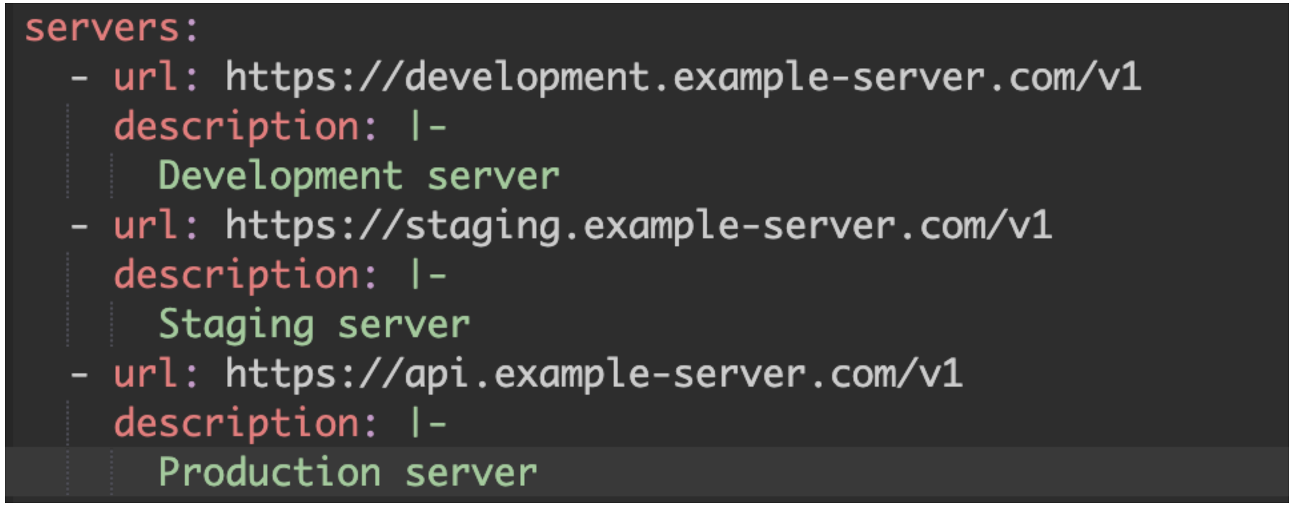
- The info and OpenAPI section of the API contract contains essential metadata
- Essentially, the info object should give an API's end-users and internal developers a high-level overview of what the API does



```yaml
openapi: 3.0.3
info:
  title: Sample OpenAPI 3.0 Document
  description: |-
    This is a example of a OpenAPI specification document
  termsOfService: http://swagger.io/terms/
  contact:
    email: john.j.brefach@ibm.com
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.11
```

Servers

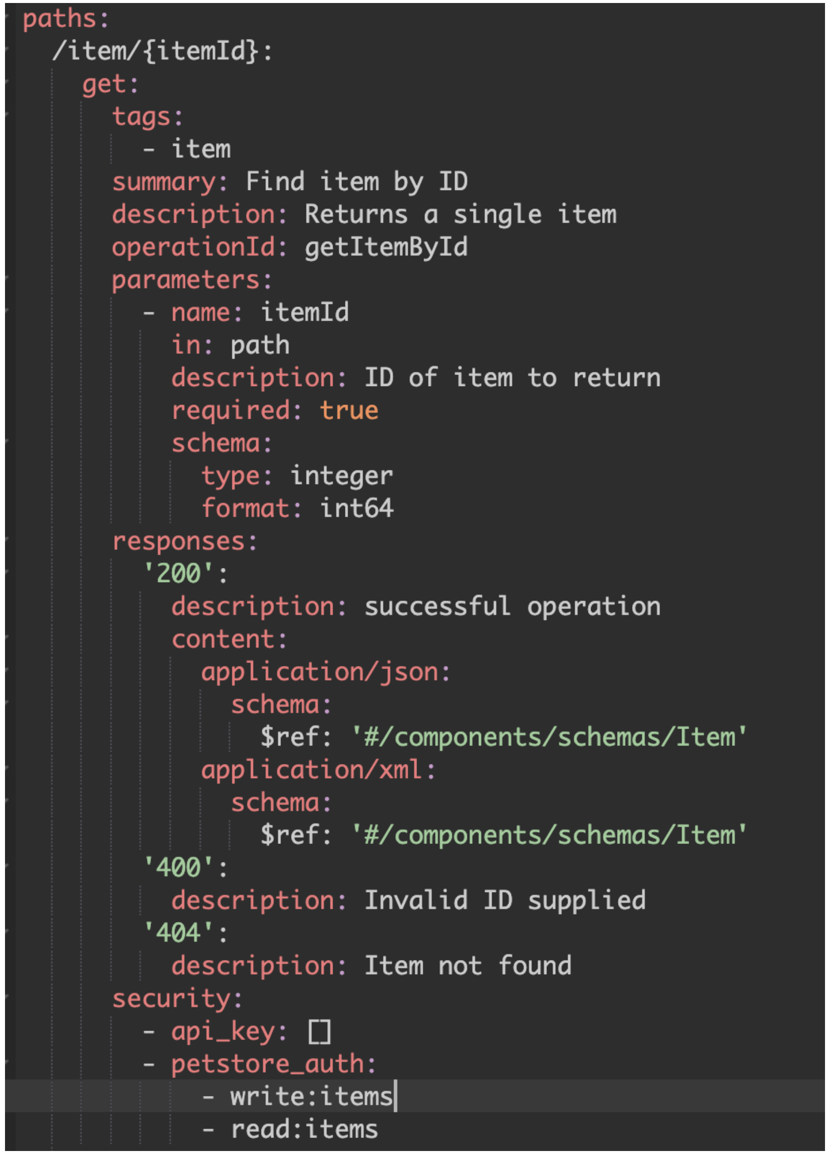
- The servers object can give a client information on where the API’s servers are located through its URL
- OAS 3.0 supports multiple servers
  - APIs usually exist in numerous environments and each environment can have its own purpose



```yaml
servers:
  - url: https://development.example-server.com/v1
    description: |-
      Development server
  - url: https://staging.example-server.com/v1
    description: |-
      Staging server
  - url: https://api.example-server.com/v1
    description: |-
      Production server
```

