SWAGGER

Swagger is a set of easy and convenient tool suite for API development. It helps with development throughout the API Lifecycle, starting from the design phase to the development, testing, and deployment phase.

It comprises multiple tools that are free, open-source, and commercially available. This allows anyone, from developers and engineers to product managers, to build remarkable APIs loved by everyone.

SmartBear Software is the parent company of Swagger, a leader in software quality tools for teams. SmartBear has also developed many notable tool suites in the software industry, like SoapUI and QAComplete.

Design Phase :

Designing an API Is the building block for API development. Swagger, with its easy to use tool suite for developers, engineers, architects, and managers, makes designing an API really easy.

Swagger helps in Modelling APIs with Accuracy since API design can be prone to errors and can be really time-consuming and difficult to identify and resolve the errors made while modeling the API. The swagger editor, which is the first editor made for API designing with the OpenAPI Specification, validates the API design in real-time, provides visual feedback simultaneously, along with checking for OAS compliance.

Swagger helps in Visualizing the API as you design them, as the best designed APIs are built with a consumer-centric approach. It provides tools like Swagger Editor and SwaggerHub containing a YAML editor. The latter comes with a visualization panel for developers to see how API will work for the consumer.

Swagger helps in Standardizing the API Design across Teams by sharing common patterns and behaviors of the API and providing a consistent RESTful interface to ease the work for designers who are building the APIs, and the API consumers. SwaggerHub comes with a built-in API Standardization tool which makes the APIs compliant with the design guidelines of the organization.

Overview and Features

Swagger Hub acts as a dedicated platform for API Design & Documentation with OpenAPI Specification. It comes with the following capabilities:

SwaggerHub contains a YAML editor that comes with a visualization panel for developers to see how API will work for the consumer.

SwaggerHub comes with a built-in API Standardization tool which makes the APIs compliant with the design guidelines of an organization

SwaggerHub has a built-in code generator to provide a self-service mechanism for consumers to download the SDKs

SwaggerHub enables Virtualization of the base operations for an API by providing a built-in mocking capability to interact with the internal and external service dependencies before the actual service is built.

SwaggerHub's versioning system helps in maintaining multiple documentation versions for an API along with incremental development on top of existing documentation which helps in the easy discovery of all available versions of the API by internal and external consumers along with the information on how to consume the API

SwaggerHub, with its built-in API Standardization feature, enables the API to be in compliance with the design guidelines of an organization.

SwaggerHub allows the Management of Teams and projects centrally enabling effective people management for an API.

SwaggerHub provides the functionality for reviewing and merging changes along with communication and tracking of issues, making the job of a developer really easy.

SwaggerHub also provides Domains, which are dedicated stores for storing the models in an API's design, like resources, operations, and data models.

An API is nothing but a description of how certain things should be when the API starts functioning. This description is written in a structured format, called OpenAPI specification. The predefined structure allows all the APIs to conform to the same style and makes their implementation and perusal easier. In general, every API specification file contains the following pieces of information:

Metadata – General information about the API, like its name, version, specification format, etc.

Resources – Since REST APIs work on resources, we can define those resources in the specification file in as much detail as required

Methods – We can also define which methods do we need to be applicable to the resources

Input and Output data – We can define the structure and examples of input and output data formats for the benefit of the developers who would like to see the same before using the API

The OpenAPI Initiative team has made an easy to understand documentation of the OAS (Open API Specification). You can find it here: link.

You need not go through the documentation right away in order to understand the rest of this course. We will cover all the concepts in a sequential manner. You can refer to the documentation for more detailed information on the concept discussed if you wish to.

YAML vs JSON

There are two major formats in which you can write an OAS file. These are:

JSON – Javascript Object Notation

YAML – Yet Another Markup Language

We prefer writing the file in YAML as it is easier to structure and less rigid in terms of format regulations.

YAML is easier to read for us humans and occupies a lesser number of lines. In any case, you can always convert a YAML file to JSON and a JSON file to YAML if need be.

A useful feature that speeds up API writing using OAS is the OAS Editor. Though you may write an API specification using a puny text editor, the feature-loaded OAS editor is recommended for faster development. The Swagger Editor gives you the following features:

It contains a set of static files that help you with auto-completing tags and properties

It helps you validate the structure of the file in real-time

You can see the documentation being trotted out at a real-time pace at the same screen

Basic Structure of the API

We can learn only to a limited extent without practicing the theory. In this section, we will see how to write a basic API specification from scratch.

In this example, we will work on the following topics:

Add basic information about the API

Endpoint for HTTP methods

Status and response for HTTP requests

CREATE YOUR FIRST API :

Go to www.swagger.io/tools/swaggerhub/

An account (free or paid) is needed to create and store APIs on the Swagger platform.

Click on "Sign Up" and follow the due process by filling in the required details.

Create an API with the following specifications:

OpenAPI Version - 2.0

Template - None

Name - MovieStore

Version - 1.0

Title - Movie Store API

Description - This API will provide information regarding the titles catalog in the system.

