Did you know that 90% of Fortune 400 companies report the use of or publishing of RESTful web services? Web service APIs are everywhere, and OpenAPI and the Swagger tools are the standard on which they are designed and documented.

The term Swagger may be more familiar to you than OpenAPI. That's because the Swagger API, the foundation of OpenAPI, came first. The Swagger API is an open source project created to address a very common issue with RESTful web services.

At the time of the Swagger API creation, there was no standard way to describe how to use a RESTful web service. RESTful web services came into common use by 2010, but most organizations that employed REST offered incomplete or wildly inconsistent documentation on how to use them. The URLs, parameters, and payloads of REST requests and responses were primarily documented through examples. The quality of this documentation varied. This made it difficult to know precisely how to use RESTful web services from one organization to the next. It also made it difficult, if not impossible, to create tools that could read documentation and automatically generate code. For client SDKs used to call web services or server stubs used to process web service calls.

In 2011, Tony Tam wanted a way to communicate exactly what endpoints, parameters, and payloads were expected for the RESTful web services he was developing, in a way that was consistent and toolable. It was, as are all great open source projects, inspired by an itch that Tony wanted the scratch. In 2015, SmartBear software, which was by then the primary contributor to the Swagger API, donated Swagger API 2.0 to the newly-formed OpenAPI initiative, a consortium of companies including SmartBear, IBM, Microsoft, and Google. In 2016, Swagger API 2.0 was renamed OpenAPI 2.0. This has caused a bit of confusion as people tend to think that the Swagger API and OpenAPI are two different things. They are not. When you see a reference to Swagger in the context of web services, most people are talking about OpenAPI. In 2017, OpenAPI 2.0 was updated to 3.0.

Today, support for either or both versions of OpenAPI is commonplace. While OpenAPI is the industry standard format that describes how to use RESTful web services, the Swagger tools, are a set of open source and commercial tools used to create, utilize, and maintain OpenAPI documents.

The Swagger Tools

Swagger tools are a set of open source tools used to create and work with the OpenAPI standard. There are five Swagger tools we are concerned with, Swagger Editor, Swagger UI, Swagger Inspector, Swagger Codegen, and SwaggerHub.
The Swagger Editor is a web editor accessible by browser that allows you to design and edit OpenAPI documents. The editor supports the use of YAML to describe each web service in an organization's web API.

The Swagger UI tool generates Pretty documentation based on the OpenAPI definitions. It generates a web page for each web service operation, which clearly illustrate the URLs, paths, and parameters.

The Swagger Editor and the Swagger UI complement each other and are shown side by side. So as you develop the OpenAPI document in the Swagger Editor, you can simultaneously see the Swagger UI web documentation generated by the OpenAPI.

Swagger Inspector provides a UI for sending and receiving request calls. It allows you to try and test RESTful web services to determine the form of the request and responses. If you have used Postman, then Swagger Inspector will look familiar to you.

Swagger Codegen takes an OpenAPI document as an input and outputs a complete client SDK and server stubs in any one of several programming languages, including JavaScript, Python, Java, C#, Ruby, and others for several different web service frameworks, including, but not limited to, Spring, Node.js, Microsoft ASP.NET, Ruby, Rust, C++, PHP, and others.

While the Swagger Editor, UI, Inspector, and Codegen are all free and open source, SwaggerHub is a commercial product offered by SmartBear. It allows you to use Swagger tools in the cloud maintained by SmartBear so that you can publish your APIs for private and public consumption and make available the Swagger UI documentation you generated. You can also install SwaggerHub on premise. SwaggerHub provides a number of features that allow organizations to coordinate, collaborate, and standardized RESTful web API development across all of its web API projects. SwaggerHub offers a lot in terms of allowing for collaboration, reuse of OpenAPI constructs, and enforcement of your organization's best practices. It's not free like other tools, but it is very powerful.