Paths

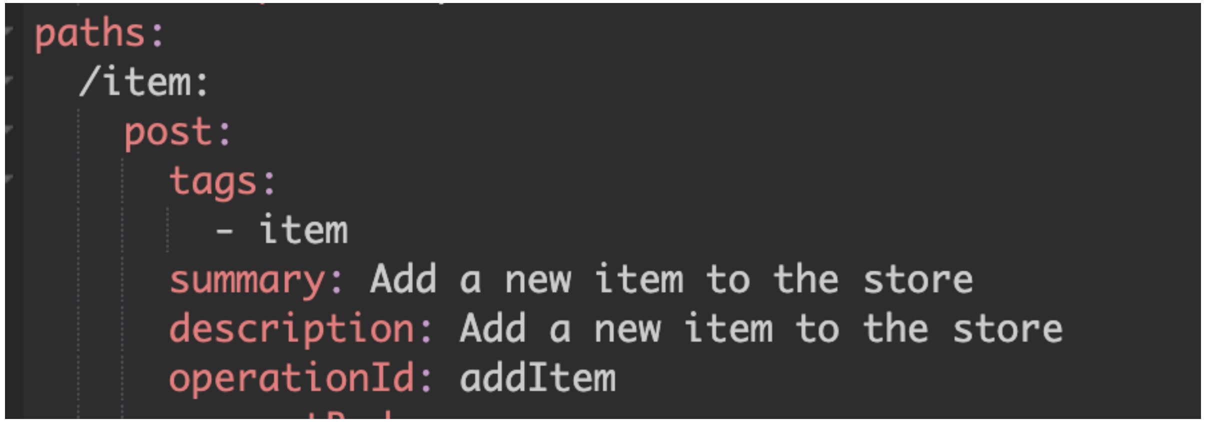
- The Paths object shows the various endpoints an API exposes and the corresponding HTTP methods.
- Parameters are the variable parts of a request. There are four types of parameters that can be specified using the OAS 3.0:
  - Path parameters, such as /users/{id}
  - Query parameters, such as /users?role=admin
  - Header parameters, such as X-MyHeader: Value
  - Cookie parameters, which are passed in the cookie header, such as Cookie: debug=0; csrftoken=BUSe35dohU301MZvDCU
- Responses are the objects returned on a request.
  - Every response is defined by its HTTP status code, and the data is returned.
  - The HTTP status codes are used to define whether the request was successful or unsuccessful



