API DEVELOPMENT PATH

OPENAPI SPECIFICATION QUICK GUIDE

v1.0

A downloadable resource of the *OpenAPI Specification for Software Developers* course

WWW.CODEWITHPRAVEEN.COM

Preface

Hi there!

I hope you are doing good.

I've created this OpenAPI quick guide to help you with your everyday programming in API. You can **use this as a reference document** to check the steps involved in creating an OpenAPI specification. You can find the key points you need to consider for each step while defining an OpenAPI Specification.

OpenAPI Specification version used: **3.0.0**

OpenAPI is the most popular way to work with APIs. **This downloadable resource is part of the OpenAPI(Swagger) Specification for Software Developers course**, which covers the essentials of OpenAPI concepts that any API programmer **must** know. Click here to know more about the companion course.

See you in the course video!
Praveen.

Table of Contents

Preface

Table of Contents

Step 1: Add the OpenAPI Version

Step 2: Add Metadata

Step 3: Add Additional References

Step 4: Add Server Details

Step 5: Add Tags to Group API Operations

Step 6: Add Paths (Part 1: Path Details)

Step 6: Add Paths (Part 2: Operations)

Step 6: Add Paths (Part 3: Parameters)

Step 6: Add Paths (Part 4: Responses)

Step 6: Add Paths (Part 5: Request Body)

Step 7: Add Components (Part 1: Components Object)

Step 7: Add Components (Part 2: Schema)

Step 7: Add Components (Part 3: Response)

Step 8: Add Security

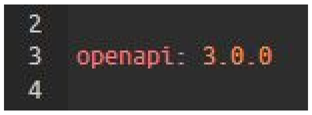
Step 1: Add the OpenAPI Version

As the first step, you need to include information about the OpenAPI version itself.

The **version you use defines the structure** of the API definitions that you will be creating. Depending upon the OpenAPI version you are targeting, the behavior will change.

You use the keyword ***openapi*** to enter the version. This is a **MANDATORY** parameter.

OpenAPI Specification is shortly referred to as OAS.



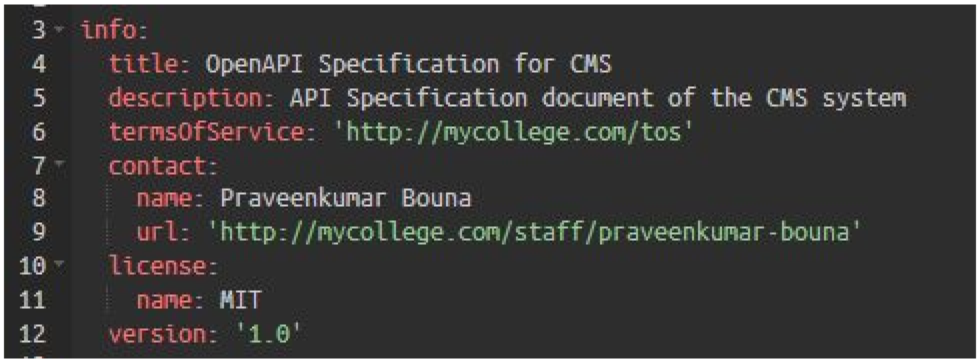
```yaml

openapi: 3.0.0

```

Fields: None

- As of this writing, the latest major version is 3.0.0.
- Note that from 3.1, the OpenAPI Consortium has decided to change the versioning format from the current x.x.x to x.x format.
- **Minor versions are being released** then and there such as 3.0.1, 3.0.2, 3.0.3. However, they generally improve readability and accuracy without any change in the behavior of the spec.
- The tools that support OAS 3.0 SHOULD be compatible with all OAS 3.0.* versions. Similarly for the minor version.
- Do not worry, you just need to **concentrate on the major version.** For example, 3.*.*.

Step 2: Add Metadata

In this step, you include **details about the API** using the *info* section. In short, you provide the metadata about the API. The metadata may be used by the clients or editing or documentation generation tools.



```yaml
info:
  title: OpenAPI Specification for CMS
  description: API Specification document of the CMS system
  termsOfService: 'http://mycollege.com/tos'
  contact:
    name: Praveenkumar Bouna
    url: 'http://mycollege.com/staff/praveenkumar-bouna'
  license:
    name: MIT
  version: '1.0'
```

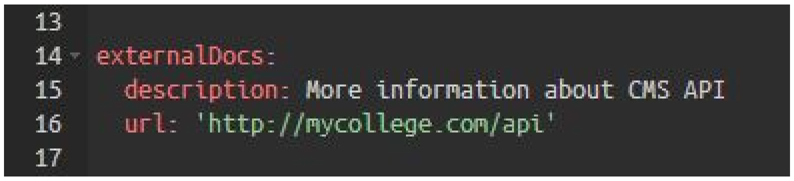
You use the ***info*** keyword to add metadata to an API. This is a **MANDATORY** parameter.

Fields: title, description, termsOfService, contact, license, version.

