

# API Design

## Index

S.no	Topics	Page No
1	Introduction	1
2	API Design Principles	2
	• Nouns are good; verbs are bad	2
	• Simplify associations - sweep complexity under the ‘?’	3
	• Handling errors	4
	• Versioning	5
	• Pagination and partial response	5
	• Responses that don’t involve resources?	6
	• Supporting multiple formats	7
	• Attribute Names	7
	• Tips for search	7
	• Consolidate API requests in one subdomain	8
	• Do Web redirects	8
	• JSON API responses Design Principles	8
3	APIARY.IO	9
	• Introduction	9
	• Features	9
	• Structure of API documentation	9
	• Steps to create API Document using API Blueprint	10
	• Steps to create API Document using Swagger	14
4	USEFUL LINKS	19

## Introduction to API's

The API's job is to make the developer as successful as possible. The orientation for APIs is to think about design choices from the application developer's point of view.

Why? Look at the value chain below – the application developer is the lynchpin of the entire API strategy. The primary design principle when crafting your API should be to maximize developer productivity and success. This is what we call pragmatic REST.

Characteristics of a well-designed API:

- Easy to read and work with
- Hard to misuse
- Complete and Concise

To start with API designing, use an API design tool.

Tools available:

<http://apiary.io>

<https://swagger.io/tools/swaggerhub/faster-api-design/>

## API Design Principles

### Nouns are good; verbs are bad

The number one principle in pragmatic RESTful design is: keep simple things simple.

#### 1.1) Keep your base URL simple and intuitive.

- There should be only 2 base URLs per resource.
- Ex: The first URL is for a collection; the second is for a specific element in the collection.

/employees

/employees/1234

#### 1.2) Keep verbs out of your base URLs

#### 1.3) Use HTTP verbs to operate on the collections and elements.

Our HTTP verbs are *POST*, *GET*, *PUT*, and *DELETE*. (We think of them as mapping to the acronym, CRUD (Create-Read-Update-Delete).)

With our two resources (/employees and /employees/1234) and the four HTTP verbs, we have a rich set of capability that's intuitive to the developer. Here is a chart that shows what we mean for our employees.

Resource	POST create	GET read	PUT Update	DELETE delete
/employees	Create a new employee	List employees	Bulk update of Employees	Delete all Employees
/employees/1234	Error	Show Employee based on ID	Update employee	Delete Employee

Table: Describing resource functionality with HTTP methods

## Simplify associations - sweep complexity under the ‘?’

In this section, we explore API design considerations when handling associations between resources and parameters like states and attributes.

### 2.1) Associations

Resources almost always have relationships to other resources. What's a simple way to express these relationships in a Web API?

Remember, we had two base URLs: /employees and employees/1234.

We're using HTTP verbs to operate on the resources and collections. Our employees belong to department. To get all the employees belonging to a specific department, or to create a new employee for that department, do a GET or a POST:

*GET /department/5678/employees*

*POST /department /5678/employees*

Now, the relationships can be complex. Departments have relationships with domain, who have relationships with location, who have relationships with other resource, and so on.

You shouldn't need too many cases where a URL is deeper than what we have above

*/resource/identifier/resource.*

### 2.2) Sweep complexity behind the ‘?’

If association's go deeper than “/resource/identifier/resource”, then sweep the parameters after ‘?’.

*GET departments/5678/employees?grade=A&domain=domain1&location=hyd*

In summary, keep your API intuitive by simplifying the associations between resources, and sweeping parameters and other complexities under the rug of the HTTP question mark.

## Handling errors

From the perspective of the developer consuming your Web API, everything at the other side of that interface is a black box. Errors therefore become a key tool providing context and visibility into how to use an API.