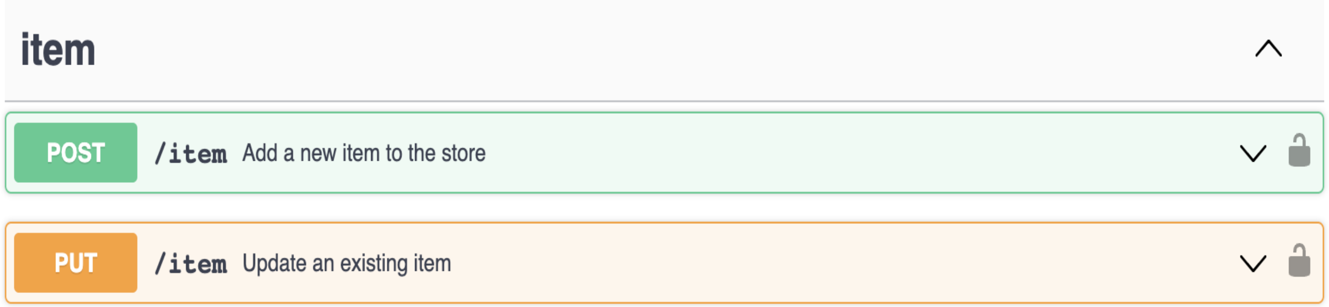
```yaml
paths:
  /item/{itemId}:
    get:
      tags:
        - item
      summary: Find item by ID
      description: Returns a single item
      operationId: getItemById
      parameters:
        - name: itemId
          in: path
          description: ID of item to return
          required: true
          schema:
            type: integer
            format: int64
      responses:
        '200':
          description: successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Item'
            application/xml:
              schema:
                $ref: '#/components/schemas/Item'
        '400':
          description: Invalid ID supplied
        '404':
          description: Item not found
      security:
        - api_key: []
        - petstore_auth:
            - write:items
            - read:items
```

Tags

- Tags are used to group various API operations.
  - This allows end-users of the API to better segment and identify what they want to use the API for. These tags can also be handled by other third-party tools which integrate or read the OAS
- Tags can automatically be added to every path operation using the tag objects
- Tags can also be given descriptions by adding an optional tags section in the root level of the API definition



```yaml
paths:
  /item:
    post:
      tags:
        - item
      summary: Add a new item to the store
      description: Add a new item to the store
      operationId: addItem
```