CREATE OPEN API DOCUMENT WITH SWAGGER EDITOR

How to start + get api + api reference/definitions

```yaml
swagger: '2.0'

info:
  title: Customer API
  description: This is the Customer API. This provides restful services for Custsomer
    web service.
  version: '1.0.0'
  contact:
    name: Sanskar Dwivedi
    url: https://github.com/mrinalsanskar
    email: sanskardwivedi716@gmail.com
  license:
    name: Apache'2.0'
    url: http://www.apache.org/licenses/LICENSE-2.0.HTML

paths:
  /customer:
    get:
      summary: reads a customer's data
      operationId: getCustomer
      description: "This operation provides a view of the customer's data in JSON.
                The operation uses the customerId to identify the customer in a
            query string."
      produces:
        - application/json
      parameters:
```

```
      parameters:
        - in: query
          name: customerId
          description: pass an optional customer id to get customer data
          type: integer
      responses:
        200:
          description: search results matching criteria
          schema:
            $ref: "#/definitions/Customer"

        404:
          description: employee with this id does not exist.

definitions:
  Customer:
    type: object
    properties:
        customerId:
          type: integer
        accountValue:
          type: number
        customerName:
          type: string
        active:
          type: boolean
```

```
        active:
          type: boolean
        address:
          type: boolean
          properties:
            name:
              type: string
            street:
              type: string
            city:
              type: string
            state:
              type: string
            zip:
              type: string
        contacts:
           type: array
           items:
            type: object
            properties:
              name:
                type: string
              email:
                type: string
              phone:
                type: string
```

DELETE API:

```
delete:
  summary: delete an existing customer
  operationId: deleteCustomer
  description: Deletes an existing customer
```

```
    produces:
      - application/json
    paramters:
      - in: path
        name: customerId
        description: the id of the customer
        required: true
        type: integer
    responses:
      200:
        description: The customer was deleted
        schema:
          $ref: "#/definitions/Customer"
      404:
        description: Customer Not Found
      500:
        description: Internal Server Error
```

PUT API:

```
/customer/{customerId}:
  put:
    summary: update existing customer
    operationId: updateCustomer
    description: Updates an existing customer with new data
    consumes:
      - application/json
    parameters:
      - in: body
        name: body
        description: the updated customer data in JSON format
        required: true
        schema:
          $ref: "#/definitions/Customer"
      - in: path
        name: customerId
        description: the id of the customer to update
        required: true
        type: integer
```

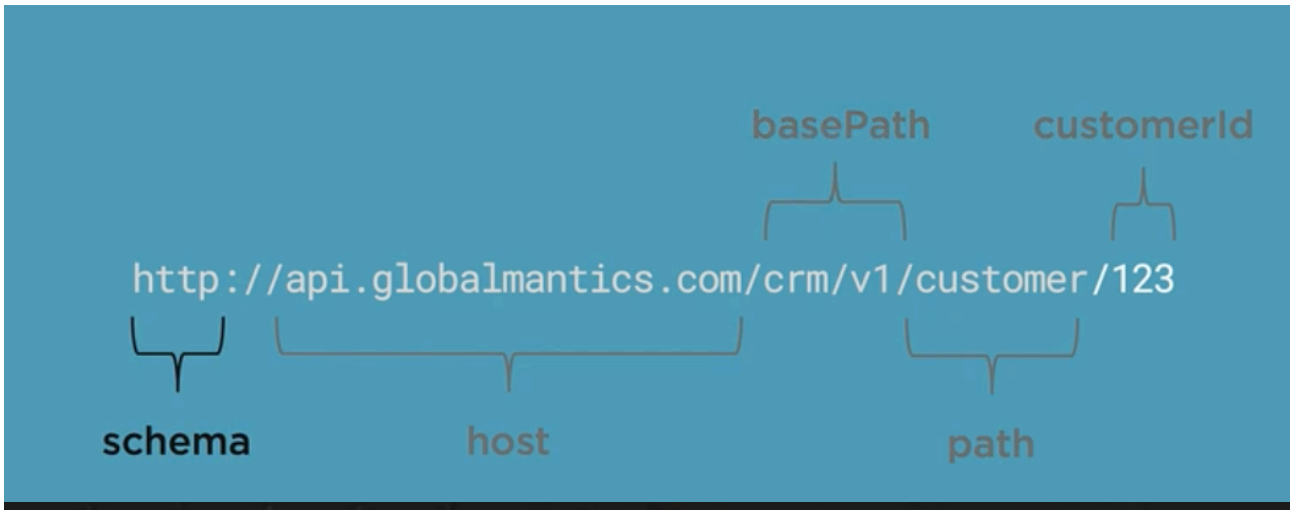
```
  - in: path
    name: customerId
    description: the id of the customer t
    required: true
    type: integer
responses:
  200:
    description: Success.
  404:
    description: Customer Not Found
  500:
    description: Internal Server Error
```

POST API:

```
    post:
      summary: adds a new customer
      operationId: addCustomer
      description: Add a new customer to the syste
      produces:
        - text/plain
      consumes:
        - application/json
      parameters:
        - in: body
          name: body
          description: the new customer data in JS
          required: true
          schema:
            $ref: '#/definitions/Customer'
```

```
responses:
  '200':
    description: "successful operation"
    schema:
      type: integer
  '405':
    description: Invalid Input
```

HOST, BASE PATH & SCHEME :




```
host: api.globlmantics.com
basePath: /crm/v1
schemes:
  - http
  - https
```

HOST, BASE PATH & SCHEMES

A web service has to be located somewhere; otherwise, you can't use it. By located, I mean there has to be a web address for the RESTful services provided by Globomantics' customer API. The paths that we defined for the GET, POST, PUT, and DELETE operations provide only relative paths for these services. What we need next is the domain, which is referred to as the host.