Owner - *your registered username*

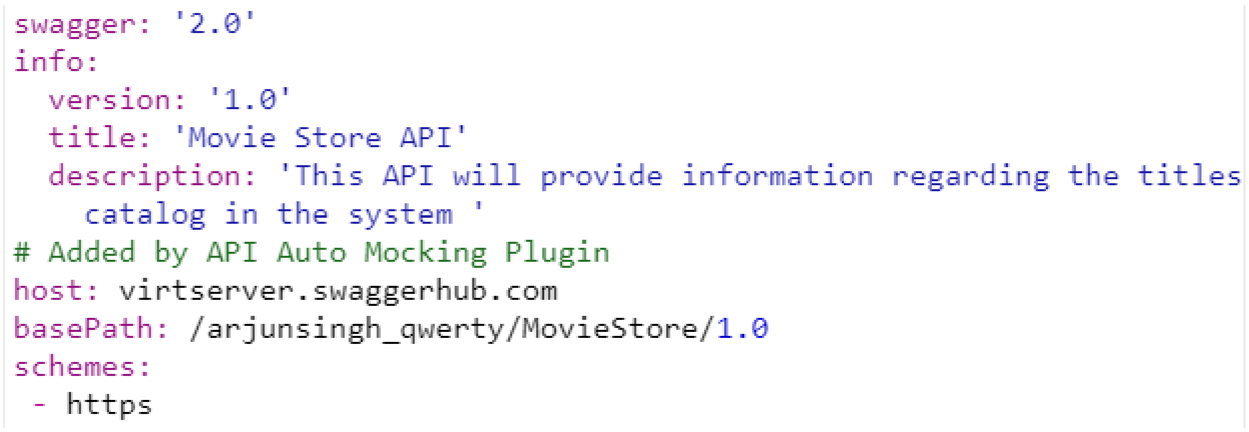
Project - None

Visibility - Public

Auto Mock API – ON

BASIC API SPEC :

Let’s begin by creating a very modest API specification. We have not specified anything here, except some basic information about the API like its name, version, etc.



```yaml
swagger: '2.0'
info:
  version: '1.0'
  title: 'Movie Store API'
  description: 'This API will provide information regarding the titles
    catalog in the system '
# Added by API Auto Mocking Plugin
host: virtserver.swaggerhub.com
basePath: /arjunsingh_qwerty/MovieStore/1.0
schemes:
 - https
```

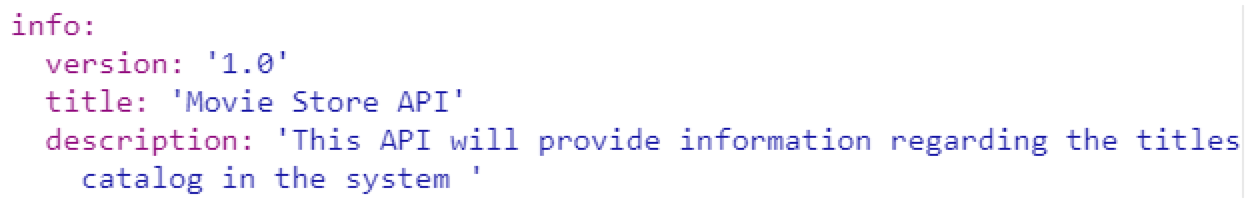
Version

OAS framework is a living document. That means that it is continuously updated with new features. Hence, we need to mention the Swagger version in the first line of the document. Currently, version 3.0 has been released but is in the initial stages of adoption. In this course we are focusing on the more prevalent version 2.0

SWAGGER 2.0

Description

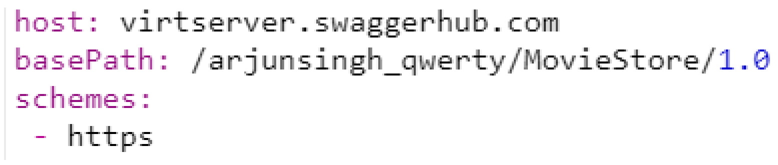
Now, we define some basic information about the API itself



```
info:
  version: '1.0'
  title: 'Movie Store API'
  description: 'This API will provide information regarding the titles
    catalog in the system '
```

URL

Next, we go for defining a URL that the consumers can access to get to the API



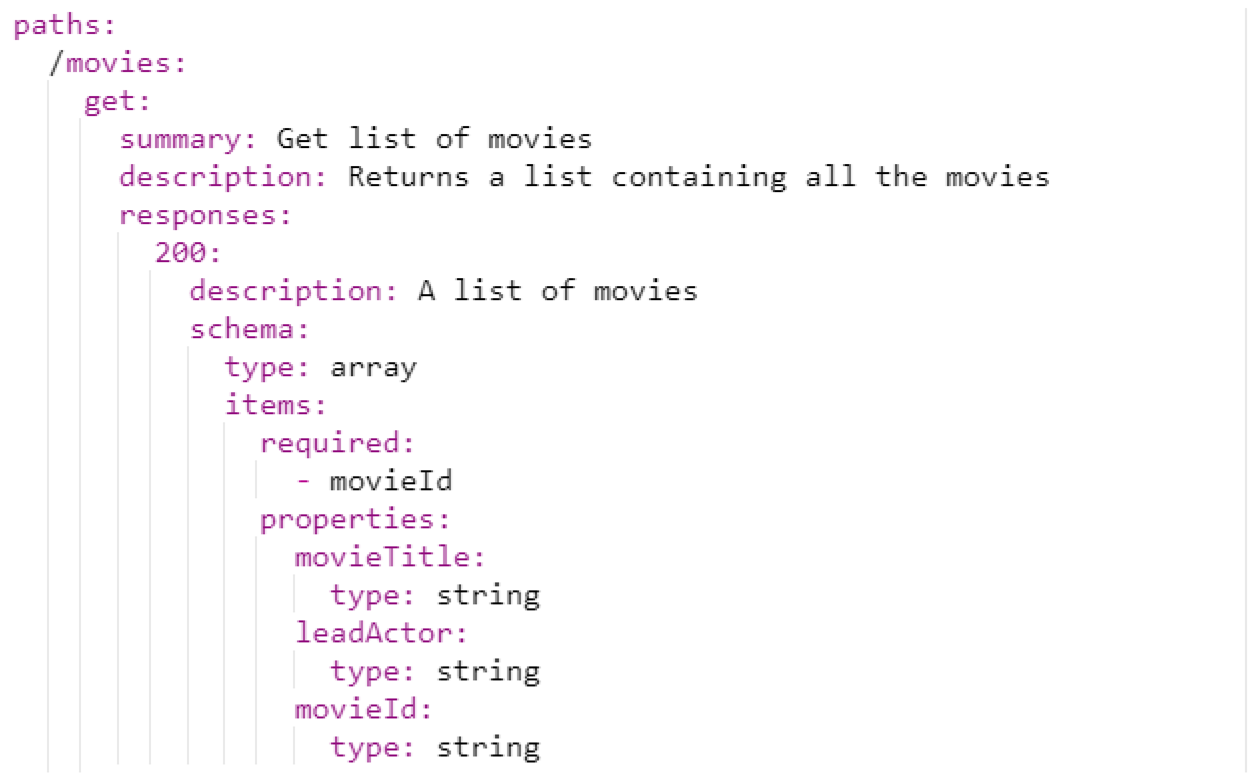
```
host: virtserver.swaggerhub.com
basePath: /arjunsingh_qwerty/MovieStore/1.0
schemes:
 - https
```