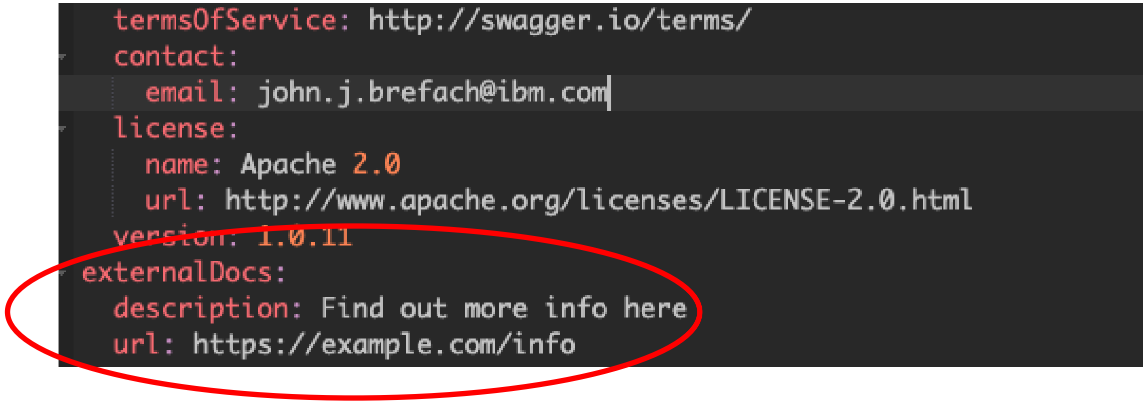
item

POST /item Add a new item to the store

PUT /item Update an existing item

External Docs

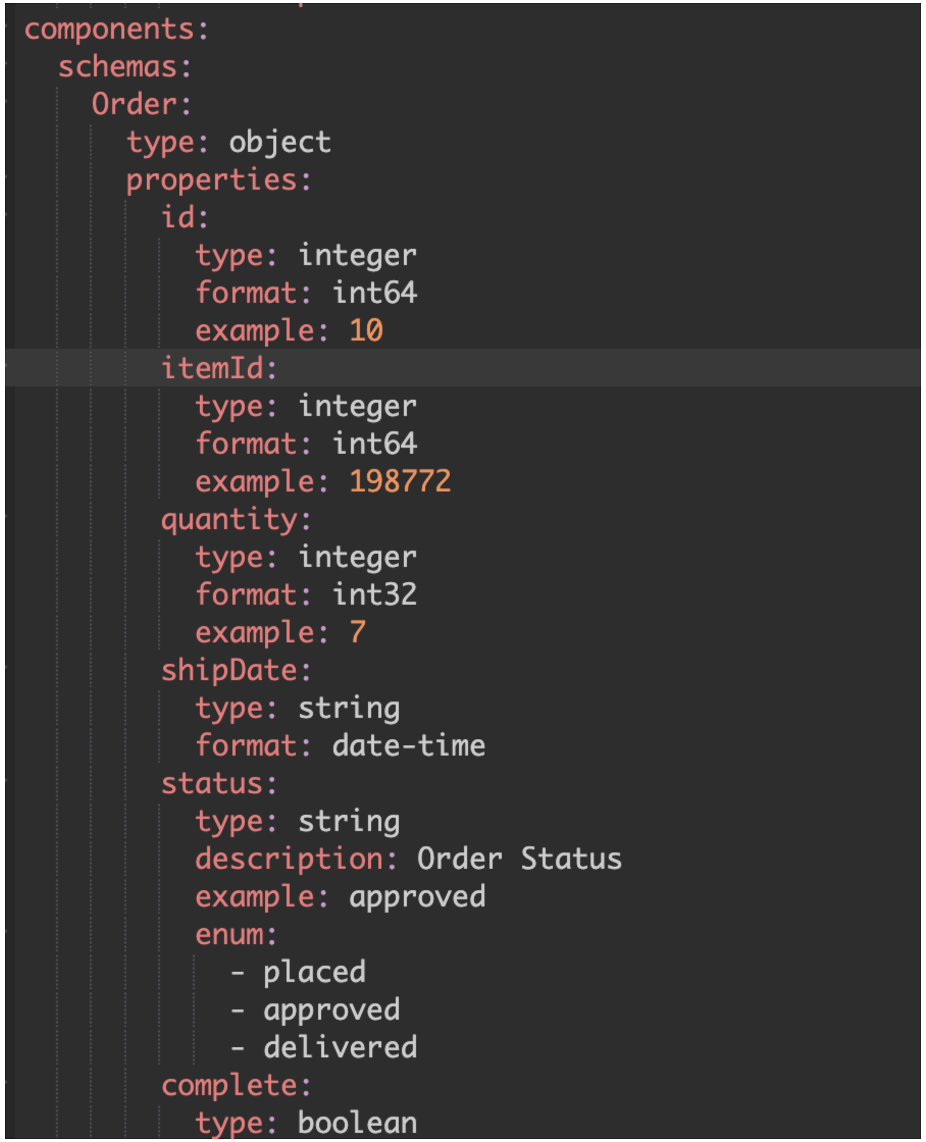
- OAS 3.0 allows an API to reference external documentation via the external documentation object
- Any additional information that an API can provide to improve integration and simplify consumption should always be considered



```
  termsOfService: http://swagger.io/terms/
  contact:
    email: john.j.brefach@ibm.com
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.11
externalDocs:
  description: Find out more info here
  url: https://example.com/info
```

Components

- The component object can hold a set of reusable objects for an APIs design.
- The reusable objects can be schemas, responses, parameters, examples, and more.
  - The exact reusable component can then be referenced in any path item



```yaml
components:
  schemas:
    Order:
      type: object
      properties:
        id:
          type: integer
          format: int64
          example: 10
        itemId:
          type: integer
          format: int64
          example: 198772
        quantity:
          type: integer
          format: int32
          example: 7
        shipDate:
          type: string
          format: date-time
        status:
          type: string
          description: Order Status
          example: approved
          enum:
            - placed
            - approved
            - delivered
        complete:
          type: boolean
```