### 3.1) How to think about errors in a pragmatic way with REST?

Error response may include following details to be more descriptive:

- Error code
- Error description
- Why error occurred
- How to resolve

### 3.2) Use HTTP status codes as error code

### 3.3) How many status codes should you use for your API?

When you boil it down, there are really only 3 outcomes in the interaction between an app and an API:

- Everything worked - success
- The application did something wrong – client error
- The API did something wrong – server error

Start by using the following 3 codes. If you need more, add them. But you shouldn't need to go beyond

- 200 - OK
- 400 - Bad Request
- 500 - Internal Server Error

If you're not comfortable reducing all your error conditions to these 3, try picking among these additional 5:

- 201 - Created
- 304 - Not Modified
- 404 – Not Found
- 401 - Unauthorized
- 403 - Forbidden

It is important that the code that is returned can be consumed and acted upon by the application's business logic - for example, in an if-then-else, or a case statement.

### 3.4) Make messages returned in the payload as verbose as possible.

Example:     {"developerMessage" : "Verbose, plain language description of the problem for the app developer with hints about how to fix it.", "userMessage": "Pass this message on to the app user if needed.", "errorCode" : 12345, "more info":  
"http://dev.teachdogrest.com/errors/12345"}

## Versioning

Versioning is one of the most important considerations when designing your Web API. Never release an API without a version and make the version mandatory. Let's see how three top API providers handle versioning.

- Twilio                     /2010-04-01/Accounts/ (Time stamp in the URL)
- salesforce.com         /services/data/v20.0/subjects/Account (use of the v. notation)
- Facebook               ?v=1.0 (Version as parameter)

### 4.1) Version numbers in a pragmatic way with REST?

- Never release an API without a version. Make the version mandatory.
- Specify the version with a 'v' prefix. Move it all the way to the left in the URL so that it has the highest scope (e.g. /v1/employees).
- Use a simple ordinal number.
- Don't use the dot notation like v1.2 because it implies a granularity of versioning that doesn't work well with APIs--it's an interface not an implementation.
- Stick with v1, v2, and so on.

### 4.2) How many versions should you maintain? Maintain at least one version back

## Pagination and partial response

Partial response allows you to give developers just the information they need.

LinkedIn         /people:(id,first-name,last-name,industry)

This request on a person returns the ID, first name, last name, and the industry. LinkedIn does partial selection using this terse :(...) syntax which isn't self-evident. Plus it's difficult for a developer to reverse engineer the meaning using a search engine.

Facebook         /joe.smith/friends?fields=id,name,picture

Google           /?fields=title,media:group(media:thumbnail)

Google and Facebook have a similar approach, which works well

### 5.1) Add optional fields in a comma-delimited list

Example:         /employees?fields=salary,location,domain

## 5.2) make it easy for developers to paginate objects in a database

It's almost always a bad idea to return every resource in a database.

Facebook uses offset and limit. Twitter uses page and rpp (records per page). LinkedIn uses start and count

To get records 50 through 75 from each system, you would use:

- Facebook - offset 50 and limit 25
- Twitter - page 3 and rpp 25 (records per page)
- LinkedIn - start 50 and count 25.

## 5.3) Use limit and offset

Limit and offset is recommended. It is more common, well understood in leading databases, and easy for developers.

*/employees?limit=25&offset=50*

## 5.4) Metadata

Include metadata with each response that is paginated that indicated to the developer the total number of records available.

## 5.5) what about defaults?

Loose rule of thumb for default pagination is limit=10 with offset=0. (offset= &limit=10)

The pagination defaults are of course dependent on your data size. If your resources are large, you probably want to limit it to fewer than 10; if resources are small, it can make sense to choose a larger limit.

## Responses that don't involve resources?

Actions like the following are your clue that you might not be dealing with a "resource" response.

- Calculate
- Translate
- Convert

## 6.1) Use verbs not nouns

For example, an API to convert 100 euros to Chinese Yen:

*/convert?from=EUR&to=CNY&amount=100*

## 6.2) make it clear in your API documentation that these "non-resource" scenarios are different.

## Supporting multiple formats

Push things out in one format and accept as many formats as necessary. You can usually automate the mapping from format to format.

Here's what the syntax looks like for a few key APIs.

Google Data ?alt=json

Foursquare /venue.json

Digg\* Accept: application/json ?type=json

### 7.1) what about default formats?

JSON is winning out as the default format. JSON is the closest thing we have to universal language. Even if the back end is built in Ruby on Rails, PHP, Java, Python etc., most projects probably touch JavaScript for the front-end. It also has the advantage of being terse - less verbose than XML.