- The ***title*** field is used to enter the main title of the API document whereas the ***description*** field is used to enter a good description for the document. Note that the ***title*** field is **MANDATORY.**
- Moreover, you include the version of your OpenAPI Specification document using the ***version*** parameter. It is a **MANDATORY** parameter. Note that you need to enclose the version within single quotes to treat it as a string. Eg. '1.0' and not 1.0
- ***termsOfService****,* optionally*,* can be used to enter the URL of your terms of service.
- In case you want to include contact details to contact for any discussion regarding this API, you add it using the ***contact*** field. It takes name, URL, email. All are optional. The contact details can refer to a specific person or an organization.
- Finally, the license information for the API can be added using the ***license*** field. It takes a **name (MANDATORY)** and **URL**.

Step 3: Add Additional References

In Step 3, you add external documentation resources to the API documentation.



```
externalDocs:
  description: More information about CMS API
  url: 'http://mycollege.com/api'
```

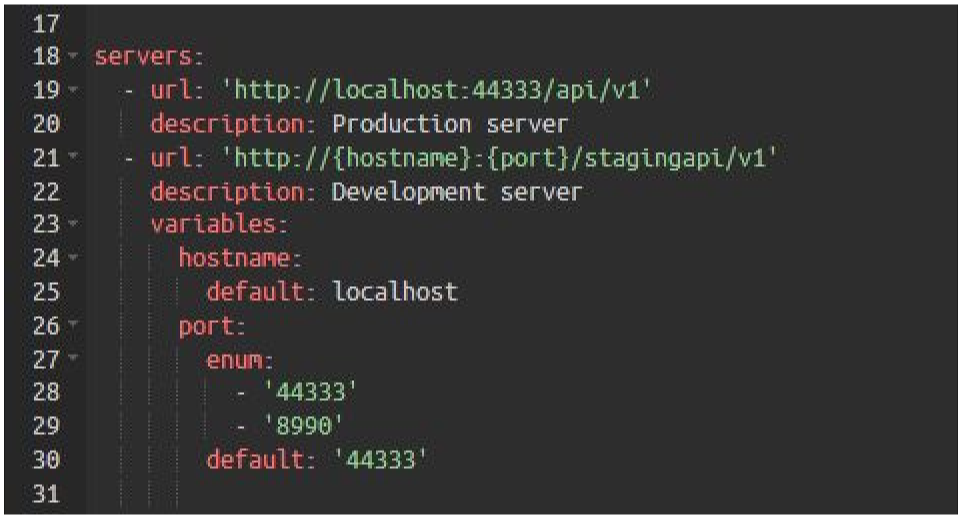
Use the keyword ***externalDocs*** to add optional external documentation references. It allows referencing an external resource for extended documentation.

Fields: description and URL.

- The ***description*** parameter can be used to provide an optional description whereas the ***url*** parameter holds the actual external documentation address.
- ***url*** is a **MANDATORY** parameter if externalDocs is added to the document.

Step 4: Add Server Details

Next, you add the server details that will be used by the clients to connect to your web services.



```yaml

servers:
  - url: 'http://localhost:44333/api/v1'
    description: Production server
  - url: 'http://{hostname}:{port}/stagingapi/v1'
    description: Development server
    variables:
      hostname:
        default: localhost
      port:
        enum:
          - '44333'
          - '8990'
        default: '44333'

```

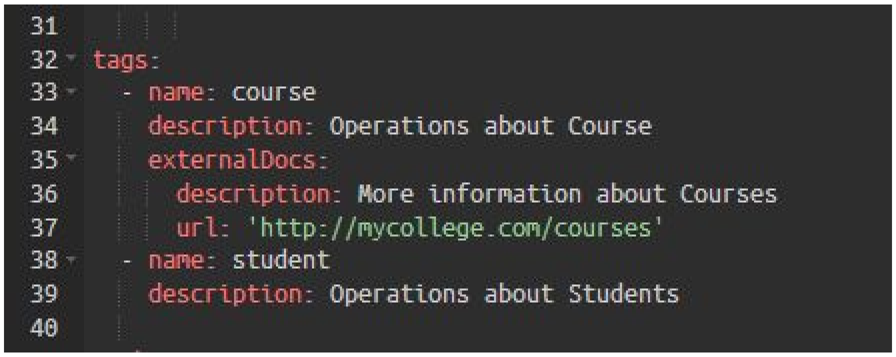
Use the keyword ***servers*** to add optional server details to your API specification. The servers section specifies one or more servers that host the current API. You can define one or several servers, such as production and staging.

Fields: url, description, variables.

- ***url*** field holds the base URL for your server. All API paths are relative to the server URL. Eg. For /users path, the final URL may be http://api.example.com/v1/users. ***url*** is a **MANDATORY** field.
- The optional ***description*** parameter can be used to provide a description.
- The optional ***variables*** field is used to create a map between the name and its value. It is used to make the URL dynamic so that the user can pass value during execution. Each variable can have **3 properties - enum, default, description**.
    - *enum* - a list of values. Note that in YAML, an array is represented using hyphens (-).
    - *default* - the default value to use if not passed.
    - *description* - description for the server variable.