Add a host element at the bottom of the OpenAPI definition. It's at the same level as the info object and the definitions object. The host we are going to use will be api.globomantics.com, so we'll set that as the host field value.

If we left it like this, then the URL used to read, create, update, and delete customer data would simply be the domain plus the path for each method. That's pretty simple, but is that what we really want? Remember that Globomantics has more than the customer API. That just happens to be the API you're defining.

We've been told that the base path for all the web services will be crm, for customer relations management, and v1 to indicate that this is the first version of Globomantics' suite of web APIs. So, we'll define the standard base path like this. **(BASE PATHS)**

Finally, we need to declare the network protocol used for communicating. In all RESTful web services, this will either be HTTP or the most secure HTTPS. We'll declare both. **(SCHEMES)**

OpenAPI 2.0: SecurityDefinitions and Security

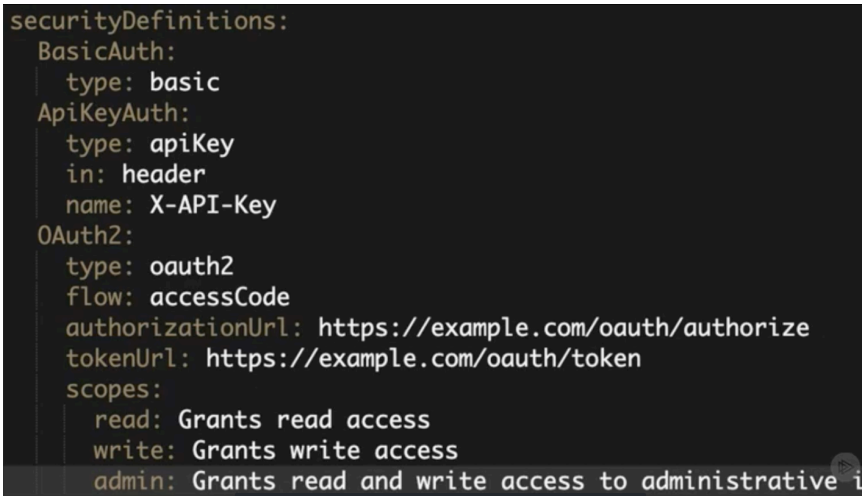
OpenAPI 2.0 supports three types of authentication, **basicAuth, API Key, and OAuth.** An OpenAPI document can specify all three, just one or two, or no authorization at all.

We will use basicAuth:

Basic authentication is part of HTTP 1.1 and 2.0 standards. It's a simple username and password authentication which is passed in the headers of the HTTP request. In most cases, when you sign into a website, you use Basic authentication.

Although the password is Base64 encoded, as is all text in HTTP, it's not encrypted. For that you have to use HTTPS, SSL over HTTP, declared under schema.

At the root of an OpenAPI document, the same level as the infoDefinition and host fields, you identify the types of authorization your web API supports in a securityDefinitions field. If I was using all three authorization types, Basic authentication, API Key, and OAuth, it might look like this.



Since we're only using Basic authentication, we need only specify that in the securityDefinitions. We do that by writing BasicAuth:, and then for the type, put basic. The second field you need to declare is security. This indicates which type of security is used for a specific operation. The empty

brackets simply mean that there's no property specific to Basic authentication. You would also use this for API Key.

Only OAuth 2.0 requires that you fill the array with any additional information. But the empty array is still required for Basic authentication and the API Key. I could go on to define authentication for every operation. But there is an alternative. You could make things a bit simpler by declaring a security field at the root level just below the securityDefinitions like this.

This applies to all the operations that do not have their own security field declared directly. While the root security field applies to all the operations that don't have a specific security field, any operation that declares its own security field overrides the root security for that specific operation.

In most cases, the security requirement is the same for all operations declared, so the root security element is used more frequently than the operation level security elements.