## Attribute names?

### 8.1) Naming convention of Attributes

- Use JSON as default
- Follow JavaScript conventions for naming attributes - Use medial capitalization (aka CamelCase) - Use uppercase or lowercase depending on type of object

This results in code that looks like the following,

```
Response: {"createdAt": 1320296464 }
```

```
var myObject = JSON.parse(response);
```

```
timing = myObject.createdAt;
```

## Tips for search

A more complex search across multiple resources requires a different design.

If you want to do a global search across resources, follow the Google model:

Global search */search?q=fluffy+fur*

Here, search is the verb and ?q represents the query

## Consolidate API requests in one subdomain

It's cleaner, easier and more intuitive for developers who you want to build cool apps using your API

Facebook, Foursquare, and Twitter also all have dedicated developer portals.

- *developers.facebook.com*
- *developers.foursquare.com*
- *dev.twitter.com*

Example:

*/drive.google.com*

*/maps.google.com*

*/mail.google.com*

## Do Web redirects

Then optionally, if you can sense from requests coming in from the browser where the developer really needs to go, you can redirect.

Say a developer type's *api.teachdogrest.com* in the browser but there's no other information for the GET request, you can probably safely redirect to your developer portal and help get the developer where they really need to be.

Api -> developers (if from browser)

Dev -> developers

developer -> developers

## JSON API responses Design Principles

### 12.1) Include/Exclude Certain Fields

An API with a single approach to responses is likely to give developers much more or less data than they need. There are downsides to each situation, for both the developer and the API provider. If an API responds with more data than a developer needs, both sides have paid a tiny latency penalty in the time and bandwidth required to transfer data nobody cares about at the moment. Similarly, if a important



field is excluded from the response, a developer may have to make a second request to fill in the missing data.

A bit too much or too little data may seem trivial, but in aggregate they can have a huge impact. An extra API call, at scale, could double the server costs of both sides. Here are how other APIs have approached the issue.

## 12.2) TimeStamp in UTC format

Example: {..... currentTime: "2018-12-06T12:30:56.632Z" .....

## APIARY.IO

APIARY is a tool to design API's and to fast-track API design. Here is the link to [Login/signup](#).

### Features:

- **Dashboard**  
Dedicated, web-based team and API Blueprint management dashboard
- **Access Control**  
Role-based access control over API documents
- **Roles**  
Admin, Editor, and Viewer roles
- **Provisioning**  
Add and remove team members from API design projects
- **Template**  
Shared API Blueprint templates to bootstrap new projects
- **Customization**  
Default settings for API Blueprint visibility and new team member provisioning
- **APIARY Editor**  
It provides APIARY editor, a tool to develop API Blueprint. The Apiary Editor is the foundation of your API design. Apiary Editor supports API Blueprint and Swagger API Description languages.

## API BLUEPRINT

DNA for your API—powerful, [open sourced](#) and developer-friendly.

### Simple Structure of API blueprint:

1. Name of the API and metadata(Name of the project)
2. API's for Resource1 (This section includes all the API call's on Resource1 Ex:  
/employees/{employeeId})
  - a. Get List of Employee [GET]
    - i. Define request and response

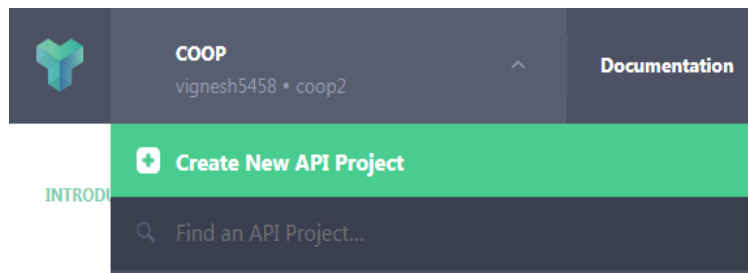
- b. Create Employee [POST]
    - i. Define request and response
  - c. Delete employee [DELETE]
    - i. Define request and response
  - d. Update employee [PUT]
    - i. Define Request and Response
3. Similarly for resource2,3..n
4. Define Data Structures required for the API.