Swagger Tooling

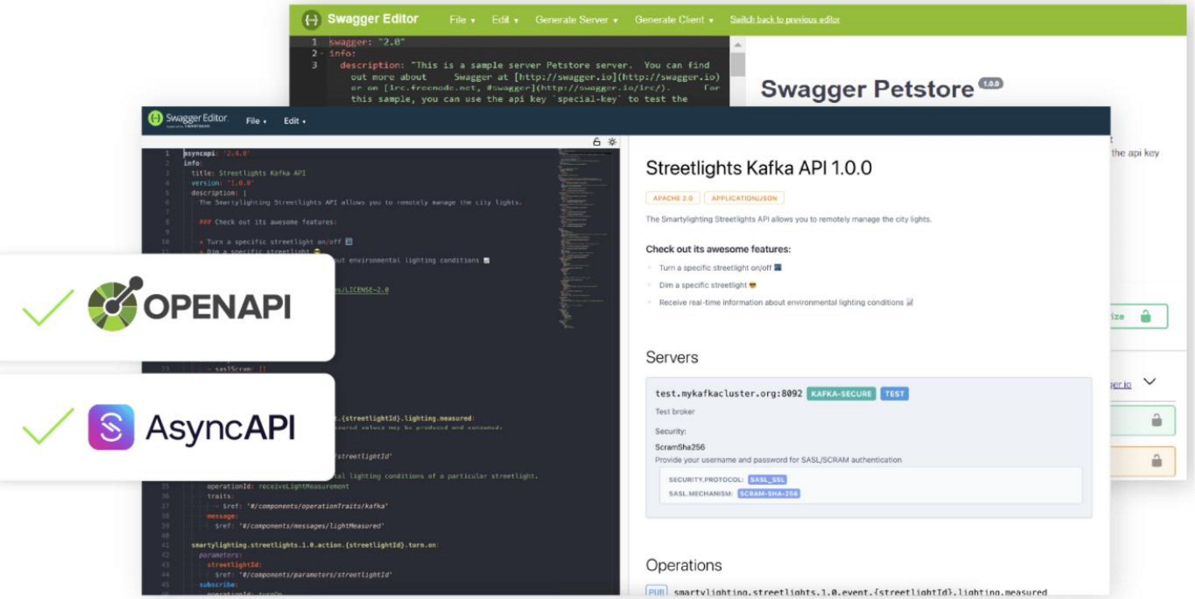
API Tools at your Disposal

Overview

- Swagger refers to the toolkit used for designing, building, and documenting APIs
  - Swagger Editor
  - Swagger CodeGen
  - Swagger UI
- The Swagger Editor is used to design the API specification, Swagger CodeGen is used to generate code based on the created specification. Swagger UI allows anyone to easily view the specification details in an easy-to-read way.

Swagger Editor

- This tool is primarily used to design, define, and document RESTful APIs.
  - This editor accepts different OpenAPI versions, includes the option to convert a written specification to YAML (or JSON), and highlights any errors that might be occurring in the specification



```
swagger: "2.0"
info:
  description: "This is a sample server Petstore server.  You can find
    out more about     Swagger at [http://swagger.io](http://swagger.io)
    or on [irc.freenode.net, #swagger](http://swagger.io/irc/).      For
    this sample, you can use the api key `special-key` to test the
```

Streetlights Kafka API 1.0.0

APACHE 2.0 APPLICATION/JSON

The Smartylighting Streetlights API allows you to remotely manage the city lights.

Check out its awesome features:

- Turn a specific streetlight on/off 🌃
- Dim a specific streetlight 😎
- Receive real-time information about environmental lighting conditions 📈

Servers

test.mykafkacluster.org:8092 KAFKA-SECURE TEST

Test broker

Security:

ScramSha256

Provide your username and password for SASL/SCRAM authentication

SECURITY.PROTOCOL: SASL_SSL

SASL.MECHANISM: SCRAM-SHA-256

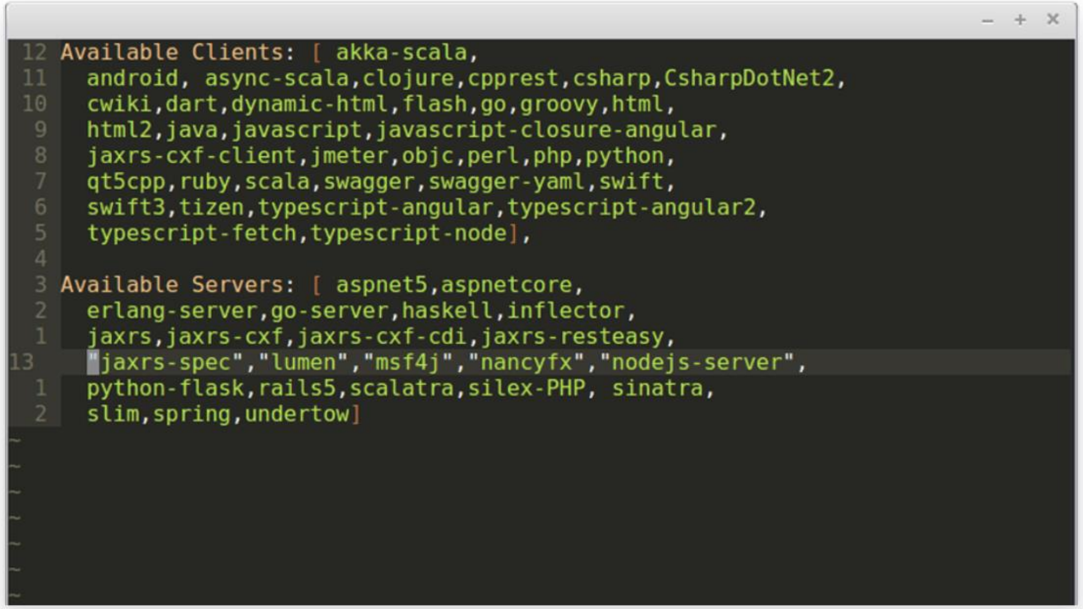
Operations

PUB smartylighting.streetlights.1.0.event.{streetlightId}.lighting.measured

https://swagger.io/tools/swagger-editor/

Swagger CodeGen

- Using Swagger CodeGen and a provided API specification, we can generate server and client-side code in many different languages.
  - The generated code will even include documentation from the provided specification.
  - Besides saving time by easily generating code, the Swagger Codegen tool provides more consistent code than writing it manually from scratch



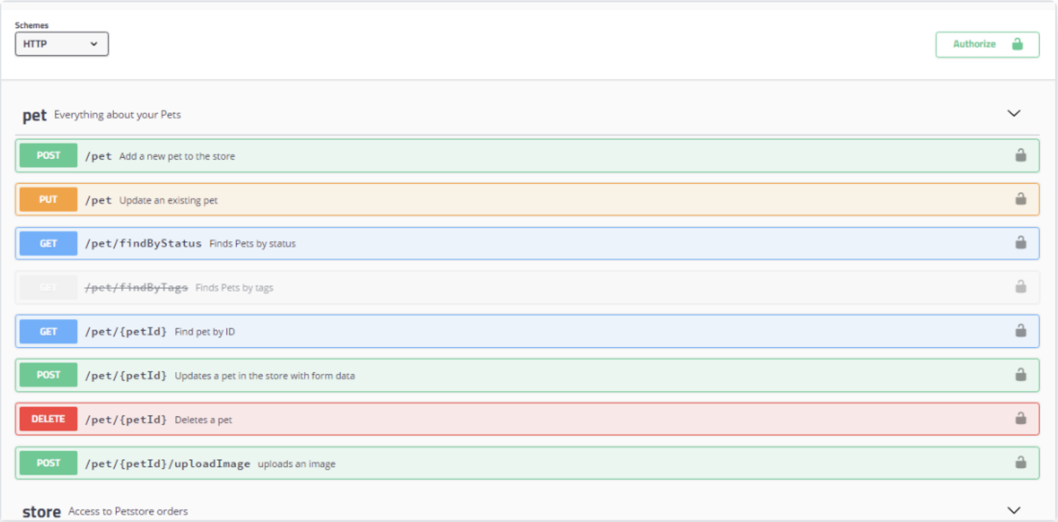
```
Available Clients: [ akka-scala,
  android, async-scala,clojure,cpprest,csharp,CsharpDotNet2,
  cwiki,dart,dynamic-html,flash,go,groovy,html,
  html2,java,javascript,javascript-closure-angular,
  jaxrs-cxf-client,jmeter,objc,perl,php,python,
  qt5cpp,ruby,scala,swagger,swagger-yaml,swift,
  swift3,tizen,typescript-angular,typescript-angular2,
  typescript-fetch,typescript-node],

Available Servers: [ aspnet5,aspnetcore,
  erlang-server,go-server,haskell,inflector,
  jaxrs,jaxrs-cxf,jaxrs-cxf-cdi,jaxrs-resteasy,
  "jaxrs-spec","lumen","msf4j","nancyfx","nodejs-server",
  python-flask,rails5,scalatra,silex-PHP, sinatra,
  slim,spring,undertow]
```

https://swagger.io/tools/swagger-codegen/

Swagger UI

- The Swagger UI tool allows anyone (development team or end-users) to visualize and interact with an API’s resources without having any of the implementation logic in place.
    - This means we don’t even have to have any code written for an end-user to see the APIs resources, endpoints, and even execute mock API calls