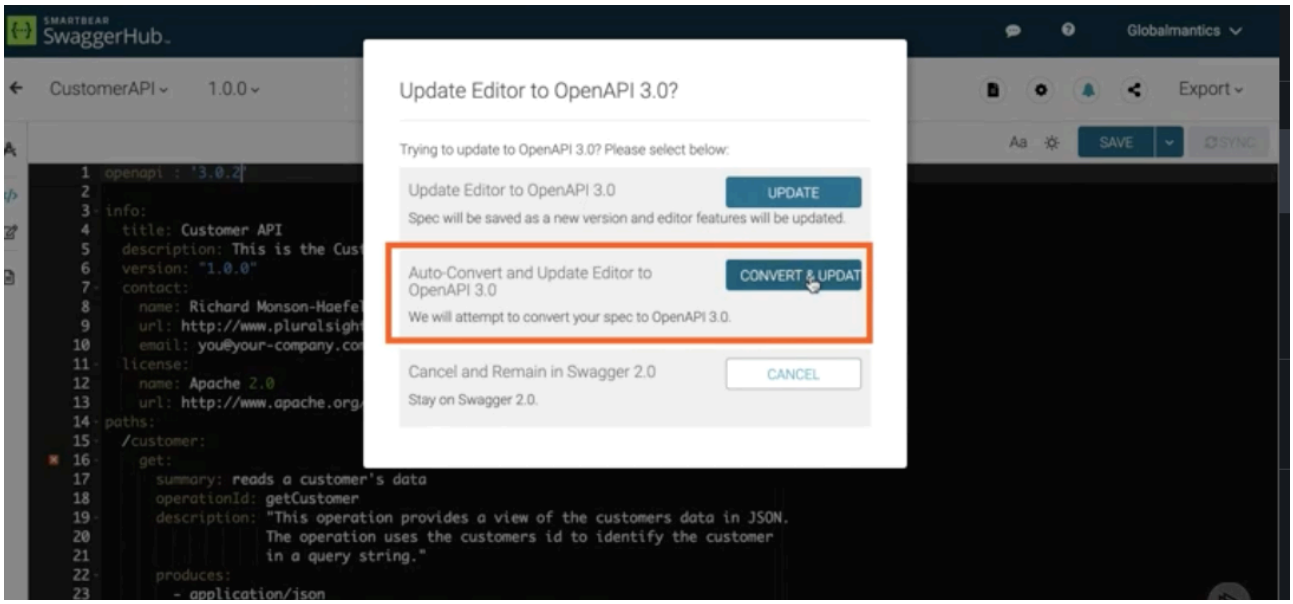
Converting to OpenAPI 3.0

To convert the Customer API we created from OpenAPI 2.0 to OpenAPI 3.0, Swagger Editor can do this automatically for you. The differences are not great, but they are important.

1. Producers and the consumer fields were replaced with content fields in the response object.
2. Definition of parameters in the body of the request has moved from the parameters field to the new requestBody field.
3. A much simpler way to declare a host, basePath, and schema using the new server field and how multiple servers can be declared for the same web service and why.

   To get started, login to your free SwaggerHub account. Now open your customer API. You're looking at the customer API for OpenAPI 2.0, also known a Swagger 2.0, format. To convert this from OpenAPI 2.0 to 3.0, all you have to do is change the name of the version and its value, like this.

   Within a second or two of making this change, the editor will pop up the dialog box with three options. You can leave the document in 2.0, but change the editor to OpenAPI 3.0, or you can convert the document to OpenAPI 3.0 and switch the editor to the new version, or you can cancel. Select the option to convert both the document and the editor to OpenAPI 3.0. The results are immediate.

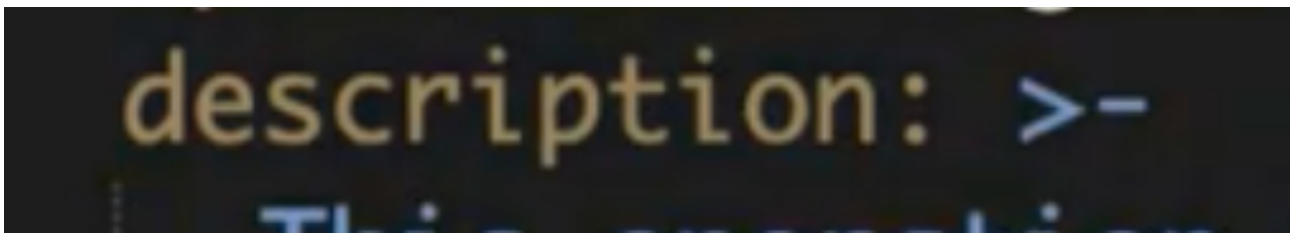


Notice the version of the customer API has changed from 1.0.0 to 1.0.0-oas3. That's the default. You can change that version if you like, by editing the document's version field, but I'm going to leave it as is.