We're going to build an API blueprint step by step for a service called Employees – a simple API allowing user to view employee's detail.

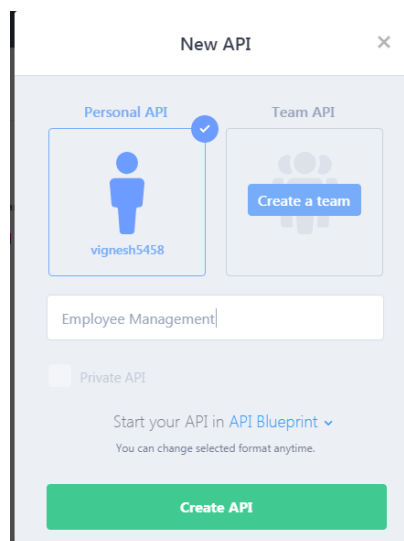
## Steps to create API documentation using API Blueprint

### STEP1: Create an API Project

- On left-top u can find the button to create new API project.



- Provide API name



- Select “API Blueprint” to create documentation using API blueprint.
- New API will be created and will be redirected to API editor where we can create API blueprint

Step2: Edit API using editor.

- API name and Meta Data Section:

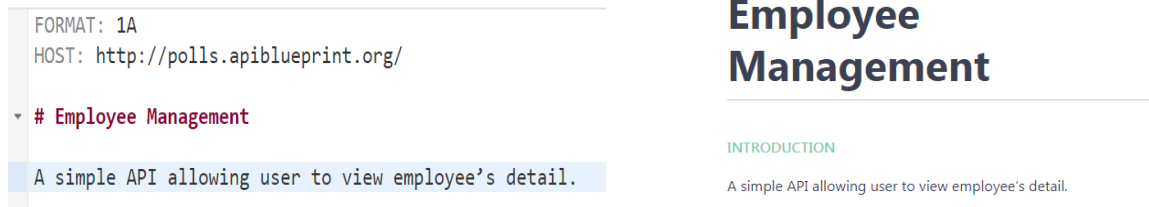


Fig: Meta data section(Leftside Blue print and rightside resultant Document )

- Define Data Structures.
  - Each API call includes request and response body and the attributes used in the body are defined in this section. Defining the whole resource structure in each call is a redundant task and to eliminate that we can define the resource as a data structure and we can use that where ever needed.
  - [Data structures](#) are defined using MSON. At the bottom of the blueprint we define data structures.
  - Usually this DS are defined at bottom of the blue print.
  - Let's see how to define a date structure for Employee object and how to use it in blueprint.

```

# Data Structures

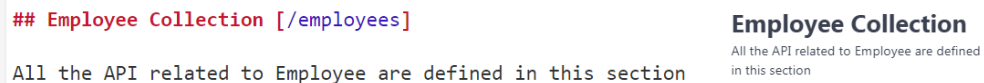
##Employee (object)
+ employeeId 1 ( number, required ) -UniqueID that defines Employee
+ name 'Vijay' (string, required) -name of the employee
+ age 33 (string, optional) - age of the employee
+ salary 40000 (number, required) - salary of the employee
+ address
  + street: '235 Ninth Street' (string, optional) - First line of employee's address
  + city: 'San Francisco' (string, required) - State of employees address
  + state: 'California' (string, required) - country to which employee belongs
  
```

Fig: Employee data structure.

Steps to define Resource using data structures:

- Resource name followed by the resource type (inheritance). [Ex: ##Employee (object)]
- Syntax to define an attribute.  
 <attributeName> <sampleValue> (<Type> , required/optional) – <description>
- If any nested objects exists it can be defined as 'address' in the above employee example.