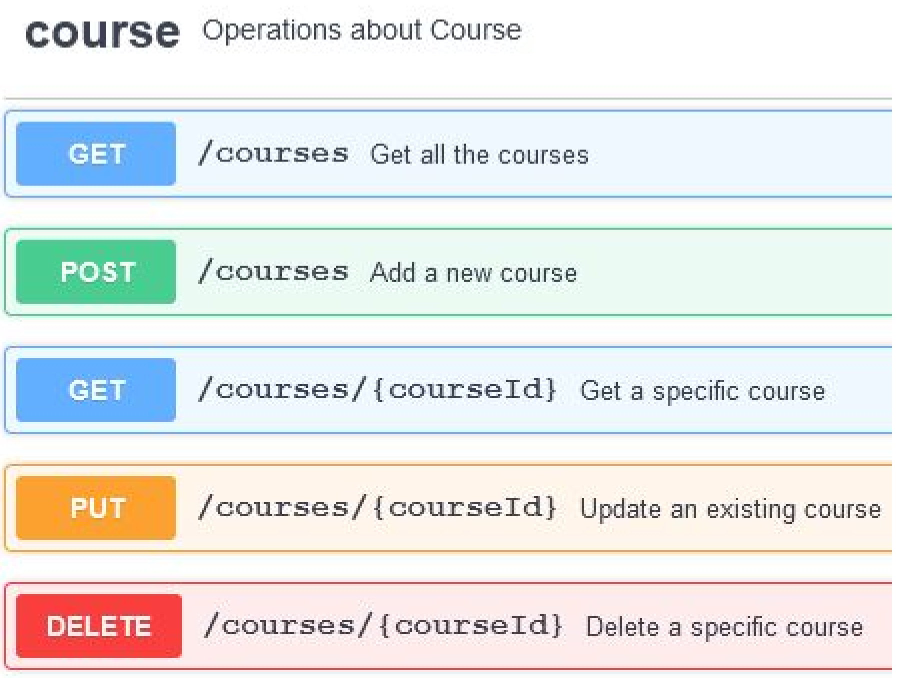
Step 5: Add Tags to Group API Operations

You add tags to group the API operations that are added to the API documentation.



```
tags:
  - name: course
    description: Operations about Course
    externalDocs:
      description: More information about Courses
      url: 'http://mycollege.com/courses'
  - name: student
    description: Operations about Students
```

Tags can be **used for the logical grouping of operations by resources** or any other qualifier. For eg., you can group all course-related operations such as add course, delete a course, update course, etc to a tag named *course*. In such a case, those operations will be shown grouped under that tag.


course Operations about Course
GET /courses Get all the courses
POST /courses Add a new course
GET /courses/{courseId} Get a specific course
PUT /courses/{courseId} Update an existing course
DELETE /courses/{courseId} Delete a specific course

They are referred to while defining an operation.

You use the ***tags*** keyword to define a tag.

It is **NOT** mandatory to have tags defined. In such a case, the operations will be grouped under *default*.

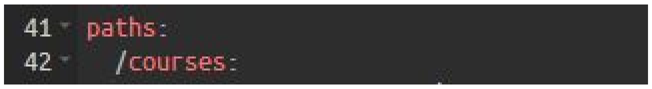
Fields: name, description, externalDocs

- The ***name*** field holds the name of the tag.
- ***description*** fields hold an optional description.
- ***externalDocs*** holds optional external documentation references for this tag.

Step 6: Add Paths (Part 1: Path Details)

In this part, you define a new path that will hold various operations for a resource.

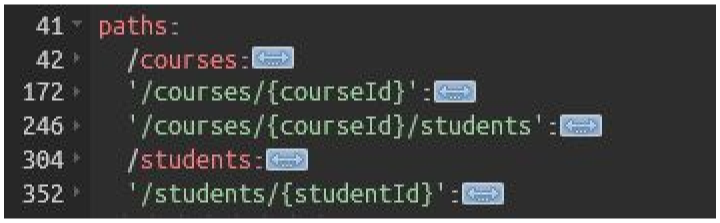
Paths define the actual path of the resource, its operations, requests, responses, and so on. Paths are defined using the ***paths*** keyword. It is then followed by the actual path.



```
paths:
  /courses:
```

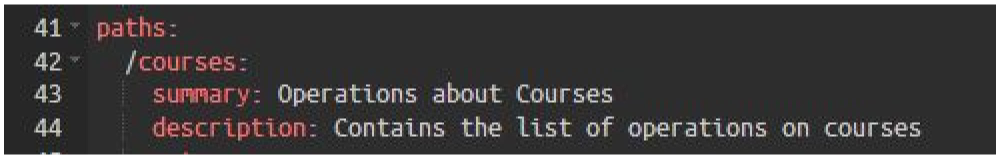
Note that the path is a **relative path** to the individual endpoints. The path is appended to the URL from the ***servers*** section to construct the full URL.

The paths can contain ***parameters*** that are dynamic and passed while calling this API. Parameters are covered in a forthcoming chapter.



```
paths:
  /courses:
  '/courses/{courseId}':
  '/courses/{courseId}/students':
  /students:
  '/students/{studentId}':
```

Fields: summary, description, get, post, delete, put, options, head, patch, trace, servers, parameters.