pet Everything about your Pets

POST /pet Add a new pet to the store

PUT /pet Update an existing pet

GET /pet/findByStatus Finds Pets by status

GET ~~/pet/findByTags~~ Finds Pets by tags

GET /pet/{petId} Find pet by ID

POST /pet/{petId} Updates a pet in the store with form data

DELETE /pet/{petId} Deletes a pet

POST /pet/{petId}/uploadImage uploads an image

store Access to Petstore orders

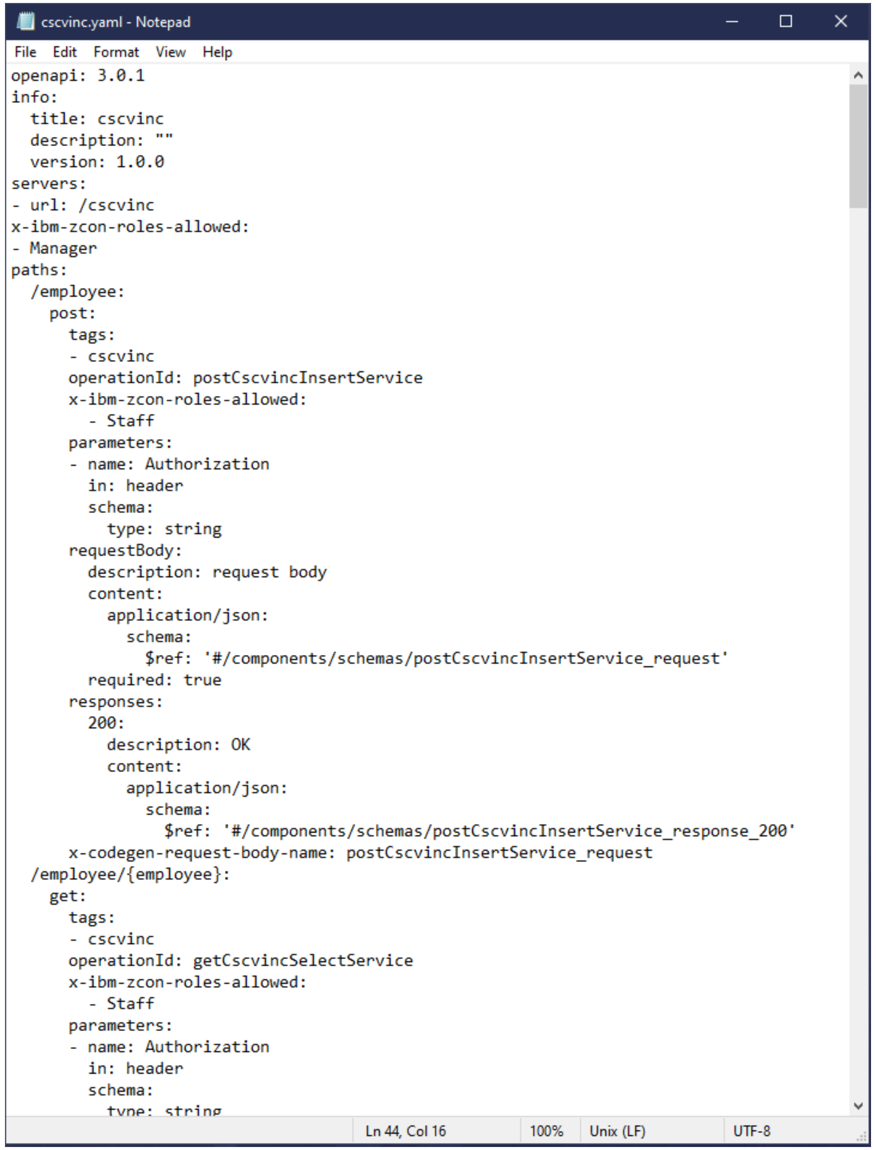
https://swagger.io/tools/swagger-ui/

Introduction to YAML

Overview of Yet Another Markup Language

YAML Overview

- Yet Another Markup Language is a standard format for storing data
- YAML and JSON are both human-readable and can represent complex data structures
  - YAML is considered more human-readable



```yaml
openapi: 3.0.1
info:
  title: cscvinc
  description: ""
  version: 1.0.0
servers:
- url: /cscvinc
x-ibm-zcon-roles-allowed:
- Manager
paths:
  /employee:
    post:
      tags:
      - cscvinc
      operationId: postCscvincInsertService
      x-ibm-zcon-roles-allowed:
        - Staff
      parameters:
      - name: Authorization
        in: header
        schema:
          type: string
      requestBody:
        description: request body
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/postCscvincInsertService_request'
        required: true
      responses:
        200:
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/postCscvincInsertService_response_200'
      x-codegen-request-body-name: postCscvincInsertService_request
  /employee/{employee}:
    get:
      tags:
      - cscvinc
      operationId: getCscvincSelectService
      x-ibm-zcon-roles-allowed:
        - Staff
      parameters:
      - name: Authorization
        in: header
        schema:
          type: string
```