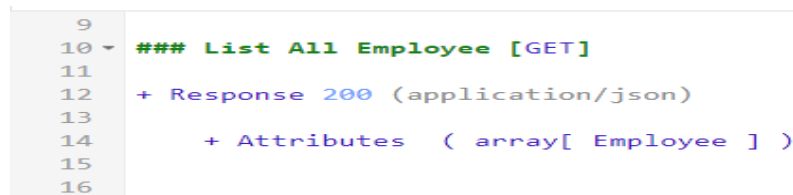
- Similarly we can create Resources like department and jobRole etc.
- To learn more [click here](#).
- Creating API calls for a resource Employee:
  - This section followed next to “API name and Meta Data Section”
  - Define URL for the resource Employee ex: [/employees]



The screenshot shows an API Blueprint section for the 'Employee Collection' resource at the URL '/employees'. The title is '## Employee Collection [/employees]' in red. Below it, a description states 'All the API related to Employee are defined in this section'. To the right, a header 'Employee Collection' is shown with a sub-header 'All the API related to Employee are defined in this section'.

Fig: Defining URL and description for the resource Employees

- Under Employee Collection resource Section lets create two API calls:
  - GET** /employees/                      **POST** /employees/
- To define GET /employees/
  - GET call with response as Array of Employee can be defined as below.



The screenshot shows the definition of the GET endpoint for the Employee Collection resource. It starts with '### List All Employee [GET]' on line 10. Line 12 adds a response: '+ Response 200 (application/json)'. Line 14 adds attributes: '+ Attributes ( array[ Employee ] )'.

- To define GET /employees/
  - POST call with request as Array of Employee and with response as Array of Employee who have been persisted, can be defined as below.



The screenshot shows the definition of the POST endpoint for the Employee Collection resource. It starts with '### Create a New Employee [POST]' on line 19. Line 21 provides a description: 'You may create your own employee using this action. It takes a JSON object containing a employee.' Line 24 adds a request: '+ Request (application/json)'. Line 26 adds attributes: '+ Attributes ( array[ Employee ] )'. Line 28 adds a response: '+ Response 201 (application/json)'. Line 30 adds attributes: '+ Attributes ( array[ Employee ] )'.

Fig: Using Data structure in blueprint

- The request can also have header section and parameters.
- Thus we have learnt how to create a simple API documentation using APIARY blueprint. Snapshots of resultant documentation are given below,

Employee Management  
vignesh5458 • employeemanagement
Documentation Inspector Editor Tests
Create a team

INTRODUCTION
REFERENCE
Employee Collection

## Employee Management

**INTRODUCTION**

A simple API allowing user to view employee's detail.

**REFERENCE**

### Employee Collection

All the API related to Employee are defined in this section

List All Employee
Create Employee's

You may create your own employee using this action. It takes a JSON object containing a employee.

Postman

Switch to Console

Employee Collection List All Employee
GET http://polls.apiblueprint.org/employees

Request
Production Raw Try

Response

ATTRIBUTES

object
employeeId 1 required number UniqueID that defines Employee
name "Vijay" required string name of the employee

Employee Management  
vignesh5458 • employeemanagement
Documentation Inspector Editor Tests
Create a team

INTRODUCTION
REFERENCE
Employee Collection

## Employee Management

**INTRODUCTION**

A simple API allowing user to view employee's detail.

**REFERENCE**

### Employee Collection

All the API related to Employee are defined in this section

List All Employee
Create Employee's

You may create your own employee using this action. It takes a JSON object containing a employee.

Switch to Console

age 33 string age of the employee
salary 40000 required number salary of the employee
address object

200

HEADERS
Content-Type: application/json

BODY
SHOW JSON SCHEMA

```

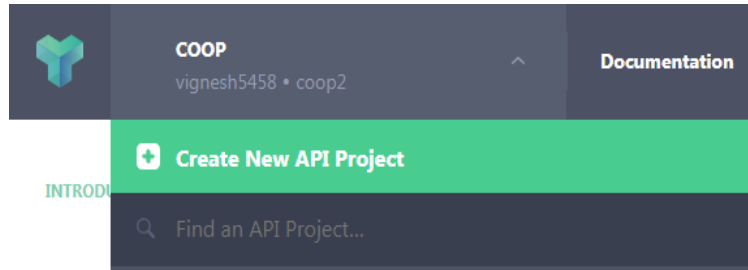
01 [
02 {
03   "employeeId": 1,
04   "name": "Vijay",
05   "salary": 40000,
06   "address": {
07     "street": "123 Ninth Street",
08     "city": "San Francisco",
09     "state": "California"
10   }
11 }
12 ]