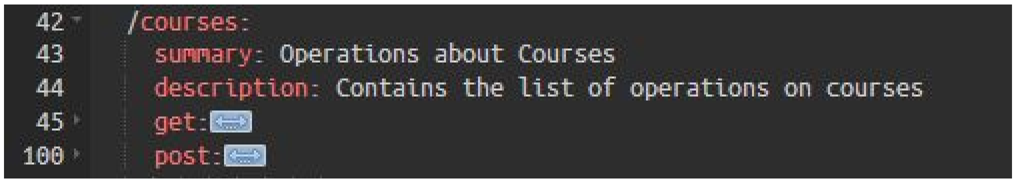


```
paths:
  /courses:
    summary: Operations about Courses
    description: Contains the list of operations on courses
```

- The optional ***summary*** field captures a nice title for the path. I **RECOMMEND** adding.
- Optional ***description*** fields hold an optional description. I **RECOMMEND** adding.

Step 6: Add Paths (Part 2: Operations)

In this part, you add operations such as GET, POST, DELETE, etc for the ***paths*** that you defined in the previous part.

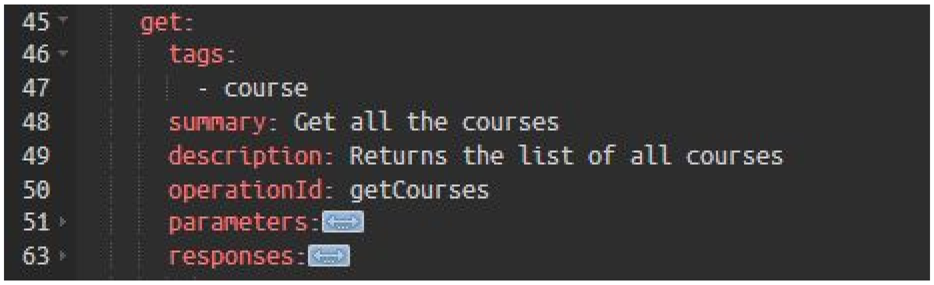


```
  /courses:
    summary: Operations about Courses
    description: Contains the list of operations on courses
    get:
    post:
```

To define an operation, use the keyword:

- ***get***, to return some details of a resource from the system.
- ***post***, to add an item to the resource collection.
- ***put***, to update an existing item.
- ***delete***, to delete an existing item

Fields: tags, summary, description, externalDocs, operationId, parameters, requestBody, responses, deprecated, and so on.

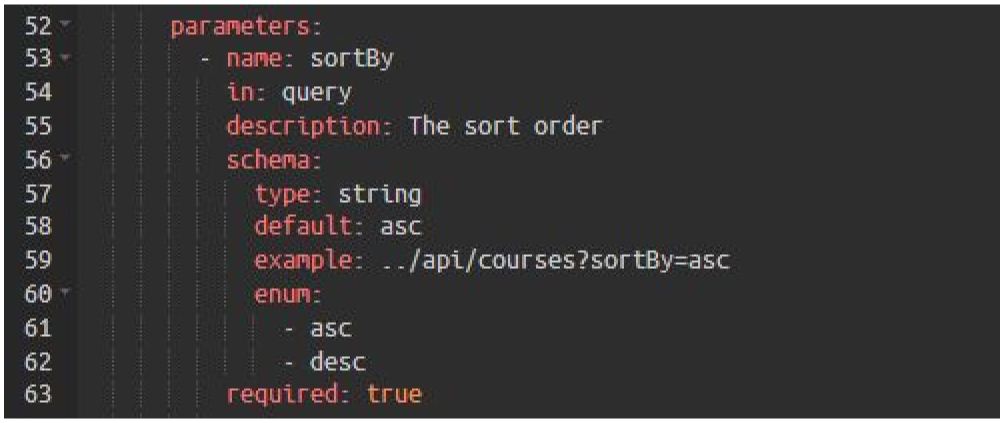


```
    get:
      tags:
        - course
      summary: Get all the courses
      description: Returns the list of all courses
      operationId: getCourses
      parameters:
      responses:
```

- Use the optional ***tags*** field to group this operation under a related topic. I **RECOMMEND** adding.
- The optional ***summary*** field captures a nice title for the path. I **RECOMMEND** adding.
- Optional ***description*** fields hold an optional description. I **RECOMMEND** adding.
- Use the ***operationId*** field to provide a unique identifier across this API for the current operation.

Step 6: Add Paths (Part 3: Parameters)

In this part, we add parameters to the path, if needed.



```
    parameters:
      - name: sortBy
        in: query
        description: The sort order
        schema:
          type: string
          default: asc
          example: ../api/courses?sortBy=asc
          enum:
            - asc
            - desc
        required: true
```

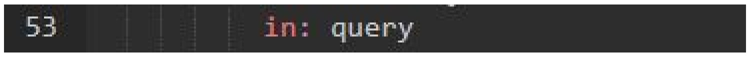
Use the keyword ***parameter*** to define parameters for an operation. A parameter is uniquely determined by its name and location.

The location information is entered using the ‘***in’*** field. The possible locations are:

- ***path*** - Parameter value is part of the operation's URL. Eg, in /items/{itemId}, the path parameter is itemId.
- ***query*** - Parameters that are appended to the URL.
  Eg. /colleges?sortOrder=xxx
  The parameter sortOrder is located in the query parameter.
- ***header*** - They are part of the header of the request.
- ***cookie*** - Part of the cookie of the API.

