Postman

API Documentation:

API documentation serves as a comprehensive guide that outlines the functionality, endpoints, parameters, response formats, and usage instructions for an Application Programming Interface (API). It acts as a reference manual for developers, enabling them to understand how to interact with the API effectively.

In Postman, you can generate API documentation directly from your collections. Below is a step-by-step guide detailing how to generate documentation from Postman for a collection:

1. **Import or Create Collection:** If you haven't already created a collection for your API requests, you can either import an existing collection or create a new one. Collections serve as containers for organizing your API requests and documentation.
2. **Add Requests to Collection:** Populate your collection with API requests by either creating new requests or importing them from existing sources. Ensure that each request is appropriately configured with the necessary parameters, headers, and body.
3. **Add Descriptions and Metadata:** For each request in your collection, add descriptive information such as a summary, description, parameters, and example responses. This metadata helps provide context and guidance to users consuming your API documentation.
4. **Generate Documentation:** Once your collection is complete and thoroughly documented, navigate to the collection and click on “View Documentation”. From the top right corner of your collection click on "Publish"
5. **Customize Documentation Settings:** In the documentation settings, you can customize the appearance and behavior of your API documentation. You can

choose the layout, theme, and authentication settings based on your preferences and requirements.

6. **Publish Documentation:** After customizing the settings, click on the "Publish" button to generate and publish your API documentation. Postman will provide you with a public URL where users can access your documentation.
7. **Share Documentation:** Share the generated documentation URL with your team members, stakeholders, or external users who need to consume or interact with your API. They can access the documentation online and refer to it for information on how to use the API endpoints.

There are alternative methods to generate API documentation, such as the "test-first-approve" approach. In this method, you begin by writing specifications for your API. In Postman, you can create an OpenAPI Specification for your API. From this specification, you can easily generate a collection. With the collection in hand, you can generate API documentation and proceed to publish it, as previously discussed.

Now, let's delve into what the OpenAPI specification entails.

OpenAPI Specification

OpenAPI Specification plays a vital role in the API development lifecycle, from design and documentation to implementation, testing, and maintenance. It promotes interoperability, collaboration, and efficiency in API development efforts.
The OpenAPI Specification (OAS) is crucial for the development of APIs (Application Programming Interfaces) for several reasons:

1. **Standardization**: OAS provides a standardized way to describe RESTful APIs. It offers a common language for developers, API providers, and consumers, making it easier to understand, collaborate, and integrate APIs across different platforms and programming languages.
2. **Documentation**: OAS serves as a comprehensive documentation for APIs. It describes endpoints, operations, parameters, request/response payloads, authentication methods, and more in a structured format. This documentation improves API discoverability and helps developers understand how to interact with the API effectively.

3. **Server Code Generation**: Similarly, OAS can be used to generate server-side code. By providing a detailed specification of the API, developers can automatically generate server implementations, reducing the manual effort required for coding.
4. **Testing and Validation**: OAS definitions can be used to validate API requests and responses. Testing tools can compare actual API behavior against the documented specification, ensuring that the API behaves as expected and adheres to the defined contract.
5. **API Governance**: OAS facilitates API governance by providing a clear contract between API providers and consumers. It allows organizations to enforce standards, monitor API usage, and ensure consistency across different APIs within the organization.
6. **Tooling Ecosystem**: OAS has a vibrant ecosystem of tools and utilities built around it. There are editors for authoring OAS documents, validators for checking compliance with the specification, code generators for client/server implementations, testing frameworks, and more.

So now let’s see how we can write OpenAPI Specification on Postman. Below is a step-by-step guide to writing OpenAPI Specification for your APIs

1. Navigate to APIs from the left sidebar.
2. Create a new API by clicking on the Plus (+) icon.
3. Under the Definition section, click on the Plus (+) icon to open a new section. From there, you can either "import a file" or select "Author from Scratch".
4. Choose a definition type and format. For example, select OpenAPI 3.0 as the definition type and YAML as the definition format.
5. Click on "Create Definition".
6. You can now begin writing your API definition using the OpenAPI specification.

Now, let's proceed to write OpenAPI specifications for our Auth and Inventory APIs.