```

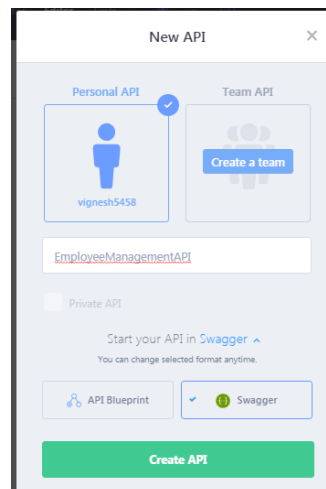
## Steps to create API documentation using Swagger

### STEP1: Create an API Project

- On left-top u can find the button to create new API project.



- Provide API name
- Select “API Blueprint” to create documentation using API blueprint.



- New API will be created and will be redirected to API editor with an existing example

### Step2: Edit API using editor.

- API name and Meta Data Section
  - Every Swagger document starts with Swagger version declaration `swagger: "2.0"`. Then you can specify `info` object for additional metadata. `title` and `version` are required parameters, others like `description` are optional. See [Known limitations](#) on what parameters are not supported.
- URL Definition:
  - `schemes` is an array protocols supported by API. You can specify them like so: `[http, https]`. Apiary will use `https` if defined
  - `host` is domain for API
  - `basePath` defines URL prefix for all defined endpoints.

For example, defining `/account` endpoint actually means `scheme://host/basePath/account`

```
1 swagger: '2.0'
2 info:
3   version: '1.0'
4   title: "EmployeeManagementAPI"
5   description: A simple API allowing the users to view employee details.
6   license:
7     name: MIT
8     url: https://github.com/apiaryio/polls-api/blob/master/LICENSE
9 host: polls.apibluprint.org
10 basePath: /
11 schemes:
12   - http
13 consumes:
14   - application/json
15 produces:
16   - application/json
```

## Employee Management

### INTRODUCTION

A simple API allowing user to view employee's detail.

Fig: Meta data section (Left side Blue print and right side resultant Document)

- Define Definitions(Data Structures).
  - Definitions have same goal as MSON. Make it easy to describe data structures and use them in API Description. Unlike MSON, definitions in Swagger are using JSON Schema and JSON Schema referencing for use inside API Description
  - Each API call includes request and response body and the attributes used in the body are defined in this section. Defining the whole resource structure in each call is a redundant task and to eliminate that we can define the resource as a data structure and we can use that where ever needed.
  - Usually this DS are defined at bottom of the swagger file.
  - Let's see how to define an Employee object and how to use it in swagger.

```
68 definitions:
69   Employee:
70     title: Employee
71     type: object
72     properties:
73       employeeId:
74         type: string
75       name:
76         type: string
77       age:
78         type: integer
79       salary:
80         type: integer
81       address:
82         $ref: '#/definitions/Address'
83     required:
84       - employeeId
85       - name
86       - address
87   Address:
88     title: Address
89     type: object
90     properties:
91       street:
92         type: string
93       city:
94         type: string
95       state:
96         type: string
97       pincode:
98         type: integer
99     required:
100       - pincode
101       - state
```

Fig: Employee data structure.

- Creating API calls for a resource Employee:
  - This section followed next to “API name and Meta Data Section”
  - Define URL for the resource Employee ex: [/employees]

```

17 paths:
18   /employees:
19     x-summary: Employee Collection
20     get:
21       summary: List All Employees
22       description:
23         Lists out all the employees
24       responses: {}
42     post:
43       summary: Create a New Question
44       description: >-
45         You may create your own question using this action. It takes a JSON
46         object containing a question and a collection of answers in the
47         form of choices.
48       parameters: {}
49       responses: {}
70
71