Fields: name, in, description, required, deprecated, schema/content, example, etc.

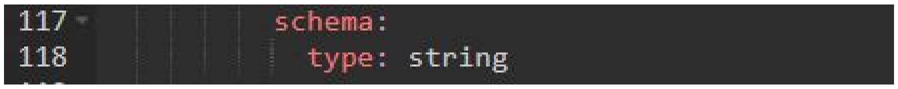
- **name** - The name of the parameter. It is a **MANDATORY** field.
- **in** - The location of the parameter. Possible values are *query, header, path,* and *cookie*. It is a **MANDATORY** field.



```
        in: query
```

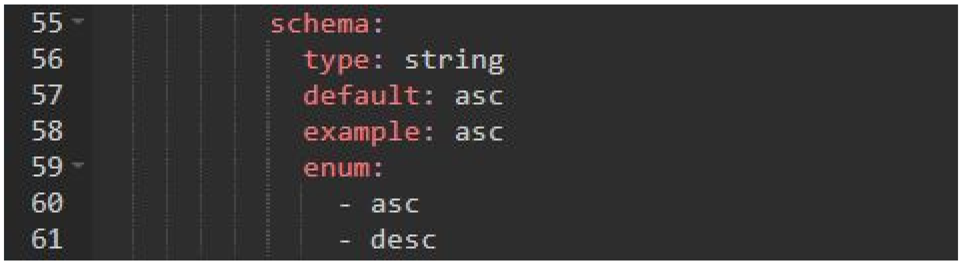
- **description** - Optional description for this parameter.
- **required** - Used to mention whether this parameter is mandatory. If the parameter location is "path", this property is **MANDATORY** and its value **MUST** be *true*.
- **deprecated** - Used to inform the clients no avoid using this parameter.

- **schema/content** - Used to define the type of the parameter. It is a **MANDATORY** field.



```
schema:
  type: string
```

Also,

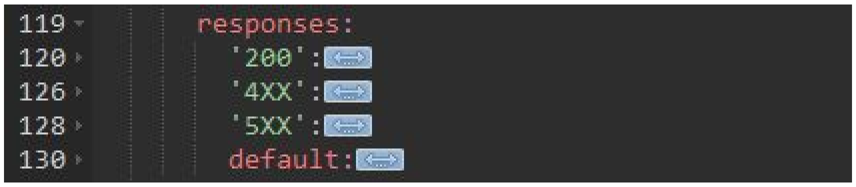


```
schema:
  type: string
  default: asc
  example: asc
  enum:
    - asc
    - desc
```

- **example** - Provide an example value for the parameter.
- **examples** - Provide one or more examples for the parameter.

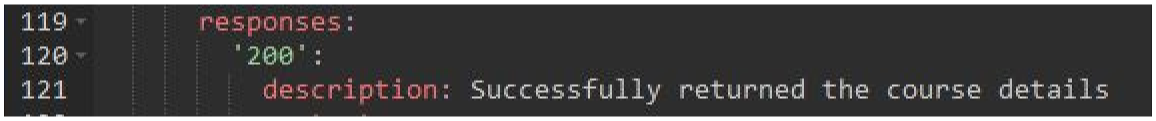
Step 6: Add Paths (Part 4: Responses)

In this part, you add responses to the operation of a path.



```
    responses:
      '200':
      '4XX':
      '5XX':
      default:
```

Use the keyword ***responses*** to define responses for the operation. It contains all the responses expected from an operation. You define one response for an HTTP Status Code.

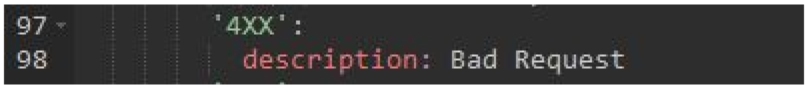


```
    responses:
      '200':
        description: Successfully returned the course details
```

An operation **MUST** have at least one response defined.

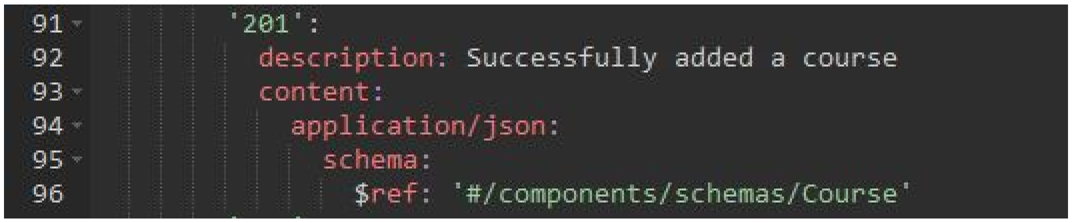
Fields: HTTP Response Code, default

- **default**: Used to refer 'all' HTTP codes that aren't captured in the specification. For any HTTP status code that isn't defined in the specification, the default response body will be considered.
- **HTTP Response Code:** Define the response for one or more HTTP status codes.
    - You can use X to represent a range of response codes. Note that it is a capital X character. Eg, 4XX represents all response codes between 400 and 499.



```
      '4XX':
        description: Bad Request
```

    - Each response contains a **description** and **content** fields.