You can switch back to the previous version by clicking on the link and choosing the 1.0.0 version. This switches to the first version of the document and changes the editor to handle OpenAPI 2.0. Changing back to the new version is just as easy. As I've said, the changes from OpenAPI 2.0 to 3.0 are not that great.

In fact, if you look at the info object, it's not all that different. Just the document version has been updated automatically to reflect the conversion to 3.0. Some aspects of the paths has changed and others have not. The name of the path field follows the same convention as used in 2.0, the path starts with a forward slash followed by zero or more subdirectories and path parameters. The method is declared next and is usually GET, POST, PUT, or DELETE. The summary, operationID, and description remain the same.

You'll notice that the description has a strange character sequence at the top. This simply indicates that the description is a block quote, an indented paragraph.



OpenAPI 3.0 supports the use of CommonMark markdown formatting version 0.27

The Content and RequestBody Fields in OpenAPI 3.0

Let's see how the content and requestBody field, new to OpenAPI 3.0, are used to declare the MIME type and schemas of data in the body of the HTTP requests and responses.

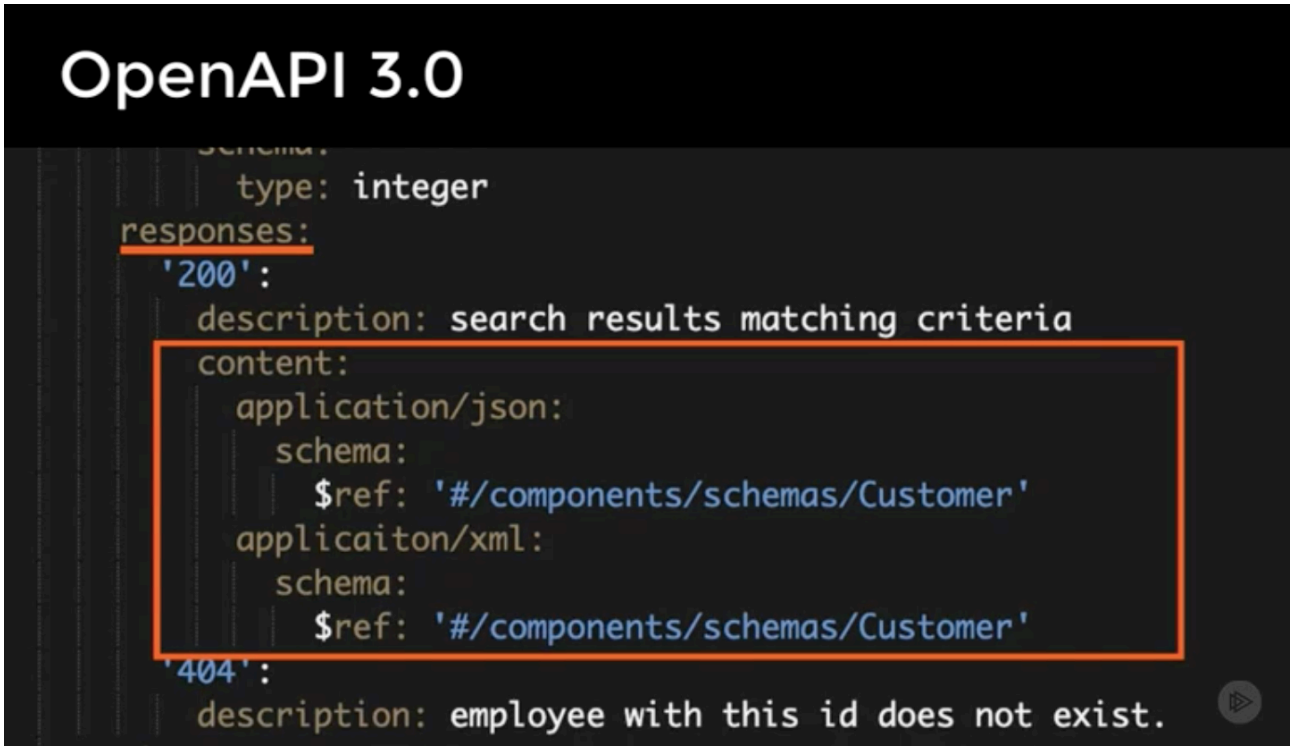
In OpenAPI 2.0, the consumes and produces fields declare the supported MIME types for the data in the body of the request and response, respectively. In 3.0, the produces and consumes fields are gone. They have been replaced with what is called content fields.