```

Fig: Defining URL and description for the resource Employees

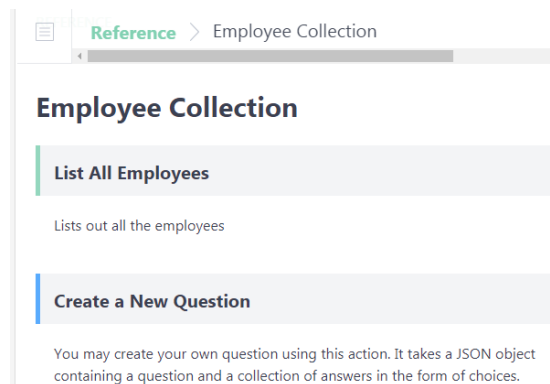


Fig: Employee Collection design

- Under Employee Collection resource Section lets create two API calls:
  - GET** /employees/                      **POST** /employees/
- To define GET /employees/
  - GET call with response as Array of Employee can be defined as below.

```

20 get:
21   summary: List All Employees
22   description:
23     Lists out all the employees
24   responses:
25     200:
26       description: Successful Response
27       schema:
28         type: array
29         items:
30           $ref: '#/definitions/Employee'
31       examples:
32         application/json:
33           - employeeId: 12652
34             name: 'Vijay'
35             age: 41
36             salary: 900000
37             address:
38               - street: park Avenue
39                 city: Hyderabad
40                 state: Andhra Pradesh
41                 pincode: 510001
42

```




Fig: Using Data structure in swagger

- To define GET /employees/
  - POST call with request as Employee and with response of the Employee who have been persisted, can be defined as below.

```
42 ▾ post:
43   summary: Create a New Question
44   description: >-
45     You may create your own question using this action. It takes a JSON
46     object containing a question and a collection of answers in the
47     form of choices.
48   parameters:
49     - name: body
50       in: body
51       required: true
52       schema:
53         $ref: '#/definitions/Employee'
54   responses:
55     201:
56       description: ''
57       schema:
58         $ref: '#/definitions/Employee'
59       examples:
60         application/json:
61           employeeId: 12652
62           name: 'Vijay'
63           age: 41
64           salary: 900000
65           address:
66             street: park Avानue
67             city: Hyderabad
68             state: Andhra Pradesh
69             pincode: 510001
70
```


Fig: Using Data structure in blueprint

- The request can also have header section and parameters.
- Thus we have learnt how to create a simple API documentation using Swagger. Snapshots of resultant documentation are given below,



EmployeeManagementAPI  
vignesh5458 • employeeemanagementapi

DocumentationInspectorEditorTests

Create a team

REFERENCE

INTRODUCTION

REFERENCE

Employee Collection

## Employee Collection

List All Employees

Create a New Question


You may create your own question using this action. It takes a JSON object containing a question and a collection of answers in the form of choices.

Switch to Console

BODY


SHOW JSON SCHEMA

```
01 [
02   {
03     "employeeId": 12652,
04     "name": "Vijay",
05     "age": 41,
06     "salary": 900000,
07     "address": [
08       {
09         "street": "park Avenue",
10         "city": "Hyderabad",
11         "state": "Andhra Pradesh",
12         "pincode": 510001
13       }
14     ]
15   }
16 ]
```



EmployeeManagementAPI  
vignesh5458 • employeeemanagementapi

DocumentationInspectorEditorTests

Create a team

REFERENCE

INTRODUCTION

REFERENCE

Employee Collection

## Employee Collection

List All Employees

Create a New Question

You may create your own question using this action. It takes a JSON object containing a question and a collection of answers in the form of choices.

Switch to Console

Request

ATTRIBUTES

employeeId	string
required	
name	string
required	
age	number
salary	number
address	object
required	

## USEFULL LINKS:

Example JSON responses available:

<https://developer.atlassian.com/server/crowd/json-requests-and-responses/>

To know the all the HTTP status codes available

<https://www.restapitutorial.com/httpstatuscodes.html>

Examples on how to create a data structure using MSON

<https://github.com/apiaryio/mson#example-1>

To know more about APIARY

<https://apiblueprint.org/documentation/specification.html>