YAML Structure

- A YAML document begins with three dashes (---) and ends with three dots (…). These characters can separate multiple YAML documents within a single file.
- The second line begins with #, which makes it a comment. Comments are ignored by parsers but are helpful since YAML files are often shared by different developers and can provide insight into the document's purpose.
- The bulk of this YAML document consists of mappings or key-value pairs, which are separated by a colon and a space (:). Every key must be a string and must be unique. Values can be nested mappings, as is the case with the values of capitals. They can also be sequences, as with the value of oceans, or scalars, as with the value of bottle.
- The use of whitespace is a crucial aspect of YAML. Notice how a line break separates each mapping. When objects are nester, indentation indicates which objects are a part of the same value. Indentation must consist of one or more spaces. Tabs are forbidden in YAML.
- YAML files should end with the extension .yaml or .yml

YAML Sequences

- YAML Sequences look similar to a list or an array in programming languages.
  - They can contain any mix of data types, including nested sequences or mappings
  - Sequences are usually displayed on multiple lines, where each element begins with a dash, followed by a space, and ends with a line break
  - Sequences can also be written on a single line surrounded by brackets
    - Numbers: [8, 12, pi]

YAML Scalars (Data types)

- All remaining data types in YAML are scalars or single value data types
  - These include: integers, floating-point numbers, Booleans, null, & strings
    - Numbers: Any number that doesn’t have a decimal point is an integer, numbers that do are floating-point
    - Booleans: The keywords True, On, and Yes evaluate to true. False, Off, and No evaluate to false
    - Null: A null value can be represented by either ~ or null (written as Null, null, or NULL)
    - Strings: Strings generally do not need quotes, two notable exceptions are as follows:
      - Use single or double quotes to create a value that would normally be interpreted as a different data type to be a string, i.e., “10” or “null”
      - Use double quotes to allow specific sequences to be escaped instead of treated as literals, such as “\n” representing a line break

API Status Codes

Detecting API Issues by Understanding Status Codes

API Status Codes

REST is built on the HTTP protocol. Therefore, our APIs should use HTTP status codes to ensure consistent and predictable behavior.

Acknowledge & Success (2xx)	
200 - OK	The request has succeeded
201 - Created	The request has succeeded & a new resource has been created
202 - Accepted	The request has been received but not completed yet
204 - No Content	The server has fulfilled the request but does not need to return a response message

Redirection (3xx)	
300 - Multiple Choices	The request has more than one possible response
301 - Moved Permanently	The URL of the requested resource has been changed permanently
302 - Found	The URL of the requested resource has been changed temporarily
303 - See Other	The response can be found under a different URI & should be retrieved using a GET method on that resource
307 - Temporary Redirect	Get the requested resource at another URI with the same method that was used in the previous request

https://restfulapi.net/http-status-codes/

API Status Codes (Cont.)

Client Error (4xx)	
400 - Bad Request	The request could not be understood by the server due to incorrect syntax
401 - Unauthorized	The request requires user authentication information
403 - Forbidden	Unauthorized request, the client does not have access rights for the resource/content
404 - Not Found	The server can not find the requested resource
405 - Method Not Allowed	The request HTTP method is known by the server but has been disabled and cannot be used for that resource

Server Error (5xx)	
500 - Internal Server Error	The server encountered an unexpected condition that prevented it from fulfilling the request
501 - Not Implemented	The HTTP method is not supported by the server and cannot be handled
502 - Bad Gateway	The server got an invalid response while working as a gateway to get the response needed to handle the request
503 - Service Unavailable	The server is not ready to handle the request
511 - Network Authentication Required	The client needs to authenticate to gain network access

Topics Covered Today

- API Overview (REST APIs)
- API Maturity Model
- z/OS Connect
- API Development Lifecycle
- Design First vs Code First
- OpenAPI Specification
- Swagger Tooling
- API Status Codes

Any Additional Questions?

Thank you for listening and for your participation

Links to Swagger tooling

- Swagger Editor
  - Online editor: https://editor.swagger.io/
  - Download locally from GitHub repository: https://github.com/swagger-api/swagger-editor
- Swagger Codegen
  - Download locally from GitHub repository: https://github.com/swagger-api/swagger-codegen
- Swagger UI
  - Download locally from GitHub repository: https://github.com/swagger-api/swagger-ui

Additional Resources

- HTTP Status codes: https://www.restapitutorial.com/httpstatuscodes.html
- OpenAPI Object: https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md#openapi-object
- Info Object: https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md#info-object
- Server Object: https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md#serverObject
- Path Object: https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md#paths-object
- External Docs Object: https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md#externalDocumentationObject