The content field is declared within parameters, responses, and the new requestBody fields and corresponds to support for different MIME types and their associated schemas.

As an example, consider the GET operation defined in OpenAPI 2.0 compared to 3.0. In the 2.0 version, the produces field tells us that the web service will return content in the body of the response that will either be JSON or XML format. This corresponds to the 200 response, which returns the customer data with a specific schema.

In OpenAPI 3.0, the produces field is omitted and the MIME type and the schema of the data are declared together under the response code using the new content field. The content field declares both the MIME type of the data and its schema for a specific HTTP response code.

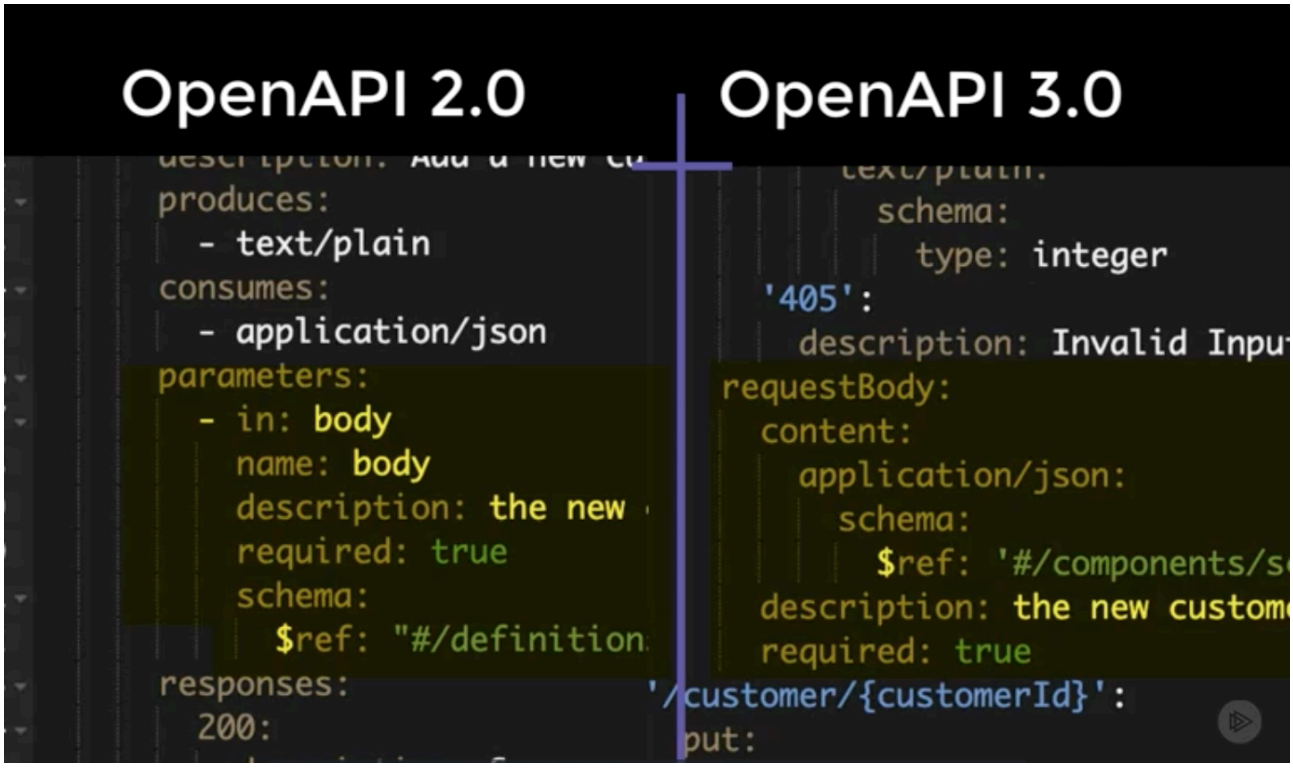
In the case of our GET operation, both versions list two MIME types for the response body, JSON and XML, but in OpenAPI 3.0 the MIME-type is tied directly to the schema. This can be very useful if, for example, the response in XML has a different structure than the response in JSON. Each could reference a different schema rather than requiring that both follow a single schema, as was the case with OpenAPI 2.0.



For responses with no data in the body, such as the 404, the declarations are the same in OpenAPI 3.0 as they were an OpenAPI 2.0.

The content fields are used in the new requestBody.