Auth APIs Specification

JavaScript
```
openapi: 3.0.0
info:
  title: Authentication Service
  version: 1.0.0
servers:
  - url: http://localhost:4005

tags:
  - name: Signup
  - name: Signin
  - name: Checkpoint

paths:
  /api/v1/signup:
    post:
      summary: User Signup
      tags:
        - Signup
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                name:
                  type: string
                email:
                  type: string
                  format: email
                password:
                  type: string
                  minLength: 8
              required:
                - name
                - email
                - password
      responses:
        201:
          description: User Created successfully
          content:
```

```
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: User Created successfully
                links:
                  type: object
                  properties:
                    signIn:
                      type: string
                      example: "${BASE_URL}/api/v1/signin"

      400:
        description: Bad requestBody
        content:
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: Name, Email and Password is required!
      409:
        description: Conflict
        content:
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: Email already exists!
      500:
        description: Internal Server Error
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Internal_Server_Error"

  /api/v1/signin:
    post:
      summary: User signIn
```

```yaml
tags:
  - Signin
requestBody:
  required: true
  content:
    application/json:
      schema:
        type: object
        properties:
          email:
            type: string
            format: email
          password:
            type: string
            minLength: 8
        required:
          - email
          - password

responses:
  200:
    description: OK
    content:
      application/json:
        schema:
          type: object
          properties:
            token:
              type: string
              example: "JWT_TOKEN"
  401:
    description: Unauthorized
    content:
      application/json:
        schema:
          type: object
          properties:
            message:
              type: string
              example: "Password is incorrect!"
  404:
    description: User Not Found
    content:
      application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: "User not found!"
        500:
          description: Internal Server Error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Internal_Server_Error"

  /api/v1/checkpoint:
     post:
      summary: Checkpoint for authenticated access
      security:
        - bearerAuth: []
      tags:
        - Checkpoint
      responses:
        200:
          description: Access Granted
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
                    example: "Access Granted"
        401:
          description: Unauthorized
          content:
            application/json:
              schema:
                type: object
                properties:
                  mesage:
                    type: string
                    example: "Access denied. No token provided."
        500:
          description: Internal Server Error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Internal_Server_Error"




components:
  schemas:
    Internal_Server_Error:
      type: object
      properties:
        message:
          type: string
          example: "Internal Server Error"
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

Now let’s create API specifications for Inventory APIs.

Inventory APIs

JavaScript
```
openapi: 3.0.0
info:
  title: Inventory
  description: |
    This is the Product Inventory Service
  version: 1.0.1

servers:
  - url: http://localhost:4001

paths:
  /api/v1/health:
    get:
      summary: Check server health
      responses:
        200:
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
                    example: Your server is healty
        500:
          description: Internal Server Error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Internal_Server_Error"



  /api/v1/products:
    post:
      summary: Create New Product
      requestBody:
        required: true
```

```yaml
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Create_Product"
      responses:
        201:
          description: Created
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Product"
        500:
          description: Internal Server Error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Internal_Server_Error"
    get:
      summary: Retrive list of products
      responses:
        200:
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
                  products:
                    type: array
                    items:
                      $ref: "#/components/schemas/Product"

        500:
          description: Internal Server Error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Internal_Server_Error"


  /api/v1/products/{productId}:
    get:
```

```
    summary: Retrive a product by id
    parameters:
      - in: path
        name: productId
        required: true
        schema:
          type: string
          example: "78d17df3-ce6b-4be6-b029-76639ae086e1"
        description: The Id of the product to retrive.
    responses:
      200:
        description: OK
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Product"

      404:
        description: Not Found
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Not_Found"
      500:
        description: Internal Server Error
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Internal_Server_Error"

  put:
    summary: Update a product by ID
    parameters:
      - in: path
        name: productId
        required: true
        schema:
          type: string
          example: "78d17df3-ce6b-4be6-b029-76639ae086e1"
        description: The ID of the product to update
    requestBody:
      required: true
      content:
        application/json:
```

```
        schema:
          type: object
          properties:
            name:
              type: string
              example: Gloves
            price:
              type: number
              example: 26.8
            quantity:
              type: integer
              example: 40
    responses:
      201:
        description: Update successfully
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Product"
      404:
        description: Not Found
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Not_Found"
      500:
        description: Internal Server Error
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Internal_Server_Error"

  delete:
    summary: Delete product by ID
    parameters:
      - in: path
       name: productId
       required: true
       schema:
        type: string
        example: "78d17df3-ce6b-4be6-b029-76639ae086e1"
       description: ID for delete product
    responses:
      200:
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
                    example: Success
        404:
          description: Not Found
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Not_Found"
        500:
          description: Internal Server Error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Internal_Server_Error"





components:
  schemas:
    Internal_Server_Error:
      type: object
      properties:
        message:
          type: string
          example: Internal Server Error


    Not_Found:
      type: object
      properties:
        message:
          type: string
          example: Product Not Found
    Product:
```