Operations

Like mentioned before, this API currently does nothing apart from describing the basic information about the API, hence we'll insert an empty set for the resources provided in the API

Let’s define an operation in this API :

Let us go ahead and make our API a tiny bit useful by adding an actual resource to it as follows:

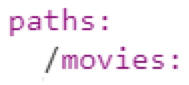


```yaml
paths:
  /movies:
    get:
      summary: Get list of movies
      description: Returns a list containing all the movies
      responses:
        200:
          description: A list of movies
          schema:
            type: array
            items:
              required:
                - movieId
              properties:
                movieTitle:
                  type: string
                leadActor:
                  type: string
                movieId:
                  type: string
```

We will break this down to understand it better:

Path

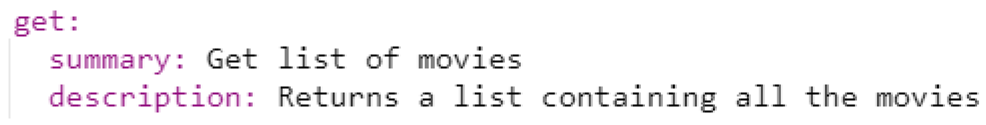
As we know that every API should consist of resources confining the RESTful standards (Refer to REST API Design Best Practices course for more information). Hence, we will add a resource here that would give us the list of movies available in the database.



```yaml
paths:
  /movies:
```

HTTP method on the path

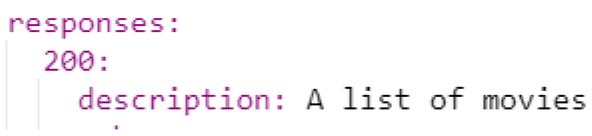
Each resource comes with its own set of verbs to perform operations on it. Here we define a GET method on the resource with the following information.



```yaml
get:
  summary: Get list of movies
  description: Returns a list containing all the movies
```

Responses

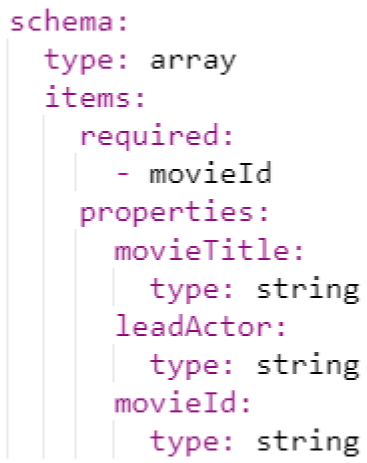
Apart from describing what the consumer needs to send to the API, we can also define what the consumer can expect from the API using status response section.



```
responses:
  200:
    description: A list of movies
```

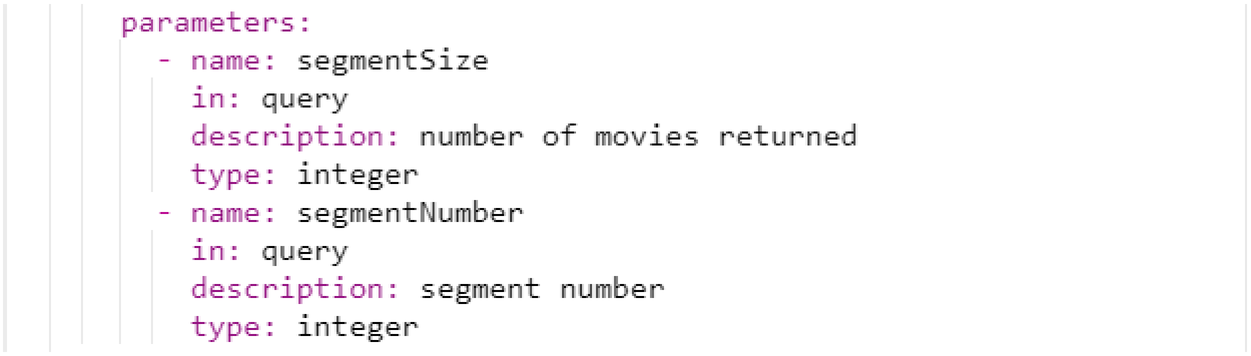
Response Content

Every HTTP response is made up of 2 components, status code and body. Here we are defining a schema for the type of data we expect from the API. We need to define a schema so that all the data present in the database conforms to a structure.



```
schema:
  type: array
  items:
    required:
      - movieId
    properties:
      movieTitle:
        type: string
      leadActor:
        type: string
      movieId:
        type: string
```