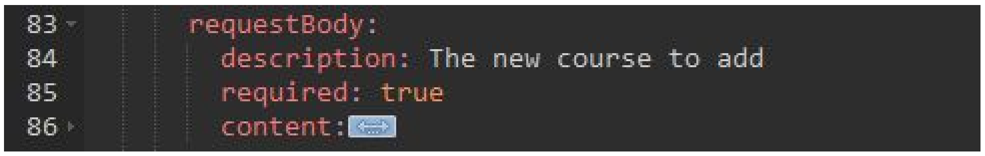


```
      '201':
        description: Successfully added a course
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Course'
```

Step 6: Add Paths (Part 5: Request Body)

In this part, you add a request body for the operations that require some input values to be passed.

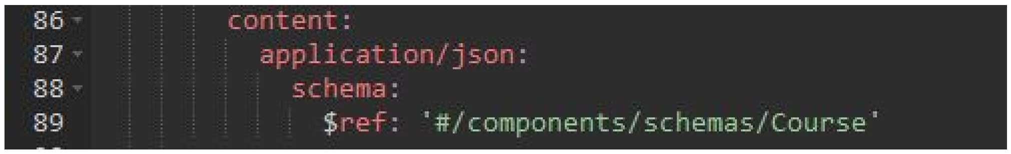
Use the keyword ***requestBody*** to define requests for an operation.



```
      requestBody:
        description: The new course to add
        required: true
        content:
```

Fields: description, required, content

- **description** - Optional description for this parameter.
- **required** - Used to mention whether this parameter is mandatory.
- **content** - The content of the request body. It is a **MANDATORY** field.



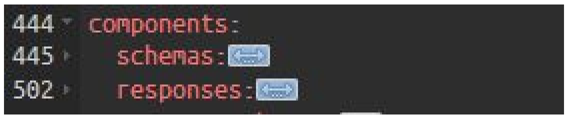
```
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Course'
```

Step 7: Add Components (Part 1: Components Object)

In the programming world, as a best practice, it is always recommended to define once and use it in many places. This way you need to make the changes only in a single place. You will have better control.

While writing OpenAPI Specification, you can follow the same principle - **define the types once and reference it in multiple places**. Components are used to achieve this in OpenAPI.

You use the keyword ***components*** to define reusable types within an OpenAPI Specification. It can be used to include schemas, responses, parameters, examples, request bodies, headers, etc.



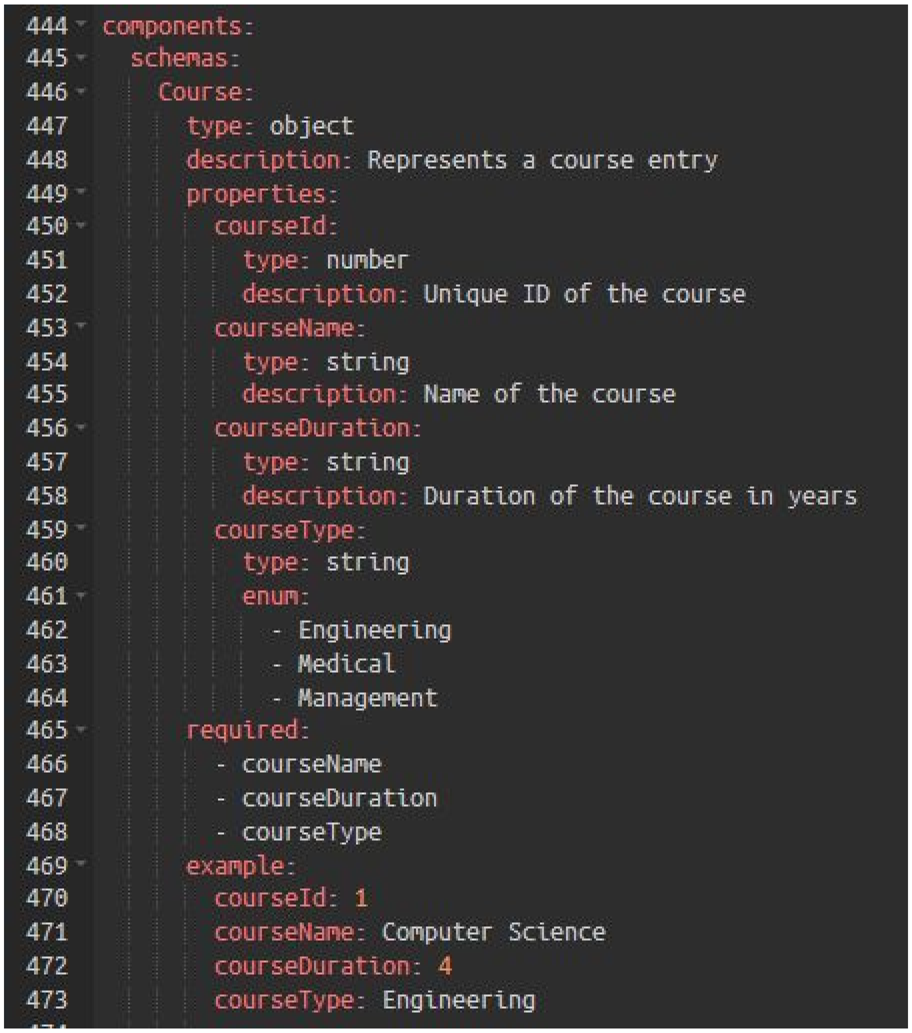
```
components:
  schemas:
  responses:
```

Fields: schemas, responses, parameters, examples, requestBodies, headers, securitySchemes, links, callbacks

Note that the objects defined within the components object will not affect **the API unless they are explicitly referenced** from other places in the API documentation.