```
    type: object
    properties:
      id:
        type: string
        example: "78d17df3-ce6b-4be6-b029-76639ae086e1"
      name:
        type: string
        example: "Sausages"
      quantity:
        type: integer
        example: 162
      price:
        type: number
        example: 166.18
      createdBy:
        type: string
        example: "59edaab2-541d-4897-84fe-3dffe406dbe2"
      createdAt:
        type: string
        format: date-time
        example: "2024-01-24T16:06:39.193Z"
      updatedAt:
        type: string
        format: date-time
        example: "2024-01-24T16:06:39.193Z"
  Create_Product:
    type: object
    properties:
      name:
        type: string
        example: "Gloves"
      price:
        type: number
        example: 501.34
      quantity:
        type: integer
        example: 21
      userID:
        type: string
        example: "03145490-4614-42cf-ad49-a598da44858a"
```

Now that we have two API specifications, one for Auth and another for Inventory, you can generate collections from these specifications by following the guide below:

1. Click on the ellipsis (...) next to your API Specification Name. From there, select "Add Collection". This action will expand a section. Next, click on "Generate from definition". This will create a collection for you.
2. Once you have the collection, you can easily generate documentation from it. Follow the process mentioned earlier.

By following these steps, you can efficiently generate collections from your API specifications and subsequently create documentation for them.

Authorization Code Flow

Many APIs are secured using various authentication mechanisms, and one common method is the "Authorization Code Flow". Now let's see what is “Authorization Code flow” is and how can we handle this flow using Postman.

The Authorization Code Flow is an OAuth 2.0 authentication flow used by applications to obtain authorization to access resources on behalf of a user. It is commonly used in web applications where the client-side code runs in a web browser.
The Authorization Code Flow provides a secure and efficient way for applications to obtain access to protected resources on behalf of users without exposing sensitive credentials. It ensures that only authorized applications can access user data and that users have control over which resources the application can access.
Here's how the Authorization Code Flow works:

1. **User Initiates Authorization**: The user initiates the authentication process by clicking a login button or accessing a protected resource on the application. The application redirects the user to the authorization server's authorization endpoint.
2. **User Authentication**: At the authorization endpoint, the user is prompted to log in and authenticate themselves. They may be required to enter their username and password or use some other form of authentication, such as social login or multi-factor authentication.
3. **Authorization Request**: After successful authentication, the authorization server prompts the user to grant permission to the application to access their resources.

The user may be presented with a consent screen detailing the scope of access requested by the application.

4. **Authorization Code Generation**: If the user grants permission, the authorization server generates an authorization code and redirects the user back to the application's redirect URI along with the authorization code appended as a query parameter.
5. **Authorization Code Exchange**: The application receives the authorization code and sends a POST request to the authorization server's token endpoint, along with authentication credentials (client ID and client secret), the authorization code, and the redirect URI.
6. **Token Retrieval**: The authorization server validates the authorization code, and client credentials, and redirects URI. If everything is valid, the authorization server responds with an access token and optionally a refresh token. The access token is used by the application to access protected resources on behalf of the user.
7. **Accessing Protected Resources**: The application includes the access token in subsequent requests to the resource server when accessing protected resources. The resource server validates the access token and grants access to the requested resources if the token is valid and authorized.
8. **Token Expiry and Refresh**: Access tokens typically have a limited lifespan. When an access token expires, the application can use the refresh token (if provided) to obtain a new access token without requiring the user to re-authenticate.

To handle the Authorization Code flow effectively in Postman, follow these steps:

1. Click on your collection name to open it.
2. Navigate to the Authorization tab.
3. Select "OAuth 2.0" as the Authorization type.
4. If you have a Callback URL, enter it. Otherwise, check the option "Authorize using browser".
5. Provide the Authorization URL. This is the authentication endpoint of your OAuth service. For example, if you are using Auth0, you can find this URL in your Auth0 dashboard.
6. Enter the Access Token URL. This is the endpoint where your OAuth service provides access tokens. Again, you can obtain this URL from your Auth0 dashboard if you're using Auth0.

7. Now, click on "Get New Access Token".

Note: To mimic the "Authorization Code" Flow, ensure that you have enabled the "Authorization code" grant type on your Auth server. This step is crucial for simulating the Authorization Code Flow correctly in Postman.

Data-Driven Testing:

Data-driven testing in Postman is a technique where you use external data sources, such as CSV files, and JSON files to drive your API tests. Instead of hardcoding test data directly into your requests, you dynamically replace variables in your requests with values from your data source during test execution. This allows you to run the same test scenario with different sets of data, making your tests more comprehensive and reusable.
Here's how data-driven testing works in Postman:

1. **Prepare Your Test Data**: First, you need to prepare your test data in an external data source. This could be a CSV file, JSON file, or any other format that Postman supports.
2. **Define Variables in Postman**: In your Postman requests, you define variables where you want to inject the data from your external source. These variables are enclosed in double curly braces, like `{{variable_name}}`.
3. **Import Data into Postman**: Import your external data source into Postman. Postman allows you to directly import CSV and JSON files into your collection or environment.
4. **Configure Data Source in Postman**: In Postman, you configure your data source (e.g., CSV file) to be used for data-driven testing. You can do this by selecting the appropriate data file in the Collection Runner or using the `pm.iterationData.get()` function in scripts.
5. **Run Your Tests**: When you run your tests using the Collection Runner or Newman (Postman's command-line tool), Postman automatically iterates through each row of your data source, replacing the variables in your requests with the corresponding values from the data source.
6. **Analyze Test Results**: After running your tests, you can analyze the test results to see how your API behaves with different sets of data. Postman provides detailed test result reports, including pass/fail statuses and response data for each iteration.

Data-driven testing in Postman is beneficial for scenarios where you need to test your API with multiple input combinations, such as different parameter values, edge cases, or scenarios specific to different user profiles. It helps in improving test coverage and identifying potential issues early in the development cycle.
Data files are limited to 1 MB in size and a maximum of 50 data rows (CSV) or 50 objects (JSON).

Now let's write a data-driven testing script for Create Product.

POST: {{inventory_base_url}}/api/v1/products

Pre-Script:

Here i retrieve data from the JSON file. You can select the external file when you run the collection from the collection runner