As our database increases in size, so does it increase the complexity in browsing through all the added records. Hence, it would be a good idea to add the facility to paginate our returned resources as follows:

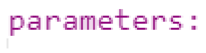


```
parameters:
  - name: segmentSize
    in: query
    description: number of movies returned
    type: integer
  - name: segmentNumber
    in: query
    description: segment number
    type: integer
```

Let us now break it down into simpler steps to understand the logic behind adding the new tags

Parameters

First up we define a parameters tag in the document. This tag will contain all the parameters required for that operation on that resource.

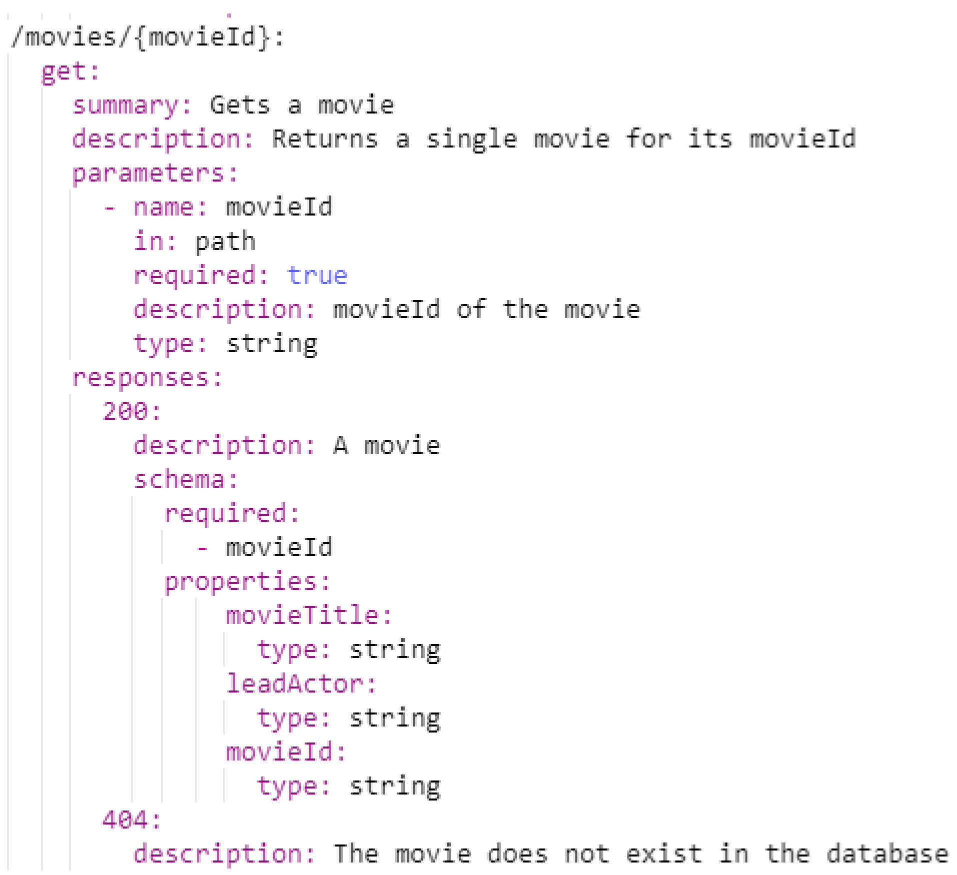


```
parameters:
```

Query Parameters

In the parameters tag, we can define the details of all the parameters we intend to add and where the parameters should go in the request structure.

Apart from obtaining a complete list of the titles present in the database, we might also want to extract only a singly movie too. For that purpose, we need a way to send the unique ID of that movie, like movieId, to the server. We can do that by placing that parameter in the query, or body, or path. It is preferred to use it in the path in order to maintain the hierarchy of the resources. This process is called defining a path parameter. We do this by adding another path to the API as follows:



```
/movies/{movieId}:
  get:
    summary: Gets a movie
    description: Returns a single movie for its movieId
    parameters:
      - name: movieId
        in: path
        required: true
        description: movieId of the movie
        type: string
    responses:
      200:
        description: A movie
        schema:
          required:
            - movieId
          properties:
            movieTitle:
              type: string
            leadActor:
              type: string
            movieId:
              type: string
      404:
        description: The movie does not exist in the database
```

Let us break it down into smaller steps

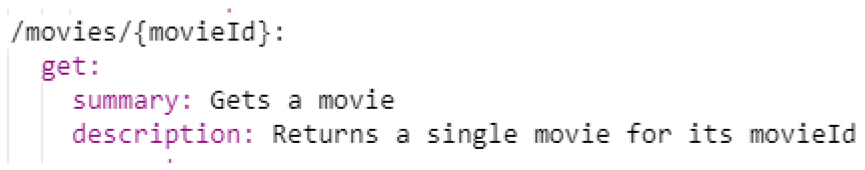
GET Method

First up, we add a separate path to our list of paths

Path Parameter

Now, in order to bring consistency in the path parameter data type, we need to define the properties of the same.