Step 7: Add Components (Part 2: Schema)

In this part, you add schema to the components section so that the types defined can be reused throughout the API.

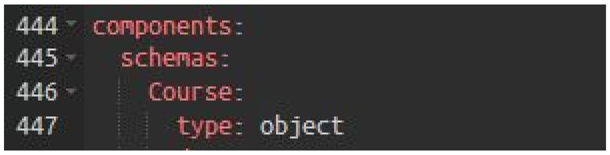


```yaml
components:
  schemas:
    Course:
      type: object
      description: Represents a course entry
      properties:
        courseId:
          type: number
          description: Unique ID of the course
        courseName:
          type: string
          description: Name of the course
        courseDuration:
          type: string
          description: Duration of the course in years
        courseType:
          type: string
          enum:
            - Engineering
            - Medical
            - Management
      required:
        - courseName
        - courseDuration
        - courseType
      example:
        courseId: 1
        courseName: Computer Science
        courseDuration: 4
        courseType: Engineering
```

First, you need to give a name for the type. Next, you define the type using various fields.

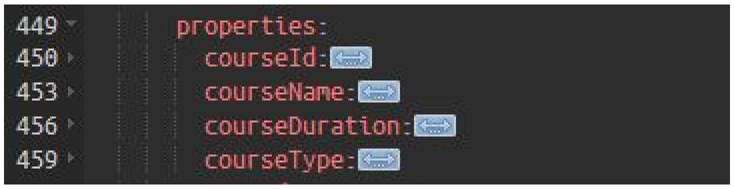
Fields: type, format, minimum, maximum, required, enum, default, description, example

- **type** - The type of the schema that you are defining. If it's a user-defined type, then use '*object'.*



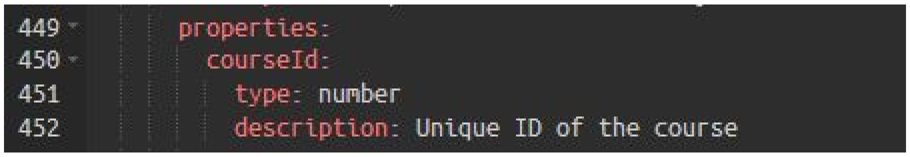
```yaml
components:
  schemas:
    Course:
      type: object
```

- **format** - The format used for the above type (if applicable).
- **description** - Description of this type.
- **properties** - One or more fields to define the object schema type.



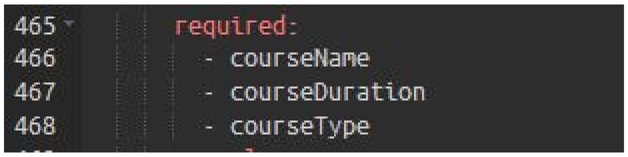
```
properties:
  courseId:
  courseName:
  courseDuration:
  courseType:
```

Note that each of the property value itself is another schema type!



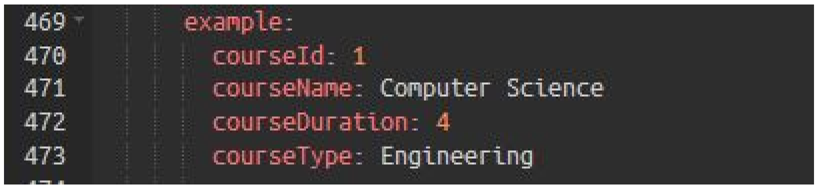
```
properties:
  courseId:
    type: number
    description: Unique ID of the course
```

- **minimum** - For applicable types, mention the minimum value it can take.
- **maximum** - For applicable types, mention the maximum value to allow.
- **required** - Mark the list of mandatory properties.



```
required:
  - courseName
  - courseDuration
  - courseType
```

- **example** - Provide example values for the type that is being defined.



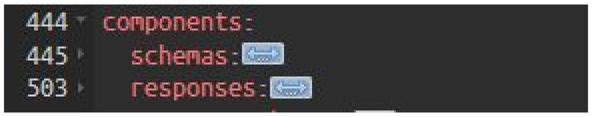
```
example:
  courseId: 1
  courseName: Computer Science
  courseDuration: 4
  courseType: Engineering
```

Step 7: Add Components (Part 3: Response)

In this part, you add the response body to the components section so that it can be reused throughout the API.

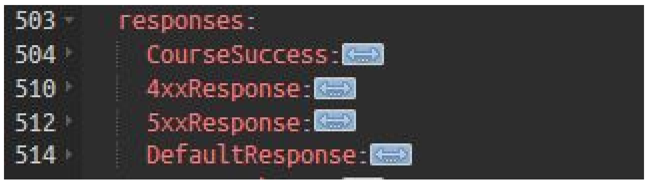
Similar to schemas, you can define responses that can be defined once and used multiple times.