```javascript
const productName=pm.iterationData.get("product_name")
const productPrice=pm.iterationData.get("product_price")
const productQuantity=pm.iterationData.get("product_quantity")
const userID=pm.iterationData.get("userID")

pm.collectionVariables.set("name",productName),
pm.collectionVariables.set("price",productPrice)
pm.collectionVariables.set("quantity",productQuantity)
pm.collectionVariables.set("userid",userID)
```

Here, I've written a test to validate whether the product has been successfully saved or not by examining the response body.

Test-Script

```javascript
const product = pm.response.json().product;
const name = pm.collectionVariables.get("name")
const price = pm.collectionVariables.get("price")
const quantity = pm.collectionVariables.get("quantity")
const userId = pm.collectionVariables.get("userid")

pm.test("is should validate the response body with the collection
varialbes", () => {
  pm.expect(product.name).to.be.equal(name)
  pm.expect(product.price).to.be.equal(price)
  pm.expect(product.quantity).to.be.equal(quantity)
  pm.expect(product.createdBy).to.be.equal(userId)

  pm.collectionVariables.set("productId",product.id)
})
```

Monitoring

In Postman, monitoring refers to the process of continuously monitoring APIs for performance, availability, and functionality. Postman Monitoring allows you to schedule and run API tests at regular intervals from multiple global locations. This helps you ensure that your APIs are functioning as expected and meeting performance requirements. You can enable monitoring for your collection by navigating to Monitors from the left sidebar of your Postman interface.
Key features of monitoring in Postman include:

1. **Scheduled Tests**: You can schedule API tests to run at specific intervals, such as every minute, hourly, daily, etc. This allows you to monitor your APIs continuously and detect any issues or performance degradation over time.
2. **Global Locations**: Postman provides a network of global monitoring locations from which you can run your tests. This allows you to simulate requests from different geographic regions and ensure that your APIs perform well for users worldwide.

3. **Performance Metrics**: Postman Monitoring collects performance metrics such as response time, latency, and uptime for your APIs. You can view these metrics in dashboards and reports to analyze trends and identify areas for improvement.
4. **Alerting**: You can set up alerts based on predefined thresholds for performance metrics. This allows you to receive notifications via email, Slack, or other channels when an API test fails or when performance metrics exceed specified limits.
5. **Custom Environments**: You can define custom environments for your monitoring tests, allowing you to parameterize your requests and configure different settings for different environments (e.g., development, staging, production).

Monitor summary

You can use the **Monitor Summary** to understand how your APIs have performed over time. Each monitor run is represented by a bar in the graph.
The upper section charts your monitor's average response time for each run, while the lower section visualizes the number of failed tests for each run across all regions. To view the exact values for failed percentage and response time, hover over each run individually.

Filtering by formula

You can filter by mathematical formula to view the average, sum, minimum, and maximum response time for each run:

- **Average** - The average of the total response time across all regions.
- **Sum** - The sum of the response time across all regions.
- **Minimum** - The minimum total response time for a run across all regions.
- **Maximum** - The maximum total response time for a run across all regions.

Select **Average** to open the menu, then select an option. To view the calculated response time value, you can hover over each run individually.

Time traverse

You can review past run results to understand what happened at a particular point in time. To do so, select **Go to** in the upper-left corner of the monitor summary or request split graph. Select the time and date, then select **Apply** to view a specific run.

Monitor activity limits

Postman maintains default limits on various team and user actions to ensure the overall performance and availability of monitoring. Postman will email your Team Admins if your team encounters these limits. In addition, team members will get an alert in Postman. Postman maintains the following monitoring limits per team:

- Maximum number of active and paused monitors: 300
- Maximum parallel runs of all monitors: 500
- Maximum parallel runs of a single monitor: 200