As an example, take a look at the POST operation in the earlier version of the customer API. In the post-operation, we send the customer data to the server, which is declared in the parameters field of the OpenAPI 2.0 version. In the new OpenAPI 3.0 version, the body parameter type has been replaced with the new requestBody field, which is at the same level as the parameters and responses fields.



The requestBody allows you to declare the MIME type and schema of data in the HTTP Request body. In this case, it lists a MIME type of JSON with the schema referencing the customer object. It could also have listed an XML MIME type with the same or a different schema. Moving the declaration of the request body parts out of the parameters and into its own field, **requestBody, makes a document easier to read and more flexible.**

Servers, Security and Components

In OpenAPI 2.0, you specify the domain, basePath, and network protocol using the host, basePath, and schema fields. The biggest limitation of this feature is that it only allowed you to define one URL for all the web operations defined in a document. If you want to add development or staging URLs, you'd have to create new documents for each endpoint.

This was changed in OpenAPI 3.0 with the introduction of the servers field, which is an ordered list of URLs allowing not only for more than one endpoint to be specified, but a lot more flexibility in their use.

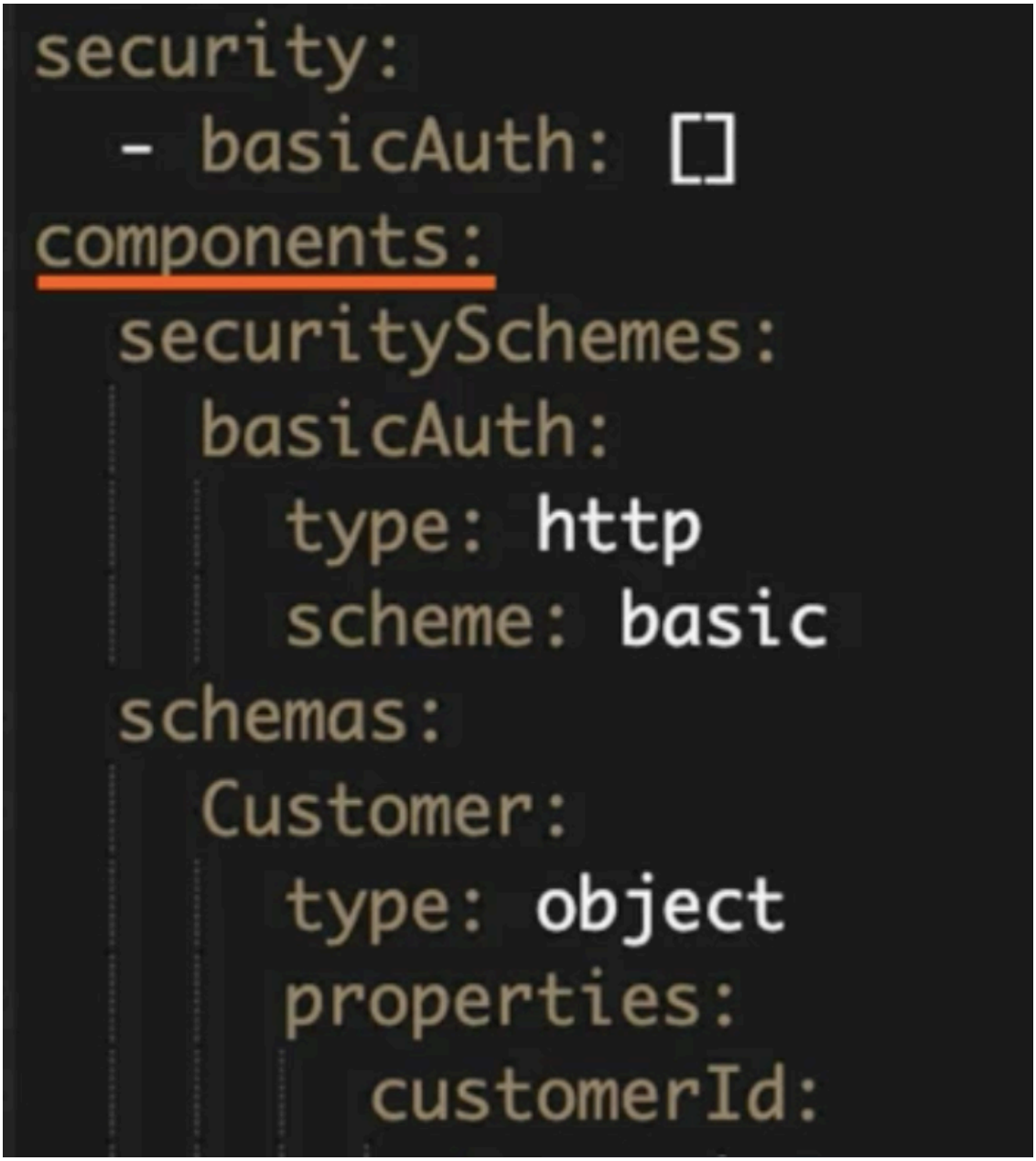
Let's see this in action In the OpenAPI 3.0 version of our customer web service. You can see how the new servers fields addresses multiple URLs already.

```yaml
servers:
  - url: 'https://api.globlmantics.com/cmr/v1'
    description: Internal Production.
  - url: 'https://stagging.globlmantics.com/cmr/v1'
    description: Internal Stagging
  - url: 'https://development.globlmantics.com/cmr/v1'
    description: Internal Development.
  - url: 'https://{customerId}.globlmantics.com/cmr/{version}'
    description: Customer Production
    variables:
      customerId:
        default: demo
        description: the customer id for subdomain
      version:
        default: v1
        enum:
          - v1
          - v2
          - v3-beta
```