Use the keyword **responses** to create reusable types under the **components** section.



```
components:
  schemas:
  responses:
```

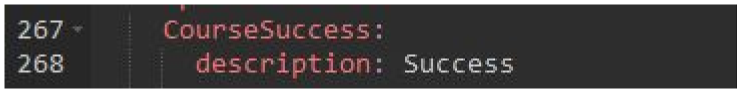
You can add any number of responses. For each response body, you provide a name followed by a set of field values.



```
  responses:
    CourseSuccess:
    4xxResponse:
    5xxResponse:
    DefaultResponse:
```

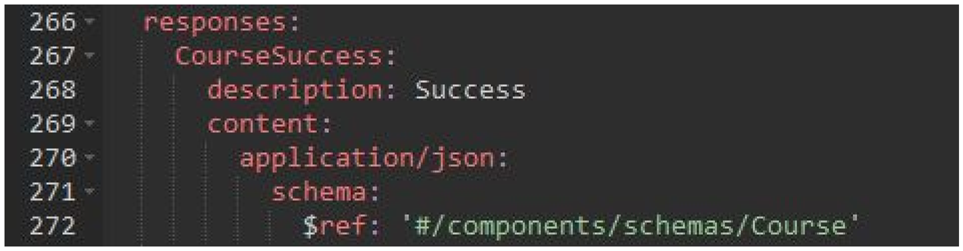
Fields: description, content

- **description** - A description for the response body. For simple responses, you can include only the description.



```
    CourseSuccess:
      description: Success
```

- **content** - The content for the response body. It follows the structure for defining a content body.



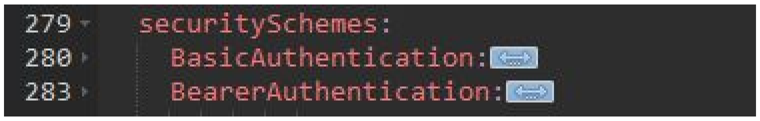
```
  responses:
    CourseSuccess:
      description: Success
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Course'
```

Note that the response body can refer to any types defined within the component section such as schemas, etc. In the above example, the Course type defined in the **schemas** section is referenced in the **responses** section.

Step 8: Add Security

In this step, you add the security mechanism to the API documentation.

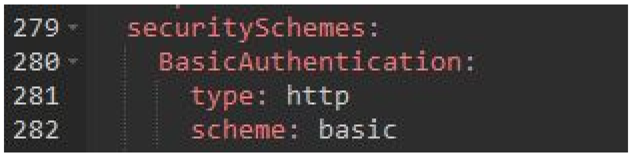
Use the keyword ***securitySchemes*** to define one or more security schemes for your API specification. It should be located under the ***components*** section.



```
  securitySchemes:
    BasicAuthentication:
    BearerAuthentication:
```

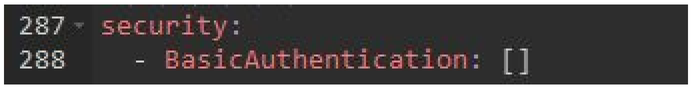
Fields: description, type, scheme

- **description** - Optional description for this parameter.
- **type** - The type of security scheme. OpenAPI Specification supported security schemes:
  - HTTP
  - API Key
  - OAuth 2
  - OpenID
- **scheme** - The individual scheme for the above ***type***.
  - Basic, Bearer, etc. for HTTP type.



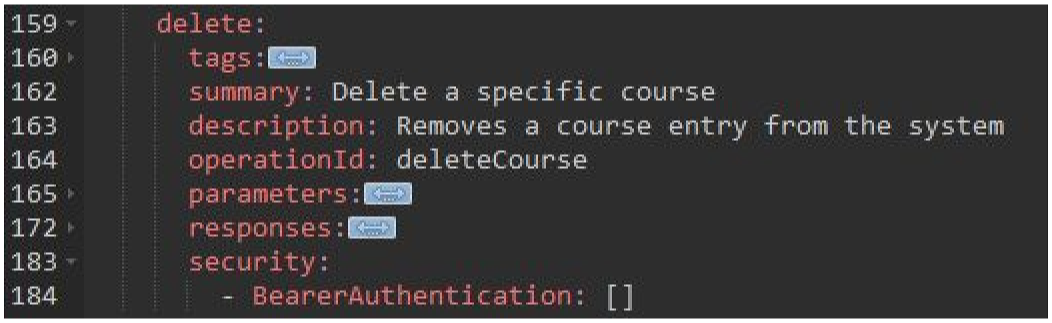
```
  securitySchemes:
    BasicAuthentication:
      type: http
      scheme: basic
```

The schemes defined can then be referred throughout the API document such as in the security section and operation section.



```
security:
  - BasicAuthentication: []
```

If required, individual operations can override this global definition by including them under their operation definition.



```
    delete:
      tags:
      summary: Delete a specific course
      description: Removes a course entry from the system
      operationId: deleteCourse
      parameters:
      responses:
      security:
        - BearerAuthentication: []
```

In the above example, irrespective of the global security scheme mentioned, the delete operation for the individual Course path will use Bearer authentication.

Thank you!

I hope this resource was helpful to you.