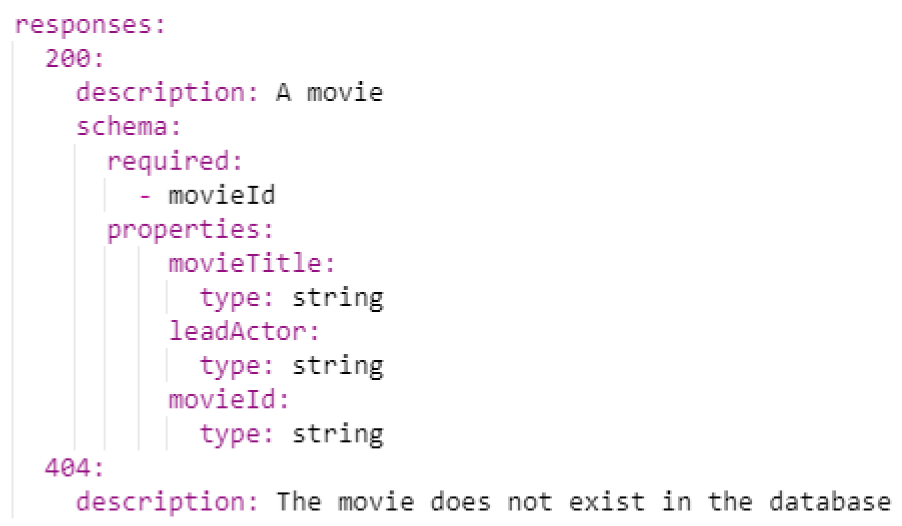
One thing to note here is that Swagger Editor by default does not provide the “required” parameter as true. In order to make this path parameter work, we need to define it as true ourselves. Otherwise, an empty string can be sent to the server for query, producing unintended results.



```
/movies/{movieId}:
  get:
    summary: Gets a movie
    description: Returns a single movie for its movieId
```

Responses

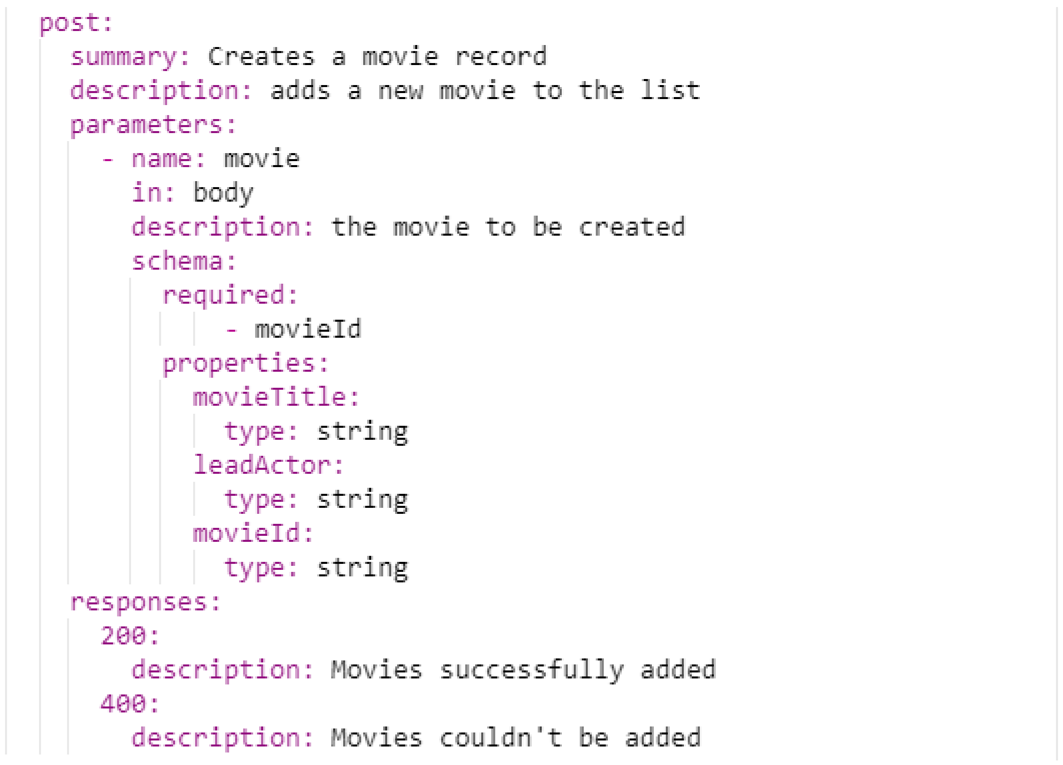
Lastly, we will add a response schema. Remember, in the previous example of accessing a list of movies, we used an array with a schema type. In this case, since we are accessing a single movie, we will add a single movie schema type along with the response for an error.



```yaml
responses:
  200:
    description: A movie
    schema:
      required:
        - movieId
      properties:
          movieTitle:
            type: string
          leadActor:
            type: string
          movieId:
            type: string
  404:
    description: The movie does not exist in the database
```

Let’s define a body parameter

After having a couple of methods to obtain information from the server, we will not add a method to send information to the server. In this case, we will add a method that would send the information of a person object to the server as follows:

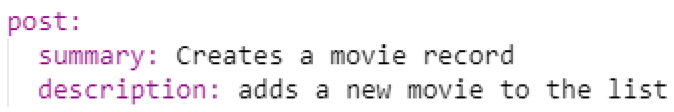


```
post:
  summary: Creates a movie record
  description: adds a new movie to the list
  parameters:
    - name: movie
      in: body
      description: the movie to be created
      schema:
        required:
            - movieId
        properties:
          movieTitle:
            type: string
          leadActor:
            type: string
          movieId:
            type: string
  responses:
    200:
      description: Movies successfully added
    400:
      description: Movies couldn't be added
```

Let us break this down further

POST Method

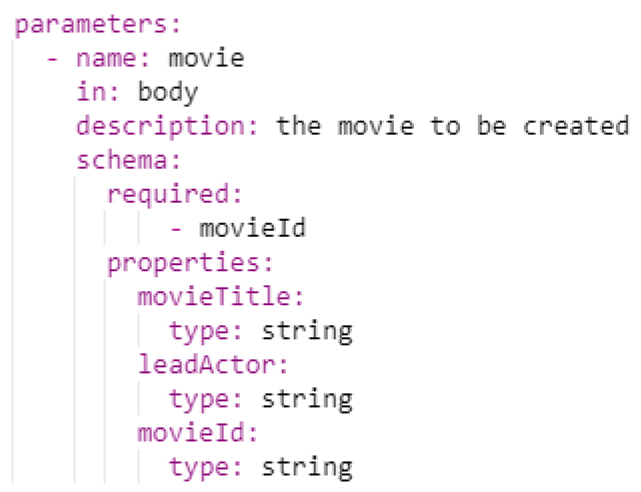
The first step is to add the method in the list of paths. We have already defined the “movies” path in the list with a GET operation. Now we will add a POST operation to the same and proceed with it



```
post:
  summary: Creates a movie record
  description: adds a new movie to the list
```

Body Parameter

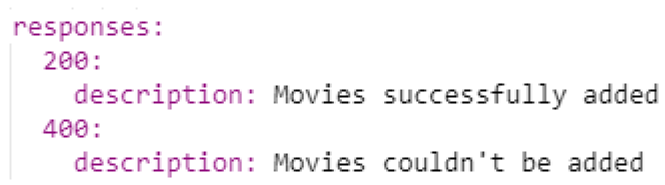
Now, as we did while obtaining information from the server, we need to add the schema for the type of object or data we need to persist in the database. The type of schema followed is the same as the response schema type in the previous example



```yaml
parameters:
  - name: movie
    in: body
    description: the movie to be created
    schema:
      required:
        - movieId
      properties:
        movieTitle:
          type: string
        leadActor:
          type: string
        movieId:
          type: string
```

Responses

Of course, we also need to define a separate type of response compared to the previous example for this operation.


```
responses:
  200:
    description: Movies successfully added
  400:
    description: Movies couldn't be added
```

Now that we have completed a sizeable API with a few methods, parameters, and responses, we notice that the size of the API has increased considerably. The more complex the API becomes from this point, the more difficult it will become to manage the size of it. Hence, we need to find a way to restructure and reuse the components to fit the API into a smaller size. We'll see how to do that in the next section.

Resources :

```
swagger: '2.0'
info:
  version: '1.0'
  title: 'Movie Store API'
  description: 'This API will provide information regarding the titles
catalog in the system '
# Added by API Auto Mocking Plugin
host: virtserver.swaggerhub.com
basePath: /arjunsingh_qwerty/MovieStore/1.0
```

```
schemes:
 - https
paths:
  /movies:
    get:
      summary: Get list of movies
      description: Returns a list containing all the movies
      responses:
        200:
          description: A list of movies
          schema:
            type: array
            items:
              required:
                - movieId
              properties:
                movieTitle:
                  type: string
                leadActor:
                  type: string
                movieId:
                  type: string
      parameters:
        - name: segmentSize
          in: query
          description: number of movies returned
          type: integer
        - name: segmentNumber
          in: query
          description: segment number
          type: integer
    post:
      summary: Creates a movie record
      description: adds a new movie to the list
      parameters:
        - name: movie
          in: body
          description: the movie to be created
          schema:
            required:
                - movieId
            properties:
              movieTitle:
                type: string
                leadActor:
                  type: string
                movieId:
                  type: string
          responses:
            200:
              description: Movies successfully added
            400:
              description: Movies couldn't be added
      /movies/{movieId}:
        get:
          summary: Gets a movie
          description: Returns a single movie for its movieId
          parameters:
            - name: movieId
              in: path
              required: true
              description: movieId of the movie
              type: string
          responses:
            200:
              description: A movie
      schema:
        required:
          - movieId
        properties:
            movieTitle:
              type: string
            leadActor:
              type: string
            movieId:
              type: string
    404:
      description: The movie does not exist in the database
```

How to make the API specification more compact

In the following section we will see how we can reduce the size of the file by creating references in our code by using the following:

Reusable definitions

Responses

Parameters

Data Model Description

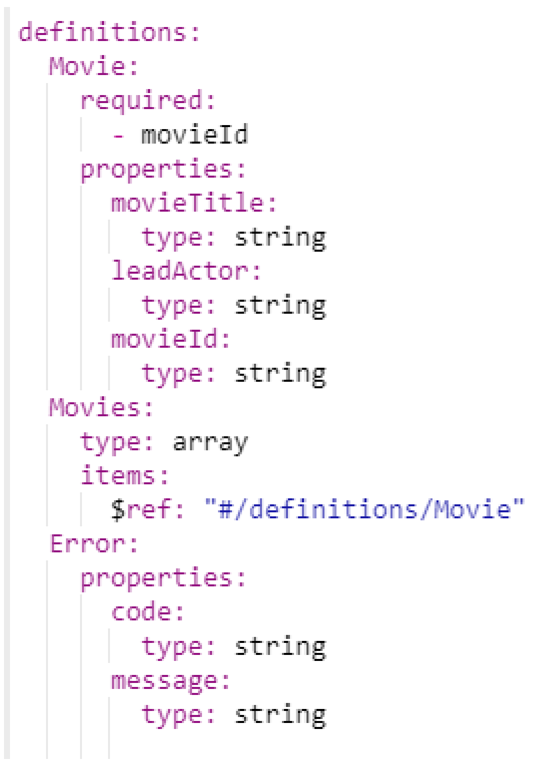
Let us take off from the previous code we wrote describing the whole API. You would notice that there is a lot of repetitive code that can be reused instead. We are using the details of the object Movie multiple times in the code. Hence, we can create a defined object for Movies and other parameters, lets see how.

By using the Definition section provided by OpenAPI Specification, we can define the code for an object once, and then define it at multiple places.

The data model consists of the type of object to be used in the API.

Defining Data Model

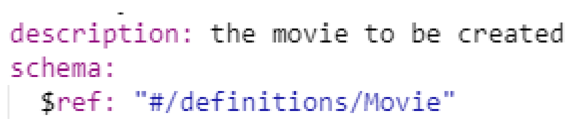
We need to have a Movie, Movies, and Error data model as shown below:



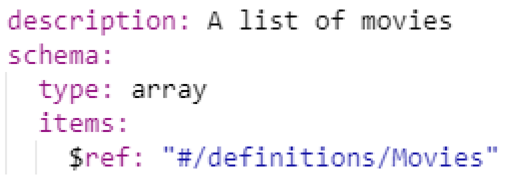
```yaml
definitions:
  Movie:
    required:
      - movieId
    properties:
      movieTitle:
        type: string
      leadActor:
        type: string
      movieId:
        type: string
  Movies:
    type: array
    items:
      $ref: "#/definitions/Movie"
  Error:
    properties:
      code:
        type: string
      message:
        type: string
```

Using the Data Model

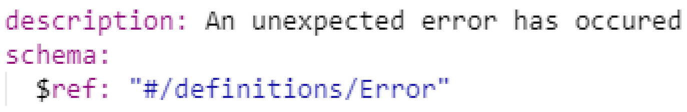
To use a data model, we will use the $ref tag as shown below:



```yaml
description: the movie to be created
schema:
  $ref: "#/definitions/Movie"
```



```yaml
description: A list of movies
schema:
  type: array
  items:
    $ref: "#/definitions/Movies"
```



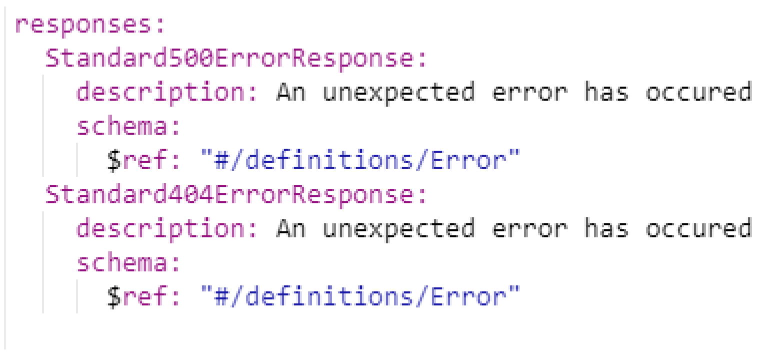
```yaml
description: An unexpected error has occured
schema:
  $ref: "#/definitions/Error"
```

Responses :

Responses can also be defined in the document for making the API even more compact.

Defining Responses

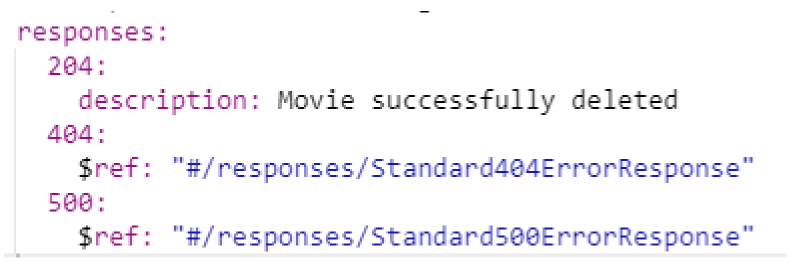
Here, the responses for most commonly used Error messages are defined.



```
responses:
  Standard500ErrorResponse:
    description: An unexpected error has occured
    schema:
      $ref: "#/definitions/Error"
  Standard404ErrorResponse:
    description: An unexpected error has occured
    schema:
      $ref: "#/definitions/Error"
```

Using Responses

Here, the responses are used in the code using $ref tag



```
responses:
  204:
    description: Movie successfully deleted
  404:
    $ref: "#/responses/Standard404ErrorResponse"
  500:
    $ref: "#/responses/Standard500ErrorResponse"
```