We have two network protocols in use, http and https, each of which is a separate element in an order list of servers. Let's get rid of the http URL. We don't want expose that to the world. Each URL is an object with multiple fields. For example, we can add a description to our https URL. The endpoint is for internal use, meaning that it's used only by Globomantics employees in production. Globomantics also has internal endpoints for staging test releases and for development. Globomantics CRM Cloud service is multi-tenet, meaning there is one endpoint for every customer. Each customer uses a different sub domain.

How is that supported? Well, you could add a URL for every customer but Globomantics has dozens of them. There's a better way. Use path parameters. These work the same as path parameters used in the paths of operations. First, I'll add another URL, which I'm just copying and pasting from the internal production URL, and then change the sub domain from API to a parameter with a variable name customerId. We'll add a description. The next thing is to define the variable. For that, I use a variables field in the endpoint, and add in the customerId variable, using the path parameter above. I'll add a comment to describe it. I also added a default value, which I'll set the demo, so that it can be used by sales people in presentations or potential customers who are evaluating the services. You can even specify a range of valid values by creating an enumeration. For example, let's replace V1 at the end of the customer path so that it's a variable like this.

Then I can define another variable below customerId named version, and give it a default value of v1,. but I can also use an enum field to restrict the possible values to v1, v2, or v3-beta. As you can see, the new server field in OpenAPI 3.0 is much more powerful and flexible than the host, basePath, schema used in OpenAPI 2.0.



There have been changes in the way security is declared as well. In OpenAPI 2.0, the authentication was declared in two fields, the security and securityDefinitions. In OpenAPI 3.0, this is only changed in terms of reuse.2 In 3.0, the security field looks much as it did in 2.0, but the securityDefinitions has been renamed securitySchemes and moved out of the route and under a new field called components. The components field replaces the definitions field used in OpenAPI 2.0 and expands its functionality beyond reusable schemes to several other fields in OpenAPI 3.0, including securitySchemes, responses, request bodies, parameters, headers, links, and others. This provides developers like you with a lot more flexibility and streamlines your definitions by eliminating repetition, both welcome improvements.

OpenAPI 2 remains in wide use today

OpenAPI 3 is growing in popularity

OpenAPI 3: more flexible and organized

Swagger Editor: OpenAPI 2 -> 3

Changes:

- Version has changed from swagger:‘2.0’ to openapi:‘3.0.2’
- info filed is unchanged
- content field replaces consumes and produces
- servers field replaces host, basePath, and schema for URLs
- components field replaces definitions

**\\\ SUMMARY **

Swagger UI is autogenerated

Very accurate documentation

Every aspect of your OpenAPI document

Easy to read, navigate and interpret

Consistent layout for all Web APIs

Using Swagger UI: Two options

- Subscribe to the SwaggerHub product
- Use Open source Swagger UI project
  - Node.js
  - Google "Swagger UI opensource install"

**\\ SWAGGER UI **

Swagger Codegen

Advantages	Disadvantages
Code generated by Swagger Codegen is fast	The generated code is fixed. Not a lot of flexibility.
Code can evolve as the OpenAPI doc evolves	Regenerating client SDKs and server stubs leaves out business logic

**\\ SWAGGER CODEGEN **

\\ SWAGGER INSPECTOR \\

Load an OpenAPI document into the Swagger Inspector to test its operations

Access a web service without an OpenAPI definition and create one