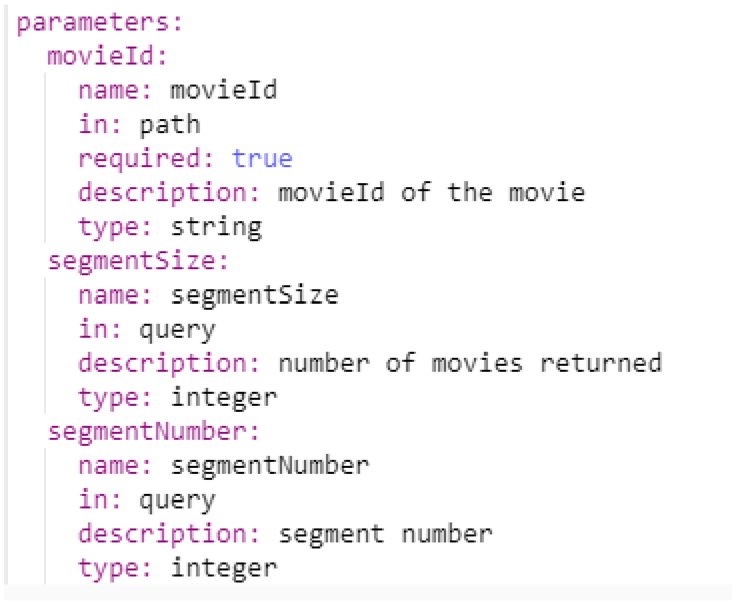
We can see that Error 404 and 500 are referred while responses code 204 is shown as it is. Depending on the frequency of a response occurring in the document, you can choose to use or not use a response.

PARAMETERS :

Parameters can also be defined to save space.

Defining Parameters

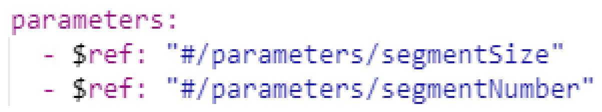
Here we are defining the parameters for movieId, segmentSize, and segmentNumber:



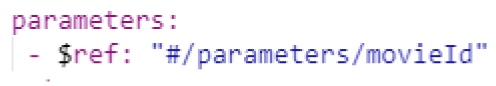
```
parameters:
  movieId:
    name: movieId
    in: path
    required: true
    description: movieId of the movie
    type: string
  segmentSize:
    name: segmentSize
    in: query
    description: number of movies returned
    type: integer
  segmentNumber:
    name: segmentNumber
    in: query
    description: segment number
    type: integer
```

Using Parameters

Now, we'll use the defined parameters as follows:



```
parameters:
  - $ref: "#/parameters/segmentSize"
  - $ref: "#/parameters/segmentNumber"
```



```
parameters:
 - $ref: "#/parameters/movieId"
```

SUMMARY :

Hence by making use of pre-defined objects, responses, and parameters, it becomes very easy to recreate the pieces of code by just referring to the definition, instead of recreating them from scratch.

RESOURCES :

```
swagger: '2.0'
info:
  version: '1.0'
  title: 'Movie Store API'
  description: 'This API will provide information regarding the titles
catalog in the system '
# Added by API Auto Mocking Plugin
host: virtserver.swaggerhub.com
basePath: /arjunsingh_qwerty/MovieStore/1.0
schemes:
 - https
paths:
  /movies:
    get:
      summary: Get list of movies
      description: Returns a list containing all the movies
      parameters:
        - $ref: "#/parameters/segmentSize"
        - $ref: "#/parameters/segmentNumber"
      responses:
        200:
          description: A list of movies
          schema:
            type: array
            items:
```

```
          $ref: "#/definitions/Movies"
      500:
        $ref: "#/responses/Standard500ErrorResponse"
  post:
    summary: Creates a movie record
    description: adds a new movie to the list
    parameters:
      - name: movie
        in: body
        description: the movie to be created
        schema:
          $ref: "#/definitions/Movie"
    responses:
      200:
        description: Movies successfully added
      400:
        description: Movies couldn't be added
      500:
        $ref: "#/responses/Standard500ErrorResponse"
/movies/{movieId}:
  parameters:
   - $ref: "#/parameters/movieId"
```

```
get:
  summary: Gets a movie
  description: Returns a single movie for its movieId
  responses:
    200:
      description: A movie
      schema:
        $ref: "#/definitions/Movie"
    404:
      description: The movie does not exist in the database
    500:
      $ref: "#/responses/Standard500ErrorResponse"
delete:
  summary: Deletes a movie
  description: Deletes a single movie for its movieId
  responses:
    204:
      description: Movie successfully deleted
    404:
      $ref: "#/responses/Standard404ErrorResponse"
    500:
      $ref: "#/responses/Standard500ErrorResponse"
```

```
definitions:
  Movie:
    required:
      - movieId
    properties:
      movieTitle:
        type: string
      leadActor:
        type: string
      movieId:
        type: string
  Movies:
    type: array
    items:
      $ref: "#/definitions/Movie"
  Error:
    properties:
      code:
        type: string
      message:
        type: string

responses:
  Standard500ErrorResponse:
    description: An unexpected error has occured
    schema:
      $ref: "#/definitions/Error"
  Standard404ErrorResponse:
    description: An unexpected error has occured
    schema:
      $ref: "#/definitions/Error"

parameters:
  movieId:
    name: movieId
    in: path
    required: true
    description: movieId of the movie
    type: string
  segmentSize:
    name: segmentSize
    in: query
    description: number of movies returned
    type: integer
  segmentNumber:
    name: segmentNumber
    in: query
    description: segment number
    type: integer
```